**1.1 进程线程的基本概念**

**1.1.1什么是进程，线程？也就是问彼此有什么区别？**

**答：**进程是资源（CPU、内存等）分配的基本单位，线程是CPU调度和分配的基本单位（程序执行的最小单位）。

1）当我们运行一个程序的时候，系统就会创建一个**进程**，并分配地址空间和其他资源，最后把进程加入就绪队列直到分配到CPU时间就可以正式运行了。

2）**线程**是进程的一个执行流，有一个初学者可能误解的概念，进程就像一个容器一样，包括程序运行的程序段、数据段等信息，但是进程其实是**不能用来运行代码**的，真正运行代码的是进程里的线程。

3）那么，来看看我们最熟悉的main()函数，我们既可以认为这是一个进程，也可以认为是一个线程。我们都知道，在C/C++中main函数是程序入口，所以准确来说main函数是程序的**主线程**。然而很神奇的地方在于，当系统在执行main函数的时候，main函数又是一个独立的进程，我们可以在main函数里创建子进程，也可以创建子线程。

4）在main函数里创建的多个子线程中，每个线程有自己的**堆栈**和**局部变量**，但多个线程也可共享同个进程下的所有共享资源，因此我们经常可以创建多个线程实现并发操作，实现更加复杂的功能。

**示例：**我们看一个实际例子来加强理解。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include  #include  #include  #include  **int** g\_cnt = 0;  //全局变量  **int** \* thread(**void** \* arg)  {  **int** m\_cnt = 0;      m\_cnt = 5;      g\_cnt++;  **return** 0;  }  **int** main(**void**)  {  **int** err = 0;      pthread\_t tid;  **int** m\_cnt = 0;      err=pthread\_create(&tid, NULL, thread, NULL);  //创建子线程  **if** (0 != err)   //检验是否创建成功      {          printf("can't creat thread: %s\n", strerror(err));      }  **while**(g\_cnt == 0)      {          usleep(300);   //延迟300毫秒，让子线程运行一会儿      }      printf("g\_cnt = %d, m\_cnt = %d\n", g\_cnt,  m\_cnt);  **return** 0;  } |

    我们可以看出main函数是一个主线程，开始执行程序，同时main函数又是一个进程，我们可以创建子线程thread（），子线程有自己的堆栈和局部变量，同时又与主线程共享全局变量，这也就是为何输出结果显示子线程改变了全局变量g\_cnt,但没有改变main函数里的同名局部变量m\_cnt。

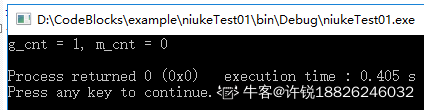


图1-1运行结果

     还有一个关键点需要注意：我们可以看到main函数有一个while循环，一开始 g\_cnt == 0，程序进入while循环后就不能做其他事情，但是子线程thread不受影响，仍然可以独立于main函数，自己做自己的事情。

**1.1.2 多进程、多线程的优缺点**

**解析：**为了理解多进程、多线程各自的优缺点之前，我们需要先了解进程和线程最大的区别和联系，一个进程由PCB（进程控制块）、数据段、代码段组成，进程本身不可以运行程序，而是像一个容器一样，先创建出一个主线程，分配给主线程一定的系统资源，这时候就可以在主线程开始实现各种功能。当我们需要实现更复杂的功能时，可以在主线程里创建多个子线程，跟人多好干活的道理一样，多个线程在同一个进程里，利用这个进程所拥有的系统资源合作完成某些功能。

理解了这些知识点，再来理解各自优缺点就很容易了。

**答**：

1）多进程更健壮，一个进程死了不影响其他进程，子进程死了也不会影响到主进程，毕竟系统会给每个进程分配独立的系统资源。多线程比较脆弱，一个线程崩溃很可能影响到整个程序，因为多个线程是在一个进程里一起合作干活的。

2） 进程性能大于线程，每个进程独立地址空间和资源，而多个线程是一起共享了同个进程里的空间和资源，结果就很明显了，线程的性能上限一定比不上进程。

3） 正因为进程性能大于线程。所以这也引发了另一重要知识点，创建多进程的系统花销远大于创建多线程。

4）多进程通讯因为需要跨越进程边界，不适合大量数据的传送，更适合小数据或者密集数据的传送。而多线程无需跨越进程边界，适合各线程间大量数据的传送，甚至还有很重要的一点，多线程可以共享同一进程里的共享内存和变量哦。

5） 多进程逻辑控制比多线程复杂，需要与主进程做好交互。根据上面几点，我们不难知道多进程是“要用来做大事”的，而多线程是“各自做件小事，合作完成大事”。所以要做大事自然就需要更复杂的逻辑控制，不像做小事那么目标明显。

6）虽然多线程逻辑控制比较简单，但是却需要复杂的线程同步和加锁控制等机制。

7）最后的一点，可能比较少见，我们可以通过增加CPU的数量来增加进程的数量，但增加不了线程的数量，即增加CPU无法提高线程数量，线程数量由进程的空间资源和线程本身栈大小确定，详情见1.1.6小节。

**1.1.3 什么时候用进程，什么时候用线程**

**解析：**还是同一个思想，进程是“要用来做大事”的，而线程是“各自做件小事，合作完成大事”，结合上节新鲜出炉的优缺点我们就很好理解什么时候用进程或者线程了。

**答：**

1）创建和销毁较频繁使用线程，因为创建进程花销大嘛。

2）需要大量数据传送使用线程，因为多线程切换速度快，不需要跨越进程边界。

3）并行操作使用线程。线程是为了实现并行操作的一个手段，也就是刚才说的需要多个并行操作“合作完成大事”，当然是使用线程啦。

4）最后可以总结为：安全稳定选进程；快速频繁选线程；

**示例**：我可以举个本科时自己做的播放器的小例子。我们知道播放器最常见的功能有上一部、下一部。

    当时是调用外部程序mplayer来播放视频，一旦某个进程调用外部程序，我们就失去了对这个进程的控制，回不到这个进程了。（科普一下，因此出现一种子进程是不需要复制父进程的内存和数据的，只是为了方便调用外部程序）

    所以我的程序框架是，因为播放视频需要的系统资源较大，所以在main函数里先创建一个子**进程**用来执行播放视频的代码，此时子进程播放视频不影响main函数。最后，main函数执行while（1）死循环一直等待有没有上一部、下一部按键按下，很明显，此时main函数在死循环里无法检验按键按下情况，而检验按键是一件容易且与主线程main函数息息相关的事情，此时就创建子**线程**来检验按键即可。

    最终，子线程检验到“下一部”按键按下，告知主线程main函数，main函数就立刻再次创建子进程再次调用外部程序mplayer播放下一个视频。

**1.1.4 多进程、多线程同步（通讯）的方法**

**答：**当我们在使用系统编程时，就会遇到多进程、多线程编程，所以必须知道在多个进程、多个线程之间都有什么常见的通讯机制，这也是嵌入式面试中高频问题之一。

进程间通讯：

（1）管道/无名管道（2）信号（3）共享内存（4）消息队列（5）信号量（6）socket

注意：临界区则是一种概念，指的是访问公共资源的程序片段，并不是一种通信方式。

线程通讯：

（1）信号量（2）读写锁（3）条件变量（4）互斥锁（5）自旋锁

**提问：互斥锁与信号量的区别？**

**答**：互斥锁用于线程的互斥，信号量用于线程的同步。这是互斥锁和信号量的根本区别，也就是互斥和同步之间的区别。同时互斥锁的作用域仅仅在于线程，信号量可以作用于线程和进程。

**1.1.5 进程的空间模型**

**解析：**32位系统中，当系统运行一个程序，就会创建一个进程，系统为其分配4G的虚拟地址空间，其中0-3G是用户空间，3-4G是内核空间，具体如图1-2，内核空间是受保护的，用户不能对该空间进行读写操作，否则可能出现段错误。其中栈空间有向下的箭头，代表数据地址增加的空间是往下的，新的数据的地址的值反而更小，堆空间则是往上。

**感谢**：牛友“后生阿风”指出在.data和.text之间少了一个.rodata数据段，用于存放C中的字符串和#define定义的常量。



图1-2 Linux内核空间模型

1）栈区：由编译器自动分配和释放，存放函数的参数值（形参）、局部变量（int a =1；还有指针变量）等，其操作方式类似于数据结构中的栈，先进后出。

2）堆区：一般由程序员分配和释放，若程序员不释放，可能会造成内存泄漏，程序结束的时候可能由操作系统回收，注意它与数据结构中的堆是两回事，分配方式类似于链表。

3）全局区（静态区）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域(.data)，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域(.bss)，程序结束后系统释放。

4)文字常量区：常量字符串放在这里，程序结束后有系统释放。

5)程序代码区(.text)：存放函数体的二进制代码。

栈的空间有限，堆是很大的自由存储区，程序在编译期对变量和函数分配内存都在栈上进行，且程序运行过程中函数调用时参数的传递也是在栈上进行。

**注意：**64位操作系统下的虚拟内存空间大小：地址空间大小不是2^32，也不是2^64，而一般是2^48。因为并不需要2^64那么大的寻址空间，过大的空间只会造成资源的浪费。所以64位Linux一般使用48位表示虚拟空间地址，40位标识物理地址。0x0000000000000000~0x00007fffffffffff表示用户空间， 0xFFFF800000000000~ 0xFFFFFFFFFFFFFFFF表示内核空间，共提供256TB(2^48)的寻址空间。

**1.1.6 一个进程可以创建多少线程，和什么有关**

**答：**一个进程创建线程的个数由虚拟内存和分配给线程的调用栈大小决定。

**解析：**

由1.1.5小节我们已经知道创建一个进程，系统会分配4G的虚拟内存，其中1G是内核空间，只有3G是用户空间，也就是我们可以利用来创建线程的空间大小，一个线程的栈的大小可以通过ulimit -s指令来查看，一般大多是8M-10M。

举个例子，这里不放假设我们创建一个线程的栈需要占用10M内存，因此在3G的空间大概可以创建300个线程。此时如果将线程栈大小增加到20M，那么个数就将减少。

**1.1.7 进程线程的状态转换图，什么时候阻塞，什么时候就绪？**

**解析：**

在此之前，我们先看看一个进程一生中，从蛋生到死亡都有可能出现什么状态。

创建态(New)：一个进程正在被创建，还没到转到就绪状态之前的状态。

就绪态(Ready)：一个进程获得了除CPU时间片之外的一切所需资源，一旦得到CPU时间片调度时即可运行。

运行/执行态(Running)：当一个进程得到CPU调度正在处理机上运行时的状态。

睡眠/挂起态：由于某些资源暂时不可得到而进入“睡眠态”，将进程挂起，等待唤醒。

阻塞/暂停态(Blocked)：一个进程正在等待某一事件而暂停运行时，如等待某资源成为可用，或等待文件读取完成等。

结束/僵尸态(Exit)：一个进程正在从系统中消失时的状态，这是因为进程结束或其它因流产所导致。

死亡态：进程生命周期结束了，将所占用的资源还给系统。

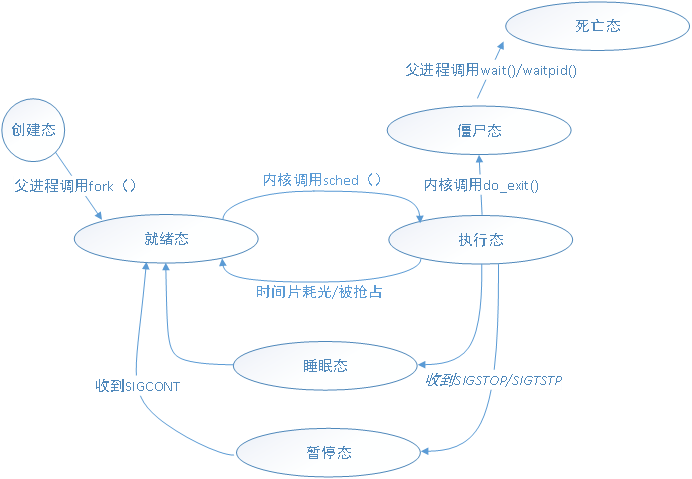


图1-3进程的一生

我们从父进程调用fork（）创建子进程开始讲起，此时子进程处于创建态，此时系统为进程分配地址和资源后将进程加入就绪队列，进入就绪态。就绪态的进程得到CPU时间片调度正式运行，进入执行态。执行态有四种常见结果：

1）当时间片耗光或者被其他进程抢占，则重新进入就绪态，等待下一次CPU时间片；

2）由于某些资源暂时不可得到而进入“睡眠态”（如欲读取的文件为空或者欲获得的某个锁还处于不可获得状态），等待资源可得后再唤醒，唤醒后进入就绪态；

3）收到SIGSTOP/SIGTSTP信号进入暂停态，直到收到SIGCONT信号重新进入就绪态；

4）进程执行结束，通过内核调用do\_exit()进入僵尸态，等待系统回收资源。当父进程调用wait()/waitpid()后接收结束子进程，该进程进入死亡态。

**1.1.8 父进程、子进程的关系以及区别**

**解析：**

我们先来看看子进程会从父进程继承了什么，以及子进程独有的数据：

子进程继承父进程：

      ○用户号UIDs和用户组号GIDs

      ○环境Environment

      ○堆栈

      ○共享内存

      ○打开文件的描述符

      ○执行时关闭（Close-on-exec）标志

      ○信号（Signal）控制设定

      ○进程组号

      ○当前工作目录

      ○根目录

      ○文件方式创建屏蔽字

      ○资源限制

      ○控制终端

子进程独有的：

      ○进程号PID

      ○不同的父进程号

      ○自己的文件描述符和目录流的拷贝

      ○子进程不继承父进程的进程正文（text），数据和其他锁定内存（memory locks）

      ○不继承异步输入和输出

父进程调用fork()以后，克隆出一个子进程，子进程和父进程拥有相同内容的代码段、数据段和用户堆栈。但其实父进程只复制了自己的PCB块，而代码段，数据段和用户堆栈内存空间是与子进程共享的。只有当子进程在运行中出现写操作时，才会产生中断，并为子进程分配内存空间。

在面试前，我们需要记清楚、分清楚几个主要的父子进程共有的资源和子进程独有的资源。

**答：**

子进程从父进程继承的主要有：用户号和用户组号；堆栈；共享内存；目录（当前目录、根目录）；打开文件的描述符；但父进程和子进程拥有独立的地址空间和PID参数、不同的父进程号、自己的文件描述符。

**1.1.9什么是进程上下文、中断上下文**

**解析**：

由1.1.5知道进程空间分为**内核空间**和**用户空间**，即内核功能模块运行在内核空间，而我们编写的应用程序运行在用户空间。其中内核运行在最高权限级别的**内核态**，这个级别有最高权限可以进行所有操作，而应用程序运行在较低级别的**用户态**，内核态和用户态有自己的内存映射，即自己的地址空间。

**进程上下文：**

正是有了不同运行状态的划分，才有了上下文的概念。当我们创建一个进程（例如main函数）需要控制一个外部设备时（比如控制一个LED灯亮），我们编写的在用户空间的代码将通过“**系统调用**（操作系统提供给用户空间的接口函数）”进入内核空间，由内核继续代表我们这个进程运行于内核空间，这时候就涉及上下文的切换。用户空间和内核空间具有不同的地址映射，通用或专用的寄存器组，而用户空间的进程要传递很多变量、参数给内核，内核也要保存用户进程的一些寄存器、变量等，以便系统调用结束后回到用户空间继续执行，所谓的进程上下文，就是一个进程在执行的时候**，CPU的所有寄存器中的值、进程的状态以及堆栈中的内容**，当内核需要切换到另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的进程上下文，以便再次执行该进程时，能够恢复切换时的状态，继续执行。

**中断上下文：**

同理，当由硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的一些变量和参数也要传递给内核，内核通过这些参数进行中断处理，中断上下文就可以理解为硬件传递过来的这些参数和内核需要保存的一些环境，主要是被中断的进程的环境。

那么运行在中断上下文的代码受到限时，有什么不能做的事呢？答案如下：

1、睡眠或者放弃CPU。

      这样做的后果是灾难性的，因为内核在进入中断之前会关闭进程调度，一旦睡眠或者放弃CPU，这时内核无法调度别的进程来执行，系统就会死掉

2、尝试获得信号量

      如果获得不到信号量，代码就会睡眠，会产生和上面相同的情况

3、执行耗时的任务

      中断处理应该尽可能快，因为内核要响应大量服务和请求，中断上下文占用CPU时间太长会严重影响系统功能。

4、访问用户空间的虚拟地址

      因为中断上下文是和特定进程无关的，它是内核代表硬件运行在内核空间，所以在终端上下文无法访问用户空间的虚拟地址。

**1.2 并发，同步，异步，互斥，阻塞，非阻塞的理解**

**1.2.1 并发，同步，异步，互斥，阻塞，非阻塞的概念**

**并发**：在操作系统中，同个处理机上有多个程序同时运行即并发。并发可分为同步和互斥。

**1）同步、互斥：**

互斥：分布在不同进程之间的若干程序片断，规定当某个进程运行其中一个程序片段时，其它进程就不能运行它们之中的任一程序片段，只能等到该进程运行完这个程序片段后才可以运行。如有同一个资源同一时间只有一个访问者可以进行访问，其他访问者需要等前一个访问者访问结束才可以开始访问该资源，但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

同步：分布在不同进程之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。所以同步就是在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。

总结：同步是一种更为复杂的互斥，而互斥是一种特殊的同步。

**2）同步、异步：**

同步：同步就是顺序执行，执行完一个再执行下一个，需要等待、协调运行。

异步：异步和同步是相对的，异步就是彼此独立,在等待某事件的过程中继续做自己的事，不需要等待这一事件完成后再工作。

**注意：**

1）线程是实现异步的一个方式。可以在主线程创建一个新线程来做某件事，此时主线程不需等待子线程做完而是可以做其他事情。

2）异步和多线程并不是一个同等关系。异步是最终目的,多线程只是我们实现异步的一种手段。

**3）阻塞，非阻塞：**

    阻塞和非阻塞是当进程在访问数据时，根据IO操作的就绪状态不同而采取的不同处理方式，比如主程序调用一个函数要读取一个文件的内容，阻塞方式下主程序会等到函数读取完再继续往下执行，非阻塞方式下，读取函数会立刻返回一个状态值给主程序，主程序不等待文件读取完就继续往下执行。一般来说可以分为：同步阻塞，同步非阻塞，异步阻塞，异步非阻塞。

4**）同步阻塞，同步非阻塞，异步阻塞，异步非阻塞：**

以发送方发出请求要接收方读取某文件内容为例。

同步阻塞：发送方发出请求后一直等待（同步），接收方开始读取文件，如果不能马上得到读取结果就一直等，直到获取读取结果再响应发送发，等待期间不可做其他操作（阻塞）。

同步非阻塞：发送方发出请求后一直等待（同步），接收方开始读取文件，如果不能马上的得到读取结果，就立即返回，接收方继续去做其他事情。此时并未响应发送发，发送方一直在等待。直到IO操作（这里是读取文件）完成后，接收方获得读取结果响应发送方，接收方才可以进入下一次请求过程。（实际不应用）

异步阻塞：发送方发出请求后，不等待响应，继续其他工作（异步），接收方读取文件如果不能马上得到结果，就一直等到返回结果后，才响应发送方，期间不能进行其他操作（阻塞）。（实际不应用）

异步非阻塞：发送方发出请求后，不等待响应，继续其他工作（异步），接收方读取文件如果不能马上得到结果，也不等待，而是马上返回取做其他事情。当IO操作（读取文件）完成以后，将完成状态和结果通知接收方，接收方在响应发送方。（效率最高）

**总结：**

1）同步与异步是对应的，它们是线程之间的关系，两个线程之间要么是同步的，要么是异步的。

2）阻塞与非阻塞是对同一个线程来说的，在某个时刻，线程要么处于阻塞，要么处于非阻塞。

3）阻塞是使用同步机制的结果，非阻塞则是使用异步机制的结果。

**1.2.2 什么是线程同步和互斥**

线程同步：每个线程之间按预定的先后次序进行运行，协同、协助、互相配合。可以理解成“你说完，我再做”。有了线程同步，每个线程才不是自己做自己的事情，而是协同完成某件大事。

线程互斥：当有若干个线程访问同一块资源时，规定同一时间只有一个线程可以得到访问权，其它线程需要等占用资源者释放该资源才可以申请访问。线程互斥可以看成是一种特殊的线程同步。

