Linux Shell

1 shell基础

shell 是系统的用户界面，它一共了用户和Linux内核之间进行交互操作的一种接口。用户在命令行中输入的每个命令都由shell先解释，然后传给Linux内核去执行。

- 一个命令行中可以输入多个命令，用分号将各个命令隔开
- 命令太长可以用反斜杠 \ 来续行
- 命令行实际上是一个可以编辑的文本缓冲区

左/右箭头键	向左/向右移动一个字符
Ctrl+a	移动到当前行的行首
Ctrl+e	移动到当前行的行尾
Ctrl+f	向前移动一个字符
Ctrl+b	向后移动一个字符
Ctrl+k	从光标处删除到本行的行尾
Ctrl+u	从光标处删除到本行的行首
Ctrl+l	清屏
Alt+f	向前移动一个单词
Alt+b	向后移动一个单词

stty -a 可以看到更多的快捷键。

- 通配符 *, ?, []

*	匹配 任意长度 的字符串（包括零个字符）
?	匹配任何 单个字符
[ ]	创建一个字符表列，方括号中的字符用来匹配或不匹配单个字符。如： [xyz] 匹配 x、y 或 z，但不能匹配 xx，xy 或者其它任意组合。 无论列表中有多少个字符，它只匹配一个字符。 [abcde] 可以简写为 [a-e] 。 另外，用感叹号作为列表的第一个字符可以起到反意作用，如： [!xyz] 表示匹配 x、y、z 以外的任意一个字符。

  - 文件名最前面的圆点“.”和路径名中的斜杠"/"必须显式匹配
  - 连字符"-"仅在方括号内有效，表示字符范围
  - "*"和"?"在方括号外面是通配符，在方括号内是普通字符
- 别名：

```
alias lf='ls -F'  # 创建别名(在当前shell中有效)，等号两边不能有空格
```

```
unalias lf  # 取消别名
```

```
alias # 查看已经创建的别名
```

1.1 管道 " | "

管道的基本含义是：将前一个命令的输出作为后一个命令的输入

```
# 输出系统中用户名的排序列表
cat /etc/passwd |\
awk -F: '{print $1}' |\
sort
```

1.2 重定向

Linux中的数据了有三种：标准输入(STDIN)，标准输出(STDOUT)和标准错误(STDERR)。

标准输入通常来自键盘；标准输出是命令的结果，通常定向到显示器；标准错误是错误信息，通常也定向到显示器

- 输入重定向 " < " : 使用文件中的内容作为输入
- 输出重定向 " > " : 将输出保存到文件中
- 如果希望保留原文件的内容，使用 " >> "
- 文件描述符：
  - 0 -- 标准输入(STDIN)
  - 1 -- 标准输出(STDOUT)
  - 2 -- 标准错误(STDERR)

```
cat x y 1>out1 2>out2  # 标准输出重定向，标准错误重定向
```

```
cat x,y 1>out1 2>&1  # 标准错误重定向到标准输出所在
```

- 引用
  - 使用转义字符 \ 或使用单引号 ' ' 或使用双引号 " "
  - 单引号对 中的字符都作为普通字符，但不允许出现另外的单引号
  - 双引号对 中的部分字符仍然保留特殊含义，如 $, \ , ' , " , 换行符
  - Shell 中的特殊字符

字符	含义	字符	含义
'	强引用	*、?、!	通配符
"	弱引用	<、>、>>	重定向
\	转义字符	-	选项标志
$	变量引用	#	注释符
;	命令分离符	空格、换行符	命令分隔符
`	命令替换：反引号中的字符串被 shell 解释为命令，在执行时，shell 首先执行该命令，并以它的标准输出取代整个反引号 (包括两个反引号) 部分		

31

1.3 shell变量

大致分为三类：内部变量、用户变量、环境变量

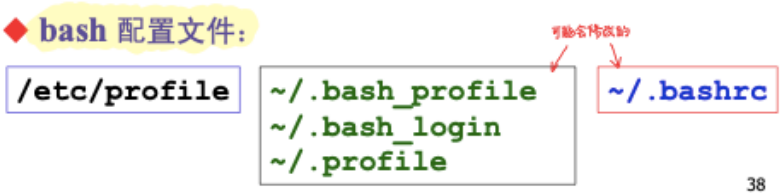
- 内部变量：系统提供，用户不能修改
- 用户变量：用户建立或修改，在shell脚本编写中经常用到
- 环境变量：决定用户工作环境的变量，可以直接在shell中使用，其中某些变量用户可以修改

变量名	含义
HOME	用户主目录
LOGNAME	登录名
USER	用户名，与登录名相同
PWD	当前目录/工作目录名
MAIL	用户的邮箱路径名
HOSTNAME	计算机的主机名
INPUTRC	默认的键盘映像
SHELL	用户所使用的 shell 的路径名
LANG	默认语言
HISTSIZE	history 所能记住的命令的最多个数
PATH	shell 查找用户输入命令的路径 (目录列表)
PS1、PS2	shell 一级、二级命令提示符

```
env                 # 查询当前shell中的环境变量
echo ${变量名}      # 查询某个变量的值
export 变量名=...   # 使变量的值对当前shell及其所有子进程都可见
```

\!	显示命令的历史编号	\h	显示机器的主机名
\#	显示命令的命令编号	\s	显示当前使用的 shell
\\	显示一个反斜杠	\u	显示用户名
\n	显示一个换行符	\W	显示当前目录名
\d	显示当前的日期	\w	显示当前目录完整路径名
\t	显示当前的时间		
\$	普通用户显示“$”，超级用户显示“#”	\nnn	显示与八进制 nnn 相对应的字符

1.4 bash配置文件



- `/etc/profile`：Linux系统中的全局bash启动脚本，任何用户登录系统时都会被执行。修改需要root权限
- 读取`/etc/profile` 文件后，bash将在用户主目录中按照顺序查找以下文件，并执行第一个找到的文件：
  - `~/.bash_profile`　`~/.bash_login`　`~/.profile`

  在这些文件中，用户可以自定义环境变量，并且能够覆盖`/etc/profile` 中的配置。
- bash启动后，将读取配置文件`~/.bashrc` 并执行文件中的所有内容
- 另外，还可以从另一个shell或者bash自身启动一个新的bash，这种过程称为非登陆交互式，此时所读入的唯一bash配置文件是 `~/.bashrc`

1.5 shell的输入和输出

- echo命令

```
echo [-e] [-n] string
```

  - -e：让echo解释string中的转义字符
  - -n：禁止echo输出后换行
  - string：字符串，可以含shell变量、转义字符等，一般用双引号括起来
- read命令

```
read variable1 variable2
read -p "提示信息" var1 var2
```

  read用空格符作为分隔符，把输入行分成多个区域，分别赋值给各个变量
- tee 命令

```
tee [-a] filename
```

  - -a：追加到文件末尾
- cat 命令

```
cat [-n][-b][-t][-e] file1 file2
```

  - -n：显示行👌
  - -b：显示行号（不含空行）

- -t : 显示制表符
- -e : 显示行结束符

创建文件：

```
cat file1 file2 > newfile # 合并文件
cat > nwefile             # 输入文本，按 ctrl+d 结束输入
```

2 进程

- 正在运行的程序叫做进程 (process)
  - 程序只有被系统载入内存并运行之后才能称为进程
- Linux每个进程都有属于自己的唯一进程号 process ID (pid)
- 查看当前运行的程序及其进程号 : ps

2.1 多进程

◆ 分时技术

所有的任务请求被排除一个队列，系统按顺序每次从这个队列中抽取一个任务来执行，这个任务执行很短的时间（几毫秒）后，系统就将它排到任务队列的末尾，然后读入队列中的下一个任务，以同样的方式执行。这样经过一段时间后，任务队列中的所有任务都被执行一次，然后又开始下一轮循环。

◆ 任务/作业

就是一个被用户指定运行的程序。如用户发出一个打印命令，就产生一个打印任务/作业，若打印成功，表示任务完成，没有成功表示任务没完成。

◆ Linux 是多用户系统，它必须协调各个用户。

Linux 给每个进程都打上了运行者的标志，用户可以控制自己的进程：给自己的进程分配不同的优先级，也可以随时终止自己的进程。

2.2 前台与后台

◆ 前台进程

指一个程序控制着标准输入/输出，在程序运行时，shell 被暂时挂起，直到该程序运行结束后，才退回到 shell。在这个过程中，用户不能再执行其它程序。

◆ 后台进程

用户不必等待程序运行结束就可以执行其它程序。

◆ 在一个终端里只能同时存在一个前台任务，但可以有多个后台任务。

❑ 运行后台进程

- 在命令最后加上“&”

```
例：sleep 60 &
```

- 如果程序已经在前台运行，需要将其改为后台运行，这时可以先安组合键 **Ctrl+z**，将任务挂起，然后利用 **bg** 命令将该程序转为后台运行
- 若要将一个后台进程转到前台运行，可以使用 **fg** 命令
- 相关命令：**jobs, bg, fg**

1. jobs 命令： 查看后台运行或被挂起的进程

```
jobs      # 显示后台进程，第一列是作业号，“+”表示当前作业，“-”表示当前作业之后的作业
jobs -l   # 显示进程号
```

2. bg 命令： 将被挂起的进程转化到后台运行

```
bg jobnumber  # jobnumber是通过jobs查出来的作业号
```

3. fg 命令： 将后台进程转化到前台运行，用法与bg类似
4. ps 命令： 查看正在运行的程序

```
ps [选项]
```

```
-A, -e   显示所有进程
  -u     查看指定用户的进程（用户名或用户ID）
  -l     长格式显示，可查看各个进程的优先权值
  -f     完全显示（显示完整的命令）
  -C     列出指定命令名称的进程
```

```
   u  增加用户名，起始时间，CPU和内存使用等信息
   a  显示终端机下用户执行的进程，包含其它用户
   f  显示进程树，等价于 --forest
   r  显示正在运行的进程
```

```
  -o     按指定的格式输出
--sort   按指定内容进行排序
```

- 常见列标志的含义

例：`ps -u jypan u`

PID	进程 ID	CMD	命令名（COMMAND）
UID	用户 ID	START	进程启动时间
USER	用户名	%CPU	进程所用CPU时间百分比
TIME	执行时间	%MEM	进程所有MEM百分比
STAT	进程状态	NI	优先权值 / nice 值

TTY	启动进程的终端	RSS	进程所用内存块数
PGID	进程组 ID	VSZ	所用虚拟内存块数

- 指定输出格式

例：`ps -u jypan -o "%U %p %c %x %t"`

- 输出格式中的常用字段

```
%U   用户名
%u   用户名
%G   用户组
%g   用户组
%C   CPU
%P   父进程
```

```
%c   命令名
%a   命令名（含选项与参数）
%p   进程号
%x   运行时间
%t   Elapsed time
%n   nice 值（代表优先权）
```

```
%r  PGID -- ID of the process group ( leader )
```

