目录

[《最全面C++面试题》---<套餐6/7学员专用> 5](#_Toc21916)

[1、知识储备 5](#_Toc5947)

[1、C++ 5](#_Toc21980)

[2、数据结构与算法（以刷题为主，如果实在没时间就略过） 170](#_Toc9963)

[3、操作系统 182](#_Toc1687)

[4、计算机网络 233](#_Toc16890)

[2、 QT篇 274](#_Toc18378)

[3 数据库 280](#_Toc30946)

[1 数据库事务有哪些？ 280](#_Toc7858)

[2、什么是 Redis？简述它的优缺点？ 280](#_Toc16365)

[3、Redis 支持哪几种数据类型？ 280](#_Toc10856)

[4、Redis ⼀般都有哪些使⽤场景？ 282](#_Toc11744)

[5 、Redis 有哪些常⻅的功能？ 284](#_Toc4714)

[6、谈谈你对索引的理解？ 284](#_Toc12519)

[7、索引的底层使⽤的是什么数据结构？ 285](#_Toc26191)

[8、谈谈你对 B+ 树的理解？ 285](#_Toc29679)

[9、为什么 InnoDB 存储引擎选⽤ B+ 树⽽不是 B 树呢？ 285](#_Toc24630)

[10、谈谈你对聚簇索引的理解？ 285](#_Toc16344)

[11、谈谈你对哈希索引的理解？ 285285](#_Toc18365)

[12、谈谈你对覆盖索引的认识？ 286](#_Toc16159)

[13、索引的分类？ 286](#_Toc24949)

[13、谈谈你对最左前缀原则的理解？ 286](#_Toc32753)

[14、怎么知道创建的索引有没有被使⽤到？或者说怎么才可以知道这条语 287](#_Toc30512)

[15、什么情况下索引会失效？即查询不⾛索引？ 287](#_Toc12235)

[16、查询性能的优化⽅法？ 288](#_Toc10149)

[17 、InnoDB 和 MyISAM 的⽐较？ 288](#_Toc9481)

[18、谈谈你对⽔平切分和垂直切分的理解？ 288](#_Toc24628)

[19、主从复制中涉及到哪三个线程？ 288](#_Toc11805)

[20、主从同步的延迟原因及解决办法？ 289](#_Toc14805)

[21、谈谈你对数据库读写分离的理解？ 289](#_Toc6976)

[22、请你描述下事务的特性？ 290](#_Toc31353)

[23、谈谈你对事务隔离级别的理解？ 290](#_Toc24276)

[24、解释下什么叫脏读、不可重复读和幻读？ 290](#_Toc18013)

[25 、MySQL 默认的隔离级别是什么？ 290](#_Toc16866)

[26、谈谈你对MVCC 的了解？ 290](#_Toc10536)

[27、说⼀下 MySQL 的⾏锁和表锁？ 291](#_Toc21829)

[28 、InnoDB 存储引擎的锁的算法有哪些？ 291](#_Toc7372)

[29 、MySQL 问题排查都有哪些⼿段？ 291](#_Toc19854)

[30 、MySQL 数据库 CPU 飙升到 500% 的话他怎么处理？ 291](#_Toc9335)

[4 分布式 291](#_Toc9888)

[5 音视频 291](#_Toc1058)

[6 嵌入式 291](#_Toc5527)

[1.  什么是嵌入式系统，相对于桌面系统有哪些特点？ 291](#_Toc8572)

[2. （场景题）嵌入式程序跑飞了，一般怎么定位和排查？ 292](#_Toc17013)

[3. 嵌入式系统中常见的 CPU 架构有哪些？ 292](#_Toc16704)

[4 嵌入式系统中常见的外设接口有哪些？ 293](#_Toc20650)

[5 用GPIO模拟I2C？ 293](#_Toc11326)

[6  I2C的时序图？ 295](#_Toc13431)

[7. 嵌入式系统中输入和输出设备的控制方法有哪些？ 296](#_Toc24988)

[8 同步串口和异步串口的区别？ 296](#_Toc1201)

[9.  IO口工作方式有哪些，分别的作用和特点是什么？ 296](#_Toc26642)

[10. （场景题）使用示波器来定位串口数据是否丢失？ 297](#_Toc13183)

[11.  RTOS是实时多任务系统吗？他的抢占式是如何实现的？？ 298](#_Toc7737)

[12 BootLoader是什么？描述一下他的启动流程？ 298](#_Toc11768)

[13 SPI四种模式比较 298](#_Toc10789)

[14. SPI协议的原理解析？ 299](#_Toc8660)

[15. 蓝牙协议、433、zigbee协议的比较？ 299](#_Toc27351)

[16. 怎样实现类比信号的采集和处理？ 300](#_Toc24974)

[17. 嵌入式系统中常见的数字信号处理器有哪些？ 300](#_Toc8789)

[19. 如何实现低功耗模式？ 300](#_Toc25257)

[20. 怎样实现数字和模拟信号的转换？ 300](#_Toc1383)

[21. 中断的作用是什么？如何实现中断处理？ 300](#_Toc28399)

[22. 嵌入式系统中常见的外设控制模式有哪些？ 300](#_Toc1605)

[23. 串口的工作原理和实现方法是什么？ 300](#_Toc10471)

[24. 如何实现定时和计数器功能？ 300](#_Toc6593)

[25. 如何实现 PWM 功能？ 26. 如何实现 A/D 转换？ 300](#_Toc17631)

[27. 嵌入式系统中如何实现 D/A 转换？ 300](#_Toc29360)

[28.  SPI 总线的工作原理和实现方法是什么？ 300](#_Toc9005)

[29. I2C 总线的工作原理和实现方法是什么？ 300](#_Toc19865)

[30. UART 和 USART 的差异和实现方法是什么？ 300](#_Toc7577)

[31.  PWM 和 DAC 的差异和应用场景是什么？ 300](#_Toc30461)

[32. USB 接口的控制方法是什么？ 301](#_Toc10517)

[33 ARM嵌入式系统中时序控制的作用是什么？如何实现？ 301](#_Toc18607)

[34. 嵌入式系统中如何实现定点运算和浮点运算？ 301](#_Toc13865)

[7 项目面试 301](#_Toc32532)

[0 公共类高频面试题 301](#_Toc19898)

[1、远控项目（基于工控环境的跨局域网的远程桌面控制系统） 306](#_Toc3399)

[2、网络天气预警系统项目 316](#_Toc5399)

[3、音视频项目 182](#_Toc3953)

[4、易栈项目 189](#_Toc31607)

[5、 易播加密播放器 193](#_Toc13529)

[6、 易播服务器（Windows版） 198](#_Toc9285)

[7、 易播服务器（Linux版） 202](#_Toc1910)

[8 HR面试题（软技能） 216](#_Toc31985)

[1、 请你自我介绍一下你自己 216](#_Toc25865)

[2、你觉得你个性上最大的优点是什么？ 216](#_Toc8730)

[3、说说你最大的缺点？ 217](#_Toc24070)

[4、你对加班的看法？ 218](#_Toc5975)

[5、你对薪资的要求？ 218](#_Toc4313)

[6、在五年的时间内，你的职业规划？ 218](#_Toc3151)

[7、 你朋友对你的评价？ 219](#_Toc6632)

[8、 你还有什么问题要问吗？ 219](#_Toc23070)

[9、如果通过这次面试我们单位录用了你，但工作一段时间却发现你根本不适合这个职位，你怎么办？ 219](#_Toc29547)

[10、在完成某项工作时，你认为领导要求的方式不是最好的，自己还有更好的方法，你应该怎么做？ 220](#_Toc6887)

[11、如果你的工作出现失误，给本公司造成经济损失，你认为该怎么办？ 220](#_Toc1718)

[12、如果你在这次考试中没有被录用，你怎么打算？ 220](#_Toc10033)

[13、如果你做的一项工作受到上级领导的表扬，但你主管领导却说是他做的，你该怎样？ 221](#_Toc14844)

[14、谈谈你对跳槽的看法？ 221](#_Toc1307)

[15、工作中你难以和同事、上司相处，你该怎么办？ 222](#_Toc14506)

[16、假设你在某单位工作，成绩比较突出，得到领导的肯定。但同时你发现同事们越来越孤立你，你怎么看这个问题？你准备怎么办？ 222](#_Toc16614)

[17、你最近是否参加了培训课程？谈谈培训课程的内容。是公司资助还是自费参加？ 222](#_Toc23141)

[18、你对于我们公司了解多少？ 222](#_Toc26244)

[19、请说出你选择这份工作的动机？ 223](#_Toc10953)

[20、你最擅长的技术方向是什么？ 223](#_Toc3266)

[21、你能为我们公司带来什么呢？ 223](#_Toc24685)

[22、最能概括你自己的三个词是什么？ 223](#_Toc9947)

[23、你的业余爱好是什么？ 223](#_Toc17589)

[24、作为被面试者给我打一下分 224](#_Toc14184)

[25、你怎么理解你应聘的职位？ 224](#_Toc23413)

[26、喜欢这份工作的哪一点？ 224](#_Toc6157)

[27、为什么要离职? 224](#_Toc20599)

[28、说说你对行业、技术发展趋势的看法？ 225](#_Toc25429)

[29、对工作的期望与目标何在？ 225](#_Toc8796)

[30、说说你的家庭。 225](#_Toc14994)

[31、就你申请的这个职位，你认为你还欠缺什么？ 226](#_Toc32123)

[32、你欣赏哪种性格的人？ 226](#_Toc12237)

[33、你通常如何处理別人的批评？ 226](#_Toc3745)

[34、你怎样对待自己的失敗？ 226](#_Toc7163)

[35、什么会让你有成就感？ 226](#_Toc5340)

[36、眼下你生活中最重要的是什么？ 226](#_Toc11285)

[37、你为什么愿意到我们公司来工作？ 226](#_Toc29383)

[38、你和别人发生过争执吗？你是怎样解决的？ 227](#_Toc971)

[39、问题：你做过的哪件事最令自己感到骄傲? 227](#_Toc17123)

[40、你新到一个部门,一天一个客户来找你解决问题,你努力想让他满意，可是始终达不到群众得满意,他投诉你们部门工作效率低,你这个时候怎么作? 227](#_Toc2787)

[41、对这项工作，你有哪些可预见的困难？” 228](#_Toc20478)

[42、如果我录用你，你将怎样开展工作？” 228](#_Toc25224)

[43、“你希望与什么样的上级共事？” 229](#_Toc6602)

[44、在完成某项工作时，你认为领导要求的方式不是最好的，自己还有更好的方法，你应该怎么做？ 229](#_Toc10771)

[45、与上级意见不一是，你将怎么办？” 229](#_Toc3839)

[46、“你工作经验欠缺，如何能胜任这项工作？” 229](#_Toc14098)

[47、您在前一家公司的离职原因是什么？” 230](#_Toc12741)

[48、“你工作经验欠缺，如何能胜任这项工作？” 231](#_Toc17051)

[49、为了做好你工作份外之事，你该怎样获得他人的支持和帮助？ 231](#_Toc16440)

[50、如果你在这次面试中没有被录用，你怎么打算？ 231](#_Toc24115)

[51、假如你晚上要去送一个出国的同学去机场，可单位临时有事非你办不可，你怎么办？ 232](#_Toc2627)

[52、如果通过这次面试我们单位录用了你，但工作一段时间却发现你根本不适合这个职位，你怎么办？ 233](#_Toc21697)

[53、你做过的哪件事最令自己感到骄傲? 233](#_Toc10025)

[54、谈谈你过去做过的成功案例 233](#_Toc118)

[55、谈谈你过去的工作经验中，最令你挫折的事情 233](#_Toc4208)

[56、如何安排自己的时间？会不会排斥加班？ 234](#_Toc9356)

[57、为什么我们要在众多的面试者中选择你？ 234](#_Toc5390)

[58、对这个职务的期许？ 234](#_Toc4669)

[59、为什么选择这个职务？ 234](#_Toc28415)

[60、为什么选择我们这家公司？ 235](#_Toc23160)

[61、你认为你在学校属于好学生吗？ 235](#_Toc22206)

[62、请谈谈如何适应办公室工作的新环境？ 235](#_Toc6670)

[63、在工作中学习到了些什么？ 235](#_Toc2856)

[64、有想过创业吗？ 236](#_Toc7277)

[65、最能概括你自己的三个词是什么？ 236](#_Toc14803)

[66、你认为你在学校属于好学生吗？ 236](#_Toc12590)

[67、除了本公司外，还应聘了哪些公司？ 236](#_Toc21228)

[68、何时可以到职？ 237](#_Toc13042)

[69、你并非毕业于名牌院校？ 237](#_Toc27915)

[70、你怎样看待学历和能力？ 237](#_Toc29297)

# 《最全面C++面试题》---<套餐6/7学员专用>

[Summarize](https://www.baidu.com/link?url=LgpPyn2z1L_OfhOACJ44l0kB8Wop-JYAjQFLCl9QPpfP0Ff9vlYJ-m0Tdw7A0eWWqby5iNfGjYclDY9ck8OlsrYG2dR9KYl5YXQPoufPeBa&wd=&eqid=911418e00008fb200000000660e67c52" \t "https://www.baidu.com/_blank)d By YiFeiyun

QQ：1069619619

# 1、知识储备

吧 

说明：红色是最重要的，

绿色次重要

没有渲染颜色为次次重要

以语法为主，网络其次，操作系统主要关注进程和线程的相关原理

## 1、C++

### 1、指针和引用的区别 \*\*\*\*\*\*\*

指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名

指针可以有多级，引用只有一级

指针可以为空，引用不能为NULL且在定义时必须初始化

指针在初始化后可以改变指向，而引用在初始化之后不可再改变

sizeof指针得到的是本指针的大小，sizeof引用得到的是引用所指向变量的大小

当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是 同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。

引用本质是一个指针，同样会占4字节内存；指针是具体变量，需要占用存储空间（具体情况还 要具体分析）。

|  |
| --- |
| #include <iostream>  void add(int\* **p**, int& **r**, int **num**) {  \***p** += **num**;  **r** += **num**;  }  int main() {  int **x** = 10;  int\* **p** = &**x**;  int& **r** = **x**;  *std*::*cout* << \***p** << " " << **r** << *std*::*endl*;  add(**p**, **r**, 5);  *std*::*cout* << \***p** << " " << **r** << *std*::*endl*;  double **nn** = 0;  double& **inf** = **nn**;  double \***ptr** = &**nn**;  *std*::*cout* << sizeof(**nn**) << sizeof(**inf**) << sizeof(**ptr**) << *std*::*endl*;// 8 8 4  return 0;  }  /\*在这个示例中，x是一个整数，p是指向x的指针，r是对x的引用。add 函数接收一个整数指针和一个整数引用，以及一个要增加的值 num。函数使用指针和引用更改x的值，将其增加 num。  在主函数中，我们首先输出了指针和引用所指向的值，然后调用 add 函数，并再次输出指针和引用所指向的值。由于传入的是指针和引用的引用，所以函数可以在不返回任何值的情况下更改传入的参数的值，这使得在有些情况下使用指针和引用更为方便。  需要注意的是，指针和引用的主要区别在于指针可以指向空值，而引用不能，指针可以随意改变指向的对象，而引用不能，指向对象后就不再改变。另外，指针需要使用\*运算符来访问所指向的对象，而引用不需要。  \*/ |

### 2、请你描述一下堆和栈的区别

（1）管理方式不同。栈由操作系统自动分配释放，无需我们手动控制；堆的申请和释放工作由程序员控制，容易产生内存泄漏；

（2）空间大小不同。每个进程拥有的栈大小要远远小于堆大小。理论上，进程可申请的堆大小为虚拟内存大小，进程栈的大小 64bits 的 Windows 默认 1MB，64bits 的 Linux 默认 10MB；

（3）生长方向不同。堆的生长方向向上，内存地址由低到高；栈的生长方向向下，内存地址由高到低。

（4）分配方式不同。堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是由操作系统完成的，比如局部变量的分配。动态分配由alloca()函数分配，但是栈的动态分配和堆是不同的，它的动态分配是由操作系统进行释放，无需我们手工实现。

（5）分配效率不同。栈由操作系统自动分配，会在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是由C/C++提供的库函数或运算符来完成申请与管理，实现机制较为复杂，频繁的内存申请容易产生内存碎片。显然，堆的效率比栈要低得多。

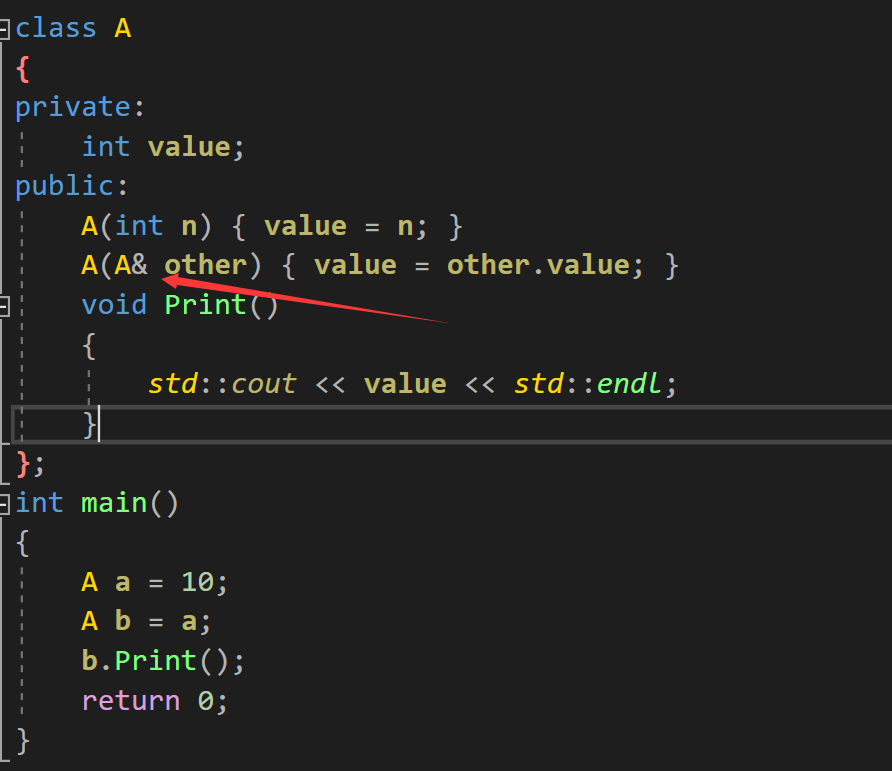
|  |  |  |
| --- | --- | --- |
|  | **堆** | **栈** |
| **管理方式** | 堆中资源由程序员控制（容易产生memory leak） | 栈资源由编译器自动管理，无需手工控制 |
| **内存管理机制** | 系统有一个记录空闲内存地址的链表，当系统收到程序申请时，遍历该链表，寻找第一个空间大于申请空间的堆结点，删 除空闲结点链表中的该结点，并将该结点空间分配给程序（大多数系统会在这块内存空间首地址记录本次分配的大小，这样delete才能正确释放本内存空间，另外系统会将多余的部分重新放入空闲链表中） | 只要栈的剩余空间大于所申请空间，系统为程序提供内存， 否则报异常提示栈溢出。（这一块理解一下链表和队列的区别，不连续空间和连续空间的区别，应该就比较好理解这两种机制的区别了） |
| **空间大小** | 堆是不连续的内存区域（因为系统是用链表来存储空闲内存地址，自然不是连续的），堆大小受限于计算机系统中有效的虚拟内存（32bit 系统理论上是4G），所以堆的空间比较灵活，比较大 | 栈是一块连续的内存区域，大小是操作系统预定好的， windows下栈大小是2M（也有是1M，在 编译时确定，VC中可设置） |
| **碎片问题** | 对于堆，频繁的new/delete会造成大量碎片，使程序效率降低 | 对于栈，它是有点类似于数据结构上的一个先进后出的栈， 进出一一对应，不会产生碎片。（看到这里我突然明白了为什么面试官在问我堆和栈的区别之前先问了我栈和队列的区别） |
| **生长方向** | 堆向上，向高地址方向增长。 | 栈向下，向低地址方向增长。 |
| **分配方式** | 堆都是动态分配（没有静态分配的堆） | 栈有静态分配和动态分配，静态分配由编译器完成（如局部变量分配），动态分配由alloca 函数分配，但栈的动态分配的资源由编译器进行释放，无需程序员实现。 |
| **分配效率** | 堆由C/C++函数库提供，机制很复杂。所以堆的效率比栈低很多。 | 栈是其系统提供的数据结构， 计算机在底层对栈提供支持， 分配专门 寄存器存放栈地址， 栈操作有专门指令。 |

### 3 .将“引用”作为函数参数有哪些特点？

1. 使用引用传参，不会创建拷贝，可以提升效率并节省了空间，比如我们要传一个很大的结构体，用引用传参就省去了拷贝这个结构体的开销。  
   2、在函数中对该变量进行修改，则参数返回后修改依然存在，与值传递不同；

3、C++的标准不允许复制构造函数传值参数，最好是传引用，在下面代码中，复制构造函数A(A other)传入的参数是A的一个实例。由于是传值参数，我们把形参复制到实参会调用复制构造函数。因此如果

允许复制构造函数传值，就会在复制构造函数内调用复制构造函数，就会形成永无休止的递归调用从而导致栈溢出。因此C++的标准不允许复制构造函数传值参数，在Visual Studio和GCC中，都将编译出错。要解决这个问题，我们可以把构造函数修改为A(const A& other),也就是把传值参数改成常量引用。



4、基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间

首先整理一下虚函数表的特征：

虚函数表是全局共享的元素，即全局仅有一个，在编译时就构造完成

虚函数表类似一个数组，类对象中存储vptr指针，指向虚函数表，即虚函数表不是函数，不是程序 代码，不可能存储在代码段

虚函数表存储虚函数的地址,即虚函数表的元素是指向类成员函数的指针,而类中虚函数的个数在编 译时期可以确定，即虚函数表的大小可以确定,即大小是在编译时期确定的，不必动态分配内存空 间存储虚函数表，所以不在堆中

根据以上特征，虚函数表类似于类中静态成员变量.静态成员变量也是全局共享，大小确定，因此最有可 能存在全局数据区，测试结果显示：

虚函数表vtable在Linux/Unix中存放在可执行文件的只读数据段中(rodata)，这与微软的编译器将虚函 数表存放在常量段存在一些差别

由于虚表指针vptr跟虚函数密不可分，对于有虚函数或者继承于拥有虚函数的基类，对该类进行实例化 时，在构造函数执行时会对虚表指针进行初始化，并且存在对象内存布局的最前面。一般分为五个区域：栈区、堆区、函数区（存放函数体等二进制代码）、全局静态区、常量区

C++中虚函数表位于只读数据段（.rodata），也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。

### 5、new / delete 与 malloc / free的异同

**相同点**

都可用于内存的动态申请和释放

**不同点**

前者是C++运算符，后者是C/C++语言标准库函数

new自动计算要分配的空间大小，malloc需要手工计算new是类型安全的，malloc不是。例如：

|  |
| --- |
| int\* **p** = new float[2]; //编译错误  int\* **p** = (int\*)*malloc*(2 \* sizeof(double));//编译无错误 |

new调用名为**operator new**的标准库函数分配足够空间并调用相关对象的构造函数，delete对指针所指对象运行适当的析构函数；然后通过调用名为**operator delete**的标准库函数释放该对象所用内存。后者均没有相关调用

后者需要库文件支持，前者不用

new是封装了malloc，直接free不会报错，但是这只是释放内存，而不会析构对象

### 6、new和delete是如何实现的？

new的实现过程是：首先调用名为**operator new**的标准库函数，分配足够大的原始为类型化的内 存，以保存指定类型的一个对象；接下来运行该类型的一个构造函数，用指定初始化构造对象；最 后返回指向新分配并构造后的的对象的指针

delete的实现过程：对指针指向的对象运行适当的析构函数；然后通过调用名为**operator delete**

的标准库函数释放该对象所用内存

### 8 、strlen和sizeof区别？

编译器在编译时就计算出了 sizeof 的结果。而 strlen 函数必须在运行时才能计算出来。

并且 sizeof 计算的是数据类型占内存的大小，而 strlen 计算的是字符串实际的长度。

数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针了。

sizeof 是一个操作符，strlen 是库函数。

sizeof 的参数可以是数据的类型，也可以是变量，而 strlen 只能以结尾为‘\0‘的字符串作参数。

|  |
| --- |
| int main()  {  const char\* **str** = "name"; 4  sizeof(**str**); // 取的是指针str的长度，是8  *strlen*(**str**); // 取的是这个字符串的长度，不包含结尾的 \0。大小是4  return 0;  } |

### 9、a和&a有什么区别？

假设数组

int a[10]; int (\*p)[10] = &a;

a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是0x00000001，加1操作后变为0x00000005。\*(a + 1) = a[1]。

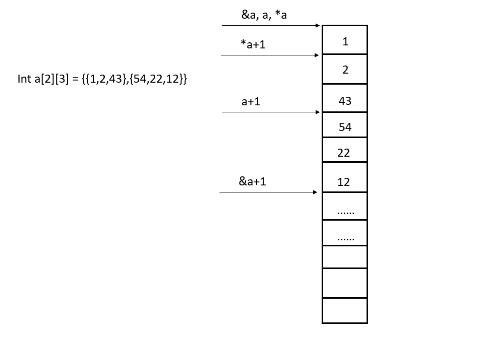
&a是数组的指针，其类型为int (\*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量）l，值为数组a尾元素后一个元素的地址。

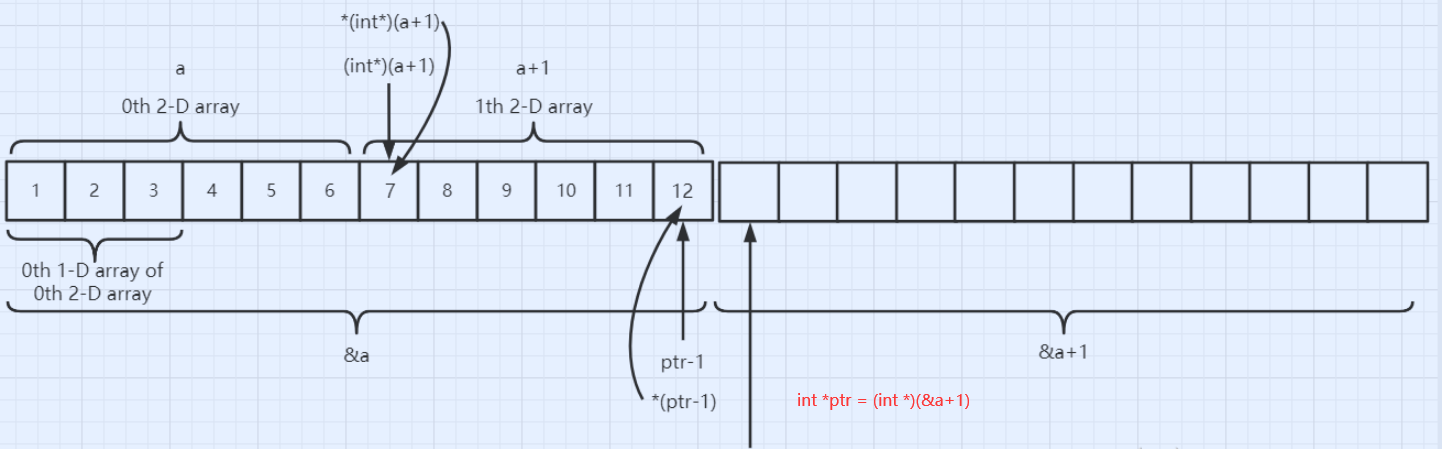
若(int \*)p ，此时输出 \*p时，其值为a[0]的值，因为被转为int \*类型，解引用时按照int类型大小来读取。

例子：求输出结果 ()

|  |
| --- |
| int main() {  int a[2][2][3] = { {{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}} };  int\* ptr = (int\*)(&a + 1);  *printf*(" %d %d", \*(int\*)(a + 1), \*(ptr - 1));  *printf*(" %d %d", \*(int\*)(\*a + 1), \*(int\*)(\*a + 2));  return 0;  } |

A 7 12 B 1 6 C 1 3 D 7 9





### 12、请你描述一下野指针和悬空指针的概念

野指针，**未被初始化的指针**

因此，为了防止出错，对于指针初始化时都是赋值为 nullptr ，这样在使用时编译器就会直接报错，产生非法内存访问。

悬空指针

悬空指针，指针最初指向的内存已经被释放了的一种指针。

|  |
| --- |
| int main(void) {  int\***P** = nu1lptr;  int\* **p2** = new int;  **P** = **p2**;  delete **p2**;  } |

此时 p和p2就是悬空指针，指向的内存已经被释放。继续使用这两个指针，行为不可预料。需要设置为 p=p2=nullptr 。此时再使用，编译器会直接保错。

避免野指针比较简单，但悬空指针比较麻烦。c++引入了智能指针，C++智能指针的本质就是避免悬空指针的产生。

产生原因及解决办法：

野指针：指针变量未及时初始化 => 定义指针变量及时初始化，要么置空。

悬空指针：指针free或delete之后没有及时置空 => 释放操作后立即置空。

### 13、C++中struct和class的区别

相同点

两者都拥有成员函数、公有和私有部分

任何可以使用class完成的工作，同样可以使用struct完成

不同点

两者中如果不对成员不指定公私有，struct默认是公有的，class则默认是私有的

class默认是private继承，而struct模式是public继承

**引申**：C++和C的struct区别

C语言中：struct是用户自定义数据类型（UDT）；C++中struct是抽象数据类型（ADT），支持成 员函数的定义，（C++中的struct能继承，能实现多态）

C中struct是没有权限的设置的，且struct中只能是一些变量的集合体，可以封装数据却不可以隐藏 数据，而且成员**不可以是函数**

C++中，struct增加了访问权限，且可以和类一样有成员函数，成员默认访问说明符为public（为 了与C兼容）

struct作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在C中必须在结构标记 前加上struct，才能做结构类型名（除：typedef struct class{};）;C++中结构体标记（结构体名） 可以直接作为结构体类型名使用，此外结构体struct在C++中被当作类的一种特例

### 14、deﬁne宏定义和const的区别

编译阶段

deﬁne是在编译的**预处理**阶段起作用，而const是在编译、运行的时候起作用

安全性

deﬁne只做替换，不做类型检查和计算，也不求解，容易产生错误，一般最好加上一个大括号包含 住全部的内容，要不然很容易出错

const常量有数据类型，编译器可以对其进行类型安全检查

内存占用

deﬁne只是将宏名称进行替换，在内存中会产生多分相同的备份。const在程序运行中只有一份备 份，且可以执行常量折叠，能将复杂的的表达式计算出结果放入常量表

宏替换发生在编译阶段之前，属于文本插入替换；const作用发生于编译过程中。 宏不检查类型；const会检查数据类型。

宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配 内存空间。

### 15、C++中const和static的作用

static

不考虑类的情况

隐藏。所有不加static的全局变量和函数具有全局可见性，可以在其他文件中使用，加了之后 只能在该文件所在的编译模块中使用

默认初始化为0，包括未初始化的全局静态变量与局部静态变量，都存在全局未初始化区

静态变量在函数内定义，始终存在，且只进行一次初始化，具有记忆性，其作用范围与局部 变量相同，函数退出后仍然存在，但不能使用

考虑类的情况

static成员变量：只与类关联，不与类的对象关联。定义时要分配空间，不能在类声明中初始 化，必须在类定义体外部初始化，初始化时不需要标示为static；可以被非static成员函数任 意访问。

static成员函数：不具有this指针，无法访问类对象的非static成员变量和非static成员函数；

**不能被声明为const、虚函数和volatile**；可以被非static成员函数任意访问

const

不考虑类的情况

const常量在定义时必须初始化，之后无法更改

const形参可以接收const和非const类型的实参，例如

|  |
| --- |
| // i 可以是 int 型或者 const int 型  void fun(const int& **i**)  {  //...  } |

考虑类的情况

const成员变量：不能在类定义外部初始化，只能通过构造函数初始化列表进行初始化，并且 必须有构造函数；不同类对其const数据成员的值可以不同，所以不能在类中声明时初始化const成员函数：const对象不可以调用非const成员函数；非const对象都可以调用；不可以 改变非mutable（用该关键字声明的变量可以在const成员函数中被修改）数据的值

### 16、类的对象存储空间？

非静态成员的数据类型大小之和。

编译器加入的额外成员变量（如指向虚函数表的指针）。为了边缘对齐优化加入的padding。

空类(无非静态数据成员)的对象的size为1, 当作为基类时, size为0.

### 17、ﬁnal和override关键字

override

当在父类中使用了虚函数时候，你可能需要在某个子类中对这个虚函数进行重写，以下方法都可以：

|  |
| --- |
| 1 class A  2 {  3 virtual void foo(); 4 }  5 class B : public A 6 {   1. void foo(); //OK 2. virtual void foo(); // OK 3. void foo() override; //OK 10 }   11  12  13 |

如果不使用override，当你手一抖，将**foo()**写成了**f00()**会怎么样呢？结果是编译器并不会报错，因为 它并不知道你的目的是重写虚函数，而是把它当成了新的函数。如果这个虚函数很重要的话，那就会对 整个程序不利。所以，override的作用就出来了，它指定了子类的这个虚函数是重写的父类的，如果你 名字不小心打错了的话，编译器是不会编译通过的：

|  |
| --- |
| 1 class A  2 {  3 virtual void foo(); 4 };  5 class B : public A 6 {   1. virtual void f00(); //OK，这个函数是B新增的，不是继承的 2. virtual void f0o() override; //Error, 加了override之后，这个函数一定是继承自   A的，A找不到就报错9 };  10 |

ﬁnal

当不希望某个类被继承，或不希望某个虚函数被重写，可以在类名和虚函数后添加ﬁnal关键字，添加

ﬁnal关键字后被继承或重写，编译器会报错。

《C++:override和ﬁnal》：<https://www.cnblogs.com/whlook/p/6501918.html>

### 18、拷贝初始化和直接初始化

当用于类类型对象时，初始化的拷贝形式和直接形式有所不同：直接初始化直接调用与实参匹配的 构造函数，拷贝初始化总是调用拷贝构造函数。拷贝初始化首先使用指定构造函数创建一个临时对 象，然后用拷贝构造函数将那个临时对象拷贝到正在创建的对象。举例如下

|  |
| --- |
| 1. string str1("I am a string");//语句1 直接初始化 2. string str2(str1);//语句2 直接初始化，str1是已经存在的对象，直接调用构造函数对str2进行初始化 3. string str3 = "I am a string";//语句3 拷贝初始化，先为字符串”I am a string“创建临时对象，再把临时对象作为参数，使用拷贝构造函数构造str3 4. string str4 = str1;//语句4 拷贝初始化，这里相当于隐式调用拷贝构造函数，而不是调用赋值运算符函数   5  6  7 |

**为了提高效率，允许编译器跳过创建临时对象这一步，**直接调用构造函数构造要创建的对象，这样 就完全等价于**直接初始化了**（语句1和语句3等价），但是需要辨别两种情况。

当拷贝构造函数为private时：语句3和语句4在编译时会报错

使用explicit修饰构造函数时：如果构造函数存在隐式转换，编译时会报错

[C++的直接初始化与复制初始化的区别：https://blog.csdn.net/qq936836/article/details/83450 218](https://blog.csdn.net/qq936836/article/details/83450218)

### 19、初始化和赋值的区别

对于简单类型来说，初始化和赋值没什么区别

对于类和复杂数据类型来说，这两者的区别就大了，举例如下：

|  |
| --- |
| class A {  public:  int num1;  int num2;  public:  A(int a = 0, int b = 0) :num1(a), num2(b) {};  A(const A& a) {};  //重载 = 号操作符函数  A& operator=(const A& a) {  num1 = a.num1 + 1;  num2 = a.num2 + 1;  return \*this  };  };  int main() {  A a(1, 1);  A a1 = a; //拷贝初始化操作，调用拷贝构造函数  A b;  b = a;//赋值操作，对象a中，num1 = 1，num2 = 1；对象b中，num1 = 2，num2 = 2  return 0;  } |

### 20、extern"C"的用法

为了能够**正确的在C++代码中调用C语言**的代码：在程序中加上extern "C"后，相当于告诉编译器这部分代码是C语言写的，因此要按照C语言进行编译，而不是C++；

### 23 编程实现strcpy函数

己知strcpy函数的原型是:

char \* strcpy(char \* strDest, const char \* strSrc) ;

(1) 不调用库函数，实现strcpy 函数。.

