# 面试学习笔记

## 进程之间的通信

### 管道：

管道是最简单，效率最差的一种通信方式。

管道本质上就是内核中的一个缓存，当进程创建一个管道后，Linux会返回两个文件描述符，一个是写入端的描述符，一个是输出端的描述符，可以通过这两个描述符往管道写入或者读取数据。

如果想要实现两个进程通过管道来通信，则需要让创建管道的进程fork子进程，这样子进程们就拥有了父进程的文件描述符，这样子进程之间也就有了对同一管道的操作。

缺点：

半双工通信，一条管道只能一个进程写，一个进程读。

一个进程写完后，另一个进程才能读，反之同理。

### 消息队列：

管道的通信方式效率是低下的，不适合进程间频繁的交换数据。这个问题，消息队列的通信方式就可以解决。A进程往消息队列写入数据后就可以正常返回，B进程需要时再去读取就可以了，效率比较高。

而且，数据会被分为一个一个的数据单元，称为消息体，消息发送方和接收方约定好消息体的数据类型，不像管道是无格式的字节流类型，这样的好处是可以边发送边接收，而不需要等待完整的数据。

但是也有缺点，每个消息体有一个最大长度的限制，并且队列所包含消息体的总长度也是有上限的，这是其中一个不足之处。

另一个缺点是消息队列通信过程中存在用户态和内核态之间的数据拷贝问题。进程往消息队列写入数据时，会发送用户态拷贝数据到内核态的过程，同理读取数据时会发生从内核态到用户态拷贝数据的过程。

### 共享内存：

共享内存解决了消息队列存在的内核态和用户态之间数据拷贝的问题。

现代操作系统对于内存管理采用的是虚拟内存技术，也就是说每个进程都有自己的虚拟内存空间，虚拟内存映射到真实的物理内存。共享内存的机制就是，不同的进程拿出一块虚拟内存空间，映射到相同的物理内存空间。这样一个进程写入的东西，另一个进程马上就能够看到，不需要进行拷贝。

（这里的物理内存貌似不是内核空间的内存？）

### 信号量：

当使用共享内存的通信方式，如果有多个进程同时往共享内存写入数据，有可能先写的进程的内容被其他进程覆盖了。

因此需要一种保护机制，信号量本质上是一个整型的计数器，用于实现进程间的互斥和同步。

信号量代表着资源的数量，操作信号量的方式有两种：

P操作：这个操作会将信号量减一，相减后信号量如果小于0，则表示资源已经被占用了，进程需要阻塞等待；如果大于等于0，则说明还有资源可用，进程可以正常执行。

V操作：这个操作会将信号量加一，相加后信号量如果小于等于0，则表明当前有进程阻塞，于是会将该进程唤醒；如果大于0，则表示当前没有阻塞的进程。

（1）信号量实现互斥：

信号量初始化为1

进程 A 在访问共享内存前，先执行了 P 操作，由于信号量的初始值为 1，故在进程 A 执行 P 操作后信号量变为 0，表示共享资源可用，于是进程 A 就可以访问共享内存。

若此时，进程 B 也想访问共享内存，执行了 P 操作，结果信号量变为了 -1，这就意味着临界资源已被占用，因此进程 B 被阻塞。

直到进程 A 访问完共享内存，才会执行 V 操作，使得信号量恢复为 0，接着就会唤醒阻塞中的线程 B，使得进程 B 可以访问共享内存，最后完成共享内存的访问后，执行 V 操作，使信号量恢复到初始值 1。