例：`ps -u jypan -o %c%p%r%n`

- 另一种使用方式

```
ps -u jypan -o user,pid,pcpu,time,etime
```

- 字段对应表

%U	用户名	user	%c	命令名	comm
%u	用户名	ruser	%a	命令名	args
%G	用户组	group	%p	进程号	pid
%g	用户组	rgroup	%x	运行时间	time
%C	CPU	pcpu	%t	Elapsed time	etime
%P	父进程	ppid	%n	nice 值	nice
%r	进程组	pgid		用户ID	uid

- 进程排序

```
1 ps au --sort=uid,-pid
```

5. nohup 命令： 使得用户退出系统后程序能继续运行

```
nohup 命令 [选项] [参数]
```

2.3 进程的优先权

- 在任务队列中的进程并不享有同等的优先权，每个进程都有一个指定的 nice值 ，从 负20到19 。
- nice值 越低，进程的优先级越高。进程的默认nice值为0。
- 在进程创建时指定优先级

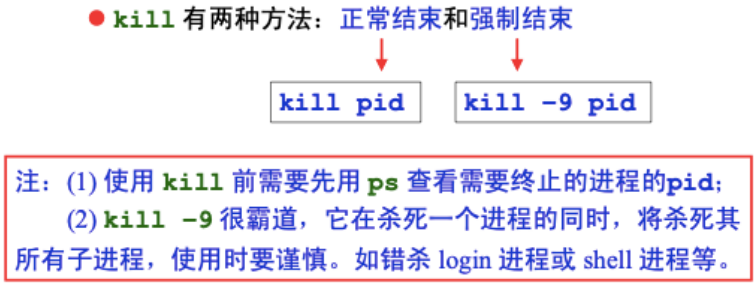
```
nice -n 命令 & # n为优先级增量。普通用户只能降低优先级，只有root用户可以增加优先级。
```

- 进程运行后调整 nice值

```
renice n [-p pid] [-u user] [-g pgid]
# n为优先级增量。普通用户只能指定正数的n，只有root用户可以指定负数的n。
```

2.4 终止进程

- 终止前台进程：Ctrl+c
- 终止后台进程：kill



3 正则表达式

3.1 基本元字符集及其含义

^	只匹配行首（可以看成是行首的标志）
$	只匹配行尾（可以看成是行尾的标志）
*	一个单字符后紧跟 *，匹配 0 个或多个此单字符
[ ]	匹配 [] 内的任意一个字符 （[^] 反向匹配）
\	用来屏蔽一个元字符的特殊含义
.	匹配任意单个字符
c\{n\}	匹配 字符 c 连续出现 n 次的情形
c\{n,\}	匹配 字符 c 至少连续出现 n 次的情形
c\{n,m\}	匹配 字符 c 连续出现次数在 n 与 m 之间的情形

注：字符 c 可以通过 [ ]，\ 或 . 来指定，但只能是单个字符。
如：[a-z]\{5\}，\$\{2,\}，.\{2,5\}

3

- 使用“[]”匹配一个字符范围或集合
  - 匹配“[]”内的字符，可以是单个字符，或字符序列，可以使用 - 表示一个字符序列范围，如 [A-Za-z0-9]
  - 当 [后面紧跟 ^ 符号时，表示不匹配方括号里内容

```
[Cc]omputer 匹配 Computer 和 computer
[^a-zA-Z]   匹配任一个非字母型字符
```

3.2 常用正则表达式举例

[Ss]igna[lL]	匹配 signal、signaL、Signal、SignaL
[Ss]igna[lL]\.	同上，但后面加一句点
^USER$	只包含 USER 的行
\.	带句点的行
^d..x..x..x	用户、同组用户及其他用户都有可执行权限的目录
^[^s]	不以 s 开始的行
[yYnN]	大写或小写的 y 或 n
.*	匹配任意多个字符
^.*$	匹配任意行
^......$	只包含 6 个字符的行

[a-zA-Z]	任意单个字母
[^a-zA-Z0-9]	非字母或数字
[^0-9\$]	非数字或美元符号
[123]	1 到 3 中一个数字
\^q	包含 ^q 的行
^.$	仅有一个字符的行
^\.[0-9][0-9]	以一个句点和两个数字开始的行

[0-9]\{2\}-[0-9]\{2\}-[0-9]\{4\} 日期格式 dd-mm-yyyy
[0-9]\{3\}\.[0-9]\{3\}\.[0-9]\{3\}\.[0-9]\{3\} 类 IP 地址格式 nnn.nnn.nnn.nnn

4

4 文本过滤操作

>> grep 查找某个字符模式并输出

- grep命令不对输入文件进行任何修改或影响

```
grep [选项] pattern file1 file2 ...
```

  - pattern： 正则表达式（用单引号括起来）或者字符串（双引号）或一个单词
  - file1 file2 ...： 输入文件列表；grep的输入也可以来自标准输入或者管道
  - 可以把匹配模式写入到文件中，每行一个，然后使用 -f 选项，将匹配模式传给grep命令

```
grep -f patternfile file1 file2 ...
```

- grep常用选项

-c	只输出匹配的行的总数
-i	不区分大小写
-h	查询多个文件时，不显示文件名
-l	查询多个文件时，只输出包含匹配模式的文件的文件名
-n	显示匹配的行及行号
-v	反向查找，即只显示不包含匹配模式的行
-s	不显示错误信息

```
grep -i 'an*' datafile
```

- grep命令举例

◆ 查询多个文件,可以使用通配符“ * ”

```
grep "math2"  *.txt
```

```
grep "12"  *
```

◆ 反向匹配

```
ps aux | grep "ssh" | grep -v "grep"
```

◆ 匹配空行

```
grep -n '^$' datafile
```

```
grep -v '^$' datafile > datafile2
```

◆ 精确匹配单词: \< 和 \>

● 找出所有包含 以 north 开头 的单词的行

```
grep '\<north' datafile
```

● 找出所有包含 以 west 结尾 的单词的行

```
grep 'west\>' datafile
```

● 找出所有包含 north 单词的行

```
grep '\<north\>' datafile
```

- grep递归搜索目录中的所有文件

```
grep -r pattern directory
```

- grep与管道

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

如果传递给 grep 的文件名参数中有目录的话，
需使用“ -d ”选项 directories are read in the same manner as normal files

```
grep -d [ACTION] directory_name
```

其中 ACTION 可以是
read：把目录文件当作普通文件来读取
skip：目录将被忽略而跳过
recurse：递归的方式读取目录下的每一个文件，可以用选项 "-r" 代替 "-d recurse"

```
grep -rl "eth0" /etc
```

>> sed 流编辑器

sed逐行处理文件或输入，并将输出结果发送到屏幕。不会修改输入文件的内容。

- sed 的 3 种调用方式

```
sed [-n][-e] 'sed_cmd' inputfile
```

● sed_cmd：使用格式：[address]sed_edit_cmd（通常用单引号括起来），其中 address 为 sed 的行定位模式，用于指定将要被 sed 编辑的行。如果省略，sed 将编辑所有的行。sed_edit_cmd 为 sed 对被编辑行将要进行的编辑操作。

● -n：缺省情况下，sed 在将下一行读入缓冲区之前，自动输出行缓冲区中的内容。此选项可以关闭自动输出。

● -e：允许调用多条 sed 命令，如：

```
sed -e 'sed_cmd1' -e 'sed_cmd2' input_file
```

```
sed [选项] -f sed_script_file inputfile
```

- 将sed命令插入脚本文件，生成sed可执行脚本，然后在命令行中直接键入脚本文件名来执行

- sed_cmd 中 addres 的定位方式

n	表示第 n 行
$	表示最后一行
m,n	表示从第 m 行到第 n 行
/pattern/	查询包含指定模式的行。如 /disk/、/[a-z]/
/pattern/,n	表示从包含指定模式的行 到 第 n 行
n,/pattern/	表示从第 n 行 到 包含指定模式的行
/模式1/,/模式2/	表示从包含模式1 到 包含模式2的行
!	反向选择， 如 m,n! 的结果与 m,n 相反

- 常用的 sed_edit_cmd

p	打印匹配行	s	替换命令
=	显示匹配行的行号	l	显示指定行中所有字符
d	删除匹配的行	r	读文件
a\	在指定行后面追加文本	w	写文件
i\	在指定行前面追加文本	n	读取指定行的下面一行
c\	用新文本替换指定的行	q	退出 sed

◆ s：替换命令，使用格式为：

```
[address]s/old/new/[gpw]
```

- address ：如果省略，表示编辑所有的行。
- g ：全局替换
- p ：打印被修改后的行
- w fname ：将被替换后的行内容写到指定的文件中

```
sed -n 's/west/east/gp'  datafile
```

```
sed -n 's/Aanny/Anndy/w newdata'  datafile
```

```
sed 's/[0-9][0-9]$/&.5/'  datafile
```

& 符号用在替换字符串中时，代表 被替换的字符串

19

◆ r：读文件，将另外一个文件中的内容附加到指定行后。

```
sed -n '$r newdata'  datafile
```

◆ w：写文件，将指定行写入到另外一个文件中。

```
sed -n '/west/w newdata'  datafile
```

◆ n：将指定行的下面一行读入编辑缓冲区。

```
sed -n '/eastern/{n;s/AM/Archie/p}' datafile
```

对指定行同时使用多个 sed 编辑命令时，需用大括号“{}”括起来，命令之间用分号“;”格开。注意与 -e 选项的区别

- sed_cmd 中可以使用shell变量，此时应使用 双引号
- 一些sed行命令集合

'/north/p'	打印所有包含 north 的行
'/north/!p'	打印所有不包含 north 的行
's/\.$//g'	删除以句点结尾的行中末尾的句点
's/^ *//g'	删除行首空格（命令中 ^ * 之间有两个空格）
's/ */ /g'	将连续多个空格替换为一个空格 命令中 */ 前有三个空格，后面是一个空格
'/^$/d'	删除空行
's/^.//g'	删除每行的第一个字符，同 's/.//'
's/^/%/g'	在每行的最前面添加百分号 %
'3,5s/d/D/'	把第 3 行到第 5 行中每行的 第一个 d 改成 D

