进程和线程篇

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。

```
#include<stdio.h>
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;
}
```

即使在while循环中，线程仍然可以运行。

2.多进程和多线程的优点？

解析：线程与进程最大的区别：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.

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（为什么？）每个进程有自己独立的地址空间和资源，正常情况下，一个进程死了不会影响另一个进程，不过如果两个进程访问同一个公共资源，比如访问同一个文件等，或者一个进程产生了严重的内存泄漏，也是会影响到其他进程的。
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（为什么？）当一个线程死了（非正常退出、死循环等）就会导致线程该占有的资源永远无法释放，从而影响其他线程的正常工作。

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，创建多进程的开销远高于多线程。

3）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。

4）多进程逻辑控制比多线程复杂，需要与主进程做好交互（也需要进程同步以及加锁）。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。

5）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。

3.什么时候用进程，什么时候用线程?

答：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，因为进程的花销大；

2）需要大量数据传送用线程，因为多线程的切换速度快，并且不需要跨域；

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）安全稳定选进程；快速频繁选线程

4. 多进程、多线程同步（通讯）的方法？

答.进程间通讯：管道，信号，共享内存，消息队列，信号量，socket

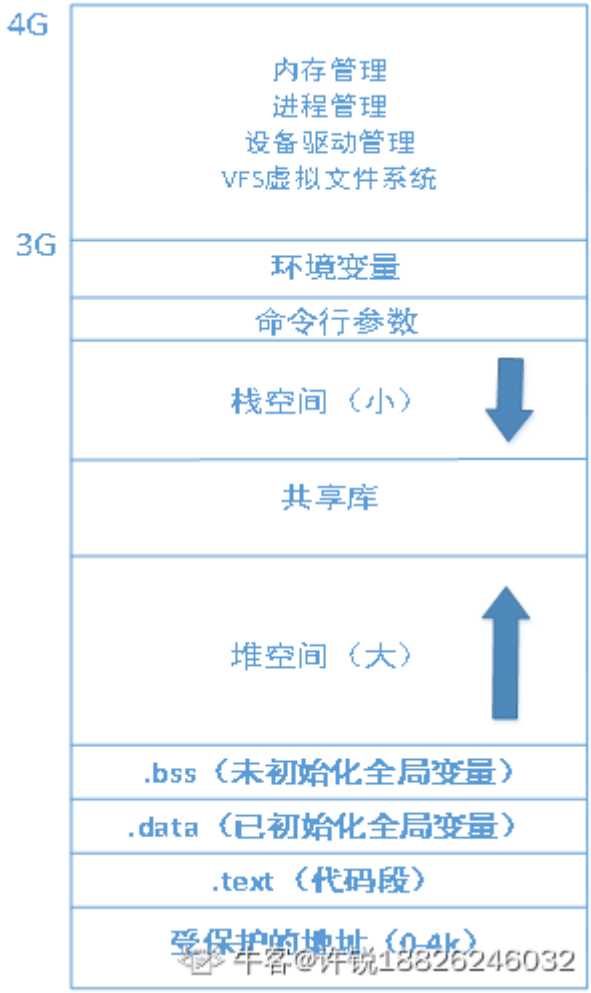
> 等待添加其每个的补充

线程通讯：信号量，读写锁，条件变量，互斥锁，自旋锁

> 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，这是其根本区别。但是信号量既可以作用于线程也可以作用于进程

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1-1，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。



注：.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

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。

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。

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。

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。

7. 进程线程的状态转换图，什么时候阻塞，什么时候就绪？

答. 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。

！图很重要
https://uploadfiles.nowcoder.com/images/20200327/545613072_1585309266567_B6C31D01D41C9E1714958F9C56D01D8F

7.1 进程的一生

答：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后接受结束子进程，该进程进入死亡态。

8 父进程和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。

9 什么是进程上下文、中断上下文？

解析：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。

答：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。

10 并发，同步，异步，互斥，阻塞，非阻塞的理解

10.1 并发，同步，异步，互斥，阻塞，非阻塞的概念

答：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）互斥、同步
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。

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：
a) 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。

b）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
答：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
答：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，而是马上返回去做其他事情。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。

11 什么是线程同步和互斥

答：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。

12 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。

13 孤儿进程、僵尸进程、守护进程的概念

答： 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。

14 如何创建守护进程

答：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：

```
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;
}
```