（2）信号量实现同步：

由于多线程下各线程的执行顺序是无法预料的，有些时候我们希望多个线程之间能够密切合作，这时候就需要考虑线程的同步问题。

信号量初始化为0

如果进程 B 比进程 A 先执行了，那么执行到 P 操作时，由于信号量初始值为 0，故信号量会变为 -1，表示进程 A 还没生产数据，于是进程 B 就阻塞等待；

接着，当进程 A 生产完数据后，执行了 V 操作，就会使得信号量变为 0，于是就会唤醒阻塞在 P 操作的进程 B；

最后，进程 B 被唤醒后，意味着进程 A 已经生产了数据，于是进程 B 就可以正常读取数据了。

### 信号：

在Linux中，为了响应各种事件，提供了几十种信号，可以通过kill -l命令查看。

如果是运行在shell终端的进程，可以通过键盘组合键来给进程发送信号，例如使用Ctrl+C产生SIGINT信号，表示终止进程。

如果是运行在后台的进程，可以通过命令来给进程发送信号，例如使用kill -9 PID产生SIGKILL信号，表示立即结束进程。

### Socket：

前面提到的管道，消息队列，共享内存，信号量和信号都是在同一台主机上进行进程间通信，如果想要跨网络和不同主机上的进程进行通信，则需要用到socket。

实际上，Socket不仅可以跨网络和不同主机进行进程间通信，还可以在同一主机进行进程间通信。

Socket是操作系统提供给程序员操作网络的接口，根据底层不同的实现方式，通信方式也不同。

## 内核态和用户态

一、内核态、用户态概念

内核态：也叫内核空间，是内核进程/线程所在的区域。主要负责运行系统、硬件交互。

用户态：也叫用户空间，是用户进程/线程所在的区域。主要用于执行用户程序。

一般的操作系统对执行权限进行分级，分别为用用户态和内核态。

用户态相较于内核态有较低的执行权限，很多操作是不被操作系统允许的，原因简单来说就是用户态出现问题（自己写的bug），也不能让操作系统崩溃呀。

内核态相当于一个介于硬件与应用之间的层，内核有ring 0的权限，可以执行任何cpu指令，也可以引用任何内存地址，包括外围设备, 例如硬盘, 网卡，权限等级最高。

用户态则权利有限，例如在内存分配中，有一部分内存是仅为内核态使用的，用户态code则不允许访问那些内存地址，每个进程只允许访问自己申请到的内存。

而且不允许访问外围设备。另外在执行cpu指令的时候也可以被高优先级抢占。大多数时间各类程序都是执行在用户态下，毕竟内核就是基础而已。

二、内核态和用户态的区别

内核态：运行的代码不受任何限制，CPU可以执行任何指令。

用户态：运行的代码需要受到CPU的很多检查，不能直接访问内核数据和程序，也就是说不可以像内核态线程一样访问任何有效地址。

操作系统在执行用户程序时，主要工作在用户态，只有在其执行没有权限完成的任务时才会切换到内核态。

三、为什么要区分内核态和用户态

保护机制。防止用户进程误操作或者是恶意破坏系统。内核态类似于C++的私有成员，只能在类内访问，用户态类似于公有成员，可以随意访问。

四、用户态切换到内核态的方式

1、系统调用（主动）

由于用户态无法完成某些任务，用户态会请求切换到内核态，内核态通过为用户专门开放的中断完成切换。

2、异常（被动）

在执行用户程序时出现某些不可知的异常，会从用户程序切换到内核中处理该异常的程序，也就是切换到了内核态。

3、外围设备中断（被动）

外围设备发出中断信号，当中断发生后，当前运行的进程暂停运行，并由操作系统内核对中断进程处理，如果中断之前CPU执行的是用户态程序，就相当于从用户态向内核态的切换。

## 进程死锁的原因

答案：资源竞争及进程推进顺序非法

一、死锁的定义：

死锁 :是指两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去。

二、原因：

（1）因为系统资源不足

（2）资源分配不当等

（3）进程运行推进顺序不合适

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

三、条件：

（1）互斥条件：一个资源每次只能被一个进程使用。

（2）不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。

（3）请求与保持条件：一个进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：若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

死锁的解除与预防：

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。

其中最简单的方法就是线程都是以同样的顺序加锁和释放锁，也就是破坏了第四个条件。

四、编写死锁程序：

public class TestDeadLock {

private static Object ob1 = new Object();

private static Object ob2 = new Object();

public static void main(String[] args){

//线程0

new Thread(new Runnable(){

@Override

public void run() {

synchronized (ob1) {

System.out.println(Thread.currentThread().getName()+"获得ob1的锁");

try {

Thread.sleep(1000);

synchronized(ob2){

System.out.println(Thread.currentThread().getName()+"获得ob2的锁");