>> awk 文本分析

awk 逐行扫描输入，按给定的模式查找出匹配的行，然后对这些行执行命令指定的操作。不会修改输入内容。

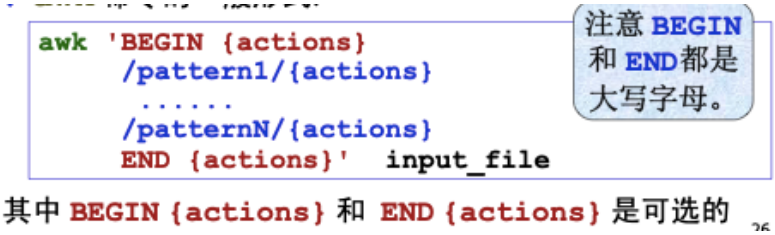
- awk的 3 种调用方式

```
awk [-F 字段分隔符] 'awk_script' input_file
```

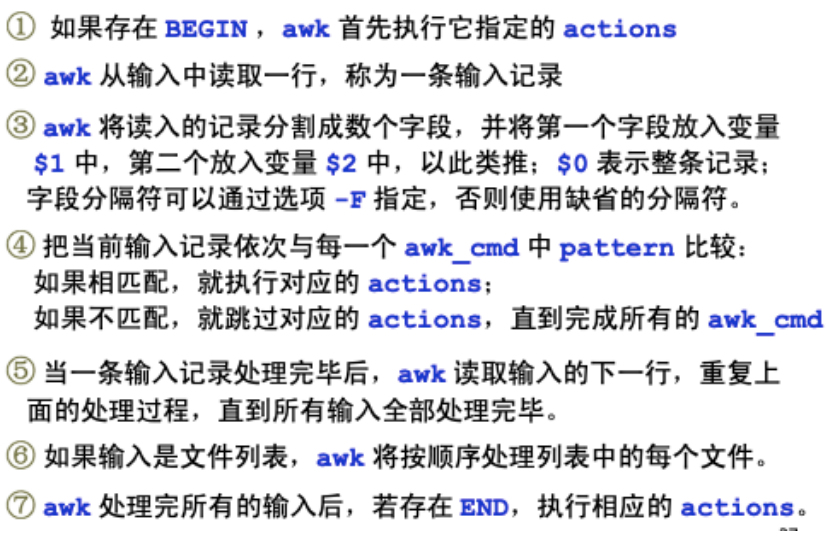
```
awk -f awd_script input_file
```

  - 将awk命令插入脚本文件，生成awk可执行脚本，然后在命令行中直接键入脚本文件名来执行

- awk_script 可以由一条或者多条 awk_cmd 组成，每条 awk_cmd 各占一行
  - 每个 awk_cmd 由两部分组成：/pattern/{actions}
  - awk_cmd 中的 /pattern/ 和 {actions} 可以省略，但不能同时省略
    - /pattern/ 省略时表示对所有的输入执行指定的 {actions}
    - {actions} 省略时表示打印匹配行
- awk命令的一般形式



- awk执行过程



- 模式匹配

① 使用正则表达式：/rexp/，如 /^A/、/A[0-9]*/

awk 中正则表达式中常用到的元字符有：

^	只匹配行首（可以看成是行首的标志）
$	只匹配行尾（可以看成是行尾的标志）
*	一个单字符后紧跟 *，匹配 0个或多个此字符
[ ]	匹配 [] 内的任意一个字符（[^] 反向匹配）
\	用来屏蔽一个元字符的特殊含义
.	匹配任意单个字符
str1|str2	匹配 str1 或 str2
+	匹配一个或多个前一字符
?	匹配零个或一个前一字符
( )	字符组

29

② 使用布尔（比较）表达式，表达式的值为真时执行相应的操作 (actions)

- 表达式中可以使用变量（如字段变量 $1,$2 等）和 /rexp/
- 表达式中的运算符有
  - 关系运算符: < > <= >= == !=
  - 匹配运算符: ~ !~

    x ~ /rexp/ 如果 x 匹配 /rexp/，则返回真；

    x!~ /rexp/ 如果 x 不匹配 /rexp/，则返回真。

```
awk '$2 > 20 {print $0}' shipped
```

```
awk '$4 ~ /^6/ {print $0}' shipped
```

  - 复合表达式： &&（逻辑与）；||（逻辑或）；!（逻辑非）
    - 表达式中有比较运算时，一般用圆括号括起来

- 字段分隔符：

  awk中的字段分隔符可以用 -F 指定，缺省是空格

```
awk -F: '{print $1}' datafile
awk -F'[ :]' '{print $1}' datafile
```

- 重定向与管道

```
awk '{print $1, $2 > "output"}' datafile
awk 'BEGIN{"date" | getline d; print d}'
```

>> sort 文本内容排序

```
sort [-bcdfimMnr][-o<输出文件>][-t<分隔字符>][+<起始栏位>-<结束栏位>][--help][--verison][文件]
```

选项	解释
-r	reverse，按降序排序。sort默认是升序排序。
-n	依照数值的大小排序。
-t <分隔字符>	指定排序时所用的栏位分隔字符。
-k num	按照第num列进行排序

```
# example：按照第三列数值升序排序
sort -t: -k 3n /etc/passwd | more
```

5 目录操作

>> ls 显示目录中的内容

```
ls [-alrtAFR] directory
```

选项	解释
-1	每列只显示一个文件或者目录名称
-a	显示所有文件或目录，包括隐藏文件(以“.”开头)、"." 和 ".."
-l	使用详细格式列表。 将权限、硬件、拥有者、群组名称、文件或目录大小、修改时间一并列出
-r	将文件以相反次序显示(原定依英文字母次序)
-t	将文件依建立时间之先后次序列出
-A	同-a，但不列出 "." (目前目录) 及 ".." (父目录)
-R	递归处理

>> pwd 显示当前工作目录

```
pwd # 没有参数，显示当前工作目录的绝对路径名称
```

>> cd 改变用户工作目录

```
cd 目录名      # 进入某个目录
cd ~用户名     # 进入用户的注册目录
```

>> mkdir 建立用户目录（同时给目录设置权限）

```
mkdir [-p][-m] 文件名
```

选项	解释
-p	若要建立目录的上层目录仍未创建 会一并创建上层目录
-m	建立目录时，设置目录的权限。 权限设置方法与chmod相同

example

```
mkdir -m 777 team2  # 建立目录，并让所有人拥有rwx的权限
```

>> rmdir 删除目录

```
rmdir [-p] 目录名  # -p:删除指定目录后，若上层目录已经为空则一并删除
```

>> cp 复制文件或目录

```
cp 源文件名 目标文件名
cp -r 源目录名 目标目录名
```

>> mv 移动文件和文件改名

```
mv 源文件列表 目标文件
```

example

```
mv team01/file1 file2 /team02  # 将team01下的两个文件移动到目录team02下
```

>> rm 删除文件或目录

```
rm [选项] 文件列表
```

选项	解释
-r	递归删除
-i	指定交互模式。在执行删除前提示确认
文件列表	希望删除的文件或目录名，用空格分隔

>> ln 在文件间建立连接

ln命令用来建立硬连接和符号连接。

- 硬连接是一个文件额外的名字，没有源文件，硬连接不能存在。
- 对于符号连接，当源文件被删除后，符号连接仍然存在

```
In [选项] 源文件 目标文件
In [选项] 源文件列表 目标目录
```

选项	解释
-s	建立一个符号连接而不是硬连接
-d	指定交互模式。在执行删除前提示确认

>> find 查找特定的文件

```
find path -option [-print] [-exec -ok command ] {} \;
```

选项	解释
-atime n	至少n*24小时内没有被访问过的文件
-ctime n	至少n*24小时内没有被修改过的文件
-amin n	n分钟之前访问过的文件
-cmin n	n分钟之前修改过的文件
-empty	文件为空
-name name_str	指定要寻找的文件或目录名名称为 name_str 可以使用正则表达式
-type x	以文件的类型作为要寻找的条件 若x为d，表示要寻找目录；为f表示要寻找普通文件 为c表示要寻找字符特殊设备，为b表示要寻找特殊块设备 为p表示寻找命名管道，为l表示要寻找符号连接
-size num	指定查找的文件大小必须大于num, k代表KB

example

```
# 找出目录中名字为core的文件并删除
find /usr/src -name core -exec rm {} \;
```

```
# 找出目录中至少7天没有被访问过的文件
find /home -atime 7 -print
```

```
# 找出目录中所有的jpg文件，且大小要超过100K
find /home -name "*.jpg" -size 100k
```

>> touch 改变文件的时间参数

touch：改变文件访问和修改时间；或者用指定时间创建新文件。

```
touch [选项] MMDDhhmmYY 文件列表
```

选项	解释
-a	只更改访问时间
-c	若目标文件不存在，不建立空的目标文件

6 文件显示操作

>> cat 显示和合并文件

可以结合多个文件，并将文件内容输出到标准输出设备。

```
cat [选项] 文件列表
```

选项	解释
-b	列出文件内容时，在所有非空白列开头编号，从1开始
-E	在每一列的最后标上 $ 号
-n	列出文件内容时，在每一列开头标上编号，从1开始

>> more 分屏显示文件

```
more [选项] 文件名
```

选项	解释
-<行数>	指定每次要显示的行数
+/<字符串>	在文件中查找选项中指定的字符串 然后显示字符串所在该页的内容
+<行数>	从指定的行数开始显示
-n	每次只显示n行
-c	不滚屏，在显示下一屏之前先清屏

>> head 显示文件的前几行

```
head [选项] 文件名
```

```
head -c N file # 显示前N个字节
head -n N file # 显示前N行
```

>> tail 显示文件的后几行

```
tail -c N file # 显示前N个字节
tail -n N file # 显示前N行
tail +N file   # 从第N行开始显示
```

7 文件权限操作

一个普通文件，r = 可以查看文件内容，w = 可以修改文件内容，x = 可以执行文件。

一个路径，r = 可以查看文件夹下的文件，w = 可以在文件夹下创建和删除文件，x = 可以进入文件夹或访问文件夹下的文件

	user	group	others
符号	rwx	rw-	r--
二进制	111	110	100
八进制	7	6	4

>> chmod 改变文件或目录的许可权限

```
chmod [选项] 模式 文件列表
```

选项	解释
u	owner of the file
g	owner's group
o	other users on the system
+	add permissions
–	remove permissions
=	clears permissions and sets to mode specified

>> chown 改变文件的所有权

```
chown [选项] 用户 文件列表
```

选项	解释
-v	详细说明所有权的变化
-r	递归改变目录及其内容的所有权

>> chgrp 改变用户分组

```
chgrp [选项] 用户 文件列表
```

选项	解释
-v	详细说明文件所属用户组的变化
-r	递归改变目录及其内容的所属的用户组

8 Vim 编辑器

三种基本工作模式： Normal 模式 / Insert模式 / Command 模式

- Normal 模式：启动vim后进入的默认模式，在该模式下，任何从键盘上输入的字符都被解释为命令。其他模式可以摁 Esc 来进入 Normal 模式。
- Inser 模式：在该模式下，任何从键盘上输入的字符都被vim当作文件内容保存起来。
- Command 模式：在该模式下，vim 会在窗口的最后一行显示一个冒号，作为command模式的提示符，等待用户输入命令。Command模式中所有命令都必须按回车后执行。执行后vim自动回到Normal模式。