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。

15 正确处理孤儿进程、僵尸进程的方法

答：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

```
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");
    }
}
```

c/c++高频面试题

1 new和malloc的区别

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，可以自动计算所申请内存的大小。malloc返回的是无类型指针，我们需要强行将其转换为实际类型的指针即强转，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）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 指针与引用的相同和区别；如何相互转换？

答：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。
void fun(int &va){}
此时调用: fun(pA);

3 struct和class的区别

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

4 static的用法（定义和用途）

答：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，并且下次调用的时候不会被初始化。

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。

博客地址：https://blog.csdn.net/majianfei1023/article/details/45290467

5 const的用法（定义和用途）

答： 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。

6 const常量和#define的区别（编译阶段、安全性、内存占用等）

答：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）。

7 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）

答：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,但是原来的常量还是不变,只能是指针或者引用。

去掉const属性：const_cast<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。

```
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;
}
```

(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。

```
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;
}
```

博客地址：https://www.cnblogs.com/xiangtingshen/p/10851851.html

8 C++如何实现多态（讲明多态实现的三个条件、实现的原理）

答：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（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。

博客：https://www.cnblogs.com/alinh/p/9636352.html 和 https://blog.csdn.net/QLeelq/article/details/111058664

9 继承和虚继承

答：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。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。

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！

10 多态的类，内存布局是怎么样的

答：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

```
#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

```
#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）

```
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
};
```

11 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。

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。

12 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。

13 什么情况下会调用拷贝构造函数（三种情况）

答：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：

```
void getObj(cexample C) {
    std::cout<<"C = " << C;
}


int main() {
    cexample test(1);

    getObj(test);
    return 0;
}
```

14 析构函数一般写成虚函数的原因

答：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。

15 构造函数为什么一般不定义为虚函数

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。

16 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。

17 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。

18 C++所有的构造函数

答：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。

```
Student(Student &b)
{

    this.x=b.x;
    this.y=b.y;
}
```

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

```
Student(int r)
{
    int num=1004;
    int age= r;
}
```

19 重写、重载、覆盖的区别

答：（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。

20 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）？ ????待弄明白

答：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。

```
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：

```
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）。

```
#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");
}
```

21 15如何避免编译器进行的隐式类型转换？

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

网络编程

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。

答：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

2 TCP、UDP的优缺点

答：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 TCP UDP适用场景

答：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。

4 典型网络模型，简单说说有哪些

答：https://uploadfiles.nowcoder.com/images/20200529/545613072_1590757300047_D1333ACC9771B37DB87F187852C7B9FE

5 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。

前端重点关注：
http的常见面试题：https://blog.csdn.net/yicixing7/article/details/79320821

6 URI（统一资源标识符）和URL（统一资源定位符）之间的区别

解析：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。

7 什么是三次握手

答：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：
https://uploadfiles.nowcoder.com/images/20200530/545613072_1590812509163_48798565974D795668A42418F050DBC7

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。

8 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。

9 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。

10 什么是四次挥手？

答： https://uploadfiles.nowcoder.com/images/20200530/545613072_1590812715420_B8ABB0C7B81DF5E331CAD0129CFE6C6C
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乘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。

11 为什么客户端最后还要等待2MSL？

答： 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。

博客地址：https://zhuanlan.zhihu.com/p/86426969

STL库

1 vector list异同

答：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::iterator和list::iterator都重载了“++ ”操作。

2 vector内存是怎么增长的vector的底层实现

答：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。

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1.5倍/2倍，取决于你用的编译器种类。

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。
```

3 为什么stl里面有sort函数list里面还要再定义一个sort？

答：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 STL底层数据结构实现

答：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。

5 vector和deque的比较

答：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。

6 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

7 几种模板插入的时间复杂度

答：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)

8 利用迭代器删除元素会发生什么？

答：（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 valset = { 1,2,3,4,5,6 };

```
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 val = { 1,2,3,4,5,6 };

```
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;
}
```

操作系统

1 堆和栈的区别

答：https://uploadfiles.nowcoder.com/images/20200604/545613072_1591240396551_CFD7B443AE1229B43D9ED1E7B10FA85E

2 堆，栈，内存泄漏，内存溢出？

答：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。