**1.2.3 线程同步与阻塞的关系？同步一定阻塞吗？阻塞一定同步吗？**

    解析：这也是网上经常看到的问题之一了，同步是个过程，阻塞是线程的一种状态：当多个线程访问同一资源时，规定同一时间只有一个线程可以进行访问，所以后访问的线程将阻塞，等待前访问的线程访问完。

    注意：线程同步不一定发生阻塞！线程同步的时候，需要协调推进速度，只有当访问同一资源出现互相等待和互相唤醒会发生阻塞。而阻塞了一定是同步，后访问的等待获取资源，线程进入阻塞状态，借以实现多线程同步的过程。

**1.3 孤儿进程、僵尸进程、守护进程的概念**

**1.3.1 基本概念**

孤儿进程：当父进程退出后它的子进程还在运行，那么这些子进程就是孤儿进程。孤儿进程将被init进程所收养，并由init进程对它们完成状态收集工作。

僵尸进程：当子进程退出后而父进程并未接收结束子进程（如调用waitpid获取子进程的状态信息），那么子进程仍停留在系统中，这就是僵尸进程。

守护进程：是在后台运行不受终端控制的进程（如输入、输出等）。网络服务大部分就是守护进程。

**1.3.2 如何创建守护进程：**

1）创建子进程，父进程退出：因为守护进程是在后台运行不受终端控制的进程，父进程退出后控制台就以为该程序结束了，我们就可以在子进程进行自己的任务，同时用户仍可以在控制台输入指令，从而在形式上做到了与控制台脱离。

2、在子进程中创建新的会话(脱离控制终端)：使用系统函数setsid()来创建一个新的会话，并担任该会话组的组长，摆脱原会话的控制==>摆脱原进程的控制==>摆脱原控制台的控制。

3、改变当前目录为根目录：1.1.7小节知道子进程继承父进程的目录信息，但进程运行时对当前目录下的文件系统不能卸载，这会有很多隐藏的麻烦，建议使用根目录作为当前目录，当然也可以使用其他目录。

4、重设文件权限掩码，关闭文件描述符：子进程还继承父进程文件权限掩码，即屏蔽掉文件权限中的对应位。此时子进程需将其重置为0，即在此时有大的权限，从而提高该守护进程灵活度。最后，关系从父进程继承的已经打开的文件描述符，如不进行关闭将造成浪费资源以及子进程所有文件系统无法卸载等错误。

代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | **int** main(**int** argc, **const** **char** \*argv[])  {      pid\_t pid;      pid = fork();      If(pid < 0)  //创建子进程失败      {  **perror**("fail to fork");  **exit**(0);      }**else** **if**(pid > 0){ //父进程退出  **exit**(0);      }**else**{   //进入子进程          setsid();  //创建新会话          umask(0); //重置文件权限掩码          pid = fork();  **if**(pid != 0)          {  **exit**(0);          }          chdir("/");  //设置当前目录为根目录  **int** maxfd = getdtablesize();  **while**(maxfd--)          {              close(maxfd);  //关闭文件描述符          }  **while**(1)          {              syslog(LOG\_INFO,"im deamon\n");              sleep(1);          }      }  **return** 0;  } |

可以发现子进程里再次创建了一个子进程，虽非必要，但却是对守护进程进行一点优化：

**第一次fork：**这里第一次fork的作用在shell终端里造成一个程序已经运行完毕的假象，同时创建新会话的进程不能是进程组组长，所以父进程是进程组组长是不能创建新会话的，需要子进程中执行。所以到这里子进程便成为了一个新会话组的组长啦。

**第二次fork：**第二次fork可以保证不会因为错误操作重新打开终端，因为只有会话组组长可以打开一个终端，再第二次fork后的子进程就不是会话组组长啦。

**1.3.3 正确处理孤儿进程、僵尸进程的方法**

孤儿进程的处理：

孤儿进程也就是没有父进程的进程，孤儿进程的处理就由进程号为1的Init进程负责，就像一个福利院一样，专门负责处理孤儿。当有孤儿进程需要处理的时候，系统就把孤儿进程的父进程设置为init，而init进程会循环地wait()它的已经退出的子进程。因此孤儿进程并不会有什么危害。

僵尸进程的处理：

如果父进程一直不调用wait/waitpid函数接收子进程，那么子进程就一直保存在系统里，占用系统资源，因此如果僵尸进程数量太多，那么就会导致系统空间爆满，无法创建新的进程，严重系统工作，因此僵尸进程需要好好处理。

正确的处理方式可以这样子：系统规定，子进程退出后，父进程会自动收到SIGCHLD信号。因此我们需要在父进程里重置signal函数。每当子进程退出，父进程都会收到SIGCHLD信号，故通过signal函数，重置信号响应函数。代码和注释如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **void**\* handler(**int** sig)  {  **int** status;  **if**(waitpid(-1, &status, WNOHANG) >= 0)      {  **printf**("child is die\n");      }  }  **int** main()  {  **signal**(SIGCHLD, handler);  **int** pid = fork();  **if**(pid > 0)  //父进程循环等待      {  **while**(1)          {              sleep(2);          }        }**else** **if**(0 == pid){   //子进程说自己die后就结束生命周期，之后父进程就收到SIGCHLD                            //信号调用handler函数接收结束子进程，打印child is die。  **printf**("i am child, i die\n");      }  } |

##注意##：handler函数里不能使用wait（）函数，比如同一时间有5个子进程都要结束了，向父进程发送SIGCHLD信号，但父进程此时就在处理其中一个，在处理结束前，收到的其他SIGCHLD信号会忽略，导致漏掉部分子进程没有处理结束。

**2.1 c和c++区别、概念相关面试题**

**2.1.1 new和malloc的区别⭐⭐⭐⭐⭐**

**解析：这个题目是我多次面试时被问到的最高频问题之一，建议各位童鞋必须弄懂。**

**答：**

1）new、delete是C++中独有的操作符，而malloc和free是C/C++中的标准库函数。

2）使用new创建对象在分配内存的时候会自动调用构造函数，同时也可以完成对对象的初始化，同理要记得delete也能自动调用析构函数。因为malloc和 free是库函数而不是运算符，不在编译器控制范围之内，所以不能够自动调用构造函数和析构函数。也就是mallloc只是单纯地为变量分配内存，free也只是释放变量的内存。

3）new返回的是指定类型的指针，并且可以自动计算所申请内存的大小。而malloc返回的是void\*类型，我们需要强行将其转换为实际类型的指针，并且需要指定好要申请内存的大小，malloc不会自动计算的。

4）C++允许重载new/delete操作符，而malloc和free是一个函数，并不能重载。

5）new内存分配失败时，会抛出bad\_alloc异常。malloc分配内存失败时返回NULL。

6）内存区域：先了解自由存储区和堆，两者不相等于的。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配。new操作符从自由存储区上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。

**2.1.2 malloc的底层实现⭐⭐⭐⭐**

**感谢**：感谢牛友“杰兮”反馈，要更加深刻的了解C++内存管理，可参考侯捷老师的《C++内存管理》，侯捷老师就不用多说了，多本著作翻译者，《effective C++》《more effective C++》等，著有《STL源码剖析》，巨佬。以下解析是我在网上所查阅整理的。

**解析**：回顾进程的空间模型，如图2-1所示，

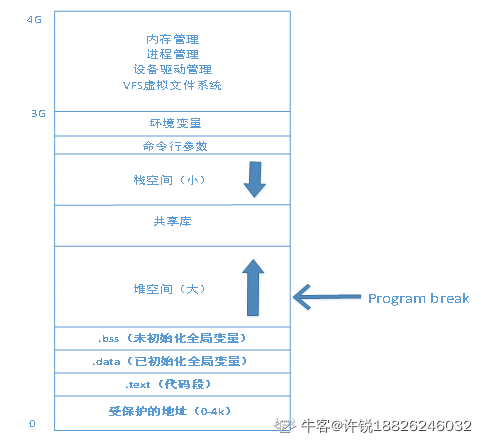


图2-1进程空间示意图

与1.1.5节的图相比，多了一个program break指针，Linux维护一个break指针，这个指针指向堆空间的某个地址。从堆起始地址到break之间的地址空间为映射好的，可以供进程访问；而从break往上，是未映射的地址空间，如果访问这段空间则程序会报错。我们用malloc进行内存分配就是从break往上进行的。

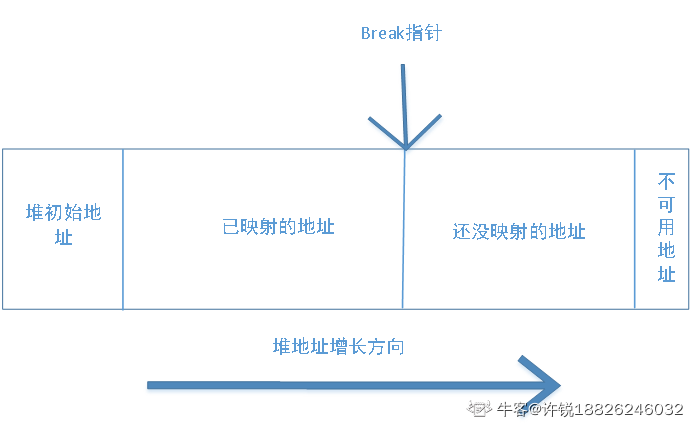


图2-2堆内部机制

获取了break地址，也就是内存申请的初始地址，下面是malloc的整体实现方案：

malloc函数的实质是它有一个将可用的内存块连接为一个长长的列表的所谓空闲链表。 调用malloc（）函数时，它沿着连接表寻找一个大到足以满足用户请求所需要的内存块。 然后，将该内存块一分为二（一块的大小与用户申请的大小相等，另一块的大小就是剩下来的字节）。 接下来，将分配给用户的那块内存存储区域传给用户，并将剩下的那块（如果有的话）返回到连接表上。 调用free函数时，它将用户释放的内存块连接到空闲链表上。 到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段， 那么空闲链表上可能没有可以满足用户要求的片段了。于是，malloc（）函数请求延时，并开始在空闲链表上检查各内存片段，对它们进行内存整理，将相邻的小空闲块合并成较大的内存块。

**2.1.3在1G内存的计算机中能否malloc(1.2G)？为什么？⭐⭐**

**答：**是有可能申请1.2G的内存的。

**解析：**回答这个问题前需要知道malloc的作用和原理，应用程序通过malloc函数可以向程序的虚拟空间申请一块**虚拟地址空间**，与物理内存没有直接关系，得到的是在虚拟地址空间中的地址，之后程序运行所提供的物理内存是由操作系统完成的。

    我们要申请空间的大小为1.2G=2 30 × 1.2 Byte ，转换为十六进制约为 4CCC CCCC ，这个数值还在 unsigned int 的表示范围。 malloc 函数要求的参数正是unsigned int。在当前正在使用的Windows环境中，可申请的最大空间超过1.9G。实际上，具体的数值会受到操作系统版本、程序本身的大小、用到的动态/共享库数量、大小、程序栈数量、大小等的影响，甚至每次运行的结果都可能存在差异，因为有些操作系统使用了一种叫做随机地址分布的技术，使得进程的堆空间变小。感兴趣的读者可以去研究操作系统中的相关内容。

    综上，是有可能通过malloc( size\_t ) 函数调用申请超过该机器物理内存大小的内存块的。

**2.1.4指针与引用的相同和区别；如何相互转换？⭐⭐⭐⭐⭐**

**感谢：感谢寞明牛友反馈区别6，现已加上解释，面试的时候能把这8个区别说出来，面试官会给你加分。**

**感谢牛友201903190906813指出区别8的错误，现在已将区别8划入“相同”，引用也是占内存的。**

**解析：指针与引用这个考点是重中之重，超级经典面试题之一。务必能说出来。**

**相同：都是地址的概念；**

指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名。

从内存分配上看：**两者都是占内存的**，程序为指针变量分配内存区域，在32位系统指针变量一般占用4字节内存，而引用本质是指针常量，所指向的对象不能改变，但指向的对象的值可以改变，引用和指针一样是地址概念，所以本身都是会占用内存的（有的编译器优化后就不占用内存了）。不过也略有区别，见下面的区别7.

**区别：**

1. 指针是一个实体，而引用仅是个别名

2. 指针和引用的自增(++)运算意义不一样，指针是对内存地址的自增，引用是对值的自增；

量或对象的地址)的大小；

3. 引用使用时无需解引用(\*)，指针需要解引用；

4. 引用只能在定义时被初始化一次，之后不可变；指针可变；

5. 引用不能为空，指针可以为空；

6.引用没有const，指针有const；（本人当初看到这句话表示疑问，这里解释一下：指针有“指针常量”即int \* const a,但是引用没有int& const a，不过引用有“常引用”即const int &a = 1）

7. “sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身的大小，在32位系统指针变量一般占用4字节内存。

**指针和引用之间怎么转换：**

1）指针转引用：把指针用\*就可以转换成对象，可以用在引用参数当中。

2）引用转指针：把引用类型的对象用&取地址就获得指针了。

|  |  |
| --- | --- |
| 1  2  3  4  5 | **int** a = 0;    **int** \*pA = &a;    **void** fun(**int** &va){} |

此时调用: fun(\*pA);

pA是指针，加个\*号后可以转换成该指针指向的对象，此时fun的形参是一个引用值，pA指针指向的对象会转换成引用va。

**2.1.5 C语言检索内存情况 内存分配的方式⭐⭐⭐**

**解析：**检索内存：顾名思义，对某段内存进行遍历搜索。

内存分配：

1、从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。

2、在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

3、从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由程序员决定，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。

**2.1.6  extern”C” 的作用**

**解析：**

我们可以在C++中使用C的已编译好的函数模块，这时候就需要用到extern”C”。这是为了避免C++ name mangling，主要用于动态链接库，使得在C++里导出函数名称与C语言规则一致（不改变），方便不同的编译器甚至是不同的开发语言调用。

那么我们来看看C++语言规则和C语言规则有何不同。如果我们定义一个函数：

int fun(int a)；

如果是C++编译器，则可能将此函数改名为int\_fun\_int,（C++重载机制就这么来的）。如果有加上extern”C修饰，则c++编译器会按照C语言编译器一样编译为\_fun。

**答：**extern”C”的作用在于C++代码中调用的C函数的声明，或C++中编译的函数要在C中调用。

**注意**：

1）C++调用一个C语言编写的.so库时，包含描述.so库中函数的头文件时，应该将对应的头文件放置在extern “C”{}格式的{}中，。

2）C中引用C++中的全局函数时，C++的头文件需要加extern “C”，而C文件中不能用extern “C”，只能使用extern关键字。

3）**也就是extern“C” 都是在c++文件里添加的！**

**2.1.7 extern容易忽略的知识点**

答：

1）头文件声明时加extern定义时不要加，因为extern可以多次声明，但只有一个定义。

2）extern在链接阶段起作用（四大阶段：预处理--编译--汇编--链接）。

**2.1.8 函数参数压栈顺序，即关于\_\_stdcall和\_\_cdecl调用方式的理解**

**解析：**

\_\_stdcall和\_\_cdecl都是函数调用约定关键字，我们先来看看\_\_stdcall和\_\_cdecl调用方式的概念：

**1）\_\_stdcall**：是StandardCall的缩写，是C++的标准调用方式，参数由右向左压入堆栈，由调用者负责把参数压入栈，最后由被调用者负责清除栈的内容。

**2）\_\_cdecl：**是C Declaration的缩写（declaration，声明），表示C语言默认的函数调用方法：参数也是由右向左压入堆栈；由调用者负责把参数压入栈，最后也由调用者负责清除栈的内容。

总结：因为以上2点，\_cdecl这种调用约定的特点是支持可变数量的参数，比如printf方法，\_\_stdcall不支持可变数量的参数。

假设函数fun（）作为调用者调用printf打印东西时，可以输入不同数量的参数，printf作为被调用者，并不知道调用者fun（）到底将多少参数压入堆栈，因此printf就没有办法自己清理堆栈，所以只有函数退出之后，由fun清理堆栈，因为fun总是知道自己传入了多少参数。

**2.1.9 重写memcpy()函数需要注意哪些问题？**

答：自己动手实现memcpy()时就需要考虑地址重叠的情况。我们来看个简单的例子。有一个5个元素的数组，不妨设为int arr = {1,2,3,4,5};考虑2种情况：

1）源地址是arr[2]，目标地址是arr[0],自前向后拷贝3个元素后arr为{3,4,5,4,5}

2）源地址是arr[0]，目标地址是arr[2],自前向后拷贝3个元素后arr为{1,2,1,2,3}

第一种情况，由低地址向高地址逐个将源地址的元素拷贝到目标地址就行，容易；

第二种情况需要注意，如果是按第一种情况由低地址拷贝到高地址，需要分3个步骤把arr[0]=1,arr[1]=2,arr[2]=3三个元素逐个拷贝，重点在于第一步是将arr[0]拷贝到arr[2]的位置，这样就会把原来的arr[2]=3改为arr[2]=1，覆盖了原来的值，因此在这种情况，我们需要自后向前拷贝，也就是高地址向低地址拷贝。也就是第一步将arr[2]放到arr[4],第二步将arr[1]放到arr[3],第一步将arr[0]放到arr[2].

看一下代码：

感谢：牛友“牛客513303394号”指出以下代码多了一个标点符号，互勉。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | **void** \*Memcpy(**void** \*dst, **const** **void** \*src, size\_t1 size)    {    **char** \*psrc;  //源地址  **char** \*pdst;  //目标地址    **if**(NULL == dst || NULL == src)      {  **return** NULL;      }    **if**((src < dst) && (**char** \*)src + size > (**char** \*)dst)  //源地址在前，对应上述情况2，需要自后//向前拷贝      {          psrc = (**char** \*)src + size - 1;          pdst = (**char** \*)dst + size - 1;  **while**(size--)          {              \*pdst-- = \*psrc--;          }      }  **else**   //源地址在后，对应上述第一种情况，直接逐个拷贝\*pdst++ = \*psrc++即可      {          psrc = (**char** \*)src;          pdst = (**char** \*)dst;  **while**(size--)          {              \*pdst++ = \*psrc++;          }      }  **return** dst;    } |

**2.1.10数组到底存放在哪里**

**解析：**

1、固定数组在函数体内分配（不带static）是在栈中的

2、固定数组是全局变量和带static前缀的局部数组是在全局数据的

3、固定数组在类中分配是在堆中的

4、动态数组（通过malloc或者new出来的空间）不管在函数体中、类中、全局变量都是在堆中

## 2.1.11   struct和class的区别

**解析**：首先要知道C++是完全兼容C语言的，因此大家可能会随着学习的深入觉得C++中的struct并没有必要保存，因为struct可以完成的事情，class都可以完成。甚至在C++中struct也可以有构造函数，析构函数，结构体之间也可以继承等等。也就是C++中的struct和class其实意义一样。

**总结**：C++中存在struct的唯一意义就是为了让C语言程序员有归属感，是为了让C++编译器兼容以前用C语言开发的项目。

**答**：两者最大区别是struct里面默认的访问控制是public，而class中的默认访问控制是private。

## 2.1.12 char和int之间的转换；

**解析：**

1）首先char与int都分为signed与unsigned类型，默认情况下都是signed类型。

2）从长字节数据类型转换为短字节数据类型，会产生截断：

如从4字节的int类型转换成1个字节的char类型，则取int数据的最低的一个字节，将这个字节的数据赋给char型数据，且是有符号的，即首位为符号位；而如果是从int转换成unsigned char类型，则整个一个字节都是数据，没有符号位。

1）从短字节类型转换为长字节类型，从char转换为int：则在前面的三个字节补符号位，即补上0xffffff（char的首位为1），或0x000000（char的首位为0）。从unsigned char转换为int，则前面补上0x000000.