}

System.out.println(Thread.currentThread().getName()+"结束");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

//线程1

new Thread(new Runnable(){

@Override

public void run() {

synchronized (ob2) {

System.out.println(Thread.currentThread().getName()+"获得ob2的锁");

synchronized (ob1) {

System.out.println(Thread.currentThread().getName() + "获得ob1的锁");

}

System.out.println(Thread.currentThread().getName() + "结束");

}

}

}).start();

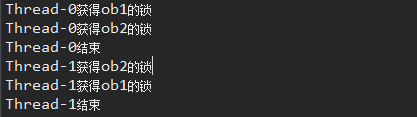
}

}

运行结果如下：

IMG_256

而不是期望的Thread-0和Thread-1能如下图一样按顺序推进执行



这是因为，Thread-0获得到ob1锁之后，sleep了1s，这个时候Thread-1已经获得了ob2的锁，而此时Thread-1想再获得ob1的锁，却发现被Thread-0占有了，同样1s后Thread-0想再获得ob2的锁，却发现被Thread-1占有了，两个线程各自占有的锁恰好是对方继续执行下去所必需的，Thread-0等待Thread-1释放ob2的锁，Thread-1等待Thread-0释放ob1的锁，相互循环，造成了死锁。

## 死锁的处理

答案：鸵鸟策略、预防策略、避免策略、检测与解除死锁

预防死锁：通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件，来防止死锁的发生。

避免死锁：在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免死锁的发生。

检测死锁：允许系统在运行过程中发生死锁，但可设置检测机构及时检测死锁的发生，并采取适当措施加以清除。

解除死锁：当检测出死锁后，便采取适当措施将进程从死锁状态中解脱出来。

6.1 预防死锁

破坏“互斥”条件:

就是在系统里取消互斥。若资源不被一个进程独占使用，那么死锁是肯定不会发生的。但一般来说在所列的四个条件中，“互斥”条件是无法破坏的。

破坏“占有并等待”条件:

破坏“占有并等待”条件，就是在系统中不允许进程在已获得某种资源的情况下，申请其他资源。即要想出一个办法，阻止进程在持有资源的同时申请其他资源。

方法一：一次申请所需的全部资源，即 “ 一次性分配”。

方法二：要求每个进程提出新的资源申请前，释放它所占有的资源。即“先释放后申请”。

破坏“不可抢占”条件：

破坏“不可抢占”条件就是允许对资源实行抢夺。

方法一：占有某些资源的同时再请求被拒绝，则该进程必须释放已占有的资源，如果有必要，可再次请求这些资源和另外的资源。

方法二：设置进程优先级，优先级高的可以抢占资源。

破坏“循环等待”条件：

将系统中的所有资源统一编号，所有进程必须按照资源编号顺序提出申请。

6.2避免死锁

加锁顺序：线程按照一定的顺序加锁。

加锁时限：线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁。

死锁检测：每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

6.3 检测死锁

一般来说，由于操作系统有并发，共享以及随机性等特点，通过预防和避免的手段达到排除死锁的目的是很困难的。这需要较大的系统开销，而且不能充分利用资源。为此，一种简便的方法是系统为进程分配资源时，不采取任何限制性措施，但是提供了检测和解脱死锁的手段：能发现死锁并从死锁状态中恢复出来。因此，在实际的操作系统中往往采用死锁的检测与恢复方法来排除死锁。

例如：

进程P1占有资源R1而申请资源R2，进程P2占有资源R2而申请资源R1，按循环等待条件，进程和资源形成了环路，所以系统是死锁状态。进程P1，P2是参与死锁的进程。

