**C/C++**

**·void \*memset(void \*ptr, int value, size\_t num);**

将内存块按字节设置为指定的值

ptr：指向要设置值的内存块的指针。

value：要设置的值，是一个无符号字符（unsigned char）。

num：要设置的字节数。

**·char \*strcpy(char \*dest, const char \*src);**

将一个字符串复制到另一个字符串中，包括字符串结束符 '\0'

**·char \*strncpy(char \*dest, const char \*src, size\_t n);**

将一个字符串前n个字符复制到另一个字符串中，不会追加‘\0’

**·size\_t strlen(const char \*str);**

返回字符串的长度（不包括结束符 '\0'）

**·int strcmp(const char \*str1, const char \*str2);**

返回值：str1<str2，返回负数

str1>str2，返回正数

str1==str2，返回0

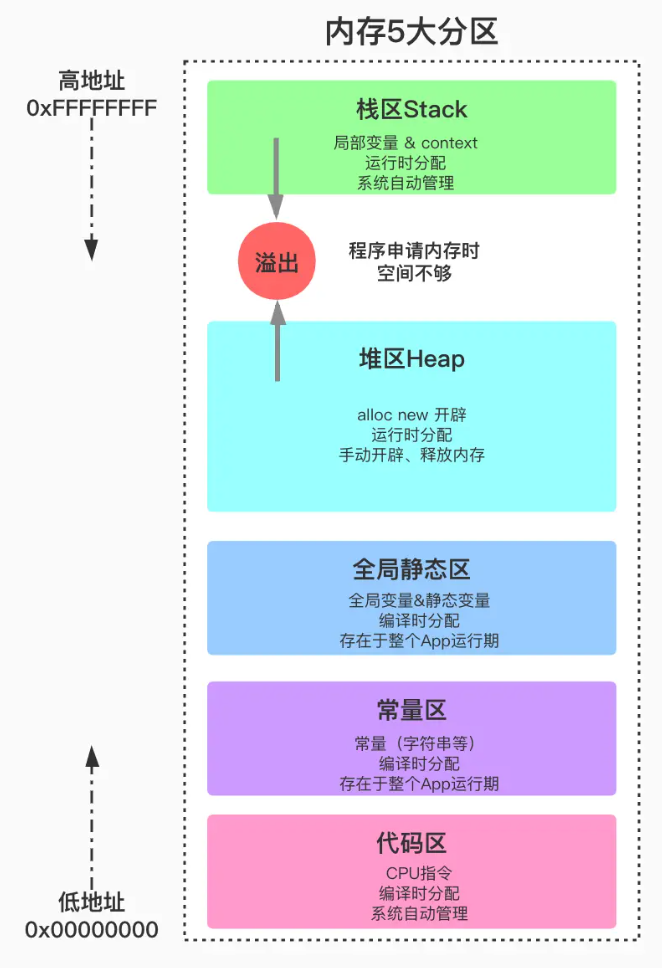
**·void \*malloc(size\_t size);**

**·void \*calloc(size\_t num, size\_t size);**

用于动态分配内存空间并将其初始化为零

**·void \*realloc(void \*ptr, size\_t size);**

用于重新分配之前通过malloc或calloc分配的内存空间的大小



**·void function(const string &str){ }** 为啥函数参数用**常引用**？

传递一个对象又不想改变这个对象的值，所以用const。又想避免不必要的复制操作，所以用引用 **&**

·**const T** 和 **T** 被视为不同的类型

**·**引用定义时必须初始化、初始化后不可更改、不可为NULL

|  |  |
| --- | --- |
| **ASCII值** | **字符** |
| **0** | **‘\0’** |
| **32** | **space** |
| **48** | **‘0’** |
| **65** | **‘A’** |
| **97** | **‘a’** |

**·内联函数**

本身是真正的函数，不需要函数调用的开销（压栈，跳转，返回）。

空间换时间

**inline void function(int x);**  //仅仅是申明函数，没有任何效果

**inline void function(int x)** //正确

**{**

**return ++x;**

**}**

一般适用于 函数规模小，不是递归且调用频s繁的函数才用inline修饰

内联只是给编译器的一个建议，编译器不一定接受

·**默认参数**

只要函数形参列表中某一个参数是默认参数，那么右面的都得是默认参数

**1、volatile**

防止编译器对变量进行优化，确保每次使用该变量时都从内存中取出而不是寄存器中。

哪些情况下使用volatile:

（1）多线程中被几个任务共享的变量（volatile并不保证原子性）

（2）一个中断服务函数中修改的供其他函数检测的变量

（3）存储器映射的硬件寄存器。因为寄存器随时可能被外设硬件修改

**2、static关键字**

注：**静态变量如果没有被初始化，则默认初始化为0**

局部变量

全局变量

普通函数

静态成员变量

1. 所有对象共享同一份数据，属于整个类而不是某个对象
2. 在编译阶段分配内存
3. 类内声明，类外初始化 （进入main前先为全局变量和静态变量分配内存空间及初始化）

静态成员函数

1. 所有对象共享同一个函数，属于整个类而不是某个对象
2. 没有this指针，所以**静态成员函数只能访问静态成员变量**

**5、union 注意内存对齐**

**union Data{**

**double i;**

**int k[5];**

**char j;**

**};**

**cout << sizeof(Data) << endl;** // **结果为 24**

**6、大端小端**

大端：低地址存高位，高地址存低位

小端：低地址存低位，高地址存高位

如何判断大小端存储模式？

int a = 0x1234;

**//** 由于int和char的长度不同，类型转换时只会留下低地址部分

**char c = (char)a;**

if(c == 0x12)

cout << "大端存储模式" << endl;

else if(c == 0x34)

cout << "小端存储模式" << endl;