## 2.1.13  static的用法（定义和用途）

解析：

在C语言中，static作用：“改变生命周期” 或者 “改变作用域”。有以下特性：

1）static局部变量：局部变量为动态存储，即指令执行到定义处才分配内存，将一个变量声明为函数的局部变量，使其变为静态存储方式(静态数据区)，那么这个局部变量在函数执行完成之后不会被释放，而是继续保留在内存中。

2）static全局变量：全局变量即定义{}外面，其本身就是静态变量，编译时就分配内存，这只会改变其连接方式，使其只在本文件内部有效，而其他文件不可连接或引用该变量。

3）static函数：对函数的连接方式产生影响，使得函数只在本文件内部有效，对其他文件是不可见的。这样的函数又叫作静态函数。使用静态函数的好处是，不用担心与其他文件的同名函数产生干扰，另外也是对函数本身的一种保护机制。如果想要其他文件可以引用本地函数，则要在函数定义时使用关键字extern，表示该函数是外部函数，可供其他文件调用。另外在要引用别的文件中定义的外部函数的文件中，使用extern声明要用的外部函数即可。

到了C++的时候，static多了几个其他的作用：

4）static类成员变量：表示这个成员为全类所共有，对类的所有对象只有一份拷贝，可以借助类名直接访问。

5）static类成员函数：表示这个函数为全类所共有，而且只能访问静态成员变量，因为这个函数不接收this指针。

## 2.1.14  const的用法（定义和用途）

Const就是常量修饰符，const变量应该在声明的时候就进行初始化，如果在声明常量的适合没有提供值，则该常量的值是不确定的，且无法修改。

const修饰主要用来修饰变量、函数形参和类成员函数：

1）const常量：定义时就初始化，以后不能更改。

2）const形参：func(const int a){};该形参在函数里不能改变

3）const修饰类成员函数：该函数对成员变量只能进行只读操作，就是const类成员函数是不能修改成员变量的数值的。

## 2.1.15 const常量和#define的区别（编译阶段、安全性、内存占用等）

**解析**：主要有以下区别

1）用#define MAX 255定义的常量是**没有类型**的（不进行类型安全检查，可能会产生意想不到的错误），所给出的是一个立即数，编译器只是把所定义的常量值与所定义的常量的名字联系起来，define所定义的宏变量在**预处理阶段**的时候进行替换，在程序中使用到该常量的地方都要进行拷贝替换；

用const float MAX = 255;定义的常量**有类型（**编译时会进行类型检查）名字，存放在内存的静态区域中，在**编译时**确定其值。在程序运行过程中const变量只有**一个拷贝**，而#define所定义的宏变量却有多个拷贝，所以宏定义在程序运行过程中所消耗的**内存**要比const变量的大得多；

2）用define定义的常量是不可以用指针变量去指向的，用const定义的常量是可以用指针去指向该常量的地址的；

3）用define可以定义一些简单的函数（宏替换只作替换，不做计算，不做表达式求解），const是不可以定义函数的.

4）宏定义的作用范围仅限于当前文件。 而默认状态下，const对象只在文件内有效，当多个文件中出现了同名的const变量时，等同于在不同文件中分别定义了独立的变量。 如果想在多个文件之间共享const对象，必须在变量定义之前添加extern关键字（在声明和定义时都要加）。

**2.1.16  volatile作用和用法**

解析：.volatile的本意是“易变的” 因为访问寄存器要比访问内存单元快的多,所以编译器一般都会作减少存取内存的优化，但有可能会读脏数据。当要求使用volatile声明变量值的时候，编译器对访问该变量的代码就不再进行优化，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。

答：volatile关键词的作用是影响编译器编译的结果，用volatile声明的变量表示该变量随时可能发生变化，与该变量有关的运算，不要进行编译优化，以免出错。

案例：我们来看看以下几个使用volatile的案例：

（1）中断服务程序中修改的供其它程序检测的变量需要加volatile；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **int** i=0;    **int** main(**void**)    {       ...  **while** (1){  **if** (i) dosomething();       }  ｝    /\* Interrupt service routine. \*/  **void** ISR\_2(**void**)  {        i=1;  } |

程序的本意是希望ISR\_2中断产生时，在main函数中调用dosomething函数，但是，由于编译器判断在main函数里面没有修改过i，因此可能只执行一次对从i到某寄存器的读操作，然后每次if判断都只使用这个寄存器里面的“i副本”，导致dosomething永远也不会被调用。如果将变量加上volatile修饰，则编译器保证对此变量的读写操作都不会被优化（肯定执行）。此例中i也应该是volatile int i；

（2）多任务环境下（如多线程）各任务间共享的标志应该加volatile；

（3）存储器映射的硬件寄存器通常也要加voliate，因为每次对它的读写都可能有不同意义。

例如：假设要对一个设备进行初始化，此设备的某一个寄存器为0xff800000。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **int**  \*output = (unsigned  **int** \*)0xff800000;//定义一个IO端口；  **int**   init(**void**)  {  **int** i;  **for**(i=0;i< 10;i++){           \*output = i;  }  } |

经过编译器优化后，编译器认为前面循环半天都是废话，对最后的结果毫无影响，因为最终只是将output这个指针赋值为9，所以编译器最后给你编译编译的代码结果相当于：

|  |  |
| --- | --- |
| 1  2  3  4 | **int**  init(**void**)  {        \*output = 9;  } |

如果你对此外部设备进行初始化的过程是必须是像上面代码一样顺序的对其赋值，显然优化过程并不能达到目的。反之如果你不是对此端口反复写操作，而是反复读操作，其结果是一样的，编译器在优化后，也许你的代码对此地址的读操作只做了一次。然而从代码角度看是没有任何问题的。这时候就该使用volatile通知编译器这个变量是一个不稳定的，在遇到此变量时候不要优化。

例如：

|  |  |
| --- | --- |
| 1 | **volatile**  **int** \*output=(**volatile** unsigned **int** \*)0xff800000;//定义一个I/O端口 |

**2.1.17有常量指针 指针常量 常量引用 没有 引用常量**

解析：这几个概念面试官偶尔会调皮的问一下，所以还是要区分好几个概念，不要混淆，以上4个概念都可以从最后两个字开始记起，后面两个字代表这是个什么东东，前面两个字代表这个是什么类型：

（1）常量指针：也叫常指针，最后两个字是“指针”，代表这是一个指针，但指向的是一个常量，如下：

|  |  |
| --- | --- |
| 1  2 | **int** a = 0;  **const** **int** \*p = &a; //不可以通过p改变a |

（2）指针常量：后面两个字是“常量”，代表这是个常量，不过是指针类型的常量，

|  |  |
| --- | --- |
| 1  2 | **int** a = 0;  **int** \***const** p = &a; //从后往前看，这是个指针常量，指向的a的值可以改变，但p本身不可改变 |

注意：如果从代码来区分常量指针指针常量，那么可以从后往前看const的位置，

|  |  |
| --- | --- |
| 1  2 | **const** **int** \*p = &a //从后往前看，const修饰的是\*p，所以指针p指向的数值不可变  **int** \***const** p = &a; //从后往前看，const修饰的是p，所以指针p本身不可变 |

（3）常量引用：后两个字是“引用”，那么这个是引用，并且是常量的引用，那么就有两个性质，如下：

|  |  |
| --- | --- |
| 1  2  3 | **double** a;  **const** **int** &r = a; //正确 性质1：不可通过常量引用r来改变a  **const** **int** &r = 10;//正确 性质2：常量引用可以直接引用具体数值 |

（4）没有引用常量：后面两个字代表这个是常量，前面代表这个是引用类型的常量，然而常量就是常量了，并没有引用类型的常量。

**2.1.18没有指向引用的指针，但是有指针的引用**

解析：如何理解这句话呢，首先，没有指向引用的指针，因为指针是本质上是指向某一块内存空间的，而引用只是一个变量的别名，本身是没有地址的，如果要创建一个指针指向某个引用，那么其实指向的是这个引用所引用的对象，看下面代码：

|  |  |
| --- | --- |
| 1  2  3 | **int** v = 1;  **int** &ri = v; //整型变量v的引用  **int** \*p = &ri; //指针p其实指向的是变量v |

其次，有指针的引用，我们直接看代码：

|  |  |
| --- | --- |
| 1  2  3 | **int** v = 1;  **int** \*p = &v;  **int** \*&rp = p; |

第一个是要理解int \*&rp = p; 这是定义了一个变量rp，还是从后往前看，距离rp左边最近的修饰符决定rp是个什么东东，剩下的就是rp的具体值。因此我们发现距离rp左边最近的是&，代表rp是个引用，所以int \*&rp = p; 可以看作int \*（&rp ）= p; 如果我们把（&rp）当作一个整体，又可以看作int \* RP = p；到此为止，我们就可以很明显的知道这句话其实就是定义了一个引用rp指向指针p。

**2.1.19 c/c++中变量的作用域**

解析：作用域规则告诉我们一个变量的有效范围，它在哪儿创建，在哪儿销毁。变量的有效作用域从它的定义点开始，到和定义变量之前最邻近的开括号配对的第一个闭括号。也就是说，作用域由变量所在的最近一对括号确定。

答：

(1)全局变量：

   全局变量是在所有函数体的外部定义的，程序的所在部分（甚至其它文件中的代码）都可以使用。全局变量不受作用域的影响（也就是说，全局变量的生命期一直到程序的结束）。如果在一个文件中使用extern关键字来声明另一个文件中存在的全局变量，那么这个文件可以使用这个数据。

(2)局部变量：

    局部变量出现在一个作用域内，它们是局限于一个函数的。局部变量经常被称为自动变量，因为它们在进入作用域时自动生成，离开作用域时自动消失。关键字auto可以显式地说明这个问题，但是局部变量默认为auto，所以没有必要声明为auto。

(3)寄存器变量

   寄存器变量是一种局部变量。关键字register告诉编译器“尽可能快地访问这个变量”。加快访问速度取决于现实，但是，正如名字所暗示的那样，这经常是通过在寄存器中放置变量来做到的。这并不能保证将变置在寄存器中，甚至也不能保证提高访问速度。这只是对编译器的一个暗示。

注意：

     使用register变量是有限制的：(1)不可能得到或计算register变量的地址; (2) register变量只能在一个块中声明（不可能有全局的或静态的register变量（c语言里register关键字可以在全局中定义变））。然而可以在一个函数中（即在参数表中）使用register变量作为一个形式参数。

  一般地，不应当推测编译器的优化器，因为它可能比我们做得更好。因此，最好避免使用关键字register。

(4)静态变量

   关键字static有一些独特的意义。通常，函数中定义局部变量在函数中作用域结束时消失。当再次调用这个函数时，会重新创建变量的存储空间，其值会被重新初始化。如果想使局部变量的值在程序的整个生命期里仍然存在，我们可以定义函数的局部变量为static(静态的)，并给它一个初始化。初始化只在函数第一次调用时执行，函数调用之间变量的值保持不变，这种方式，函数可以“记住”函数调用之间的一些信息片断。这也就是所谓的静态局部变量，具有局部作用域，它只被初始化一次，自从第一次被初始化直到程序运行结束都一直存在，它和全局变量的区别在于全局变量对所有的函数都是可见的，而静态局部变量只在定义自己的函数体内始终可见。

   我们可能奇怪为什么不使用全局变量。static局部变量的优点是在函数范围之外它是不可用的，所以它不可能被轻易改变。这会使错误局部化。

   此外同样存在静态全局变量，具有全局作用域，它与全局变量的区别在于如果程序包含多个文件的话，它作用于定义它的文件里，不能作用到其它文件里，即被static关键字修饰过的变量具有文件作用域。这样即使两个不同的源文件都定义了相同名字的静态全局变量，它们也是不同的变量。

(5)外部变量

   extern告诉编译器存在着一个变量和函数，即使编译器在当前的文件中没有看到它。这个变量或函数可能在一个文件或者在当前文件的后面定义。例如extern int i;编译器会知道i肯定作为全局变量存在于某处。当编译器看到变量i的定义时，并没有看到别的声明，所以知道它在文件的前面已经找到了同样声明的i。

(6) const常量

    const告诉编译器这个名字表示常量，不管是内部的还是用户定义的数据类型都可以定义为const。如果定义了某对象为常量，然后试图改变它，编译器将会产生错误。在C++中一个const必须有初始值。

(7) volatile变量

    限定词const告诉编译器“这是不会改变的”(这就是允许编译器执行额外的优化)；而限定词volatile则告诉编译器“不知道何时变化”，防止编译器依据变量的稳定性作任何优化。

**2.1.20 c++中类型转换机制？各适用什么环境？dynamic\_cast转换失败时，会出现什么情况？（对指针，返回NULL对。引用，抛出bad\_cast异常）**

 解析：C++中，四个与类型转换相关的关键字：static\_cast、const\_cast、reinterpret\_cast dynamic\_cast。

**（1）static\_cast**

特点：静态转换，在编译处理期间。

应用场合：主要用于C++中内置的基本数据类型之间的转换，但是没有运行时类型的检测来保证转换的安全性。

 用于基类和子类之间的指针或引用之间的转换，这种转换把子类的指针或引用转换为基类表示是安全的；进行下行转换，把基类的指针或引用转换为子类表示时，由于没有进行动态类型检测，所以是不安全的。上行安全下行不安全

 把void类型的指针转换成目标类型的指针（不安全）。

 不能用于两个不相关的类型转换。

 不能把const对象转换成非const对象。

**（2）const\_cast**

特点：去常转换，编译时执行。不是运行时执行

应用场合：const\_cast操作不能在不同的种类间转换。相反，它仅仅把它作用的表达式转换成常量。它可以使一个本来不是const类型的数据转换成const类型的，或者把const属性去掉。

去掉const属性：const\_case<int\*> (&num)，常用，因为**不能把一个const变量直接赋给一个非const变量，必须要转换。**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **int** main()  {  **const** **int** constant = 26;  **const** **int**\* const\_p = &constant;  **int**\* modifier = **const\_cast**<**int**\*>(const\_p);      \*modifier = 3;      cout<< "constant:  "<<constant<<endl;  //26      cout<<"\*modifier:  "<<\*modifier<<endl;   //3  **return** 0;  } |

加上const属性：const int\* k = const\_case<const int\*>(j)，一般很少用，因为可以把一个非const变量直接赋给一个const变量，比如：const int\* k = j;

const\_case只能转换**指针或引用** 不能转换变量

const int i = 3;

int j = const\_cast<int>(i);是不行的

**（3）reinterpret\_cast**

特点： 重解释类型转换

应用场合：它有着和c风格强制类型转换同样的功能；它可以转化任何的内置数据类型为其他的类型，同时它也可以把任何类型的指针转化为其他的类型；它的机理是对二进制进行重新的解释，不会改变原来的格式。

**（4）dynamic\_cast < *type-id* > ( *expression* )**

含义：将一个指向基类的指针转换成指向派生类的指针；如果失败，返回空指针。

该运算符将expression转换成type\_id类型的对象。type\_id必须是类的指针，类的引用或者空类型的指针。

a、如果type\_id是一个指针类型，那么expression也必须是一个指针类型，如果type\_id是一个引用类型，那么expression也必须是一个引用类型。

b、如果type\_id是一个空类型的指针，在运行的时候，就会检测expression的实际类型，结果是一个由expression决定的指针类型。

c、如果type\_id不是空类型的指针，在运行的时候指向expression对象的指针能否可以转换成type\_id类型的指针。

d、在运行的时候决定真正的类型，如果向下转换是安全的，就返回一个转换后的指针，若不安全，则返回一个空指针。

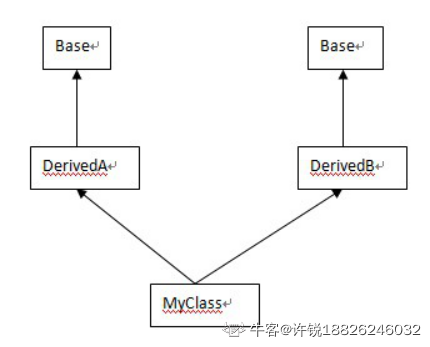
e、主要用于上下行之间的转换，也可以用于类之间的交叉转换。上行转换时和static\_cast效果一样，下行转换时，具有检测功能，比static\_cast更安全。

[复制代码](https://blog.nowcoder.net/n/0e63cb499c5947908912e16ec77c143d)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | **class** CBasic{  **public**:        CBasic(){};      ~CBasic(){};  **virtual** **void** speak() {     //要有virtual才能实现多态，才能使用dynamic cast，如果父类没有虚函数，是编译不过的  **printf**("dsdfsd");      }  **private**:    };    //哺乳动物类  **class** cDerived:**public** CBasic{  **public**:      cDerived(){};      ~cDerived(){};  **private**:  };    **int** main()  {       CBasic  cBasic;       CDerived  cDerived;         CBasic \* pB1 = **new** CBasic;       CBasic \* pB2 = **new** CDerived;         //dynamic cast failed, so pD1 is null.  pB1指向对象和括号里的Derived \*不一样，转换失败       CDerived \* pD1 = **dynamic\_cast**<CDerived \* > (pB1);         //dynamic cast succeeded, so pD2 points to  CDerived object       //dynamic cast 用于将指向子类的父类指针或引用，转换为子类指针或引用 ，pB2指向对象和括号里的Derived \*一样，转换成功       CDerived \* pD2 = **dynamic\_cast**<CDerived \* > (pB2);         //dynamci cast failed, so throw an exception.       CDerived & rD1 = **dynamic\_cast**<CDerived &> (\*pB1);         //dynamic cast succeeded, so rD2 references to CDerived object.       CDerived & rD2 = **dynamic\_cast**<CDerived &> (\*pB2);  **return** 0;  } |

**2.2.1继承和虚继承**

解析：因为C++支持多重继承，那么在这种情况下会出现重复的基类这种情况，也就是说可能出现将一个类两次作为基类的可能性。比如像下面的情况



为了节省内存空间，可以将DeriverdA、DeriverdB对Base的继承定义为虚拟继承，而A就成了虚拟基类。实现的代码如 下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **class** Base    **class** DeriverdA:**public** **virtual** Base; //虚继承    **class** DeriverdB:**public** **virtual** Base; //虚继承    **class** D:**public** DeriverdA,DeriverdB;  //普通继承 |

虚拟继承在一般的应用中很少用到，所以也往往被忽视，这也主要是因为在C++中，多重继承是不推荐的，也并不常用，而一旦离开了多重继承，虚拟继承就完全失去了存在的必要因为这样只会降低效率和占用更多的空间。

**注意**：不要全部都使用虚继承，因为虚继承会破坏继承体系，不能按照平常的继承体系来进行类型转换（如C++提供的强制转换函数static\_cast对继承体系中的类对象转换一般可行的，这里就不行了）。所以不要轻易使用虚继承，更不要在虚继承的基础上进行类型转换，切记切记！

**2.2.2多态的类，内存布局是怎么样的**

解析：关于类的内存布局主要是考某个类所占用的内存大小，以下通过几个案例加以分析。

（1）虚继承：如果是虚继承，那么就会为这个类创建一个虚表指针，占用4个字节

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <stdio.h>  **class** A {  **public**:  **int** a;  }; //sizeof(A)=4，因为a是整形，占用4字节    **class** B : **virtual** **public** A {  **public**:  **int** b;  };//sizeof(B)=4(A副本)+4(虚表指针占用4字节)+4(变量b占用4字节)=12    **class** C : **virtual** **public** B {  };//sizeof(c)= 12(B副本)+4(虚表指针) = 16，如果这里改为直接继承，那么sizeof(c)=12，因为此时就没有虚表指针了 |

（2）多重继承：如果是以**虚继承**实现多重继承，记得减掉基类的副本

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <stdio.h>  **class** A {  **public**:  **int** a;  };//sizeof(A) = 4    **class** B : **virtual** **public** A {  };// sizeof(B) =4+4=8    **class** C : **virtual** **public** A {  };//sizeof(C) =4+4=8    **class** D : **public** B, **public** C{  };  //sizeof(D)=8+8-4=12这里需要注意要减去4，因为B和C同时继承A，只需要保存一个A的副本就好了，sizeof(D)=4(A的副本)+4(B的虚表)+4(C的虚表)=12，也可以是8（B的副本）+8（c的副本）-4（A的副本）=12 |

（3）普通继承（含有：空类、虚函数）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **class** A      //result=1  空类所占空间的大小为1  {  };    **class** B     //result=8  1+4   字节对齐后为 8  {  **char** ch;  **virtual** **void** func0()  {  }  };    **class** C    //result=8  1+1+4 字节对齐后为 8，没有继承的，此时类里即使出现多个虚函数，也只有一个虚指针  {  **char** ch1;  **char** ch2;  **virtual** **void** func()  {  }    //也只有一个虚指针  **virtual** **void** func1() {  }    //也只有一个虚指针  };    **class** D: **public** A, **public** C  //result=12  8（C的副本）+4（整形变量d占用4字节）=12  {  **int** d;  **virtual** **void** func()  {  }   //继承了C，C里已经有一个虚指针，此时D自己有虚函数，  **virtual** **void** func1()  {  } //也不会创建另一个虚指针，所以D本身就变量d需要4字节  };    **class** E: **public** B, **public** C  //result=20   8（ B的副本）+8（C的副本）+4（E本身）=20  {  **int** e;  **virtual** **void** func0()  {  }   //同理，E不会创建另一个虚指针，所以E本身就变量e需  **virtual** **void** func1()  {  }  //要4字节  }; |