命令	功能
`vim filename`	从第一行开始编辑 filename 文件
`vim +n filename`	从第 n 行开始编辑filename 文件
`vim + filename`	从最后一行开始编辑 filename 文件
`vim +/pattern filename`	从包含pattern的第一行开始编辑文件
`vim -r filename`	在系统崩溃之后恢复 filename 文件
`vim -R filename`	以只读方式编辑 filename 文件

8.1 退出vim

- 保存文件并退出 vim，可在 Normal 模式下
  - 连按两次大写字母 Z，若文件被修改过，则保存后退出
  - :w 保存当前编辑文件，但并不退出
  - :q 系统退出 vim 返回到 shell
  - :wq 表示存盘并退出
  - :q! 放弃所作修改而直接退到 shell 下
  - :x 同 Normal 模式下的 ZZ 命令功能相同
  - :w fname 另存为，新文件名为 fname

8.2 光标移动操作

❑ Normal 模式下的光标的简单移动方式

光标左移一格	h 或 BACKSPACE
光标右移一格	l 或 SPACE
光标上移一行	k 或 Ctrl+p
光标下移一行	j 或 Ctrl+n

- 在命令前输入一个数字 n (重复因子)，则光标就相应移动 n 个位置。如 5h 表示向左移动 5 个位置
- Normal 模式下也可以使用方向键移动光标

❑ Normal 模式下的光标的快速移动方式

^	光标移到所在行的开始
0	同 ^
$	光标移到所在行的末尾 (前面可加重复因子)
nG	光标移到第 n 行，若不指定 n，则移到最后一行
:$	光标移到最后一行
-	光标移到上面一行的开始 (前面可加重复因子)
+	光标移到下面一行的开始 (前面可加重复因子)
Enter	同 +

- Ctrl+g: 显示光标所在行位置以及文件状态信息

8.2.1 按单词移动光标

❑ 按单词 (word) 移动光标 (Normal 模式下)

◆ 在 vim 中 “单词” (word) 有两种含义。

- 广义的单词：它可以是两个空格之间的任何内容。
- 狭义的单词：此时，英文单词、标点符号和非字母字符（如 !、@、#、$、%、^、&、*、(、-、+、{、[、~、|、\、<、>、/ 等）均被当成是一个单词。

◆ vim 中一般大写命令中使用的是广义的单词，而小写命令则使用狭义的单词。

例： jypan@math.ecnu.edu.cn

一个广义单词，9个狭义单词

17

❑ 按字 (word) 移动光标 (Normal 模式下)

w	光标右移一个单词
W	右移一个以空格作为分隔符的单词
b	光标左移一个单词
B	左移一个以空格作为分隔符的单词
e	光标右移到一个单词的结尾
E	右移一个以空格作为分隔符的单词结尾

- 注：以上命令前都可以加重复因子

8.3 文本输入

❑ 在 Normal 模式下用户输入的任何字符都被 vim 当作命令加以解释执行，如果用户要将输入的字符当作是文本内容时，则应将 vim 的工作模式从 Normal 模式切换到 Insert 模式

❑ 插入 (insert) 命令

i	从光标所在位置前开始插入文本，插入过程中可以使用 BACKSPACE 键删除错误的输入
I	从当前行的行首前开始插入文本

❑ 附加 (append) 命令

a	在光标当前所在位置之后追加新文本
A	从光标所在行的行尾开始插入新文本

❑ 新开行 (open) 命令

o	在光标所在行的下面新开一行，并将光标置于该行的行首，等待输入文本
O	在光标所在行的上面插入一行，并将光标置于该行的行首，等待输入文本

8.4 文本删除

❑ 在 Insert 模式下，可以用 BACKSPACE 或 Delete 键将输错或不需要的文本删除，但此时有一个限制就是只能删除本行的内容，无法删除其它行的内容。

❑ 在 Normal 模式下，vim 提供了许多删除命令。这些命令大多是以 d 开头的。

● 删除单个字符（删除少量字符的快捷方法）

x	删除光标处的字符。若在 x 之前加上一个数字 n，则删除从光标所在位置开始 向右的 n 个字符
X	删除光标前的那个字符。若在 X 之前加数字 n，则删除从光标前那个字符开始 向左的 n 个字符

◆ 删除多个字符

dd	删除光标所在的整行。在 dd 前可加上一个数字 n，表示删除当前行及其后 n-1 行的内容。
D，d$	删除从光标所在处开始到行尾的内容
d0	删除从光标前一个字符开始到行首的内容。
dw	删除一个单词。若光标处在某个词的中间，则从光标所在位置开始删至词尾。可在 dw 之前加一个数字 n，表示删除 n 个指定的单词。

类似的命令还有：dH、dM、dL ...

8.5 取消和重复

❑ 取消上一命令，也称复原命令，是非常有用的命令，它可以取消前一次的误操作或不合适的操作对文件造成的影响，使之回复到这种误操作或不合适操作被执行之前的状态。

u	取消前一次的误操作，可连续使用，以取消更前的误操作。
U	将当前行恢复到该行的原始状态。如果用 U 命令后再按 U 键时，表示取消刚才 U 命令执行的操作，也就是又恢复到第一次使用 U 命令之前的状态，结果是什么都没做。
Ctrl+r	撤消以前的撤消命令，恢复以前的操作结果。

❑ 重复命令也是一个非常常用的命令。在文本编辑中经常会碰到需要机械地重复一些操作，这时就需要用到重复命令。它可以让用户方便地再执行一次前面刚完成的某个复杂的命令。

❑ 重复命令只能在 Normal 模式下工作，在该模式下按点“.”键即可。

CUDA/Cpp 编程

> Compiler : NVCC
> device = GPU
> host = CPU
> kernel = functions that run on the device

1 NVIDIA GPU

GPU（Graphics Processing Unit），图形处理器，又称显卡，是一种专门在个人电脑、工作站、游戏机和一些移动设备（如平板电脑、智能手机等）上做图像和图形相关运算工作的微处理器。

目前全球有两大GPU生产商，分别是NVIDIA和ATI(已被AMD收购)。这里主要介绍NVIDIA GPU。

1.1 影响GPU性能的因素

1. **显存的类型**
   显存一般分为两类：GDDR显存是焊装在GPU附近的PCB板上；HBM2显存则被封装在GPU芯片内部，因而HBM2显存与GPU的通讯速率较高。

其他的一些显存常用指标为：容量、带宽、位宽和速率。
位宽是指GPU一次能传递的数据宽度，位宽越大，一次性能传输的数据就越多，显卡的性能提升就越明显。

$$\text{显存带宽}(GB/s) = \text{显存数据频率}(Gbps) \times \text{显存等效位宽}(bit)/8$$

2. **CUDA core(SP) 数量**

SP (Streaming Processor)，流处理单元，也称为CUDA core，是GPU上最基本的处理单元。在GPU上运行的具体的指令和任务都是在SP上处理的。GPU进行并行计算，也就是很多个SP同时做处理。因此SP数目越多，GPU的性能就越强大。

3. **GPU之间数据传输速率**
   GPU之间的互联通信有两种方案：NVLink方案和PCIe方案。使用NVLink互联的GPU通信速率一般是PCIe的近十倍。

1.2 显卡比较：GeForce与Tesla

1. **ECC内存的错误检测和纠正**
   GeForce系列显卡不具备错误检测和纠正的功能，但Tesla系列GPU因为GPU核心内部的寄存器、L1/L2缓存和显存都支持ECC校验功能，所以Tesla不仅能检测并纠正单比特错误也可以发现并警告双比特错误，这对保证计算结果的准确性来说非常重要.
2. **DAAA引擎**
   GeForce产品一般只有单个DMA引擎，同时只能在一个方向上传输数据。如果数据正在上传到GPU，则在上传完成之前，无法返回由GPU计算的任何结果。同样，从GPU返回的结果将阻止任何需要上传到GPU的新数据。

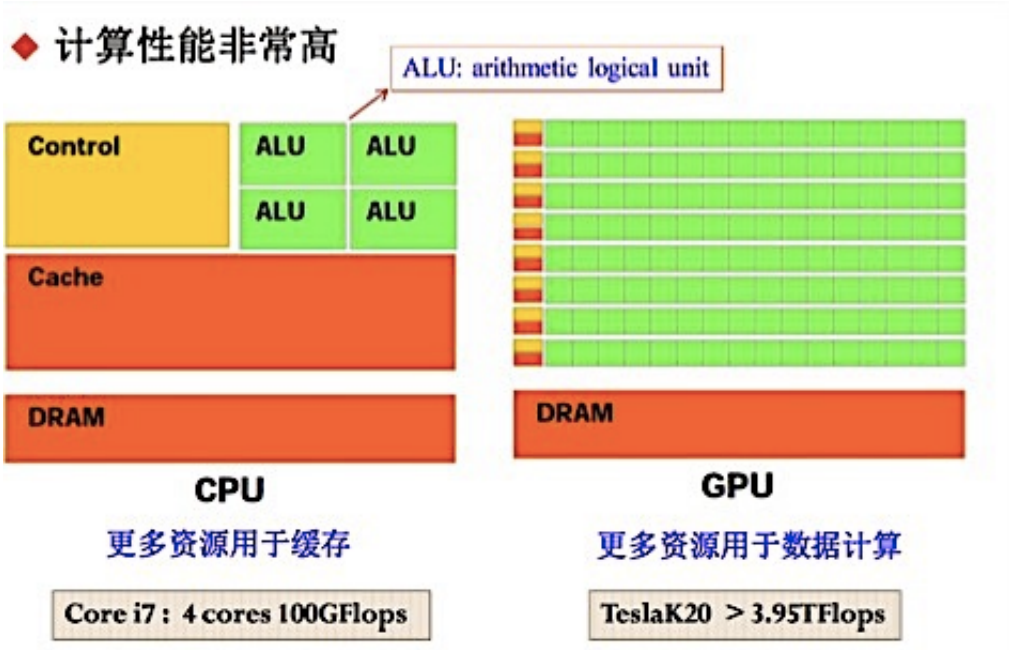
Tesla GPU产品采用双DMA引擎，数据可以在CPU和GPU之间同时输入和输出，无需等待，效率更高.

