

Shell 其实就是一种脚本语言，也是一个可以用来连接内核和用户的软件，我们编写完源码后不用编译，直接运行源码即可。

bash, a general purpose scripting language.

command [arguments]

interactive and batch / 交互式和批处理

Gnuplot (在Linux下基于Qt的一套绘图工具)

https://blog.csdn.net/weixin\_42014622/article/details/82962634

LaTeX（LTEX，音译“拉泰赫”）是一种基于ΤΕΧ的排版系统

https://blog.csdn.net/pipisorry/article/details/54571521

关于shell 和 bash 之间的关系

<https://blog.csdn.net/weixin_42432281/article/details/88392219>

|  |  |  |
| --- | --- | --- |
| Platform | Linux | Windows |
| 字符界面模式 | shell | Command Prompt |
| 命令行 | bash | command |
| 批处理模式 | batch mode | 批处理 bat |

commands: aliases, functions, built-ins, keywords, and executables.

Text, letter

Description automatically generated

**Here document** 的使用总结

https://blog.csdn.net/liumiaocn/article/details/86715953

Sudo apt install / upgrade/update \*\*\*\*

Sudo apt-get install \*\*\*\*\*

确定型随机数是一个伪随机数，因为给定一相相同的种子seed，它就会生成相同的随机数，所以这种随机是假的，称为伪随机数。

如果想得到不同的随机序列，需要确保每次使用不同的种子，生成不同的序列。

Mersenne Twister算法译为马特赛特旋转演算法，是伪随机数发生器之一，其主要作用是生成伪随机数。

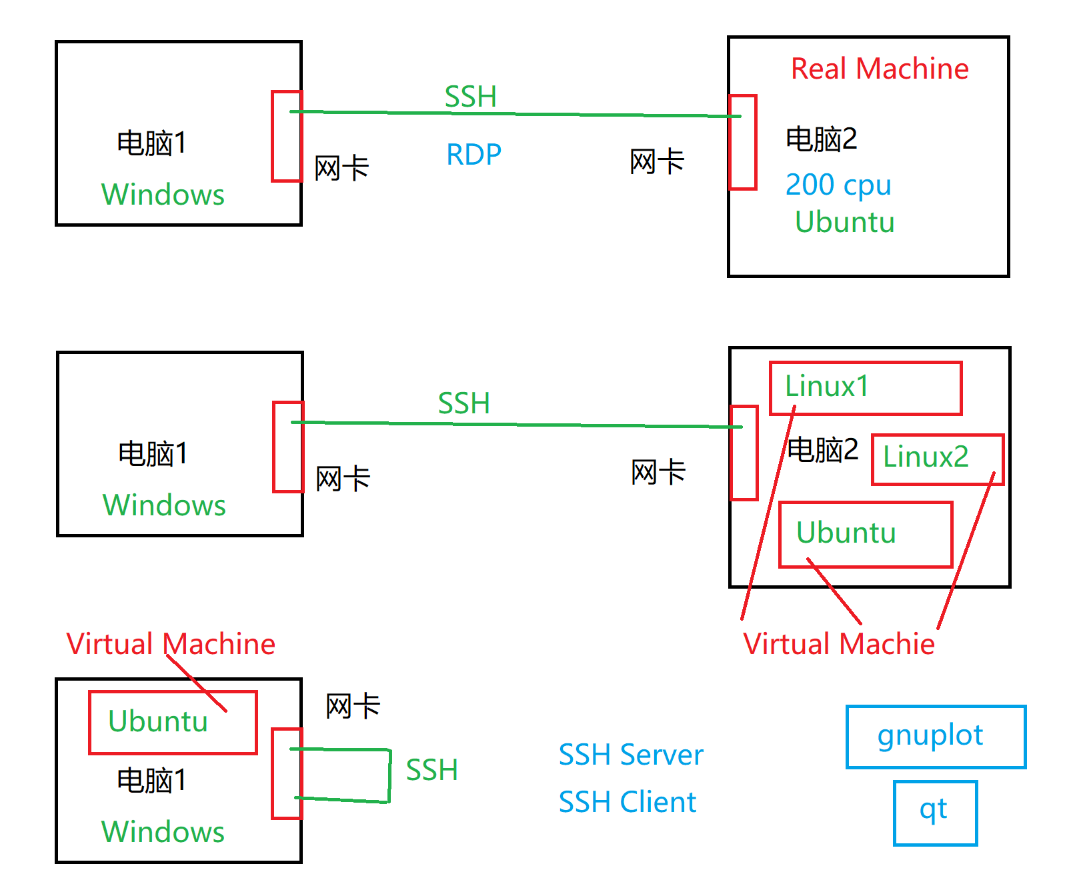
由于bash是不支持浮点型的计算，bash利用expr只能计算整数的基本运算。目前如果想要在bash中计算浮点型，只能通过bc或者awk来帮忙。

Echo $(expr $n1 + $n2)

Echo $(( $n1 + $n2 ))

Touch 创建文件

Vim/vi 文件，也可以创建文件



关于makefile和make:

<https://blog.csdn.net/ZBraveHeart/article/details/123187908>

Makefile文件中的命令有一定规范，一旦该文件编写好以后，在Linux命令行中执行一条make命令即可自动编译整个工程。

Text

Description automatically generated

它表示，为了达成目标，必须1，先有依赖，2，顺序执行下面的命令

Makefile并不会关心命令是如何执行的，仅仅只是会去执行所有定义的命令，和我们平时直接输入命令行是一样的效果。

1、目标即要生成的文件。如果目标文件的更新时间晚于依赖文件更新时间，则说明依赖文件没有改动，目标文件不需要重新编译。否则会进行重新编译并更新目标文件。

2、默认情况下Makefile的第一个目标为终极目标。

3、依赖：即目标文件由哪些文件生成。

4、命令：即通过执行命令由依赖文件生成目标文件。注意每条命令之前必须有一个tab保持缩进，这是语法要求（会有一些编辑工具默认tab为4个空格，会造成Makefile语法错误）。

5、all：Makefile文件默认只生成第一个目标文件即完成编译，但是我们可以通过all 指定所需要生成的目标文件。

**makefile中wildcard的理解**

<https://blog.csdn.net/dianqicyuyan/article/details/123509400>

wildcard 用来明确表示通配符。因为在 Makefile 里，变量实质上就是 C/C++ 中的宏，也就是说，如果一个表达式如 objs = \*.o ，则 objs 的值就是 \*.o ，而不是表示所有的 .o 文件。若果要使用通配符，那么就要使用 wildcard 来声明 \* 这个符号，使 \* 符号具有通配符的功能。

gnu make工具的[Makefile](https://so.csdn.net/so/search?q=Makefile&spm=1001.2101.3001.7020" \t "_blank)中$(SRC:%.c=%.o)的含义：**将SRC变量中所有以.c结尾的文件名替换成对应的以.o结尾的文件名，然后赋回给SRC。**

[makefile](https://so.csdn.net/so/search?q=makefile&spm=1001.2101.3001.7020)**中.PHONY的作用是什么？**