  （4）虚继承（多重继承和虚函数）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **class** CommonBase  {  **int** co;  };// size = 4    **class** Base1: **virtual** **public** CommonBase  {  **public**:  **virtual** **void** print1() {  }  **virtual** **void** print2() {  }  **private**:  **int** b1;  };//4（父类副本）+4（自己有虚函数，加1个虚指针空间）+4（自身变量b1）+4(虚继承再加1个虚指针空间)=16    **class** Base2: **virtual** **public** CommonBase  {  **public**:  **virtual** **void** dump1() {  }  **virtual** **void** dump2() {  }  **private**:  **int** b2;  };//同理16    **class** Derived: **public** Base1, **public** Base2  {  **public**:  **void** print2() {  }  **void** dump2() {  }  **private**:  **int** d;  };//16+16-4+4=32 |

前辈总结说：如果不是虚继承的类，即便有虚函数也不会因此增加存储空间，如果是虚继承的类，没有虚函数就添加一个虚指针空间，有虚函数不论多少个，就添加两个虚指针空间。  
本人将前辈总结归纳为：如果此时类里有一个或多个虚函数，那么需要加1个虚指针空间，如果还是虚继承，那么需要再加1个虚指针空间，最多就2个虚指针空间。

（5）虚继承与虚函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **class** A  {  **public**:  **virtual** **void** aa() {  }  **virtual** **void** aa2() {  } //如果此时类里有一个或多个虚函数，那么需要加1个虚指针空间  **private**:  **char** ch[3];  }; // 1+4 =补齐= 8    **class** B: **virtual** **public** A //如果还是虚继承，那么需要再加1个虚指针空间，最多就2个虚指//针空间。  {  **public**:  **virtual** **void** bb() {  }  **virtual** **void** bb2() {  }  }; // 8（副本）+4(虚继承)+4（虚指针）= 16 |

【小结】重要的事情讲三遍！！！

如果此时类里有一个或多个虚函数，那么需要加1个虚指针空间，如果还是虚继承，那么需要再加1个虚指针空间，最多就2个虚指针空间。

**2.2.3被隐藏的基类函数如何调用或者子类调用父类的同名函数和父类成员变量**

解析：父类的同名函数和父类成员变量被隐藏不代表其不存在，只是藏起来而已，C++有两种方法可以调用被隐藏的函数：

1）用using关键字：使用using后，父类的同名函数就不再隐藏，可以直接调用，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **class** Child:**public** Parent  {  **public**:      Child(){};  **using** Parent::add;  **int** add(**void**){};  }; |

2）用域操作符，可以调用基类中被隐藏的所有成员函数和变量。

如子类child和父类father都有add（）函数，可以通过下面代码实现子类对象调用父类的add（）函数：

Child c;

c.Parent::add(10);

**2.2.4 C++如何实现多态（讲明多态实现的三个条件、实现的原理）**

**解析**：只要是有涉及到c++的面试，面试官百分百会问到多态相关的问题，尤其是让你解释下多态实现的原理，此时首先要知道多态实现的三个条件：

1）要有继承

2）要有虚函数重写

3）要有父类指针（父类引用）指向子类对象

**答：**

编译器发现一个类中有虚函数，便会立即为此类生成虚函数表vtable。虚函数表的各表项为指向类里面的虚函数的指针。编译器还会在此类中隐含插入一个指针vptr（对 vc 编译器来说，它插在类的内存地址的第一个位置上）指向虚函数表。调用此类的构造函数时，在类的构造函数中，编译器会隐含执行vptr 与 vtable 的关联代码，即将vptr 指向对应的 vtable，将类与此类的vtable 联系了起来。

另外在调用类的构造函数时，指向基础类的指针此时已经变成指向具体的类的this 指针，这样依靠此 this 指针即可得到正确的 vtable，如此才能真正与函数体进行连接，这就是动态联编，实现多态的基本原理。

上面这段话可能有点难以理解，我本人当初也是理解好一会哈，我们直接看个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | #include "stdafx.h"  #include <iostream>  #include <stdlib.h>  **using** **namespace** std;    **class** Father  {  **public**:  **void** Face()      {          cout << "Father's face" << endl;      }    **virtual** **void** Say()      {          cout << "Father say hello" << endl;      }  };    **class** Son:**public** Father  {  **public**:  **void** Say()      {          cout << "Son say hello" << endl;      }  };    **void** main()  {      Son son;      Father \*pFather=&son; //隐式类型转换      pFather->Say();  } |

我们重点来看这行代码Father \*pFather=&son;

此时指向基础类的指针pFather已经变成指向具体的类son的this指针，那么我们调用这个pFather父类指针，就相当于调用了**等号右边**的类即子类son的this指针，这个this所能调用的函数，自然就是子类son本身的函数。即pFather->Say();这行代码调用的是子类的Say（）函数。因此我们就成功的实现了用父类指针pFather调用子类函数，也就是实现了多态。

**2.2.5对拷贝构造函数 深浅拷贝 的理解 拷贝构造函数作用及用途？什么时候需要自定义拷贝构造函数？**

解析：简单的来说，浅拷贝是增加了一个指针，指向原来已经存在的内存。浅拷贝在多个对象指向一块空间的时候，释放一个空间会导致其他对象所使用的空间也被释放了，再次释放便会出现错误。

而深拷贝是增加了一个指针，并新开辟了一块空间让指针指向这块新开辟的空间。深拷贝和浅拷贝的不同之处，仅仅在于修改了下拷贝构造函数，以及赋值运算符的重载。就类对象而言，相同类型的类对象是通过拷贝构造函数来完成整个复制过程的。

**提问：**什么时候需要自定义拷贝构造函数？

**答**：

默认拷贝构造函数执行的是浅拷贝，对于凡是包含动态分配成员或包含指针成员的类都应该提供拷贝构造函数；在提供拷贝构造函数的同时，还应该考虑重载"="赋值操作符号。

**2.2.6析构函数可以抛出异常吗？为什么不能抛出异常？除了资源泄露，还有其他需考虑的因素吗？**

解析：

C++标准指明析构函数**不能、也不应该抛出异常**。C++异常处理模型最大的特点和优势就是对C++中的面向对象提供了最强大的无缝支持。那么如果对象在运行期间出现了异常，C++异常处理模型有责任清除那些由于出现异常所导致的已经失效了的对象(也即对象超出了它原来的作用域)，并释放对象原来所分配的资源，这就是调用这些对象的析构函数来完成释放资源的任务，所以从这个意义上说，**析构函数已经变成了异常处理的一部分**。

1）如果析构函数抛出异常，则异常点之后的程序不会执行，如果析构函数在异常点之后执行了某些必要的动作比如释放某些资源，则这些动作不会执行，会造成诸如资源泄漏的问题。

2）通常异常发生时，c++的机制会调用已经构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常（析构函数里delete this指针也会造成程序崩溃，因为delete this指针就是要调用析构函数，这样就变成无限循环了），会造成程序崩溃的问题。

答：析构函数不能抛出异常，除了资源泄露还可能造成程序崩溃。

**2.2.7什么情况下会调用拷贝构造函数（三种情况）**

解析：类的对象需要拷贝时，拷贝构造函数将会被调用。以下情况都会调用拷贝构造函数：

1.一个对象以值传递的方式传入函数体

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11 | //全局函数，传入的是对象  **void** g\_Fun(CExample C)  {      cout<<"test"<<endl;  }  **int** main()  {      CExample test(1);      //传入对象      g\_Fun(test);  } |   调用g\_Fun()时，会产生以下几个重要步骤：  (1)test对象传入形参时，会先会产生一个临时变量，就叫C吧。  (2)然后调用拷贝构造函数把test的值给C。整个这两个步骤有点像：CExample C(test);  (3)等g\_Fun()执行完后,析构掉C对象。 |

2.一个对象以值传递的方式从函数返回

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | //全局函数  CExample g\_Fun()  {      CExample temp(0);  **return** temp;  }  **int** main()  {      g\_Fun();  } |   当g\_Fun()函数执行到return时，会产生以下几个重要步骤：  (1)先会产生一个临时变量，就叫XXXX吧。  (2)然后调用拷贝构造函数把temp的值给XXXX。整个这两个步骤有点像：CExample XXXX(temp);  (3)在函数执行到最后先析构temp局部变量。  (4)等g\_Fun()执行完后再析构掉XXXX对象. |

3.一个对象需要通过另外一个对象进行初始化。

|  |
| --- |
| CExample A(100);  CExample B = A; //这句和下句都会调用拷贝构造函数。  CExample B(A); |

**2.2.8析构函数一般写成虚函数的原因**

在实现多态时（基类指针可以指向子类的对象），如果析构函数是虚函数，那么当用基类操作子类的时候（基类指针可以指向子类的对象），如果删除该基类指针时，就会调用该基类指针指向的子类析构函数，而子类的析构函数又自动调用基类的析构函数，这样整个子类的对象完全被释放。这是最理想的结果。

如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不会像上一段那样调用子类析构函数，这样就会造成子类对象析构不完全。所以，将析构函数声明为虚函数是十分必要的。

**2.2.9构造函数为什么一般不定义为虚函数**

注意：面试时起码回答出第一和第二点，最好还有第三点，第四第五是加分项。

答：

1）**从存储空间角度**：大家都知道虚函数相应一个指向vtable虚函数表的指针，而这个指向vtable的指针事实上是存储在对象的内存空间的。如果此时构造函数是虚的，就须要通过vtable来调用，但是对象还没有实例化，也就是内存空间还没有vtable，怎么找vtable呢？所以构造函数不能是虚函数。

2）**从使用角度**：虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。

3）**从实现上看**：vbtl在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。

4）构造函数不须要是虚函数，也不同意是虚函数，由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去訪问它但析构却不一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。

5）当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的VPTR。因此，它仅仅能知道它是“当前”类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR将保持被初始化为指向这个VTABLE,但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置VPTR指向它的VTABLE，等.直到最后的构造函数结束。VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置VPTR指向它自己的VTABLE。假设函数调用使用虚机制，它将仅仅产生通过它自己的VTABLE的调用，而不是最后的VTABLE（全部构造函数被调用后才会有最后的VTABLE）。

**2.2.10什么是纯虚函数**

解析：纯虚函数声明：virtual函数类型 函数名（参数表列）= 0；

纯虚函数只有函数的名字而不具备函数的功能，不能被调用。纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对他进行定义。如果在基类中没有保留函数名字，则无法实现多态性。如果在一个类中声明了纯虚函数，在其派生类中没有对其函数进行定义，则该虚函数在派生类中仍然为纯虚函数。

注意：

（1）纯虚函数没有函数体；

（2）最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是虚函数”；

（3）这是一个声明语句，最后有分号。

Ps：抽象类：不用定义对象而只作为一种基本类型用作继承的类叫做抽象类（也叫接口类），凡是包含纯虚函数的类都是抽象类，抽象类的作用是作为一个类族的共同基类，为一个类族提供公共接口，抽象类不能实例化出对象。纯虚函数在派生类中重新定义以后，派生类才能实例化出对象。

**2.2.11静态绑定和动态绑定的介绍**

解析：

静态类型：对象在声明时采用的类型，在编译期既已确定；

动态类型：通常是指一个指针或引用目前所指对象的类型，是在运行期决定的；

静态绑定：绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期；

动态绑定：绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期；

非虚函数一般都是静态绑定，而虚函数都是动态绑定（如此才可实现多态性）。

问题：引用是否能实现动态绑定，为什么引用可以实现

答：只有指定为虚函数的成员函数才能进行动态绑定，且必须通过基类类型的引用或指针进行函数调用，因为每个派生类对象中都拥有基类部分，所以可以使用基类类型的指针或引用来引用派生类对象。而指针或引用是在运行期根据他们绑定的具体对象确定。

**2.2.12 C++所有的构造函数**

在面向对象编程中，创建对象时系统会自动调用构造函数来初始化对象，构造函数是一种特殊的类成员函数，它有如下特点：

1.构造函数的名子必须和类名相同，不能任意命名；

2.构造函数没有返回值；

3.构造函数可以被重载，但是每次对象创建时只会调用其中的一个；

C++中的构造函数可以分为4类：

（1）默认构造函数。以Student类为例，默认构造函数的原型为

Student(）；//没有参数

Student(int num=10,int age=10);

（2）初始化构造函数

Student(int num，int age）；//有参数

（3）复制（拷贝）构造函数

Student(const Student&）；//形参是本类对象的引用

（4）转换构造函数

Student(int r)；//形参是其他类型变量，且只有一个形参

1）默认构造函数和初始化构造函数在定义类的对象的时候，完成对象的初始化工作。

Student s2（1002,1008）;

2）复制构造函数用于复制本类的对象。

默认的复制构造函数可能会发生【浅拷贝】的问题

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Student(Student &b)  {    **this**.x=b.x;    **this**.y=b.y;    } |

Student [s3](https://www.baidu.com/s?wd=s3&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)（s2）;//将对象s2复制给s3。注意复制和赋值的概念不同。

Student s4;

s4=s2;//这种情况叫做赋值，自己体会吧

3）转换构造函数的作用是将一个其他类型的数据转换为一个类的对象。转换构造函数也是一种构造函数，它遵循构造函数的一般原则，我们通常把仅有一个参数的构造函数用作类型转换，所把它称为转换构造函数。

转换构造函数中的类型数据可以是普通类型，也可以是类类型。

下面的转换构造函数，将int类型的r转换为Student类型的对象，对象的age为r，num为1004

|  |
| --- |
|  |

|  |  |
| --- | --- |
| 1  2  3  4  5 | Student(**int** r)  {  **int** num=1004；  **int** age= r；  ｝ |

**2.2.13重写、重载、覆盖的区别**

解析：至于什么是重写、重载、覆盖，读者可以自行了解，这是必须掌握的重点概念哦。

（1）重写和重载主要有以下几点不同。

范围的区别：被重写的和重写的函数在两个类中，而重载和被重载的函数在同一个类中。

参数的区别：被重写函数和重写函数的参数列表一定相同，而被重载函数和重载函数的参数列表一定不同。

virtual的区别：重写的基类中被重写的函数必须要有virtual 修饰，而重载函数和被重载函数可以被virtual修饰，也可以没有。

（2）隐藏和重写、重载有以下几点不同。

与重载的范围不同：和重写一样，隐藏函数和被隐藏函数不在同一个类中。

参数的区别：隐藏函数和被隐藏的函数的参数列表可以相同，也可不同，但是函数名肯定要相同。

当参数不相同时，无论基类中的参数是否被virtual修饰，基类的函数都是被隐藏，而不是被重写。

说明：虽然重载和覆盖都是实现多态的基础，但是两者实现的技术完全不相同，达到的目的也是完全不同的，覆盖是动态态绑定的多态，而重载是静态绑定的多态。

**2.2.14成员初始化列表的概念，为什么用成员初始化列表会快一些（性能优势）？**

从概念上讲，调用构造函数时，对象在程序进入构造函数函数体之前被创建。也就是说，调用构造函数的时候，**先创建对象，再进入函数体**。所以如果类成员里面有**引用**数据成员与const数据成员，因为他们在创建时初始化，若是在构造函数中初始化则会报错。

只有构造函数可以使用初始化列表语法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **class** MyClass  {  **private**:  **int** a;  **int** b;  **const** **int** max;  };    MyClass(**int** c)  {      a = 0;      b = 0;      mac = c;//这里会出错 const数据成员若是在构造函数中初始化则会报错。  } |

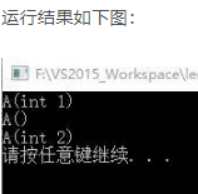
正确的是：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | MyClass(**int** x):a(0),b(0),max(x)  {    }    MyClass(**int** x):max(x)  {      a = 0;      b = 0;  } |

对于普通数据类型，复合类型（指针，引用）等，在成员初始化列表和构造函数体内进行，在性能和结果上都是一样的。对于用户定义类型（类类型），结果上相同，但是性能上存在很大的差别。因为类类型的数据成员对象在进入函数体是已经构造完成，也就是说在成员初始化列表处进行构造对象的工作，这时调用一个构造函数，在进入函数体之后，进行的是对已经构造好的类对象的赋值，又调用个拷贝赋值操作符才能完成（如果并未提供，则使用编译器提供的默认按成员赋值行为）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include<iostream>  Using **namespace** std;  Class A  {  Public:      A()      {          cout<<"A()"<<endl;      }      A(**int** a)      {          value = a;          cout<<"A(int"<<value<<")"<<endl;      }      A(**const** A& a)      {          value = a.value;          cout<<"A(const A& a):"<<value<<endl;      }  **int** value;  };    **class** B  {  **public**:      B():a(1)      {          b = A(2);      }      A a;      A b;  };    **int** main()  {      B b;  **system**("pause");  } |

以上代码对于变量a使用初始化列表初始化，对于变量b使用构造函数初始化；



对于用户定义类型，使用列表初始化可以减少一次默认构造函数调用过程.简单的来说，对于用户定义类型：

1）如果使用类初始化列表，直接调用对应的构造函数即完成初始化

2）如果在构造函数中初始化，**那么首先调用默认的构造函数，然后调用指定的构造函数，要调用2次，所以不推荐在构造函数内初始化**

**2.2.15如何避免编译器进行的隐式类型转换；（用explicit）**

答：

explicit关键字的作用就是防止类构造函数的隐式自动转换. explicit关键字只对有一个参数的类构造函数有效, 如果类构造函数参数大于或等于两个时, 是不会产生隐式转换的, 所以explicit关键字也就无效了.但是, 也有一个例外, 就是当除了第一个参数以外的其他参数都有默认值的时候, explicit关键字依然有效, 此时, 当调用构造函数时只传入一个参数, 等效于只有一个参数的类构造函数

# 3.1.1 TCP、UDP的区别

解析：TCP---传输控制协议,提供的是面向连接、可靠的字节流服务。当客户和服务器彼此交换数据前，必须先在双方之间建立一个TCP连接，之后才能传输数据。

UDP---用户数据报协议，是一个简单的面向数据报的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。

总结为以下几点：

 1）TCP是面向连接的，UDP是面向无连接的

 2）UDP程序结构较简单

 3）TCP是面向字节流的，UDP是基于数据报的

 4）TCP保证数据正确性，UDP可能丢包

 5）TCP保证数据顺序，UDP不保证

## 3.1.2 TCP、UDP的优缺点

TCP优点：可靠稳定

     TCP的可靠体现在TCP在传输数据之前，会有三次握手来建立连接，而且在数据传递时，有确认.窗口.重传.拥塞控制机制，在数据传完之后，还会断开来连接用来节约系统资源。

TCP缺点：慢，效率低，占用系统资源高，易被攻击

    在传递数据之前要先建立连接，这会消耗时间，而且在数据传递时，确认机制.重传机制.拥塞机制等都会消耗大量时间，而且要在每台设备上维护所有的传输连接。然而，每个连接都会占用系统的CPU，内存等硬件资源。因为TCP有确认机制.三次握手机制，这些也导致TCP容易被利用，实现DOS. DDOS. CC等攻击。

UDP优点：快，比TCP稍安全

     UDP没有TCP拥有的各种机制，是一种无状态的传输协议，所以传输数据非常快，没有TCP的这些机制，被攻击利用的机会就少一些，但是也无法避免被攻击。

UDP缺点：不可靠，不稳定

    因为没有TCP的这些机制，UDP在传输数据时，如果网络质量不好，就会很容易丢包，造成数据的缺失。

## 3.1.3 TCP UDP适用场景

TCP：当对网络质量有要求时，比如HTTP，HTTPS，FTP等传输文件的协议；POP，SMTP等邮件传输的协议

UDP：对网络通讯质量要求不高时，要求网络通讯速度要快的场景

**问：TCP改进：（这个随便说说就行了）**

答：许多调整的参数可用于增强TCP的性能，包括数据段、定时器和窗口的大小。TCP实现中含有大量拥塞避免算法，如俊启动、选择重传和选择确认，它通常能改进像Internet这样的共享网络的性能。但在许多拥塞控制算法，特别是慢启动中，当中等数量数据正在一个具有较大带宽延迟特性的链路上传输时，会产生端到端通信的低效带宽利用问题。对此需要有相应的解决办法。