**悬浮指针**：当指针所指向的对象被释放，但没有将其置为NULL或指向其他合法地址的指针

**野指针**：未初始化的指针

**8、静态链接 和 动态链接**

预处理：处理关于”#”的指令，例如#define #include

gcc **-E** test.c -o test.i

编译：将预处理后的文件转换为汇编代码

gcc **-S** test.i -o test.s

汇编：将汇编代码转换为机器码，得到目标文件 .o

gcc **-c** test.s -o test.o

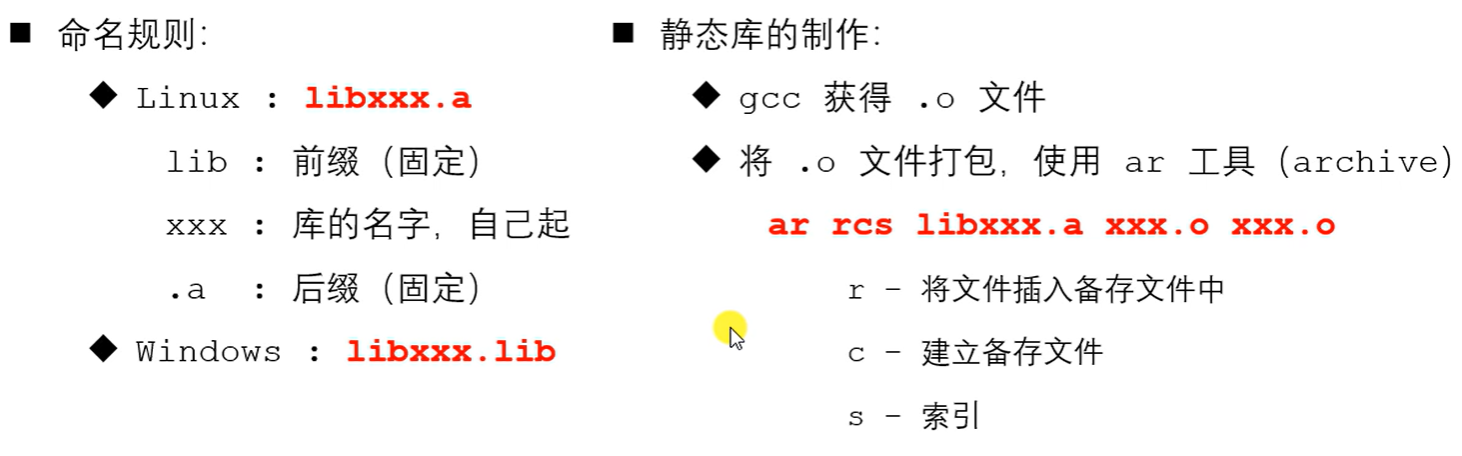
链接：将目标文件与其他目标文件、库文件等链接起来，生成可执行文件 或 库

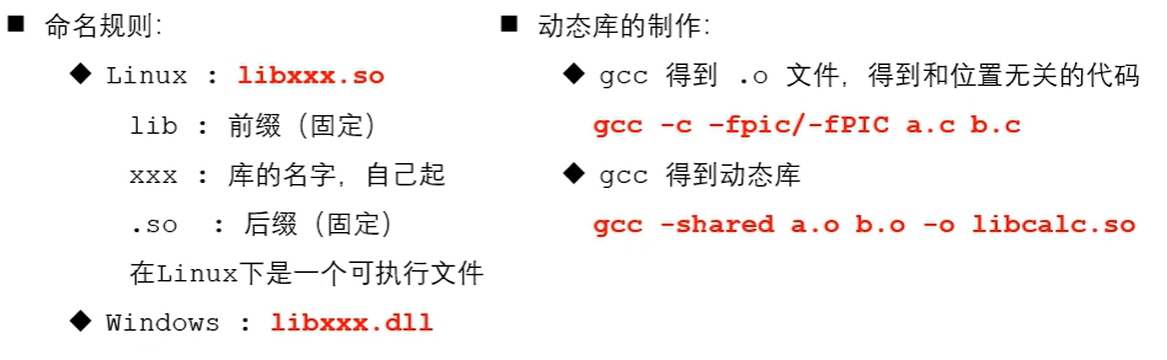
gcc test.o -o test **-L**../lib **-l**pthread

-L 指定库路径

-l 指定要链接的库名称

**Windows下（.lib | .dll） Linux下（.a | .so）**





**静态链接**： 链接时将汇编生成的目标文件**.o**和引用到的库文件(**.lib**、 **.a**)直接一起链接打包为一个可执行文件。这样可执行文件就包含了所有需要的目标代码，不再依赖外部库文件。当程序运行时，操作系统加载整个可执行文件到内存，并直接执行其中的代码。

**优点**：

1、无需额外的依赖

2、没有运行时的库加载开销

**缺点**：

1、可执行文件的体积大

2、库文件更新则需要重新编译和链接整个程序。

**动态链接**：运行时加载并链接程序所需要的库文件。可执行文件只包含程序的目标代码，而使用的函数库被编译为独立的共享库文件（**.dll**、 **.so**）。当程序执行时，操作系统根据需要加载这些共享库，并将其链接到程序中。

**优点**：

1、多个程序可以共享同一个动态链接库，内存占用小

2、共享库更新时，无需重新编译整个程序，只需替换动态链接库文件即可

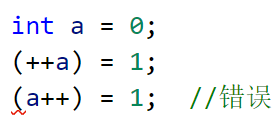
3、因为不包含所有依赖的库代码，可执行文件相对较小

**缺点**：

1. 由于需要在运行时加载和链接库文件，所以动态链接的程序在启动时可能会有轻微的性能开销。
2. 程序执行需要依赖正确版本的动态链接库，如果缺少或者版本不兼容，会导致运行错误

·CDemo &operator++() ; //前置运算符重载

·CDemo operator++(int); //后置运算符重载



**9、多态**

静态多态：编译期确定 函数重载、运算符重载和函数模板、类模板

动态多态：运行期确定

多态性是基于继承和重写（覆盖）的概念实现的。在继承关系中，子类可以使用父类的成员变量和成员函数，同时也可以重写父类的成员函数，以实现更具体的功能。当父类类型的指针或引用指向子类对象时，就可以利用多态性，调用子类中重写的成员函数，从而实现动态绑定和灵活的行为。 同一个接口，使用不同的实例而实现不同的功能

动态多态的必要条件：

、有继承关系

、子类重写基类的虚函数

、基类指针指向子类对象

编译器看到满足多态就会根据指针找到对应对象的虚函数表，然后访问虚函数。

动态多态如何实现的？

当编译器发现类中有虚函数时，会创建一张虚函数表，把虚函数的函数入口地址放到虚函数表中，并在对象中增加一个虚函数表指针vptr，用于指向类的虚函数表。当派生类覆盖基类的虚函数时，会将虚函数表中对应的函数指针进行替换，从而调用派生类中覆盖后的虚函数，从而实现动态绑定。

·每个类都有自己的虚函数表，所以vptr值不一样

·一个类的所有对象共用一个虚函数表，所以vptr值一样

**注意：基类中的纯虚析构需要具体实现。类外实现**

·虚函数只能是类中非静态的成员函数。

·class A{

public:

void func1() { cout << "func1" << endl; }

virtual void func2() { cout << "func2" << endl; }

};

A \*pa = nullptr;