下面我们再来看一看死锁检测算法。算法使用的数据结构是如下这些：

占有矩阵A：n\*m阶，其中n表示并发进程的个数，m表示系统的各类资源的个数，这个矩阵记录了每一个进程当前占有各个资源类中资源的个数。

申请矩阵R：n\*m阶，其中n表示并发进程的个数，m表示系统的各类资源的个数，这个矩阵记录了每一个进程当前要完成工作需要申请的各个资源类中资源的个数。

空闲向量T：记录当前m个资源类中空闲资源的个数。

完成向量F：布尔型向量值为真（true）或假（false），记录当前n个并发进程能否进行完。为真即能进行完，为假则不能进行完。

临时向量W：开始时W：=T。

算法步骤：

（1）W：=T，

对于所有的i=1，2，…，n，

如果A[i]=0，则F[i]：=true；否则，F[i]：=false

（2）找满足下面条件的下标i：

F[i]：=false并且R[i]〈=W

如果不存在满足上面的条件i，则转到步骤（4）。

（3）W：=W+A[i]

F[i]：=true

转到步骤（2）

（4）如果存在i，F[i]：=false，则系统处于死锁状态，且Pi进程参与了死锁。什么时候进行死锁的检测取决于死锁发生的频率。如果死锁发生的频率高，那么死锁检测的频率也要相应提高，这样一方面可以提高系统资源的利用率，一方面可以避免更多的进程卷入死锁。如果进程申请资源不能满足就立刻进行检测，那么每当死锁形成时即能被发现，这和死锁避免的算法相近，只是系统的开销较大。为了减小死锁检测带来的系统开销，一般采取每隔一段时间进行一次死锁检测，或者在CPU的利用率降低到某一数值时，进行死锁的检测。

6.4 解除死锁

资源剥夺法：挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但应防止被挂起的进程长时间得不到资源，而处于资源匮乏的状态。

撤销进程法：强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。撤销的原则可以按进程优先级和撤销进程代价的高低进行。

进程回退法：让一（多）个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点。

## 操作系统中进程调度策略有哪几种？

答案：FCFS(先来先服务)，优先级，时间片轮转，多级反馈

一.先来先服务调度算法

先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

二.短作业(进程)优先调度算法

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

三.高优先权优先调度算法

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

1. 非抢占式优先权算法：

在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2. 抢占式优先权调度算法：

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j 的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

3.容易出现优先级倒置现象：

优先级反转是指一个低优先级的任务持有一个被高优先级任务所需要的共享资源。高优先任务由于因资源缺乏而处于受阻状态，一直等到低优先级任务释放资源为止。而低优先级获得的CPU时间少，如果此时有优先级处于两者之间的任务，并且不需要那个共享资源，则该中优先级的任务反而超过这两个任务而获得CPU时间。如果高优先级等待资源时不是阻塞等待，而是忙循环，则可能永远无法获得资源，因为此时低优先级进程无法与高优先级进程争夺CPU时间，从而无法执行，进而无法释放资源，造成的后果就是高优先级任务无法获得资源而继续推进。

4.优先级反转案例解释：

不同优先级线程对共享资源的访问的同步机制。优先级为高和低的线程tall和线程low需要访问共享资源，优先级为中等的线程mid不访问该共享资源。当low正在访问共享资源时，tall等待该共享资源的互斥锁，但是此时low被mid抢先了，导致mid运行tall阻塞。即优先级低的线程mid运行，优先级高的tall被阻塞。

5.优先级倒置解决方案：

1）设置优先级上限，给临界区一个高优先级，进入临界区的进程都将获得这个高优先级，如果其他试图进入临界区的进程的优先级都低于这个高优先级，那么优先级反转就不会发生。

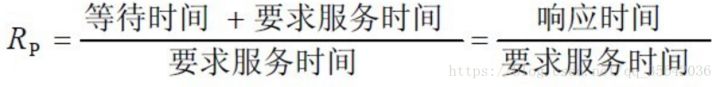
2）优先级继承，当一个高优先级进程等待一个低优先级进程持有的资源时，低优先级进程将暂时获得高优先级进程的优先级别，在释放共享资源后，低优先级进程回到原来的优先级别。嵌入式系统VxWorks就是采用这种策略。