(2) 解释为什么要返回char \*.

|  |
| --- |
| void\* my\_memmove(void\* dst, const void\* src, size\_t n)  {  char\* s\_dst;  char\* s\_src;  s\_dst = (char\*)dst;  s\_src = (char\*)src;  if(s\_dst > s\_src && (s\_src+n > s\_dst)) { //第二种内存覆盖的情形。  s\_dst = s\_dst+n-1;  s\_src = s\_src+n-1;  while(n--) {  \*s\_dst-- = \*s\_src--;  }  }else {  while(n--) {  \*s\_dst++ = \*s\_src++;  }  }  return dst;  }  ————————————————  版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。    原文链接：https://blog.csdn.net/scarificed/article/details/127206289 |

### 23、C和C++的类型安全

什么是类型安全？

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。

“类型安全”常被用来形容编程语言，其根据在于该门编程语言是否提供保障类型安全的机制；有的时候 也用“类型安全”形容某个程序，判别的标准在于该程序是否隐含类型错误。

类型安全的编程语言与类型安全的程序之间，没有必然联系。好的程序员可以使用类型不那么安全的语 言写出类型相当安全的程序，相反的，差一点儿的程序员可能使用类型相当安全的语言写出类型不太安 全的程序。绝对类型安全的编程语言暂时还没有。

1. C的类型安全

C只在局部上下文中表现出类型安全，比如试图从一种结构体的指针转换成另一种结构体的指针时，编 译器将会报告错误，除非使用显式类型转换。然而，C中相当多的操作是不安全的。以下是两个十分常 见的例子：

printf格式输出



上述代码中，使用%d控制整型数字的输出，没有问题，但是改成%f时，明显输出错误，再改成%s时， 运行直接报segmentation fault错误

malloc函数的返回值

malloc是C中进行内存分配的函数，它的返回类型是void\*即空类型指针，常常有这样的用法char\* pStr=(char\*)malloc(100\*sizeof(char))，这里明显做了显式的类型转换。

类型匹配尚且没有问题，但是一旦出现int\* pInt=(int\*)malloc(100\*sizeof(char))就很可能带来一些问题，而这样的转换C并不会提示错误。

1. C++的类型安全

如果C++使用得当，它将远比C更有类型安全性。相比于C语言，C++提供了一些新的机制保障类型安 全：

操作符new返回的指针类型严格与对象匹配，而不是void\*

C中很多以void\*为参数的函数可以改写为C++模板函数，而模板是支持类型检查的；

引入const关键字代替#deﬁne constants，它是有类型、有作用域的，而#deﬁne constants只是简单的文本替换

一些#deﬁne宏可被改写为inline函数，结合函数的重载，可在类型安全的前提下支持多种类型， 当然改写为模板也能保证类型安全

C++提供了**dynamic\_cast**关键字，使得转换过程更加安全，因为dynamic\_cast比static\_cast涉及 更多具体的类型检查。

例1：使用void\*进行类型转换



例2：不同类型指针之间转换

|  |
| --- |
| 1. #include<iostream> 2. using namespace std; 3 3. class Parent{}; 4. class Child1 : public Parent 6 { 5. public: 6. int i; 7. Child1(int e):i(e){} 10 };   11 class Child2 : public Parent 12 {   1. public: 2. double d; 3. Child2(double e):d(e){} 16 };   17 int main() 18 {  19 Child1 c1(5);  20 Child2 c2(4.1);   1. Parent\* pp; 2. Child1\* pc1; 23   24 pp=&c1;   1. pc1=(Child1\*)pp; // 类型向下转换 强制转换，由于类型仍然为Child1\*，不造成错误 2. cout<<pc1->i<<endl; //输出：5 27   28 pp=&c2;  29 pc1=(Child1\*)pp; //强制转换，且类型发生变化，将造成错误30 cout<<pc1->i<<endl;// 输出：1717986918  31 return 0; 32 }  33  34  35  36 |

上面两个例子之所以引起类型不安全的问题，是因为程序员使用不得当。第一个例子用到了空类型指针void\*，第二个例子则是在两个类型指针之间进行强制转换。因此，想保证程序的类型安全性，应尽量 避免使用空类型指针void\*，尽量不对两种类型指针做强制转换。

### 24、为什么析构函数一般写成虚函数

由于类的多态性，基类指针可以指向派生类的对象，如果删除该基类的指针，就会调用该指针指向的派 生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。

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 数而不调用派生类析构函数，这样就会造成派生类对象析构不完全，造成内存泄漏。

所以将析构函数声明为虚函数是十分必要的。在实现多态时，当用基类操作派生类，在析构时防止只析 构基类而不析构派生类的状况发生，要将基类的析构函数声明为虚函数。

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3 3. class Parent{ 4. public: 5. Parent(){ 6. cout << "Parent construct function" << endl; 8 };   9 ~Parent(){  10 cout << "Parent destructor function" <<endl; 11 }  12 };  13   1. class Son : public Parent{ 2. public: 3. Son(){ 4. cout << "Son construct function" << endl; 18 };   19 ~Son(){  20 cout << "Son destructor function" <<endl; 21 }  22 };  23  24 int main() 25 {   1. Parent\* p = new Son(); 2. delete p; 3. p = NULL; 4. return 0; 30 } 5. //运行结果： 6. //Parent construct function 7. //Son construct function 8. //Parent destructor function 35   36  37 |

将基类的析构函数声明为虚函数：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3 3. class Parent{ 4. public: 5. Parent(){ 6. cout << "Parent construct function" << endl; 8 };   9 virtual ~Parent(){  10 cout << "Parent destructor function" <<endl; 11 }  12 };  13   1. class Son : public Parent{ 2. public: 3. Son(){ 4. cout << "Son construct function" << endl; 18 };   19 ~Son(){  20 cout << "Son destructor function" <<endl; 21 }  22 };  23  24 int main() 25 {   1. Parent\* p = new Son(); 2. delete p; 3. p = NULL; 4. return 0; 30 } 5. //运行结果： 6. //Parent construct function 7. //Son construct function 8. //Son destructor function 9. //Parent destructor function 36   37  38 |

但存在一种特例，在 CRTP 模板中，不应该将析构函数声明为虚函数，理论上所有的父类函数都不应该声明为虚函数，因为这种继承方式，不需要虚函数表。

### 25、构造函数能否声明为虚函数或者纯虚函数，析构函数呢？

析构函数：

析构函数可以为虚函数，并且一般情况下基类析构函数要定义为虚函数。

只有在基类析构函数定义为虚函数时，调用操作符delete销毁指向对象的基类指针时，才能准确调 用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。

**析构函数可以是纯虚函数**，含有纯虚函数的类是抽象类，此时不能被实例化。但派生类中可以根据 自身需求重新改写基类中的纯虚函数。

构造函数：

构造函数不能定义为虚函数。在构造函数中可以调用虚函数，不过此时调用的是正在构造的类中的 虚函数，而不是子类的虚函数，因为此时子类尚未构造好。

虚函数对应一个vtable(虚函数表)，类中存储一个vptr指向这个vtable。如果构造函数是虚函数， 就需要通过vtable调用，可是对象没有初始化就没有vptr，无法找到vtable，所以构造函数不能是 虚函数。

### 26、C++中的重载、重写（覆盖）和隐藏的区别

1. 重载（overload）

重载是指在同一范围定义中的同名成员函数才存在重载关系。主要特点是函数名相同，参数类型和数目 有所不同，不能出现参数个数和类型均相同，仅仅依靠返回值不同来区分的函数。重载和函数成员是否 是虚函数无关。举个例子：

|  |
| --- |
| 1 class A{  2 ...   1. virtual int fun(); 2. void fun(int); 3. void fun(double, double); 4. static int fun(char);   7 ...  8 }  9  10  11 |

1. 重写（覆盖）（override）

重写指的是在派生类中覆盖基类中的同名函数，**重写就是重写函数体**，**要求基类函数必须是虚函数**且： 与基类的虚函数有相同的参数个数

与基类的虚函数有相同的参数类型

与基类的虚函数有相同的返回值类型

举个例子：

|  |
| --- |
| 1. //父类 2. class A{ 3. public: 4. virtual int fun(int a){} 5 } 5. //子类 6. class B : public A{ 7. public: 8. //重写,一般加override可以确保是重写父类的函数 9. virtual int fun(int a) override{} 11 }   12  13  14 |

重载与重写的区别：

重写是父类和子类之间的垂直关系，重载是不同函数之间的水平关系重写要求参数列表相同，重载则要求参数列表不同，返回值不要求

重写关系中，调用方法根据对象类型决定，重载根据调用时实参表与形参表的对应关系来选择函数 体

1. 隐藏（hide）

隐藏指的是某些情况下，派生类中的函数屏蔽了基类中的同名函数，包括以下情况：

两个函数参数相同，但是基类函数不是虚函数。**和重写的区别在于基类函数是否是虚函数。**举个例 子：

|  |
| --- |
| 1. //父类 2. class A{ 3. public: 4. void fun(int a){ 5. cout << "A中的fun函数" << endl; 6 }   7 };   1. //子类 2. class B : public A{ 3. public: 4. //隐藏父类的fun函数 5. void fun(int a){ 6. cout << "B中的fun函数" << endl; 14 }   15 };   1. int main(){ 2. B b; 3. b.fun(2); //调用的是B中的fun函数 4. b.A::fun(2); //调用A中fun函数 5. return 0; 21 }   22  23  24 |

两个函数参数不同，无论基类函数是不是虚函数，都会被隐藏。和重载的区别在于两个函数不在同 一个类中。举个例子：

//父类

class A {

public:

virtual void fun(int a) {

*cout* << "A中的fun函数" << *endl*;

}

};

//子类

class B : public A {

public:

//隐藏父类的fun函数

virtual void fun(char\* a) {

*cout* << "A中的fun函数" << *endl*;

}

};

int main() {

B b;

b.fun(2); //报错，调用的是B中的fun函数，参数类型不对

b.A::fun(2); //调用A中fun函数

return 0;

}

### 27、C++的多态如何实现

C++的多态性，**一言以蔽之**就是：

在基类的函数前加上**virtual**关键字，在派生类中重写该函数，运行时将会根据所指对象的实际类型来 调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的 函数。

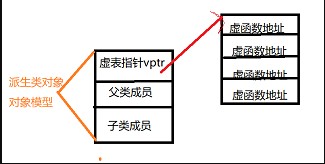
举个例子：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3 3. class Base{ 4. public: 5. virtual void fun(){ 6. cout << " Base::func()" <<endl; 8 }   9 };  10   1. class Son1 : public Base{ 2. public: 3. virtual void fun() override{ 4. cout << " Son1::func()" <<endl; 15 }   16 };  17  18 class Son2 : public Base{ 19  20 };  21  22 int main() 23 {   1. Base\* base = new Son1; 2. base->fun(); 3. base = new Son2; 4. base->fun(); 5. delete base; 6. base = NULL; 7. return 0; 31 } 8. // 运行结果 9. // Son1::func() 10. // Base::func() 35   36 |

|  |
| --- |
| 37 |

例子中，Base为基类，其中的函数为虚函数。子类1继承并重写了基类的函数，子类2继承基类但没有 重写基类的函数，从结果分析子类体现了多态性，那么为什么会出现多态性，其底层的原理是什么？这 里需要引出虚表和虚基表指针的概念。

虚表：虚函数表的缩写，类中含有virtual关键字修饰的方法时，编译器会自动生成虚表

虚表指针：在含有虚函数的类实例化对象时，对象地址的前四个字节存储的指向虚表的指针

上图中展示了虚表和虚表指针在基类对象和派生类对象中的模型，下面阐述实现多态的过程：

1. 编译器在发现基类中有虚函数时，会自动为每个含有虚函数的类生成一份虚表，该表是一个一维 数组，虚表里保存了虚函数的入口地址
2. 编译器会在每个对象的前四个字节中保存一个虚表指针，即**vptr**，指向对象所属类的虚表。在构 造时，根据对象的类型去初始化虚指针vptr，从而让vptr指向正确的虚表，从而在调用虚函数时，能找 到正确的函数
3. 所谓的合适时机，在派生类定义对象时，程序运行会自动调用构造函数，在构造函数中创建虚表 并对虚表初始化。在构造子类对象时，会先调用父类的构造函数，此时，编译器只“看到了”父类，并为 父类对象初始化虚表指针，令它指向父类的虚表；当调用子类的构造函数时，为子类对象初始化虚表指 针，令它指向子类的虚表
4. 当派生类对基类的虚函数没有重写时，派生类的虚表指针指向的是基类的虚表；当派生类对基类 的虚函数重写时，派生类的虚表指针指向的是自身的虚表；当派生类中有自己的虚函数时，在自己的虚 表中将此虚函数地址添加在后面

这样指向派生类的基类指针在运行时，就可以根据派生类对虚函数重写情况动态的进行调用，从而实现 多态性。

### 28、C++有哪几种的构造函数 \*\*\*\*\*

C++中的构造函数可以分为4类：

默认构造函数

初始化构造函数（有参数） 拷贝构造函数

移动构造函数（move和右值引用） 委托构造函数

转换构造函数

举个例子：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3 3. class Student{ 4. public: 5. Student(){//默认构造函数，没有参数 6. this->age = 20;   8 this->num = 1000;  9 };   1. Student(int a, int n):age(a), num(n){}; //初始化构造函数，有参数和参数列表 2. Student(const Student& s){//拷贝构造函数，这里与编译器生成的一致 3. this->age = s.age; 4. this->num = s.num; 14 }; 5. Student(int r){ //转换构造函数,形参是其他类型变量，且只有一个形参 6. this->age = r;   17 this->num = 1002;  18 };   1. ~Student(){} 2. public: 3. int age; 4. int num; 23 };   24   1. int main(){ 2. Student s1;   27 Student s2(18,1001);   1. int a = 10; 2. Student s3(a); 3. Student s4(s3); 31 4. printf("s1 age:%d, num:%d\n", s1.age, s1.num); 5. printf("s2 age:%d, num:%d\n", s2.age, s2.num); 6. printf("s3 age:%d, num:%d\n", s3.age, s3.num); 7. printf("s2 age:%d, num:%d\n", s4.age, s4.num); 8. return 0; 37 }   38 //运行结果  39 //s1 age:20, num:1000 40 //s2 age:18, num:1001 41 //s3 age:10, num:1002 42 //s2 age:10, num:1002 43  44  45 |

默认构造函数和初始化构造函数在定义类的对象，完成对象的初始化工作复制构造函数用于复制本类的对象

转换构造函数用于将其他类型的变量，隐式转换为本类对象

《浅谈C++中的几种构造函数》：<https://blog.csdn.net/zxc024000/article/details/51153743>

### 29、浅拷贝和深拷贝的区别

浅拷贝

浅拷贝只是拷贝一个指针，并没有新开辟一个地址，拷贝的指针和原来的指针指向同一块地址，如果原 来的指针所指向的资源释放了，那么再释放浅拷贝的指针的资源就会出现错误。

深拷贝

深拷贝不仅拷贝值，还开辟出一块新的空间用来存放新的值，即使原先的对象被析构掉，释放内存了也 不会影响到深拷贝得到的值。在自己实现拷贝赋值的时候，如果有指针变量的话是需要自己实现深拷贝 的。

#include <iostream>

#include <string.h>

using namespace *std*;

class Student

{

private:

int *num*;

char\* *name*;

public:

Student() {

*name* = new char(20);

*cout* << "Student" << *endl*;

};

~Student() {

*cout* << "~Student " << &*name* << *endl*;

delete *name*;

*name* = *NULL*;

};

Student(const Student& *s*) {//拷贝构造函数

//浅拷贝，当对象的name和传入对象的name指向相同的地址

*name* = *s*.*name*;

//深拷贝

//name = new char(20);

//memcpy(name, s.name, strlen(s.name));

*cout* << "copy Student" << *endl*;

};

};

int main()

{

{// 花括号让s1和s2变成局部对象，方便测试

Student s1;

Student s2(s1);// 复制对象

}

*system*("pause");

return 0;

}

//浅拷贝执行结果：

//Student

//copy Student

//~Student 0x7fffed0c3ec0

//~Student 0x7fffed0c3ed0

//\*\*\* Error in `/tmp/815453382/a.out': double free or corruption (fasttop):

//0x0000000001c82c20 \* \*\*

//深拷贝执行结果：

//Student

//copy Student

//~Student 0x7fffebca9fb0

//~Student 0x7fffebca9fc0

从执行结果可以看出，浅拷贝在对象的拷贝创建时存在风险，即被拷贝的对象析构释放资源之后，拷贝 对象析构时会再次释放一个已经释放的资源，深拷贝的结果是两个对象之间没有任何关系，各自成员地 址不同。

### 41、public，protected和private访问和继承权限/public/protected/private

**的区别？**

public的变量和函数在类的内部外部都可以访问。protected的变量和函数只能在类的内部和其派生类中访问。private修饰的元素只能在类内访问。

（一）访问权限

派生类可以继承基类中除了构造/析构、赋值运算符重载函数之外的成员，但是这些成员的访问属性在 派生过程中也是可以调整的，三种派生方式的访问权限如下表所示：注意外部访问并不是真正的外部访 问，而是在通过派生类的对象对基类成员的访问。



派生类对基类成员的访问形象有如下两种：

内部访问：由派生类中新增的成员函数对从基类继承来的成员的访问

**外部访问**：在派生类外部，通过派生类的对象对从基类继承来的成员的访问

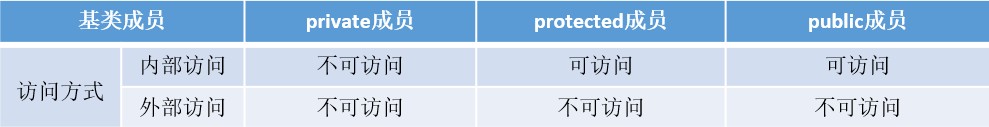
（二）继承权限

public继承

公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，都保持原有的状态，而基类的私有 成员任然是私有的，不能被这个派生类的子类所访问

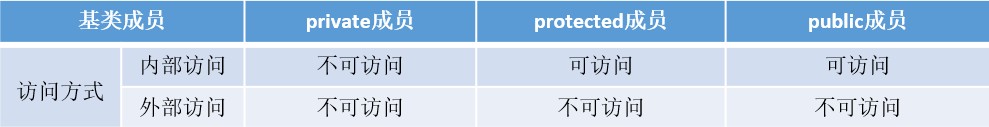
protected继承

保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成 员函数或友元函数访问，基类的私有成员仍然是私有的，访问规则如下表



private继承

私有继承的特点是基类的所有公有成员和保护成员都成为派生类的私有成员，并不被它的派生类的子类 所访问，基类的成员只能由自己派生类访问，无法再往下继承，访问规则如下表



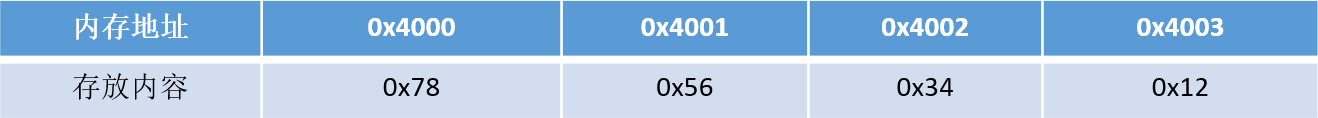
### 如何用代码判断大小端存储

大端存储：字数据的高字节存储在低地址中

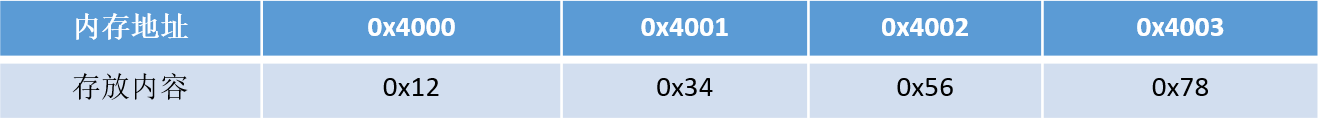
小端存储：字数据的低字节存储在低地址中例如：32bit的数字0x12345678

所以在Socket编程中，往往需要将操作系统所用的小端存储的IP地址转换为大端存储，这样才能进行网 络传输

小端模式中的存储方式为：



大端模式中的存储方式为：



了解了大小端存储的方式，如何在代码中进行判断呢？下面介绍两种判断方式：

**方式一：使用强制类型转换**-这种法子不错

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3. int main() 4 {   5 int a = 0x1234;   1. //由于int和char的长度不同，借助int型转换成char型，只会留下低地址的部分 2. char c = (char)(a);   8 if (c == 0x12)  9 cout << "big endian" << endl;   1. else if(c == 0x34) 2. cout << "little endian" << endl; 12 }   13  14  15 |

方式二：巧用union联合体

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3. //union联合体的重叠式存储，endian联合体占用内存的空间为每个成员字节长度的最大值 4. union endian 5 { 5. int a; 6. char ch; 8 };   9 int main() 10 {  11 endian value;  12 value.a = 0x1234;   1. //a和ch共用4字节的内存空间 2. if (value.ch == 0x12) 3. cout << "big endian"<<endl; 4. else if (value.ch == 0x34) 5. cout << "little endian"<<endl; 18 }   19  20  21 |

### 43、什么情况下会调用拷贝构造函数

用类的一个实例化对象去初始化另一个对象的时候函数的参数是类的对象时（非引用传递）

函数的返回值是函数体内局部对象的类的对象时 ,此时虽然发生（Named return Value优化）NRV

优化，但是由于返回方式是值传递，所以会在返回值的地方调用拷贝构造函数

另：第三种情况在Linux g++ 下则不会发生拷贝构造函数，不仅如此即使返回局部对象的引用，依然不会发生拷贝构造函数

总结就是：即使发生NRV优化的情况下，Linux+ g++的环境是不管值返回方式还是引用方式返回的方式都不会发生拷贝构造函数，而Windows + VS2019在值返回的情况下发生拷贝构造函数，引用返回方式则不发生拷贝构造函数。

在c++编译器发生NRV优化，如果是引用返回的形式则不会调用拷贝构造函数，如果是值传递的方式依 然会发生拷贝构造函数。

在VS2019下进行下述实验：

举个例子：

class A {

public:

A() {};

A(const A& a)

{

*cout* << "copy constructor is called" << *endl*;

};

~A() {};

};

void useClassA(A a) {}

A getClassA()//此时会发生拷贝构造函数的调用，虽然发生NRV优化，但是依然调用拷贝构造函数

{

A a;

return a;

}

//A& getClassA2()// VS2019下，此时编辑器会进行（Named return Value优化）NRV优化,

// 不调用拷贝构造函数 ，如果是引用传递的方式返回当前函数体内生成的对象时，并不发生拷贝构造函数

// 的调用

//{

// A a;

// return a;

//}

int main()

{

A a1, a2, a3, a4;

A a2 = a1;  //调用拷贝构造函数,对应情况1

useClassA(a1);//调用拷贝构造函数，对应情况2

a3 = getClassA();//发生NRV优化，但是值返回，依然会有拷贝构造函数的调用 情况3

a4 = getClassA2(a1);//发生NRV优化，且引用返回自身，不会调用

return 0;

}

情况1比较好理解

情况2的实现过程是，调用函数时先根据传入的实参产生临时对象，再用拷贝构造去初始化这个临时对 象，在函数中与形参对应，函数调用结束后析构临时对象

情况3在执行return时，理论的执行过程是：产生临时对象，调用拷贝构造函数把返回对象拷贝给临时 对象，函数执行完先析构局部变量，再析构临时对象， 依然会调用拷贝构造函数

### 44、C++中有几种类型的new

在C++中，new有三种典型的使用方法：plain new，nothrow new和placement new

1. plain new

言下之意就是普通的new，就是我们常用的new，在C++中定义如下：

|  |
| --- |
| 1. void\* operator new(std::size\_t) throw(std::bad\_alloc); 2. void operator delete(void \*) throw(); 3   4  5  6 |

因此**plain new**在空间分配失败的情况下，抛出异常**std::bad\_alloc**而不是返回NULL，因此通过判断返回值是否为NULL是徒劳的，举个例子：

|  |
| --- |
| 7 #include <iostream>  #include <string>  using namespace std;  int main()  {  try  {   1. char \*p = new char[10e11]; 2. delete p;   10 }  11 catch (const std::bad\_alloc &ex) 12 {  13 cout << ex.what() << endl; 14 }  15 return 0; 16 }  17 //执行结果：bad allocation  18  19  20  21 |

1. nothrow new

nothrow new在空间分配失败的情况下是不抛出异常，而是返回NULL，定义如下：

|  |
| --- |
| 1. void \* operator new(std::size\_t,const std::nothrow\_t&) throw(); 2. void operator delete(void\*) throw(); 3   4  5  6 |

举个例子：

|  |
| --- |
| 1. #include <iostream> 2. #include <string> 3. using namespace std; 4   5 int main() 6 {   1. char \*p = new(nothrow) char[10e11]; 2. if (p == NULL)   9 {  10 cout << "alloc failed" << endl; 11 }   1. delete p; 2. return 0; 14 }   15 //运行结果：alloc failed  16  17  18  19 |

1. placement new

这种new允许在一块已经分配成功的内存上重新构造对象或对象数组。placement new不用担心内存分配失败，因为它根本不分配内存，它做的唯一一件事情就是调用对象的构造函数。定义如下：

|  |
| --- |
| 1. void\* operator new(size\_t,void\*); 2. void operator delete(void\*,void\*); 3   4  5  6 |

使用placement new需要注意两点：

palcement new的主要用途就是反复使用一块较大的动态分配的内存来构造不同类型的对象或者他们的数组

placement new构造起来的对象数组，要显式的调用他们的析构函数来销毁（析构函数并不释放对象的内存），千万不要使用delete，这是因为placement new构造起来的对象或数组大小并不一定等于原来分配的内存大小，使用delete会造成内存泄漏或者之后释放内存时出现运行时错误。

举个例子：

|  |
| --- |
| 1. #include <iostream> 2. #include <string> 3. using namespace std; 4. class ADT{ 5. int i; 6. int j; 7. public: 8. ADT(){   9 i = 10;  10 j = 100;  11 cout << "ADT construct i=" << i << "j="<<j <<endl; 12 }  13 ~ADT(){  14 cout << "ADT destruct" << endl; 15 }  16 };  17 int main() 18 {   1. char \*p = new(nothrow) char[sizeof ADT + 1]; 2. if (p == NULL) { 3. cout << "alloc failed" << endl; 22 } 4. ADT \*q = new(p) ADT; //placement new:不必担心失败，只要p所指对象的的空间足够   ADT创建即可   1. //delete q;//错误!不能在此处调用delete q; 2. q->ADT::~ADT();//显示调用析构函数 3. delete[] p; 4. return 0; 28 } 5. //输出结果： 6. //ADT construct i=10j=100 7. //ADT destruct 32   33  34  35 |

《【C++】几种类型的new介绍》：<https://www.jianshu.com/p/9b57e769c3cb>

### 45、C++中NULL和nullptr区别

算是为了与C语言进行兼容而定义的一个问题吧

NULL来自C语言，一般由宏定义实现，而 nullptr 则是C++11的新增关键字。**在C语言中，NULL被定义为(void\*)0,而在C++语言中，NULL则被定义为整数0**。编译器一般对其实际定义如下：

|  |
| --- |
| 1. #ifdef cplusplus 2. #define NULL 0 3. #else 4. #define NULL ((void \*)0) 5. #endif 6   7  8  9 |

在C++中指针必须有明确的类型定义。但是将NULL定义为0带来的另一个问题是无法与整数的0区分。 因为C++中允许有函数重载，所以可以试想如下函数定义情况：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3 3. void fun(char\* p) { 4. cout << "char\*" << endl; 6 }   7   1. void fun(int p) { 2. cout << "int" << endl; 10 }   11  12 int main() 13 {   1. fun(NULL); 2. return 0; 16 }   17 //输出结果：int  18  19  20  21 |

那么在传入NULL参数时，会把NULL当做整数0来看，如果我们想调用参数是指针的函数，该怎么办

呢?。nullptr在C++11被引入用于解决这一问题，nullptr可以明确区分整型和指针类型，能够根据环境 自动转换成相应的指针类型，但不会被转换为任何整型，所以不会造成参数传递错误。

nullptr的一种实现方式如下：

|  |
| --- |
| 1. const class nullptr\_t{ 2. public: 3. template<class T> inline operator T\*() const{ return 0; } 4. template<class C, class T> inline operator T C::\*() const { return 0; } 5. private: 6. void operator&() const; 7. } nullptr = {}; 8   9  10  11 |

以上通过模板类和运算符重载的方式来对不同类型的指针进行实例化从而解决了(void\*)指针带来参数类 型不明的问题，**另外由于nullptr是明确的指针类型，所以不会与整形变量相混淆。**但nullptr仍然存在 一定问题，例如：

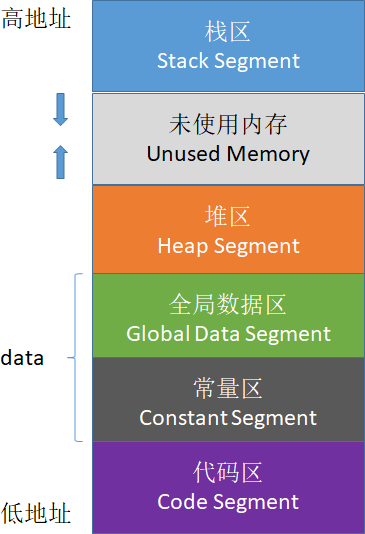
|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3   4 void fun(char\* p) 5 {  6 cout<< "char\* p" <<endl; 7 }  8 void fun(int\* p) 9 {  10 cout<< "int\* p" <<endl; 11 }  12  13 void fun(int p) 14 {  15 cout<< "int p" <<endl; 16 }  17 int main() 18 {   1. fun((char\*)nullptr);//语句1 2. fun(nullptr);//语句2 3. fun(NULL);//语句3 4. return 0; 23 } 5. //运行结果： 6. //语句1：char\* p 7. //语句2:报错，有多个匹配 8. //3：int p 28   29  30 |

在这种情况下存在对不同指针类型的函数重载，此时如果传入nullptr指针则仍然存在无法区分应实际调 用哪个函数，这种情况下必须显示的指明参数类型。

《NULL和nullptr区别》：<https://blog.csdn.net/qq_39380590/article/details/82563571>

### 46、简要说明C++的内存分区

C++中的内存分区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区和代码区。如下图所示



**栈**：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

**堆**：就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个

new 就要对应一个 delete 。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收

**自由存储区**：就是那些由 malloc 等分配的内存块，它和堆是十分相似的，不过它是用 free 来结束自己的生命的

**全局/静态存储区**：全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量和静态变 量又分为初始化的和未初始化的，在C++里面没有这个区分了，它们共同占用同一块内存区，在该区定 义的变量若没有初始化，则会被自动初始化，例如int型变量自动初始为0

**常量存储区**：这是一块比较特殊的存储区，这里面存放的是常量，不允许修改

**代码区**：存放函数体的二进制代码

《C/C++内存管理详解》：<https://chenqx.github.io/2014/09/25/Cpp-Memory-Management/>

### 47、C++的异常处理的方法

在程序执行过程中，由于程序员的疏忽或是系统资源紧张等因素都有可能导致异常，任何程序都无法保 证绝对的稳定，常见的异常有：

数组下标越界

除法计算时除数为0

动态分配空间时空间不足

...