pa->func1();//**调用成功，这是普通成员函数的调用，直接通过对象的静态类型调用**

pa->func2();//**调用失败，虚函数需要对vptr指针获取虚函数表，无法操作空指针获取**

**虚析构：**

在删除指向子类对象的基类指针时，可以确保先调用子类的析构函数释放子类中堆内存，防止内存泄漏。

**10、继承**

**友元关系不会被继承 ！**

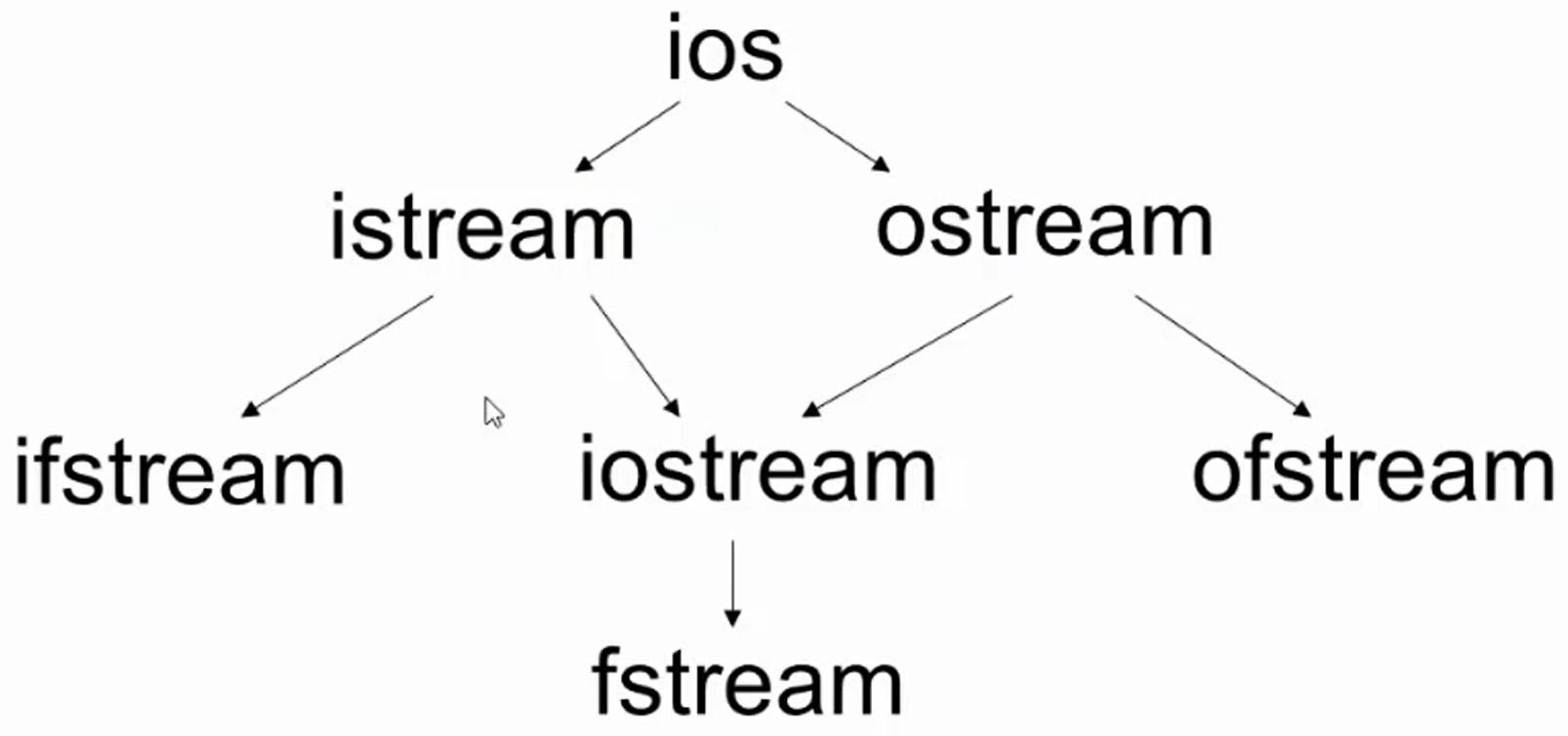
**多继承** 菱形继承 ---> 出现数据二义性、数据冗余

虚继承解决上述问题：

在虚继承的类中，会定义一个虚基类表指针 vbptr ,指向虚基类表。

而虚基类表中会存在偏移量，这个量就是表的地址到父类数据地址的距离。

在一般的继承中，基类的变量排列在派生类变量之前；而当出现虚继承时，虚基类的变量会被放在最后，而编译器会为派生类维护一张虚基类表，表中存放了虚基类变量首地址相对于派生类变量首地址的偏移值，并在构造函数中为每个派生类对象生成一个指向虚基类表的虚基类指针vbptr。



**11、构造函数的调用**

**1、\*\*显式调用\*\***

Person p = Person();

Person p = Person(18);

Person p = Person(18, "xiaoming");

Person p; //注意不能写成 Person p();容易让编译器认为是一个函数。

Person p(18);

Person p(18, "xiaoming");

**2、\*\*隐式调用\*\*** 不需要我们自己写Person()，直接将所有参数写到一对大括号中，编译器会根据大括号中实参的数量和类型自动找到与之匹配的构造函数调用。

Person p = { };

Person p = 18;

Person p = {18，"xiaoming"};

·**public protected private**

注意：控制的是**类内成员**的访问权限

·**匿名对象**

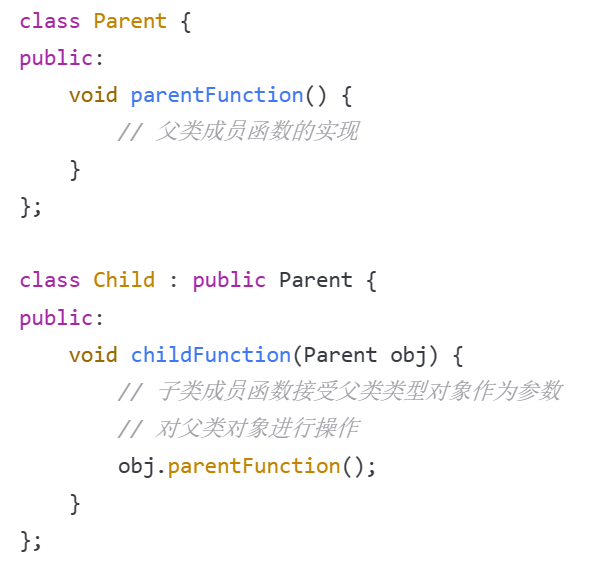
·**常对象只能调用常成员函数**

·如果**一个类中包含成员类**，那么在构造该类的对象时，构造函数的执行顺序取决于成员类在类中的声明顺序

·**构造函数的初始化**



·子类中的成员函数的参数 可以是父类类型的对象



**智能指针 #include <memory>**

RAII（Resource Acquisition is Initilization）,即资源获取即初始化，也就是在构造函数中申请分配资源，在析构函数中释放资源。

* **unique\_ptr** 表示对一个对象的独占所有权，即同一时间只能有一个unique\_ptr指向同一个对象。当unique\_ptr被销毁时，它会自动释放它所管理的对象的内存。

禁用了拷贝操作，采用std::move进行所有权的转移

* **shared\_ptr** 允许多个shared\_ptr共享对同一对象的所有权。它使用引用计数来追踪有多少个shared\_ptr指向相同的对象，当最后一个shared\_ptr被销毁时，所管理的对象的内存会被释放。

shared\_ptr在类内部维护着一份引用计数，用来记录该份资源被几个shared\_ptr共享

当一个shared\_ptr被销毁时（调用析构函数）,析构函数内就会将该计数 -1

如果引用计数减为0后，表明当前shared\_ptr是最后一个使用该资源，使用完之后必须释放所管理的资源。

* **weak\_ptr** 弱引用指针。解决两个或多个shared\_ptr相互引用时，其引用计数永远不会减为0，即解决循环引用导致的内存泄漏问题。

**x++是右值**，因为在后置++操作中编译器首先会生成一份x值的临时复制，然后才对x递增，最后返回临时复制内容。

**++x是左值**，它是直接对x递增后马上返回其自身

**左值**，可以被取地址 &

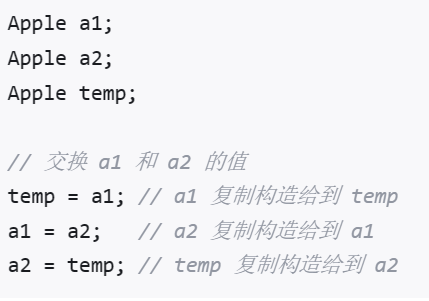
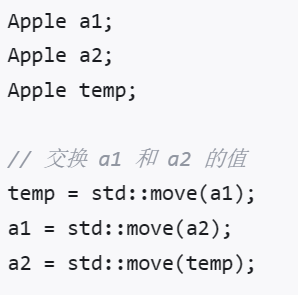
**右值**，临时值，不能被取地址 （函数返回值、lambda表达式、字面量......）

**右值引用** 🡪 对右值的引用

int num = 10;

int &&a = num; //错误，右值引用不能初始化为左值

int &&a = 10;

盒子和内容的概念

std::move的语义其实是将变量（左值）转换为了右值，然后对其所有权进行了移动。

**右值引用的作用：1、**减少内存复制 **2、**延长临时对象的生命周期 **3、**完美转发

/\*\*\*\*\*/





·在这段代码中，主函数中接收函数返回的变量使用了 && 右值引用。

具体的含义是，对于 test() 函数返回的值，我们直接使用 a 变量来接管（直接指向返回的 Person 类实例），test 函数里创建的 tmp 临时变量也就不需要析构了，直接交给主函数的 a 变量继续使用。临时变量的拷贝构造也就省去了

·为什么这里可以用右值引用直接接管呢？我们可以这么理解：

test() 函数返回的是一个具体的值（在这里是一个 Person 类的实例），满足右值的定义，其在函数返回的时候生命周期理应就结束了。如果此时没有变量接管的话，这个类实例就会被析构。既然都已经是不要的值了，且主函数跟test的作用域已经不同，直接使用也不会有冲突，那我们完全可以给这个值所在的内存区域定义一个新的别名（也就是主函数中的 a），将这个值直接接管，在主函数中继续使用，也就不需要再拷贝构造一个新的了。