<https://blog.csdn.net/weixin_41969690/article/details/106727385>

.PHONY all clean起到“欺骗”的作用，欺骗make别管别的，让你干嘛你就干嘛的意思。

缺省情况下，makefile中的target是一个文件，上面的定义就是告诉make命令，后面的target不是文件，而只是一个label而已

Makefile有三个非常有用的变量。分别是$@，$^，$<代表的意义分别是：

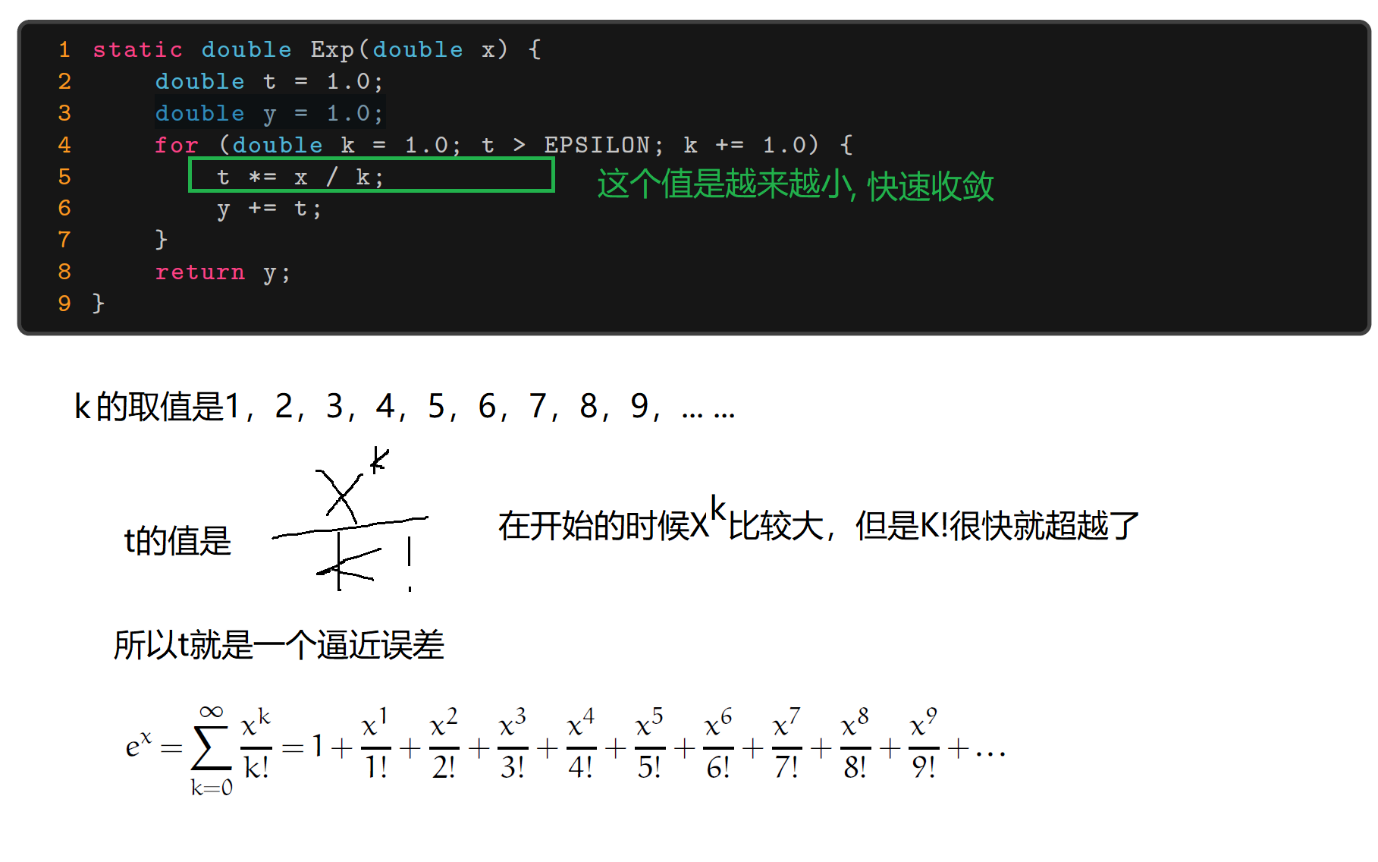
$@--目标文件，$^--所有的依赖文件，$<--第一个依赖文件。

mathlib.o : mathlib.c mathlib-test.c …

$< = mathlib.c

$@ = mathlib.o

$^ = mathlib.c mathlib-test.c …



关于LateX 排版系统

<https://blog.csdn.net/weixin_35805593/article/details/113454369>

documentclass{class} 为配置环境，可大致理解为C++中的 using namespace std;

usepackage{}为宏包调用，可大致理解为 #include<>等库函数的调用

正文区，以begin{document}打头，且有且只能有一个。这一点又与C++中的main函数如出一辙

latex运用%来做注释

主要用latex来输出数学公式

数学公式的输出有两种形式，一种为行内输出一种为行间输出。

分别用$ $ 与 $$ $$来表示。

latex的一个重大特点是“所见不是所得”，因此我们在命令行中另起一行，生成的pdf文件并不会换行，latex的换行指令为空一行另起。

{} 表示作为一个整体来统一处理

\[ \] 表示另起一行的公式，相当于 $$ $$

c\_m 2^m 表示下标和上标

软件人员必须了解的几个硬件知识

中央处理器CPU，内存Memory，硬盘Hard Disk

C/C++语言必须了解的几个底层知识

内存空间，地址，指针，数组，数组长度/大小

&a[0] = a &a[1] = a+1, 同一个数组内才能这样操作

位数：32位，可访问地址空间为2^32 = 4G，64位可访问地址空间为2^64，

18,446,744,073,709,551,616，理论上很大的一个值，但实际没有用到，足够程序运行

数据结构相关的几个知识

数组Array（可变数组），链表LinkedList（单向链表/双向链表），队列Queue，栈Stack，堆Heap，二叉树 Binary Tree，内存

堆Heap就是一棵完全二叉树，同时满足最大堆或最小堆的

对堆栈的误解

默认情况下，在32 位上运行的线程的最大栈大小为256 KB，64 位上的最大栈大小为512 KB。可以通过系统函数进行修改。

关于程序设计的几点提示：（以asgn2为例子）

1. 读懂需求 （主要是老师提供的asgn2.pdf，特别是Your task部分，一定要读懂每一句的意思）
2. 列出文件，就是在Deliverables节中的内容

Mathlib.h 函数的申明，不能修改

Mathlib.c 在mathlib.h中的函数的实现

Mathlib-test.c main函数必须在这个文件，解析命令行，定义OPTIIONS

Makefile

README.md

DESIGN.pdf

WRITEUP.pdf

1. 根据文件的功能，先实现框架，保证能运行，可测试

比较明确的细节可以先用注释标出

4，再细化每一个文件的功能

细化的原则也是先简单，后细节

5，对复杂的问题进行拆解，分成多个函数，每个函数也是先框架，再细节

6，每细化一个函数，测试一个函数

测试时可以通过输出，断点查看

7，完成步骤2中的所有文件

有些工具必须先准备好，“工欲善其事，必先利其器”，“事半功倍”