3. **GPU Direct RDAAA**
   NVIDIA的GPU-Direct技术可大大提高GPU之间的数据传输速度，RDMA功能则可以对多台机器之间的数据传输提供最大的性能提升。GeForce GPU只能支持单台机器内部的P2P GPU Direct,不支持跨主机的GPU-Direct RDMA。Tesla GPU 则完全支持 GPU Direct RDMA 和各种其他 GPU Direct 功能，这对GPU机器的集群部署非常有帮助。
4. **Hyper-Q的支持**
   Hyper-Q代理允许多个CPU线程或进程在单个GPU上启动工作。GeForce GPU仅仅支 持CUDA Streams的Hyper-Q，也就是说GeForce只能从单独的CPU内核有效地接受并运行并行计算，但跨多台计算机运行的应用程序将无法有效地启动GPU上的工作。Tesla则具备完整的Hpyer-Q支持能力，更适合多个GPU集群的并行计算
5. **Volta架构中的 Tensor Core**

2 GPU架构 -- 硬件角度

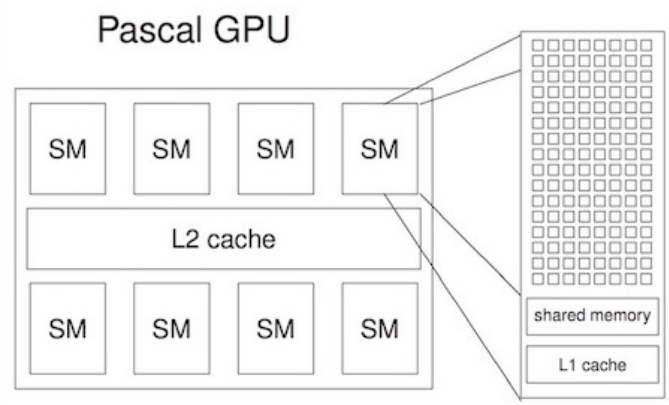
2.1 General Idea : GPU v.s. CPU

- GPU与CPU结构对比：



Pros of GPU		Cons of GPU
更大的内存、带宽		不能并行化的工作帮助不大
更大量的执行单元		不具有分支预测
对比cpu，显卡的价格较为低廉		GPGPU模型尚不成熟

2.2 Hardware View of GPU



$$GPU = \text{显存}(L1cache + L2cache) + \text{计算单元}$$

2.3 厘清SP, SM

首先要明确：SP（streaming processor），SM（streaming multiprocessor）是硬件（GPU hardware）概念。
而thread，block，grid，warp是软件上的（CUDA）概念。

- 术语释义
  - **SP**（Streaming Processor），流处理单元，也称为CUDA core，是GPU上最基本的处理单元。在GPU上运行的具体的指令和任务都是在SP上处理的。一个 SP 对应一个 thread。

- **Warp**：warp 是 SM 调度和执行的基础概念，通常一个 SM 中的 SP(thread) 会分成几个 warp (也就是SP在SM中是进行分组的，物理上进行的分组)。一般每一个 warp 中有32个 thread。这个 warp 中的32个thread(sp) 是一起工作的，执行相同的指令，如果没有这么多 thread 需要工作，那么这个 warp 中的一些 thread(sp) 是不工作的。

  （每一个线程都有自己的寄存器内存和 local memory，一个warp中的线程是同时执行的，也就是当进行并行计算时，线程数尽量为32的倍数。如果线程数不上32的倍数的话：假如是1，则warp会生成一个掩码，当一个指令控制器对一个warp单位的线程发送指令时，32个线程中只有一个线程在真正执行，其他31个 进程会进入静默状态。）

- **SM**（Streaming Multiprocessor）：多个SP加上其他的一些资源组成一个SM。也叫GPU大核，包括其他资源如：warp scheduler，register，shared memory等。

  一个 SM 中的所有 SP 是先分成 warp 的，共享同一个memory和instruction unit（指令单元）。
  SM 可以看做GPU的心脏（对比CPU核心），register 和 shared memory 是 SM 的稀缺资源。CUDA将这些资源分配给所有驻留在 SM 中的threads。因此，这些有限的资源就使每个 SM 中 active warps 有非常严格的限制，也就限制了并行能力。参考链接。

2.4 总结

GPU中每个SM都设计成支持数以百计的线程并行执行，并且每个GPU都包含了很多的SM，所以GPU支持成百上千的线程并行执行。当一个kernel启动后，thread会被分配到这些SM中执行。大量的thread可能会被分配到不同的SM，**同一个block中的threads必然在同一个SM中并行执行**。每个thread拥有自己的程序计数器和状态寄存器，并且用该线程自己的数据执行指令，这就是所谓的单指令多线程结构(SIMT)。

一个 SP 可以执行一个thread，但是**实际上并不是所有的thread能够在同一时刻执行**。NVIDIA把32个threads组成一个warp。**warp是调度和运行的基本单元。warp中所有threads并行的执行相同的指令。**一个warp 需要占用一个SM运行，多个 warps 需要轮流进入 SM。由SM的硬件 warp scheduler 负责调度。目前每个 warp 包含 32 个 threads（Nvidia保留修改数量的权利）。所以，一个GPU上 resident thread 最多只有 $SM \times warp$ 个。

3 What is CUDA

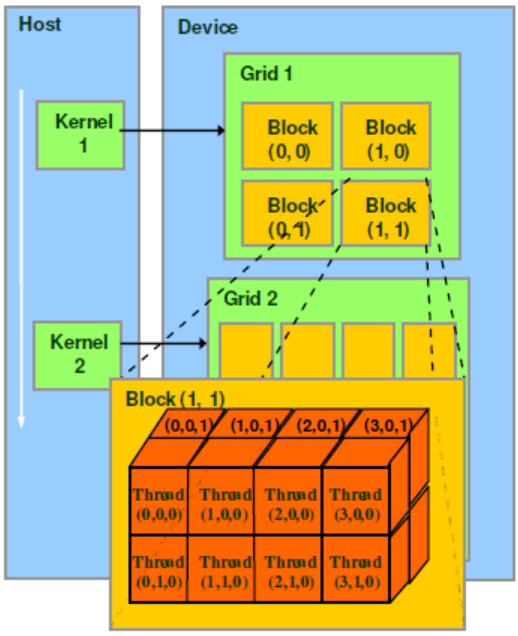
CUDA的全名是Computed Unified Device Architecture，是一个统一的计算框架。CUDA既不是一个软件也不是一个纯硬件，是软硬结合的计算体系。它是由NVIDIA推出的通用并行计算框架，包含CUDA指令集架构和GPU内部的并行计算引擎。

从CUDA体系结构的组成来说，包含了三个部分：开发库(SDK)，运行环境(toolkit)和驱动(driver)。

它诞生是为了让GPU能够有可用的编程环境，使得开发人员可以用程序控制GPU的硬件进行并行计算。

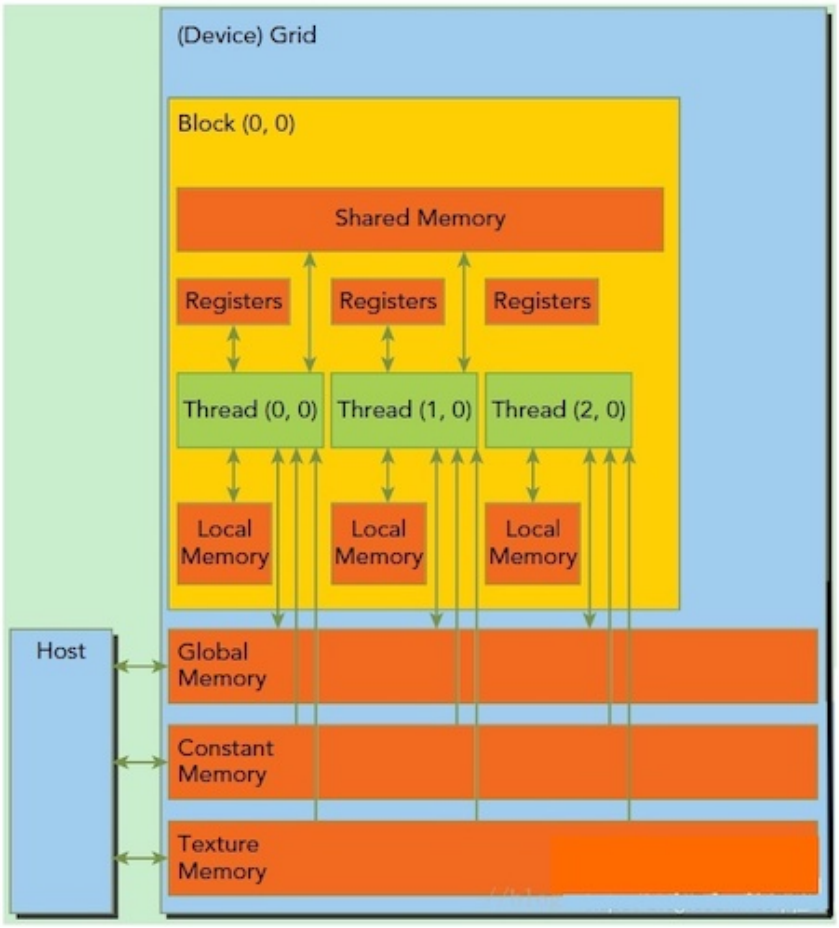
CUDA控制的GPU程序运行过程

- At the top level，我们有一个主进程，该主进程在CPU上运行并执行以下步骤：
    1. 初始化GPU卡
    2. 在Host和Device上分配内存
    3. 数据从Host复制到Device内存
    4. launches multiple instances of execution “kernel” on device
    5. 将数据从Device内存复制到Host
    6. 根据需要重复3-5
    7. 释放所有内存并终止



- 而在GPU上，有以下过程：(instance == block)
    1. execution kernel 的每个 instance 都在SM上执行
    2. 如果 instance 数超过了SM的数量：如果有足够的寄存器和共享内存，则每个SM一次将运行多个instance。其他 instance 将在队列中等待并稍后执行.
    3. 一个 instance 中的所有线程都可以访问本地共享内存 (local shared memory)，但看不到其他 instance 在做什么（即使它们在同一SM上）。
    4. 无法保证 instance 的执行顺序。

4 CUDA内存模型 -- 软件角度



- **thread**：一个CUDA的并行程序会被以许多个threads来执行。每个 thread 都有自己的一份 register 和 local memory 的空间。
- **block**：一组 thread 构成一个 block，这些 thread 共享一份 shared memory。
- **grid**：多个blocks会构成一个grid。不同的 grid 有各自的 global memory、constant memory 和 texture memory。**同一个grid内所有的thread (包括不同block的thread) 都可以共享这些 global memory、constant memory、和 texture memory。**

线程访问这几类存储器的速度是：register > local memory > shared memory > global memory。

每一个时钟周期内，warp（一个block里面一起运行的thread）包含的thread数量是有限的，现在的规定是32个。其中各个线程对应的数据资源不同 (指令相同但是数据不同）。一个block中含有 16 个 warp。所以一个block中最多含有512个线程。

5 编程要点

5.1 Scope Specifier

通过关键字定义函数，控制某个程序在CPU上跑还是在GPU上跑

	Execute On	Callable From
__device__ float DeviceFunc	device	device
__global__ void KnernelFunc	device	host
__host__ float HostFunc	host	host

5.2 CPU和GPU间的数据传输

- GPU内存分配/回收内存的函数接口：

```
cudaError_t cudaMalloc(void **devPtr, size_t  size );
cudaError_t cudaFree(void *devPtr);
```

- 数据传输的函数接口：

```
cudaError_t cudaMemcpy(
  void *dst, const void *src, size_t count, enum cudaMemcpyKind kind);
```

其中 cudaMemcpyKind 有几种类型，分别是：

```
cudaMemcpyHostToDevice; // CPU到GPU
cudaMemcpyDeviceToHost; // GPU到CPU
cudaMemcpyDeviceToDevice; // GPU到GPU
```

5.3 Kernel

怎么用代码表示线程组织模型?