/\*\*\*\*\*/

**四种类型转换：**

**const\_cast<T>(expression)**

**：**用于去除const或volatile属性

**dynamic\_cast<T>(expression)**

**：**子类向父类转换（有运行时类型检查），转换失败时对于指针会返回目标类型的nullptr，对于引用返回bad\_cast异常

**static\_cast<T>(expression)**

**：**基本类型转换、

子类向父类转换（没有运行时类型检查）、

void\*指针转换

**reinterpret\_cast<T>(expression)**

**：**用于将一个指针或引用重新解释为不同类型，无论它们是否相关

**Lambda表达式**

Shell

**ls -al ll**

**which** 查看指定命令所在路径。如 which ls

**rm -i** 删除文件 / 目录。 -i 删之前会确认 -r 递归的 -f 强制的

**touch** 将每个文件的访问及修改时间更新为当前时间。如果文件不存在，则创建一个字节数为0的文件

**cp -i** 复制文件/目录。 -i 复制之前会确认 -a 所有（属性、权限） -r 递归的

**chmod** chmod u+x,g+rw,o=rwx aaa.c

u g o

rw- rw- r--

6 4 4

chmod 644 aaa.c

**chown** 改变文件或目录的拥有者和所属组

chown root aaa.c

chown lulu aaa.c

chown lulu:admin aaa.c 用户：组

**mkdir** -p 确保目录存在，不存在就创建一个 parent

**cd** -切换到上次目录 **~**切换到用户目录 如：/home/lulu

**ps** 用于显示当前正在运行的进程 ps -aux

**pwd** 显示当前工作目录路径

**mv** 将文件或目录 移动或重命名

**time** time ./a.out

real 程序执行的实际时间 = sys + user + 等待时间

user 用户空间消耗时间

sys 内核空间消耗时间

**tar**

tar **-zcvf** file.tar.gz file 压缩

tar **-zxvf**  file.tar.gz -C path 解压到指定路径下

**top** 查看操作系统的信息，如进程、CPU占用率、内存信息等，类似于任务管理器

**kill** 向进程发送信号signal

kill -9 进程号 杀死进程 kill -l 列出所有信号

9 SIGKILL 进程直接杀死

15 SIGTERM 进程友好退出，可以被阻塞、处理和忽略

**echo** 终端或文件中输出字符串

echo $PATH

输出重定向 > 覆盖输出（会覆盖原文件内的内容）

>> 追加输出（不覆盖原文件的内容）

**cat** 查看文件里的内容，输出到终端 cat /proc/cpuinfo

**df** 查看磁盘空间占用 -h 易读 -a 所有

**du**  查看文件或目录占用大小 du -sh mydir

-sh 仅显示当前文件或目录大小

-a 显示当前目录下及其子目录下每个文件和目录的大小

**uname** 显示系统的详细信息 uname -r 显示内核版本

**man**手册 1shell命令、 2系统调用、 3 C库函数

**grep** 搜索文件中的内容

如 grep -n aaa filename.txt grep -n “aaa\*” filename.txt

-i：忽略大小写。ignore

-w：仅匹配整个单词。whole

-n：显示匹配行及行号

-r ：当前目录下递归搜索所有文件。recursion grep -r hello ./home/lulu

**find** 查找文件/目录 -name -type b/d/c/p/l/f 块文件/目录/字符文件/管道/符号/普通文件

如：find **./** -name “aaa.\*”

find **./** -name aaa.c

find **./** -type f 搜索文件

find **./** -type d 搜索目录

**vim**

**ctrl + f** 翻下一页 front

**ctrl + b** 翻上一页 back

**u** 撤销

**v** 进入可视化

**y** 复制

**d** 剪切

**yy** 复制一行

**dd** 剪切一行

**p** 粘贴

查看当前Linux提供的Shell：cat /etc/shells

/bin/sh

/bin/bash

/usr/bin/sh

/usr/bin/bash

查看当前Linux默认Shell解释器：echo $SHELL

shebang行 #！

#!/bin/bash 指定shell解释器

echo “hello world”

bash hello.sh 当前shell终端环境会单独开一个子shell即 bash来执行hello.sh

./hello.sh 如果hello.sh里shebang行指定了另一个shell，./hello.sh会使用指定的shell执行脚本

直接在当前shell终端环境执行脚本： **.** hello.sh 、source hello.sh，不会单独开一个shell执行

系统变量：$HOME $PWD $SHELL $USER等

查看当前shell中定义的所有变量：set

**STL**

容器：容纳各种数据类型的通用数据结构，是类模板

迭代器：依次存取容器中元素，类似于指针（iterator和const\_iterator两种类型）

算法：用来操作容器中的数据的函数，是函数模板

**1、string**

构造函数

string(); //创建一个空的字符串，如 string str;

string(const string &str); //使用一个string对象初始化另一个string对象

string(const char \*s); //使用字符串s初始化

string(int n, char c); //使用n个字符c初始化

基本赋值操作

string& operator=(const char\* s); //char\*类型字符串 赋值给当前的字符串

string& operator=(const string &s); //把字符串 s 赋给当前的字符串

string& operator=(char c); //字符赋值给当前的字符串

string& assign(const char \*s); //把字符串 s 赋给当前的字符串

string& assign(const char \*s, int n); //把字符串s的前n个字符赋给当前的字符串

string& assign(const string &s); //把字符串 s 赋给当前字符串

string& assign(int n, char c); //用 n 个字符 c 赋给当前字符串

string& assign(const string &s, int start, int n); //将s从start索引号开始n个字符赋值给字符串

字符串存取操作

char& operator[](int n); //通过[]方式取字符

char& at(int n); //通过 at 方法获

字符串拼接操作

string& operator+=(const string& str); //重载+=操作符

string& operator+=(const char\* str); //重载+=操作符

string& operator+=(const char c); //重载+=操作符

string& append(const char \*s); //把字符串 s 连接到当前字符串结尾

string& append(const char \*s, int n); //把字符串 s 的前 n 个字符连接到当前字符串结尾

string& append(const string &s); //同 operator+=()

string& append(const string &s, int pos, int n); //把字符串 s 中从 pos 开始的 n 个字符连接到当前字符串结尾

string& append(int n, char c);//在当前字符串结尾添加 n 个字符 c

字符串比较操作

compare 函数在>时返回正整数，<时返回负整数，==时返回 0。比较区分大小写，比较时参考字典顺序，排越前面的越小。大写的 A 比小写的 a 小。

int compare(const string &s) const; //与字符串 s 比较

int compare(const char \*s) const; //与字符串 s 比较

string 插入和删除操作

string& insert(int pos, const char\* s); //插入字符串

string& insert(int pos, const string& str); //插入字符串

string& insert(int pos, int n, char c); //在指定位置插入 n 个字符 c

string& erase(int pos, int n = npos); //删除从 Pos 开始的 n 个字符

**string 和 c-style 字符串转换**

//string 转 char\*

string str = "itcast";

const char\* cstr = str.c\_str();

//char\* 转 string

char\* s = "itcast";

string str(s);

**2、vector 动态数组 单向开口连续内存空间**

构造函数

vector<T> v; //采用模板实现类实现，默认构造函数

vector<T> (v.begin(), v.end()); //将 v[begin(), end())区间中的元素拷贝给本身。

vector<T> (n, elem); //构造函数将 n 个 elem 拷贝给本身。

vector<T> (const vector &vec); //拷贝构造函数。

常用赋值操作

assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。

assign(n, elem); //将 n 个 elem 拷贝赋值给本身。

vector& operator=(const vector &vec); //重载等号操作符

swap(vec); //将 vec 与本身的元素互换。

大小操作

size(); //返回容器中元素的个数

empty(); //判断容器是否为空

resize(int num); //重新指定容器的长度为 num，若容器变长，则以默认值填充新位

置。如果容器变短，则末尾超出容器长度的元素被删除。

resize(int num, elem); //重新指定容器的长度为 num，若容器变长，则以 elem 值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

capacity(); //容器的容量

reserve(int len); //容器预留 len 个元素长度，预留位置不初始化，元素不可访问。

at(int idx); //返回索引idx所指的数据，如果idx越界，抛出out\_of\_range异常。

operator[]; //返回索引 idx 所指的数据，越界时，运行直接报错

front(); //返回容器中第一个数据元素

back(); //返回容器中最后一个数据元素

插入删除操作

insert(const\_iterator pos, int count, ele); //迭代器指向位置 pos 插入 count个元素 ele.