## 3.1.4 TCP为什么是可靠连接

答：因为tcp传输的数据满足3大条件，不丢失，不重复，按顺序到达，

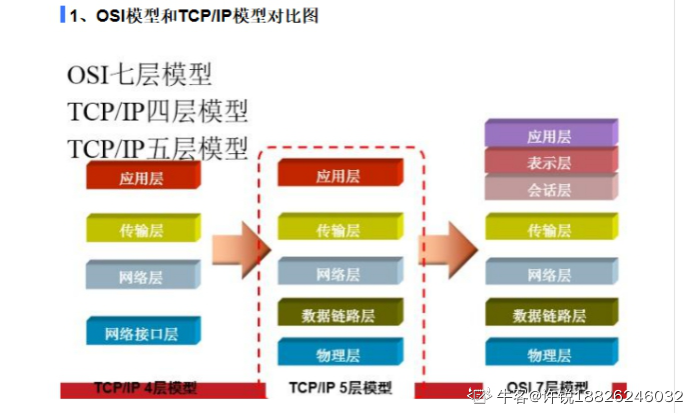
TCP协议保证数据传输可靠性的方式主要有：

* 校验和
* 序列号
* 确认应答
* 超时重传
* 连接管理
* 流量控制
* 拥塞控制

 csdn有个文章《网络基础：TCP协议-如何保证传输可靠性》写得简单易懂。

## 3.1.5典型网络模型，简单说说有哪些；

看看以下的图即可，四层和七层的比较常见。



## 3.1.6 Http1.1和Http1.0的区别

答：

    在http1.0中，当建立连接后，客户端发送一个请求，服务器端返回一个信息后就关闭连接，当浏览器下次请求的时候又要建立连接，显然这种不断建立连接的方式，会造成很多问题。

    在http1.1中，引入了持续连接的概念，通过这种连接，浏览器可以建立一个连接之后，发送请求并得到返回信息，然后继续发送请求再次等到返回信息，也就是说客户端可以连续发送多个请求，而不用等待每一个响应的到来。

    其他的可以自行百度”“HTTP常见面试题  ”。嵌入式稍作了解即可。

## 3.1.7 URI（统一资源标识符）和URL（统一资源定位符）之间的区别

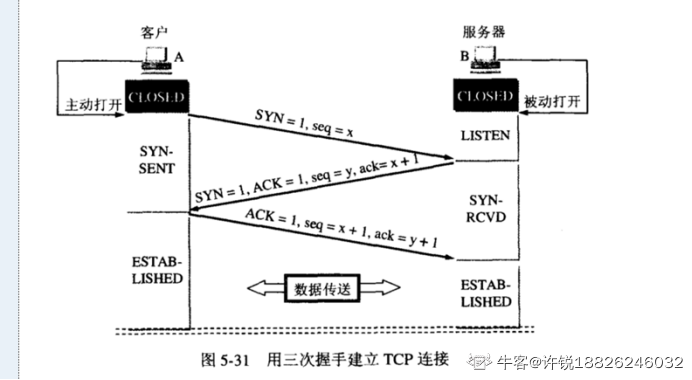
解析：

你可能觉得URI和URL可能是相同的概念，其实并不是，URI和URL都定义了资源是什么，但URL还定义了该如何访问资源。URL是一种具体的

URI，它是URI的一个子集，它不仅唯一标识资源，而且还提供了定位该资源的信息。URI是一种语义上的抽象概念，可以是绝对的，也可以是相对的，而URL则必须提供足够的信息来定位，是绝对的。

## 3.2.1什么是三次握手⭐⭐⭐⭐⭐

解析：第一次握手客户跟服务器说要创建链接，第二次，服务器同意客户端的链接，然后要再和客户端确认一遍是不是真要链接，第三次客户端说真要链接，这才链接。如下图：



**上图需要会画出来，很可能要你现场画哦！！**

最初两端的TCP进程都处于CLOSED关闭状态，A主动打开连接，而B被动打开连接。B的TCP服务器进程先创建传输控制块TCB，准备接受客户进程的连接请求。然后服务器进程就处于LISTEN（收听）状态，等待客户的连接请求。若有，则作出响应。

**第一次握手**：起初两端都处于CLOSED关闭状态，**：**A的TCP客户进程也是首先创建传输控制块TCB，然后向B发出连接请求报文段，Client将标志位SYN置为1，随机产生一个值seq=x，并将该数据包发送给Server，Client进入SYN-SENT状态，等待Server确认；

**第二次握手**：Server收到数据包后由标志位SYN=1得知Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=x+1，随机产生一个值seq=y，并将该数据包发送给Client以确认连接请求，Server进入SYN-RCVD状态，此时操作系统为该TCP连接分配TCP缓存和变量；

**第三次握手**：Client收到确认后，检查ack是否为x+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=y+1，并且此时操作系统为该TCP连接分配TCP缓存和变量，并将该数据包发送给Server，Server检查ack是否为y+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client和Server就可以开始传输数据。

**注意**：Server第二次握手将ACK置一，且进行资源分配

            Client第三次握手将ACK置一，且进行资源分配

## 3.2.2为什么三次握手中客户端还要发送一次确认呢？可以二次握手吗？

答：**主要为了防止已失效的连接请求报文段突然又传送到了B，因而产生错误**。如A发出连接请求，但因连接请求报文丢失而未收到确认，于是A再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接，A工发出了两个连接请求报文段，其中第一个丢失，第二个到达了B，但是第一个丢失的报文段只是在**某些网络结点长时间滞留了，延误到连接释放以后的某个时间才到达B**，此时B误认为A又发出一次新的连接请求，于是就向A发出确认报文段，同意建立连接，不采用三次握手，只要B发出确认，就建立新的连接了，此时A不理睬B的确认且不发送数据，则B一致等待A发送数据，浪费资源。

网上看到的其他答案：C是Client，S是Server。

答：现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机S和C之间的通信，假定C给S发送一个连接请求分组，S收到了这个分组，并发 送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，C在S的应答分组在传输中被丢失的情况下，将不知道S是否已准备好，不知道S建立什么样的序列号，C甚至怀疑S是否收到自己的连接请求分组。在这种情况下，C认为连接还未建立成功，将忽略S发来的任何数据分 组，只等待连接确认应答分组。而S在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

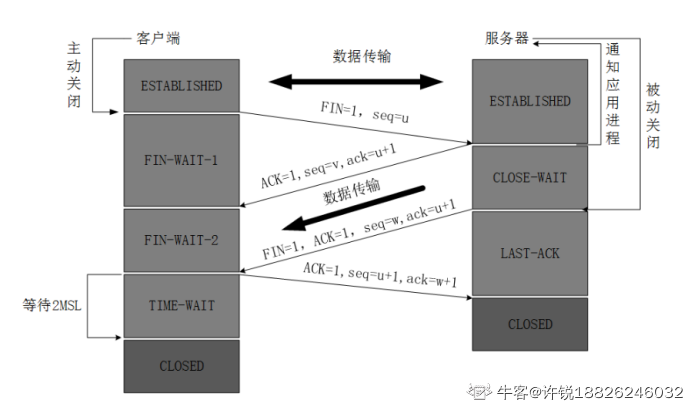
## 3.2.3为什么服务端易受到SYN攻击？

答：服务器端的资源分配是在二次握手时分配的，而客户端的资源是在完成三次握手时分配的，所以服务器容易受到SYN洪泛攻击，SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server则回复确认包，并等待Client确认，由于源地址不存在，因此Server需要不断重发直至超时，这些伪造的SYN包将长时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络拥塞甚至系统瘫痪。

防范SYN攻击措施：降低主机的等待时间使主机尽快的释放半连接的占用，短时间受到某IP的重复SYN则丢弃后续请求。

## 3.2.4什么是四次挥手

解析：



1客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为seq=u（等于前面已经传送过来的数据的最后一个字节的序号加1）吧要消耗一个序号。

2服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号seq=v，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

3客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

4服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为seq=w，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。

5客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2∗ ∗\*MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。

6服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

## 3.2.5为什么客户端最后还要等待2MSL？

解析：MSL（Maximum Segment Lifetime），TCP允许不同的实现可以设置不同的MSL值。

  第一，保证客户端发送的最后一个ACK报文能够到达服务器，因为这个ACK报文可能丢失，站在服务器的角度看来，我已经发送了FIN+ACK报文请求断开了，客户端还没有给我回应，应该是我发送的请求断开报文它没有收到，于是服务器又会重新发送一次，而客户端就能在这个2MSL时间段内收到这个重传的报文，接着给出回应报文，并且会重启2MSL计时器。

  第二，防止类似与“三次握手”中提到了的“已经失效的连接请求报文段”出现在本连接中。客户端发送完最后一个确认报文后，在这个2MSL时间中，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样新的连接中不会出现旧连接的请求报文。

## 3.2.6为什么建立连接是三次握手，关闭连接确是四次挥手呢？

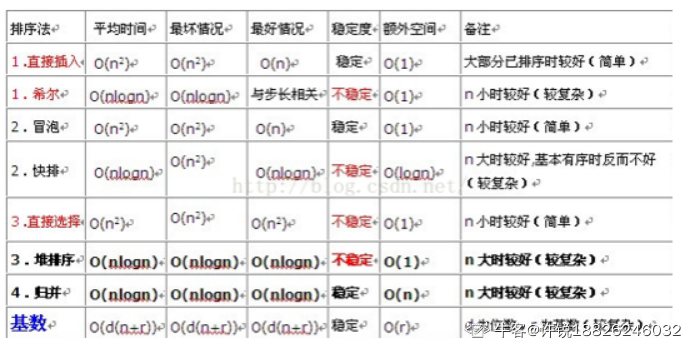
    解析：建立连接的时候，服务器在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。

    而关闭连接时，服务器收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送，从而导致多了一次。

# 4.1 排序算法

**本人算法题目主要是看《剑指offer》，并在牛客网同步刷《剑指offer》专栏的题目。**

## 4.1.1各种排序算法的时间空间复杂度、稳定性（这个表格很重要，一定要记清楚）



问：什么是稳定性？

答：排序算法稳定性的简单形式化定义为：如果Ai = Aj，排序前Ai在Aj之前，排序后Ai还在Aj之前，则称这种排序算法是稳定的。通俗地讲就是保证排序前后两个相等的数的相对顺序不变。对于不稳定的排序算法，只要举出一个实例，即可说明它的不稳定性；而对于稳定的排序算法，必须对算法进行分析从而得到稳定的特性。需要注意的是，排序算法是否为稳定的是由具体算法决定的，不稳定的算法在某种条件下可以变为稳定的算法，而稳定的算法在某种条件下也可以变为不稳定的算法。例如，对于冒泡排序，原本是稳定的排序算法，如果将记录交换的条件改成A[i] >= A[i + 1]，则两个相等的记录就会交换位置，从而变成不稳定的排序算法。

稳定性：（快些选一堆美女一起玩儿）是不稳定的，其他都是稳定的。

快速排序（快），希尔排序（些），选择排序（选），堆排序（一堆）

总结：

稳定排序：  插入排序，冒泡排序，二叉树排序，归并排序

不稳定排序： 快速排序（快），希尔排序（些），选择排序（选），堆排序（一堆）。

Ps：还有一种排序称为“桶排序”，虽然不如以上算法常见，但本人曾被问到，刚好不懂，读者引以为戒哈。

## 4.1.2各种排序算法什么时候有最好情况、最坏情况（尤其是快排）

解析：

    1）冒泡排序经过优化以后，最好时间复杂度可以达到O(n)。设置一个标志位，如果有一趟比较中没有发生任何交换，可提前结束，因此在正序情况下，时间复杂度为O(n)。

    2）选择排序在最坏和最好情况下，都必须在剩余的序列中选择最小（大）的数，与已排好序的序列后一个位置元素做交换，依次最好和最坏时间复杂度均为O(n^2)。

   3） 插入排序是在把已排好序的序列的后一个元素插入到前面已排好序(需要选择合适的位置)的序列中，在正序情况下时间复杂度为O(n)。

    4）堆是完全二叉树，因此树的深度一定是log(n)+1，最好和最坏时间复杂度均为O(n\*log(n))。

    5）归并排序是将大数组分为两个小数组，依次递归，相当于二叉树，深度为log(n)+1，因此最好和最坏时间复杂度都是O(n\*log(n))。

    6）当快速排序在正序或逆序情况下，每次划分只得到比上一次划分少一个记录的子序列，用递归树画出来，是一棵斜树，此时需要n-1次递归，且第i次划分要经过n-i次关键字比较才能找到第i个记录，因此时间复杂度是\sum\_{i=1}^{n-1}(n-i)=n(n-1)/2，即O(n^2)。

    出现以上原因是因为基本的快速排序选取第一个元素作为主元，所以可以改进为“随机化快速排序”即随机选择一个元素作为主元，这种情况下虽然最坏情况仍然是O(n^2)，但最坏情况不再依赖于输入数据，而是由于随机函数取值不佳。实际上，随机化快速排序得到理论最坏情况的可能性仅为1/(2^n)。所以随机化快速排序可以对于绝大多数输入数据达到O(nlogn)的期望时间复杂度。随机化快速排序的唯一缺点在于，一旦输入数据中有很多的相同数据，随机化的效果将直接减弱。对于极限情况，即对于n个相同的数排序，随机化快速排序的时间复杂度将毫无疑问的降低到O(n^2)。解决方法是用一种方法进行扫描，使没有交换的情况下主元保留在原位置。

## 4.1.3冒泡排序

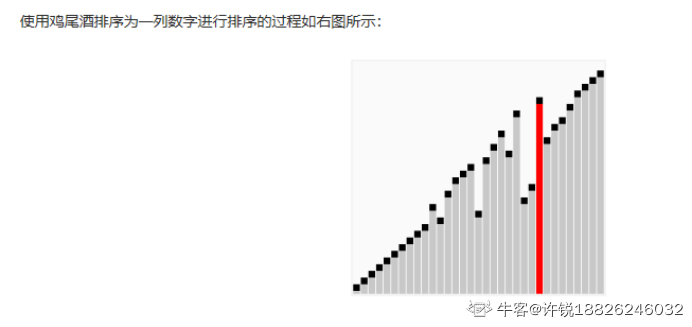
**1.1冒泡排序**

算法的运作如下：比较相邻的元素，如果前一个比后一个大，就把它们两个调换位置。对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。针对所有的元素重复以上的步骤，除了最后一个。持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。注意：第二个for循环j < length - i - 1要注意**再-1**哦

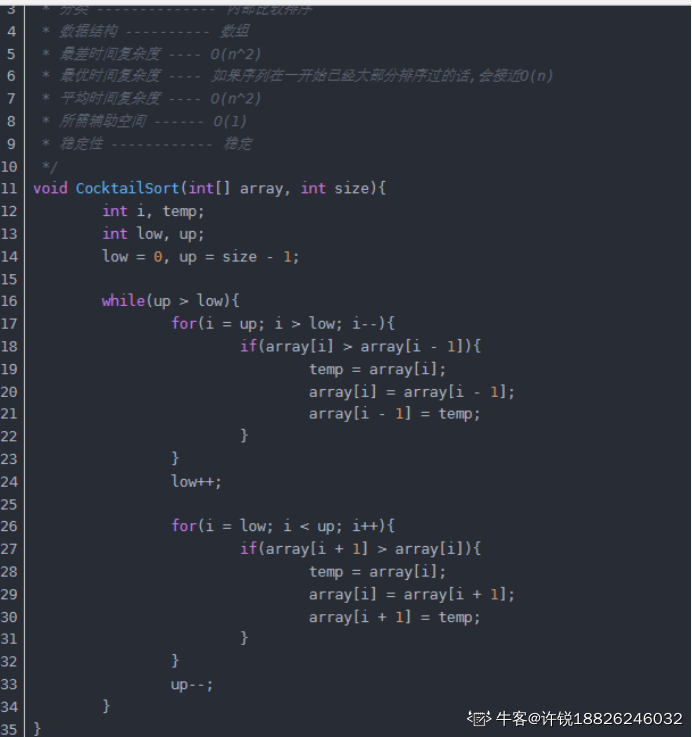


1.2冒泡排序的改进：鸡尾酒排序（本人不曾被问到，不过如果能主动说出来会有加分）

    鸡尾酒排序，也叫定向冒泡排序，是冒泡排序的一种改进。此算法与冒泡排序的不同处在于从低到高然后从高到低，而冒泡排序则仅从低到高去比较序列里的每个元素。他可以得到比冒泡排序稍微好一点的效能。

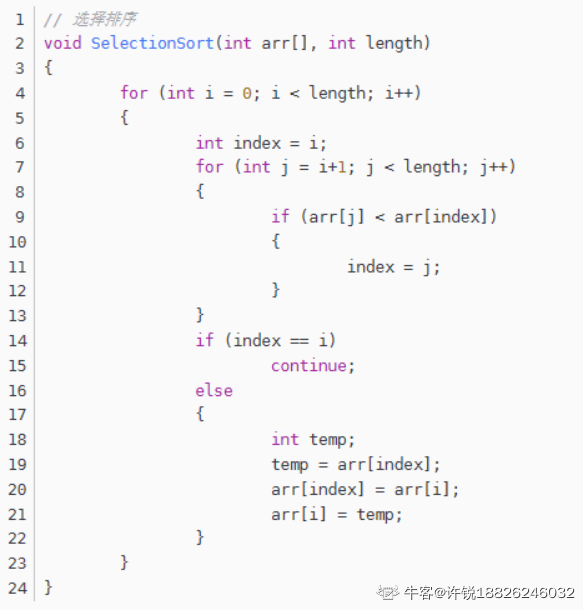


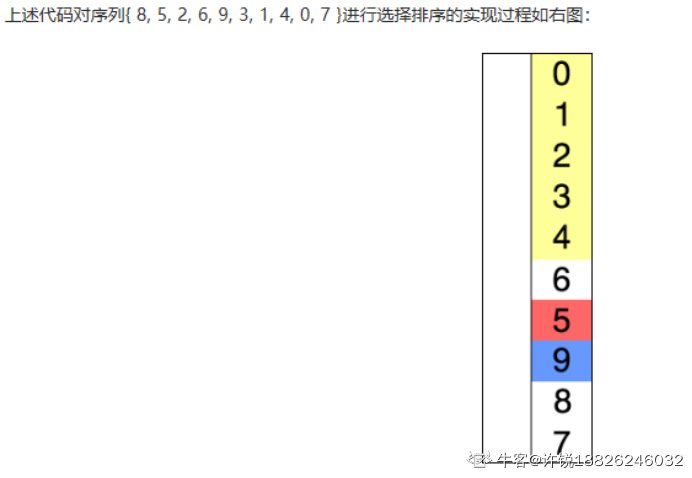
**感谢**：感谢 [牛客502009481号](https://www.nowcoder.com/profile/502009481) 指出下图11行，int[] array应该改为int\* array ！！！



## 4.1.4选择排序

    选择排序也是一种简单直观的排序算法。它的工作原理很容易理解：初始时在序列中找到最小（大）元素，放到序列的起始位置作为已排序序列；然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。注意选择排序与冒泡排序的区别：冒泡排序通过依次交换相邻两个顺序不合法的元素位置，从而将当前最小（大）元素放到合适的位置；而选择排序每遍历一次都记住了当前最小（大）元素的位置，最后仅需一次交换操作即可将其放到合适的位置。