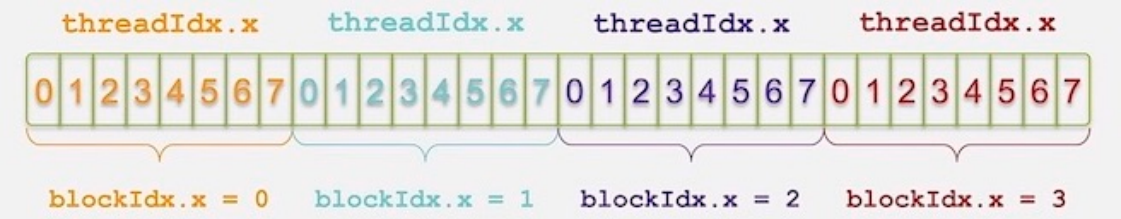
Triple angle brackets mark a call from host code to device code (also called a "kernel launch").

```
cudaError_t kernel_routine<<<gridDim, blockDim>>>(args);
// 函数名称<<<block数目，thread数目，shared memory大小>>>(参数...);
```

- **gridDim** is the number of instances of the kernel (the “grid” size), i.e., number of blocks
- **blockDim** is the number of threads within each instance (the “block” size) , i.e. , number of threads.
- The more general form **allows gridDim and blockDim to be 2D or 3D** to simplify application programs

5.4 数据在GPU内存的存放

当kernel想要取用某一块内存的数据时，需要计算数据所在的位置：




```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

一个问题：当数据并不正好是blockDim.x的倍数的时候，该怎么办?

Ans: 在kernelFunc中多传一个参数，避免溢出

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

```
// 同时需要更新KernelFunc
kernel_routine<<< (N+M-1)/M, M>>>(da, db, dc, N);
```

- Sample Code：**practice1c**

```
// include files
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "helper_cuda.h"

// random init
void random_inits(int *a, int N){
  srand(2019);
  for(int i=0; i<N; i++){
    a[i] = rand()%100 + 1;
  }
}

//kernel routine
__global__ void add_two_vec(int *a, int *b, int *c, int n){
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  if (tid < n) c[tid] = a[tid] + b[tid];
}

// main code
int main(int argc, const char **argv){
  int *a, *b, *c;
  int *da, *db, *dc;
```

```
26    int nblocks, nthreads, nsize;
27
28    // initialise card
29    findCudaDevice(argc, argv);
30
31    // set number of blocks, and threads per block
32    nblocks  = 2;
33    nthreads = 8;
34    nsize    = nblocks*nthreads ;
35
36    // init vector in host device
37    a = (int *)malloc(nsize*sizeof(int)); random_inits(a, nsize);
38    b = (int *)malloc(nsize*sizeof(int)); random_inits(b, nsize);
39    c = (int *)malloc(nsize*sizeof(int));
40
41    // allocate memory for array
42    checkCudaErrors(cudaMallocManaged(&da, nsize*sizeof(int)));
43    checkCudaErrors(cudaMallocManaged(&db, nsize*sizeof(int)));
44    checkCudaErrors(cudaMallocManaged(&dc, nsize*sizeof(int)));
45
46    // copy data from host to device
47    checkCudaErrors(cudaMemcpy(da, a, nsize*sizeof(int),
cudaMemcpyHostToDevice));
48    checkCudaErrors(cudaMemcpy(db, b, nsize*sizeof(int),
cudaMemcpyHostToDevice));
49
50    // execute kernel
51    add_two_vec<<<nblocks,nthreads>>>(da, db, dc, nsize);
52    getLastCudaError("add_teo_vec failed\n");
53
54    // Get back to Host
55    checkCudaErrors(cudaMemcpy(c, dc, nsize*sizeof(int),
cudaMemcpyDeviceToHost));
56
57    // Print
58    for (int i=0; i<nsize; i++){
59      printf(" a,  b, c  =  %d  %d  %d \n", a[i], b[i], c[i]);
60    }
61
62    // Clean up
63    checkCudaErrors(cudaFree(da));
64    checkCudaErrors(cudaFree(db));
65    checkCudaErrors(cudaFree(dc));
66    free(a); free(b); free(c);
67
68    return 0;
69  }
```

5.5 Others

- **Coordinating Host & Device**
  - Kernel launches are asynchronous
  - CPU needs to synchronize before consuming the results
    - **cudaMemcpy()** Blocks the CPU until the copy is complete. Copy begins when all preceding CUDA calls have completed
    - **cudaMemcpyAsync()** Asynchronous, does not block the CPU.
    - **cudaDeviceSynchronize()** Blocks the CPU until all preceding CUDA calls have completed
- **Reporting Errors**
  All CUDA API calls return an error code (**cudaError_t**). It is : Error in the API call itself **OR** Error in an earlier asynchronous operation (e.g. kernel).
  To get the error code for the last error:

```
cudaError_t cudaGetLastError(void);
```

  To get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t);

// for example
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

- **Device Management**
  Application can query and select GPUs

```
cudaError_t cudaGetDeviceCount(int *count);
cudaError_t cudaSetDevice(int device);
cudaError_t cudaGetDevice(int *device);
cudaError_t cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

  Multiple threads can sharea device.

  A single thread can manage multiple devices.

```
cudaError_t cudaSetDevice(i); //to select current device
cudaError_t cudaMemcpy(...); //for peer-to-peer copies
```

6 内存共享与同步: an example

__shared__ declares data that is available for **all threads within a certain block**. (not visible to threads in other blocks)

Use **__syncthreads()** as a barrier to prevent data hazard.

```
__global__ void stencil_1d(int *in, int *out) {
    // declare a shared array, availabel for all threads in this block
    __shared__ int temp[blockDim.x + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex – RADIUS] = in[gindex – RADIUS];
        temp[lindex + blockDim.x] = in[gindex + blockDim.x];
    }

    // Because we have no idea which thread runs first, we need to
    // Synchronize (ensure all the data is available)
    __syncthreads( );

    // Apply the stencil (i.e. 1-D conv)
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

7 GCC命令(会考)

```
gcc [选项][参数] 源文件
```

选项	解释
-L	给gcc添加额外的搜索库的路径
-l	给gcc指定具体的库名
-o	生成目标文件名

gcc(选项)(参数)

编译成可执行文件
gcc –c –I /usr/dev/mysql/include test.c –o test.o

最后我们把所有目标文件链接成可执行文件:
gcc –L /usr/dev/mysql/lib –lmysqlclient test.o –o test

Linux下的库文件分为两大类分别是动态链接库（通常以.so结尾）和静态链接库（通常以.a结尾），二者的区别仅在于程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的

GCC在链接时优先使用动态链接库，只有当动态链接库不存在时才考虑使用静态链接库，如果需要的话可以在编译时加上-static选项，强制使用静态链接库
gcc –L /usr/dev/mysql/lib –static –lmysqlclient test.o –o test

- Contents of Makefile

```
INC := -I$(CUDA_HOME)/include -I.
LIB := -L$(CUDA_HOME)/lib64 -lcudart

NVCCFLAGS := -lineinfo -arch=sm_35 --ptxas-options=-v --use_fast_math
# NVCCFLAGS := -lineinfo -arch=sm_50 --ptxas-options=-v --use_fast_math

all:  prac1a prac1b prac1c

prac1a: prac1a.cu Makefile
  nvcc prac1a.cu -o prac1a $(INC) $(NVCCFLAGS) $(LIB)

prac1b: prac1b.cu Makefile
  nvcc prac1b.cu -o prac1b $(INC) $(NVCCFLAGS) $(LIB)

prac1c: prac1c.cu Makefile
  nvcc prac1c.cu -o prac1c $(INC) $(NVCCFLAGS) $(LIB)

clean:
  rm -f prac1a prac1b prac1c
```

8 Graphics的主要内容

建模(Modeling) 、渲染(Rendering)、动画(Animation)、人机交互(Human–computer Interaction, HCI)

PySpark 基础

1 使用spark运行python程序

```
$SPARK_HOME/bin/spark-submit YOUR_PYTHON_FILE_PATH <options>
```

2 创建pyspark工作环境

```
# 使用SparkContext创建
from pyspark import SparkContext,SparkConf
conf = SparkConf().setAppName("Project Name").setMaster("local[*]")
sc = SparkContext.getOrCreate(conf) #视情况新建session或利用已有的session
```

任何Spark程序都是SparkContext开始的，SparkContext的初始化需要一个SparkConf对象，SparkConf包含了Spark集群配置的各种参数(比如主节点的URL)。初始化后，就可以使用SparkContext对象所包含的各种方法来创建和操作RDD和共享变量。

```
# 使用SparkSession创建
from pyspark.sql import SparkSession
spark = SparkSession\
        .builder\
        .appName("TestSpark")\
        .getOrCreate()
dataRDD = spark.read.text(sys.argv[1]).rdd #读入数据，创建RDD
```

SparkSession是Spark 2.0引入的新概念。SparkSession为用户提供了统一的切入点，来让用户学习spark的各项功能。 SparkSession读入数据后返回的是一个DataFrame，可以使用. RDD 方法将之转换为RDD。

3 初始化RDD

3.1 sc.parallelize方法

适用于：本地内存中已经有一份序列数据(比如python的list)，可以通过**sc.parallelize**去初始化一个RDD。当执行这个操作以后，序列数据将被自动分块(partitioned)，并且把每一块送到集群上的不同机器上。

该方法可以把**Python list，NumPy array**或者**Pandas Series, Pandas DataFrame**转成Spark RDD。