push\_back(ele); //尾部插入元素 ele

pop\_back(); //删除最后一个元素

erase(const\_iterator start, const\_iterator end); //删除迭代器从 start 到 end 之间的元素

erase(const\_iterator pos); //删除迭代器指向的元素

clear(); //删除容器中所有元素

**3、deque 双端队列 双向开口连续内存空间**

deque 没有容量的概念，因为它是动态的以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，换句话说，像vector 那样，“旧空间不足而重新配置一块更大空间，然后复制元素，再释放旧空间”这样的事情在 deque 身上是不会发生的

**4、stack 先进后出，不允许有遍历行为，所以不提供迭代器**

构造函数

stack<T> stk; //stack 采用模板类实现，stack 对象的默认构造形式：

stack<T> (const stack &stk); //拷贝构造函数

赋值操作

stack& operator=(const stack &stk); //重载等号操作符

数据存取操作

push(elem); //向栈顶添加元素

pop(); //从栈顶移除第一个元素

top(); //返回栈顶元素

大小操作

empty(); //判断堆栈是否为空

size(); //返回堆栈的大小

**5、queue 队列 先进先出，不允许有遍历行为，所以不提供迭代器**

构造函数

queue<T> que; //queue 采用模板类实现，queue 对象的默认构造形式：

queue<T> (const queue &que); //拷贝构造函数

存取、插入和删除操作

push(elem); //往队尾添加元素

pop(); //从队头移除第一个元素

back(); //返回最后一个元素

front(); //返回第一个元素

赋值操作

queue& operator=(const queue &que); //重载等号操作符

大小操作

empty(); //判断队列是否为空

size(); //返回队列的大小

**6、list 双向链表**

不支持随机访问

**7、set** **/ mutiset** 红黑树

所有元素都会根据元素的键值自动排序，set 的元素即是键值又是实值。

multiset 键值可重复

**8、map / mutimap** 红黑树

所有元素都会根据元素的键值自动排序，map元素同时拥有键值和实值。

multimap 键值可重复

Linux

pid\_t pid = fork(); // 创建子进程

返回值：

fork()的返回值会返回两次。一次是在父进程中，一次是在子进程中。

在父进程中返回创建的子进程的ID,

在子进程中返回0

在父进程中返回-1，表示创建子进程失败，并且设置errno

getpid( ) 获取当前进程id

getppid( ) 获取当前进程的父id

getuid( ) 获取当前进程的用户id

getgid( ) 获取当前进程的组id

execl() 执行一个程序

execlp()

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid,int \*wstatus,int options); //指定pid清理进程，可以不阻塞

死锁的必要条件：

1. 互斥条件，一段时间内一个资源只能被一个线程占用
2. 不可剥夺条件，线程已获得的资源在未使用完之前不能被剥夺，只能在使用完自己释放
3. 请求和保持条件，当线程因请求资源而阻塞时，对已获得的资源保持不放
4. 循环等待条件，多个线程之间必须形成循环等待

线程pthread上的锁：

读写锁

锁只有一把，读模式锁，写模式锁

读共享，写独占 🡪允许多个线程同时持有读锁，但只允许一个线程持有写锁

写锁优先级高

互斥锁

对共享资源进行锁定，保证同一时刻只有一个线程去操作资源

自旋锁

一种忙轮询的锁，它不会将线程阻塞，而是在获取锁失败时轮询等待，直到成功获

取锁为止

条件变量

条件变量本身不是锁！通常和互斥锁一起使用

**pthread\_cond\_wait()**

1. 阻塞等待条件变量满足
2. 释放已掌握的互斥锁，相当于pthread\_mutex\_unlock()

当被唤醒，pthread\_cond\_wait()返回时

1、重新加锁，相当于pthread\_mutex\_lock()

2、解除阻塞

int pthread\_create ( pthread\_t \*tid,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine)(void \*),

void \*arg );

pthread\_t pthread\_self(void);

void pthread\_exit(void \*retval); 线程退出

int pthread\_join(pthread\_t tid, void \*\*retval); 阻塞等待线程退出，获取线程退出状态

int pthread\_detach(pthread\_t tid); 线程分离

int pthread\_cancel(pthread\_t tid); 取消点

内存映射mmap：

将文件或设备的内容映射到进程的虚拟地址空间，从而直接访问内存操作文件或设备，无需文件I/O调用。

**void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);**

addr: 期望的内存映射起始地址。如果传入 NULL，系统会选择合适的地址。

length: 映射区域的长度（以字节为单位）。

prot: 映射区域的保护标志，用于指定访问权限。常见的标志有：

PROT\_READ: 允许读取

PROT\_WRITE: 允许写入

PROT\_EXEC: 允许执行

PROT\_NONE: 不允许访问

flags: 映射的类型和行为标志。常见的标志有：

MAP\_SHARED: 映射的内容对所有映射该文件的进程可见。

MAP\_PRIVATE: 映射的内容对该进程私有，修改不会写回文件。

MAP\_ANONYMOUS: 映射匿名内存（不与任何文件关联），需要将 fd 设置为 -1。

fd: 要映射的文件描述符。如果 flags 包含 MAP\_ANONYMOUS，fd 应设置为-1。

offset: 文件中的偏移量，映射从文件的这个偏移量开始。

返回值：

成功时返回映射区域的指针。

失败时返回 MAP\_FAILED，并设置 errno。

**int munmap(void \*addr, size\_t length);**

addr: 映射区域的起始地址。

length: 映射区域的长度。

munmap 成功时返回 0，失败时返回 -1，并设置 errno。

**int atoi(const char \*str);** 字符串转换为整数

返回值：

转换后的整数值。

如果字符串中第一个字符为 字母 或 符号，atoi 会返回 0。

如果转换的数字超出了 int 类型的范围，atoi 会返回 INT\_MAX 或 INT\_MIN。

printf("test\_value1 = %d\n", atoi(“23”) ); //23

printf("test\_value2 = %d\n", atoi("DLUTBruceZhang23") );//0

printf("test\_value3 = %d\n", atoi("23DLUTBruceZhang") );//23

**Socket**

**以太网**，一种局域网（LAN）技术 | 是物理层和数据链路层的技术，用于在局域网中建立连接和传输数据

**因特网**是一个全球性的计算机网络，由许多相互连接的局域网和广域网构成。通过标准互联网协议（TCP/IP协议）进行通信

**万维网**则是一种基于因特网的信息系统，它使用了标准化的HTML语言和HTTP协议，通过超链接把各种文本、图片、音频和视频等多媒体资源组织起来，使得用户可以轻松地跨越因特网上的不同站点和页面之间进行浏览、检索和交互，可以说万维网是建立在因特网之上的一个应用层面