比如Visual Studio，Windows和Ubuntu上文件的共享，Ubuntu上的可视化编辑器，等等

VS 2022 Professional key

共享文件夹，修改Ubuntu屏幕大小

<https://blog.csdn.net/Inochigohan/article/details/119791518>

作为CS专业，要善于利用网络资源，比如一些教学视频，在线的小工具

概率论和数理统计

<https://www.bilibili.com/video/BV1Pb411771f/?p=9&spm_id_from=pageDriver&vd_source=342115c1d2b02981c05a987047b37614>

求补码计算器

<http://www.99cankao.com/numbers/twos-complement.php>

二进制计算器

<https://miniwebtool.com/zh-cn/binary-calculator/?number1=0100&operate=1&number2=1010>

理论上的内容，最后用小程序进行验证等

比如Quiz上的题目

关于函数的调用

1，函数的调用可以分为三类：自己的函数调用，系统库函数调用，以及第三方库函数调用

但是本质上是一样的，都包括

a, 函数的申明(declare)，说明函数的名称，参数（包括数量和类型），返回值（类型）

b函数的实现(implement)，就是函数的功能

通常情况下，函数的申明放在.h文件中，函数的实现一般放在.c文件中，而调用者必须include 相应的.h文件，.c文件不用include，但是.c文件必须一起在一个工程里进行link

比如：

rsa.h 和 rsa.c randstate.h 和 randstate.c numtheory.h 和 numtheory.c

如果是自己的函数调用，而且自己定义的函数功能比较单一，只为某个文件中的某个特定功能来实现，可以不用.h文件，而直接和调用者放在同一个.c文件中

比如：

void usage(char\* exec)

void output\_mpz\_t(char\* pre, mpz\_t op)

但其必须放在调用者的前面，这时候，函数的实现，同时也带有函数的申明功能

如果不想把函数的实现放在调用者的前面的话，就必须在调用者之前单独进行函数的申明

【举例说明】

void usage(char\* exec);

… …(调用者代码)

void usage(char\* exec) ｛… …｝

2，对于系统库和第三方库，不可能提供.c文件给我们，但是他们一定会提供.h给我们，否则，我们就不知道该如何调用函数，包括函数名，函数参数类型，参数个数，返回值类型等，所以一定会提供.h文件

比如：

gmp.h (第三方库) stdio.h （系统库）

虽然系统库和第三方库不能提供.c文件，但是它们一定会提供由.c所生成的库文件给我们，否则我们就只能通过compile（编译过程，只需要.h文件，就是函数的申明），无法通过link（链接过程，必须需要函数的实现，要么是.c文件，要么是库文件）

|  |  |  |  |
| --- | --- | --- | --- |
|  | 函数的申明 | 函数的实现 |  |
| 自己定义的函数 | .h | .c | .c加入工程中 |
| 第三方库函数 | .h | .c生成的库文件 | 库文件加入工程中 |
| 系统库函数 | .h | .c生成的库文件 | 不用我们加，link会自动加 |

要明白程序生成有二个过程： (不要跟编辑edit搞混了)

一个是编译compile .c -> .a / .c -> .obj

一个是链接link .a -> .o / .obj -> .exe

此外，还有一个是程序的运行过程

关于库的使用（静态库和动态库）

前面说了，函数的实现是放在.c文件中，实现函数的具体功能，

比如

int mpz\_set\_str (mpz\_ptr, const char \*, int);

如果我不想让别人知道我的具体实现，或者担心别人修改了我的实现代码，导致调用者出错，我们一般不愿意直接把.c文件提供给别人，但是如果你不提供函数的实现，别人就无法link，以至无法run。

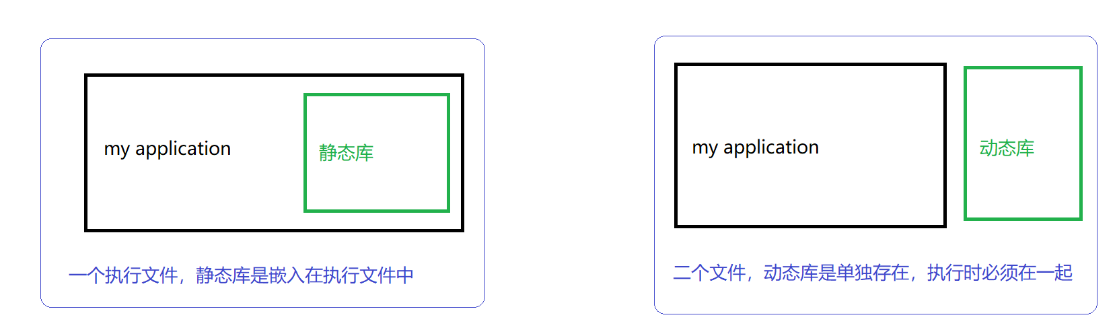
几乎所有的OS都提供了一种机制，保证你不提供源代码，只提供一种叫库的文件（别人无法看到具体的实现，当然也无法进行修改）

函数库可分为二类：

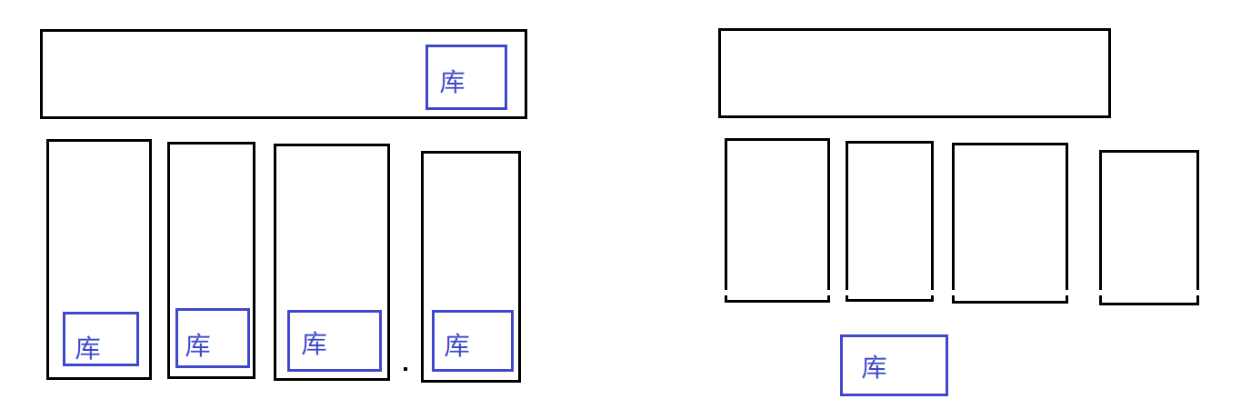
静态库 static library

动态库 dynamic library

虽然它们的存在形式不一样，但它们都是由.c文件编译而成，是包含了函数的具体实现



|  |  |  |
| --- | --- | --- |
|  | 好处 | 坏处 |
| 静态库 | 只有一个可执行文件 | 在多模块情况下，有重复代码，浪费空间 |
| 动态库 | 只有一份代码，节省空间 | 有多个文件，少了任一个都不能执行 |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数申明 | 函数实现 | 何时调用 | 其它辅助文件 |
| 静态库 | .h | .lib (windows)  .a (unix类) | 在link时 | 无 |
| 动态库 | .h | .dll (windows)  .o (unix类) | 在run时 | 一个小的.lib文件 |