```
# 利用list创建一个RDD
rdd = sc.parallelize([1,2,3,4,5],numSlices=None)
rdd.getNumPartitions() # 查看被划分成了几份
```

3.2 sc.textFile等方法

1. **读入文本文件**

创建RDD的另一个方法是直接把文本读到RDD。文本的每一行都会被当做一个item。

不过需要注意的一点是，Spark一般默认给定的路径是指向HDFS的，如果要从本地读取文件的话，给一个**file://**开头的全局路径。

```
# read txt file
rdd = sc.textFile("file://" + FilePath) # return an RDD of Strings.
```

一些简单的RDD方法：

```
rdd.first() # Return the first element in this RDD.
rdd.collect() # Return a list that contains all of the elements in this RDD.
```

可以**sc.wholeTextFiles读入整个文件夹的所有文件**。但是要特别注意，这种读法，**RDD中的每个item实际上是一个形如(文件名，文件所有内容)的元组**。

2. **读入CSV文件**

```
import csv
import StringIO

def loadRecord(line):
  """Parse a CSV line"""
  input = StringIO.StringTO(line)
  reader = csv.DictReader(input, fieldnames=["name, favouriteAnimal"])
  return reader.next()

input = sc.textFile(inputFile).map(loadRecord)
```

4 RDD Operations

Transformations	$map(f : T \Rightarrow U)$	: RDD[T] ⇒ RDD[U]
	$filter(f : T \Rightarrow Bool)$	: RDD[T] ⇒ RDD[T]
	$flatMap(f : T \Rightarrow Seq[U])$	: RDD[T] ⇒ RDD[U]
	$sample(fraction : Float)$	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (V, V) \Rightarrow V)$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	: (RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	: (RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : V \Rightarrow W)$	: RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : Comparator[K])$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : Partitioner[K])$	: RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	: RDD[T] ⇒ Long
	$collect()$	: RDD[T] ⇒ Seq[T]
	$reduce(f : (T, T) \Rightarrow T)$	: RDD[T] ⇒ T
	$lookup(k : K)$	: RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : String)$	: Outputs RDD to a storage system, *e.g.*, HDFS

4.1 常用的RDD Transformation

RDD API官方文档

- `map` (***f, preservesPartitioning=False***)

Return a new RDD by applying a function to each element of this RDD.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

- `mapPartitions` (***f, preservesPartitioning=False***)

Return a new RDD by applying a function to each partition of this RDD.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> def f(iterator): yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

- `mapValues` (***f***)

Pass each **value** in the **key-value pair RDD** through **a map function** without changing the keys; this also retains the original RDD's partitioning.

```
>>> x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b",
["grapes"])])
>>> def f(x): return len(x)
>>> x.mapValues(f).collect()
[('a', 3), ('b', 1)]
```

- `filter` (***f***)

Return a new RDD containing only the elements that satisfy a predicate.

```
>>> rdd = sc.parallelize([1, 2, 3, 4, 5])
>>> rdd.filter(lambda x: x % 2 == 0).collect()
[2, 4]
```

- `flatMap` (*f*, *preservesPartitioning=False*)

Return a new RDD by first applying a function to all elements of this RDD, and then **flattening the results**.

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
>>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
```

- `partitionBy` (*numPartitions*, *partitionFunc=...*)

Return a copy of the RDD partitioned using the specified partitioner.

```
>>> pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1]).map(lambda x: (x, x))
>>> sets = pairs.partitionBy(2).glom().collect()
>>> len(set(sets[0]).intersection(set(sets[1])))
0
```

- `glom` ()

Return an RDD created by coalescing all elements within **each partition into a list**.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> sorted(rdd.glom().collect())
[[1, 2], [3, 4]]
```

- `groupBy` (*f*, *numPartitions=None*, *partitionFunc=*)

Return an RDD of grouped items.

```
>>> rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
>>> result = rdd.groupBy(lambda x: x % 2).collect()
>>> sorted([(x, sorted(y)) for (x, y) in result])
[(0, [2, 8]), (1, [1, 1, 3, 5])]
```

- `groupByKey` (*numPartitions=None*, *partitionFunc=*)

Group the values for each key in the RDD **into a single sequence**. Hash-partitions the resulting RDD with numPartitions partitions.

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.groupByKey().mapValues(len).collect())
[('a', 2), ('b', 1)]
>>> sorted(rdd.groupByKey().mapValues(list).collect())
[('a', [1, 1]), ('b', [1])]
```

- 集合操作
  - **`cartesian`(*other*)**

    Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where `a` is in `self` and `b` is in `other`.

```
>>> rdd = sc.parallelize([1, 2])
>>> sorted(rdd.cartesian(rdd).collect())
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

  - **`union`(*other*)**

    Return the union of this RDD and another one.

```
>>> rdd = sc.parallelize([1, 1, 2, 3])
>>> rdd.union(rdd).collect()
[1, 1, 2, 3, 1, 1, 2, 3]
```

  - **`join`(*other*, *numPartitions=None*)**

    Return an RDD containing all pairs of elements with matching keys in `self` and `other`.

    Each pair of elements will be returned as a **(k, (v1, v2)) tuple**, where (k, v1) is in `self` and (k, v2) is in `other`. Performs a hash join across the cluster.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> sorted(x.join(y).collect())
[('a', (1, 2)), ('a', (1, 3))]
```

  - **`cogroup`(*other*, *numPartitions=None*)**

    For each key k in `self` or `other`, return a resulting RDD that contains a tuple with the list of values for that key in `self` as well as `other`.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> [(x, tuple(map(list, y))) for x, y in
sorted(list(x.cogroup(y).collect()))]
[('a', ([1], [2])), ('b', ([4], []))]
```

- `sample`(*withReplacement, fraction, seed=None*)

Return a sampled subset of this RDD.

Parameters

withReplacement – can elements be sampled multiple times (replaced when sampled out)

fraction – expected size of the sample as a fraction of this RDD's size without replacement

seed – seed for the random number generator

Note : This is **not guaranteed to provide exactly the fraction** specified of the total count of the given `DataFrame`.

```
>>> rdd = sc.parallelize(range(100), 4)
>>> 6 <= rdd.sample(False, 0.1, 81).count() <= 14
True
```

- `sortBy`(*keyfunc, ascending=True, numPartitions=None*)

Sorts this RDD by the given keyfunc

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[1]).collect()
[('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
```

- `sortByKey`(*ascending=True, numPartitions=None, keyfunc=...*)

Sorts this RDD, which is assumed to consist of (key, value) pairs.

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortByKey().first()
('1', 3)
>>> sc.parallelize(tmp).sortByKey(True, 1).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> sc.parallelize(tmp).sortByKey(True, 2).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> tmp2 = [('Mary', 1), ('had', 2), ('a', 3), ('little', 4), ('lamb', 5)]
>>> tmp2.extend([('whose', 6), ('fleece', 7), ('was', 8), ('white', 9)])
>>> sc.parallelize(tmp2).sortByKey(True, 3, keyfunc=lambda k:
k.lower()).collect()
[('a', 3), ('fleece', 7), ('had', 2), ('lamb', 5),...('white', 9),
('whose', 6)]
```

- `reduceByKey`(*func, numPartitions=None, partitionFunc=*)

Merge the values for each key using an associative and commutative reduce function.

This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.

Output will be partitioned with `numPartitions` partitions, or the default parallelism level if `numPartitions` is not specified. Default partitioner is hash-partition.

```
>>> from operator import add
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.reduceByKey(add).collect())
[('a', 2), ('b', 1)]
```

4.2 常用的RDD Action

- **`count`()**

Return the number of elements in this RDD.

```
>>> sc.parallelize([2, 3, 4]).count()
3
```

- **`countByKey`()**

Count the number of elements for each key, and **return the result to the master as a dictionary**.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.countByKey().items())
[('a', 2), ('b', 1)]
```

- **`collect`()**

Return a list that contains all of the elements in this RDD.

Note : This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver’s memory.

- **`take`(*num*)**

Take the first num elements of the RDD.

It works by first scanning one partition, and use the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

Note : This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver’s memory.

```
>>> sc.parallelize([2, 3, 4, 5, 6]).cache().take(2)
[2, 3]
>>> sc.parallelize([2, 3, 4, 5, 6]).take(10)
[2, 3, 4, 5, 6]
>>> sc.parallelize(range(100), 100).filter(lambda x: x > 90).take(3)
[91, 92, 93]
```

- `top` (*num*, *key=None*)

Get the top N elements from an RDD. It returns the **list sorted in descending order**.

Note : This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
>>> sc.parallelize([10, 4, 2, 12, 3]).top(1)
[12]
>>> sc.parallelize([2, 3, 4, 5, 6], 2).top(2)
[6, 5]
>>> sc.parallelize([10, 4, 2, 12, 3]).top(3, key=str)
[4, 3, 2]
```

- `reduce` (*f*)

Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
15
>>> sc.parallelize((2 for _ in range(10))).map(lambda x:
1).cache().reduce(add)
10
>>> sc.parallelize([]).reduce(add)
Traceback (most recent call last):
    ...
ValueError: Can not reduce() empty RDD
```

- `lookup` (*key*)

Return the list of values in the RDD for key key. This operation is done efficiently if the RDD has a known partitioner by only searching the partition that the key maps to.

```
>>> l = range(1000)
>>> rdd = sc.parallelize(zip(l, l), 10)
>>> rdd.lookup(42)  # slow
[42]
>>> sorted = rdd.sortByKey()
>>> sorted.lookup(42)  # fast
[42]
>>> sorted.lookup(1024)
[]
>>> rdd2 = sc.parallelize([(('a', 'b'), 'c')]).groupByKey()
>>> list(rdd2.lookup(('a', 'b'))[0])
['c']
```

- **save operations**, i.e. `saveAsTextFile`(***path*, *compressionCodecClass=None***)
- `foreach`(***f***)

Applies a function to all elements of this RDD.

```
>>> def f(x): print(x)
>>> sc.parallelize([1, 2, 3, 4, 5]).foreach(f)
```

5 SparkContext

5.1 Example

1. Sum: `accumulator`(***value*, *accum_param=None***)

   Create an `Accumulator` with the given initial value, using a given `AccumulatorParam` helper object to define how to add values of the data type if provided. Default AccumulatorParams are used for integers and floating-point numbers if you do not provide one. For other types, a custom AccumulatorParam can be used.