如果不及时对这些异常进行处理，程序多数情况下都会崩溃。

1. try、throw和catch关键字

C++中的异常处理机制主要使用**try**、**throw**和**catch**三个关键字，其在程序中的用法如下：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3. int main() 4 { 4. double m = 1, n = 0; 5. try { 6. cout << "before dividing." << endl;   8 if (n == 0)  9 throw - 1; //抛出int型异常   1. else if (m == 0) 2. throw - 1.0; //拋出 double 型异常 3. else 4. cout << m / n << endl; 5. cout << "after dividing." << endl; 15 } 6. catch (double d) { 7. cout << "catch (double)" << d << endl; 18 }   19 catch (...) {  20 cout << "catch (...)" << endl; 21 }   1. cout << "finished" << endl; 2. return 0; 24 } 3. //运行结果 4. //before dividing. 27 //catch (...)   28 //finished 29  30  31 |

代码中，对两个数进行除法计算，其中除数为0。可以看到以上三个关键字，程序的执行流程是先执行 try包裹的语句块，如果执行过程中没有异常发生，则不会进入任何catch包裹的语句块，如果发生异 常，则使用throw进行异常抛出，再由catch进行捕获，throw可以抛出各种数据类型的信息，代码中使 用的是数字，也可以自定义异常class。**catch根据throw抛出的数据类型进行精确捕获（不会出现类型 转换），如果匹配不到就直接报错，可以使用catch(...)的方式捕获任何异常（不推荐）。**当然，如果catch了异常，当前函数如果不进行处理，或者已经处理了想通知上一层的调用者，可以在catch里面再throw异常。

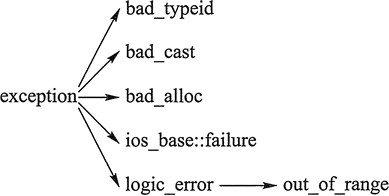
1. 函数的异常声明列表

有时候，程序员在定义函数的时候知道函数可能发生的异常，可以在函数声明和定义时，指出所能抛出 异常的列表，写法如下：

|  |
| --- |
| 1 int fun() throw(int,double,A,B,C){...}; 2  3  4 |

这种写法表名函数可能会抛出int,double型或者A、B、C三种类型的异常，如果throw中为空，表明不 会抛出任何异常，如果没有throw则可能抛出任何异常

1. C++标准异常类 exception

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来的，如下图所示

bad\_typeid：使用typeid运算符，如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会拋出此异常，例如：

|  |
| --- |
| 1. #include <iostream> 2. #include <typeinfo> 3. using namespace std; 4 4. class A{ 5. public: 6. virtual ~A(); 8 };   9   1. using namespace std; 2. int main() { 3. A\* a = NULL; 4. try { 5. cout << typeid(\*a).name() << endl; // Error condition 15 } 6. catch (bad\_typeid){ 7. cout << "Object is NULL" << endl; 18 }   19 return 0; 20 }  21 //运行结果：bject is NULL  22  23  24  25 |

bad\_cast：在用 dynamic\_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会拋出此异常

bad\_alloc：在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常

out\_of\_range:用 vector 或 string的at 成员函数根据下标访问元素时，如果下标越界，则会拋出此异常

《C++异常处理（try catch throw）完全攻略》：<http://c.biancheng.net/view/422.html>

### 48、static的用法和作用？

1. 先来介绍它的第一条也是最重要的一条：隐藏。（static函数，static变量均可） 当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性。
2. static的第二个作用是保持变量内容的持久。（static变量中的记忆功能和全局生存期）存储在静态数 据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存 储区：全局变量和static变量，只不过和全局变量比起来，static可以控制变量的可见范围，说到底static还是用来隐藏的。
3. static的第三个作用是默认初始化为0（static变量）

其实全局变量也具备这一属性，因为全局变量也存储在静态数据区。在静态数据区，内存中所有的字节 默认值都是0x00，某些时候这一特点可以减少程序员的工作量。

1. static的第四个作用：C++中的类成员声明static
2. 函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其 值在下次调用时仍维持上次的值；
3. 在模块内的static全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；
4. 在模块内的static函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模 块内；
5. 在类中的static成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；
6. 在类中的static成员函数属于整个类所拥有，这个函数不接收this指针，因而只能访问类的static成员 变量。

类内：

1. static类对象必须要在类外进行初始化，static修饰的变量先于对象存在，所以static修饰的变量要在 类外初始化；
2. 由于static修饰的类成员属于类，不属于对象，因此static类成员函数是没有this指针的，this指针是 指向本对象的指针。正因为没有this指针，所以static类成员函数不能访问非static的类成员，只能访问static修饰的类成员；
3. static成员函数不能被virtual修饰，static成员不属于任何对象或实例，所以加上virtual没有任何实际 意义；静态成员函数没有this指针，虚函数的实现是为每一个对象分配一个vptr指针，而vptr是通过this 指针调用的，所以不能为virtual；虚函数的调用关系，this->vptr->ctable->virtual function

### 49、静态变量什么时候初始化

如果是静态初始化，是在main()函数之前，程序加载时初始化；如果是动态初始化，则要区分是局部静态变量还是全局变量或者全局静态变量，局部静态变量则在函数第一次执行到时初始化，否则在main()函数之前，运行时初始化。

1. 初始化只有一次，但是可以多次赋值，在主程序之前，编译器已经为其分配好了内存。
2. 静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配 好了内存，但在C和C++中静态局部变量的初始化节点又有点不太一样。在C中，初始化发生在代码执行 之前，编译阶段分配好内存之后，就会进行初始化，所以我们看到在C语言中无法使用变量对静态局部 变量进行初始化，在程序运行结束，变量所处的全局内存会被全部回收。
3. 而在C++中，初始化时在执行相关代码时才会进行初始化，主要是由于C++引入对象后，要进行初始 化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的 特定操作，并非简单地分配内存。所以C++标准定为全局或静态对象是有首次用到时才会进行构造，并 通过atexit()来管理。在程序结束，按照构造顺序反方向进行逐个析构。所以在C++中是可以使用变量对 静态局部变量进行初始化的。

### 50、const关键字?

1. 阻止一个变量被改变，可以使用const关键字。在定义该const变量时，通常需要对它进行初始化， 因为以后就没有机会再去改变它了；
2. 对指针来说，可以指定指针本身为const，也可以指定指针所指的数据为const，或二者同时指定为

const；

1. 在一个函数声明中，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；
2. 对于类的成员函数，若指定其为const类型，则表明其是一个常函数，不能修改类的成员变量，类的 常对象只能访问类的常成员函数；
3. 对于类的成员函数，有时候必须指定其返回值为const类型，以使得其返回值不为“左值”。
4. const成员函数可以访问非const对象的非const数据成员、const数据成员，也可以访问const对象内 的所有数据成员；
5. 非const成员函数可以访问非const对象的非const数据成员、const数据成员，但不可以访问const对 象的任意数据成员；
6. 一个没有明确声明为const的成员函数被看作是将要修改对象中数据成员的函数，而且编译器不允许 它为一个const对象所调用。因此const对象只能调用const成员函数。
7. const类型变量可以通过类型转换符const\_cast将const类型转换为非const类型；
8. const类型变量必须定义的时候进行初始化，因此也导致如果类的成员变量有const类型的变量，那 么该变量必须在类的初始化列表中进行初始化；
9. 对于函数值传递的情况，因为参数传递是通过复制实参创建一个临时变量传递进函数的，函数内只 能改变临时变量，但无法改变实参。则这个时候无论加不加const对实参不会产生任何影响。但是在引 用或指针传递函数调用中，因为传进去的是一个引用或指针，这样函数内部可以改变引用或指针所指向 的变量，这时const 才是实实在在地保护了实参所指向的变量。因为在编译阶段编译器对调用函数的选择是根据实参进行的，所以，只有引用传递和指针传递可以用是否加const来重载。一个拥有顶层const 的形参无法和另一个没有顶层const的形参区分开来。

### 52、形参与实参的区别？

1. 形参变量只有在被调用时才分配内存单元，在调用结束时， 即刻释放所分配的内存单元。因此，形参只有在函数内部有效。 函数调用结束返回主调函数后则不能再使用该形参变量。
2. 实参可以是常量、变量、表达式、函数等， 无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值， 以便把这些值传送给形参。 因此应预先用赋值，输入等办法使实参获得确定值， 会产生一个临时变量。
3. 实参和形参在数量上，类型上，顺序上应严格一致， 否则会发生“类型不匹配”的错误。
4. 函数调用中发生的数据传送是单向的。 即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。 因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。
5. 当形参和实参不是指针类型时，在该函数运行时，形参和实参是不同的变量，他们在内存中位于不 同的位置，形参将实参的内容复制一份，在该函数运行结束的时候形参被释放，而实参内容不会改变。

### 53、值传递、指针传递、引用传递的区别和效率

1. 值传递：有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象 或是大的结构体对象，将耗费一定的时间和空间。（传值）
2. 指针传递：同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为4字节的地 址。（传值，传递的是地址值）
3. 引用传递：同样有上述的数据拷贝过程，但其是针对地址的，相当于为该数据所在的地址起了一个 别名。（传地址）
4. 效率上讲，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、 清晰。

### 54、什么是类的继承？

1. 类与类之间的关系

has-A包含关系，用以描述一个类由多个部件类构成，实现has-A关系用类的成员属性表示，即一个类的 成员属性是另一个已经定义好的类；

use-A，一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来 实现；

is-A，继承关系，关系具有传递性；

1. 继承的相关概念

所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称 为子类或者派生类，被继承的类称为父类或者基类；

1. 继承的特点

子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，子类对象可以当做父类对象使 用；

1. 继承中的访问控制

public、protected、private

1. 继承中的构造和析构函数
2. 继承中的兼容性原则

### 55、什么是内存池，如何实现

内存池（Memory Pool） 是一种**内存分配**方式。通常我们习惯直接使用new、malloc 等申请内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性 能。内存池则是在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备 用。当有新的内存需求时，就从内存池中分出一部分内存块， 若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升。

这里简单描述一下《STL源码剖析》中的内存池实现机制：

allocate包装malloc,deallocate包装free

一般是一次20\*2个的申请，先用一半，留着一半，为什么也没个说法，侯捷在STL那边书里说好像是

C++委员会成员认为20是个比较好的数字，既不大也不小

* 1. 首先客户端会调用malloc()配置一定数量的区块（固定大小的内存块，通常为8的倍数），假设40 个32bytes的区块，其中20个区块（一半）给程序实际使用，1个区块交出，另外19个处于维护状 态。剩余20个（一半）留给内存池，此时一共有（20\*32byte）
  2. 客户端之后有有内存需求，想申请（20\*64bytes）的空间，这时内存池只有（20\*32bytes），就 先将（10\*64bytes)个区块返回，1个区块交出，另外9个处于维护状态，此时内存池空空如也
  3. 接下来如果客户端还有内存需求，就必须再调用malloc()配置空间，此时新申请的区块数量会增加 一个随着配置次数越来越大的附加量，同样一半提供程序使用，另一半留给内存池。申请内存的时 候用永远是先看内存池有无剩余，有的话就用上，然后挂在0-15号某一条链表上，要不然就重新 申请。
  4. 如果整个堆的空间都不够了，就会在原先已经分配区块中寻找能满足当前需求的区块数量，能满足 就返回，不能满足就向客户端报bad\_alloc异常

allocator就是用来分配内存的，最重要的两个函数是allocate和deallocate，就是用来申请内存和回收 内存的，外部（一般指容器）调用的时候只需要知道这些就够了。内部实现，目前的所有编译器都是直 接调用的::operator new()和::operator delete()，说白了就是和直接使用new运算符的效果是一样的， 所以说它们都没做任何特殊处理。

最开始GC2.9之前：

new和 operator new 的区别：new 是个运算符，编辑器会调用 operator new(0)

operator new()里面有调用malloc的操作，那同样的 operator delete()里面有调用的free的操作

GC2.9的alloc的一个比较好的分配器的实现规则

维护一条0-15号的一共16条链表，其中0表示8 bytes ，1表示 16 bytes,2表示 24bytes。。。。而15 表示 16\* 8 = 128bytes，如果在申请时并不是8的倍数，那就找刚好能满足内存大小的那个位置。比如想申请 12，那就是找16了，想申请 20 ，那就找 24 了

但是现在GC4.9及其之后 也还有，变成\_pool\_alloc这个名字了，不再是默认的了，你需要自己去指定它可以自己指定，比如说vector<string, gnu\_cxx::pool\_alloc> vec;这样来使用它，现在用的又回到以前

那种对malloc和free的包装形式了

### 55、从汇编层去解释一下引用

|  |
| --- |
| 1 9: int x = 1;  2  3 00401048 mov dword ptr [ebp-4],1  4  5 10: int &b = x;  6  7 0040104F lea eax,[ebp-4] 8  9 00401052 mov dword ptr [ebp-8],eax 10  11  12 |

x的地址为ebp-4，b的地址为ebp-8，因为栈内的变量内存是从高往低进行分配的，所以b的地址比x的 低。

lea eax,[ebp-4] 这条语句将x的地址ebp-4放入eax寄存器

mov dword ptr [ebp-8],eax 这条语句将eax的值放入b的地址

ebp-8中上面两条汇编的作用即：将x的地址存入变量b中，这不和将某个变量的地址存入指针变量是一 样的吗？所以从汇编层次来看，的确引用是通过指针来实现的。

### 56、深拷贝与浅拷贝是怎么回事？

1. 浅复制 ：只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变 了，那么浅复制出来的对象也会相应改变。

深复制 ：在计算机中开辟了一块新的内存地址用于存放复制的对象。

1. 在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复 制给另一个对象，如A=B。这时，如果B中有一个成员变量指针已经申请了内存，那A中的那个成员变量 也指向同一块内存。这就出现了问题：当B把内存释放了（如：析构），这时A内的指针就是野指针了， 出现运行错误。

### 57、C++模板是什么，你知道底层怎么实现的？

1. 编译器并不是把函数模板处理成能够处理任意类的函数；编译器从函数模板通过具体类型产生不同 的函数；编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对 参数替换后的代码进行编译。
2. 这是因为函数模板要被实例化后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的 头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

### 58、new和delete的实现原理， delete是如何知道释放内存的大小的额？

1、 new简单类型直接调用operator new分配内存；

而对于复杂结构，先调用operator new分配内存，然后在分配的内存上调用构造函数； 对于简单类型，new[]计算好大小后调用operator new；

对于复杂数据结构，new[]先调用operator new[]分配内存，然后在p的前四个字节写入数组大小n，然后调用n次构造函数，针对复杂类型，new[]会额外存储数组大小；

① new表达式调用一个名为operator new(operator new[])函数，分配一块足够大的、原始的、未命名的内存空间；

② 编译器运行相应的构造函数以构造这些对象，并为其传入初始值；

③ 对象被分配了空间并构造完成，返回一个指向该对象的指针。

2、 delete简单数据类型默认只是调用free函数；复杂数据类型先调用析构函数再调用operator delete；针对简单类型，delete和delete[]等同。假设指针p指向new[]分配的内存。因为要4字节存储数 组大小，实际分配的内存地址为[p-4]，系统记录的也是这个地址。delete[]实际释放的就是p-4指向的内 存。而delete会直接释放p指向的内存，这个内存根本没有被系统记录，所以会崩溃。

3、 需要在 new [] 一个对象数组时，需要保存数组的维度，C++ 的做法是在分配数组空间时多分配了 4 个字节的大小，专门保存数组的大小，在 delete [] 时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

### 59、malloc与free的实现原理？

1、 在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk、mmap、，

munmap这些系统调用实现的;

2、 brk是将数据段(.data)的最高地址指针\_edata往高地址推,mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。这两种方式分配的都是虚拟内存，没有分 配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内 存，然后建立虚拟内存和物理内存之间的映射关系；

3、 malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推；malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配；brk分配的内存需要等到高地址内存释放以后才 能释放，而mmap分配的内存可以单独释放。当最高地址空间的空闲内存超过128K（可由M\_TRIM\_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最 高地址空闲内存超过128K，于是内存紧缩。

4、 malloc是从堆里面申请内存，也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间 大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

### 60、类成员初始化方式？构造函数的执行顺序 ？为什么用成员初始化列表会快一些？

1. 赋值初始化，通过在函数体内进行赋值初始化；列表初始化，在冒号后使用初始化列表进行初始 化。

这两种方式的主要区别在于：

对于在函数体中初始化,是在所有的数据成员被分配内存空间后才进行的。

列表初始化是给数据成员分配内存空间时就进行初始化,就是说分配一个数据成员只要冒号后有此数据成 员的赋值表达式(此表达式必须是括号赋值表达式),那么分配了内存空间后在进入函数体之前给数据成员 赋值，就是说初始化这个数据成员此时函数体还未执行。

1. 一个派生类构造函数的执行顺序如下：

① 虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。

② 基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。

③ 类类型的成员对象的构造函数（按照初始化顺序）

④ 派生类自己的构造函数。

1. 方法一是在构造函数当中做赋值的操作，而方法二是做纯粹的初始化操作。我们都知道，C++的赋值 操作是会产生临时对象的。临时对象的出现会降低程序的效率。

### 62、什么是内存泄露，如何检测与避免

内存泄露

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)内存块，使用完后必须显式释放的内存。应用程序般使用malloc,、realloc、new等函数从堆中分配到块内存，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

避免内存泄露的几种方式

计数法：使用new或者malloc时，让该数+1，delete或free时，该数-1，程序执行完打印这个计 数，如果不为0则表示存在内存泄露

一定要将基类的析构函数声明为**虚函数**对象数组的释放一定要用**delete []**

有new就有delete，有malloc就有free，保证它们一定成对出现

检测工具

Linux下可以使用**Valgrind工具**Windows下可以使用**CRT库**

### 64、介绍面向对象的三大特性，并且举例说明

三大特性：继承、封装和多态

1. 继承

**让某种类型对象获得另一个类型对象的属性和方法。**

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展 常见的继承有三种方式：

* 1. 实现继承：指使用基类的属性和方法而无需额外编码的能力
  2. 接口继承：指仅使用属性和方法的名称、但是子类必须提供实现的能力
  3. 可视继承：指子窗体（类）使用基窗体（类）的外观和实现代码的能力（C++里好像不怎么用）

例如，将人定义为一个抽象类，拥有姓名、性别、年龄等公共属性，吃饭、睡觉、走路等公共方法，在 定义一个具体的人时，就可以继承这个抽象类，既保留了公共属性和方法，也可以在此基础上扩展跳 舞、唱歌等特有方法

1. 封装

数据和代码捆绑在一起，避免外界干扰和不确定性访问。

封装，也就是**把客观事物封装成抽象的类**，并且类可以把自己的数据和方法只让可信的类或者对象操 作，对不可信的进行信息隐藏，例如：将公共的数据或方法使用public修饰，而不希望被访问的数据或 方法采用private修饰。

1. 多态

同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为

（重载实现编译时多态，虚函数实现运行时多态）。

多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根 据当前赋值给它的子对象的特性以不同的方式运作。**简单一句话：允许将子类类型的指针赋值给父类类 型的指针**

实现多态有二种方式：覆盖（override），重载（overload）。覆盖：是指子类重新定义父类的虚函数 的做法。重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数 类型不同，或许两者都不同）。例如：基类是一个抽象对象——人，那教师、运动员也是人，而使用这 个抽象对象既可以表示教师、也可以表示运动员。

### 65、C++中类的数据成员和成员函数内存分布情况

C++类是由结构体发展得来的，所以他们的成员变量（C语言的结构体只有成员变量）的内存分配机制 是一样的。下面我们以类来说明问题，如果类的问题通了，结构体也也就没问题啦。 类分为成员变量和成员函数，我们先来讨论成员变量。

一个类对象的地址就是类所包含的这一片内存空间的首地址，这个首地址也就对应具体某一个成员变量 的地址。（在定义类对象的同时这些成员变量也就被定义了），举个例子：

#include <iostream>

using namespace *std*;

class Person

{

public:

Person()

{

this->age = 23;

}

void printAge()

{

*cout* << this->age << *endl*;

}

~Person() {}

public:

int age;

};

int main()

{

Person p;

*cout* << "对象地址：" << &p << *endl*;

*cout* << "age地址：" << &(p.age) << *endl*;

*cout* << "对象大小：" << sizeof(p) << *endl*;

*cout* << "age大小：" << sizeof(p.age) << *endl*;

return 0;

}

//输出结果

//对象地址：0x7fffec0f15a8

//age地址：0x7fffec0f15a8

//对象大小：4

//age大小：4

从代码运行结果来看，对象的大小和对象中数据成员的大小是一致的，也就是说，成员函数不占用对象 的内存。这是因为所有的函数都是存放在代码区的，不管是全局函数，还是成员函数。要是成员函数占 用类的对象空间，那么将是多么可怕的事情：定义一次类对象就有成员函数占用一段空间。 我们再来补充一下静态成员函数的存放问题：**静态成员函数与一般成员函数的唯一区别就是没有this指针**，因此不 能访问非静态数据成员，就像我前面提到的，**所有函数都存放在代码区，静态函数也不例外。所有有人 一看到 static 这个单词就主观的认为是存放在全局数据区，那是不对的。**

### 66、成员初始化列表的概念，为什么用它会快一些？

成员初始化列表的概念

在类的构造函数中，不在函数体内对成员变量赋值，而是在构造函数的花括号前面使用冒号和初始化列 表赋值

效率

用初始化列表会快一些的原因是，对于类型，它少了一次调用构造函数的过程，而在函数体中赋值则会 多一次调用。而对于内置数据类型则没有差别。举个例子：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3. class A   4 {  5 public: |

|  |
| --- |
| 6 A()  7 {  8 cout << "默认构造函数A()" << endl; 9 }  10 A(int a)  11 {   1. value = a; 2. cout << "A(int "<<value<<")" << endl; 14 }   15 A(const A& a)  16 {   1. value = a.value; 2. cout << "拷贝构造函数A(A& a): "<<value << endl; 19 }   20 int value; 21 };  22  23 class B  24 {  25 public:  26 B() : a(1)  27 {  28 b = A(2);  29 }   1. A a; 2. A b;   32 };  33 int main() 34 {  35 B b;  36 }  37  38 //输出结果： 39 //A(int 1)  40 //默认构造函数A() 41 //A(int 2)  42  43  44  45 |

从代码运行结果可以看出，在构造函数体内部初始化的对象b多了一次构造函数的调用过程，而对象a则 没有。由于对象成员变量的初始化动作发生在进入构造函数之前，对于内置类型没什么影响，但**如果有 些成员是类**，那么在进入构造函数之前，会先调用一次默认构造函数，进入构造函数后所做的事其实是 一次赋值操作(对象已存在)，所以**如果是在构造函数体内进行赋值的话，等于是一次默认构造加一次赋 值，而初始化列表只做一次赋值操作。**

[《为什么用成员初始化列表会快一些？》：https://blog.csdn.net/JackZhang\_123/article/detail s/82590368](https://blog.csdn.net/JackZhang_123/article/details/82590368)

### 67、(超重要)构造函数为什么不能为虚函数？析构函数为什么要虚函数？

**1、 从存储空间角度，**虚函数相应一个指向vtable虚函数表的指针，这大家都知道，但是这个指向

vtable的指针事实上是存储在对象的内存空间的。

问题出来了，假设构造函数是虚的，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，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、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。

**4、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。

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）。

因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual function主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象 还没有产生，如何使用virtual函数来完成你想完成的动作。

直接的讲，C++中基类采用virtual虚析构函数是**为了防止内存泄漏。**

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采 用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的 析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而 产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

### 68、析构函数的作用，如何起作用？

1. 构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传递参数，从主函数 传递到其他的函数里面，这样就使其他的函数里面有值了。

规则，只要你一实例化对象，系统自动回调用一个构造函数就是你不写，编译器也自动调用一次。

1. 析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务处理，可以是释放对象分配的内存 空间；特点：析构函数与构造函数同名，但该函数前面加~。

析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。 当撤销对象时，

编译器也会自动调用析构函数。

每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。 一般析构函数定义为类的公有成员。

### 69、构造函数和析构函数可以调用虚函数吗，为什么

1. 在C++中，提倡不在构造函数和析构函数中调用虚函数；
2. 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则 运行的是为构造函数或析构函数自身类型定义的版本；
3. 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数 时不安全的，故而C++不会进行动态联编；
4. 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析 构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函 数没有任何意义。

### 70、构造函数、析构函数的执行顺序？构造函数和拷贝构造的内部都干了啥？

1. 构造函数顺序

① 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

② 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序， 而不是它们出现在成员初始化表中的顺序。

③ 派生类构造函数。

1. 析构函数顺序

① 调用派生类的析构函数；

② 调用成员类对象的析构函数；

③ 调用基类的析构函数。

### 71、虚析构函数的作用，父类的析构函数是否要设置为虚函数？

1. C++中基类采用virtual虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只 会调用基类的析构函数，而不会调用派生类的析构函数。

那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

1. 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调 用其每一个虚基类以及上一层基类的析构函数。

因此，缺乏任何一个基类析构函数的定义，就会导致链接失败，最好不要把虚析构函数定义为纯虚析构 函数。

### 72、构造函数析构函数可否抛出异常

1. C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数 中发生异常，控制权转出构造函数之外。

因此，在对象b的构造函数中发生异常，对象b的析构函数不会被调用。因此会造成内存泄漏。

1. 用auto\_ptr对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危 机，不再需要在析构函数中手动释放资源；
2. 如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用

terminate函数让程序结束；

1. 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函 数执行不全，就是没有完成他应该执行的每一件事情。

### 73、构造函数一般不定义为虚函数的原因

1. 创建一个对象时需要确定对象的类型，而虚函数是在运行时动态确定其类型的。在构造一个对象 时，由于对象还未创建成功，编译器无法知道对象的实际类型
2. 虚函数的调用需要虚函数表指针vptr，而该指针存放在对象的内存空间中，若构造函数声明为虚 函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表vtable地址用来调用虚构造函数了
3. 虚函数的作用在于通过父类的指针或者引用调用它的时候能够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动调用的，不可能通过父类或者引用去调用，因此就规定构造函数不能是虚 函数
4. 析构函数一般都要声明为虚函数，这个应该是老生常谈了，这里不再赘述

《为什么C++不能有虚构造函数，却可以有虚析构函数》：<https://dwz.cn/lnfW9H6m>

### 74、类什么时候会析构？

1. 对象生命周期结束，被销毁时；
2. delete指向对象的指针时，或delete指向对象的基类类型指针，而其基类虚构函数是虚函数时；
3. 对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用。

### 75、构造函数或者析构函数中可以调用虚函数吗

简要结论：

从语法上讲，调用完全没有问题。

但是从效果上看，往往不能达到需要的目的。

《Eﬀective C++》的解释是：

派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。 同样，进入基类析构函数时，对象也是基类类型。

举个例子：

|  |
| --- |
| 4 #include<iostream>  using namespace std;  class Base 5 {   1. public: 2. Base()   8 {  9 Function(); 10 }  11  12 virtual void Function()  13 {  14 cout << "Base::Fuction" << endl; 15 }  16 ~Base()  17 {  18 Function(); 19 }  20 };  21  22 class A : public Base 23 {  24 public:  25 A()  26 {  27 Function(); 28 }  29  30 virtual void Function()  31 {  32 cout << "A::Function" << endl; 33 }  34 ~A()  35 {  36 Function(); 37 }  38 };  39  40 int main() 41 {   1. Base\* a = new Base; 2. delete a;   44 cout << "-------------------------" <<endl;   1. Base\* b = new A;//语句1 2. delete b; 47 } 3. //输出结果 4. //Base::Fuction 5. //Base::Fuction   51 //-------------------------   1. //Base::Fuction 2. //A::Function 3. //Base::Fuction 55   56  57  58 |

语句1讲道理应该体现多态性，执行类A中的构造和析构函数，从实验结果来看，语句1并没有体现，执 行流程是先构造基类，所以先调用基类的构造函数，构造完成再执行A自己的构造函数，析构时也是调 用基类的析构函数，也就是说构造和析构中调用虚函数并不能达到目的，应该避免

《构造函数或者析构函数中调用虚函数会怎么样？》：<https://dwz.cn/TaJTJONX>

### 76、智能指针的原理、常用的智能指针及实现

##### 原理

智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄 漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源

##### 常用的智能指针

1. **shared\_ptr**

实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象 时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会 减1，当计数为0的时候会自动的释放动态分配的资源。

智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针 每次创建类的新对象时，初始化指针并将引用计数置为1

当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数

对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0， 则删除对象），并增加右操作数所指对象的引用计数

调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

##### unique\_ptr

unique\_ptr采用的是独享所有权语义，一个非空的unique\_ptr总是拥有它所指向的资源。转移一个unique\_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique\_ptr不支持普通的 拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将 要被销毁）；如果你拷贝一个unique\_ptr，那么拷贝结束后，这两个unique\_ptr都会指向相同的资源， 造成在结束时对同一内存指针多次释放而导致程序崩溃。

##### weak\_ptr

weak\_ptr：弱引用。 引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak\_ptr打破环形引用。weak\_ptr是一个弱引用，它是为了配合shared\_ptr 而引入的一种智能指针，它指向一个由shared\_ptr管理的对象而不影响所指对象的生命周期，也就是 说，它只引用，不计数。如果一块内存被shared\_ptr和weak\_ptr同时引用，当所有shared\_ptr析构了之 后，不管还有没有weak\_ptr引用该内存，内存也会被释放。所以weak\_ptr不保证它指向的内存一定是 有效的，在使用之前使用函数lock()检查weak\_ptr是否为空指针。

##### auto\_ptr

主要是为了解决“有异常抛出时发生内存泄漏”的问题 。因为发生异常而无法正常释放内存。

auto\_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique\_ptr则无拷贝语 义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。

auto\_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。STL容器中的元素经常要支持拷贝、赋值操 作，在这过程中auto\_ptr会传递所有权，所以不能在STL中使用。

##### 智能指针shared\_ptr代码实现：

|  |
| --- |
| 1. template<typename T> 2. class SharedPtr 3 { 3. public: 4. SharedPtr(T\* ptr = NULL):\_ptr(ptr), \_pcount(new int(1)) 6 {}   7  8 SharedPtr(const SharedPtr& s):\_ptr(s.\_ptr), \_pcount(s.\_pcount){ 9 \*(\_pcount)++;  10 }  11   1. SharedPtr<T>& operator=(const SharedPtr& s){ 2. if (this != &s)   14 {  15 if (--(\*(this->\_pcount)) == 0)  16 {   1. delete this->\_ptr; 2. delete this->\_pcount; 19 } 3. \_ptr = s.\_ptr; 4. \_pcount = s.\_pcount;   22 \*(\_pcount)++;  23 }  24 return \*this; 25 }  26 T& operator\*()  27 {  28 return \*(this->\_ptr); 29 }  30 T\* operator->()  31 {  32 return this->\_ptr; 33 }  34 ~SharedPtr()  35 {  36 --(\*(this->\_pcount));  37 if (this->\_pcount == 0)  38 {   1. delete \_ptr; 2. \_ptr = NULL; 3. delete \_pcount; 4. \_pcount = NULL; 43 }   44 }   1. private: 2. T\* \_ptr; 3. int\* \_pcount;//指向引用计数的指针48 };   49  50  51  52  53 |

《智能指针的原理及实现》：<https://blog.csdn.net/lizhentao0707/article/details/81156384>

第 88／297页

### 77、构造函数的几种关键字

##### default

default关键字可以显式要求编译器生成合成构造函数，防止在调用时相关构造函数类型没有定义而报错

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3   4 class CString 5 {   1. public: 2. CString() = default; //语句1 3. //构造函数 4. CString(const char\* pstr) : \_str(pstr){} 5. void\* operator new() = delete;//这样不允许使用new关键字 6. //析构函数 7. ~CString(){} 8. public: 9. string \_str; 15 };   16  17  18 int main() 19 {   1. auto a = new CString(); //语句2 2. cout << "Hello World" <<endl; 3. return 0; 23 } 4. //运行结果 5. //Hello World 26   27  28  29 |

如果没有加语句1，语句2会报错，表示找不到参数为空的构造函数，将其设置为default可以解决这个 问题

##### delete

delete关键字可以删除构造函数、赋值运算符函数等，这样在使用的时候会得到友善的提示

#include <iostream>

using namespace std;

class CString

{

public:

void\* operator new() = delete;//这样不允许使用new关键字

//析构函数

~CString(){}

};

int main()

{

auto a = new CString(); //语句1

cout << "Hello World" <<endl;

return 0;

}

在执行语句1时，会提示new方法已经被删除，如果将new设置为私有方法，则会报惨不忍睹的错误， 因此使用delete关键字可以更加人性化的删除一些默认方法

##### 0

将虚函数定义为纯虚函数（纯虚函数无需定义，= 0只能出现在类内部虚函数的声明语句处；当然，也可以为纯虚函数提供定义，不过函数体必须定义在类的外部）

[《C++构造函数的default和delete》：https://blog.csdn.net/u010591680/article/details/71101 737](https://blog.csdn.net/u010591680/article/details/71101737)

### 78、C++的四种强制转换reinterpret\_cast/const\_cast/static\_cast

**/dynamic\_cast**

##### reinterpret\_cast

reinterpret\_cast (expression)

type-id 必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以用于类型之间进行强制转换。

##### const\_cast

const\_cast<type\_id> (expression)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression

的类型是一样的。用法如下：

常量指针被转化成非常量的指针，并且仍然指向原来的对象常量引用被转换成非常量的引用，并且仍然指向原来的对象const\_cast一般用于修改底指针。如const char \*p形式

##### static\_cast

static\_cast < type-id > (expression)

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。它主要有如下 几种用法：

用于类层次结构中基类（父类）和派生类（子类）之间指针或引用引用的转换进行上行转换（把派生类的指针或引用转换成基类表示）是安全的

进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是 不安全的

用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要 开发人员来保证。

把空指针转换成目标类型的空指针把任何类型的表达式转换成void类型

注意：static\_cast不能转换掉expression的const、volatile、或者 unaligned属性。

##### dynamic\_cast

有类型检查，基类向派生类转换比较安全，但是派生类向基类转换则不太安全

dynamic\_cast (expression)

该运算符把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void\* 如果 type-id 是类指针类型，那么expression也必须是一个指针，如果 type-id 是一个引用，那么

expression 也必须是一个引用

dynamic\_cast运算符可以在执行期决定真正的类型，也就是说expression必须是多态类型。如果下行转 换是安全的（也就说，如果基类指针或者引用确实指向一个派生类对象）这个运算符会传回适当转型过 的指针。如果 如果下行转换不安全，这个运算符会传回空指针（也就是说，基类指针或者引用没有指向一个派生类对象）

dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换 在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的

在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全举个例子：

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  class Base  {  public:  Base() :b(1) {}  virtual void fun() {};  int b;  };  class Son : public Base  {  public:  Son() :d(2) {}  int d;  };  int main()  {  int n = 97;  //reinterpret\_cast  int\* p = &n;  //以下两者效果相同  char\* c = reinterpret\_cast<char\*> (p);  char\* c2 = (char\*)(p);  cout << "reinterpret\_cast输出：" << \*c2 << *endl*;  //const\_cast  const int\* p2 = &n;  int\* p3 = const\_cast<int\*>(p2);  \*p3 = 100;  cout << "const\_cast输出：" << \*p3 << *endl*;  Base\* b1 = new Son;  Base\* b2 = new Base;  //static\_cast  Son\* s1 = static\_cast<Son\*>(b1); //同类型转换  Son\* s2 = static\_cast<Son\*>(b2); //下行转换，不安全  cout << "static\_cast输出：" << *endl*;  cout << s1->d << *endl*;  cout << s2->d << *endl*; //下行转换，原先父对象没有d成员，输出垃圾值  //dynamic\_cast  Son\* s3 = dynamic\_cast<Son\*>(b1); //同类型转换  Son\* s4 = dynamic\_cast<Son\*>(b2); //下行转换，安全  cout << "dynamic\_cast输出：" << *endl*;  cout << s3->d << *endl*;  if (s4 == nullptr)  cout << "s4指针为nullptr" << *endl*;  else  cout << s4->d << *endl*;  return 0;  }  //输出结果  //reinterpret\_cast输出：a  //const\_cast输出：100  //static\_cast输出：  //2  //-33686019  //dynamic\_cast输出：  //2  //s4指针为nullptr |

从输出结果可以看出，在进行下行转换时，dynamic\_cast安全的，如果下行转换不安全的话其会返回空 指针，这样在进行操作的时候可以预先判断。而使用static\_cast下行转换存在不安全的情况也可以转换 成功，但是直接使用转换后的对象进行操作容易造成错误。

### 79、C++函数调用的压栈过程

从代码入手，解释这个过程：

#include <iostream>

using namespace std;

int f(int n)

{

cout << n << *endl*;

return n;

}

void func(int param1, int param2)