对于静态库，只需要一个包含了所有函数实现代码的.lib文件，.lib文件是比较大的（因为包含了所有的实现代码）

对于动态库，.dll只是在运行时才用得上，那么在link时，它根本不知道函数有没有实现，（注意.h文件只是一个申明，也许是一个假申明），所以它也需要一个.lib文件表明它实现了哪些函数（只是表明实现了，不需要具体的实现代码，具体的实现是在.dll中，所以.lib是很小的）

一个具体的例子，就是我们在assignment5中所用到的第三方库gmp

在windows平台上，我们除了提供gmp.h – 函数的申明

还提供了libgmp-10.dll – 动态库，函数的具体实现

此外，还要提供一个小的配合动态库的libgmp-10.lib，给调用者说明动态库中具体实现了哪些函数

（注意，为了区分真正的静态库，原则上我们不把这个小的.lib称为静态库，只是称为导出函数库，虽然它的后缀也是.lib）

关于gmp库的build

在数论当中，我们要用到大数（其长度远超过当前计算机的64位），gmp就是专门处理大数的一个专用第三方库，它是开源代码的，每个人都可以从它的网站上下到源代码，但是它不提供各个平台上的具体的库，所以就需要我们自己进行build

下面就讨论在Windows上的build（因为Unix和Linux通过package管理工具能够自动下载到）

|  |  |  |
| --- | --- | --- |
| 平台 | gmp |  |
| Unix-like | 包安装 | apt install |
| windows | 1，下载别人build好的 | 简单，快捷 |
| 2，自己build | 麻烦，但是会学到许多知识 |

gmp提供了在Unix-like (类Unix系统，包括Linux，Ubuntu都属于此类)的build，如果想在Windows上进行build，必须安装[MinGW – (Minimalist GNU for Windows)](https://www.baidu.com/link?url=P2K3dd2HS9j2GrIYawjFDKYn0mItXf-_Bt_MyWtPmwIz-Q-WiSCFmal0IPLH2ji46RG3Myrtdp9irrPpn2qsva&wd=&eqid=b8fd954e0011b0c6000000066367ad1c" \t "_blank)

此外，还需要安装msys MSYS是一个小型的GNU环境,包括基本的bash,make等等

二者之间的关系见下面link:

https://blog.csdn.net/u012294613/article/details/126460773

事实上，在现在的许多MinGW的安装包中已经包含了msys，对于这类的MinGW，我们就不需要再重复安装msys了，直接可以用msys

win10+编译gmp大数库（亲测可用）

<https://blog.csdn.net/qq_45662588/article/details/122558604>

windows环境下的gmp大数运算库的配置（visual studio 2019 + gmp-6.2.0 + msys

https://blog.csdn.net/qq\_45662588/article/details/122560098

**关于C语言程序的几点警示：**

1. C语言和其它语言最大的区别是指针

指针，本质上就是指向某一块内存，用这一块内存的“地址”来表示指针的“值”

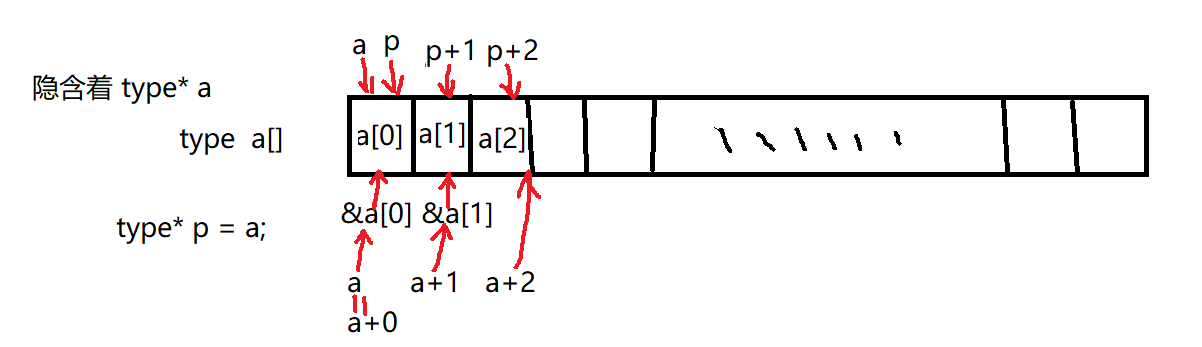
A picture containing text

Description automatically generated

int a = 1234

int\* p = &a (then, p = 0x052C0948)

数组，本质上就是系统中一块连续的内存，所以指针和数组在某些时候是可以变通和互换的



**既然指针就是地址，那么地址一定要是有效的才能访问，访问无效物地址一定要crash（程序崩溃）**

怎么保证一个指针是有效的？

A, 所有的指针最好在申明时进行初始化（一个有效值或者NULL），或者在申明时就进行赋值（其实赋值也是一种初始化）

B，所有的指针如果不再使用时，最好置成NULL

C，指针在使用时，一定要进行为NULL判断，即 if(p)

2，（不只是针对C，针对所有的程序语言）

要想一段程序能稳定，正确地运行，除了需要编程者深厚的技术知识和良好的代码规范外，最重要的是时间，任何人写的程序都不可能第一版就涵盖所有的情况，因为user case是千变万化的，有硬件的，有环境的，有网络的，等等，**所以调试一定是必不可少的，而调试就意味着时间。**

如果一段程序存在了很长时间，也许它的代码风格很乱，但它绝对是稳定的，因为它踩过了所有的坑，经历了所有情况的检验。

**所以程序一定要留出充足的时间**

我有一个feature，代码已经写好了快一个月了，但一直没有进产品线，因为每次进出，在线上一测试就会发现新的问题，需要调试和修改，为了不影响其它人的进度，就只能rollback

等前面发现的问题改好，再进产品线，又会碰上新的情况，又要rollback

不是说我的代码质量不好，风格不好，是很多用户情况你是不可预知的，一定是到了产品线才有可能发生，而在自己的环境或者公司内部环境，永远不会出现这些情况

关于文件的换行（就是另起一行）的问题



这是ASCII表，注意加红框的两个代码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **十进制** | **十六进制** | **代码** | **字符解释** | **转义符** |
| 10 | 0A | LF | 换行/新行 | \n |
| 13 | 0D | CR | 回车 | \r |

由于计算机的键盘是由电传打字机发展过来的，在传统的电传打字机中，回车表示回到起始位置，换行表示到下一行，所以需要回车和换行二个符号来表示

Windows系统继承了这种风格，所以在Windows系统上生成的文本文件，最后都是CRLF （\r\n），哪怕你在输出文件时，**只用\n，系统还是会在最后给你补上\r\n**，但是如果你去读的时候，**系统又会把\r\n转换成一个\n返回给你！！！**

A picture containing chart

Description automatically generated

但是Unix/Linux系统只用换行一个符号来表示，所以结尾是LF (\n)