## 4.1.5插入排序

具体算法描述如下：

1）从第一个元素开始，该元素可以认为已经被排序

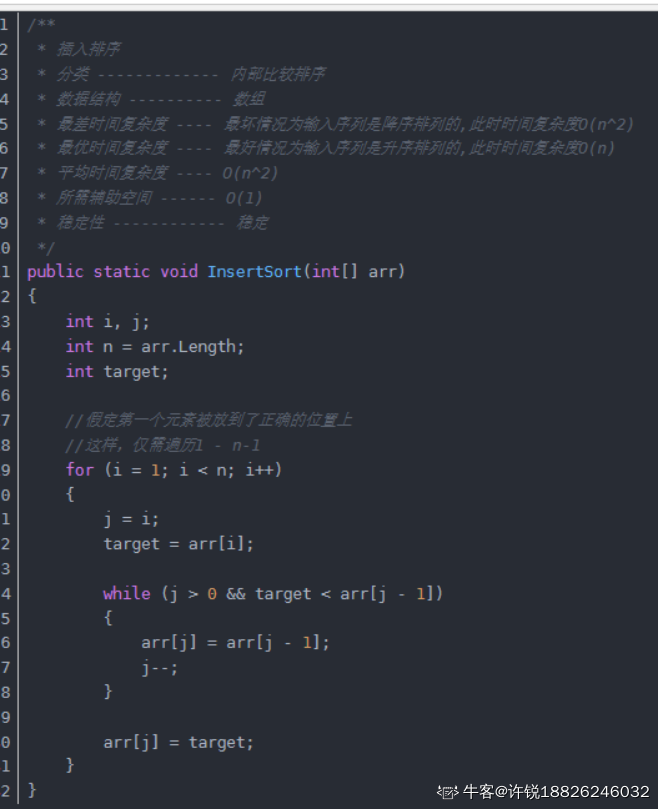
2）取出下一个元素，在已经排序的元素序列中从后向前扫描

3）如果该元素（已排序）大于新元素，将该元素移到下一位置

4）重复步骤3，直到找到已排序的元素小于或者等于新元素的位置

5）将新元素插入到该位置后

6）重复步骤2~5



**插入排序的改进：二分插入排序**

**改进的就是在查找插入的位置的时候，不要按顺序循环查找改用二分查找**

**感谢：Iron20180918173256牛友指导，重要的事情说3遍。**

**下图中间区域的 left = left + 1 改为 left = mid + 1; right = right - 1 改为 right = mid - 1**

**下图中间区域的 left = left + 1 改为 left = mid + 1; right = right - 1 改为 right = mid - 1**

**下图中间区域的 left = left + 1 改为 left = mid + 1; right = right - 1 改为 right = mid - 1**



## 4.1.6希尔排序

希尔排序(Shell's Sort)在插入排序算法的基础上进行了改进，算法的时间复杂度与前面几种算法相比有较大的改进。其算法的基本思想是：先将待排记录序列分割成为若干子序列分别进行插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行一次直接插入排序。



[复制代码](https://blog.nowcoder.net/n/2b9bfa8c73524827bd3ac9fdc75c52d8)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **void** ShellSort(**int** arr[], **int** length)  {  **int** increasement = length;  **int** i, j, k;  **do**{      //确定分组的增量          increasement = increasement / 3 + 1;  **for** (i = 0; i < increasement; i++)          {  **for** (j = i + increasement; j < length; j += increasement)              {  **if** (arr[j] < arr[j - increasement])                  {  **int** temp = arr[j];  **for** (k = j - increasement; k >= 0 && temp < arr[k]; k -= increasement)                      {                          arr[k + increasement] = arr[k];                      }                      arr[k + increasement] = temp;                  }              }          }          } **while** (increasement > 1);  }    **int** main()  {  **int** a[8] = { 6, 5, 3, 1, 8, 7, 2, 4 };      ShellSort(a, 8);  **for**(**int** i = 0; i < 8; ++i)      {          cout<<a[i]<<" ";      }  **return** 0;  } |

## 4.1.7归并排序(Merge Sort)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | **void** Merge(**int** arr[], **int** start, **int** mid, **int** end, **int** \* temp)  {      //合并两个有序序列  **int** length = 0;      //表示辅助空间有多少个元素  **int** i\_start = start;  **int** i\_end = mid;  **int** j\_start = mid + 1;  **int** j\_end = end;  **while** (i\_start <= i\_end && j\_start <= j\_end)      {  **if** (arr[i\_start] < arr[j\_start])          {              temp[length] = arr[i\_start];              length++;              i\_start++;          }  **else**          {              temp[length] = arr[j\_start];              length++;              j\_start++;          }      }  **while** (i\_start <= i\_end)      {          temp[length] = arr[i\_start];          i\_start++;          length++;      }    **while** (j\_start <= j\_end)      {          temp[length] = arr[j\_start];          length++;          j\_start++;      }        //把辅助空间的数据放到原空间  **for** (**int** i = 0; i < length; i++)      {          arr[start + i] = temp[i];      }  }    **void** MergeSort(**int** arr[], **int** start, **int** end, **int** \* temp)  {  **if** (start >= end)  **return**;  **int** mid = (start + end) / 2;      MergeSort(arr, start, mid, temp);      MergeSort(arr, mid + 1, end, temp);      Merge(arr, start, mid, end, temp);  }    **int** main()  {  **int** a[8] = { 6, 5, 3, 1, 8, 7, 2, 4 };  **int** b[8] = { 0 };      MergeSort(a, 0, 8 - 1, b);  //要-1  **for**(**int** i = 0; i < 8; ++i)      {          cout<<b[i]<<" ";      }  **return** 0;  } |

无论是原数组a还是辅助数组b，都可以如下输出。



## 4.1.8快速排序 (Quick Sort)

1）从序列中挑出一个元素，作为”基准”(pivot).

2）把所有比基准值小的元素放在基准前面，所有比基准值大的元素放在基准的后面（相同的数可以到任一边），这个称为分区(partition)操作。

3）对每个分区递归地进行步骤1~2，递归的结束条件是序列的大小是0或1，这时整体已经被排好序了。

快速排序分为非随机快排和随机快排，代码如下：

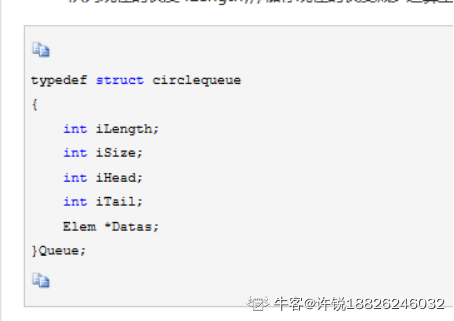
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107 | /\*\*\*\*\*\*\*\*\*\*\*\*\*非随机化快速排序代码如下：\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **int** Partition(**int**\* arr, **int** length, **int** start, **int** end)  {  **if**(NULL == arr || length <= 0 || start < 0 || end <= 0 || end >= length)  **return** -1;  **int** baseVal = arr[start];  **int** m\_start = start;  **int** m\_end = end;  **while**(m\_start < m\_end) //非随机快排和随机快排的循环是不一样的。      {  **while**(m\_start < m\_end && arr[m\_end] >= baseVal) //从右向左找比基准数小的数     //至少要有个是有“等于”的              --m\_end;  **if**(m\_start < m\_end)          {              arr[m\_start] = arr[m\_end];              ++m\_start;          }    **while**(m\_start < m\_end && arr[m\_start] < baseVal) //从左向右找比基准数大的数              ++m\_start;  **if**(m\_start < m\_end)          {              arr[m\_end] = arr[m\_start];              --m\_end;          }      }      arr[m\_start] = baseVal;//把基准数放到i的位置  **return** m\_start;  }  **void** QuickSort(**int**\* arr, **int** length, **int** start, **int** end)  //非随机快排和随机快排只有partition函数不同，此时还是相同的  {  **if**(start == end)  **return**;  **int** index = Partition(arr, length, start, end);  **if**(index > start)          QuickSort(arr, length, start, index - 1);  **if**(index < end)          QuickSort(arr, length , index + 1, end);  }  **int** main()  {  **int** arr[] = {          5,8,8,65,3,4,71,2,5,120, 546      };      QuickSort(arr, **sizeof**(arr)/**sizeof**(**int**), 0, **sizeof**(arr)/**sizeof**(**int**) - 1);  **for**(**int** i = 0; i < **sizeof**(arr)/**sizeof**(**int**); ++i)      {          cout<<arr[i]<<" ";      }  **return** 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*随机化快速排序代码如下：\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*随机化快速排序代码如下：\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*随机化快速排序代码如下：\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*随机化快速排序代码如下：\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **void** Swap(**int**\* a, **int**\* b)  {  **int** temp = \*a;      \*a = \*b;      \*b = temp;  }  **int** Partition(**int** arr[], **int** length, **int** start, **int** end)  {  **if**(arr == NULL || start < 0 || end >= length || length <= 0)      {  **return** -1;          cout<<"error"<<endl;      }  **int** random = **rand**() % (end - start + 1) + start;  **int** small = start - 1;      Swap(&arr[random], &arr[end]);  **for**(**int** i = start; i < end; ++i)  //非随机快排和随机快排的循环是不一样的。      {  **if**(arr[i] < arr[end])          {              ++small;  **if**(i != small)              {                  Swap(&arr[small], &arr[i]);              }          }      }      ++small;      Swap(&arr[small], &arr[end]);  **return** small;  //曾经我返回的是random调了很久才发现是small  }  **void** QuickSort(**int** arr[], **int** length, **int** start, **int** end)  {  **if**(start == end)  **return** ；  **int** index = Partition(arr, length, start, end);  **if**(index > start)          QuickSort(arr, length, start, index - 1);  **if**(index < end)          QuickSort(arr, length, index + 1, end);  }  **int** main()  {  **int** a[8] = { 6, 5, 3, 1, 8, 7, 2, 4 };      QuickSort(a, 8, 0, 7);  **for**(**int** i = 0; i < 8; ++i)      {          cout<<a[i]<<" ";      }  **return** 0;  } |

## 4.1.9快排的partition函数与归并的Merge函数

解析：本人曾被问到这个问题，所以需要对快排和归并排序提前了解好，读者多看上述两种排序的代码多理解即可。

## 4.1.10循环队列的实现

解析：循环队列虽然不是常见的排序算法，但是本人也曾2次被遇到，并且有一次在广州敏视是手撕代码，现在写出来，还好有提前准备。**循环队列的实现主要有push、front、pop接口。**



结构体可以这么设计，循环队列就是要有一个头指针，一个尾指针，还是以链表的形式设计，队列空和满的时候，头尾指针都是相等的，此时通过判断length，length为0代表空，不为零代表满。入队时从队尾添加，出队时从对头出。

## 4.2.1 vector list异同

解析：

1）数据结构的区别

vector与数组类似，拥有一段连续的内存空间，并且起始地址不变。便于随机访问，时间复杂度为O（1），但因为内存空间是连续的，所以在进入插入和删除操作时，会造成内存块的拷贝，时间复杂度为O（n）。此外，当数组内存空间不足，会采取扩容，通过重新申请一块更大的内存空间进行内存拷贝。

list底层是由双向链表实现的，因此内存空间不是连续的。根据链表的实现原理，List查询效率较低，时间复杂度为O（n），但插入和删除效率较高。只需要在插入的地方更改指针的指向即可，不用移动数据。

2）迭代器支持不同

异：vector中，iterator支持”+“、”+=“，”<"等操作。而list中则不支持。

同：vector<int>::iterator和list<int>::iterator都重载了“++ ”操作。

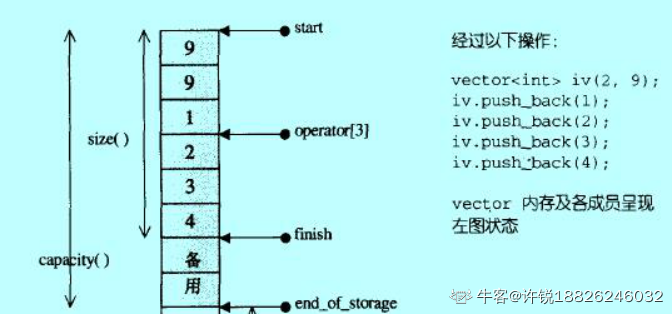
## 4.2.2 vector内存是怎么增长的vector的底层实现

**vector空间的动态增长**

    当添加元素时，如果vector空间大小不足，则会以原大小的两倍另外配置一块较大的新空间，然后将原空间内容拷贝过来，在新空间的内容末尾添加元素，并释放原空间。vector的空间动态增加大小，并不是在原空间之后的相邻地址增加新空间，因为vector的空间是线性连续分配的，不能保证原空间之后有可供配置的空间。因此，对vector的任何操作，一旦引起空间的重新配置，指向原vector的所有迭代器就会失效。

**vector的size(),capacity(),reserve(),resize()函数**

    vector对象的内存布局如下图所示。



     start迭代器指向已用空间的首元素，finish指向已用空间的尾元素的下一个位置，end\_of\_storage指向可用空间的末尾。

     size()函数返回的是已用空间大小，capacity()返回的是总空间大小，capacity()-size()则是剩余的可用空间大小。当size()和capacity()相等，说明vector目前的空间已被用完，如果再添加新元素，则会引起vector空间的动态增长。

     由于动态增长会引起重新分配内存空间、拷贝原空间、释放原空间，这些过程会降低程序效率。因此，可以使用reserve(n)预先分配一块较大的指定大小的内存空间，这样当指定大小的内存空间未使用完时，是不会重新分配内存空间的，这样便提升了效率。只有当n>capacity()时，调用reserve(n)才会改变vector容量。

    resize()成员函数只改变元素的数目，不改变vector的容量。

**结论：**

    1.空的vector对象，size()和capacity()都为0

    2.当空间大小不足时，新分配的空间大小为原空间大小的2倍。

    3.使用reserve()预先分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率。

    4.当reserve()分配的空间比原空间小时，是不会引起重新分配的。比如此时capacity是20，只reserve（15）是不会重新分配的

    5. resize()函数只改变容器的元素数目，未改变容器大小。

## 4.2.3为什么stl里面有sort函数list里面还要再定义一个sort

**解析：这是本人在面试CVTE遇到的面试题。**

1、支持随机存取迭代器的（连续存储空间）vector、deque(双向存取vector)使用STL的sort函数。

2、不支持随机存取迭代器的（链式非连续存储空间）list(双向链表)、slist(单向链表forward\_list)，不能使用STL的sort函数，因此都会在类中定义sort()成员函数，使用对象名调用即可。

3、关系型容器中基于红黑树的set、multiset、map、multimap，本身就有自动从大到小排序的功能。所以不需要sort函数。

4、stack、queue没有迭代器，各元素的出入口特定，不能进行排序。

5、基于哈希表的(hash)unordered\_set/multiset/map/multimap，都是未排序的，当然因为计算hash再存储的特性，也不需要进行排序。

## 4.2.4利用迭代器删除元素会发生什么？

解析：

（1）对于关联容器（如map，set，multimap，multiset），删除当前的iterator，仅仅会使当前的iterator失效，只要在erase时，递增当前的iterator即可。这是因为map之类的容器，使用了红黑树来实现，插入，删除一个结点不会对其他结点造成影响。使用方式如下例子：set<int> valset = { 1,2,3,4,5,6 };

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | set<**int**>::iterator iter;    **for** (iter = valset.begin(); iter != valset.end(); )  {  **if** (3 == \*iter)            valset.erase(iter++);  **else**            ++iter;  } |

因为传给erase的是iter的一个副本，iter++是下一个有效的迭代器。

（2）对于序列式容器（如vector，deque，list等），删除当前的iterator（所指向的元素）会使后面所有元素的iterator都失效。这是因为vector，deque使用了连续分配的内存，删除一个元素导致后面所有的元素会向前移动一个位置。不过erase方法可以返回下一个有效的iterator。使用方式如下,例如：vector<int> val = { 1,2,3,4,5,6 };

[复制代码](https://blog.nowcoder.net/n/0f92e6e2fdb44c9688a484a6c79d2e3d)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | vector<**int**>::iterator iter;    **for** (iter = val.begin(); iter != val.end(); )  {  **if** (3 == \*iter)            iter = val.erase(iter);     //返回下一个有效的迭代器，无需+1  **else**            ++iter;  } |

## 4.2.5  STL底层数据结构实现

解析：常见的vector，list，队列，set，map需要知道底层结构。

1.vector      底层数据结构为数组 ，支持快速随机访问

2.list            底层数据结构为双向链表，支持快速增删

3.deque       底层数据结构为一个中央控制器和多个缓冲区，详细见STL源码剖析P146，支持首尾（中间不能）快速增删，也支持随机访问 也是连续数组，可以理解为双向vecvtor

deque是一个双端队列(double-ended queue)，也是在堆中保存内容的.它的保存形式如下:[堆1] --> [堆2] -->[堆3] --> ...

    每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是list和vector的结合品.

4.stack        底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时

5.queue     底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时（stack和queue其实是适配器,而不叫容器，因为是对容器的再封装）

6.priority\_queue     的底层数据结构一般为vector为底层容器，堆heap为处理规则来管理底层容器实现

7.set                   底层数据结构为红黑树，有序，不重复

8.multiset           底层数据结构为红黑树，有序，可重复

9.map                底层数据结构为红黑树，有序，不重复

10.multimap      底层数据结构为红黑树，有序，可重复

11.hash\_set      底层数据结构为hash表，无序，不重复

12.hash\_multisey   底层数据结构为hash表，无序，可重复

13.hash\_map         底层数据结构为hash表，无序，不重复

14.hash\_multimap 底层数据结构为hash表，无序，可重复

**Ps：queue stack priority\_queue（这三个不是容器，而是适配器，因为是对容器的再封装）没有迭代器，其他都有。**

## 4.2.6 vector和deque的比较

解析：

1）vector.at()比deque.at()效率高，应该是因为deque的开始位置不是固定的；

2）deque支持头部快速插入删除vector头部插入删除效率很慢 （vector好像没有push\_front的函数）；

3）需要大量释放操作，vector更快。

## 4.2.7 map是如何实现的，查找效率是多少

答：红黑树实现，它可以在O(log n)时间内做查找、插入和删除

## 4.2.8几种模板插入的时间复杂度

**主要的有set和map插入删除是O（logn）  其他头尾插入删除是O（1） 中间是O(n)**

# 5.1 Linux内核相关

## 5.1.1 了解Linux内核吗？（内核的组成）

解析：

UNIX系统由内核、shell、文件系统（系统调用和共用函数库）和应用程序等4部分组成。

一．Linux内核

内核是操作系统的核心，具有很多最基本功能，如虚拟内存、多任务、共享库、需求加载、可执行程序和TCP/IP网络功能。Linux内核的模块分为以下几个部分：存储管理、CPU和进程管理、文件系统、设备管理和驱动、网络通信、系统的初始化和系统调用等。

二．Linux shell

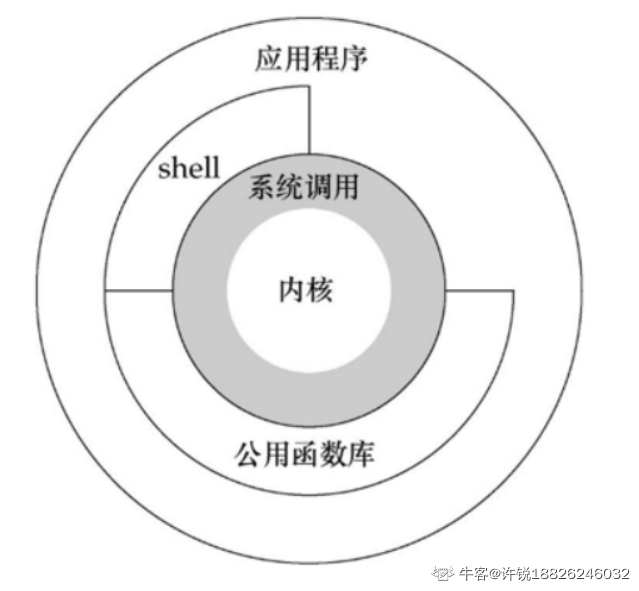
shell是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行，是一个命令解释器。另外，shell编程语言具有普通编程语言的很多特点，用这种编程语言编写的shell程序与其他应用程序具有同样的效果。

三．Linux文件系统

文件系统是文件存放在磁盘等存储设备上的组织方法。Linux系统能支持多种目前流行的文件系统，如EXT2、EXT3、FAT、FAT32、VFAT和ISO9660。

四．Linux应用程序

标准的Linux系统一般都有一套都有称为应用程序的程序集，它包括文本编辑器、编程语言、XWindow、办公套件、Internet工具和数据库等。



内核分为  进程管理系统 、 内存管理系统 、 I/O管理系统和文件管理系统 等四个子系统

## 5.1.2用户空间与内核通信方式有哪些？

解析：

      1)首先想到的是系统调用，用户空间进程通过系统调用进入内核空间，访问指定的内核空间数据；

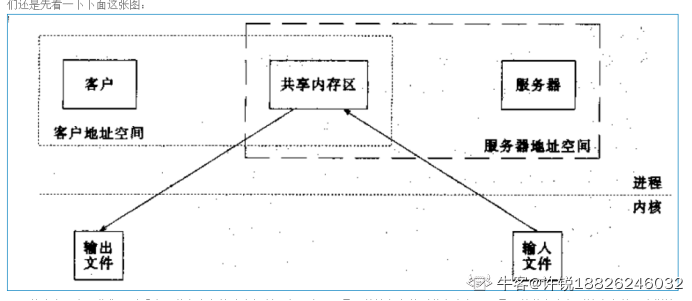
      2).其次是驱动程序，用户空间进程可以使用封装后的系统调用接口访问驱动设备节点，以和运行在内和空间的驱动程序通信；