{

int var1 = param1;

int var2 = param2;

printf("var1=%d,var2=%d", f(var1), f(var2));//如果将printf换为cout进行输

出，输出结果则刚好相反

}

int main(int argc, char\* argv[])

{

func(1, 2);

return 0;

}

//输出结果

//2

//1

//var1=1,var2=2

当函数从入口函数main函数开始执行时，编译器会将我们操作系统的运行状态，main函数的返回地 址、main的参数、mian函数中的变量、进行依次压栈；

当main函数开始调用func()函数时，编译器此时会将main函数的运行状态进行压栈，再将func()函数的 返回地址、func()函数的参数从右到左、func()定义变量依次压栈；

当func()调用f()的时候，编译器此时会将func()函数的运行状态进行压栈，再将的返回地址、f()函数的参 数从右到左、f()定义变量依次压栈

从代码的输出结果可以看出，函数f(var1)、f(var2)依次入栈，而后先执行f(var2)，再执行f(var1)，最后 打印整个字符串，将栈中的变量依次弹出，最后主函数返回。

[《C/C++函数调用过程分析》：https://www.cnblogs.com/biyeymyhjob/archive/2012/07/20/2 601204.html](https://www.cnblogs.com/biyeymyhjob/archive/2012/07/20/2601204.html)

[《C/C++函数调用的压栈模型》：https://blog.csdn.net/m0\_37717595/article/details/8036841 1](https://blog.csdn.net/m0_37717595/article/details/80368411)

### 81、C++中将临时变量作为返回值时的处理过程

首先需要明白一件事情，临时变量，在函数调用过程中是被压到程序进程的栈中的，当函数退出时，临 时变量出栈，即临时变量已经被销毁，临时变量占用的内存空间没有被清空，但是可以被分配给其他变 量，所以有可能在函数退出时，该内存已经被修改了，对于临时变量来说已经是没有意义的值了

C语言里规定：16bit程序中，返回值保存在ax寄存器中，32bit程序中，返回值保持在eax寄存器中，如 果是64bit返回值，edx寄存器保存高32bit，eax寄存器保存低32bit

由此可见，函数调用结束后，返回值被临时存储到寄存器中，并没有放到堆或栈中，也就是说与内存没 有关系了。当退出函数的时候，临时变量可能被销毁，但是返回值却被放到寄存器中与临时变量的生命 周期没有关系

如果我们需要返回值，一般使用赋值语句就可以了

《【C++】临时变量不能作为函数的返回值？》：<https://www.wandouip.com/t5i204349/>

（栈上的内存分配、拷贝过程）

### 82、关于this指针你知道什么？全说出来

this指针是类的指针，指向对象的首地址。

this指针只能在成员函数中使用，在全局函数、静态成员函数中都不能用this。

this指针只有在成员函数中才有定义，且存储位置会因编译器不同有不同存储位置。

##### this指针的用处

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。this作用域是在类内部， 当在类的**非静态成员函数**中访问类的**非静态成员**的时候（全局函数，静态函数中不能使用this指针）， 编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针， 编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this 进行

##### this指针的使用

一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return \*this； 另外一种情况是当形参数与成员变量名相同时用于区分，如this->n = n （不能写成n = n） **类的this指针有以下特点**

(1）**this**只能在成员函数中使用，全局函数、静态函数都不能使用this。实际上，**成员函数默认第一个 参数**为**T \* const this**

如：

|  |
| --- |
| 1. class A{ 2. public: 3. int func(int p){} 4 };   5  6  7  8 |

其中，**func**的原型在编译器看来应该是：

##### int func(A \* const this,int p);

（2）由此可见，**this**在成员函数的开始前构造，在成员函数的结束后清除。这个生命周期同任何一个 函数的参数是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的this 参数传递进去。如：

|  |
| --- |
| 1 A a;  2 a.func(10);  3 //此处，编译器将会编译成： 4 A::func(&a,10);  5  6  7  8 |

看起来和静态函数没差别，对吗？不过，区别还是有的。编译器通常会对this指针做一些优化，因此，

this指针的传递效率比较高，例如VC通常是通过ecx（计数寄存器）传递this参数的。

### 83、几个this指针的易混问题

##### this指针是什么时候创建的？

this在成员函数的开始执行前构造，在成员的执行结束后清除。

但是如果class或者struct里面没有方法的话，它们是没有构造函数的，只能当做C的struct使用。采用TYPE xx的方式定义的话，在栈里分配内存，这时候this指针的值就是这块内存的地址。采用new的方式创建对象的话，在堆里分配内存，new操作符通过eax（累加寄存器）返回分配的地址，然后设置给指 针变量。之后去调用构造函数（如果有构造函数的话），这时将这个内存块的地址传给ecx，之后构造 函数里面怎么处理请看上面的回答

##### this指针存放在何处？堆、栈、全局变量，还是其他？

this指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器，甚至全局变量。在汇编级 别里面，一个值只会以3种形式出现：立即数、寄存器值和内存变量值。不是存放在寄存器就是存放在 内存中，它们并不是和高级语言变量对应的。

##### this指针是如何传递类中的函数的？绑定？还是在函数参数的首参数就是this指针？那么，this指针 又是如何找到“类实例后函数的”？

大多数编译器通过ecx（寄数寄存器）寄存器传递this指针。事实上，这也是一个潜规则。一般来说，不 同编译器都会遵从一致的传参规则，否则不同编译器产生的obj就无法匹配了。

在call之前，编译器会把对应的对象地址放到eax中。this是通过函数参数的首参来传递的。this指针在 调用之前生成，至于“类实例后函数”，没有这个说法。类在实例化时，只分配类中的变量空间，并没有 为函数分配空间。自从类的函数定义完成后，它就在那儿，不会跑的

##### this指针是如何访问类中的变量的？

如果不是类，而是结构体的话，那么，如何通过结构指针来访问结构中的变量呢？如果你明白这一点的 话，就很容易理解这个问题了。

在C++中，类和结构是只有一个区别的：类的成员默认是private，而结构是public。

this是类的指针，如果换成结构体，那this就是结构的指针了。

##### 我们只有获得一个对象后，才能通过对象使用this指针。如果我们知道一个对象this指针的位置，可 以直接使用吗？

**this指针只有在成员函数中才有定义。**因此，你获得一个对象后，也不能通过对象使用this指针。所 以，我们无法知道一个对象的this指针的位置（只有在成员函数里才有this指针的位置）。当然，在成 员函数里，你是可以知道this指针的位置的（可以通过&this获得），也可以直接使用它。

##### 每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数？

普通的类函数（不论是成员函数，还是静态函数）都不会创建一个函数表来保存函数指针。只有虚函数 才会被放到函数表中。但是，即使是虚函数，如果编译期就能明确知道调用的是哪个函数，编译器就不 会通过函数表中的指针来间接调用，而是会直接调用该函数。正是由于this指针的存在，用来指向不同 的对象，从而确保不同对象之间调用相同的函数可以互不干扰

《C++中this指针的用法详解》<http://blog.chinaunix.net/uid-21411227-id-1826942.html>

### 84、构造函数、拷贝构造函数和赋值操作符的区别

##### 构造函数

对象不存在，没用别的对象初始化，在创建一个新的对象时调用构造函数

##### 拷贝构造函数

对象不存在，但是使用别的已经存在的对象来进行初始化

##### 赋值运算符

对象存在，用别的对象给它赋值，这属于重载“=”号运算符的范畴，“=”号两侧的对象都是已存在的 举个例子：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3   4 class A  5 {  6 public:  7 A()  8 {  9 cout << "我是构造函数" << endl; 10 }  11 A(const A& a)  12 {  13 cout << "我是拷贝构造函数" << endl; 14 }  15 A& operator = (A& a)  16 {   1. cout << "我是赋值操作符" << endl; 2. return \*this; 19 }   20 ~A() {};  21 };  22  23 int main() 24 {   1. A a1; //调用构造函数 2. A a2 = a1; //调用拷贝构造函数 3. a2 = a1; //调用赋值操作符 4. return 0; 29 }   第 96／297页 |

|  |
| --- |
| 1. //输出结果 2. //我是构造函数 3. //我是拷贝构造函数 4. //我是赋值操作符34   35  36  37 |

### **拷贝构造函数和赋值运算符重载的区别？**

拷贝构造函数是函数，赋值运算符是运算符重载。

拷贝构造函数会生成新的类对象，赋值运算符不能。

拷贝构造函数是直接构造一个新的类对象，所以在初始化对象前不需要检查源对象和新建对象是否 相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有 内存分配则需要先把内存释放掉。

形参传递是调用拷贝构造函数（调用的被赋值对象的拷贝构造函数），但并不是所有出现"="的地 方都是使用赋值运算符，如下：

// 赋值运算符操作

s2 = s;

// 调用拷贝构造函数

Student s; Student s1 = s;

Student s2;

1

2

3

4

注：类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符

### 86、智能指针的作用；

1. C++11中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释 放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。
2. 智能指针在C++11版本之后提供，包含在头文件中，shared\_ptr、unique\_ptr、weak\_ptr。shared\_ptr多个指针指向相同的对象。shared\_ptr使用引用计数，每一个shared\_ptr的拷贝都指向相同 的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，自动删除 所指向的堆内存。shared\_ptr内部的引用计数是线程安全的，但是对象的读取需要加锁。
3. 初始化。智能指针是个模板类，可以指定类型，传入指针通过构造函数初始化。也可以使用make\_shared函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如std::shared\_ptr p4 = new int(1);的写法是错误的

拷贝和赋值。拷贝使得对象的引用计数增加1，赋值使得原对象引用计数减1，当计数为0时，自动释放 内存。后来指向的对象引用计数加1，指向后来的对象

1. unique\_ptr“唯一”拥有其所指对象，同一时刻只能有一个unique\_ptr指向给定对象（通过禁止拷贝语 义、只有移动语义来实现）。相比与原始指针unique\_ptr用于其RAII的特性，使得在出现异常的情况

下，动态资源能得到释放。unique\_ptr指针本身的生命周期：从unique\_ptr指针创建时开始，直到离开 作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用delete操作符，用户可指定其他 操作)。unique\_ptr指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象， 如创建智能指针时通过构造函数指定、通过reset方法重新指定、通过release方法释放所有权、通过移 动语义转移所有权。

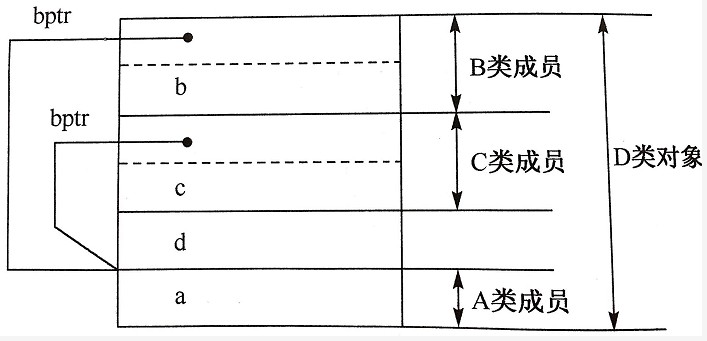
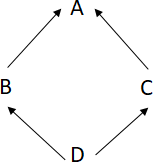
1. 智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。 每次创建类的新对象时，初始化指针并将引用计数置为1；当对象作为另一对象的副本而创建时，拷贝 构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指 对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数；调用 析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）。
2. weak\_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 shared\_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared\_ptr. weak\_ptr只是提供了对管理对象的一个访问手段。weak\_ptr 设计的目的是为配合 shared\_ptr 而引入的一种智能指针来协助 shared\_ptr 工作, 它只可以从一个shared\_ptr 或另一个 weak\_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少.

### 89、什么是虚拟继承

由于C++支持多继承，除了public、protected和private三种继承方式外，还支持虚拟（virtual）继承， 举个例子：

|  |
| --- |
| 1. #include <iostream> 2. using namespace std; 3 3. class A{} 4. class B : virtual public A{}; 5. class C : virtual public A{}; 6. class D : public B, public C{}; 8   9 int main() 10 {   1. cout << "sizeof(A)：" << sizeof A <<endl; // 1，空对象，只有一个占位 2. cout << "sizeof(B)：" << sizeof B <<endl; // 4，一个bptr指针，省去占位,不需要对齐 3. cout << "sizeof(C)：" << sizeof C <<endl; // 4，一个bptr指针，省去占位,不需要对齐 4. cout << "sizeof(D)：" << sizeof D <<endl; // 8，两个bptr，省去占位,不需要对齐   15 }  16  17  18 |

上述代码所体现的关系是，B和C虚拟继承A，D又公有继承B和C，这种方式是一种**菱形继承或者钻石继 承**，可以用如下图来表示



**虚拟继承的情况下，无论基类被继承多少次，只会存在一个实体。**虚拟继承基类的子类中，子类会增加 某种形式的指针，或者指向虚基类子对象，或者指向一个相关的表格；表格中存放的不是虚基类子对象 的地址，就是其偏移量，此类指针被称为bptr，如上图所示。如果既存在vptr又存在bptr，某些编译器 会将其优化，合并为一个指针

### 91、静态类型和动态类型，静态绑定和动态绑定的介绍

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；

从上面的定义也可以看出，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）。

举个例子：

#include <iostream>

using namespace std;

class A {

public:

/\*virtual\*/ void func() { std::*cout* << "A::func()\n"; }

};

class B : public A {

public:

void func() { std::*cout* << "B::func()\n"; }

};

class C : public A {

public:

void func() { std::*cout* << "C::func()\n"; }

};

int main()

{

C\* pc = new C(); //pc的静态类型是它声明的类型C\*，动态类型也是C\*；

B\* pb = new B(); //pb的静态类型和动态类型也都是B\*；

A\* pa = pc;      //pa的静态类型是它声明的类型A\*，动态类型是pa所指向的对象pc的类型

C\* ；

pa = pb;         //pa的动态类型可以更改，现在它的动态类型是B\*，但其静态类型仍是声明时候的A\* ；

C\* pnull = *NULL*; //pnull的静态类型是它声明的类型C\*,没有动态类型，因为它指向了NULL；

pa->func();      //A::func() pa的静态类型永远都是A\*，不管其指向的是哪个子类，都是直接调用A::func()；

pc->func();      //C::func() pc的动、静态类型都是C\*，因此调用C::func()；

pnull->func();   //C::func() 不用奇怪为什么空指针也可以调用函数，因为这在编译期就确定了，和指针空不空没关系；

return 0;

}

如果将A类中的virtual注释去掉，则运行结果是：

|  |
| --- |
| 1. pa->func(); //B::func() 因为有了virtual虚函数特性，pa的动态类型指向B\*，因此先在   B中查找，找到后直接调用；   1. pc->func(); //C::func() pc的动、静态类型都是C\*，因此也是先在C中查找； 2. pnull->func(); //空指针异常，因为是func是virtual函数，因此对func的调用只能等到运行 期才能确定，然后才发现pnull是空指针；   4  5  6 |

在上面的例子中，

如果基类A中的func不是virtual函数，那么不论pa、pb、pc指向哪个子类对象，对func的调用都 是在定义pa、pb、pc时的静态类型决定，早已在编译期确定了。

同样的空指针也能够直接调用no-virtual函数而不报错（这也说明一定要做空指针检查啊！），因 此静态绑定不能实现多态；

如果func是虚函数，那所有的调用都要等到运行时根据其指向对象的类型才能确定，比起静态绑 定自然是要有性能损失的，但是却能实现多态特性；

本文代码里都是针对指针的情况来分析的，但是对于引用的情况同样适用。

至此总结一下静态绑定和动态绑定的区别：

静态绑定发生在编译期，动态绑定发生在运行期； 对象的动态类型可以更改，但是静态类型无法更改； 要想实现动态，必须使用动态绑定；

在继承体系中只有虚函数使用的是动态绑定，其他的全部是静态绑定；

建议：

绝对不要重新定义继承而来的非虚(non-virtual)函数（《Eﬀective C++ 第三版》条款36），因为这样导致函数调用由对象声明时的静态类型确定了，而和对象本身脱离了关系，没有多态，也这将给程序留下 不可预知的隐患和莫名其妙的BUG；另外，在动态绑定也即在virtual函数中，要注意默认参数的使用。 当缺省参数和virtual函数一起使用的时候一定要谨慎，不然出了问题怕是很难排查。

看下面的代码：

#include <iostream>

using namespace std;

class E

{

public:

virtual void func(int i = 0)

{

std::*cout* << "E::func()\t" << i << "\n";

}

};

class F : public E

{

public:

virtual void func(int i = 1)

{

std::*cout* << "F::func()\t" << i << "\n";

}

};

void test2()

{

F\* pf = new F();

E\* pe = pf;

pf->func(); //F::func() 1 正常，就该如此；

pe->func(); //F::func() 0 哇哦，这是什么情况，调用了子类的函数，却使用了基类中参 数的默认值！

}

int main()

{

test2();

return 0;

}

《C++中的静态绑定和动态绑定》：<https://www.cnblogs.com/lizhenghn/p/3657717.html>

### 93、引用是否能实现动态绑定，为什么可以实现？

可以。

引用在创建的时候必须初始化，在访问虚函数时，编译器会根据其所绑定的对象类型决定要调用哪个函 数。注意只能调用虚函数。

#include <iostream>

using namespace std;

class Base

{

public:

virtual void  fun()

{

cout << "base :: fun()" << *endl*;

}

};

class Son : public Base

{

public:

virtual void  fun()

{

cout << "son :: fun()" << *endl*;

}

void func()

{

cout << "son :: not virtual function" << *endl*;

}

};

int main()

{

Son s;

Base& b = s; // 基类类型引用绑定已经存在的Son对象，引用必须初始化

s.*fun*(); //son::fun()

b.*fun*(); //son :: fun()

return 0;

}

**需要说明的是虚函数才具有动态绑定**，上面代码中，Son类中还有一个非虚函数func()，这在b对象中是 无法调用的，如果使用基类指针来指向子类也是一样的。

### 94、全局变量和局部变量有什么区别？

生命周期不同：全局变量随主程序创建和创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至 局部循环体等内部存在，退出就不存在；

使用方式不同：通过声明后全局变量在程序的各个部分都可以用到；局部变量分配在堆栈区，只能在局 部使用。

操作系统和编译器通过内存分配的位置可以区分两者，全局变量分配在全局数据段并且在程序开始运行 的时候被加载。局部变量则分配在堆栈里面 。

《C++经典面试题》：<https://www.cnblogs.com/yjd_hycf_space/p/7495640.html>

### 96、 怎样判断两个浮点数是否相等？

对两个浮点数判断大小和是否相等不能直接用==来判断，会出错！明明相等的两个数比较反而是不相 等！对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值！浮点数与0的比较 也应该注意。与浮点数的表示方式有关。

### 97、方法调用的原理（栈，汇编）

1. 机器用栈来传递过程参数、存储返回信息、保存寄存器用于以后恢复，以及本地存储。而为单个过 程分配的那部分栈称为帧栈；帧栈可以认为是程序栈的一段，它有两个端点，一个标识起始地址，一个 标识着结束地址，两个指针结束地址指针esp，开始地址指针ebp;
2. 由一系列栈帧构成，这些栈帧对应一个过程，而且每一个栈指针+4的位置存储函数返回地址；每一 个栈帧都建立在调用者的下方，当被调用者执行完毕时，这一段栈帧会被释放。由于栈帧是向地址递减 的方向延伸，因此如果我们将栈指针减去一定的值，就相当于给栈帧分配了一定空间的内存。如果将栈 指针加上一定的值，也就是向上移动，那么就相当于压缩了栈帧的长度，也就是说内存被释放了。
3. 过程实现

① 备份原来的帧指针，调整当前的栈帧指针到栈指针位置；

② 建立起来的栈帧就是为被调用者准备的，当被调用者使用栈帧时，需要给临时变量分配预留内存；

③ 使用建立好的栈帧，比如读取和写入，一般使用mov，push以及pop指令等等。

④ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存器，不过此时这些值可能已经不在栈顶了

⑤ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存器，不过此时这些值可能已经不在栈顶了。

⑥ 释放被调用者的栈帧，释放就意味着将栈指针加大，而具体的做法一般是直接将栈指针指向帧指针，因此会采用类似下面的汇编代码处理。

⑦ 恢复调用者的栈帧，恢复其实就是调整栈帧两端，使得当前栈帧的区域又回到了原始的位置。

⑧ 弹出返回地址，跳出当前过程，继续执行调用者的代码。

1. 过程调用和返回指令

① call指令

② leave指令

③ ret指令

### 98、C++中的指针参数传递和引用参数传递有什么区别？底层原理你知道吗？

1. 指针参数传递本质上是值传递，它所传递的是一个地址值。

值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主 调函数传递进来的实参值，从而形成了实参的一个副本（替身）。

值传递的特点是，被调函数对形式参数的任何操作都是作为局部变量进行的，不会影响主调函数的实参 变量的值（形参指针变了，实参指针不会变）。

1. 引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的 是由主调函数放进来的实参变量的地址。

被调函数对形参（本体）的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实 参变量（根据别名找到主调函数中的本体）。

因此，被调函数对形参的任何操作都会影响主调函数中的实参变量。

1. 引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引 用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。

而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想 通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用。

1. 从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变 量所对应地址。

指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的 地址值（与实参名字不同，地址相同）。

符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象 则不能修改。

### 99、类如何实现只能静态分配和只能动态分配

1. 前者是把new、delete运算符重载为private属性。后者是把构造、析构函数设为protected属性，再 用子类来动态创建
2. 建立类的对象有两种方式：

① 静态建立，静态建立一个类对象，就是由编译器为对象在栈空间中分配内存；

② 动态建立，A \*p = new A();动态建立一个类对象，就是使用new运算符为对象在堆空间中分配内

存。这个过程分为两步，第一步执行operator new()函数，在堆中搜索一块内存并进行分配；第二步调用类构造函数构造对象；

1. 只有使用new运算符，对象才会被建立在堆上，因此只要限制new运算符就可以实现类对象只能建立 在栈上，可以将new运算符设为私有。

### 100、什么情况会自动生成默认构造函数？

1. 带有默认构造函数的类成员对象，如果一个类没有任何构造函数，但它含有一个成员对象，而后者有 默认构造函数，那么编译器就为该类合成出一个默认构造函数。

不过这个合成操作只有在构造函数真正被需要的时候才会发生；

如果一个类A含有多个成员类对象的话，那么类A的每一个构造函数必须调用每一个成员对象的默认构造 函数而且必须按照类对象在类A中的声明顺序进行；

1. 带有默认构造函数的基类，如果一个没有任务构造函数的派生类派生自一个带有默认构造函数基类， 那么该派生类会合成一个构造函数调用上一层基类的默认构造函数；
2. 带有一个虚函数的类
3. 带有一个虚基类的类
4. 合成的默认构造函数中，只有基类子对象和成员类对象会被初始化。所有其他的非静态数据成员都不 会被初始化。

### 101、抽象基类为什么不能创建对象？

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

1. 抽象类的定义：

称带有纯虚函数的类为抽象类。

1. 抽象类的作用：

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

1. 使用抽象类时注意：

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

抽象类是不能定义对象的。一个纯虚函数不需要（但是可以）被定义。

一、纯虚函数定义

纯虚函数是一种特殊的虚函数，它的一般格式如下：

|  |
| --- |
| 1 class <类名> 2 {  3 virtual <类型><函数名>(<参数表>)=0; 4 …  5 };  6 |

在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给 该基类的派生类去做。这就是纯虚函数的作用。

纯虚函数可以让类先具有一个操作名称，而没有操作内容，让派生类在继承时再去具体地给出定义。 凡是含有纯虚函数的类叫做抽象类。这种类不能声明对象，只是作为基类为派生类服务。除非在派生类 中完全实现基类中所有的的纯虚函数，否则，派生类也变成了抽象类，不能实例化对象。

二、纯虚函数引入原因

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

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

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;）。若要使派生类为非抽象类，则编译器要求在派生类中，必须对纯虚函数予以重载以实现多态性。同时含有纯虚函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。 例如，绘画程序中，shape作为一个基类可以派生出圆形、矩形、正方形、梯形等， 如果我要求面积总和的话，那么会可以使用一个 shape \* 的数组，只要依次调用派生类的area()函数了。如果不用接口就没法定义成数组，因为既可以是circle ,也可以是square ,而且以后还可能加上rectangle，等等.

三、相似概念

1、多态性

指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译 时多态性，运行时多态性。

a.编译时多态性：通过重载函数实现b.运行时多态性：通过虚函数实现。

2、虚函数

虚函数是在基类中被声明为virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重 载。

3、抽象类

包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对 象。

### 102、 继承机制中对象之间如何转换？指针和引用之间如何转换？

1. 向上类型转换

将派生类指针或引用转换为基类的指针或引用被称为向上类型转换，向上类型转换会自动进行，而且向 上类型转换是安全的。

1. 向下类型转换

将基类指针或引用转换为派生类指针或引用被称为向下类型转换，向下类型转换不会自动进行，因为一 个基类对应几个派生类，所以向下类型转换时不知道对应哪个派生类，所以在向下类型转换时必须加动 态类型识别技术。RTTI技术，用dynamic\_cast进行向下类型转换。

### 103、函数指针？

1. 什么是函数指针?

函数指针指向的是特殊的数据类型，函数的类型是由其返回的数据类型和其参数列表共同决定的，而函 数的名称则不是其类型的一部分。

一个具体函数的名字，如果后面不跟调用符号(即括号)，则该名字就是该函数的指针(注意：大部分情况 下，可以这么认为，但这种说法并不很严格)。

1. 函数指针的声明方法

int (\*pf)(const int&, const int&); (1)

上面的pf就是一个函数指针，指向所有返回类型为int，并带有两个const int&参数的函数。注意\*pf两边的括号是必须的，否则上面的定义就变成了：

int \*pf(const int&, const int&); (2)

而这声明了一个函数pf，其返回类型为int \*， 带有两个const int&参数。

1. 为什么有函数指针

函数与数据项相似，函数也有地址。我们希望在同一个函数中通过使用相同的形参在不同的时间使用产 生不同的效果。

1. 一个函数名就是一个指针，它指向函数的代码。一个函数地址是该函数的进入点，也就是调用函数 的地址。函数的调用可以通过函数名，也可以通过指向函数的指针来调用。函数指针还允许将函数作为 变元传递给其他函数；
2. 两种方法赋值：

指针名 = 函数名； 指针名 = &函数名

### 104、 内存泄漏的后果？如何监测？解决方法？

1. 内存泄漏

内存泄漏是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物 理上消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制；

1. 后果

只发生一次小的内存泄漏可能不被注意，但泄漏大量内存的程序将会出现各种证照：性能下降到内存逐 渐用完，导致另一个程序失败；

1. 如何排除

使用工具软件BoundsChecker，BoundsChecker是一个运行时错误检测工具，它主要定位程序运行时 期发生的各种错误；

调试运行DEBUG版程序，运用以下技术：CRT(C run-time libraries)、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境OUTPUT窗口)，综合分析内存泄漏的原因，排除内存泄漏。

1. 解决方法智能指针。
2. 检查、定位内存泄漏

检查方法：在main函数最后面一行，加上一句\_CrtDumpMemoryLeaks()。调试程序，自然关闭程序让 其退出，查看输出：

输出这样的格式{453}normal block at 0x02432CA8,868 bytes long

被{}包围的453就是我们需要的内存泄漏定位值，868 bytes long就是说这个地方有868比特内存没有释放。

定位代码位置

在main函数第一行加上\_CrtSetBreakAlloc(453);意思就是在申请453这块内存的位置中断。然后调试程 序，程序中断了，查看调用堆栈。加上头文件#include <crtdbg.h>

<https://blog.csdn.net/u012680662/article/details/136187474>

### 105、使用智能指针管理内存资源，RAII是怎么回事？

1. RAII全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。

因为C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会 自动调用析构函数。所以，在RAII的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑 定。

1. 智能指针（std::shared\_ptr和std::unique\_ptr）即RAII最具代表的实现，使用智能指针，可以实现自 动的内存管理，再也不需要担心忘记delete造成的内存泄漏。

毫不夸张的来讲，有了智能指针，代码中几乎不需要再出现delete了。

### 106、说一说你理解的内存对齐以及原因

1、 分配内存的顺序是按照声明的顺序。

2、 每个变量相对于起始位置的偏移量必须是该变量类型大小的整数倍，不是整数倍空出内存，直到偏移量是整数倍为止。

3、 最后整个结构体的大小必须是里面变量类型最大值的整数倍。

添加了#pragma pack(n)后规则就变成了下面这样： 1、 偏移量要是n和当前变量大小中较小值的整数倍

2、 整体大小要是n和最大变量大小中较小值的整数倍

3、 n值必须为1,2,4,8…，为其他值时就按照默认的分配规则

### 107、 结构体变量比较是否相等

1. 重载了 “==” 操作符

|  |
| --- |
| 1 struct foo { 2  3 int a; 4  5 int b; 6  7 bool operator==(const foo& rhs) \*//\* \*操作运算符重载\*  8  9 {  10  11 return( a == rhs.a) && (b == rhs.b); 12  13 }  14  15 };  16  17 |

1. 元素的话，一个个比；
2. 指针直接比较，如果保存的是同一个实例地址，则(p1==p2)为真；

### 108、 函数调用过程栈的变化，返回值和参数变量哪个先入栈？

1、调用者函数把被调函数所需要的参数按照与被调函数的形参顺序相反的顺序压入栈中,即:从右向左 依次把被调函数所需要的参数压入栈;

2、调用者函数使用call指令调用被调函数,并把call指令的下一条指令的地址当成返回地址压入栈中(这 个压栈操作隐含在call指令中);

3、在被调函数中,被调函数会先保存调用者函数的栈底地址(push ebp),然后再保存调用者函数的栈顶地址,即:当前被调函数的栈底地址(mov ebp,esp);

4、在被调函数中,从ebp的位置处开始存放被调函数中的局部变量和临时变量,并且这些变量的地址按照 定义时的顺序依次减小,即:这些变量的地址是按照栈的延伸方向排列的,先定义的变量先入栈,后定义的变 量后入栈;

### 111、说一说你了解的关于lambda函数的全部知识

1. 利用lambda表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象；
2. 每当你定义一个lambda表达式后，编译器会自动生成一个匿名类（这个类当然重载了()运算符），我 们称为闭包类型（closure type）。那么在运行时，这个lambda表达式就会返回一个匿名的闭包实例， 其实一个右值。所以，我们上面的lambda表达式的结果就是一个个闭包。闭包的一个强大之处是其可 以通过传值或者引用的方式捕捉其封装作用域内的变量，前面的方括号就是用来定义捕捉模式以及变 量，我们又将其称为lambda捕捉块。
3. lambda表达式的语法定义如下：

[capture] （parameters） mutable ->return-type {statement};

1. lambda必须使用尾置返回来指定返回类型，可以忽略参数列表和返回值，但必须永远包含捕获列表 和函数体；

### 112、将字符串“hello world”从开始到打印到屏幕上的全过程?

1. 用户告诉操作系统执行HelloWorld程序（通过键盘输入等）
2. 操作系统：找到helloworld程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信 息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址。
3. 操作系统：创建一个新进程，将HelloWorld可执行文件映射到该进程结构，表示由该进程执行helloworld程序。
4. 操作系统：为helloworld程序设置cpu上下文环境，并跳到程序开始处。 5．执行helloworld程序的第一条指令，发生缺页异常 6．操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行helloworld程序7．helloword程序执行puts函数（系统调用），在显示器上写一字符串
5. 操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要 写的字符串送给该进程
6. 操作系统：控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法 的操作，然后将字符串转换成像素，将像素写入设备的存储映像区
7. 视频硬件将像素转换成显示器可接收和一组控制数据信号
8. 显示器解释信号，激发液晶屏12．OK，我们在屏幕上看到了HelloWorld

### 114、为什么模板类一般都是放在一个h文件中

1. 模板定义很特殊。由template<…>处理的任何东西都意味着编译器在当时不为它分配存储空间，它 一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的 多重定义。

所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

1. 在分离式编译的环境下，编译器编译某一个.cpp文件时并不知道另一个.cpp文件的存在，也不会去 查找（当遇到未决符号时它会寄希望于连接器）。这种模式在没有模板的情况下运行良好，但遇到模板 时就傻眼了，因为模板仅在需要的时候才会实例化出来。

所以，当编译器只看到模板的声明时，它不能实例化该模板，只能创建一个具有外部连接的符号并期待 连接器能够将符号的地址决议出来。

然而当实现该模板的.cpp文件中没有用到模板的实例时，编译器懒得去实例化，所以，整个工程的.obj 中就找不到一行模板实例的二进制代码，于是连接器也黔驴技穷了。

### 115、C++中类成员的访问权限和继承权限问题

1. 三种访问权限

① public:用该关键字修饰的成员表示公有成员，该成员不仅可以在类内可以被 访问，在类外也是可以被访问的，是类对外提供的可访问接口；

② private:用该关键字修饰的成员表示私有成员，该成员仅在类内可以被访问，在类体外是隐藏状态；

③ protected:用该关键字修饰的成员表示保护成员，保护成员在类体外同样是隐藏状态，但是对于该类的派生类来说，相当于公有成员，在派生类中可以被访问。

1. 三种继承方式

① 若继承方式是public，基类成员在派生类中的访问权限保持不变，也就是说，基类中的成员访问权限，在派生类中仍然保持原来的访问权限；

② 若继承方式是private，基类所有成员在派生类中的访问权限都会变为私有(private)权限；

③ 若继承方式是protected，基类的共有成员和保护成员在派生类中的访问权限都会变为保护

(protected)权限，私有成员在派生类中的访问权限仍然是私有(private)权限。

### 116、你知道重载运算符吗？

1、 我们只能重载已有的运算符，而无权发明新的运算符；对于一个重载的运算符，其优先级和结合律与内置类型一致才可以；不能改变运算符操作数个数；

2、 两种重载方式：成员运算符和非成员运算符，成员运算符比非成员运算符少一个参数；下标运算符、箭头运算符必须是成员运算符；

3、 引入运算符重载，是为了实现类的多态性；

4、 当重载的运算符是成员函数时，this绑定到左侧运算符对象。成员运算符函数的参数数量比运算符对象的数量少一个；至少含有一个类类型的参数；

5、 从参数的个数推断到底定义的是哪种运算符，当运算符既是一元运算符又是二元运算符（+，-，\*，

&）；

6、 下标运算符必须是成员函数，下标运算符通常以所访问元素的引用作为返回值，同时最好定义下标运算符的常量版本和非常量版本；

7、 箭头运算符必须是类的成员，解引用通常也是类的成员；重载的箭头运算符必须返回类的指针；

### 117、全局变量和static变量的区别

1、全局变量（外部变量）的说明之前再冠以static就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。

这两者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程 序由多个原文件组成时，非静态的全局变量在各个源文件中都是有效的。