Text

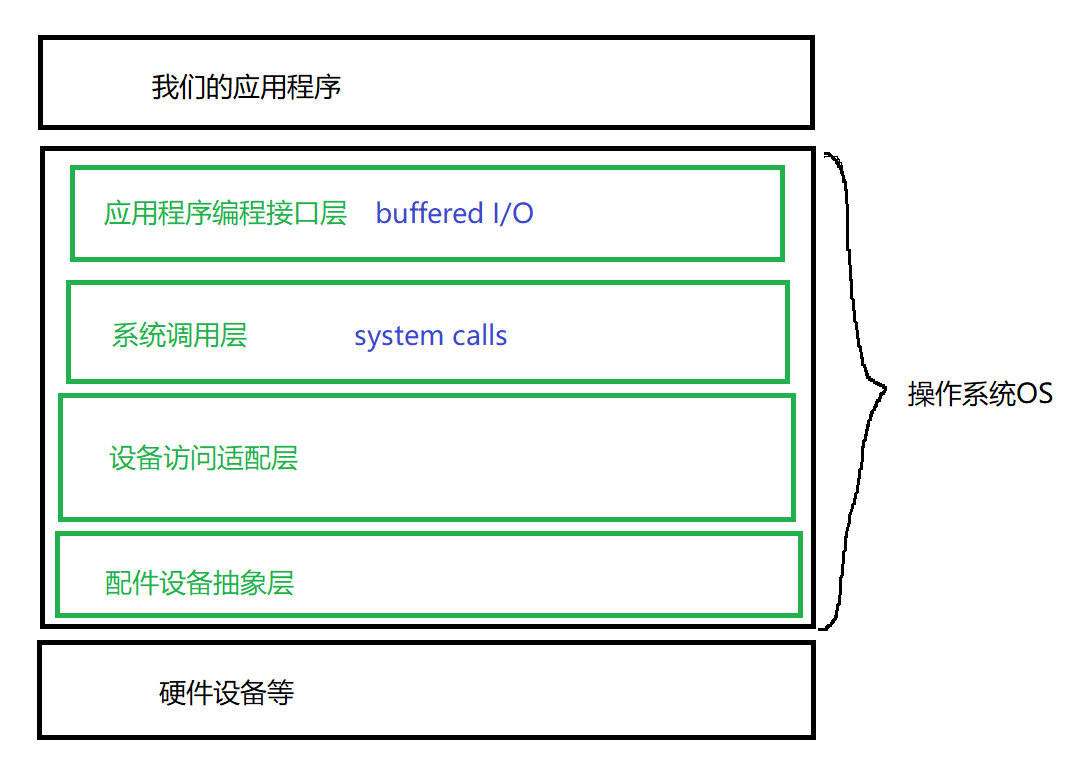
Description automatically generated

所以如果在Windows系统下生成的文件，如果copy到Linux系统后，除了要处理最后的换行符\n，还要处理换行符之前的回车符\r

让人迷惑的，如果在Linux系统下生成的文件，如果copy到Windows系统后，虽然最后只有一个换行符\n，它仍然能正常显示

Asgn7明确提出对文件IO的具体要求，

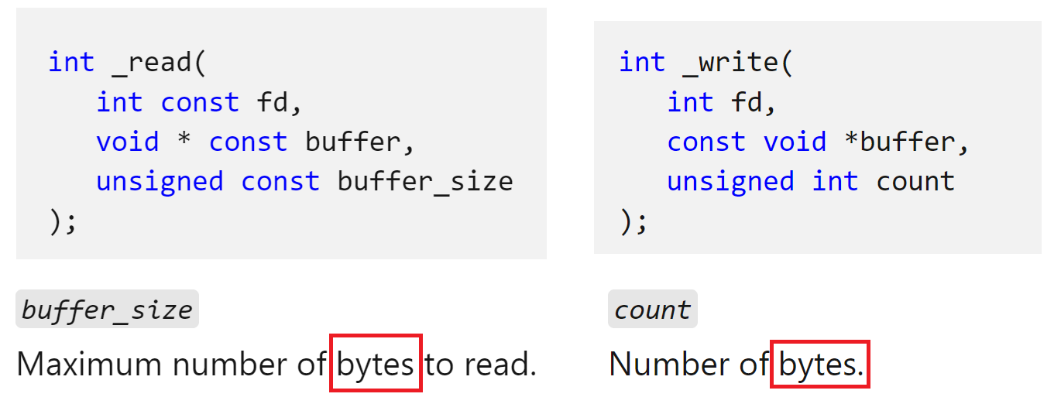
即：不要使用buffered I/O函数（这些函数来自<stdio.h>头文件），而必须使用更底层的system calls(syscalls)，下面对这两者的不同进行说明



下面只针对作业要求的IO部分进行比较

|  |  |  |
| --- | --- | --- |
|  | Buffered IO | System calls |
| 打开文件 | fopen | open |
| 关闭文件 | fclose | close |
| 从文件中读数据 | fread fgets fgetc | read |
| 写数据到文件 | fwrite fguts | write |
| 所在的头文件 | stdio.h | unistd.h / fcntl.h |
| 文件描述/文件句柄 | FILE\* 结构体指针 | int 整型数 |
|  |  |  |
| 标准输入 | stdin | 0 |
| 标准输出 | stdout | 1 |
| 标准错误输出 | stderr | 2 |
|  |  |  |

所以本次作业中，不能跟上次作业一样使用stdin / stdout，而是要使用 0 / 1 来表示标准输入和标准输出



记住以下几点：

1, 所以无论是read，还是write，都是按字节(byte)来写，而且最小单位，必须是1个字节（8 bits）

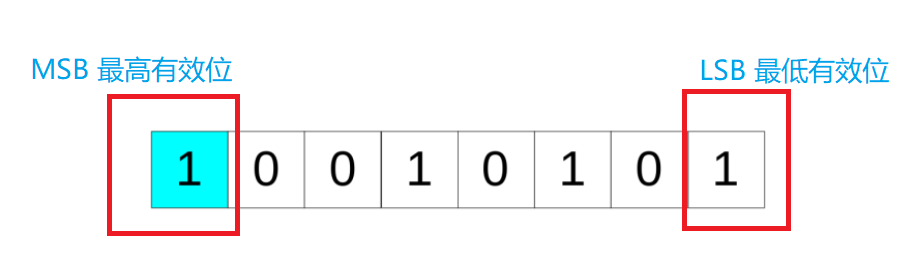
2, 所以没有按位(bit)进行read和write，所有的位都必须组合成字节(byte)

3, 对于bit，如何组成byte，需要了解一下LSB，MSB，以及大端Big-endian和小端Little-endian

*LSB*（Least Significant Bit），意为最低有效位，即在位数最右边的位

*MSB*（Most Significant Bit），意为最高有效位，即在位数最左边的位，对于有符号数来说，如果MSB为1，就表示是负数，为0，就表示是正数

LSB和MSB是以一个字节内的二进制机器码为单位进行排序。



如果以一个字节(byte = 8 bits)来举例的话，LSB就是bit 0，MSB就是bit 7

*大端（Big-endian）：数据的高位字节存放在地址的低端 低位字节存放在地址高端；*

*小端（Little-endian）：数据的高位字节存放在地址的高端 低位字节存放在地址低端；*

需要注意的是：大小端是以字节为单位，对所有字节进行数据储存的方式。

*uint32\_t* temp = 0x12345678;