**MAC地址** 唯一标识一个网卡，一台设备若有多个网卡，则每个网卡都会有一个唯一的 MAC 地址 **6个字节，**长度**48位**

**IP地址** 唯一标识一台主机，为互联网上的每一台主机分配一个逻辑地址，以此来屏蔽MAC物理地址的差异，逻辑地址 **4个字节，长度32位**

**端口号** 唯一标识一个进程 **2个字节，长度16位**

**OSI七层模型 （标准）**

应用层：各种应用软件，包括 Web 应用

表示层：数据格式标识，基本压缩加密功能

会话层：控制应用程序之间会话能力；如不同软件数据分发给不同软件

传输层：端到端传输数据的基本功能；如 TCP、UDP （端口）

网络层：定义IP编址，定义路由功能；如不同设备的数据转发 （主机）

数据链路层：定义数据的基本格式--数据帧，如何传输，如何标识；

物理层：底层数据传输，比特流

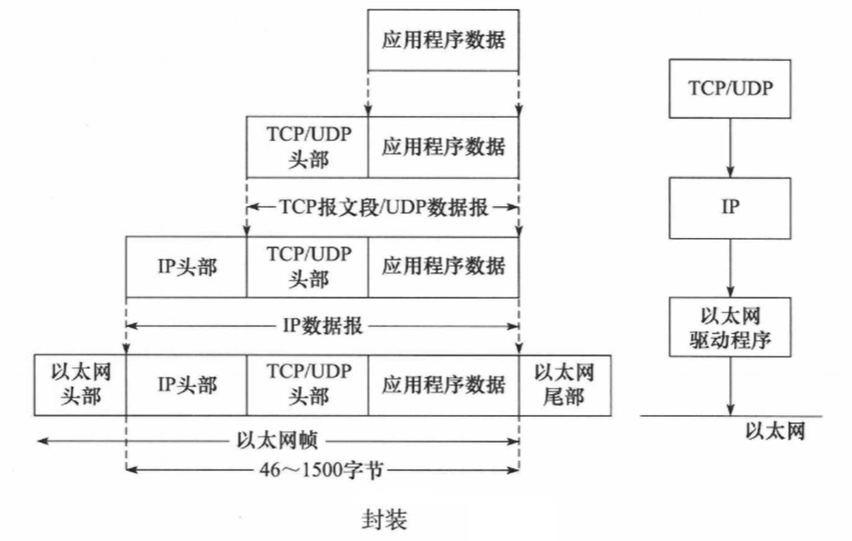
**TCP/IP四层模型 （实现）**

4、应用层 HTTP协议，FTP协议，NFS协议

3、传输层 TCP/UDP协议

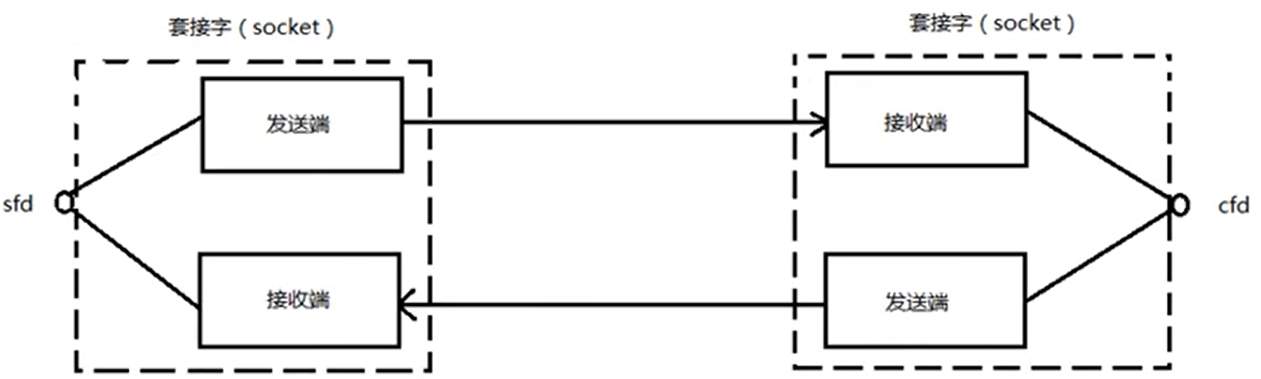
2、网络层 IP协议，ICMP协议，IGMP协议

1、网络接口层 以太网帧协议 ARP协议（通过ip查找mac）、RARP协议



通信过程中，套接字一定是**成对出现**的

一个文件描述符指向一个套接字(该套接字内部由内核借助两个缓冲区实现读写)



网络字节序 网络中的数据流一般采用大端字节序

uint32\_t **htonl**(uint32\_t hostlong); 主机 🡪 网络 （IP地址）

uint16\_t **htons**(uint16\_t hostshort); 主机 🡪 网络 （端口号）

uint32\_t **ntohl**(uint32\_t hostlong); 网络 🡪 主机 （IP地址）

uint16\_t **ntohs**(uint16\_t hostshort); 网络 🡪 主机 （端口号）

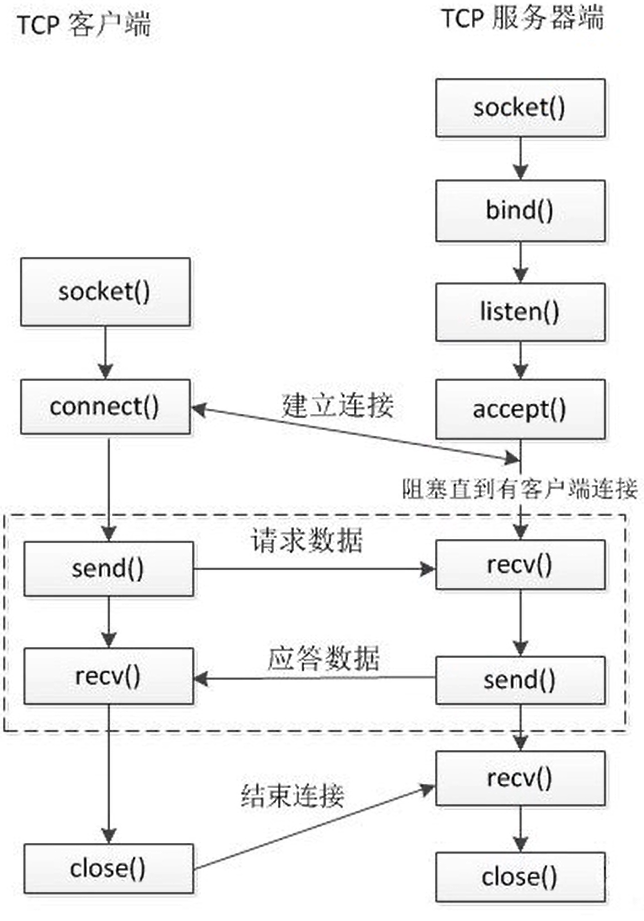
IP地址转换函数：

int **inet\_pton**(int af, const char \*src, void \*dst);

点分十进制IP 🡪 二进制网络字节序

const char \***inet\_ntop**(int af, const void \*src, char \*dst, socklen\_t size);