而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中 不能使用它。由于静态全局变量的作用域限于一个源文件内，只能为该源文件内的函数公用，因此可以 避免在其他源文件中引起错误。

static全局变量与普通的全局变量的区别是static全局变量只初始化一次，防止在其他文件单元被引用。2.static函数与普通函数有什么区别？

static函数与普通的函数作用域不同。尽在本文件中。只在当前源文件中使用的函数应该说明为内部函

数（static），内部函数应该在当前源文件中说明和定义。

对于可在当前源文件以外使用的函数应该在一个头文件中说明，要使用这些函数的源文件要包含这个头 文件。

static函数与普通函数最主要区别是static函数在内存中只有一份，普通静态函数在每个被调用中维持一 份拷贝程序的局部变量存在于（堆栈）中，全局变量存在于（静态区）中，动态申请数据存在于（堆）

### 118、 静态成员与普通成员的区别是什么？

1. 生命周期

静态成员变量从类被加载开始到类被卸载，一直存在；

普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；

1. 共享方式

静态成员变量是全类共享；普通成员变量是每个对象单独享用的；

1. 定义位置

普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；

1. 初始化位置

普通成员变量在类中初始化；静态成员变量在类外初始化；

1. 默认实参

可以使用静态成员变量作为默认实参，

### 119、隐式转换，如何消除隐式转换？

1、C++的基本类型中并非完全的对立，部分数据类型之间是可以进行隐式转换的。所谓隐式转换，是 指不需要用户干预，编译器私下进行的类型转换行为。很多时候用户可能都不知道进行了哪些转换

2、C++面向对象的多态特性，就是通过父类的类型实现对子类的封装。通过隐式转换，你可以直接将 一个子类的对象使用父类的类型进行返回。在比如，数值和布尔类型的转换，整数和浮点数的转换等。 某些方面来说，隐式转换给C++程序开发者带来了不小的便捷。C++是一门强类型语言，类型的检查是 非常严格的。

3、 基本数据类型 基本数据类型的转换以取值范围的作为转换基础（保证精度不丢失）。隐式转换发生在从小->大的转换中。比如从char转换为int。从int->long。自定义对象 子类对象可以隐式的转换为父类对象。

4、 C++中提供了explicit关键字，在构造函数声明的时候加上explicit关键字，能够禁止隐式转换。

5、如果构造函数只接受一个参数，则它实际上定义了转换为此类类型的隐式转换机制。可以通过将构 造函数声明为explicit加以制止隐式类型转换，关键字explicit只对一个实参的构造函数有效，需要多个 实参的构造函数不能用于执行隐式转换，所以无需将这些构造函数指定为explicit。

### 120、 虚函数的内存结构，那菱形继承的虚函数内存结构呢

参考：<https://blog.csdn.net/haoel/article/details/1948051/>

菱形继承的定义是：两个子类继承同一父类，而又有子类同时继承这两个子类。例如a,b两个类同时继 承c，但是又有一个d类同时继承a,b类。

### 121、多继承的优缺点，作为一个开发者怎么看待多继承

1. C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。
2. 多重继承的优点很明显，就是对象可以调用多个基类中的接口；
3. 如果派生类所继承的多个基类有相同的基类，而派生类对象需要调用这个祖先类的接口方法，就会容 易出现二义性
4. 加上全局符确定调用哪一份拷贝。比如pa.Author::eat()调用属于Author的拷贝。
5. 使用虚拟继承，使得多重继承类Programmer\_Author只拥有Person类的一份拷贝。

### 122、迭代器：++it、it++哪个好，为什么

1. 前置返回一个引用，后置返回一个对象

|  |
| --- |
| 1 // ++i实现代码为： 2  3 int& operator++() 4  5 {  6  7 \*this += 1;  8  9 return \*this; 10  11 }  12  13 |

1. 前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低

|  |
| --- |
| 1 //i++实现代码为： 2  3 int operator++(int) 4  5 {  6  7 int temp = \*this; 8  9 ++\*this;  10  11 return temp; 12  13 }  14  15 |

### 123、C++如何处理多个异常的？

1. C++中的异常情况：

语法错误（编译错误）：比如变量未定义、括号不匹配、关键字拼写错误等等编译器在编译时能发现的 错误，这类错误可以及时被编译器发现，而且可以及时知道出错的位置及原因，方便改正。

运行时错误：比如数组下标越界、系统内存不足等等。这类错误不易被程序员发现，它能通过编译且能 进入运行，但运行时会出错，导致程序崩溃。为了有效处理程序运行时错误，C++中引入异常处理机制 来解决此问题。

1. C++异常处理机制：

异常处理基本思想：执行一个函数的过程中发现异常，可以不用在本函数内立即进行处理， 而是抛出该异常，让函数的调用者直接或间接处理这个问题。

C++异常处理机制由3个模块组成：try(检查)、throw(抛出)、catch(捕获)

抛出异常的语句格式为：throw 表达式；如果try块中程序段发现了异常则抛出异常。

|  |
| --- |
| 1 try  2 {  3 可能抛出异常的语句；（检查）  4 }  5 catch（类型名[形参名]）//捕获特定类型的异常 6 {  7 //处理1；  8 }  9 catch（类型名[形参名]）//捕获特定类型的异常10 {  11 //处理2；  12 }  13 catch（…）//捕获所有类型的异常14 {  15 }  16  17 |

### 124、模板和实现可不可以不写在一个文件里面？为什么？

因为在编译时模板并不能生成真正的二进制代码，而是在编译调用模板类或函数的CPP文件时才会去找 对应的模板声明和实现，在这种情况下编译器是不知道实现模板类或函数的CPP文件的存在，所以它只 能找到模板类或函数的声明而找不到实现，而只好创建一个符号寄希望于链接程序找地址。

但模板类或函数的实现并不能被编译成二进制代码，结果链接程序找不到地址只好报错了。

《C++编程思想》第15章(第300页)说明了原因：模板定义很特殊。由template<…>处理的任何东西都 意味着编译器在当时不为它分配存储空间，

它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板 的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

### 125、在成员函数中调用delete this会出现什么问题？对象还可以使用吗？

1、在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码内容，类的成员函数单独放 在代码段中。在调用成员函数时，隐含传递一个this指针，让成员函数知道当前是哪个对象在调用它。 当调用delete this时，类对象的内存空间被释放。在delete this之后进行的其他任何函数调用，只要不涉及到this指针的内容，都能够正常运行。一旦涉及到this指针，如操作数据成员，调用虚函数等，就 会出现不可预期的问题。

2、为什么是不可预期的问题？

delete this之后不是释放了类对象的内存空间了么，那么这段内存应该已经还给系统，不再属于这个进程。照这个逻辑来看，应该发生指针错误，无访问权限之类的令系统崩溃的问题才对啊？这个问题牵涉 到操作系统的内存管理策略。delete this释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是 可以访问的，你可以加上100，加上200，但是其中的值却是不确定的。当你获取数据成员，可能得到 的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

3、 如果在类的析构函数中调用delete this，会发生什么？

会导致堆栈溢出。原因很简单，delete的本质是“为将被释放的内存调用一个或多个析构函数，然后，释 放内存”。显然，delete this会去调用本对象的析构函数，而析构函数中又调用delete this，形成无限递归，造成堆栈溢出，系统崩溃。

### 126、如何在不使用额外空间的情况下，交换两个数？你有几种方法

|  |
| --- |
| 1 1) 算术  2  3 x = x + y; 4 y = x - y; 5  6 x = x - y; 7  8 2) 异或  9   1. x = x^y;// 只能对int,char.. 2. y = x^y; 3. x = x^y;   13 x ^= y ^= x; 14  15 |

### 127、你知道strcpy和memcpy的区别是什么吗？

1、复制的内容不同。strcpy只能复制字符串，而memcpy可以复制任意内容，例如字符数组、整型、 结构体、类等。

2、复制的方法不同。strcpy不需要指定长度，它遇到被复制字符的串结束符"\0"才结束，所以容易溢 出。memcpy则是根据其第3个参数决定复制的长度。

3、用途不同。通常在复制字符串时用strcpy，而需要复制其他类型数据时则一般用memcpy

### 128、volatile关键字的作用？

volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比

如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再 进行优化，从而可以提供对特殊地址的稳定访问。声明时语法：int volatile vInt; 当要求使用 volatile 声明的变量的值的时候，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。而且读取的数据立刻被保存。

volatile用在如下的几个地方：

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；
2. 多任务环境下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volatile说明，因为每次对它的读写都可能由不同意义；

### 129、如果有一个空类，它会默认添加哪些函数？

|  |
| --- |
| 1 1) Empty(); // 缺省构造函数// 2  3 2) Empty( const Empty& ); // 拷贝构造函数// 4  5 3) ~Empty(); // 析构函数//  6  7 4) Empty& operator=( const Empty& ); // 赋值运算符// 8  9 |

**130、C++中标准库是什么？**

1. C++ 标准库可以分为两部分：

标准函数库： 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C 语言。面向对象类库： 这个库是类及其相关函数的集合。

1. 输入/输出 I/O、字符串和字符处理、数学、时间、日期和本地化、动态分配、其他、宽字符函数
2. 标准的 C++ I/O 类、String 类、数值类、STL 容器类、STL 算法、STL 函数对象、STL 迭代器、STL 分配器、本地化库、异常处理类、杂项支持库

### 131、你知道const char\* 与string之间的关系是什么吗？

1. string 是c++标准库里面其中一个，封装了对字符串的操作，实际操作过程我们可以用const char\*给string类初始化
2. 三者的转化关系如下所示：

*a*)  string转const char\*

*string* s = “abc”;

const char\* c\_s = s.*c\_str*();

*b*)  const char\* 转string，直接赋值即可

const char\* c\_s = “abc”;

*string* s(c\_s);

c)  string 转char\*

*string* s = “abc”;

char\* c;

const int len = s.*length*();

c = new char[len + 1];

strcpy(c, s.*c\_str*());

d)  char\* 转string

char\* c = “abc”;

*string* s(c);

e)  const char\* 转char\*

const char\* cpc = “abc”;

char\* pc = new char[*strlen*(cpc) + 1];

strcpy(pc, cpc);

f)  char\* 转const char\* ，直接赋值即可

char\* pc = “abc”;

1. const char\* cpc = pc;

### 132、为什么拷贝构造函数必须传引用不能传值？

1. 拷贝构造函数的作用就是用来复制对象的，在使用这个对象的实例来初始化这个对象的一个新的实 例。
2. 参数传递过程到底发生了什么？

将地址传递和值传递统一起来，归根结底还是传递的是"值"(地址也是值，只不过通过它可以找到另一 个值)！

* 1. 值传递:

对于内置数据类型的传递时，直接赋值拷贝给形参(注意形参是函数内局部变量)； 对于类类型的传递时，需要首先调用该类的拷贝构造函数来初始化形参(局部对象)；

如void foo(class\_type obj\_local){}, 如果调用foo(obj); 首先class\_type obj\_local(obj) ,这样就定义了局部变量obj\_local供函数内部使用

* 1. 引用传递:

无论对内置类型还是类类型，传递引用或指针最终都是传递的地址值！而地址总是指针类型(属于简单 类型), 显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用(对于类类型).

上述1) 2)回答了为什么拷贝构造函数使用值传递会产生无限递归调用，内存溢出。

拷贝构造函数用来初始化一个非引用类类型对象，如果用传值的方式进行传参数，那么构造实参需要调 用拷贝构造函数，而拷贝构造函数需要传递实参，所以会一直递归。

### 133、你知道空类的大小是多少吗？

1. C++空类的大小不为0，不同编译器设置不一样，vs设置为1；
2. C++标准指出，不允许一个对象（当然包括类对象）的大小为0，不同的对象不能具有相同的地址；
3. 带有虚函数的C++类大小不为1，因为每一个对象会有一个vptr指向虚函数表，具体大小根据指针大 小确定；
4. C++中要求对于类的每个实例都必须有独一无二的地址,那么编译器自动为空类分配一个字节大小， 这样便保证了每个实例均有独一无二的内存地址。

### 134、你什么情况用指针当参数，什么时候用引用，为什么？

1. 使用引用参数的主要原因有两个：

程序员能修改调用函数中的数据对象

通过传递引用而不是整个数据–对象，可以提高程序的运行速度

1. 一般的原则：

对于使用引用的值而不做修改的函数：

如果数据对象很小，如内置数据类型或者小型结构，则按照值传递；

如果数据对象是数组，则使用指针（唯一的选择），并且指针声明为指向const的指针；

如果数据对象是较大的结构，则使用const指针或者引用，已提高程序的效率。这样可以节省结构所需 的时间和空间；

如果数据对象是类对象，则使用const引用（传递类对象参数的标准方式是按照引用传递）；

1. 对于修改函数中数据的函数：

如果数据是内置数据类型，则使用指针如果数据对象是数组，则只能使用指针

如果数据对象是结构，则使用引用或者指针

如果数据是类对象，则使用引用

### 135、静态函数能定义为虚函数吗？常函数呢？说说你的理解

1、static成员不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义的。

2、静态与非静态成员函数之间有一个主要的区别，那就是静态成员函数没有this指针。

虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来 访问它，因为它是类的一个成员，并且vptr指向保存虚函数地址的vtable.对于静态成员函数，它没有 this指针，所以无法访问vptr。

这就是为何static函数不能为virtual，虚函数的调用关系：this -> vptr -> vtable ->virtual function

### 136、this指针调用成员变量时，堆栈会发生什么变化？

当在类的非静态成员函数访问类的非静态成员时，编译器会自动将对象的地址传给作为隐含参数传递给 函数，这个隐含参数就是this指针。

即使你并没有写this指针，编译器在链接时也会加上this的，对各成员的访问都是通过this的。

例如你建立了类的多个对象时，在调用类的成员函数时，你并不知道具体是哪个对象在调用，此时你可 以通过查看this指针来查看具体是哪个对象在调用。This指针首先入栈，然后成员函数的参数从右向左 进行入栈，最后函数返回地址入栈。

### 137、你知道静态绑定和动态绑定吗？讲讲？

1. 对象的静态类型：对象在声明时采用的类型。是在编译期确定的。
2. 对象的动态类型：目前所指对象的类型。是在运行期决定的。对象的动态类型可以更改，但是静态 类型无法更改。
3. 静态绑定：绑定的是对象的静态类型，某特性（比如函数依赖于对象的静态类型，发生在编译期。
4. 动态绑定：绑定的是对象的动态类型，某特性（比如函数依赖于对象的动态类型，发生在运行期。

### 144、成员初始化列表会在什么时候用到？它的调用过程是什么？

1. 当初始化一个引用成员变量时；
2. 初始化一个const成员变量时；
3. 当调用一个基类的构造函数，而构造函数拥有一组参数时；
4. 当调用一个成员类的构造函数，而他拥有一组参数；
5. 编译器会一一操作初始化列表，以适当顺序在构造函数之内安插初始化操作，并且在任何显示用户 代码前。list中的项目顺序是由类中的成员声明顺序决定的，不是初始化列表中的排列顺序决定的。

### 147、哪些函数不能是虚函数？把你知道的都说一说

1. 构造函数，构造函数初始化对象，派生类必须知道基类函数干了什么，才能进行构造；当有虚函数 时，每一个类有一个虚表，每一个对象有一个虚表指针，虚表指针在构造函数中初始化；
2. 内联函数，内联函数表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型 确定，所以内联函数不能是虚函数；
3. 静态函数，静态函数不属于对象属于类，静态成员函数没有this指针，因此静态函数设置为虚函数没 有任何意义。
4. 友元函数，友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说 法。
5. 普通函数，普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚函数。

### 148、说一说strcpy、sprintf与memcpy这三个函数的不同之处

1. 操作对象不同

① strcpy的两个操作对象均为字符串

② sprintf的操作源对象可以是多种数据类型，目的操作对象是字符串

③ memcpy的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。

1. 执行效率不同

memcpy最高，strcpy次之，sprintf的效率最低。

1. 实现功能不同

① strcpy主要实现字符串变量间的拷贝

② sprintf主要实现其他数据类型格式到字符串的转化

③ memcpy主要是内存块间的拷贝。

### 149、将引用作为函数参数有哪些好处？

1. 传递引用给函数与传递指针的效果是一样的。

这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中 对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

1. 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；

而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量 的副本；

如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传 递参数的效率和所占空间都好。

1. 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配 存储单元，且需要重复使用"\*指针变量名"的形式进行运算，这很容易产生错误且程序的阅读性较差；

另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

### 150、你知道数组和指针的区别吗？

1. 数组在内存中是连续存放的，开辟一块连续的内存空间；数组所占存储空间：sizeof（数组名）；数 组大小：sizeof(数组名)/sizeof(数组元素数据类型)；
2. 用运算符sizeof 可以计算出数组的容量（字节数）。sizeof(p),p 为指针得到的是一个指针变量的字节数，而不是p 所指的内存容量。
3. 编译器为了简化对数组的支持，实际上是利用指针实现了对数组的支持。具体来说，就是将表达式 中的数组元素引用转换为指针加偏移量的引用。
4. 在向函数传递参数的时候，如果实参是一个数组，那用于接受的形参为对应的指针。也就是传递过 去是数组的首地址而不是整个数组，能够提高效率；
5. 在使用下标的时候，两者的用法相同，都是原地址加上下标值，不过数组的原地址就是数组首元素 的地址是固定的，指针的原地址就不是固定的。

### 151、如何阻止一个类被实例化？有哪些方法？

1. 将类定义为抽象基类或者将构造函数声明为private；
2. 不允许类外部创建类对象，只能在类内部创建对象

### 152、 如何禁止程序自动生成拷贝构造函数？

1. 为了阻止编译器默认生成拷贝构造函数和拷贝赋值函数，我们需要手动去重写这两个函数，某些情 况下，为了避免调用拷贝构造函数和拷贝赋值函数，我们需要将他们设置成private，防止被调用。
2. 类的成员函数和friend函数还是可以调用private函数，如果这个private函数只声明不定义，则会产 生一个连接错误；
3. 针对上述两种情况，我们可以定一个base类，在base类中将拷贝构造函数和拷贝赋值函数设置成private,那么派生类中编译器将不会自动生成这两个函数，且由于base类中该函数是私有的，因此，派 生类将阻止编译器执行相关的操作。

### 153、你知道Debug和release的区别是什么吗？

1. 调试版本，包含调试信息，所以容量比Release大很多，并且不进行任何优化（优化会使调试复杂 化，因为源代码和生成的指令间关系会更复杂），便于程序员调试。Debug模式下生成两个文件，除 了.exe或.dll文件外，还有一个.pdb文件，该文件记录了代码中断点等调试信息；
2. 发布版本，不对源代码进行调试，编译时对应用程序的速度进行优化，使得程序在代码大小和运行 速度上都是最优的。（调试信息可在单独的PDB文件中生成）。Release模式下生成一个文件.exe或.dll 文件。
3. 实际上，Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发 布版本。

### 154、模板会写吗？写一个比较大小的模板函数

|  |
| --- |
| 1 #include<iostream> 2   1. using namespace std; 2. template<typename type1,typename type2>//函数模板5   6 type1 Max(type1 a,type2 b) 7  8 {  9  10 return a > b ? a : b; 11  12 }  13  14 void main() 15  16 {  17  18 cout<<"Max = "<<Max(5.5,'a')<<endl; 19  20 }  21  22 |

**155、智能指针出现循环引用怎么解决？**

弱指针用于专门解决shared\_ptr循环引用的问题，weak\_ptr不会修改引用计数，即其存在与否并不影响 对象的引用计数器。循环引用就是：两个对象互相使用一个shared\_ptr成员变量指向对方。弱引用并不 对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理 的对象是否已经被释放，从而避免访问非法内存。

### 156、strcpy函数和strncpy函数的区别？哪个函数更安全？

1. 函数原型

|  |
| --- |
| 1. char\* strcpy(char\* strDest, const char\* strSrc) 2. char\* strncpy(char\* strDest, const char\* strSrc, int pos) 3   4 |

1. strcpy函数: 如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buﬀer Overﬂow)的错误情况，在编写程序时请特别留意，或者用strncpy()来取代。

strncpy函数：用来复制源字符串的前n个字符，src 和 dest 所指的内存区域不能重叠，且 dest 必须有足够的空间放置n个字符。

1. 如果目标长>指定长>源长，则将源长全部拷贝到目标长，自动加上’\0’ 如果指定长<源长，则将源长中按指定长度拷贝到目标字符串，不包括’\0’ 如果指定长>目标长，运行时错误 ；

### 157、static\_cast比C语言中的转换强在哪里？

1. 更加安全；
2. 更直接明显，能够一眼看出是什么类型转换为什么类型，容易找出程序中的错误；可清楚地辨别代 码中每个显式的强制转；可读性更好，能体现程序员的意图

### 158、成员函数里memset(this,0,sizeof(\*this))会发生什么

1. 有时候类里面定义了很多int,char,struct等c语言里的那些类型的变量，我习惯在构造函数中将它们初 始化为0，但是一句句的写太麻烦，所以直接就memset(this, 0, sizeof \*this);将整个对象的内存全部置 为0。对于这种情形可以很好的工作，但是下面几种情形是不可以这么使用的；
2. 类含有虚函数表：这么做会破坏虚函数表，后续对虚函数的调用都将出现异常；
3. 类中含有C++类型的对象：例如，类中定义了一个list的对象，由于在构造函数体的代码执行之前就 对list对象完成了初始化，假设list在它的构造函数里分配了内存，那么我们这么一做就破坏了list对象的 内存。

### 159、你知道回调函数吗？它的作用？

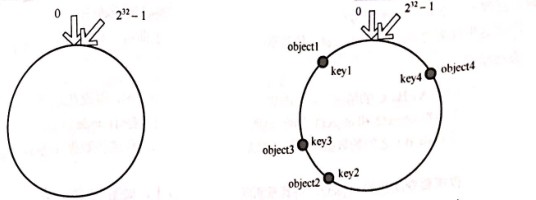
1. 当发生某种事件时，系统或其他函数将会自动调用你定义的一段函数；
2. 回调函数就相当于一个中断处理函数，由系统在符合你设定的条件时自动调用。为此，你需要做三 件事：1，声明；2，定义；3，设置触发条件，就是在你的函数中把你的回调函数名称转化为地址作为 一个参数，以便于系统调用；
3. 回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个 函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数；
4. 因为可以把调用者与被调用者分开。调用者不关心谁是被调用者，所有它需知道的，只是存在一个 具有某种特定原型、某些限制条件（如返回值为int）的被调用函数。

### 160、什么是一致性哈希？

##### 一致性哈希

一致性哈希是一种哈希算法，就是**在移除或者增加一个结点时，能够尽可能小的改变已存在key的映射 关系**

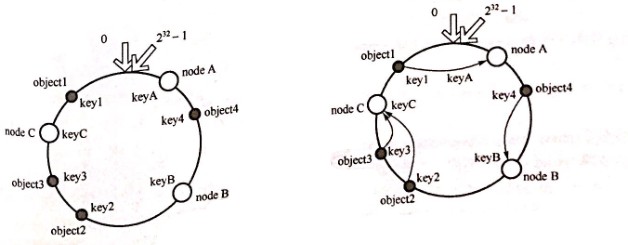
尽可能少的改变已有的映射关系，一般是沿着顺时针进行操作，回答之前可以先想想，真实情况如何处 理

一致性哈希将整个哈希值空间组**织成一个虚拟的圆环**，假设哈希函数的值空间为0~2^32-1，整个哈希空 间环如下左图所示

一致性hash的基本思想就是使用相同的hash算法将数据和结点都映射到图中的环形哈希空间中，上右 图显示了4个数据object1-object4在环上的分布图

##### 结点和数据映射

假如有一批服务器，可以根据IP或者主机名作为关键字进行哈希，根据结果映射到哈希环中，3台服务 器分别是nodeA-nodeC

现在有一批的数据object1-object4需要存在服务器上，则可以使用相同的哈希算法对数据进行哈希，其 结果必然也在环上，可以沿着顺时针方向寻找，找到一个结点（服务器）则将数据存在这个结点上，这 样数据和结点就产生了一对一的关联，如下图所示：

##### 移除结点

如果一台服务器出现问题，如上图中的nodeB，则受影响的是其逆时针方向至下一个结点之间的数据， 只需将这些数据映射到它顺时针方向的第一个结点上即可，下左图



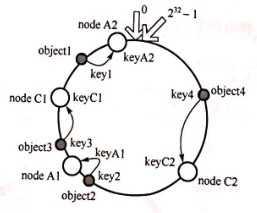
##### 添加结点

如果新增一台服务器nodeD，受影响的是其逆时针方向至下一个结点之间的数据，将这些数据映射到

nodeD上即可，见上右图

##### 虚拟结点

假设仅有2台服务器：nodeA和nodeC，nodeA映射了1条数据，nodeC映射了3条，这样数据分布是不 平衡的。引入虚拟结点，假设结点复制个数为2，则nodeA变成：nodeA1和nodeA2，nodeC变成： nodeC1和nodeC2，映射情况变成如下：



这样数据分布就均衡多了，平衡性有了很大的提高

《程序员求职宝典》王道论坛

### 161、什么是纯虚函数，与虚函数的区别

##### 虚函数和纯虚函数区别？

虚函数是为了实现动态编联产生的，目的是通过基类类型的指针指向不同对象时，自动调用相应 的、和基类同名的函数（使用同一种调用形式，既能调用派生类又能调用基类的同名函数）。虚函 数需要在基类中加上virtual修饰符修饰，因为virtual会被隐式继承，所以子类中相同函数都是虚函 数。当一个成员函数被声明为虚函数之后，其派生类中同名函数自动成为虚函数，在派生类中重新 定义此函数时要求函数名、返回值类型、参数个数和类型全部与基类函数相同。

纯虚函数只是相当于一个接口名，但含有纯虚函数的类不能够实例化。

纯虚函数首先是虚函数，其次它没有函数体，取而代之的是用“=0”。

既然是虚函数，它的函数指针会被存在虚函数表中，由于纯虚函数并没有具体的函数体，因此它在虚函 数表中的值就为0，而具有函数体的虚函数则是函数的具体地址。

一个类中如果有纯虚函数的话，称其为抽象类。抽象类不能用于实例化对象，否则会报错。抽象类一般 用于定义一些公有的方法。子类继承抽象类也必须实现其中的纯虚函数才能实例化对象。

举个例子：

#include <iostream>

using namespace std;

class Base

{

public:

virtual void fun1()

{

cout << "普通虚函数" << *endl*;

}

virtual void fun2() = 0;

virtual ~Base() {}

};

class Son : public Base

{

public:

virtual void fun2()

{

cout << "子类实现的纯虚函数" << *endl*;

}

};

int main()

{

Base\* b = new Son;

b->fun1(); //普通虚函数

b->fun2(); //子类实现的纯虚函数

return 0;

}

### 162、C++从代码到可执行程序经历了什么？

##### 预编译

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下：

* 1. 删除所有的#deﬁne，展开所有的宏定义。
  2. 处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。
  3. 处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其 他

文件。

4. 删除所有的注释，“//”和“/\*\*/”。

1. 保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重

复引用。

1. 添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是 能够显示行号。

##### 编译

把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相 应

的汇编代码文件。

* 1. 词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分 割成一系列的记号。
  2. 语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的 语法树是一种以表达式为节点的树。
  3. 语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进 行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能

确定

的语义。

* 1. 优化：源代码级别的一个优化过程。
  2. 目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语 言

表示。

* 1. 目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移 来替代乘法运算、删除多余的指令等。

##### 汇编

将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过 来，汇编过程有汇编器as完成。经汇编之后，产生目标文件(与可执行文件格式几乎一

样)xxx.o(Windows

下)、xxx.obj(Linux下)。

##### 链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链 接：

##### 静态链接

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库 中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个 目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本；

更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西， 在执行的时候运行速度快。

##### 动态链接

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形 成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多分，副 本，而是这多个程序在执行时共享同一份副本；

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运 行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损 失。

[《操作系统（三）》：https://www.nowcoder.com/tutorial/93/675fd4af3ab34b2db0ae65085 5aa52d5](https://www.nowcoder.com/tutorial/93/675fd4af3ab34b2db0ae650855aa52d5)

### 164、迭代器失效的情况

以vector为例：

插入元素：

1、尾后插入：size < capacity时，首迭代器不失效尾迭代失效（未重新分配空间），size == capacity

时，所有迭代器均失效（需要重新分配空间）。

2、中间插入：中间插入：size < capacity时，首迭代器不失效但插入元素之后所有迭代器失效，size

== capacity时，所有迭代器均失效。

删除元素：

尾后删除：只有尾迭代失效。

中间删除：删除位置之后所有迭代失效。

deque 和 vector 的情况类似,

而list双向链表每一个节点内存不连续, 删除节点仅当前迭代器失效,erase返回下一个有效迭代器;

map/set等关联容器底层是红黑树删除节点不会影响其他节点的迭代器, 使用递增方法获取下一个迭代器 mmp.erase(iter++);

unordered\_(hash) 迭代器意义不大, rehash之后, 迭代器应该也是全部失效.

### 167、介绍一下几种典型的锁

##### 读写锁

多个读者可以同时进行读

写者必须互斥（只允许一个写者写，也不能读者写者同时进行）

写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

##### 互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责 线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以 互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线 程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

##### 条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个 线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足 时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变 量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说互斥锁是线程间互 斥的机制，条件变量则是同步机制。

##### 自旋锁

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。 如果别的线程长时期占有锁那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的 场景，这个时候效率比较高。

[《互斥锁、读写锁、自旋锁、条件变量的特点总结》：https://blog.csdn.net/RUN32875094/arti cle/details/80169978](https://blog.csdn.net/RUN32875094/article/details/80169978)

### 168、说一下C++左值引用和右值引用

C++11正是通过引入右值引用来优化性能，具体来说是通过移动语义来避免无谓拷贝的问题，通过move语义来将临时生成的左值中的资源无代价的转移到另外一个对象中去，通过完美转发来解决不能 按照参数实际类型来转发的问题（同时，完美转发获得的一个好处是可以实现移动语义）。

1. 在C++11中所有的值必属于左值、右值两者之一，右值又可以细分为纯右值、将亡值。在C++11中可 以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值（将亡值或纯右值）。举 个例子，int a = b+c, a 就是左值，其有变量名为a，通过&a可以获取该变量的地址；表达式b+c、函数int func()的返回值是右值，在其被赋值给某一变量前，我们不能通过变量名找到它，＆(b+c)这样的操作则不会通过编译。
2. C++11对C++98中的右值进行了扩充。在C++11中右值又分为纯右值（prvalue，Pure Rvalue）和将亡值（xvalue，eXpiring Value）。其中纯右值的概念等同于我们在C++98标准中右值的概念，指的是临时变量和不跟对象关联的字面量值；将亡值则是C++11新增的跟右值引用相关的表达式，这样表达式 通常是将要被移动的对象（移为他用），比如返回右值引用T&&的函数返回值、std::move的返回值， 或者转换为T&&的类型转换函数的返回值。将亡值可以理解为通过“盗取”其他变量内存空间的方式获取 到的值。在确保其他变量不再被使用、或即将被销毁时，通过“盗取”的方式可以避免内存空间的释放和 分配，能够延长变量值的生命期。
3. 左值引用就是对一个左值进行引用的类型。右值引用就是对一个右值进行引用的类型，事实上，由 于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。右值引用和左值引用都是属于引用 类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类 型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右 值引用则是不具名（匿名）变量的别名。左值引用通常也不能绑定到右值，但常量左值引用是个“万能” 的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。不过常量左值所引用的右值在 它的“余生”中只能是只读的。相对地，非常量左值只能接受非常量左值对其进行初始化。
4. 右值值引用通常不能绑定到任何的左值，要想绑定一个左值到右值引用，通常需要std::move()将左 值强制转换为右值。

##### 左值和右值

左值：表示的是可以获取地址的表达式，它能出现在赋值语句的左边，对该表达式进行赋值。但是修饰 符const的出现使得可以声明如下的标识符，它可以取得地址，但是没办法对其进行赋值

|  |
| --- |
| 1 const int& a = 10; 2  3  4  5 |

右值：表示无法获取地址的对象，有常量值、函数返回值、lambda表达式等。无法获取地址，但不表 示其不可改变，当定义了右值的右值引用时就可以更改右值。

##### 左值引用和右值引用

左值引用：传统的C++中引用被称为左值引用

右值引用：C++11中增加了右值引用，右值引用关联到右值时，右值被存储到特定位置，右值引用指向 该特定位置，也就是说，右值虽然无法获取地址，但是右值引用是可以获取地址的，该地址表示临时对 象的存储位置

##### 这里主要说一下右值引用的特点：

特点1：通过右值引用的声明，右值又“重获新生”，其生命周期与右值引用类型变量的生命周期一 样长，只要该变量还活着，该右值临时量将会一直存活下去

特点2：右值引用独立于左值和右值。意思是右值引用类型的变量可能是左值也可能是右值 特点3：T&& t在发生自动类型推断的时候，它是左值还是右值取决于它的初始化。

#include <bits/stdc++.h>

using namespace std;

template<typename T>

void fun(T&& t) {

cout << t << *endl*;

}

int getInt()

{

return 5;

}

int main() {

int a = 10;

int& b = a;  //b是左值引用

int& c = 10;  //错误，c是左值不能使用右值初始化

int&& d = 10;  //正确，右值引用用右值初始化

int&& e = a;  //错误，e是右值引用不能使用左值初始化

const int& f = a; //正确，左值常引用相当于是万能型，可以用左值或者右值初始化

const int& g = 10;//正确，左值常引用相当于是万能型，可以用左值或者右值初始化

const int&& h = 10; //正确，右值常引用

const int& aa = h;//正确

int& i = getInt();  //错误，i是左值引用不能使用临时变量（右值）初始化

int&& j = getInt();  //正确，函数返回值是右值

fun(10); //此时fun函数的参数t是右值

fun(a); //此时fun函数的参数t是左值

return 0;

}

《c++右值引用以及使用》：<https://www.cnblogs.com/likaiming/p/9045642.html>

《从4行代码看右值引用》：<https://www.cnblogs.com/likaiming/p/9029908.html>

### 170、 vector与list的区别与应用？怎么找某vector或者list的倒数第二个元素

1. vector数据结构

vector和数组类似，拥有一段连续的内存空间，并且起始地址不变。因此能高效的进行随机存取，时间 复杂度为o(1);但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复 杂度为o(n)。另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。连续存储结 构：vector是可以实现动态增长的对象数组，支持对数组高效率的访问和在数组尾端的删除和插入操 作，在中间和头部删除和插入相对不易，需要挪动大量的数据。它与数组最大的区别就是vector不需程 序员自己去考虑容量问题，库里面本身已经实现了容量的动态增长，而数组需要程序员手动写入扩容函 数进形扩容。

1. list数据结构

list是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以list的随机存取非常 没有效率，时间复杂度为o(n);但由于链表的特点，能高效地进行插入和删除。非连续存储结构：list是 一个双链表结构，支持对链表的双向遍历。每个节点包括三个信息：元素本身，指向前一个元素的节点