比如以上面数值为例：

小端表示为： 78 56 34 12

大端表示为： 12 34 56 78

地址低端 🡪 地址高端

*A picture containing logo

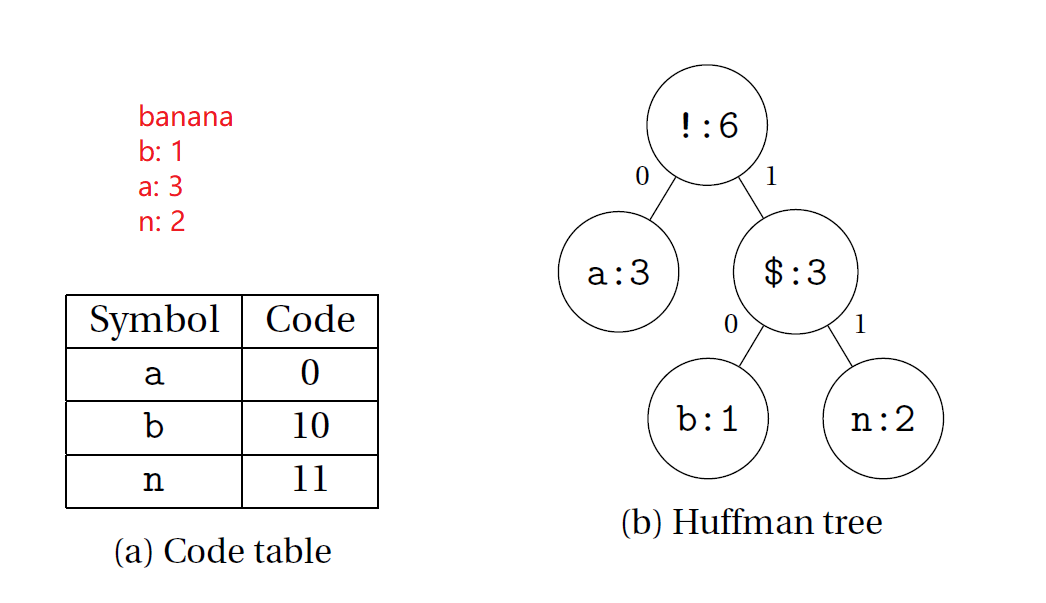
Description automatically generated*

另外还要明白：大小端的方式是由CPU硬件来决定的，不是由OS和软件来决定的

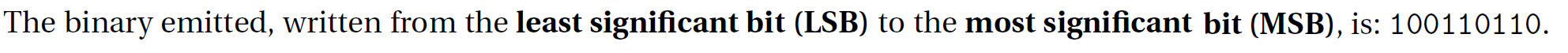
A picture containing application

Description automatically generated

关于asgn中code编码，就要用到LSB / MSB 相关的知识，同时还要用到bits的拼接知识



也就是说a用1 bit (0)，b用2 bits (10)，n用2 bits (11)来编码，但是bit是不能写文件（前面已经讲过），如果直接把bit扩展成byte又是浪费，所以就要进行bits的拼接，即每8 bits拼接成1 byte，而拼接的时候就要考虑bit的顺序，是LSB，还是MSB



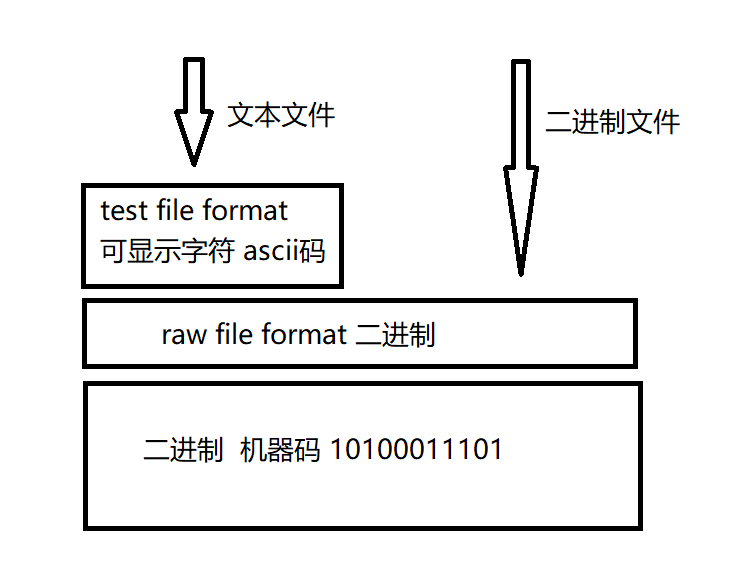
一共需要 3\*1+1\*2+2\*2 = 9 bits

转换成正常顺序则是 0,1101 1001

在写文件前，必须扩展成2 bytes 0000 0000,1101 1001 (00 D9)

写文件时，又是按小端显示，则为D9 00

关于文件的模式：

从图上可以看出，文件是有二种模式，一种是二进制数据来表述的文件，比较底层，另一种是由可显示的ascii码来表述的文件，这个就是我们所说的文本文件，比如.txt文件，文本文件由于内容是可显示的字符，所以比较直观可见，但事实上文本文件的底层依然是二进制的，只不过是我们人眼看不懂二进制的意思。