      3).共享内存mmap，在代码中调用接口，实现内核空间与用户空间的地址映射，在实时性要求很高的项目中为首选，省去拷贝数据的时间等资源，但缺点是不好控制；

      4).最后，copy\_to\_user()、copy\_from\_user()，是在驱动程序中调用接口，实现用户空间与内核空间的数据拷贝操作，应用于实时性要求不高的项目中。

**问： IPC（进程间通信）机制中最快的，可以跨进程的是什么？**

**答：共享内存。**



我们可以看出，共享内存的消息复制只有两次。一是，从输入文件到共享内存；二是，从共享内存到输出文件。这样就很大程度上提高了数据存取的效率。

**以下这个问题忘了哪里看到的，当时也保存下来，但比较少接触到，读者当作阅读看看即可。**

**问：Linux的用户态与内核态的转换方法？**

**解答**：Linux下内核空间与用户空间进行通信的方式主要有syscall（system call）、procfs、ioctl和netlink等。还有copy\_to\_user  copy\_from\_user共享内存mmap  驱动

1）syscall：一般情况下，用户进程是不能访问内核的。它既不能访问内核所在的内存空间，也不能调用内核中的函数。Linux内核中设置了一组用于实现各种系统功能的子程序，用户可以通过调用他们访问linux内核的数据和函数，这些系统调用接口（SCI）称为系统调用；

2）procfs：是一种特殊的伪文件系统 ，是Linux内核信息的抽象文件接口，大量内核中的信息以及可调参数都被作为常规文件映射到一个目录树中，这样我们就可以简单直接的通过echo或cat这样的文件操作命令对系统信息进行查取；应该就是根目录里的/proc

3）netlink：用户态应用使用标准的socket API就可以使用netlink提供的强大功能；

4）ioctl：函数是文件结构中的一个属性分量，就是说如果你的驱动程序提供了对ioctl的支持，用户就可以在用户程序中使用ioctl函数控制设备的I/O通道。

在这几个通信方式中，**效率最高的方式**，选择netlink，原因如下：

全双工：procfs是基于文件系统，用于内核向用户发送消息；syscall是用户访问内核。它们都是单工通信方式。netlink是一种特殊的通信方式，用于在内核空间和用户空间传递消息，是一种双工通信方式。使用地址协议簇AF\_NETLINK，使用头文件include/linux/netlink.h；

易于添加：为新特性添加system call、或者procfs是一件复杂的工作，它们会污染kernel（内核），破坏系统的稳定性，这是非常危险的。Netlink的添加，对内核的影响仅在于向netlink.h中添加一个固定的协议类型，然后内核模块和应用层的通信使用一套标准的API。

**感谢**：感谢牛友寞明，指出5.1.2我总共整理开头4个和下面4个总共8个知识点，有点懵（可见评论），在此解释一下：下面4点中syscall就是系统调用，对应上面的第1点系统调用，ioctl那里写着“用户就可以在用户程序中使用ioctl函数控制设备的I/O通道”，对应上面的第2点驱动程序，这个你要自己写过驱动才能理解ioctl的应用，其他两个我其实还没接触过，水平有限。所以说“读者当作阅读看看即可”，面试问到，回答上面4点即可。

## 5.1.3系统调用read()/write()，内核具体做了哪些事情

解析：本题是本人曾亲自被问到，但感觉还是比较少人会问此问题，**所以可能其他人的面经没有这个问题，读者可稍微了解即可。**

 用户空间read()-->内核空间sys\_read()-->scull\_fops.read-->scull\_read()；

该过程分为两个部分：用户空间的处理和核心空间的处理。在用户空间中通过 0x80 中断的方式将控制权交给内核处理，内核接管后，经过6个层次的处理最后将请求交给磁盘，由磁盘完成最终的数据拷贝操作。在这个过程中，调用了一系列的内核函数

## 5.1.4 系统调用的作用

解析：

1、为了管理硬件资源和为应用程序开发人员提供良好的环境来使应用程序具有更好的兼容性，为了达到这个目的，内核提供一系列具备预定功能的多内核函数，通过一组称为系统调用（system call)的接口呈现给用户。系统调用把应用程序的请求传给内核，调用相应的的内核函数完成所需的处理，将处理结果返回给应用程序。

2、具有多任务处理的功能，通常靠进程来实现。

3、为了安全问题，一些I/O操作的指令都被限制在只有内核模式可以执行，因此操作系统有必要提供接口来为应用程序提供诸如读取磁盘某位置的数据的接口，这些接口就被称为系统调用。

4、当操作系统接收到系统调用请求后，会让处理器进入内核模式，从而执行诸如I/O操作，修改基址寄存器内容等指令，而当处理完系统调用内容后，操作系统会让处理器返回用户模式，来执行用户代码。

## 5.1.5内核态，用户态的区别

**解析：**

     系统态(也称为管态或核心态)，操作系统在系统态运行——运行操作系统程序

    用户态(也称为目态)，应用程序只能在用户态运行——运行用户程序

     当一个进程在执行用户自己的代码时处于用户运行态（用户态），此时特权级最低，为3级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态。Ring3状态不能访问Ring0的地址空间，包括代码和数据；当一个进程因为系统调用陷入内核代码中执行时处于内核运行态（内核态），此时特权级最高，为0级。执行的内核代码会使用当前进程的内核栈，每个进程都有自己的内核栈。

    用户运行一个程序，该程序创建的进程开始时运行自己的代码，处于用户态。如果要执行文件操作、网络数据发送等操作必须通过write、send等系统调用，这些系统调用会调用内核的代码。进程会切换到Ring0，然后进入3G-4G中的内核地址空间去执行内核代码来完成相应的操作。内核态的进程执行完后又会切换到Ring3，回到用户态。这样，用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用。这说的保护模式是指通过内存页表操作等机制，保证进程间的地址空间不会互相冲突，一个进程的操作不会修改另一个进程地址空间中的数据。

## 5.1.6 bootloader内核 根文件的关系

**解析：**

启动顺序：bootloader->linuxkernel->rootfile

后者需要前者提供功能支持，前者的目的就是启动后者。

嵌入式Linux系统启动过程Linux引导的整个过程：当系统首次引导时，或系统被重置时，处理器会执行一个位于Flash/ROM中的已知位置处的代码，Bootloader就是这第一段代码。它主要用来初始化处理器及外设，然后调用Linux内核。Linux内核在完成系统的初始化之后需要挂载某个文件系统作为根文件系统（RootFilesystem），然后加载必要的内核模块，启动应用程序。（一个嵌入式Linux系统从软件角度看可以分为四个部分：引导加载程序（Bootloader），Linux内核，文件系统，应用程序。）

嵌入式Linux系统通过Bootloader引导，一上电，就要执行Bootloader来初始化系统。在完成对系统的初始化任务之后，它会将非易失性存储器（通常是Flash或DOC等）中的Linux内核拷贝到RAM中去，然后跳转到内核的第一条指令处继续执行，从而启动Linux内核。Bootloader和Linux内核有着密不可分的联系。

## 5.1.7 Bootloader多数有两个阶段的启动过程：

解析：

**Stage1:是汇编的**

1）基本的硬件初始化（关闭看门狗和中断，MMU（带操作系统），CACHE。 配置系统工作时钟）

2）为加载stage2准备RAM空间

3）拷贝内核映像和文件系统映像到RAM中

4）设置堆栈指针sp

5）跳到stage2的入口点

**Stage2:c语言的**

1）初始化本阶段要使用到的硬件设备（led uart等）

2）检测系统的内存映射

3）加载内核映像和文件系统映像

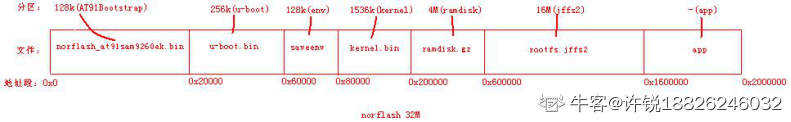
4）设置内核的启动参数

嵌入式系统中广泛采用的非易失性存储器通常是Flash，而Bootloader就位于该存储器的最前端，所以系统上电或复位后执行的第一段程序便是Bootloader。

## 5.1.8 linux的内核是由bootloader装载到内存中的？

解析：linux的内核的确是由bootloader装载到内存中的。linux的bootloader有2个部分组成：bootstrap和uboot。所以更准确点的说法是：linux的内核是由uboot装载到内存中的。内核文件本身是存放在硬盘的文件系统中，这句话就是错的。内核和文件系统是分开存储的。uboot读取kernel到内存是从kernel开始存储的地址开始读取的，而读取开始位置和读取大小，是由环境变量决定的。所以这个时候不需要文件系统的。

给你张图片，便于理解吧。这张图片是bootstrap、uboot、环境变量、kernel、文件系统在nandflash里面的存储分布



## 5.1.9为什么需要BootLoader

答：整个系统的加载启动任务就完全由BootLoader来完成。BootLoader就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

## 5.1.10 Linux内核同步方式总结

解析：

内核抢占：如果进程正在执行内核函数时（即它在内核态运行时），允许发生内核切换（被替换的进程是正在执行内核函数的进程），这个内核就是抢占的。

运行在内核态的进程可以自动放弃cpu，称为计划性进程切换，抢占式内核中，进程被迫放弃CPU，称为强制性进程切换。抢占内核的主要特点是：一个内核态运行的进程，可能在执行内核函数期间被另一个进程取代。

内核抢占就会发生一种情况：两个或两个以上的交叉内核路径嵌套时，就可能出现竞争条件。还有一种情况，多核环境下，多个core上的进程同时进入内核，就会出现访问的竞争。因此内核需要同步。内核同步技术主要有以下方式：

2.原子操作6.顺序锁8 信号量

## 5.1.11为什么自旋锁不能睡眠 而在拥有信号量时就可以？

解析：

自旋锁禁止处理器抢占；而信号量不禁止处理器抢占。

基于这个原因，如果自旋锁在锁住以后进入睡眠，由于不能进行处理器抢占，其他系统进程将都不能获得CPU而运行，因此不能唤醒睡眠的自旋锁，因此系统将不响应任何操作（除了中断或多核的情况，下面会讨论）。而信号量在临界区睡眠后，其他进程可以用抢占的方式继续运行，从而可以实现内存拷贝等功能而使得睡眠的信号量程序由于获得了等待的资源而被唤醒，从而恢复了正常的代码运行。

## 5.1.12 linux下检查内存状态的命令

1）top

2）free

3）cat /proc/meminfo

4）vmstat

问：linux中怎么查看堆栈

答：有两种方法：

第一种：pstack进程ID

第二种，使用gdb然后attach进程ID，然后再使用命令thread apply all bt

## 5.2.1大小端的区别以及各自的优点，哪种时候用？

解析：

大端优点：符号位在所表示的数据的内容的第一个字节中，便于快速判断数据的正负和大小

小端优点：低地址放低字节，所以在强制转换时不需要调整字节的内容。而且CPU做数值运算时从内存中依次从低到高取数据进行运算直到最后刷新最高位的符号位，这样运算方式会更高效。

以下是牛客网刷到的一个题目，觉得对理解大小端有帮助，就保存了下来，题目如下：



1）如果按照大端模式存储：从低地址到高地址：20 15 08 10

  存放高字节放低地址，输出从低地址到高地址：20 15 08 10 so 结果就是 2015  810

2）如果按照小端模式存储：从低地址到高地址：10 08 15 20

   存放高字节放高地址，输出从高地址到低地址：08 10 20 15  so 结果就是 810 2015

 Ps：

1）Interl x86系列芯片使用小端存储模式，ARM的芯片默认小端，但可以切换到大端；

2）网络上普遍采用大端模式，使用大端的CPU： power pc 、DSP … …

3）大小端是由CPU架构决定的， 不是软件决定！！！

## 5.2.2 一个程序从开始运行到结束的完整过程（四个过程）⭐⭐⭐⭐⭐

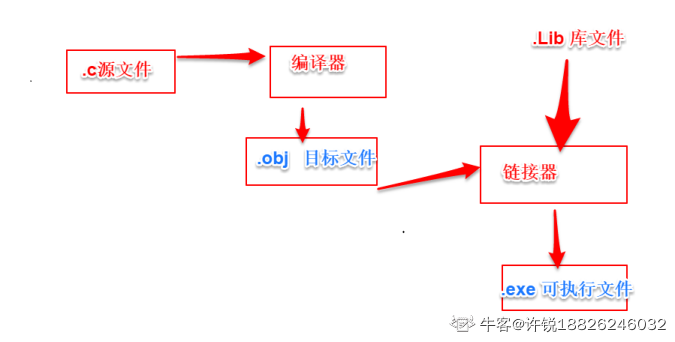
一个C语言程序到执行完文件的全过程，在linux里面用gcc编译的程序时，编译的过程可以细分为四个阶段：

1）预处理（Pre-Processing）

2）编译（Compiling）

3）汇编（Assembling）  
  
4）链接（Linking）

源程序到可执行程序的过程。在这个过程中，会发生如下的变化：



.c文件生成.obj文件的过程，称为编译，.obj文件生成到.exe文件的过程，称为链接。.obj文件就是一个是程序编译生成的二进制文件，当.exe文件生成以后.obj文件就会被删除。

1、预处理

在预编译的过程中，主要处理源代码中的预处理指令，引入头文件，去除注释，处理所有的条件编译指令，宏的替换，添加行号，保留所有的编译器指令。

当进行预编译以后的文件中将不再存在宏，所有的宏都已经被替代。当想要判断宏是否正确或者头文件包含是否正确时，也可以通过预编译来查看。

2、编译

在预处理结束后，进行的是编译。编译过程所进行的是对预处理后的文件进行语法分析，词法分析，语义分析，符号汇总，然后生成汇编代码。

3、汇编

汇编过程将汇编代码转成二进制文件，二进制文件就可以让机器来读取。每一条汇编语句都会产生一句机器语言。

4、链接

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数等等。所有这些问题，都需要经链接程序的处理方能得以解决。链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。

链接分为静态链接和动态链接：

静态链接：后缀是.a，主要在编译的时候将库文件里面代码搬迁到可执行的文件中；

动态链接：后缀是.so,主要在执行的时候需要转换到库文件代码执行；

两种链接的优缺点：

（1）静态的链接产生的可执行的文件体积比较的大；而动态链接的可执行文件的体积比较小；

（2）动态的链接的编译的效率比较的高；

（3）静态链接的可执行的文件执行的效率高

（4）静态链接的可执行的文件的“布局”比较好一点；

## 5.2.3什么是堆，栈，内存泄漏和内存溢出？

解析：

一般我们常说的内存泄漏是指堆内存的泄漏。堆内存是指程序从堆中分配的，大小任意的（内存块的大小可以在程序运行期决定），使用完后必须显式释放的内存。应用程序一般使用malloc，calloc，realloc，new等函数从堆中分配到一块内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了。

内存泄漏（一般指堆内存的泄漏）可以分为4 类：

1. 常发性内存泄漏

2. 偶发性内存泄漏

3. 一次性内存泄漏

4. 隐式内存泄漏

从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般的用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会最终消耗尽系统所有的内存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害性则非常大，因为较之于常发性和偶发性内存,泄漏它更难被检测到

**内存泄漏**是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。

一个盘子用尽各种方法只能装4 个果子，你装了5个，结果掉倒地上不能吃了。这就是溢出！比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。

**内存溢出**就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

**内存越界**：向系统申请了一块内存，而在使用内存时，超出了申请的范围（常见的有使用特定大小数组时发生内存越界）

注意：内存越界跟内存溢出的区别，前者是在使用系统提供的内存时，做了一些超出申请的内存范围的操作；而后者则是在申请内存大小时就已超出系统能提供的。  
    缓冲区溢出是指当计算机向缓冲区内填充数据位数时超过了缓冲区本身的容量溢出的数据覆盖在合法数据上,理想的情况是程序检查数据长度并不允许输入超过缓冲区长度的字符,但是绝大多数程序都会假设数据长度总是与所分配的储存空间相匹配,这就为缓冲区溢出埋下隐患.操作系统所使用的缓冲区又被称为"堆栈". 在各个操作进程之间,指令会被临时储存在"堆栈"当中,"堆栈"也会出现缓冲区溢出。

**栈溢出**就是缓冲区溢出的一种。由于缓冲区溢出而使得有用的存储单元被改写,往往会引发不可预料的后果。程序在运行过程中，为了临时存取数据的需要，一般都要分配一些内存空间，通常称这些空间为缓冲区。如果向缓冲区中写入超过其本身长度的数据，以致于缓冲区无法容纳，就会造成缓冲区以外的存储单元被改写，这种现象就称为缓冲区溢出。

**栈溢出**就是缓冲区溢出的一种。

**注意：**在程序员设计的代码中包含的“内存溢出”漏洞实在太多了。导致内存溢出问题的原因有很多，比如：  
(1) 使用非类型安全(non-type-safe)的语言如 C/C++ 等。  
(2) 以不可靠的方式存取或者复制内存缓冲区。  
(3) 编译器设置的内存缓冲区太靠近关键数据结构。  
下面来分析这些因素：  
    1. 内存溢出问题是 C 语言或者 C++ 语言所固有的缺陷，它们既不检查数组边界，又不检查类型可靠性(type-safety)。众所周知，用 C/C++ 语言开发的程序由于目标代码非常接近机器内核，因而能够直接访问内存和寄存器，这种特性大大提升了 C/C++ 语言代码的性能。只要合理编码，C/C++ 应用程序在执行效率上必然优于其它高级语言。然而，C/C++ 语言导致内存溢出问题的可能性也要大许多。其他语言也存在内容溢出问题，但它往往不是程序员的失误，而是应用程序的运行时环境出错所致。  
    2. 当应用程序读取用户(也可能是恶意攻击者)数据，试图复制到应用程序开辟的内存缓冲区中，却无法保证缓冲区的空间足够时(换言之，假设代码申请了 N 字节大小的内存缓冲区，随后又向其中复制超过 N 字节的数据)。内存缓冲区就可能会溢出。想一想，如果你向 12 盎司的玻璃杯中倒入 16 盎司水，那么多出来的 4 盎司水怎么办？当然会满到玻璃杯外面了！  
    3. 最重要的是，C/C++ 编译器开辟的内存缓冲区常常邻近重要的数据结构。现在假设某个函数的堆栈紧接在在内存缓冲区后面时，其中保存的函数返回地址就会与内存缓冲区相邻。此时，恶意攻击者就可以向内存缓冲区复制大量数据，从而使得内存缓冲区溢出并覆盖原先保存于堆栈中的函数返回地址。这样，函数的返回地址就被攻击者换成了他指定的数值；一旦函数调用完毕，就会继续执行“函数返回地址”处的代码。非但如此，C++ 的某些其它数据结构，比如 v-table 、例外事件处理程序、函数指针等，也可能受到类似的攻击。

## 5.2.4堆和栈的区别

解析：堆栈（英语：stack）又称为栈或堆叠，是计算机科学中一种特殊的串列形式的抽象数据类型，其特殊之处在于只能允许在链表或数组的一端（称为堆栈顶端指针，英语：top）进行加入数据（英语：push）和输出数据（英语：pop）的运算。另外堆栈也可以用一维数组或链表的形式来完成。堆栈的另外一个相对的操作方式称为队列。所以所谓的堆栈就是数据结构里的栈或者队列，而堆和栈就是进程内存空间里的堆和栈。

下面的图总结的非常好，读者可多复习。



## ****5.2.5 死锁的原因、条件 创建一个死锁，以及如何预防****

解析：面试中常问死锁的原因和必要条件，务必记清楚。死锁预防就当作提升。

产生死锁的原因主要是：

（1） 因为系统资源不足。

（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。

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

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。

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 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。

（3） 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。

（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。

死锁预防：

我们可以通过破坏死锁产生的4个必要条件来 预防死锁，由于资源互斥是资源使用的固有特性是无法改变的。

1破坏“不可剥夺”条件：一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到 系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动，执行。

2 破坏”请求与保持条件“：第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源。第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源。

3 破坏“循环等待”条件：采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程。

## 5.2.6硬链接与软链接的区别？

解析：硬链接与软链接是常见面试题。

 1>首先什么是链接？

     链接操作实际上是给系统中已有的某个文件指定另外一个可用于访问它的名称。对于这个新的文件名，我们可以为之指定不同的访问权限 ，以控制对信息的共享和安全性的问题。 如果链接指向目录，用户就可以利用该链接直接进入被链接的目录而不用打一大堆的路径名。而且，即使我们删除这个链接，也不会破坏原来的目录。