二进制网络字节序 🡪 点分十进制IP



绑定ip和端口

设置监听上限

阻塞监听客户端连接，返回一个新的socket用于通信

**I/O多路复用**

处理高并发：内核监听多个socket

**select**

**poll**

**epoll**

CAN/CANFD

Controller Area Network（控制器局域网络）

异步，无需时钟线，通信速率由设备各自约定

半双工，先占先得、非破坏性仲裁

差分信号通信，（CAN\_H，CAN\_L），无需共地

高速CAN，125K~1Mbps，<40m

低速CAN，10K~125Kbps，<1km

11位/29位报文ID，用于区分消息功能

1~8字节有效数据（CANFD：1~64字节）

图示

描述已自动生成



手机屏幕截图

描述已自动生成

**隐形 --> 默认电平**

线与机制

首先分清同步串行通信和异步串行通信

同步sync：有根专门的CLK时钟线作同步，根据CLK时钟沿采样数据

异步async：双方约定通信速率，接收方内部定时采样

🡪 CAN

帧类型：数据帧、远程帧、错误帧、过载帧、帧间隔

1.位填充：

连续**5**个相同位后插入一个相反位，产生跳变沿，用于同步

2.位时序：

1 bit = 10 Tq

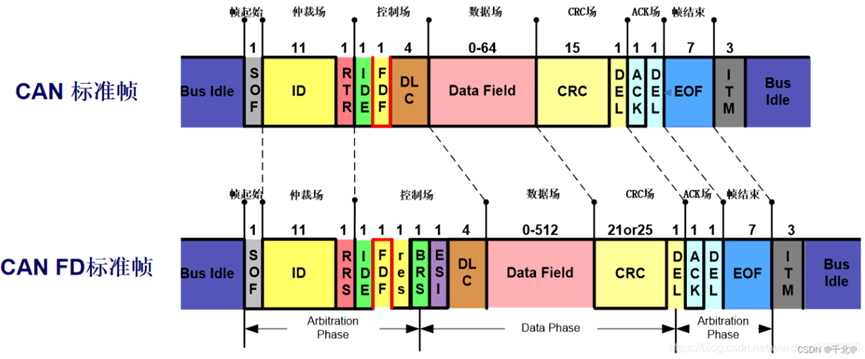
SS同步段、PTS、PBS1、PBS2

卡通人物

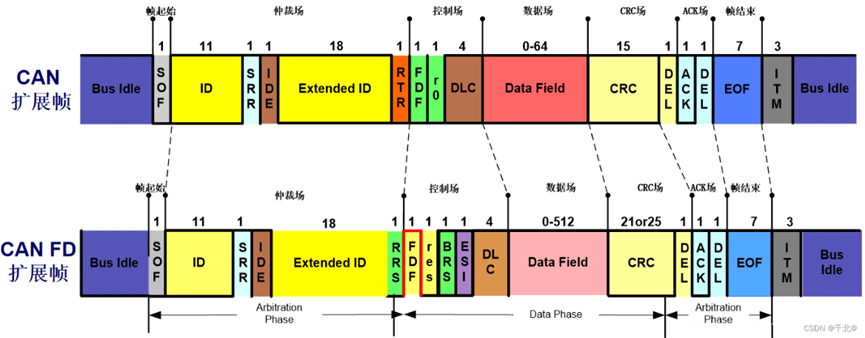
中度可信度描述已自动生成

硬同步 再同步

3.数据帧格式：

****



****



SOF，帧开始，1位。0开始

ID，报文ID，区分功能，同时决定优先级

SRR，无意义保留位

RTR。0数据帧，1远程帧

IDE，区分标准帧还是扩展帧。0标准帧，1扩展帧

r0/r1，保留位

DLC，4位，表示数据长度

DEL，界定符，1位。1

ACK，期间，发送方发送1，接收方接收正确发送0

EOF，帧结束，7位。1结束

4.仲裁：

连续11个隐形1电平即认为总线空闲

1、多个设备不同时有发送需求，那肯定先占先得

2、多个设备同时有发送需求，非破坏性仲裁（逐位）（根据ID号小的优先级高）

🡪 CANFD

·只对数据帧做了调整

·取消了对远程帧的支持

图示

描述已自动生成

日程表

描述已自动生成

RRS，替代RTR，常显性0电平，取消了对远程帧的支持

FDF。常隐形1，表示CANFD帧

res，

BRS。0 数据段使用与仲裁段相同的速率，1数据段使用更高的速率

ESI，指示发送节点的错误状态。0主动错误，1被动错误

**<linux/list.h>**

**list\_for\_each**

定义: list\_for\_each(pos, head)

作用: 遍历一个链表，不允许在遍历过程中删除节点。

参数:

pos：用于存储链表中当前节点指针的迭代器（通常是 struct list\_head \* 类型）。

head：链表头指针，类型为 struct list\_head \*。

**list\_for\_each\_safe**

定义: list\_for\_each\_safe(pos, n, head)

作用: 在遍历链表时，可以安全地删除当前节点。

参数:

pos：用于存储链表中当前节点的指针。

n：用于存储下一个节点的指针，确保遍历时节点被安全删除。

head：链表头指针。

**list\_for\_each\_entry**

定义: list\_for\_each\_entry(pos, head, member)

作用: 遍历包含struct list\_head成员的结构体链表并直接获取包含该成员的结构体指针。

参数:

pos：用于存储当前链表节点的结构体指针。

head：链表头指针。

member：struct list\_head 在结构体中的成员名称。

**list\_for\_each\_entry\_safe**

定义: list\_for\_each\_entry\_safe(pos, n, head, member)

作用: 安全地遍历包含 struct list\_head 成员的结构体链表，并直接获取包含该成员的结构体指针，允许在遍历过程中删除当前节点。

参数:

pos：用于存储当前节点的结构体指针。

n：用于存储下一个节点的结构体指针，确保安全删除。

head：链表头指针。

member：struct list\_head 在结构体中的成员名称。

Make

Targets1：Dependencies

Commands

…

Targets2：Dependencies

Commands

…

默认第一个目标为要生成的最终目标

注意通配符%和\*的区别

**％** 是 Makefile 规则中使用的，主要用于定义模式规则时保持目标和依赖项的对应关系。

**\***  不涉及规则模式的对应关系。

比如下面OBJ := $(**patsubst** %.c, %.o, $(SRC))这句，表示将变量SRC中每个 .c 文件 对应的替换为 .o文件，file.c替换为file.o，foo.c替换为foo.o这样。

SRC := $(**wildcard** \*.c)

这会将当前目录下所有 .c 文件的名称赋值给变量 SRC。

OBJ := $(**patsubst** %.c, %.o, $(SRC))

这会将变量SRC中每个文件名的 .c后缀替换为 .o，生成目标文件名列表赋值给变量OBJ。

文件搜寻：

1、VPATH = src **:** ../headers

上面的定义指定两个目录，“src”和“../headers”，make会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

2、vpath %.h ../headers

该语句表示，要求make在“../headers”目录下搜索所有以 .h 结尾的文件。（如果某文件在当前目录没有找到的话）

变量赋值 **:=** 和 **=** 的区别：

1、 VAR1 = value1

VAR2 **=** $(VAR1)

VAR1 = value2

all:

@echo $(VAR2) # 输出 value2 **=** 变量用的时候再展开

2、 VAR1 = value1

VAR2 **:=** $(VAR1)

VAR1 = value2

all:

@echo $(VAR2) # 输出 value1 **:=** 变量定义的时候立即展开

模式规则：

objects = foo.o bar.o

all: **$(**objects**)**

$(objects): %.o: %.c

**$(**CC**)** -c **$(**CFLAGS**)** **$<** -o **$@**

上面的例子中，指明了我们的目标从 $object 中获取， %.o 表明要所有以 .o 结尾的目标，也就是 foo.o bar.o ，也就是变量 $object 集合的模式，而依赖模式 %.c 则取模式 %.o 的 % ，也就是 foo bar ，并为其加下 .c 的后缀，于是，我们的依赖目标就是 foo.c bar.c 。而命令中的 $< 和 $@ 则是自动化变量， $< 表示第一个依赖文件， $@ 表示目标集（也就是“foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

foo.o : foo.c

**$(**CC**)** -c **$(**CFLAGS**)** foo.c -o foo.o

bar.o : bar.c

**$(**CC**)** -c **$(**CFLAGS**)** bar.c -o bar.o

内核编译：

**obj-m** 用于指定要编译成可加载模块的目标

**obj-y** 用于指定要编译进内核的目标

CMake

mkdir build

cmake <CMakeLists.txt对应的文件路径>

make

·指定使用的最低cmake版本

**cmake\_minimum\_required**(VERSION 3.12)

·定义工程名称

**project**(test)

·生成一个可执行程序

**add\_executable**(app aaa.c bbb.c ccc.c ddd.c)

·定义变量

**set**(SRC\_LIST add.c div.c mult.c sub.c main.c)

**add\_executable**(app ${SRC\_LIST})

·指定C++标准

**set**(CMAKE\_CXX\_STANDARD 11)

·指定可执行程序输出路径

**set**(HOME /home/robin/linux/sort)

**set**(EXECUTABLE\_OUTPUT\_PATH ${HOME}/bin)

·搜索文件

**aux\_source\_directory**(<dir> <variable>)

dir: 要搜索的目录

variable: 将dir目录下搜索到的源文件列表存储到该变量中

**file**(GLOB/GLOB\_RECURSE 变量名 要搜索的文件路径和文件类型)

GLOB: 将指定目录下搜索到的满足条件的所有文件名生成一个列表，存储到变量

GLOB\_RECURSE: 递归搜索指定目录

·指定头文件路径

**include\_directories**(头文件路径)

**PROJECT\_SOURCE\_DIR**

指向 最近一次调用project()命令的 CMakeLists.txt 文件 所在的目录。

在包含多个子项目的情况下，每个子项目（每次调用project()命令）都会重新设置 PROJECT\_SOURCE\_DIR，这使得每个子项目可以有自己独立的源代码目录路径。