其实我们所接触的90%以上的文件是二进制文件，二进制文件一般要通过专门的软件来查看，比如.doc，.pdf等。

文件操作时缺省的模式：

不同的OS对文件缺省的模式有不同的定义，而且需要进行修改

比如在Windows下缺省的模式是\_O\_TEXT，即文本文件模式

Graphical user interface, text

Description automatically generated

为了不受系统缺省模式的影响，我们一般在调用open时，指定文件模式

*O\_BINARY or O\_TEXT*

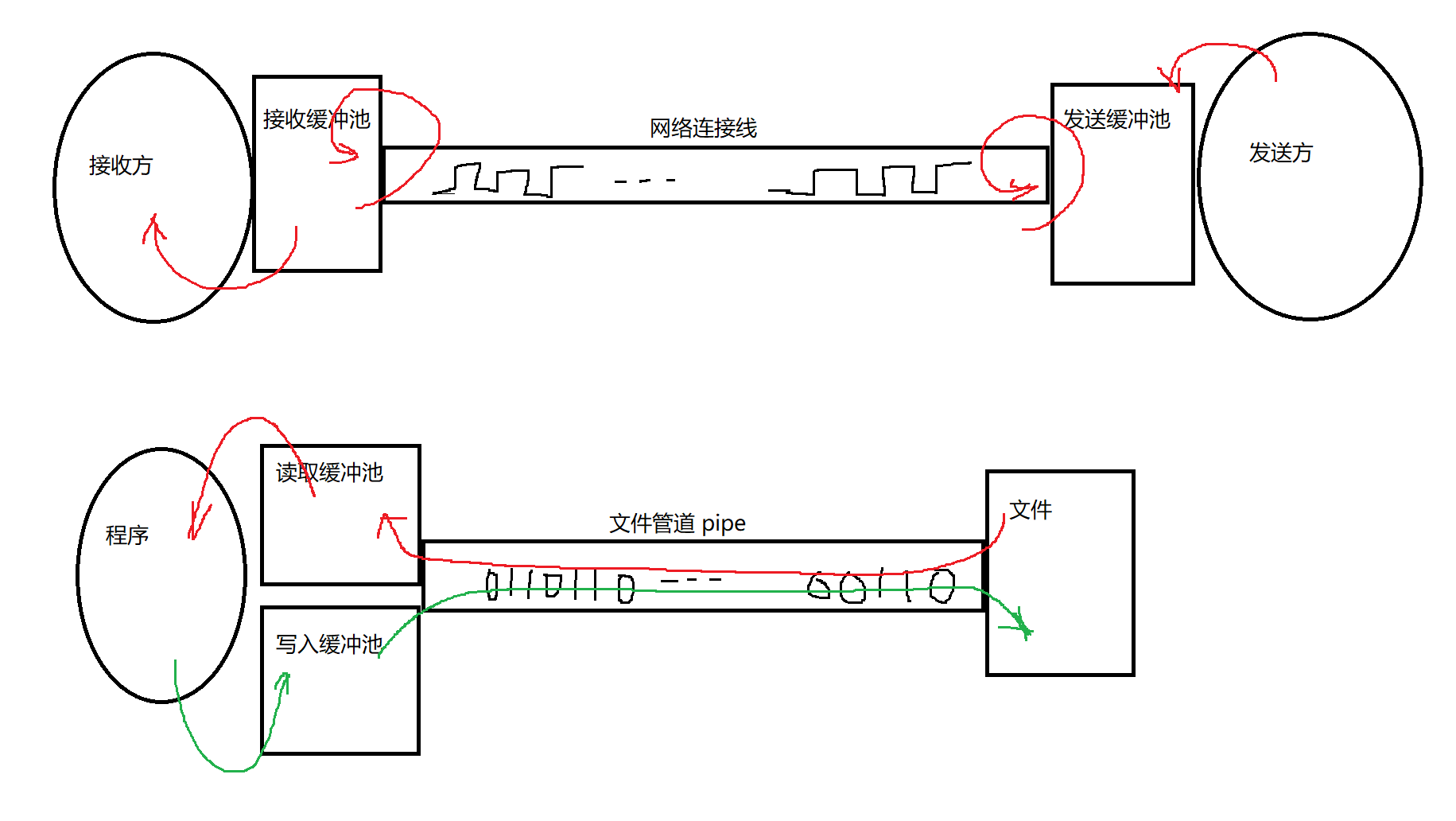
二进制文件查看工具 UtraEditor

文本文件比较工具 Beyond Compare

关于文件的读写：

文件的读写，非常类似于网络上数据的接收和发送

两者区别在于，网络上永远是一对（发送和接收，发送方是主动者），而文件的读写是单向（要么读，要么写，但无论是读，还是写，程序这一方永远是主动者），另一边只是一个被动存储数据的文件设备



由于文件的读取，不是直接操作文件（主要是性能上的考虑），而是从读取缓冲池中读取数据，就有可能当我要求读100个数据时，当时的缓冲池只有51个数据，那后续该怎么办？

1. 后面的数据我暂时不要了，等到下次读，先处理读取的51个数据
2. 循环读，再调用一次read函数，去读剩下的100-51=49个数据

这二种方式都有应用，具体情况根据程序逻辑而定

同理，对于文件的写入，由于是把数据写到一个写入缓冲池，也意味着写入缓冲池空余的空间不够当时写入数据所要求的空间，就会只有部分数据被写入的问题。

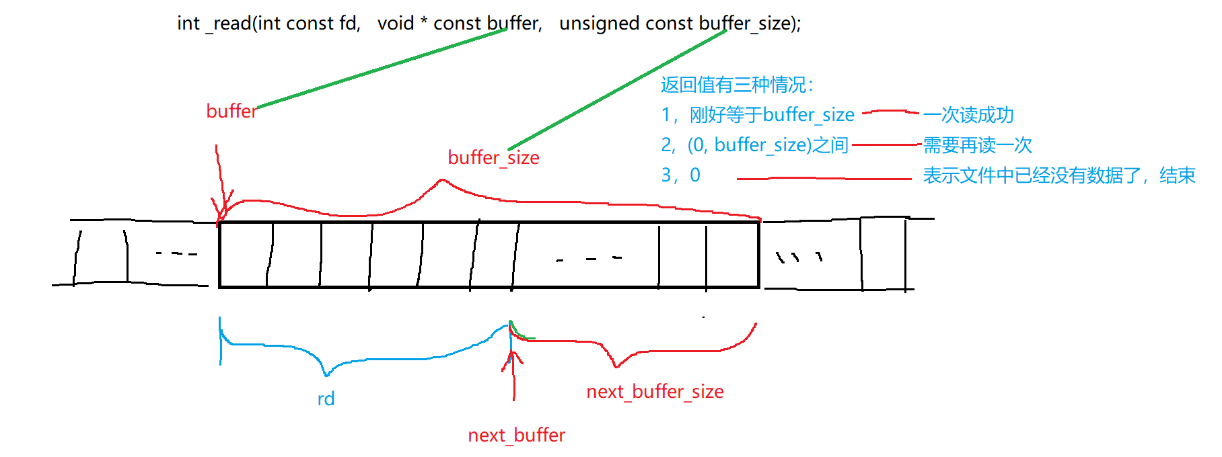
解决文件同上面文件的读取是一样的

相比文件的读取来说，文件的写入碰到缓冲区空间不够的概率要远远低于文件的读取，但是从程序设计的角度来看，哪怕再小的概率，我们必须进行处理

文件读取的最小单位就是byte，不可能是bit

Bit buffer，它去按byte从文件中读，然后程序自己从bit buffer取bit

文件的读取示意图：



文件的写入跟读取是一样的逻辑

讨论几个小的知识点：

1. 调试Debug 和 测试 Test

Debug和Test是完全不同的两个概念

* Test是模拟用户真实的实际环境，用真实的数据进行运行，对比运行后的结果，是否满足设计的要求

比如：代码上传后，后台进行的full test，它会根据各种情况进行test

再比如：我们在本地用真实的文件运行测试(gp11.txt)

Test的主要目的，就是看程序是否满足设计要求的各种功能，一般我们不看中间过程，而只看最后的结果

* Debug是用少量的、可以手工计算的小数据，对程序进行单步Step的跟踪，查看每一步运行结果是否正确

比如：我们自己造一个小数据 banana

Debug的前提条件是我们自己能根据这些小数据，手工推算出每一步的结果，然后我们对程序每一步的运行结果跟推算结果进行对比，一般我们先用Step over/F10，如果发现结果不对时，我们再使用Step in/F11

Debug的主要目的有二个：一是通过Step了解程序每一步的基本功能，二是看程序的某一步运行结果是否正确，功能是否符合设计要求

1. 用户User 和 用户组Group

现代的操作系统OS都是多用户系统（Unix/Linux从设计之初就是多用户系统，早期的Window是单用户系统，从Win95开始后，也是多用户系统了）

这就意味着同一台电脑上可以运行多个用户帐号（由于人机交互界面只能一个用户独享，所以带有用户界面的系统只能同时运行一个用户帐号，但是它的底层是支持多用户的，你可以在不同的时间运行不同的用户帐号。另外如果作为后端的服务程序，由于它只是以字符界面工作，没有交互界面，不存在屏幕，键盘，鼠标等设计的占用，所以是可以以多用户帐号同时运行的）