（prev）和指向下一个元素的节点（next）。因此list可以高效率的对数据元素任意位置进行访问和插 入删除等操作。由于涉及对额外指针的维护，所以开销比较大。

区别：

vector的随机访问效率高，但在插入和删除时（不包括尾部）需要挪动数据，不易操作。list的访问要遍 历整个链表，它的随机访问效率低。但对数据的插入和删除操作等都比较方便，改变指针的指向即可。 list是单向的，vector是双向的。vector中的迭代器在使用后就失效了，而list的迭代器在使用之后还可 以继续使用。

3)

int mySize = vec.size();vec.at(mySize -2);

list不提供随机访问，所以不能用下标直接访问到某个位置的元素，要访问list里的元素只能遍历，不过 你要是只需要访问list的最后N个元素的话，可以用反向迭代器来遍历：

### 171、STL 中vector删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间？

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。 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。

5、resize()函数只改变容器的元素数目，未改变容器大小。

6、用reserve(size\_type)只是扩大capacity值，这些内存空间可能还是“野”的，如果此时使用“[ ]”来访问，则可能会越界。而resize(size\_type new\_size)会真正使容器具有new\_size个对象。

不同的编译器，vector有不同的扩容大小。在vs下是1.5倍，在GCC下是2倍；

空间和时间的权衡。简单来说， 空间分配的多，平摊时间复杂度低，但浪费空间也多。

使用k=2增长因子的问题在于，每次扩展的新尺寸必然刚好大于之前分配的总和，也就是说，之前分配 的内存空间不可能被使用。这样对内存不友好。最好把增长因子设为(1,2)

对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到

O(n)的时间复杂度，因此，使用成倍的方式扩容。

如何释放空间

由于vector的内存占用空间只增不减，比如你首先分配了10,000个字节，然后erase掉后面9,999个，留 下一个有效元素，但是内存占用仍为10,000个。所有内存空间是在vector析构时候才能被系统回收。empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使clear()，vector所占用的内存空 间依然如故，无法保证内存的回收。

如果需要空间动态缩小，可以考虑使用deque。如果vector，可以用swap()来帮助你释放内存。

|  |
| --- |
| 1. vector(Vec).swap(Vec); 2. 将Vec的内存空洞清除； 3. vector().swap(Vec); 4. 清空Vec的内存；   5 |

### 172、容器内部删除一个元素

1. 顺序容器（序列式容器，比如vector、deque）

erase迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list除外)，所 以不能使用erase(it++)的方式，但是erase的返回值是下一个有效迭代器；

It = c.erase(it);

1. 关联容器(关联式容器，比如map、set、multimap、multiset等)

erase迭代器只是被删除元素的迭代器失效，但是返回值是void，所以要采用erase(it++)的方式删除迭 代器；

c.erase(it++)

### 173、STL迭代器如何实现

1、 迭代器是一种抽象的设计理念，通过迭代器可以在不了解容器内部原理的情况下遍历容器，除此之外，STL中迭代器一个最重要的作用就是作为容器与STL算法的粘合剂。

2、 迭代器的作用就是提供一个遍历容器内部所有元素的接口，因此迭代器内部必须保存一个与容器相关联的指针，然后重载各种运算操作来遍历，其中最重要的是\*运算符与->运算符，以及++、--等可能需 要重载的运算符重载。这和C++中的智能指针很像，智能指针也是将一个指针封装，然后通过引用计数 或是其他方法完成自动释放内存的功能。

3、最常用的迭代器的相应型别有五种：value type、diﬀerence type、pointer、reference、iterator catagoly;

### 174、map、set是怎么实现的，红黑树是怎么能够同时实现这两种容器？ 为什么使用红黑树？

1. 他们的底层都是以红黑树的结构实现，因此插入删除等操作都在O(logn时间内完成，因此可以完成 高效的插入删除；
2. 在这里我们定义了一个模版参数，如果它是key那么它就是set，如果它是map，那么它就是map； 底层是红黑树，实现map的红黑树的节点数据类型是key+value，而实现set的节点数据类型是value
3. 因为map和set要求是自动排序的，红黑树能够实现这一功能，而且时间复杂度比较低。

### 175、如何在共享内存上使用stl标准库？

1. 想像一下把STL容器，例如map, vector, list等等，放入共享内存中，IPC一旦有了这些强大的通用数据结构做辅助，无疑进程间通信的能力一下子强大了很多。

我们没必要再为共享内存设计其他额外的数据结构，另外，STL的高度可扩展性将为IPC所驱使。STL容 器被良好的封装，默认情况下有它们自己的内存管理方案。

当一个元素被插入到一个STL列表(list)中时，列表容器自动为其分配内存，保存数据。考虑到要将STL容 器放到共享内存中，而容器却自己在堆上分配内存。

一个最笨拙的办法是在堆上构造STL容器，然后把容器复制到共享内存，并且确保所有容器的内部分配 的内存指向共享内存中的相应区域，这基本是个不可能完成的任务。

1. 假设进程A在共享内存中放入了数个容器，进程B如何找到这些容器呢？

一个方法就是进程A把容器放在共享内存中的确定地址上（ﬁxed oﬀsets），则进程B可以从该已知地址上获取容器。另外一个改进点的办法是，进程A先在共享内存某块确定地址上放置一个map容器，然后 进程A再创建其他容器，然后给其取个名字和地址一并保存到这个map容器里。

进程B知道如何获取该保存了地址映射的map容器，然后同样再根据名字取得其他容器的地址。

### 176、map插入方式有几种？

|  |
| --- |
| 1 1) 用insert函数插入pair数据，  2  3 mapStudent.insert(pair<int, string>(1, "student\_one")); 4  5 2) 用insert函数插入value\_type数据  6  7 mapStudent.insert(map<int, string>::value\_type (1, "student\_one")); 8  9 3) 在insert函数中使用make\_pair()函数  10  11 mapStudent.insert(make\_pair(1, "student\_one")); 12  13 4) 用数组方式插入数据  14  15 mapStudent[1] = "student\_one"; 16  17  18 |

### 178、vector越界访问下标，map越界访问下标？vector删除元素时会不会释放空间？

1. 通过下标访问vector中的元素时不会做边界检查，即便下标越界。

也就是说，下标与ﬁrst迭代器相加的结果超过了ﬁnish迭代器的位置，程序也不会报错，而是返回这个 地址中存储的值。

如果想在访问vector中的元素时首先进行边界检查，可以使用vector中的at函数。通过使用at函数不但 可以通过下标访问vector中的元素，而且在at函数内部会对下标进行边界检查。

1. map的下标运算符[]的作用是：将key作为下标去执行查找，并返回相应的值；如果不存在这个key， 就将一个具有该key和value的某人值插入这个map。
2. erase()函数，只能删除内容，不能改变容量大小;

erase成员函数，它删除了itVect迭代器指向的元素，并且返回要被删除的itVect之后的迭代器，迭代器 相当于一个智能指针;clear()函数，只能清空内容，不能改变容量大小;如果要想在删除内容的同时释放 内存，那么你可以选择deque容器。

### 179、map中[]与ﬁnd的区别？

1. map的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关 键码，就将一个具有该关键码和值类型的默认值的项插入这个map。
2. map的ﬁnd函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回 尾迭代器。

### 180、 STL中list与queue之间的区别

1. list不再能够像vector一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在；
2. list插入操作和结合才做都不会造成原有的list迭代器失效;
3. list不仅是一个双向链表，而且还是一个环状双向链表，所以它只需要一个指针；
4. list不像vector那样有可能在空间不足时做重新配置、数据移动的操作，所以插入前的所有迭代器在 插入操作之后都仍然有效；
5. deque是一种双向开口的连续线性空间，所谓双向开口，意思是可以在头尾两端分别做元素的插入和 删除操作；可以在头尾两端分别做元素的插入和删除操作；
6. deque和vector最大的差异，一在于deque允许常数时间内对起头端进行元素的插入或移除操作，二 在于deque没有所谓容量概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间 并链接起来，deque没有所谓的空间保留功能。

### 183、常见容器性质总结？

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

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.

1. 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
2. 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）
3. 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
4. set 底层数据结构为红黑树，有序，不重复
5. multiset 底层数据结构为红黑树，有序，可重复
6. map 底层数据结构为红黑树，有序，不重复
7. multimap 底层数据结构为红黑树，有序，可重复11.unordered\_set 底层数据结构为hash表，无序，不重复12.unordered\_multiset 底层数据结构为hash表，无序，可重复
8. unordered\_map 底层数据结构为hash表，无序，不重复
9. unordered\_multimap 底层数据结构为hash表，无序，可重复

### 184、vector的增加删除都是怎么做的？为什么是1.5或者是2倍？

1. 新增元素：vector通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块 更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素；
2. 对vector的任何操作，一旦引起空间重新配置，指向原vector的所有迭代器就都失效了 ；
3. 初始时刻vector的capacity为0，塞入第一个元素后capacity增加为1；
4. 不同的编译器实现的扩容方式不一样，VS2015中以1.5倍扩容，GCC以2倍扩容。

对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到

O(n)的时间复杂度，因此，使用成倍的方式扩容。

1. 考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以2二倍的 方式扩容，或者以1.5倍的方式扩容。
2. 以2倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能 再被使用，所以最好倍增长因子设置为(1,2)之间：
3. 向量容器vector的成员函数pop\_back()可以删除最后一个元素.
4. 而函数erase()可以删除由一个iterator指出的元素，也可以删除一个指定范围的元素。
5. 还可以采用通用算法remove()来删除vector容器中的元素.
6. 不同的是：采用remove一般情况下不会改变容器的大小，而pop\_back()与erase()等成员函数会改变 容器的大小。

### 185、说一下STL每种容器对应的迭代器

|  |  |
| --- | --- |
| **容器** | **迭代器** |
| vector、deque | 随机访问迭代器 |
| stack、queue、priority\_queue | 无 |
| list、(multi)set/map | 双向迭代器 |
| unordered\_(multi)set/map、forward\_list | 前向迭代器 |

**186、STL中vector的实现**

vector是一种序列式容器，其数据安排以及操作方式与array非常类似，两者的唯一差别就是对于空间 运用的灵活性，众所周知，array占用的是静态空间，一旦配置了就不可以改变大小，如果遇到空间不 足的情况还要自行创建更大的空间，并手动将数据拷贝到新的空间中，再把原来的空间释放。vector则 使用灵活的动态空间配置，维护一块**连续的线性空间**，在空间不足时，可以自动扩展空间容纳新元素， 做到按需供给。其在扩充空间的过程中仍然需要经历：**重新配置空间，移动数据，释放原空间**等操作。

这里需要说明一下动态扩容的规则：以原大小的两倍配置另外一块较大的空间（或者旧长度+新增元素 的个数），源码：

|  |
| --- |
| 1 const size\_type len = old\_size + max(old\_size, n); 2  3  4  5 |

Vector扩容倍数与平台有关，在Win + VS 下是 1.5倍，在 Linux + GCC 下是 2 倍测试代码：

|  |
| --- |
| 1. #include <iostream> 2. #include <vector> 3. using namespace std; 4   5 int main() 6 {   1. //在Linux + GCC下 2. vector<int> res(2,0); 3. cout << res.capacity() <<endl; //2 4. res.push\_back(1); 5. cout << res.capacity() <<endl;//4 6. res.push\_back(2); 7. res.push\_back(3); 8. cout << res.capacity() <<endl;//8 9. return 0; 16   17  18 //在 win 10 + VS2019下   1. vector<int> res(2,0); 2. cout << res.capacity() <<endl; //2 3. res.push\_back(1); 4. cout << res.capacity() <<endl;//3 5. res.push\_back(2); 6. res.push\_back(3); 7. cout << res.capacity() <<endl;//6 26   27  28 }  29  30  31 |

运行上述代码，一开始配置了一块长度为2的空间，接下来插入一个数据，长度变为原来的两倍，为4， 此时已占用的长度为3，再继续两个数据，此时长度变为8，可以清晰的看到空间的变化过程

需要注意的是，频繁对vector调用push\_back()对性能是有影响的，这是因为每插入一个元素，如果空 间够用的话还能直接插入，若空间不够用，则需要重新配置空间，移动数据，释放原空间等操作，对程 序性能会造成一定的影响

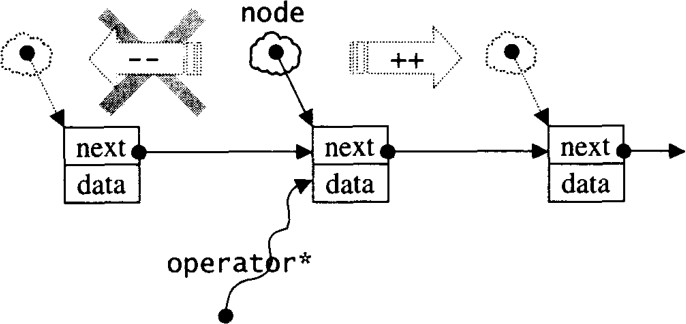
《STL源码剖析》 侯捷 P115-128

### 187、STL中slist的实现

list是双向链表，而slist（single linked list）是单向链表，它们的主要区别在于：前者的迭代器是双向的Bidirectional iterator，后者的迭代器属于单向的Forward iterator。虽然slist的很多功能不如list灵活，但是其所耗用的空间更小，操作更快。

根据STL的习惯，插入操作会将新元素插入到指定位置之前，而非之后，然而slist是不能回头的，只能 往后走，因此在slist的其他位置插入或者移除元素是十分不明智的，但是在slist开头却是可取的，slist 特别提供了insert\_after()和erase\_after供灵活应用。考虑到效率问题，slist只提供push\_front()操作， 元素插入到slist后，存储的次序和输入的次序是相反的

slist的单向迭代器如下图所示：



slist默认采用alloc空间配置器配置节点的空间，其数据结构主要代码如下

|  |
| --- |
| 1. template <class T, class Allco = alloc> 2. class slist 3 {   4 ...  5 private:  6 ...   1. static list\_node\* create\_node(const value\_type& x){}//配置空间、构造元素 2. static void destroy\_node(list\_node\* node){}//析构函数、释放空间 3. private: 4. list\_node\_base head; //头部 5. public: 6. iterator begin(){} 7. iterator end(){} 8. size\_type size(){} 9. bool empty(){} 10. void swap(slist& L){}//交换两个slist，只需要换head即可 11. reference front(){} //取头部元素 12. void push\_front(const value& x){}//头部插入元素 13. void pop\_front(){}//从头部取走元素20 ...   21 }  22  23  24  25 |

举个例子：

第 155／297页

|  |
| --- |
|  |

|  |
| --- |
| 1. #include <forward\_list> 2. #include <algorithm> 3. #include <iostream> 4. using namespace std; 5   6 int main() 7 {   1. forward\_list<int> fl; 2. fl.push\_front(1); 3. fl.push\_front(3); 4. fl.push\_front(2); 5. fl.push\_front(6); 6. fl.push\_front(5); 14 7. forward\_list<int>::iterator ite1 = fl.begin(); 8. forward\_list<int>::iterator ite2 = fl.end(); 9. for(;ite1 != ite2; ++ite1) 18 {   19 cout << \*ite1 <<" "; // 5 6 2 3 1  20 }  21 cout << endl; 22  23 ite1 = find(fl.begin(), fl.end(), 2); //寻找2的位置  24   1. if (ite1 != ite2) 2. fl.insert\_after(ite1, 99); 3. for (auto it : fl)   28 {  29 cout << it << " "; //5 6 2 99 3 1  30 }  31 cout << endl; 32   1. ite1 = find(fl.begin(), fl.end(), 6); //寻找6的位置 2. if (ite1 != ite2) 3. fl.erase\_after(ite1); 4. for (auto it : fl)   37 {  38 cout << it << " "; //5 6 99 3 1  39 }   1. cout << endl; 2. return 0; 42 }   43  44  45  46 |

需要注意的是C++标准委员会没有采用slist的名称，forward\_list在C++ 11中出现，它与slist的区别是没有size()方法。

《STL源码剖析》 侯捷

### 188、STL中list的实现

相比于vector的连续线型空间，list显得复杂许多，但是它的好处在于插入或删除都只作用于一个元素空 间，因此list对空间的运用是十分精准的，对任何位置元素的插入和删除都是常数时间。list不能保证节 点在存储空间中连续存储，也拥有迭代器，迭代器的“++”、“--”操作对于的是指针的操作，list提供的迭 代器类型是双向迭代器：Bidirectional iterators。

list节点的结构见如下源码：

|  |
| --- |
| 1. template <class T> 2. struct list\_node{ 3. typedef void\* void\_pointer; 4. void\_pointer prev; 5. void\_pointer next; 6. T data; 7 }   8  9  10 |

从源码可看出list显然是一个双向链表。list与vector的另一个区别是，在插入和接合操作之后，都不会 造成原迭代器失效，而vector可能因为空间重新配置导致迭代器失效。

此外list也是一个环形链表，因此只要一个指针便能完整表现整个链表。list中node节点指针始终指向尾 端的一个空白节点，因此是一种“前闭后开”的区间结构

list的空间管理默认采用alloc作为空间配置器，为了方便的以节点大小为配置单位，还定义一个list\_node\_allocator函数可一次性配置多个节点空间

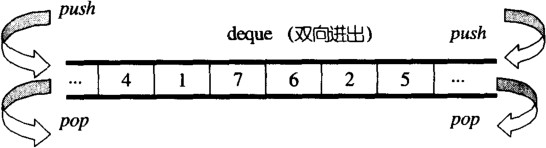
由于list的双向特性，其支持在头部（front)和尾部（back)两个方向进行push和pop操作，当然还支持

erase，splice，sort，merge，reverse，sort等操作，这里不再详细阐述。

《STL源码剖析》 侯捷 P128-142

### 189、STL中的deque的实现

vector是单向开口（尾部）的连续线性空间，deque则是一种双向开口的连续线性空间，虽然vector也 可以在头尾进行元素操作，但是其头部操作的效率十分低下（主要是涉及到整体的移动）



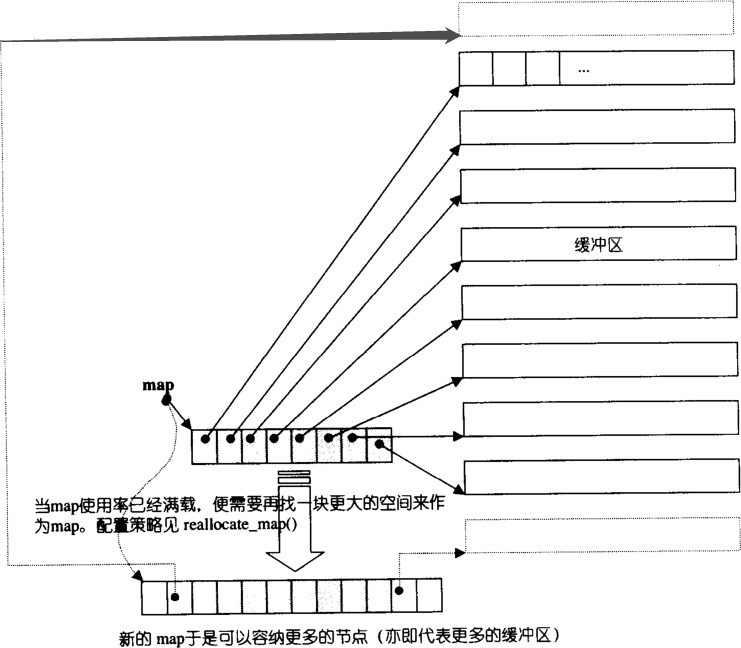
deque和vector的最大差异一个是deque运行在常数时间内对头端进行元素操作，二是deque没有容量 的概念，它是动态地以分段连续空间组合而成，可以随时增加一段新的空间并链接起来

deque虽然也提供随机访问的迭代器，但是其迭代器并不是普通的指针，其复杂程度比vector高很多， 因此除非必要，否则一般使用vector而非deque。如果需要对deque排序，可以先将deque中的元素复 制到vector中，利用sort对vector排序，再将结果复制回deque

deque由一段一段的定量连续空间组成，一旦需要增加新的空间，只要配置一段定量连续空间拼接在头 部或尾部即可，因此deque的最大任务是如何维护这个整体的连续性

deque的数据结构如下：

|  |
| --- |
| 1 class deque 2 {  3 ...   1. protected: 2. typedef pointer\* map\_pointer;//指向map指针的指针 3. map\_pointer map;//指向map 4. size\_type map\_size;//map的大小 5. public:   9 ...   1. iterator begin(); 2. itertator end();   12 ...  13 }  14  15  16 |

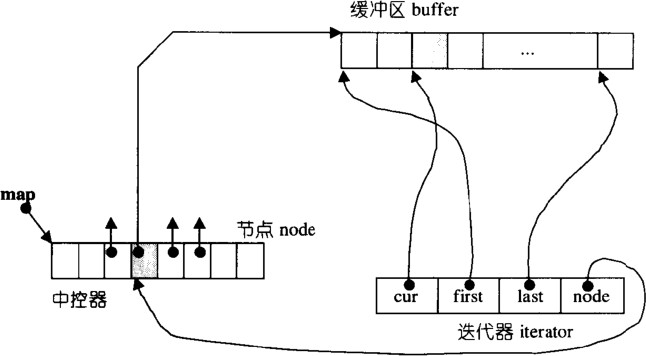


deque内部有一个指针指向map，map是一小块连续空间，其中的每个元素称为一个节点，node，每 个node都是一个指针，指向另一段较大的连续空间，称为缓冲区，这里就是deque中实际存放数据的 区域，默认大小512bytes。整体结构如上图所示。

deque的迭代器数据结构如下：

|  |
| --- |
| 1 struct deque\_iterator 2 {  3 ...   1. T\* cur;//迭代器所指缓冲区当前的元素 2. T\* first;//迭代器所指缓冲区第一个元素 3. T\* last;//迭代器所指缓冲区最后一个元素 4. map\_pointer node;//指向map中的node 8 ...   9 }  10  11  12 |

从deque的迭代器数据结构可以看出，为了保持与容器联结，迭代器主要包含上述4个元素



deque迭代器的“++”、“--”操作是远比vector迭代器繁琐，其主要工作在于缓冲区边界，如何从当前缓冲 区跳到另一个缓冲区，当然deque内部在插入元素时，如果map中node数量全部使用完，且node指向 的缓冲区也没有多余的空间，这时会配置新的map（2倍于当前+2的数量）来容纳更多的node，也就是 可以指向更多的缓冲区。在deque删除元素时，也提供了元素的析构和空闲缓冲区空间的释放等机制。

《STL源码剖析》 侯捷 P143-164

### 190、STL中stack和queue的实现

##### stack

stack（栈）是一种先进后出（First In Last Out）的数据结构，只有一个入口和出口，那就是栈顶，除了获取栈顶元素外，没有其他方法可以获取到内部的其他元素，其结构图如下：



stack这种单向开口的数据结构很容易由**双向开口的deque和list**形成，只需要根据stack的性质对应移 除某些接口即可实现，stack的源码如下：

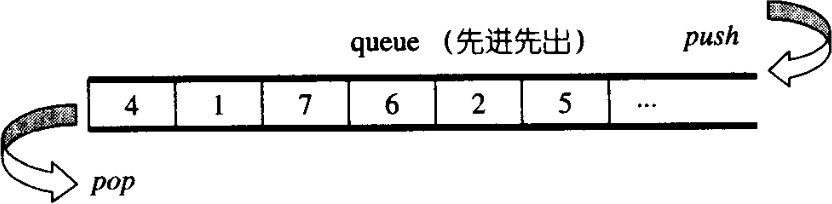
|  |
| --- |
| 1. template <class T, class Sequence = deque<T> > 2. class stack 3 {   4 ...   1. protected: 2. Sequence c; 3. public: 4. bool empty(){return c.empty();} 5. size\_type size() const{return c.size();} 6. reference top() const {return c.back();} 7. const\_reference top() const{return c.back();} 8. void push(const value\_type& x){c.push\_back(x);} 9. void pop(){c.pop\_back();} 14 };   15  16  17 |

从stack的数据结构可以看出，其所有操作都是围绕Sequence完成，而Sequence默认是deque数据结 构。stack这种“修改某种接口，形成另一种风貌”的行为，成为adapter(配接器)。常将其归类为container adapter而非container

stack除了默认使用deque作为其底层容器之外，也可以使用双向开口的list，只需要在初始化stack时， 将list作为第二个参数即可。由于stack只能操作顶端的元素，因此其内部元素无法被访问，也不提供迭 代器。

##### queue

queue（队列）是一种先进先出（First In First Out）的数据结构，只有一个入口和一个出口，分别位于最底端和最顶端，出口元素外，没有其他方法可以获取到内部的其他元素，其结构图如下：



类似的，queue这种“先进先出”的数据结构很容易由双向开口的deque和list形成，只需要根据queue的 性质对应移除某些接口即可实现，queue的源码如下：

template <class T, class Sequence = *deque*<T> >

class queue

{

...

protected:

Sequence c;

public:

bool empty() { return c.*empty*(); }

*size\_type* size() const { return c.*size*(); }

*reference* front() const { return c.front(); }

*const\_reference* front() const { return c.front(); }

void push(const *value\_type*& x) { c.*push\_back*(x); }

void pop() { c.*pop\_front*(); }

}

从queue的数据结构可以看出，其所有操作都也都是是围绕Sequence完成，Sequence默认也是deque 数据结构。queue也是一类container adapter。

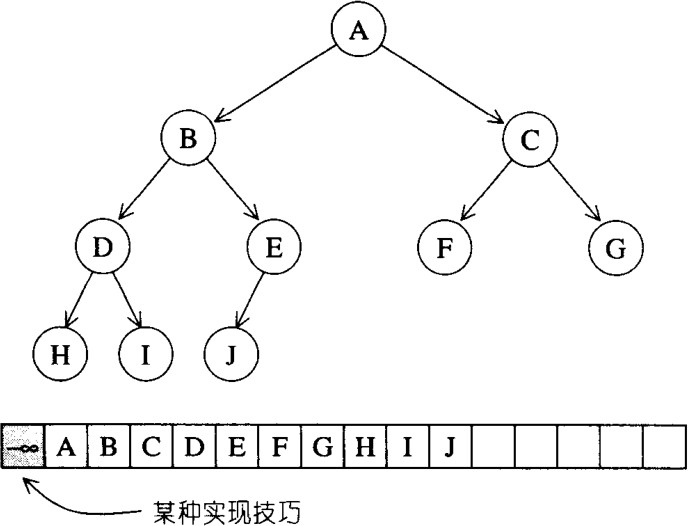
同样，queue也可以使用list作为底层容器，不具有遍历功能，没有迭代器。

《STL源码剖析》 侯捷

### 191、STL中的heap的实现

heap（堆）并不是STL的容器组件，是priority queue（优先队列）的底层实现机制，因为binary max heap（大根堆）总是最大值位于堆的根部，优先级最高。

binary heap本质是一种complete binary tree（完全二叉树），整棵binary tree除了最底层的叶节点之外，都是填满的，但是叶节点从左到右不会出现空隙，如下图所示就是一颗完全二叉树



完全二叉树内没有任何节点漏洞，是非常紧凑的，这样的一个好处是可以使用array来存储所有的节 点，因为当其中某个节点位于$i$处，其左节点必定位于$2i$处，右节点位于$2i+1$处，父节点位于

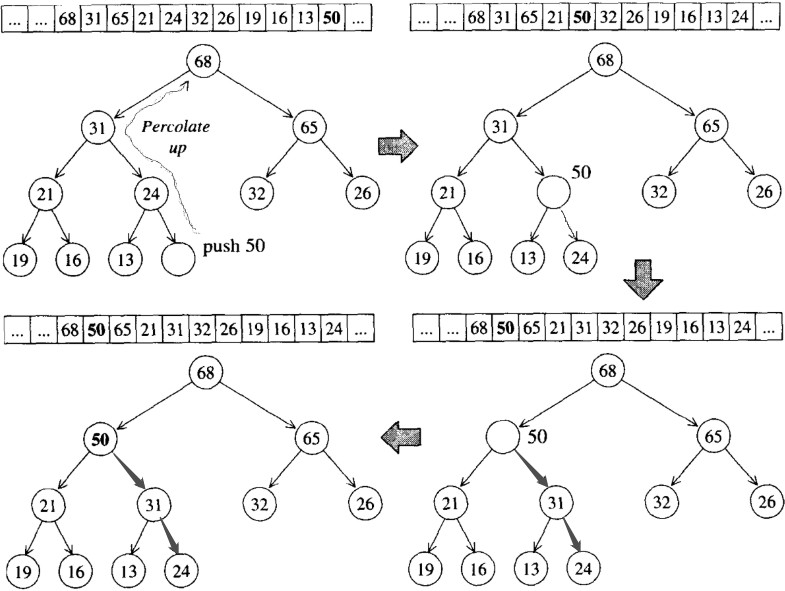
$i/2$（向下取整）处。这种以array表示tree的方式称为隐式表述法。

因此我们可以使用一个array和一组heap算法来实现max heap（每个节点的值大于等于其子节点的值）和min heap（每个节点的值小于等于其子节点的值）。由于array不能动态的改变空间大小，用vector代替array是一个不错的选择。

那heap算法有哪些？常见有的插入、弹出、排序和构造算法，下面一一进行描述。

##### push\_heap插入算法

由于完全二叉树的性质，新插入的元素一定是位于树的最底层作为叶子节点，并填补由左至右的第一个 空格。事实上，在刚执行插入操作时，新元素位于底层vector的end()处，之后是一个称为percolate up（上溯）的过程，举个例子如下图：

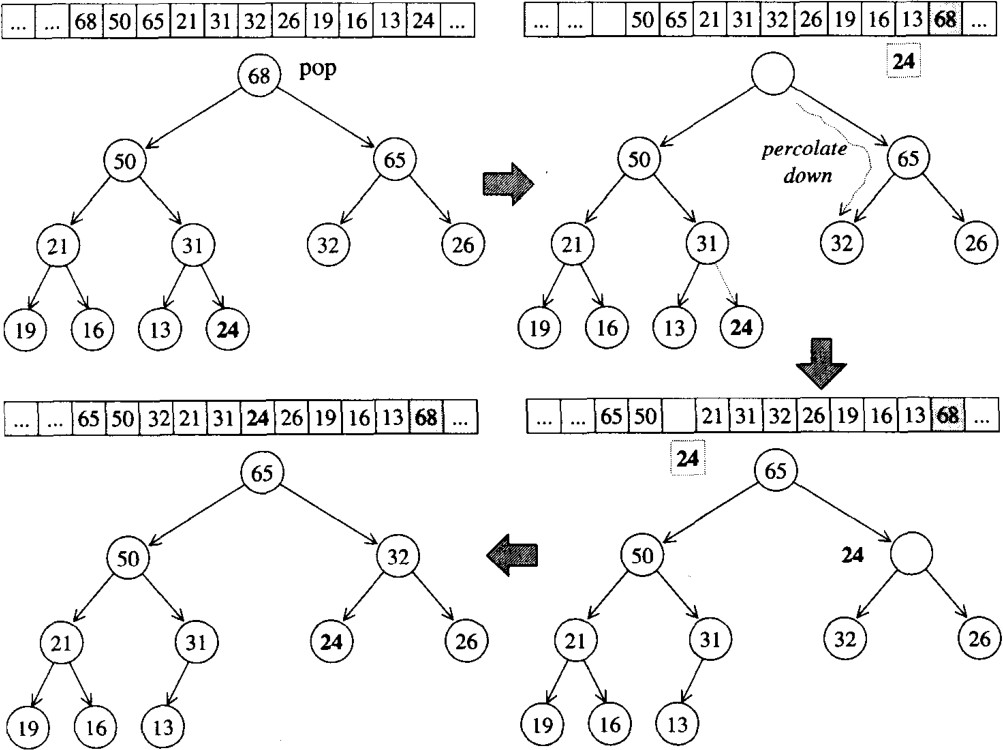


新元素50在插入堆中后，先放在vector的end()存着，之后执行上溯过程，调整其根结点的位置，以便 满足max heap的性质，如果了解大根堆的话，这个原理跟大根堆的调整过程是一样的。

##### pop\_heap算法

heap的pop操作实际弹出的是根节点吗，但在heap内部执行pop\_heap时，只是将其移动到vector的最 后位置，然后再为这个被挤走的元素找到一个合适的安放位置，使整颗树满足完全二叉树的条件。这个 被挤掉的元素首先会与根结点的两个子节点比较，并与较大的子节点更换位置，如此一直往下，直到这 个被挤掉的元素大于左右两个子节点，或者下放到叶节点为止，这个过程称为percolate down（下

溯）。举个例子：



根节点68被pop之后，移到了vector的最底部，将24挤出，24被迫从根节点开始与其子节点进行比较， 直到找到合适的位置安身，需要注意的是pop之后元素并没有被移走，如果要将其移走，可以使用pop\_back()。

##### sort算法

一言以蔽之，因为pop\_heap可以将当前heap中的最大值置于底层容器vector的末尾，heap范围减1， 那么不断的执行pop\_heap直到树为空，即可得到一个递增序列。

##### make\_heap算法

将一段数据转化为heap，一个一个数据插入，调用上面说的两种percolate算法即可。 代码实测：

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main()

{

*vector*<int> v = { 0,1,2,3,4,5,6 };

*make\_heap*(v.*begin*(), v.*end*()); //以vector为底层容器

for (auto i : v)

{

cout << i << " "; // 6 4 5 3 1 0 2

}

cout << *endl*;

v.*push\_back*(7);

*push\_heap*(v.*begin*(), v.*end*());

for (auto i : v)

{

cout << i << " "; // 7 6 5 4 1 0 2 3

}

cout << *endl*;

*pop\_heap*(v.*begin*(), v.*end*());

cout << v.*back*() << *endl*; // 7

v.*pop\_back*();

for (auto i : v)

{

cout << i << " "; // 6 4 5 3 1 0 2

}

cout << *endl*;

*sort\_heap*(v.*begin*(), v.*end*());

for (auto i : v)

{

cout << i << " "; // 0 1 2 3 4 5 6

}

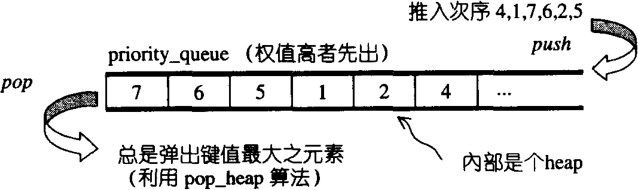
return 0;

}

《STL源码剖析》 侯捷

### 192、STL中的priority\_queue的实现

priority\_queue，优先队列，是一个拥有权值观念的queue，它跟queue一样是顶部入口，底部出口， 在插入元素时，元素并非按照插入次序排列，它会自动根据权值（通常是元素的实值）排列，权值最 高，排在最前面，如下图所示。



默认情况下，priority\_queue使用一个max-heap完成，底层容器使用的是一般为vector为底层容器， 堆heap为处理规则来管理底层容器实现 。priority\_queue的这种实现机制导致其不被归为容器，而是一种容器配接器。关键的源码如下：

template <class T, class Squence = *vector*<T>,

class Compare = *less*<typename *Sequence*::value\_tyoe> >

class priority\_queue {

...