**CMAKE\_SOURCE\_DIR**

指向最顶层的 CMakeLists.txt 文件所在的目录，即整个项目的根目录。

这个变量在整个项目构建过程中保持不变，无论当前处理的是哪个 CMakeLists.txt 文件。用于定义整个项目范围内的源代码路径。

**CMAKE\_CURRENT\_SOURCE\_DIR**

**---------**

**PROJECT\_BINARY\_DIR**

构建目录

**CMAKE\_BINARY\_DIR**

构建目录

**CMAKE\_CURRENT\_BINARY\_DIR**

·静态库

**add\_library**( libxxx.a STATIC <源文件….>)

·动态库

**add\_library**( libxxx.so SHARED <源文件….>)

·设置动态库/静态库生成路径

**set**(LIBRARY\_OUTPUT\_PATH <path>)

·链接静态库

指定库路径

**link\_directories**(${PROJECT\_SOURCE\_DIR}/lib)

指定静态库

**link\_libraries**(libxxx.a libyyy.a libzzz.a)

**link\_libraries**(xxx yyy zzz)

·链接动态库

**target\_link\_libraries**( <target> libxxx.so PRIVATE libyyy.so INTERFACE libzzz.so)

target: 可执行程序 或 库

动态库的链接具有传递性，默认PUBLIC。

如D🡪A🡪B，C

若A🡪 PRIVATE B，则D🡪A时，D不能使用B中的数据

若A🡪C，则D🡪A时，D是可以知道C中的数据的

若A🡪INTERFACE B, INTERFACE C，A中可以使用B，C中的函数，但不知道使用的函数是B中的还是C中的。

·添加子目录（包含CMakeLists.txt文件）

**add\_subdirectory**(source\_dir )

内核驱动基本模块 **struct kobject**

kobject kset

**struct cdev**

字符设备结构体

dev\_t 32位数据类型 高12bit为主设备号，低20bit为次设备号

MAJOR(dev\_t)

MINOR(dev\_t)

MKDEV(ma, mi)

1、申请设备号

静态分配设备号：

事先知道主、次设备号时使用 cat /proc/devices

int **register\_chrdev\_region**(dev\_t from, unsigned count, const char \*name)

from：可以通过MKDEV(ma, mi)来指定

动态申请设备号：

int **alloc\_chrdev\_region**(dev\_t \*dev, unsigned baseminor, unsigned count, const char \*name)

2、初始化cdev结构体

void **cdev\_init**(struct cdev \*cdev, const struct file\_operations \*ops)

3、将初始化后的cdev结构体注册到内核中

int **cdev\_add**(struct cdev \*cdev, dev\_t dev, unsigned count)

4、创建设备类和设备节点

struct class \***class\_create**(struct module \*owner, const char \*name) /sys/class/目录下

struct device \***device\_create**(struct class \*class, /dev/目录下

struct device \*parent,

dev\_t devt,

void \*drvdata,

const char \*fmt, ...)

5、删除注销

void **class\_destroy**(struct class \*class);

void **device\_destroy**(struct class \*class, dev\_t dev)

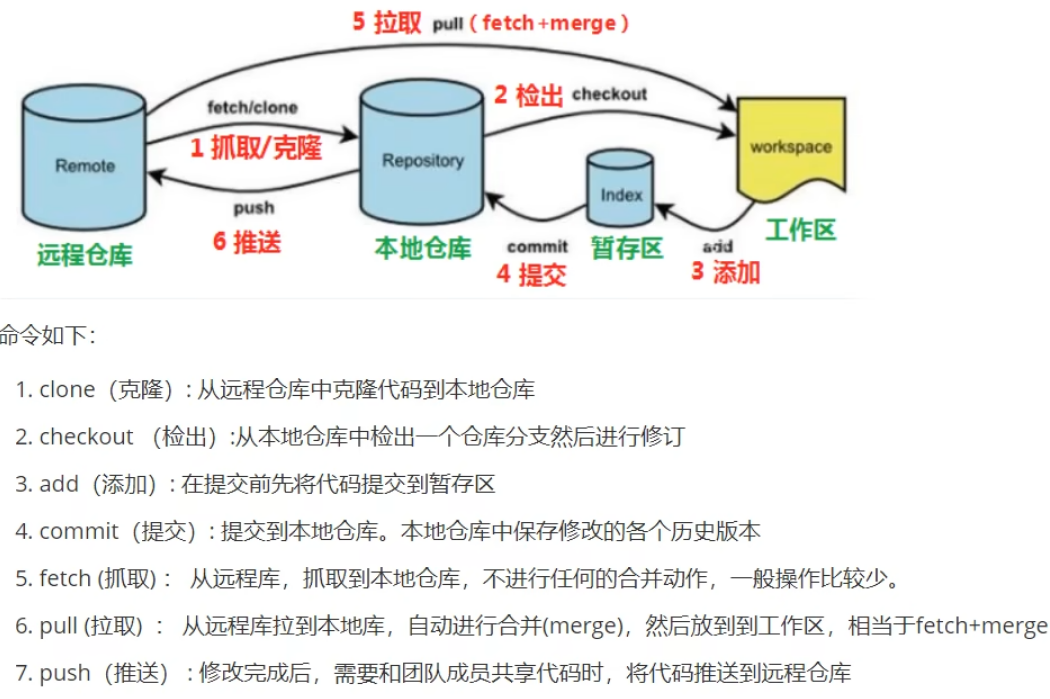
void **unregister\_chrdev\_region**(dev\_t from, unsigned count)

void **cdev\_del**(struct cdev \*cdev)

创建一个设备文件**mknod** 如：mknod /dev/demo **c** 251 0 主设备号251次设备号0

一切皆文件

Git



HEAD指向当前所在分支

master、hotfix、feature等指向各自分支的版本

git config –global user.name ”lulu”

git config –global user.email ”lulu@163.com”

查看配置信息：

git config --global user.name

git config --global user.email

git config --list

创建仓库：

git init初始化一个本地仓库

直接git clone一个远程仓库

git status

git add **./**

git commit -m ”add file.c”

查看提交日志：

git log –all 显示所有分支的提交信息

git log –oneline 提交信息显示为一行

git log –graph 以图的形式显示

git reflog

将添加到暂存区的文件删除：

git rm –cached <file>

指定退回某一次提交的版本：

git reset –soft <commitID>

保留工作区和暂存区的所有修改内容

git reset –hard <commitID> 如: git reset –hard 2a8b667

丢弃工作区和暂存区的所有修改内容

git reset –mixed <commitID> （默认参数）

保留工作区的修改内容，丢弃暂存区的修改内容

git branch

默认列出本地分支

-r 只列出远程分支

-a 列出所有分支

-v 查看本地分支及对应的提交记录

-d 删除合并后的分支，有未合并的分支会提示错误

-D 强制删除分支（即使未合并）

git branch <分支名> 创建分支（但并不切换到新分支里）

git checkout -b 创建并切换到新分支

git checkout <分支名> 切换分支

合并分支：首先应切换分支到master分支，站在master分支上将别的分支合并过来。

git merge hotfix 将hotfix分支合并到当前master分支

解决冲突：

1. git status 检测当前状态查看哪些文件冲突
2. 解决冲突
3. git add <file> 将解决冲突后的文件添加到暂存区
4. git commit -m “commit record”

git remote -v 列出所有远程仓库的URL和别名

git remote add 添加一个新的远程仓库并指定别名和URL

eg: git remote add origin https://github.com/user/repo.git

将本地的local-branch分支推送到远程origin仓库的remote-branch分支：

git push origin <本地分支>:<远程分支>

将远程origin仓库的remote-branch分支拉取到本地的remote-branch分支：（fetch+merge）

git pull origin <远程分支>:<本地分支>

获取特定远程分支的更新：

git fetch <remote> <branch>