同时为了方便对用户User进行有效管理，在User之上，还是一个Group，所有的用户User必须属于某一个用户组Group。

比如 :

|  |  |  |
| --- | --- | --- |
| Group | group\_1 | group\_2 |
| User | Zhangsan, Lishi, bigdog | Wangwu, Xuliu, bigdog |

* 任何一个User必须属于一个Group
* 一个User可以属于多个Group
* 一个Group下面可以有多个User

由此可以扩展出三级层次结构：

User 自己，比如Zhangsan

User所在的Group，比如group\_1

除了group\_1外其它的所有组Other Group，比如group\_2

1. 文件访问权限 Permission

由于每一种操作系统OS的底层逻辑都不同，所以对文件访问根据的设计和控制也不相同，相比来说Unix/Linux比Windows考虑更全面更合理一些，所以下面我们是以Unix/Linux来为例进行说明（同时会提及Windows的特别之处）

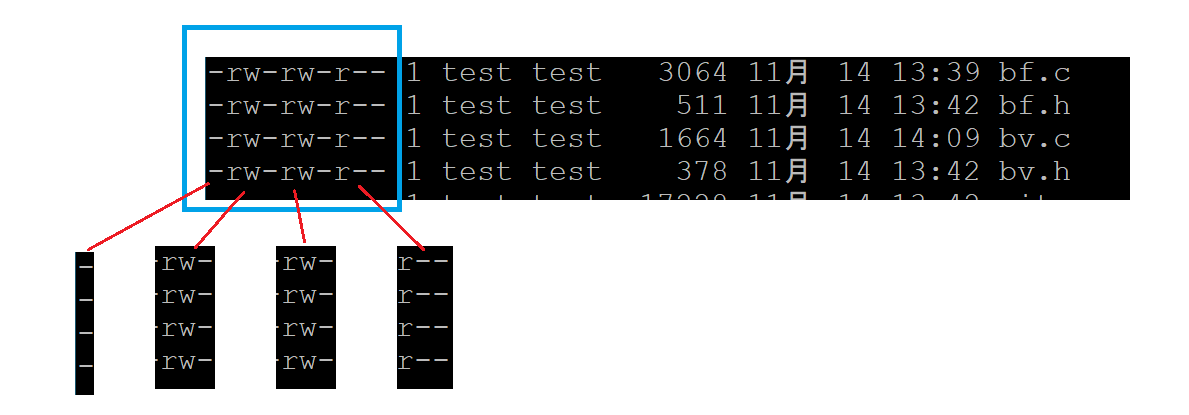
Linux文件有三种权限，分别是读权限Read，用r表示，写权限Write，用w表示，以及执行权限Execute，用x表示，由于这三种权限是可以进行组合的，所以需要用三个位Bit来表示 \*\*\* (421)

|  |  |  |  |
| --- | --- | --- | --- |
| 读权限Read | 可以查看文件内容 | r - 4 | 1\*\* 有此权限 / 0\*\* 无此权限 |
| 写权限Write | 可以编辑文件内容 | w - 2 | \*1\* 有此权限 / \*0\* 无此权限 |
| 执行权限Execute | 能够执行该文件 | x - 1 | \*\*1 有此权限 / \*\*0 无此权限 |

说明：对Windows系统来说，只有读写权限的控制，没有执行权限的控制，因为在Windows系统下，只有特殊格式的文件才能够执行，这种格式就叫PE(Portable Executable)文件结构格式，比如.exe文件

在了解了文件的三种权限后，再结合上面的知识点2，你会发现Unix/Linux下不只是控制用户User对文件的操作，还要控制用户组Group对文件的操作，所以就有了下面更复杂的控制逻辑：

(在Linux terminal下，用命令ls -l 或者 ll，可以查看文件权限)



第1列表示文件类型，其中d表示为目录，-表示为普通文件，l表示为软链接；

第2列rw-， 表示属主用户User拥有的权限为：可读 可写 不可执行；用数字表示是6

第3列rw-， 表示属主用户所在的组所拥有的权限：可读 可写 不可执行；用数字表示是6

第4列r--， 表示其它组所拥有的权限：可读 不可写 不可执行；用数字表示是4

所以上面权限也可以用数字表示为664

修改文件权限的bash命令就是chmod （对应的函数调用是fchmod，Windows上没有这个函数）

+ 表示新增

* 表示删除

= 表示设置

有时候我记不住这三组的先后顺序时，总是都统一设置上，

比如：chmod +777 check.sh

其实这是一种懒惰的做法。

# 日志分析查看——grep,sed,sort,awk运用

准备工具：sort, tail, less, uniqu,grep,sed,awk

tail -400f demo.log = tail -n 400 -f demo.log

#监控最后400行日志文件的变化

less demo.log

#查看日志文件，支持上下滚屏，查找功能

uniq -c demo.log

#标记该行重复的数量，不重复值为1

grep 简单使用

grep [选项]...模式 [文件]... （模式是正则表达式）

grep -o 'order-fix.curr\_id:' demo.log

#-o选项只提取order-fix.curr\_id:xxx的内容（而不是一整行

grep -c 'ERROR' demo.log

#输出文件demo.log中查找所有包行ERROR的行的数量

如果把grep比作过滤器，那sed就是个修改器了。

小结：sed就是遍历对于输入文件的每一行，如果该行匹配地址1，地址2的范围之内，那么就对这一行执行命令

3.对记录进行排序 sort

4.统计日志相关记录数 awk

awk简单使用：

awk [-v 变量名=变量值] [-Fre] [--] '模式 { 语句 }' 变量名=变量值 文件名

awk处理方式也与sed类似，对文件中的每一个输入行进行处理，每个处理首先判断是否是模式中匹配的行，是的话就具体执行相应的语句。

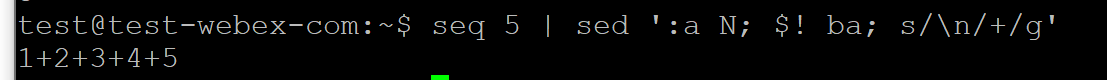
不同的是，awk侧重与对每一行的列进行处理，并且，awk脚本和c语言类似也拥有变量，条件判断，循环等复杂语句

sort、grep、sed、awk等命令

https://blog.csdn.net/Ananbei/article/details/80948309

seq 100 | sed ':a; N; $! ba; s/\n/+/g' | bc

seq 100 | sed ':a; N; ba; s/\n/+/g' | bc



Sed 和 awk 的功能比较接近

Sed 语法更简单一点，awk 语法相对来说复杂一点

Unix 文件系统，文件描述符（一个小的非负整数）

特别的，文件描述符0，1，2分别赋给stdin, stdout, stderr

Unix使用< 和 > 重定向和管道 | 改变输入/输出的IO

open 返回一个文件描述符

fopen 返回一个文件指针

Hash函数很重要

Table

Description automatically generated

Text

Description automatically generated with low confidence

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text

Description automatically generated with medium confidence

Timeline

Description automatically generated

ulimit -c unlimited

sudo sh -c ‘echo core > /proc/sys/kernel/core\_pattern'

sudo bash -c 'echo core > /proc/sys/kernel/core\_pattern'

gdb ./keygen -c core

bt

循环单链表