protected:

*Sequence* c; // 底层容器

Compare comp; // 元素大小比较标准

public:

bool empty() const { return c.*empty*(); }

*size\_type* size() const { return c.*size*(); }

*const\_reference* top() const { return c.*front*() }

void push(const *value\_type*& x)

{

c.*push\_heap*(x);

*push\_heap*(c.*begin*(), c.*end*(), comp);

}

void pop()

{

*pop\_heap*(c.*begin*(), c.*end*(), comp);

c.*pop\_back*();

}

};

priority\_queue的所有元素，进出都有一定的规则，只有queue顶端的元素（权值最高者），才有机会 被外界取用，它没有遍历功能，也不提供迭代器

举个例子：

|  |
| --- |
| 1. #include <queue> 2. #include <iostream> 3. using namespace std; 4   5 int main() 6 {  7 int ia[9] = {0,4,1,2,3,6,5,8,7 };   1. priority\_queue<int> pq(ia, ia + 9); 2. cout << pq.size() <<endl; // 9 3. for(int i = 0; i < pq.size(); i++) 11 {   12 cout << pq.top() << " "; // 8 8 8 8 8 8 8 8 8  13 }   1. cout << endl; 2. while (!pq.empty())   16 {  17 cout << pq.top() << ' ';// 8 7 6 5 4 3 2 1 0  18 pq.pop(); 19 }  20 return 0; 21 }  22  23  24  25 |

《STL源码剖析》 侯捷

### 193、STL中set的实现？

STL中的容器可分为序列式容器（sequence）和关联式容器（associative），set属于关联式容器。

set的特性是，所有元素都会根据元素的值自动被排序（默认升序），set元素的键值就是实值，实值就 是键值，set不允许有两个相同的键值

set不允许迭代器修改元素的值，其迭代器是一种constance iterators

标准的STL set以RB-tree（红黑树）作为底层机制，几乎所有的set操作行为都是转调用RB-tree的操作行为，这里补充一下红黑树的特性：

每个节点不是红色就是黑色根结点为黑色

如果节点为红色，其子节点必为黑

任一节点至（NULL）树尾端的任何路径，所含的黑节点数量必相同

关于红黑树的具体操作过程，比较复杂读者可以翻阅《算法导论》详细了解。举个例子：

|  |
| --- |
| 1. #include <set> 2. #include <iostream> 3. using namespace std; 4   5  6 int main() 7 {  8 int i;  9 int ia[5] = { 1,2,3,4,5 };   1. set<int> s(ia, ia + 5); 2. cout << s.size() << endl; // 5 3. cout << s.count(3) << endl; // 1 4. cout << s.count(10) << endl; // 0 14 5. s.insert(3); //再插入一个3 6. cout << s.size() << endl; // 5 7. cout << s.count(3) << endl; // 1 18 8. s.erase(1); 9. cout << s.size() << endl; // 4 21 10. set<int>::iterator b = s.begin(); 11. set<int>::iterator e = s.end(); 24 for (; b != e; ++b)   25 cout << \*b << " "; // 2 3 4 5  26 cout << endl; 27   1. b = find(s.begin(), s.end(), 5); 2. if (b != s.end()) 3. cout << "5 found" << endl; // 5 found 31 4. b = s.find(2); 5. if (b != s.end()) 6. cout << "2 found" << endl; // 2 found 35 7. b = s.find(1); 8. if (b == s.end()) 9. cout << "1 not found" << endl; // 1 not found 10. return 0; 40 }   41  42  43  44 |

关联式容器尽量使用其自身提供的ﬁnd()函数查找指定的元素，效率更高，因为STL提供的ﬁnd()函数是 一种顺序搜索算法。

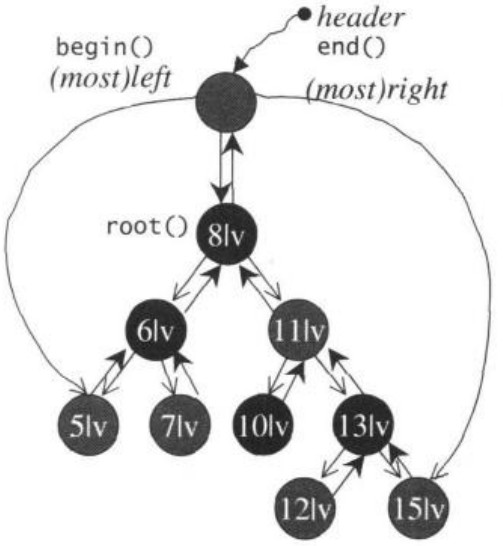
《STL源码剖析》 侯捷

### 194、STL中map的实现

map的特性是所有元素会根据键值进行自动排序。map中所有的元素都是pair，拥有键值(key)和实值(value)两个部分，并且不允许元素有相同的key

一旦map的key确定了，那么是无法修改的，但是可以修改这个key对应的value，因此map的迭代器既 不是constant iterator，也不是mutable iterator

标准STL map的底层机制是RB-tree（红黑树），另一种以hash table为底层机制实现的称为hash\_map。map的架构如下图所示



map的在构造时缺省采用递增排序key，也使用alloc配置器配置空间大小，需要注意的是在插入元素 时，调用的是红黑树中的insert\_unique()方法，而非insert\_euqal()（multimap使用）

举个例子：

#include <map>

#include <iostream>

#include <string>

using namespace std;

|  |
| --- |
| 6 int main() 7 {   1. map<string, int> maps; 2. //插入若干元素 3. maps["jack"] = 1; 4. maps["jane"] = 2; 5. maps["july"] = 3; 6. //以pair形式插入 7. pair<string, int> p("david", 4); 8. maps.insert(p); 9. //迭代输出元素 10. map<string, int>::iterator iter = maps.begin(); 11. for (; iter != maps.end(); ++iter) 19 { 12. cout << iter->first << " "; 13. cout << iter->second << "--"; //david 4--jack 1--jane 2--july 3-- 22 } 14. cout << endl; 15. //使用subscipt操作取实值 16. int num = maps["july"]; 17. cout << num << endl; // 3 18. //查找某key 19. iter = maps.find("jane"); 20. if(iter != maps.end()) 21. cout << iter->second << endl; // 2 22. //修改实值 23. iter->second = 100; 24. int num2 = maps["jane"]; // 100 25. cout << num2 << endl; 35   36 return 0; 37 }  38  39  40  41 |

需要注意的是subscript（下标）操作既可以作为左值运用（修改内容）也可以作为右值运用（获取实 值）。例如：

|  |
| --- |
| 1. maps["abc"] = 1; //左值运用 2. int num = masp["abd"]; //右值运用3   4  5  6 |

无论如何，subscript操作符都会先根据键值找出实值，源码如下：

|  |
| --- |
| 1 ...  2 T& operator[](const key\_type& k) 3 {  4 return (\*((insert(value\_type(k, T()))).first)).second; 5 }  6 ...  7 |

代码运行过程是：首先根据键值和实值做出一个元素，这个元素的实值未知，因此产生一个与实值型别 相同的临时对象替代：

|  |
| --- |
| 1 value\_type(k, T()); 2  3 |

再将这个对象插入到map中，并返回一个pair：

|  |
| --- |
| 1 pair<iterator,bool> insert(value\_type(k, T())); 2 |

pair第一个元素是迭代器，指向当前插入的新元素，如果插入成功返回true，此时对应左值运用，根据 键值插入实值。插入失败（重复插入）返回false，此时返回的是已经存在的元素，则可以取到它的实值

|  |
| --- |
| 1. (insert(value\_type(k, T()))).first; //迭代器 2. \*((insert(value\_type(k, T()))).first); //解引用 3. (\*((insert(value\_type(k, T()))).first)).second; //取出实值4 |

由于这个实值是以引用方式传递，因此作为左值或者右值都可以

《STL源码剖析》 侯捷

### 195、set和map的区别，multimap和multiset的区别

set只提供一种数据类型的接口，但是会将这一个元素分配到key和value上，而且它的compare\_function用的是 identity()函数，这个函数是输入什么输出什么，这样就实现了set机制，set 的key和value其实是一样的了。其实他保存的是两份元素，而不是只保存一份元素

map则提供两种数据类型的接口，分别放在key和value的位置上，他的比较function采用的是红黑树的comparefunction（），保存的确实是两份元素。

他们两个的insert都是采用红黑树的insert\_unique() 独一无二的插入 。

multimap和map的唯一区别就是：multimap调用的是红黑树的insert\_equal(),可以重复插入而map调 用的则是独一无二的插入insert\_unique()，multiset和set也一样，底层实现都是一样的，只是在插入的 时候调用的方法不一样。

##### 红黑树概念

面试时候现场写红黑树代码的概率几乎为0，但是红黑树一些基本概念还是需要掌握的。

1、它是二叉排序树（继承二叉排序树特显）：

若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值。若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值。

左、右子树也分别为二叉排序树。

2、它满足如下几点要求：

树中所有节点非红即黑。

根节点必为黑节点。

红节点的子节点必为黑（黑节点子节点可为黑）。从根到NULL的任何路径上黑结点数相同。

3、查找时间一定可以控制在O(logn)。

### 196、STL中unordered\_map和map的区别和应用场景

map支持键值的自动排序，底层机制是红黑树，红黑树的查询和维护时间复杂度均为$O(logn)$，但是 空间占用比较大，因为每个节点要保持父节点、孩子节点及颜色的信息

unordered\_map是C++ 11新添加的容器，底层机制是哈希表，通过hash函数计算元素位置，其查询时间复杂度为O(1)，维护时间与bucket桶所维护的list长度有关，但是建立hash表耗时较大

从两者的底层机制和特点可以看出：map适用于有序数据的应用场景，unordered\_map适用于高效查 询的应用场景

### 197、hashtable中解决冲突有哪些方法？

线性探测

使用hash函数计算出的位置如果已经有元素占用了，则向后依次寻找，找到表尾则回到表头，直到找到 一个空位

开链

每个表格维护一个list，如果hash函数计算出的格子相同，则按顺序存在这个list中

再散列

发生冲突时使用另一种hash函数再计算一个地址，直到不冲突

二次探测

使用hash函数计算出的位置如果已经有元素占用了，按照$1^2$、$2^2$、$3^2$...的步长依次寻找， 如果步长是随机数序列，则称之为伪随机探测

公共溢出区

一旦hash函数计算的结果相同，就放入公共溢出区

### static关键字用法

1. 先来介绍它的第一条也是最重要的一条：隐藏。（static函数，static变量均可） 当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性。

2. static的第二个作用是保持变量内容的持久。（static变量中的记忆功能和全局生存期）存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和static变量，只不过和全局变量比起来，static可以控制变量的可见范围，说到底static还是用来隐藏的。

3. static的第三个作用是默认初始化为0（static变量）

其实全局变量也具备这一属性，因为全局变量也存储在静态数据区。在静态数据区，内存中所有的字节 默认值都是0x00，某些时候这一特点可以减少程序员的工作量。

4. static的第四个作用：C++中的类成员声明static

1) 函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其 值在下次调用时仍维持上次的值；

2) 在模块内的static全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；

3) 在模块内的static函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模 块内；

4) 在类中的static成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

5) 在类中的static成员函数属于整个类所拥有，这个函数不接收this指针，因而只能访问类的static成员 变量。

类内：

6) static类对象必须要在类外进行初始化，static修饰的变量先于对象存在，所以static修饰的变量要在 类外初始化；

7) 由于static修饰的类成员属于类，不属于对象，因此static类成员函数是没有this指针的，this指针是 指向本对象的指针。正因为没有this指针，所以static类成员函数不能访问非static的类成员，只能访问static修饰的类成员；

8) static成员函数不能被virtual修饰，static成员不属于任何对象或实例，所以加上virtual没有任何实际 意义；静态成员函数没有this指针，虚函数的实现是为每一个对象分配一个vptr指针，而vptr是通过this 指针调用的，所以不能为virtual；虚函数的调用关系，this->vptr->ctable->virtual functionic全局变量一般来说存储在静态存储区。如果static全局变量未初始化，存储在bss段；初始化后存在data段。但static全局变量的存储位置和编译器优化相关，也可能存储在堆区。

### 类的对象存储空间

非静态成员的数据类型大小之和。

编译器加入的额外成员变量（如指向虚函数表的指针）。为了边缘对齐优化加入的padding。

空类(无非静态数据成员)的对象的size为1, 当作为基类时, size为0.

### 200、静态变量和全局变量的区别

两者作用域不同，全局变量的作用域是整个工程项目，静态全局变量作用域只在当前文件。只在定义该变量的源文件内有效， 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用， 因此可以避免在其它源文件中引起错误。所以把全局变量改变为静态变量后是改变了它的作用域， 限制了它的使用范围。

把局部变量改变为静态变量之后是改变了它的存储方式和生命期。 把全局变量改变为静态变量之后是改变了它的作用域，限制了使用范围。

<https://blog.csdn.net/hello_wordmy/article/details/78571218>

### 201、static存放在哪个区？

静态数据（即使是函数内部的静态局部变量）也存放在全局数据区，初始化不为0的在data段，初始化为0和未初始化的在bss段

### 类的对象存储空间

非静态成员的数据类型大小之和。

编译器加入的额外成员变量（如指向虚函数表的指针）。为了边缘对齐优化加入的padding。

空类(无非静态数据成员)的对象的size为1, 当作为基类时, size为0.

### 模板是在什么时候进行实例化

当使用了类模板实例的名字，并且上下文环境要求存在类的定义时。

对象类型是一个类模板实例，当对象被定义时。此点被称作类的实例化点。

一个指针或引用指向一个类模板实例，当检查这个指针或引用所指的对象时。

<https://www.cnblogs.com/live-in-city/archive/2013/11/13/3420577.html>

### 204、C++中的重载、重写（覆盖）和隐藏的区别

(1)重载：这个最好理解：相同函数名，函数参数类型和数量不同，注意重载并不关心还回值类型，此外重载必须是在同一个类中。

（2）重写、覆盖是同一个东西：都是在继承和派生的时候才可能出现的情况，重写（覆盖）是指在派生类中存在重新定义函数，其函数名、参数类型和数量必须与基类中的函数保持一致，只有函数体（也就是花括号里面的内容）不同。重写的函数必须要有virtual关键字修饰，否则不能称之为重写或者覆盖。

（3）隐藏也是在继承和派生的时候才可能出现的情况，隐藏是指派生类的函数屏蔽了其基类的函数，注意只要派生类中函数名与基类中函数名相同，基类函数都会被隐藏。注意：隐藏方法无法实现多态性。

[C++中重载(overload)、重写（override）和隐藏的区别\_c++中overried什么意思-CSDN博客](https://blog.csdn.net/wy749929317/article/details/123560096?ops_request_misc=%7B%22request%5Fid%22%3A%22171100404616800188571599%22%2C%22scm%22%3A%2220140713.130102334..%22%7D&request_id=171100404616800188571599&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~baidu_landing_v2~default-2-123560096-null-null.142^v99^pc_search_result_base5&utm_term=C++%E4%B8%AD%E7%9A%84%E9%87%8D%E8%BD%BD%E3%80%81%E9%87%8D%E5%86%99%EF%BC%88%E8%A6%86%E7%9B%96%EF%BC%89%E5%92%8C%E9%9A%90%E8%97%8F%E7%9A%84%E5%8C%BA%E5%88%AB&spm=1018.2226.3001.4187)

### 205、C++有哪几种的构造函数

****C++中的构造函数可以分为4类：****

****默认构造函数****

****初始化构造函数（有参数） 拷贝构造函数****

****移动构造函数（move和右值引用） 委托构造函数****

****转换构造函数****

**<https://blog.csdn.net/zxc024000/article/details/51153743>**

### 206、C++的多态如何实现

****C++的多态性，一言以蔽之就是：****

****在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据所指对象的实际类型来 调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的 函数。****

**<https://www.cnblogs.com/crbhf/p/14924629.html>**

### 207、内联函数和宏定义的区别

****主要区别****

****内联函数在编译时展开，宏在预编译时展开****

****内联函数直接嵌入到目标代码中，宏是简单的做文本替换内联函数有类型检测、语法判断等功能，而宏没有****

****内联函数是函数，宏不是****

****宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义****

****内联函数代码是被放到符号表中，使用时像宏一样展开，没有调用的开销，效率很高；****

****在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且 具有返回值。****

****内联函数本身是函数，强调函数特性，具有重载等功能。****

****内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员，进而提升效率。而 当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了。****

<https://zhuanlan.zhihu.com/p/682283354>

### 208、构造函数、析构函数、虚函数可否声明为内联函数

首先，将这些函数声明为内联函数，在语法上没有错误。因为inline同register一样，只是个建议，编译 器并不一定真正的内联。

register关键字：这个关键字请求编译器尽可能的将变量存在CPU内部寄存器中，而不是通过内存 寻址访问，以提高效率 构造函数和析构函数声明为内联函数是没有意义的

《Eﬀective C++》中所阐述的是：将构造函数和析构函数声明为inline是没有什么意义的，即编译器并不真正对声明为inline的构造和析构函数进行内联操作，因为编译器会在构造和析构函数中添加额外的 操作（申请/释放内存，构造/析构对象等），致使构造函数/析构函数并不像看上去的那么精简。其次， class中的函数默认是inline型的，编译器也只是有选择性的inline，将构造函数和析构函数声明为内联 函数是没有什么意义的。

<https://blog.csdn.net/SwordArcher/article/details/101099631>

### 209、auto、decltype和decltype(auto)的用法

基本用法与区别

auto 总是推导出“值类型”，绝不会是“引用”,如果有引用，auto会把引用去掉，推导出值类型； auto 可以附加上 const、volatile、\*、& 这样的类型修饰符，得到新的类型。

auto x = 10L; // auto推导为long，x是long

auto& x1 = x; // auto推导为long，x1是long&

auto\* x2 = &x; // auto推导为long，x2是long\*

const auto& x3 = x; // auto推导为long，x3是const long&

auto x4 = &x3; // auto推导为const long\*，x4是const long\*

auto 只能用于“初始化 “， “自动类型推导”要求必须从表达式推导，那在没有表达式的时候

就需要使用decltype + 表达式，自己解决类型推导问题。

decltype 的形式很像函数，后面的圆括号里就是可用于计算类型的表达式（和 sizeof 有点 类似），其他方面就和 auto 一样了，也能加上 const、\*、& 来修饰 ，它是用来直接声明变量的。

int x = 0; // 整型变量

decltype(x) x1; // 推导为int，x1是int

decltype(x)& x2 = x; // 推导为int，x2是int&，引用必须赋值

decltype(x)\* x3; // 推导为int，x3是int\*

decltype(&x) x4; // 推导为int\*，x4是int\*

decltype(&x)\* x5; // 推导为int\*，x5是int\*\*

decltype(x2) x6 = x2; // 推导为int&，x6是int&，引用必须赋值

decltype 不仅能够推导出值类型，还能够推导出引用类型，也就是表达式的“原始类 型”。

delctype的缺点是 是写起来略麻烦，特别在用于初始化的时候，表达式要重复两次 （左边的类型计算，右边的初始化）

当decltype(auto)被用于声明变量时，该变量必须立即初始化。假设该变量的初始化表达式为e，那么该变量的类型将被推导为decltype(e)。也就是说在推导变量类型时，先用初始化表达式替换decltype(auto)当中的auto，然后再根据decltype的语法规则来确定变量的类型。

int x = 0; // 整型变量

decltype(auto) x2 = &x; // 推导为int\*

decltype(auto) x3 = x1; // 推导为int&

常见用法

auto的 range-based for 用法

不需要关心容器元素类型、迭 代器返回值和首末位置，就能非常轻松地完成遍历操作

vector<int> v = {2,3,5,7,11}; // vector顺序容器

for(const auto& i : v) { // 常引用方式访问元素，避免拷贝代价

cout << i << ","; // 常引用不会改变元素的值

}

for(auto& i : v) { // 引用方式访问元素

i++; // 可以改变元素的值

cout << i << ",";

}

decltype 简化函数指针

// UNIX信号函数的原型，看着就让人晕

void (\*signal(int signo, void (\*func)(int)))(int)

// 使用decltype可以轻松得到函数指针类型

using sig\_func\_ptr\_t = decltype(&signal) ;

decltype在类内部实现参数推导

在定义类的时候，auto 被禁用了。可以用decltype获取表达式类型，配合using生成别名

class DemoClass final {

public:

using set\_type = std::set<int>; // 集合类型别名

private:

set\_type m\_set; // 使用别名定义成员变量

// 使用decltype计算表达式的类型，定义别名

using iter\_type = decltype(m\_set.begin());

iter\_type m\_pos; // 类型别名定义成员变量

};

<https://blog.csdn.net/qq_42604176/article/details/120615736>

### 210、public，protected和private访问和继承权限/public/protected/private

### 的区别

访问限定符

C++ 通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限（也称为可见性），分别表示：公有的、受保护的、私有的。

**class** **Base** {**public:**

*// 公有成员***protected:**

*// 受保护成员***private:**

*// 私有成员*}

访问权限

所谓访问权限，就是能不能使用该类中的成员。

一般地，在类的内部，无论成员被声明为哪种，都是可以互相访问的；但在类的外部，如通过类的对象，则只能访问 public 属性的成员，不能访问protected、private属性的成员。

对象（object）是类（class）的一个实例（instance）。

具体如下：

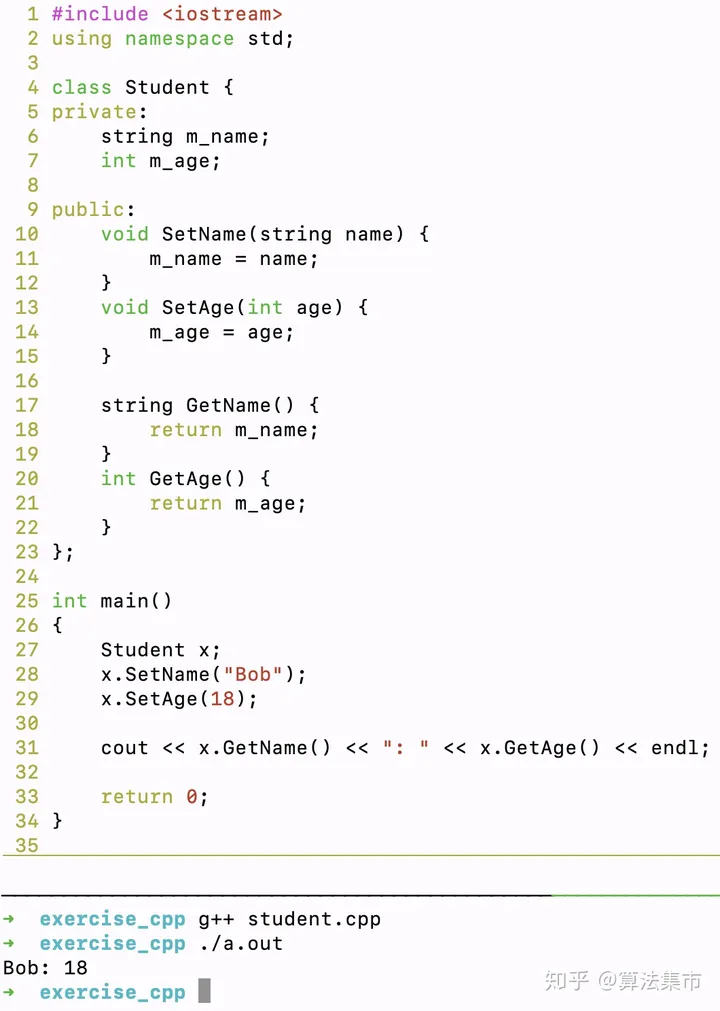
* public：可以被该类中的函数、子类的函数、友元函数访问，也可以由该类的对象访问；
* protected：可以被该类中的函数、子类的函数、友元函数访问，但不可以由该类的对象访问；
* private：可以被该类中的函数、友元函数访问，但不可以由子类的函数、该类的对象、访问。

private 关键字的作用在于更好地隐藏类的内部实现。

根据C++的软件设计规范，在实际项目开发中，类的成员变量以及只在类内部使用的成员函数，都建议声明为 private，而将允许通过对象调用的成员函数声明的 public。

成员变量声明为private，如何给它们赋值，以及获取它们的值呢？

通常需要添加两个public属性的成员函数，一个用来设置成员变量的值，一个用来读取成员变量的值。



* 如果声明不写 public、protected、private，则默认为 private；
* 声明public、protected、private的顺序可以任意；
* 在一个类中，public、protected、private 可以出现多次，每个限定符的有效范围到出现另一个限定符或类结束为止。但为了使程序清晰，应该使每种限定符只出现一次。
* <https://zhuanlan.zhihu.com/p/157363590>

### 211、判断当前机器的大小端存储

什么是大小端呢？

大端字节序存储：把一个数的低字节序的内容放在高地址处，相反，把高字节序的内容存放在低地址处，这种存储方式称为大端

小端字节序存储：把一个数的低字节序的内容存放在低地址处，把高字节序的内容存放在高地址处，这种存储方式称为小端

那么如何写一个程序来判断大小端呢，具体代码如下：

方法1：

判断当前机器的大小端存储

#include <stdio.h>

#include <stdlib.h>

int check\_sys()

{

union

{

int i;

char c;

}un;

un.i = 1;

return un.c;

}

int main()

{

int ret = check\_sys();

if (ret == 1)

{

printf("小端\n");

}

else

{

printf("大端\n");

}

return 0;

}

方法2：

#include <stdio.h>

#include <stdlib.h>

int check\_sys()

{

int a = 1;

return \*(char\*)&a;//返回1表示小端，返回0表示大端

}

int main()

{

if (check\_sys() == 1)

{

printf("小端\n");

}

else

{

printf("大端\n");

}

system("pause");

return 0;

}

<https://blog.csdn.net/qq_42270373/article/details/82432917>

### 212、C++的异常处理的方法

1、程序运行时常会碰到一些异常情况，例如：

做除法的时候除数为 0；

用户输入年龄时输入了一个负数；

用 new 运算符动态分配空间时，空间不够导致无法分配；

访问数组元素时，下标越界；打开文件读取时，文件不存在。

这些异常情况，如果不能发现并加以处理，很可能会导致程序崩溃。  
  
所谓“处理”，可以是给出错误提示信息，然后让程序沿一条不会出错的路径继续执行；也可能是不得不结束程序，但在结束前做一些必要的工作，如将内存中的数据写入文件、关闭打开的文件、释放动态分配的内存空间等。  
  
一发现异常情况就立即处理未必妥当，因为在一个函数执行过程中发生的异常，在有的情况下由该函数的调用者决定如何处理更加合适。尤其像库函数这类提供给程序员调用，用以完成与具体应用无关的通用功能的函数，执行过程中贸然对异常进行处理，未必符合调用它的程序的需要。  
  
此外，将异常分散在各处进行处理不利于代码的维护，尤其是对于在不同地方发生的同一种异常，都要编写相同的处理代码也是一种不必要的重复和冗余。如果能在发生各种异常时让程序都执行到同一个地方，这个地方能够对异常进行集中处理，则程序就会更容易编写、维护。  
  
鉴于上述原因，[C++](https://c.biancheng.net/cplus/" \t "https://c.biancheng.net/view/_blank) 引入了异常处理机制。其基本思想是：函数 A 在执行过程中发现异常时可以不加处理，而只是“拋出一个异常”给 A 的调用者，假定为函数 B。  
  
拋出异常而不加处理会导致函数 A 立即中止，在这种情况下，函数 B 可以选择捕获 A 拋出的异常进行处理，也可以选择置之不理。如果置之不理，这个异常就会被拋给 B 的调用者，以此类推。  
  
如果一层层的函数都不处理异常，异常最终会被拋给最外层的 main 函数。main 函数应该处理异常。如果main函数也不处理异常，那么程序就会立即异常地中止。

2、C++异常处理语法

C++ 通过 throw 语句和 try...catch 语句实现对异常的处理。throw 语句的语法如下：

throw  表达式;

该语句拋出一个异常。异常是一个表达式，其值的类型可以是基本类型，也可以是类。  
  
try...catch 语句的语法如下：

try {  
    语句组  
}  
catch(异常类型) {  
    异常处理代码  
}  
...  
catch(异常类型) {  
    异常处理代码  
}

catch 可以有多个，但至少要有一个。  
  
不妨把 try 和其后{}中的内容称作“try块”，把 catch 和其后{}中的内容称作“catch块”。  
  
try...catch 语句的执行过程是：

* 执行 try 块中的语句，如果执行的过程中没有异常拋出，那么执行完后就执行最后一个 catch 块后面的语句，所有 catch 块中的语句都不会被执行；
* 如果 try 块执行的过程中拋出了异常，那么拋出异常后立即跳转到第一个“异常类型”和拋出的异常类型匹配的 catch 块中执行（称作异常被该 catch 块“捕获”），执行完后再跳转到最后一个 catch 块后面继续执行。

例如下面的程序：

1. #include <iostream>
2. **u[sin](https://c.biancheng.net/ref/sin.html" \t "https://c.biancheng.net/view/_blank)g** **namespace** std;
3. int main()
4. {
5. double m ,n;
6. cin >> m >> n;
7. **try** {
8. cout << "before dividing." << endl;
9. **if**( n == 0)
10. **throw** -1; //抛出int类型异常
11. **else**
12. cout << m / n << endl;
13. cout << "after dividing." << endl;
14. }
15. **catch**(double d) {
16. cout << "catch(double) " << d << endl;
17. }
18. **catch**(int e) {
19. cout << "catch(int) " << e << endl;
20. }
21. cout << "finished" << endl;
22. **return** 0;
23. }
24. 程序的运行结果如下：  
    9 6↙  
    before dividing.  
    1.5  
    after dividing.  
    finished  
      
    说明当 n 不为 0 时，try 块中不会拋出异常。因此程序在 try 块正常执行完后，越过所有的 catch 块继续执行，catch 块一个也不会执行。  
      
    程序的运行结果也可能如下：  
    9 0↙  
    before dividing.  
    catch\(int) -1  
    finished  
      
    当 n 为 0 时，try 块中会拋出一个整型异常。拋出异常后，try 块立即停止执行。该整型异常会被类型匹配的第一个 catch 块捕获，即进入catch(int e)块执行，该 catch 块执行完毕后，程序继续往后执行，直到正常结束。  
      
    如果拋出的异常没有被 catch 块捕获，例如，将catch(int e)，改为catch(char e)，当输入的 n 为 0 时，拋出的整型异常就没有 catch 块能捕获，这个异常也就得不到处理，那么程序就会立即中止，try...catch 后面的内容都不会被执行。

3、能够捕获任何异常的 catch 语句

如果希望不论拋出哪种类型的异常都能捕获，可以编写如下 catch 块：

catch(...) {  
    ...  
}

这样的 catch 块能够捕获任何还没有被捕获的异常。例如下面的程序：

1. #include <iostream>
2. **using** **namespace** std;
3. int main()
4. {
5. double m, n;
6. cin >> m >> n;
7. **try** {
8. cout << "before dividing." << endl;
9. **if** (n == 0)
10. **throw** - 1; //抛出整型异常
11. **else** **if** (m == 0)
12. **throw** - 1.0; //拋出 double 型异常
13. **else**
14. cout << m / n << endl;
15. cout << "after dividing." << endl;
16. }
17. **catch** (double d) {
18. cout << "catch (double)" << d << endl;
19. }
20. **catch** (...) {
21. cout << "catch (...)" << endl;
22. }
23. cout << "finished" << endl;
24. **return** 0;
25. }

程序的运行结果如下：  
9 0↙  
before dividing.  
catch (...)  
finished  
  
当 n 为 0 时，拋出的整型异常被catchy(...)捕获。  
  
程序的运行结果也可能如下：  
0 6↙  
before dividing.  
catch (double) -1  
finished  
  
当 m 为 0 时，拋出一个 double 类型的异常。虽然catch (double)和catch(...)都能匹配该异常，但是catch(double)是第一个能匹配的 catch 块，因此会执行它，而不会执行catch(...)块。  
  
由于catch(...)能匹配任何类型的异常，它后面的 catch 块实际上就不起作用，因此不要将它写在其他 catch 块前面。

异常再抛出：

如果一个函数在执行过程中拋出的异常在本函数内就被 catch 块捕获并处理，那么该异常就不会拋给这个函数的调用者（也称为“上一层的函数”）；如果异常在本函数中没有被处理，则它就会被拋给上一层的函数。例如下面的程序：

1. #include <iostream>
2. #include <string>
3. **using** **namespace** std;
4. **class** CException
5. {
6. **public**:
7. string msg;
8. CException(string s) : msg(s) {}
9. };
10. double Devide(double x, double y)
11. {
12. **if** (y == 0)
13. **throw** CException("devided by zero");
14. cout << "in Devide" << endl;
15. **return** x / y;
16. }
17. int CountTax(int salary)
18. {
19. **try** {
20. **if** (salary < 0)
21. **throw** - 1;
22. cout << "counting tax" << endl;
23. }
24. **catch** (int) {
25. cout << "salary < 0" << endl;
26. }
27. cout << "tax counted" << endl;
28. **return** salary \* 0.15;
29. }
30. int main()
31. {
32. double f = 1.2;
33. **try** {
34. CountTax(-1);
35. f = Devide(3, 0);
36. cout << "end of try block" << endl;
37. }
38. **catch** (CException e) {
39. cout << e.msg << endl;
40. }
41. cout << "f = " << f << endl;
42. cout << "finished" << endl;
43. **return** 0;
44. }

程序的输出结果如下：  
salary < 0  
tax counted  
devided by zero  
f=1.2  
finished  
  
CountTa 函数拋出异常后自行处理，这个异常就不会继续被拋给调用者，即 main 函数。因此在 main 函数的 try 块中，CountTax 之后的语句还能正常执行，即会执行f = Devide(3, 0);。  
  
第 35 行，Devide 函数拋出了异常却不处理，该异常就会被拋给 Devide 函数的调用者，即 main 函数。拋出此异常后，Devide 函数立即结束，第 14 行不会被执行，函数也不会返回一个值，这从第 35 行 f 的值不会被修改可以看出。  
  
Devide 函数中拋出的异常被 main 函数中类型匹配的 catch 块捕获。第 38 行中的 e 对象是用复制构造函数初始化的。  
  
如果拋出的异常是派生类的对象，而 catch 块的异常类型是基类，那么这两者也能够匹配，因为派生类对象也是基类对象。  
  
虽然函数也可以通过返回值或者传引用的参数通知调用者发生了异常，但采用这种方式的话，每次调用函数时都要判断是否发生了异常，这在函数被多处调用时比较麻烦。有了异常处理机制，可以将多处函数调用都写在一个 try 块中，任何一处调用发生异常都会被匹配的 catch 块捕获并处理，也就不需要每次调用后都判断是否发生了异常。  
  
有时，虽然在函数中对异常进行了处理，但是还是希望能够通知调用者，以便让调用者知道发生了异常，从而可以作进一步的处理。在 catch 块中拋出异常可以满足这种需要。例如：

1. #include <iostream>
2. #include <string>
3. **using** **namespace** std;
4. int CountTax(int salary)
5. {
6. **try** {
7. **if**( salary < 0 )
8. **throw** string("zero salary");
9. cout << "counting tax" << endl;
10. }
11. **catch** (string s ) {
12. cout << "CountTax error : " << s << endl;
13. **throw**; //继续抛出捕获的异常
14. }
15. cout << "tax counted" << endl;
16. **return** salary \* 0.15;
17. }
18. int main()
19. {
20. double f = 1.2;
21. **try** {
22. CountTax(-1);
23. cout << "end of try block" << endl;
24. }
25. **catch**(string s) {
26. cout << s << endl;
27. }
28. cout << "finished" << endl;
29. **return** 0;
30. }

程序的输出结果如下：  
CountTax error:zero salary  
zero salary  
finished  
  
第 14 行的throw;没有指明拋出什么样的异常，因此拋出的就是 catch 块捕获到的异常，即 string("zero salary")。这个异常会被 main 函数中的 catch 块捕获。

函数的异常声明列表：

为了增强程序的可读性和可维护性，使程序员在使用一个函数时就能看出这个函数可能会拋出哪些异常，C++ 允许在函数声明和定义时，加上它所能拋出的异常的列表，具体写法如下：

void func() throw (int, double, A, B, C);

或

void func() throw (int, double, A, B, C){...}

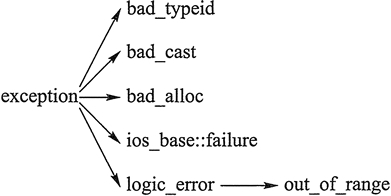
上面的写法表明 func 可能拋出 int 型、double 型以及 A、B、C 三种类型的异常。异常声明列表可以在函数声明时写，也可以在函数定义时写。如果两处都写，则两处应一致。  
  
如果异常声明列表如下编写：

void func() throw ();

则说明 func 函数不会拋出任何异常。  
  
一个函数如果不交待能拋出哪些类型的异常，就可以拋出任何类型的异常。  
  
函数如果拋出了其异常声明列表中没有的异常，在编译时不会引发错误，但在运行时， Dev C++ 编译出来的程序会出错；用 Visual Studio 2010 编译出来的程序则不会出错，异常声明列表不起实际作用。

C++异常标准类：

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来的。常用的几个异常类如图 1 所示。

  
图1：常用的异常类

bad\_typeid、bad\_cast、bad\_alloc、ios\_base::failure、out\_of\_range 都是 exception 类的派生类。C++ 程序在碰到某些异常时，即使程序中没有写 throw 语句，也会自动拋出上述异常类的对象。这些异常类还都有名为 what 的成员函数，返回字符串形式的异常描述信息。使用这些异常类需要包含头文件 stdexcept。  
  
下面分别介绍以上几个异常类。本节程序的输出以 Visual Studio 2010为准，Dev C++ 编译的程序输出有所不同。

#### 1) bad\_typeid

使用 typeid 运算符时，如果其操作数是一个多态类的[指针](https://c.biancheng.net/c/80/" \t "https://c.biancheng.net/view/_blank)，而该指针的值为 NULL，则会拋出此异常。

#### 2) bad\_cast

在用 dynamic\_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会拋出此异常。程序示例如下：

1. #include <iostream>
2. #include <stdexcept>
3. **using** **namespace** std;
4. **class** Base
5. {
6. **virtual** void func() {}
7. };
8. **class** Derived : **public** Base
9. {
10. **public**:
11. void Print() {}
12. };
13. void PrintObj(Base & b)
14. {
15. **try** {
16. Derived & rd = **dynamic\_cast** <Derived &>(b);
17. //此转换若不安全，会拋出 bad\_cast 异常
18. rd.Print();
19. }
20. **catch** (bad\_cast & e) {
21. cerr << e.what() << endl;
22. }
23. }
24. int main()
25. {
26. Base b;
27. PrintObj(b);
28. **return** 0;
29. }

程序的输出结果如下：  
Bad dynamic\_cast!  
  