2>硬链接

    硬链接只能引用同一文件系统中的文件。它引用的是文件在文件系统中的物理索引(也称为inode)。当您移动或删除原始文件时，硬链接不会被破坏，因为它所引用的是文件的物理数据而不是文件在文件结构中的位置。硬链接的文件不需要用户有访问原始文件的权限，也不会显示原始文件的位置，这样有助于文件的安全。如果您删除的文件有相应的硬链接，那么这个文件依然会保留，直到所有对它的引用都被删除。

3>软链接（符号链接）

    软连接，其实就是新建立一个文件，这个文件就是专门用来指向别的文件的（那就和windows 下的快捷方式的那个文件有很接近的意味）。软连接产生的是一个新的文件，但这个文件的作用就是专门指向某个文件的，删了这个软连接文件，那就等于不需要这个连接，和原来的存在的实体原文件没有任何关系，但删除原来的文件，则相应的软连接不可用（cat那个软链接文件，则提示“没有该文件或目录“）。

4> 硬链接与软链接的区别；

硬连接是不会建立inode的，他只是在文件原来的inode link count域再增加1而已，也因此硬链接是不可以跨越文件系统的。相反都是软连接会重新建立一个inode，当然inode的结构跟其他的不一样，他只是一个指明源文件的字符串信息。一旦删除源文件，那么软连接将变得毫无意义。而硬链接删除的时候，系统调用会检查inode link count的数值，如果他大于等于1，那么inode不会被回收。因此文件的内容不会被删除。

硬链接实际上是为文件建一个别名，链接文件和原文件实际上是**同一个文件**。可以通过ls -i来查看一下，这两个文件的inode号是同一个，说明它们是同一个文件；而软链接建立的是一个指向，即链接文件内的内容是指向原文件的指针，它们是两个文件。

软链接可以跨文件系统，硬链接不可以；软链接可以对一个不存在的文件名(filename)进行链接（当然此时如果你vi这个软链接文件，linux会自动新建一个文件名为filename的文件）,硬链接不可以（其文件必须存在，inode必须存在）；软链接可以对目录进行连接，硬链接不可以。两种链接都可以通过命令 ln 来创建。ln默认创建的是硬链接。使用-s 开关可以创建软链接。

**总结：**

（1）软连接可以跨文件系统，硬连接不可以

（2）硬连接不管有多少个，都指向的是同一个I节点，会把结点连接数增加，只有符号链接才产生新的inode节点，只要结点的连接数不是0，文件就一直存在不管你删除的是源文件还是连接的文件。只要有一个存在文件就存在。 当你修改源文件或者连接文件任何一个的时候，其他的文件都会做同步的修改。软链接不直接使用i节点号作为文件指针, **而是使用文件路径名作为指针**。所以删除连接文件对源文件无影响，但是删除源文件，连接文件就会找不到要指向的文件。软链接有自己的inode, 并在磁盘上有一小片空间存放路径名。

（3）软连接可以对一个不存在的文件名或者目录进行连接。

## 5.2.7虚拟内存，虚拟地址与物理地址的转换

解析：

虚拟内存：将硬盘里的一部分当作内存使用

首先了解下这个



操作系统在分配内存时，有时候会产生一些空闲但是无法被正常使用的内存区域，这些就是内存碎片，或者称为内存零头，这些内存零头一共分为两类：内零头和外零头。

1）内零头是指进程在向操作系统请求内存分配时，系统满足了进程所需要的内存需求后，还额外还多分了一些内存给该进程，**也就是说额外多出来的这部分内存归该进程所有，其他进程是无法访问的。**

2）外零头是指内存中存在着一些空闲的内存区域，**这些内存区域虽然不归任何进程所有，但是因为内存区域太小，无法满足其他进程所申请的内存大小而形成的内存零头。**

页式存储管理是以页为单位（页面的大小由系统确定，且大小是固定的）向进程分配内存的，例如：假设内存总共有100K,分为10页，每页大小为10K。现在进程A提出申请56K内存，**因为页式存储管理是以页为单位进程内存分配的**，所以系统会向进程A提供6个页面，也就是60K的内存空间，那么在最后一页中进程只使用了6K，从而多出了4K的内存碎片，但是这4K的内存碎片系统已经分配给进程A了，其他进程是无法再访问这些内存区域的，这种内存碎片就是内零头。

3）段式存储管理是段（段的大小是程序逻辑确定，且大小不是固定的）为单位向进程进行内存分配的，进程申请多少内存，系统就给进程分配多少内存，这样就不会产生内零头，但是段式分配会产生外零头。

例如：假设内存总的大小为100K，现在进程A向系统申请60K的内存，系统在满足了进程A的内存申请要求后，还剩下40K的空闲内存区域；这时如果进程B向系统申请50K的内存区域，而系统只剩下了40K的内存区域，虽然这40K的内存区域不归任何进程所有，但是因为大小无法满足进程B的要求，所以也无法分配给进程B，这样就产生了外零头。请求段式存储管理是在段式存储管理的基础上增加了请求调段功能和段置换功能。

所以段式和请求段式存储管理会产生外零头，选BD。

### 分页内存管理 （提升的题目，也可不看）

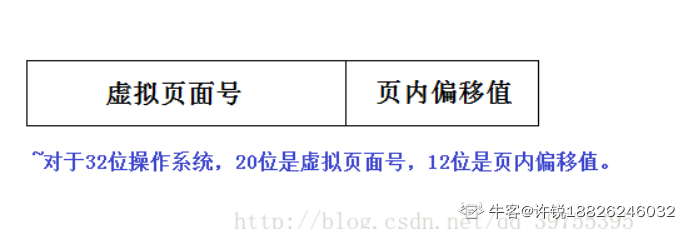
1交换内存管理中的机制——基址和极限将程序发出的虚拟地址加上基址得到物理地址”。

解决方法——分页内存管理

1.1分页内存管理

将虚拟内存空间和物理内存空间皆划分成大小相同的页面，例如4KB、8KB和16KB等。并将页作为内存空间的最小分配单位，一个程序的一个页面(虚拟页面)可以存放在任何一个物理页面中。

一个程序发出的虚拟地址由虚拟页面号和页内偏移值两部分组成，组成见下：

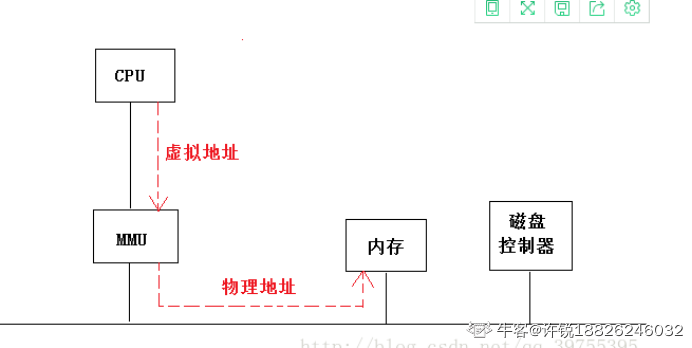


1.2分页内存管理是如何解决交换内存管理中的两个问题的？

1.空间浪费：通过将内存空间划分成大小一样的页面，并且将其作为内存分配的基本单位，这样就避免了大量外部碎片的积累，让内存空间得到有效利用。

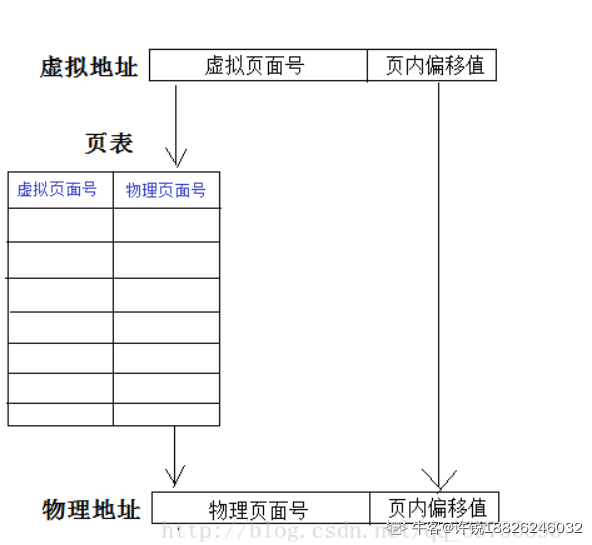
2.程序受限：分页内存管理下，允许一个进程的部分虚拟页面存放在物理压面(物理内存)中，另一部分存放在磁盘上，等到需要使用时再将其从磁盘中加载到物理内存中。也就是说，当程序需要额外的空间时，只需要对其分配新的页即可，这样做使得程序的增长效率较高。

1.3虚拟地址->物理地址的转化(地址翻译)



翻译工作则是交给MMU(内存管理单元)，它只对虚拟地址的页面号进行翻译，而不处理页内偏移值。

MMU为每一个程序都配备了一个页表，里面存放的是虚拟页面到物理页面的映射，如果MMU接收到了程序发出的虚拟地址，在查找相对应的物理页面号时，没有找到，那么将会通过缺页中断来将需要的虚拟页面从磁盘（因为虚拟内存 存放于磁盘中吧）中加载到物理内存的页面中。



并且随着虚拟页面的进出内存，页表的内容也是不断地变化的。

1.4页表中的具体信息

MMU为每一个程序配备的页表除了有虚拟地址到物理地址的映射(虚拟页面号到物理页面号)，还有其他的与页面的管理活动有关的信息，因为mmu要依赖页表来进行与页面相关的管理活动。

## 5.2.8计算机中，32bit与64bit有什么区别？

解析：

64bit计算主要有两大优点：可以进行更大范围的整数运算；可以支持更大的内存。 64位CPU一次可提取64位数据，比32位提高了一倍，理论上性能会提升1倍。但这是建立在64bit操作系统，64bit软件的基础上的。

但是我们不能因为数字上的变化，而简单的认为64bit处理器的性能是32bit处理器性能的两倍。实际上在32bit应用程序下，32bit处理器的性能甚至会更强，即使是64bit处理器，目前情况下也是在32bit应用下性能更强。所以要认清64bit处理器的优势，但不可迷信64bit。

64位操作系统下的虚拟内存空间大小：地址空间大小不是2^32，也不是2^64，而一般是2^48。因为并不需要2^64那么大的寻址空间，过大的空间只会造成资源的浪费。所以64位Linux一般使用48位表示虚拟空间地址，40位标识物理地址。0x0000000000000000~0x00007fffffffffff表示用户空间，0xFFFF800000000000~ 0xFFFFFFFFFFFFFFFF表示内核空间，共提供256TB(2^48)的寻址空间

## 5.2.9中断和异常的区别

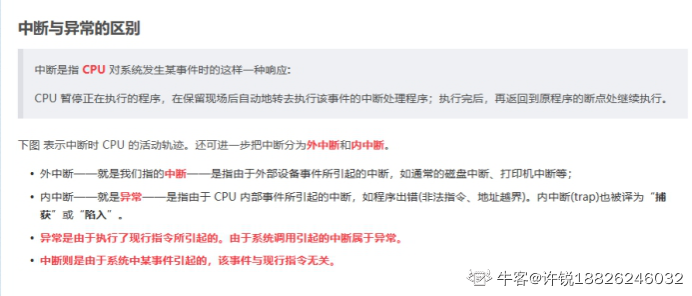
**问：中断分为哪两种，两个的区别是什么，应该怎么设计？（即中断优先级的设置）**

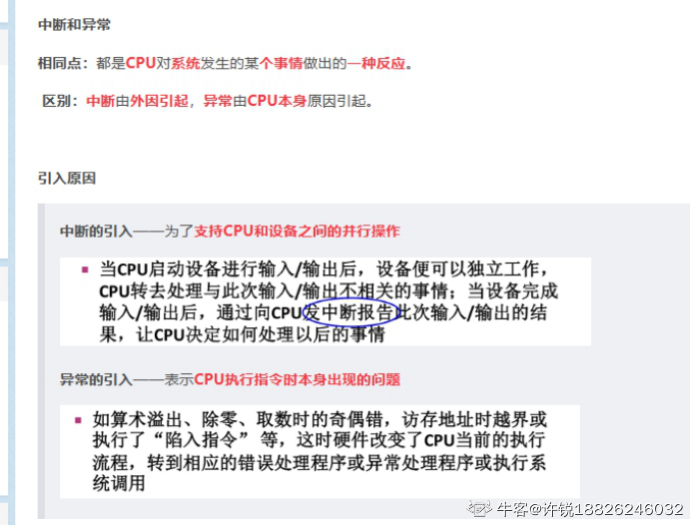
答：中断是异步事件，异常是同步事件。

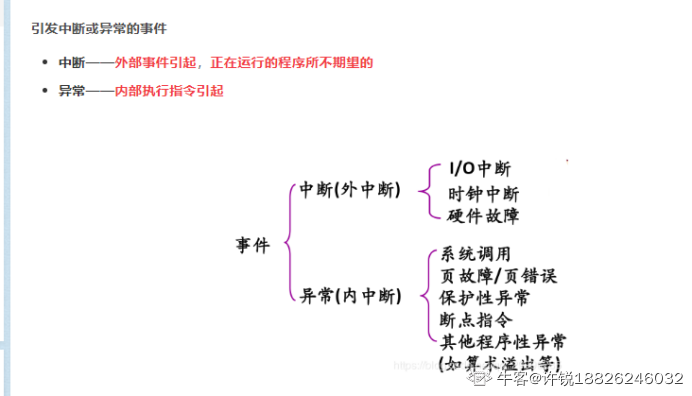
1）**同步中断**是由cpu内部的电信号产生的中断，其特点为当前执行的指令结束后才转而产生中断，由于有cpu主动产生，其执行点必然是可控的。

2）**异步中断**是由cpu的外设产生的电信号引起的中断，其发生的时间点不可预期。

3）**同步中断**也称为异常，要么是代码错误引起的，此时cpu通过发送相关的信号来处理异常（通常可能是杀死进程的信号）；要么是cpu必须处理的一些异常条件，此事cpu执行异常处理函数来恢复异常。







## 5.2.10中断怎么发生，中断处理大概流程

解析：

请求中断→响应中断→关闭中断→保留断点→中断源识别→保护现场→中断服务子程序→恢复现场→中断返回。

**请求中断：**当某一中断源需要CPU为其进行中断服务时，就输出中断请求信号，使中断控制系统的中断请求触发器置位，向CPU请求中断。系统要求中断请求信号一直保持到CPU对其进行中断响应为止；

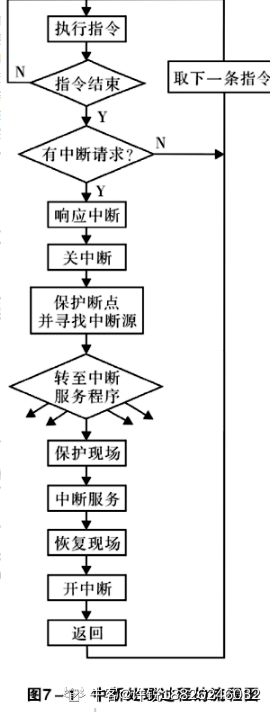
**中断响应：**CPU对系统内部中断源提出的中断请求必须响应，而且自动取得中断服务子程序的入口地址，执行中断服务子程序。对于外部中断，CPU在执行当前指令的最后一个时钟周期去查询INTR引脚，若查询到中断请求信号有效，同时在系统开中断（即IF=1）的情况下，CPU向发出中断请求的外设回送一个低电平有效的中断应答信号，作为对中断请求INTR的应答，系统自动进入中断响应周期；

**保护现场：**主程序和中断服务子程序都要使用CPU内部寄存器等资源，为使中断处理程序不破坏主程序中寄存器的内容，应先将断点处各寄存器的内容（主要是当前IP（将要执行的下一条地址）和CS值（代码段地址））压入堆栈保护起来，再进入的中断处理。现场保护是由用户使用PUSH指令来实现的；

**中断服务：**中断服务是执行中断的主体部分，不同的中断请求，有各自不同的中断服务内容，需要根据中断源所要完成的功能，事先编写相应的中断服务子程序存入内存，等待中断请求响应后调用执行；

**恢复现场：**当中断处理完毕后，用户通过POP指令将保存在堆栈中的各个寄存器的内容弹出，即恢复主程序断点处寄存器的原值。

**中断返回：**在中断服务子程序的最后要安排一条中断返回指令IRET（interrupt return），执行该指令，系统自动将堆栈内保存的IP（将要执行的下一条地址）和CS值（代码段地址）弹出，从而恢复主程序断点处的地址值，同时还自动恢复标志寄存器FR或EFR的内容，使CPU转到被中断的程序中继续执行。



## 5.2.11  Linux 操作系统挂起、休眠、关机相关命令

解析：

挂起是一种省电模式，系统将机器的硬盘、显示器等外部设备停止工作，而CPU、内存仍然工作，等待用户随时唤醒，再次唤醒需要按键盘上的键数次。

体眠是一种更加省电的模式，它将内存中的数据保存于硬盘中，使CPU也停止工作，当再次使用时需按开关机键，机器将会恢复到您的执行休眠时的状态，而不用再次执行启动操作系统复杂的过程。

待机是将当前处于运行状态的数据保存在内存中，机器只对内存供电，而硬盘、屏幕和CPU等部件则停止供电。由于数据存储在速度快的内存中，因此进入等待状态和唤醒的速度比较快。不过这些数据是保存在内存中，如果断电则会使数据丢

**关机重启：**

关机命令有halt，  init 0，  poweroff   ，shutdown -h时间，其中shutdown是最安全的

重启命令有reboot，init 6,，shutdow -r时间

在linux命令中reboot是重新启动，**shutdown -r now是立即停止然后重新启动，都说他们两个是一样的，其实是有一定的区别的。**

**shutdown**命令可**以安全地关闭**或重启Linux系统，它在系统关闭之前给系统上的所有登录用户提示一条警告信息。该命令还允许用户指定一个时间参数，可以是一个精确的时间，也可以是从现在开始的一个时间段。

精确时间的格式是hh:mm，表示小时和分钟，时间段由+ 和分钟数表示。系统执行该命令后会自动进行数据同步的工作。

该命令的一般格式: shutdown [选项] [时间] [警告信息]

命令中各选项的含义为:

- k 并不真正关机而只是发出警告信息给所有用户

- r 关机后立即重新启动

- h 关机后不重新启动

- f 快速关机重启动时跳过fsck

- n 快速关机不经过init 程序

- c 取消一个已经运行的shutdown

需要特别说明的是该命令只能由超级用户使用。

例1，系统在十分钟后关机并且马上重新启动: # shutdown –r +10  再重新启动linux系统的同时把内存中的信息写入硬盘

例2，系统马上关机并且不重新启动:# shutdown –h now

halt是最简单的关机命令，其实际上是调用shutdown -h命令。halt执行时，杀死应用进程，文件系统写操作完成后就会停止内核。

halt命令的部分参数如下：

[-f] 没有调用shutdown而强制关机或重启

[-i] 关机或重新启动之前，关掉所有的网络接口

[-p] 关机时调用poweroff，此选项为缺省选项

**reboot**的工作过程与halt类似，其作用是重新启动，而halt是关机。其参数也与halt类似。reboot命令重启动系统时是删除所有的进程，而不是平稳地终止它们。因此**，使用reboot命令可以快速地关闭系统，但如果还有其它用户在该系统上工作时，就会引起数据的丢失。所以使用reboot命令的场合主要是在单用户模式。**

**init**是所有进程的祖先，其进程号始终为1。init用于切换系统的运行级别，切换的工作是立即完成的。**init 0命令用于立即将系统运行级别切换为0，即关机；init 6命令用于将系统运行级别切换为6，即重新启动**

## 5.2.12数据库为什么要建立索引，以及索引的缺点

解析：嵌入式问的不多，当作了解。

回忆一下小时候查字典的步骤，索引和字典目录的概念是一致的。字典目录可以让我们不用翻整本字典就找到我们需要的内容页数，然后翻到那一页就可以。索引也是一样，索引是对记录按照多个字段进行排序的一种展现。对表中的某个字段建立索引会创建另一种数据结构，其中保存着字段的值，每个值还包括指向与它相关记录的指针。这样，就不必要查询整个数据库，自然提升了查询效率。同时，索引的数据结构是经过排序的，因而可以对其执行二分查找，那就更快了

先说优点：

    1）大大加快数据的检索速度，这也是创建索引的最主要的原因

    2）加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

    3）在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

 再说缺点：

  1）创建索引需要耗费一定的时间，但是问题不大，一般索引只要build一次

  2）索引需要占用物理空间，特别是聚集索引，需要较大的空间

  3）当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度，这个是比较大的问题。