这里还有一个八卦，1997年的美国的火星探测器（使用的就是vxworks)就遇到一个优先级反转问题引起的故障。简单说下，火星探测器有一个信息总线，有一个高优先级的总线任务负责总线数据的存取，访问总线都需要通过一个互斥锁（共享资源出现了）；还有一个低优先级的，运行不是很频繁的气象搜集任务，它需要对总线写数据，也就同样需要访问互斥锁；最后还有一个中优先级的通信任务，它的运行时间比较长。平常这个系统运行毫无问题，但是有一天，在气象任务获得互斥锁往总线写数据的时候，一个中断发生导致通信任务被调度就绪，通信任务抢占了低优先级的气象任务，而无巧不成书的是，此时高优先级的总线任务正在等待气象任务写完数据归还互斥锁，但是由于通信任务抢占了CPU并且运行时间比较长，导致气象任务得不到CPU时间也无法释放互斥锁，本来是高优先级的总线任务也无法执行，总线任务无法及时执行的后果被探路者认为是一个严重错误，最后就是整个系统被重启。Vxworks允许优先级继承，然而遗憾的工程师们将这个选项关闭了。

3）第三种方法就是临界区禁止中断，通过禁止中断来保护临界区，采用此种策略的系统只有两种优先级：可抢占优先级和中断禁止优先级。前者为一般进程运行时的优先级，后者为运行于临界区的优先级。火星探路者正是由于在临界区中运行的气象任务被中断发生的通信任务所抢占才导致故障，如果有临界区的禁止中断保护，此一问题也不会发生。

四、高响应比优先调度算法：

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：



在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

五、时间片轮转法：

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

六、多级反馈队列调度算法：

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。

(2)当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n队列便采取按时间片轮转的方式运行。

(3)仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

**类的静态成员和非静态成员有何区别？**

答案：类的静态成员每个类只有一个，非静态成员每个对象一个

## 纯虚函数

### 概念

（1）定义一个函数为纯虚函数，才代表函数没有被实现。

（2）定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加 =0:

virtual void funtion1()=0

### 引入原因

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。

声明了纯虚函数的类是一个抽象类。所以，用户不能创建类的实例，只能创建它的派生类的实例。

纯虚函数最显著的特征是：它们必须在继承类中重新声明函数（不要后面的＝0，否则该派生类也不能实例化），而且它们在抽象类中往往没有定义。

定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。

纯虚函数的意义，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的默认实现。所以类纯虚函数的声明就是在告诉子类的设计者，“你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它”。

### 抽象类

抽象类是一种特殊的类，它是为了抽象和设计的目的而建立的，它处于继承层次结构的较上层。

（1）抽象类的定义：称带有纯虚函数的类为抽象类。

（2）抽象类的作用：抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

（3）使用抽象类时注意：

抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

抽象类是不能定义对象的。

## 数组和链表的区别

答案：数组：数据顺序存储，固定大小 链表：数据可以随机存储，大小可动态改变

作为[线性表](https://so.csdn.net/so/search?q=%E7%BA%BF%E6%80%A7%E8%A1%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Time_Limit/article/details/_blank)的两种存储方式 —— 链表和数组，这对相爱相杀的好基友有着各自的优缺点。接下来，我们梳理一下这两种方式。

## main函数返回1

int main()

{

int x=3;

printf("%d",x);

return 1;

} 问函数既然不会被其它函数调用，为什么要返回1？

答案：mian中，c标准认为0表示成功，非0表示错误。具体的值是某中具体出错信息

## 绝对地址

要对绝对地址0x100000赋值，我们可以用 (unsigned int\*)0x100000 = 1234; 那么要是想让程序跳转到绝对地址是0x100000去执行，应该怎么做？

答案：\*((void (\*)( ))0x100000 ) ( ); 首先要将0x100000强制转换成函数指针,即: (void (\*)())0x100000 然后再调用它: \*((void (\*)())0x100000)(); 用typedef可以看得更直观些: typedef void(\*)() voidFuncPtr; \*((voidFuncPtr)0x100000)();

函数指针：一个指针变量，存放的是函数的地址

定义形式：

类型 （\*指针变量名）（参数列表）；

如：

int (\*p)(int x,int y);

## 数组求出数据的元素个数

已知一个数组table，用一个宏定义，求出数据的元素个数

答案：#define NTBL #define NTBL (sizeof(table)/sizeof(table[0]))