   ```
   >>> acBalTuples.collect()
   [('SB10001', '50000'),
    ('SB10002', '12000'),
    ('SB10003', '3000'),
    ('SB10004', '8500'),
    ('SB10005', '5000')]
   >>> balanceTotal = sc.accumulator(0.0) # create accumulator
   >>> balanceTotal.value
   0.0
   >>> acBalTuples.foreach(lambda bals: balanceTotal.add(float(bals[1])))
   #sum
   >>> balanceTotal.value
   78500.0
   ```

6 Useful Links

RDD

SparkContext

DataFrame

7 Spark 与 Hadoop

7.1 Mapreduce的缺陷

1. 最大缺点：中间过程需要存放到磁盘中，IO开销大；此外，服务器之间通信开销大
2. 抽象层次低，需要手工代码
3. 只提供Map和Reduce，表达能力欠缺
4. 对复杂计算的支持不足，需要开发者维护大量的Job
5. 处理逻辑隐藏在代码细节中，没有整体逻辑
6. Reduce任务必须等待所有的Map任务完成后才可以开始
7. 时延高，只能处理批数据，不适合交互式数据、实时数据、迭代式数据处理

7.2 RDD

- 什么是弹性分布式数据集 (RDD)？
  - 不变的，容错的，并行的数据结构
  - 显式的将数据存储到磁盘和内存中，并能控制数据的分区
- 为什么要RDD?
  - 因为RDD支持数据处理的四种常见模型
    - 迭代算法 / 关联查询 / MapReduce / 流式处理
- RDD的核心API：Map / Reduce (ReduceByKey) / Filter / FlatMap
- RDD的持久化：cache() 和 persist()
  - cache()只是缓存到默认的缓存级别：**只使用内存 StorageLevel.MEMORY_ONLY**
  - persist()可以自定义缓存级别
- RDD的创建方法：
  1. 从文件系统中创建
  2. 通过Scala集合对象并行化生成
  3. 通过对已经存在的RDD transform生成
  4. 通过改变其他RDD的持久化状态

7.3 Spark与Hadoop对比

	Spark	Hadoop
应用场景	内存可容纳 ∼ TB级 多次操作特定数据集，迭代运算 搜索引擎 - PageRank 计算相似 SSSP(单源最短路径)	极大数据量 ∼ >TB — PB级 单次海量数据的离线分析处理 大规模Web信息搜集 数据密集型并行计算
性能	适合数据不太大、内存放得下 重复读取同样数据迭代计算	适合不能全部读入内存的数据处理 比如单次读取、数据转化、数据整合
上手	Scala、Java、Python 支持交互式命令模式	Java编写，需要学习语法 有一些工具简化
兼容性	Spark兼容Hadoop数据源	/
容错	内存失效，需要手动设置checkpoint	硬盘存储静态数据 失败时从出错点自动重启执行
数据处理	实时数据处理或批处理，迭代任务	批处理

Robot

1 什么是Robot

- A physically-embodied, artificially intelligent device with sensing and actuation.
- Robot Institute of America:

  Robot is a **reprogrammable, multifunctional manipulator** designed to move material, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks.

- A robot is a mechanical apparatus designed to do the work of a man. Its components are usually electromechanical and are guided by a computer program or electronic circuitry.

1.1 Uses ad Advantages f Robots

- Used in vehicles and car factories
- Mounting circuits on electronic devise e.g. mobile phones
- Working where there might be danger e.g. nuclear leaks and bomb disposal
- Surgeons are performing robotic surgeries to avoid jiggles and movement in microscopically aided surgery or brain surgery
- Mail delivery to various mail stations throughout the building in large corporations
- Toy robots are a good source of entertaining for the kids e.g. dancing and talking robots
- Robots do not get bored or tired and they can work 24/7 without salary and food

1.2 Disadvantages of Robots

- It needs a high supply of power
- People can lose jobs in factories
- It needs maintenance to keep it running
- It cost a lot of money to make or buy a robot as they are very expensive
- A robot can not respond in time of danger as human can

1.3 Essential Characteristics of robots

- **Sensing**: The robot should be able to sense its surroundings and that is only possible with the help of sensors.
- **Types of sensors**: light sensors (eye) , touch sensors(hands) , hearing sensors(ears) or chemical sensors(nose)
- **Movement**: A robot needs to be able to move around its environment whether by rolling on wheels , walking , snaking or skating.
- **Energy**: A robot needs to be able to power itself which depends upon its power resources e.g. batteries , power generators or fuel.
- **Intelligence**: A robot needs to be intelligent and smart which is only possible by the programmer person.

And the workspace of robot is physically outside the device.

2 Robot 种类

- **Mobile Robots**: They are able to move around in their environment and not fixed to one physical location.
- **Industrial Robots**: They are used in industrial manufacturing environment e.g. welding , material handling , painting and others.
- **Domestic Or Household Robots**: Robots used at home such as robotic vacuum cleaner , robotic pool cleaner and sweeper.
- **Medical Robots**: Robots used in medicine and medical institutions e.g. surgery robots
- **Service Robots**: Robots that donot fall into other types by usage e.g. robots used for research.
- **Military Robots**: they are used in military e.g. bomb disposal robot , different transportation robots and reconnaissance drones

Fixed Robot | Mobile Robots (Wheeled robot / Legged robots / Swimming robots / Flying robots) | Collaborative Robot

3 ROS 基础概要

3.1 ROS解决什么问题

ROS = Robot Opearting System

ROS作为一个开源的软件系统，宗旨是构建一个能够整合不同研究成果，实现算法发布、代码重用、的通用机器人软件平台。

3.2 ROS是什么

- The Robot Operating System (ROS) is **a flexible framework for writing robot software**. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.
- ROS提供了类似操作系统的中间件。很多开源的运动规划、定位导航、仿真、感知等软件功能包使得这一平 台的功能变得更加丰富，发展更加迅速。到目前为止，ROS在机器人的感知、物体识别、脸部识别、姿势识别、 运动、运动理解、结构与运动、立体视觉、控制、规划等多个领域都有相关应用。

3.3 ROS的特点

1. 分布式通信架构

   一个使用ROS的系统包括一系列进程，这些进程存在于多个不同的主机并且在运行过程中通过端对端的拓扑 结构进行联系。虽然基于中心服务器的那些软件框架也可以实现多进程和多主机的优势，但是在这些框架中， 当各电脑通过不同的网络进行连接时，中心数据服务器就会发生问题。

   ROS将每个工作进程都看作是一个节点，使用节点管理器进行统一管理。并供了一套消息传递机制。可以 分散由计算机视觉和语音识别等功能带来的实时计算压力，能够适应多机器人遇到的挑战。

2. 多语言支持

   由于所有节点的通信都是通过网络套字节来实现的，这意味着只要能够供套字节的接口，节点程序就可 以使用任何编程语言来实现。ROS不依赖任何特定的编程语言，现在已经支持多种不同的语言实现，例如C++、 Python、Octave和LISP，也包含其他语言的多种接口实现。ROS现在支持许多种不同的语言，例如C++、Python、 Java、Octave和LISP，也包含其他语言的多种接口实现。ROS采用了一种独立于编程语言的接口定义语言 (IDL)，并实现了多种编程语言对IDL的封装，使得各个不同编程语言编写的“节点”程序也能够透明的进行 消息传递。

3. 丰富的工具包

   为了管理复杂的ROS软件框架，通过大量的小工具去编译和运行多种多样的ROS组件，从而设计成了内核， 而不是构建一个庞大的开发和运行环境。 这些工具担任了各种各样的任务，例如，组织源代码的结构，获取 和设置配置参数，形象化端对端的拓扑连接，测量频带使用宽度，生动的描绘信息数据，自动生成文档等等。

4. 丰富的应用包

   ROS提供了很多开源的运动规划、定位导航、仿真、感知等软件功能包使得这一平台的功能变得更加丰富， 发展更加迅速。到目前为止，ROS在机器人的感知、物体识别、脸部识别、姿势识别、运动、运动理解、结构 与运动、立体视觉、控制、规划等多个领域都有相关应用。

5. 强大的生态系统

   ROS以分布式的关系遵循这BSD许可，也就是说允许各种商业和非商业的工程进行免费开发，这也是ROS得 到广泛认可的原因之一。得益于开源社区的大力支持，ROS系统才得以快速发展。开源社区主要供:发行版(Distribution)、 软件库(Repository)、ROS维基(ROSWiki)、右键列表(MAiling list)、ROS问答(ROS Answer)、博客 (Blog)--www.ros.org/news。

Large Scale Visual Search

1 CBIR: Content-based Image Retrieval

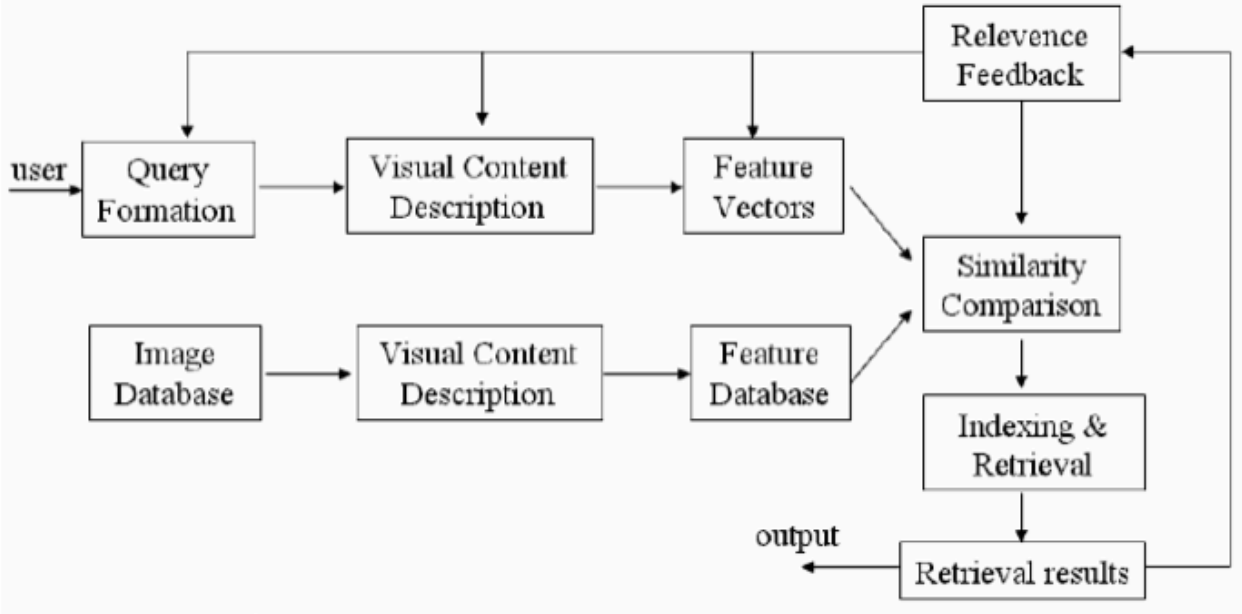


Figure 1-1. Diagram for content-based image retrieval system

1.1 Process

- User wants to search for, say, many rose images
    - He submits an existing picture as query.
- The system will extract image features for this query.
- It will compare these features with that of other images in a database.
- Relevant results will be displayed to the user.