在 PrintObj 函数中，通过 dynamic\_cast 检测 b 是否引用的是一个 Derived 对象，如果是，就调用其 Print 成员函数；如果不是，就拋出异常，不会调用 Derived::Print。

#### 3) bad\_alloc

在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常。程序示例如下：

1. #include <iostream>
2. #include <stdexcept>
3. **using** **namespace** std;
4. int main()
5. {
6. **try** {
7. char \* p = **new** char[0x7fffffff]; //无法分配这么多空间，会抛出异常
8. }
9. **catch** (bad\_alloc & e) {
10. cerr << e.what() << endl;
11. }
12. **return** 0;
13. }

程序的输出结果如下：  
bad allocation  
ios\_base::failure  
  
在默认状态下，输入输出流对象不会拋出此异常。如果用流对象的 exceptions 成员函数设置了一些标志位，则在出现打开文件出错、读到输入流的文件尾等情况时会拋出此异常。此处不再赘述。

#### 4) out\_of\_range

用 vector 或 string 的 at 成员函数根据下标访问元素时，如果下标越界，则会拋出此异常。例如：

1. #include <iostream>
2. #include <stdexcept>
3. #include <vector>
4. #include <string>
5. **using** **namespace** std;
6. int main()
7. {
8. vector<int> v(10);
9. **try** {
10. v.at(100) = 100; //拋出 out\_of\_range 异常
11. }
12. **catch** (out\_of\_range & e) {
13. cerr << e.what() << endl;
14. }
15. string s = "hello";
16. **try** {
17. char c = s.at(100); //拋出 out\_of\_range 异常
18. }
19. **catch** (out\_of\_range & e) {
20. cerr << e.what() << endl;
21. }
22. **return** 0;
23. }

程序的输出结果如下：  
invalid vector <T> subscript  
invalid string position  
  
如果将v.at(100)换成v[100]，将s.at(100)换成s[100]，程序就不会引发异常（但可能导致程序崩溃）。因为 at 成员函数会检测下标越界并拋出异常，而 operator[] 则不会。operator [] 相比 at 的好处就是不用判断下标是否越界，因此执行速度更快。

### 静态变量什么时候初始化

1) 初始化只有一次，但是可以多次赋值，在主程序之前，编译器已经为其分配好了内存。

2) 静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配 好了内存，但在C和C++中静态局部变量的初始化节点又有点不太一样。在C中，初始化发生在代码执行 之前，编译阶段分配好内存之后，就会进行初始化，所以我们看到在C语言中无法使用变量对静态局部 变量进行初始化，在程序运行结束，变量所处的全局内存会被全部回收。

3) 而在C++中，初始化时在执行相关代码时才会进行初始化，主要是由于C++引入对象后，要进行初始 化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的 特定操作，并非简单地分配内存。所以C++标准定为全局或静态对象是有首次用到时才会进行构造，并 通过atexit()来管理。在程序结束，按照构造顺序反方向进行逐个析构。所以在C++中是可以使用变量对 静态局部变量进行初始化的。

### 214、const关键字

1、const修饰普通变量

用const修饰普通变量实际上就是定义了一个常量，以下两种定义形式在本质上是一样的。它的含义是：const修饰的类型为TYPE的变量value是不可变的。 一般对于const变量名都是全大写的。

TYPE const ValueName = value;

const TYPE ValueName = value;

如果我们强行修改const定义的常量，就会报错。

实际上，虽然不能直接对常变量进行修改，但是我们可以通过指针对常变量进行修改：

所以当指针指向const常变量时，为避免通过指针修改const常变量，我们应当相应地用const指针指向const常变量。

const int AMOUNT = 100；

int const \*p = &AMOUNT；

用const定义的常量有什么作用呢？

假设我们在代码中经常使用到一个数字100，我们可以定义一个常量，让常量的值100：

const int AMOUNT = 100;

如果我们没定义常量，我们需要将100修改为200，需要将代码出现的所有100依次修改为200，而如果我们定义了常量，我们只需要修改常变量的值为200即可，这样就增加了代码的可维护性。

如果将const改为外部连接,作用于扩大至全局,编译时会分配内存,并且可以不进行初始化,仅仅作为声明,编译器认为在程序其他地方进行了定义。

extend const int ValueName = value;

2、const修饰指针

（1）const修饰p:

int i = 10;

int\* const p = &i;

const修饰指针p表示指针不可修改，即一旦得到了某个变量的地址，不能再指向其它变量：

int i = 10;

int \* const p = &i;

p++;//p指针指向下一个元素，错误

虽然指针不可修改，但是可以修改指针所指向的变量的值：

int i = 10;

int \* const p = &i;

\*p = 26;//没问题

（2）const修饰\*p：

int i = 10;

const int \* p = &i;

const修饰\*p表示不可通过指针修改其所指变量的值：

int i = 10;

const int\* p = &i;

\*p = 26;//通过指针修改其所指变量的值,错误

虽然不可以通过指针修改指针所指变量的值，但是变量i本身可以做任何变化，如：

int i = 10;

const int\* p = &i;

i=26;

i++;

p也可以变：

p = &j;

（3）const修饰p和\*p

const int\* const p;

const修饰p和\*p表示指针不能变，指针所指的变量也不能变。

注意以下的区别：

int i;

const int\* p1 = &i;//不能通过指针修改

int const\* p2 = &i;//不能通过指针修改

int \*const p3 = &i;//指针不能修改

/\*判断哪个被const了的标志是const在\*的前面还是后面\*/

4、const修饰数组

数组变量实际上就是const的指针，所以不能直接赋值:

int a[]

//a--->int\* const a;

const修饰数组表明数组的每个元素都是const int，无法修改，所以必须通过初始化进行赋值，否则写出来后就无法进行赋值了。

const int a[]

//a---->const int \* const a;

5、const修饰函数形参

（1）const修饰普通形参变量

void function(const int Var);

这表示形参不会发生改变，但实际上这是没有意义的，因为我们通常是为了保证外部的实参数据不发生变化，这里形参实际上是实参的拷贝，实参本来就不会发生变化。

（2）const修饰指针形参

void function(const char\* Var);

我们把外部实参的地址赋值给用const修饰的指针形参，这样我们就无法通过指针修改其所指的外部实参，保护了数据的安全性。

但如果是这种const指针形参就毫无意义：

void function(char\* const Var);

因为这表示指针形参不会改变，但是我们依然可以通过指针修改传过来的外部实参，无法保证外部数据的安全性。

（3）const修饰引用形参

void function(const Type& Var); //引用参数在函数内不可以改变

参数为引用，将外部实参传递给引用形参，传递的是外部实参本身，无需进行拷贝，增加了效率，同时参数是const引用，无法通过引用修改实参，保证了外部数据的安全性。

注：

如果函数参数是非const的引用/指针，它就不能接收const变量（地址），因为会造成权限的放大，会报错，它只能接收非const变量（地址），而如果函数参数是const的引用/指针，它既可以接收const变量（地址），也可以接收非const变量（地址），这是权限的缩小，没问题，如果函数是const普通变量，那么它可以接收const变量，也可以接收非const变量，因为不会造成权限的放大。

6、const 修饰函数返回值

const修饰函数返回值其实用的并不是很多。

（1）const修饰普通类型的返回值

const int fun1();

这个实际上是毫无意义的，因为返回的实际上是临时变量，临时变量本身就具有常性。

（2）const修饰指针类型的返回值

第一种情况是const修饰\*p：

const int \* fun2();

const int \*p = fun2();//调用正常

int\* p2 = fun2();//调用失败

这里简单分析下第三句代码调用失败的原因，因为fun2()实际上是一个const int\*指针，我们无法通过指针修改其所指变量，而把fun2()这个const int\*指针赋值给p2后，p2是非const指针，可以通过指针修改其所指变量，这就造成了指针所指变量的访问权限的放大，因此调用失败。

第二种情况就是const修饰p

int\* const fun3();

int \* const p = fun3();//调用正常

int \*p2 = fun3();//调用正常

看第三句代码，为什么这种情况，返回的int\* const指针可以赋值给非const指针呢？

我们可以把fun3()看作一个int \* const指针，尽管该指针不可以修改，但是我们可以通过该指针修改其所指变量，赋值给p2后，我们依然可以通过指针修改其所指变量，对指针所指变量的访问权限没有发生变化，这样的赋值当然没有问题了。

7、const修饰成员变量

const修饰类的成员变量，表示成员常量，不能被修改，同时它只能在初始化列表中赋值。

class A

{

…

const int nValue;//成员常量不能被修改

…

A(int x): nValue(x) { };//只能在初始化列表中赋值

}

8、const修饰成员函数

将const修饰的“成员函数”称之为const成员函数。this指针的类型是：类类型\* const this，即成员函数中，不能修改this指针，但是其所指向对象的成员变量可以被修改，const修饰类成员函数，实际上就是将this限定为：const 类类型 \* const this，表明在该成员函数中不能对调用对象的任何成员进行修改，起到保护对象的作用。

格式： 将const关键字放在函数的括号后面

class Date

{

public:

void Display()const

{

cout<<\_year<<endl;

}

//该成员函数实际上是这样

/\*

void Display(const Date \*const this)

{

cout<<this->\_year<<endl;

}\*/

private:

int \_year;

int \_month;

int \_day;

};

关于const成员函数，以下几点需要我们注意：

就像尽可能地将函数形参中的引用和指针声明为const一样，只要成员函数不修改调用对象，就应该将其声明为const，这样既可以避免调用对象被修改，而且普通对象和const对象都可以调用（不会造成读写权限的放大）。

前面提到过，读写权限只能缩小，不能放大，所以，const对象只能调用const成员函数，不可以调用非const成员函数，因为const对象不可以被修改，如果const对象调用非const成员函数，非const成员函数可以修改调用它的const对象，这是权限的放大，会报错。非const对象可以调用非const成员函数，也可以调用const成员函数，因为这是权限的缩小。

const修饰类的成员函数，则该成员函数不能调用类中任何非const成员函数，因为非const成员函数可能会对const对象造成修改，引起权限的放大。

const成员函数能够访问对象的const成员，而其他成员函数不可以。

<https://blog.csdn.net/weixin_44049823/article/details/128735316>

### 215、指针和const的用法

1) 当const修饰指针时，由于const的位置不同，它的修饰对象会有所不同。

2) int \*const p2中const修饰p2的值,所以理解为p2的值不可以改变，即p2只能指向固定的一个变量地址，但可以通过\*p2读写这个变量的值。顶层指针表示指针本身是一个常量

3) int const \*p1或者const int \*p1两种情况中const修饰\*p1，所以理解为\*p1的值不可以改变，即不可以给\*p1赋值改变p1指向变量的值，但可以通过给p赋值不同的地址改变这个指针指向。

底层指针表示指针所指向的变量是一个常量。

### 216、形参与实参的区别？

1) 形参变量只有在被调用时才分配内存单元，在调用结束时， 即刻释放所分配的内存单元。因此，形参只有在函数内部有效。 函数调用结束返回主调函数后则不能再使用该形参变量。

2) 实参可以是常量、变量、表达式、函数等， 无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值， 以便把这些值传送给形参。 因此应预先用赋值，输入等办法使实参获得确定值， 会产生一个临时变量。

3) 实参和形参在数量上，类型上，顺序上应严格一致， 否则会发生“类型不匹配”的错误。

4) 函数调用中发生的数据传送是单向的。 即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。 因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

5) 当形参和实参不是指针类型时，在该函数运行时，形参和实参是不同的变量，他们在内存中位于不 同的位置，形参将实参的内容复制一份，在该函数运行结束的时候形参被释放，而实参内容不会改变。

1. 值传递、指针传递、引用传递的区别和效率

1) 值传递：有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象 或是大的结构体对象，将耗费一定的时间和空间。（传值）

2) 指针传递：同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为4字节的地 址。（传值，传递的是地址值）

3) 引用传递：同样有上述的数据拷贝过程，但其是针对地址的，相当于为该数据所在的地址起了一个 别名。（传地址）

4) 效率上讲，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、清晰。

### 217、什么是类的继承？

1) 类与类之间的关系

has-A包含关系，用以描述一个类由多个部件类构成，实现has-A关系用类的成员属性表示，即一个类的 成员属性是另一个已经定义好的类；

use-A，一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来 实现；

is-A，继承关系，关系具有传递性；

2) 继承的相关概念

所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称 为子类或者派生类，被继承的类称为父类或者基类；

3) 继承的特点

子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，子类对象可以当做父类对象使用；

4) 继承中的访问控制

public、protected、private

5) 继承中的构造和析构函数

6) 继承中的兼容性原则

### 218、什么是内存池，如何实现

内存池（Memory Pool） 是一种内存分配方式。通常我们习惯直接使用new、malloc 等申请内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性 能。内存池则是在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备 用。当有新的内存需求时，就从内存池中分出一部分内存块， 若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升。

<https://zhuanlan.zhihu.com/p/375537583>

### 219、类成员初始化方式？构造函数的执行顺序 ？为什么用成员初始化列表会快一些？

1、类成员初始化方式

赋值初始化，通过在函数体内进行赋值初始化：

对于在函数体中初始化,是在所有的数据成员被分配内存空间后才进行的。

class G{

private:

int n;

public:

G(int x){

this->n = x;

cout << "G()" << endl;

}

~G(){

cout << "~G()" << endl;

}

};

列表初始化，在冒号后使用初始化列表进行初始化。

列表初始化是给数据成员分配内存空间时就进行初始化

就是说分配一个数据成员只要冒号后有此数据成员的赋值表达式(此表达式必须是括号赋值表达式),那么分配了内存空间后在进入函数体之前给数据成员赋值，就是说初始化这个数据成员此时函数体还未执行。

class G{

private:

int n;

public:

G(int x):n(x){

cout << "G()" << endl;

}

~G(){

cout << "~G()" << endl;

}

};

2、构造函数的执行顺序是什么？

虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。

基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。

类类型的成员对象的构造函数（按照初始化顺序）

派生类自己的构造函数。

#include<iostream>

#include<string.h>

using namespace std;

class A{

private:

int n;

public:

A(int x):n(x){

cout << "A()" << endl;

}

~A(){

cout << "~A()" << endl;

}

};

class B: public A{

private:

int n;

public:

B(int x):A(x), n(x){

cout << "B()" << endl;

}

~B(){

cout << "~B()" << endl;

}

};

class C: public A{

private:

int n;

public:

C(int x):A(x), n(x){

cout << "C()" << endl;

}

~C(){

cout << "~C()" << endl;

}

};

class G{

private:

int n;

public:

G(int x):n(x){

cout << "G()" << endl;

}

~G(){

cout << "~G()" << endl;

}

};

class D: public C, public B{

private:

int n;

G g;

public:

D(int x):C(x), B(x+1), n(x), g(x+2){

cout << "D()" << endl;

}

~D(){

cout << "~D()" << endl;

}

};

int main()

{

D \*ptr = new D(10);

delete ptr;

return 0;

}

输出结果：

注意：这里同级别的，构造函数的执行顺序与初始化列表中的顺序无关，与继承顺序或定义类对象的顺序有关。

A()//虚拟基类

C()//普通基类

B()//普通基类

G()//类类型的成员对象的构造函数

D()//派生类自己的构造函数

~D()

~G()

~B()

~C()

~A()

3、有哪些情况必须用到成员列表初始化？作用是什么？

必须使用成员初始化的四种情况：

① 当初始化一个引用成员时；

② 当初始化一个常量成员时；

③ 当调用一个基类的构造函数，而它拥有一组参数时；

④ 当调用一个成员类的构造函数，而它拥有一组参数时；

<https://blog.csdn.net/qq_45060471/article/details/129767269>

### 220、对象复用的了解，零拷贝的了解

对象复用指得是设计模式，对象可以采用不同的设计模式达到复用的目的，最常见的就是继承和组合模式了。

零拷贝：零拷贝主要的任务就是避免CPU将数据从一块存储拷贝到另外一块存储，主要就是利用各种零拷贝技术，避免让CPU做大量的数据拷贝任务，减少不必要的拷贝，或者让别的组件来做这一类简单的数据传输任务，让CPU解脱出来专注于别的任务。这样就可以让系统资源的利用更加有效。

零拷贝技术常见linux中，例如用户空间到内核空间的拷贝，这个是没有必要的，我们可以采用零拷贝技术，这个技术就是通过mmap，直接将内核空间的数据通过映射的方法映射到用户空间上，即物理上共用这段数据。

<https://www.jianshu.com/p/fad3339e3448>

1. 构造函数和析构函数可以调用虚函数吗，为什么

1) 在C++中，提倡不在构造函数和析构函数中调用虚函数；

2) 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则 运行的是为构造函数或析构函数自身类型定义的版本；

3) 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数 时不安全的，故而C++不会进行动态联编；

4) 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析 构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函 数没有任何意义。

222、构造函数、析构函数的执行顺序？构造函数和拷贝构造的内部都干了啥

1) 构造函数顺序

① 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

② 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序， 而不是它们出现在成员初始化表中的顺序。

③ 派生类构造函数。

2) 析构函数顺序

① 调用派生类的析构函数；

② 调用成员类对象的析构函数；

③ 调用基类的析构函数。

223、虚析构函数的作用，父类的析构函数是否要设置为虚函数

1) C++中基类采用virtual虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只 会调用基类的析构函数，而不会调用派生类的析构函数。

那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

2) 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调 用其每一个虚基类以及上一层基类的析构函数。

因此，缺乏任何一个基类析构函数的定义，就会导致链接失败，最好不要把虚析构函数定义为纯虚析构 函数。

1. 构造函数析构函数可否抛出异常

1) C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数 中发生异常，控制权转出构造函数之外。

因此，在对象b的构造函数中发生异常，对象b的析构函数不会被调用。因此会造成内存泄漏。

2) 用auto\_ptr对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危 机，不再需要在析构函数中手动释放资源；

3) 如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用

terminate函数让程序结束；

4) 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函 数执行不全，就是没有完成他应该执行的每一件事情。

225、构造函数一般不定义为虚函数的原因

（1） 创建一个对象时需要确定对象的类型，而虚函数是在运行时动态确定其类型的。在构造一个对象 时，由于对象还未创建成功，编译器无法知道对象的实际类型

（2） 虚函数的调用需要虚函数表指针vptr，而该指针存放在对象的内存空间中，若构造函数声明为虚 函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表vtable地址用来调用虚构造函数了

（3） 虚函数的作用在于通过父类的指针或者引用调用它的时候能够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动调用的，不可能通过父类或者引用去调用，因此就规定构造函数不能是虚 函数

（4） 析构函数一般都要声明为虚函数，这个应该是老生常谈了，这里不再赘述

<https://dwz.cn/lnfW9H6m>

226、类什么时候会析构

1) 对象生命周期结束，被销毁时；

2) delete指向对象的指针时，或delete指向对象的基类类型指针，而其基类虚构函数是虚函数时；

3) 对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用。

227、构造函数或者析构函数中可以调用虚函数吗

从语法上讲，调用完全没有问题。

但是从效果上看，往往不能达到需要的目的。

《Eﬀective C++》的解释是：

派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。 同样，进入基类析构函数时，对象也是基类类型。

<https://dwz.cn/TaJTJONX>

1. 构造函数的几种关键字

default关键字可以显式要求编译器生成合成构造函数，防止在调用时相关构造函数类型没有定义而报错、 delete关键字可以删除构造函数、赋值运算符函数等，这样在使用的时候会得到友善的提示

<https://www.51cto.com/article/780302.html>

### 229、智能指针的原理、常用的智能指针及实现

智能指针是一种用于管理动态分配内存的对象，它们的目的是帮助避免内存泄漏和悬挂指针等常见的内存管理问题。智能指针通过使用 RAII（Resource Acquisition Is Initialization）技术，在对象构造时分配资源，在对象析构时释放资源，从而确保资源在使用完毕后被正确释放。

### 原理

智能指针的原理基于以下几个关键概念：

1. \*\*引用计数\*\*：智能指针通常会维护一个引用计数，用于记录指向动态分配内存的指针数量。当引用计数变为零时，表示没有任何指针指向该内存块，可以安全释放内存。

2. \*\*析构函数\*\*：智能指针中的析构函数负责释放动态分配的内存。当智能指针对象超出其作用域时，析构函数会被调用，从而自动释放所管理的资源。

3. \*\*复制语义\*\*：智能指针可以像原始指针一样进行复制，但在复制时会增加引用计数。当智能指针对象被销毁或赋值时，引用计数会相应地增加或减少。

### 常用的智能指针及实现

1. \*\*std::unique\_ptr\*\*：`std::unique\_ptr` 是 C++11 提供的智能指针，它独占（unique）所指向的对象，在其生命周期内不能被复制。当 `std::unique\_ptr` 被销毁时，它所管理的资源会被自动释放。由于其独占性质，`std::unique\_ptr` 适合用于管理单一所有权的资源。实现上，`std::unique\_ptr` 使用了移动语义来确保资源的唯一性。

2. \*\*std::shared\_ptr\*\*：`std::shared\_ptr` 也是 C++11 提供的智能指针，它允许多个智能指针共享同一个对象，并使用引用计数来跟踪对象的所有者数量。只有当所有 `std::shared\_ptr` 对象都被销毁时，所管理的资源才会被释放。`std::shared\_ptr` 适用于需要多个指针共享资源所有权的情况。实现上，`std::shared\_ptr` 使用了原子操作来确保引用计数的线程安全性。

3. \*\*std::weak\_ptr\*\*：`std::weak\_ptr` 也是 C++11 提供的智能指针，它是 `std::shared\_ptr` 的一种弱引用，不会增加引用计数。`std::weak\_ptr` 允许观察 `std::shared\_ptr` 的对象，但不拥有对象的所有权。通常用于解决 `std::shared\_ptr` 的循环引用问题。由于 `std::weak\_ptr` 不增加引用计数，因此不会影响对象的生命周期。实现上，`std::weak\_ptr` 通常与 `std::shared\_ptr` 共同使用。

这些智能指针的使用可以有效地管理动态分配的内存，减少内存泄漏和悬挂指针等问题的发生，提高程序的健壮性和可维护性。

<https://blog.csdn.net/lizhentao0707/article/details/81156384>

### 230、C++的四种强制转reinterpret\_cast/const\_cast/static\_cast/dynamic\_cast

在C++中，有四种强制类型转换运算符，它们分别是`reinterpret\_cast`、`const\_cast`、`static\_cast`和`dynamic\_cast`。它们用于在不同类型之间进行类型转换，但在使用时需要注意安全性和语义合理性。

1. \*\*reinterpret\_cast\*\*：

- `reinterpret\_cast` 主要用于进行底层的强制类型转换，它可以将一个指针类型转换为另一个不相关的指针类型，也可以将一个指针类型转换为整数类型，或者将整数类型转换为指针类型。这种转换是不安全的，因为它完全依赖于底层平台的特定实现，可能导致未定义行为或运行时错误。因此，应该谨慎使用 `reinterpret\_cast`，并确保转换的语义是合理的。

- 示例：

```cpp

int i = 10;

double\* dp = reinterpret\_cast<double\*>(&i); // 不安全的类型转换

```

2. \*\*const\_cast\*\*：

- `const\_cast` 用于去除指针或引用的常量性或增加常量性。它可以将指向常量对象的指针或引用转换为指向非常量对象的指针或引用，也可以将指向非常量对象的指针或引用转换为指向常量对象的指针或引用。这种转换通常用于解决函数参数为常量问题或者在重载函数中调用非常量版本的情况。

- 示例：

```cpp

const int i = 10;

int\* ip = const\_cast<int\*>(&i); // 去除常量性

```

3. \*\*static\_cast\*\*：

- `static\_cast` 用于执行静态类型转换，它主要用于基本类型之间的转换，以及具有继承关系的指针或引用之间的转换。在指针或引用之间的转换中，如果转换是不安全的（如向上转换），编译器会发出警告，但允许进行转换。`static\_cast` 在编译时进行类型检查，因此它是一种比较安全的类型转换。

- 示例：

```cpp

int i = 10;

double d = static\_cast<double>(i); // 安全的类型转换

```

4. \*\*dynamic\_cast\*\*：

- `dynamic\_cast` 主要用于在继承层次结构中进行安全的向下转型。它只能用于具有虚函数的类，它会在运行时检查对象的实际类型是否与转换的目标类型兼容，如果兼容则进行转换，否则返回空指针（对于指针类型）或抛出 `std::bad\_cast` 异常（对于引用类型）。因此，`dynamic\_cast` 是一种比较安全的类型转换，但只能用于多态类型之间的转换。

- 示例：

```cpp

class Base { virtual void foo() {} };

class Derived : public Base {};

Base\* b = new Derived;

Derived\* d = dynamic\_cast<Derived\*>(b); // 安全的向下转型

```

总的来说，四种强制类型转换运算符各有不同的用途和安全性级别，正确选择并谨慎使用能够确保程序的正确性和安全性。

<https://blog.csdn.net/yi_chengyu/article/details/121921622>

### C++函数调用的压栈过程

C++函数调用的压栈过程

C++函数调用的压栈过程是指在函数调用时，将函数的相关信息和局部变量等数据存储在栈（Stack）上的过程。下面是C++函数调用的一般压栈过程：

压入返回地址：

在函数调用之前，将下一条指令的地址（函数调用后执行的下一条指令）压入栈中，以便函数执行完后能够返回到正确的位置。

压入参数：

将函数调用时传递的参数按照从右到左的顺序压入栈中，以便在函数内部能够访问这些参数。

压入返回值地址（仅适用于有返回值的函数）：

如果函数有返回值，则在调用函数之前会为返回值分配一块内存，并将其地址压入栈中，以便函数返回后将结果存储到返回值地址所指向的位置。

分配局部变量空间：

在函数调用时，会为局部变量分配内存空间。这些局部变量的存储空间通常位于栈帧（Stack Frame）中，栈帧是每个函数调用所使用的栈空间。

执行函数调用：

跳转到被调用函数的入口点，开始执行函数体。

需要注意的是，栈的管理是由编译器和操作系统来完成的，具体的实现可能有所不同。此外，上述过程是一个简化的描述，实际压栈过程可能还涉及寄存器的保存、异常处理、动态内存分配等。

函数调用完成后，会按照相反的顺序将栈上的数据出栈，恢复到调用函数的状态，并将控制权返回给调用函数。

总结：C++函数调用的压栈过程包括压入返回地址、压入参数、压入返回值地址（有返回值的函数）、分配局部变量空间等步骤。这些步骤确保函数能够正确执行，并能够访问所需的数据。栈的管理由编译器和操作系统完成，过程可能因编译器和操作系统的不同而有所差异。

一个简单的示例：

#include <iostream>

​

void funcB(int x) {

int y = x + 1;

std::cout << "Inside funcB: y = " << y << std::endl;

}

​

void funcA(int a, int b) {

int c = a + b;

funcB(c);

std::cout << "Inside funcA: c = " << c << std::endl;

}

​

int main() {

int numA = 5;

int numB = 10;

funcA(numA, numB);

std::cout << "Inside main: numA = " << numA << ", numB = " << numB << std::endl;

​

return 0;

}

在上述示例中，main函数调用了funcA函数，而funcA函数又调用了funcB函数。通过观察示例代码，我们可以看到函数调用的压栈过程。

main函数压栈过程：

main函数开始执行，将下一条指令的地址压入栈中。

numA和numB作为参数传递给funcA函数，按照从右到左的顺序压入栈中。

为funcA函数的局部变量分配内存空间（在此示例中，c变量）。

funcA函数压栈过程：

funcA函数开始执行，将下一条指令的地址压入栈中。

a和b参数值分别从栈中弹出，并分配给funcA函数的局部变量。

为funcB函数的参数c分配内存空间。

funcB函数压栈过程：

funcB函数开始执行，将下一条指令的地址压入栈中。

x参数值从栈中弹出，并分配给funcB函数的局部变量。

为funcB函数的局部变量y分配内存空间。

执行完毕后，栈会按照相反的顺序将数据出栈，恢复到调用函数的状态。

栈帧情况：

main函数的栈帧：

main函数开始执行，为numA和numB分配内存空间，将它们的初始值压入栈中。

为函数funcA的参数numA和numB分配内存空间。

funcA函数的栈帧：

funcA函数开始执行，为a和b分配内存空间。参数numA和numB的值从栈中弹出，并存储到a和b中。

为局部变量c分配内存空间。

funcB函数的栈帧：

funcB函数开始执行，为参数x分配内存空间。参数c的值从栈中弹出，并存储到x中。

为局部变量y分配内存空间。

每个函数调用都会创建一个新的栈帧，用于存储函数的局部变量、参数和其他相关信息。栈帧的创建和销毁由编译器自动管理，确保函数调用的正确性和局部变量的安全性。

最后，程序输出结果如下：

Inside funcB: y = 16

Inside funcA: c = 21

Inside main: numA = 5, numB = 10

上述示例展示了函数调用的压栈过程，包括参数的传递、局部变量的分配和返回地址的保存。这个过程确保了函数执行的正确性，并提供了局部变量的存储空间。

<https://blog.csdn.net/weixin_42492218/article/details/131449215>

### 请描述C++的内存管理方式

C++的内存管理方式主要包括静态内存管理、自动内存管理和动态内存管理三种方式。

1. \*\*静态内存管理\*\*：

- 静态内存分配发生在程序编译时，由编译器根据变量声明进行分配。静态变量的内存分配在程序的数据段（data segment）中完成，其生命周期从程序启动到程序结束。

- 静态内存管理的优点是简单高效，不会产生额外的运行时开销，但缺点是内存空间固定，不够灵活。

2. \*\*自动内存管理\*\*：

- 自动内存分配发生在程序运行时，由编译器自动管理。自动变量的内存分配在栈（stack）上完成，其生命周期与所在作用域的开始和结束相关。

- 自动内存管理的优点是方便快捷，无需手动释放内存，但缺点是内存分配大小受限于栈的大小，对于大对象或需要动态生命周期管理的对象不够灵活。

3. \*\*动态内存管理\*\*：

- 动态内存分配发生在程序运行时，由程序员手动管理。通过动态内存分配运算符 `new` 和 `delete` 或者 `new[]` 和 `delete[]` 来进行动态内存的分配和释放。

- 动态内存分配在堆（heap）上完成，其生命周期由程序员显式控制，可以灵活地分配和释放内存，但需要手动管理内存生命周期，避免内存泄漏和悬挂指针等问题。

动态内存管理是C++中最灵活、最常用的内存管理方式，可以根据程序需要动态地分配和释放内存，但也是最容易出错的方式。因此，在使用动态内存管理时，需要谨慎管理内存生命周期，避免内存泄漏和内存访问错误。

### c和C++的类型安全

什么是类型安全？

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。

“类型安全”常被用来形容编程语言，其根据在于该门编程语言是否提供保障类型安全的机制；有的时候 也用“类型安全”形容某个程序，判别的标准在于该程序是否隐含类型错误。

类型安全的编程语言与类型安全的程序之间，没有必然联系。好的程序员可以使用类型不那么安全的语 言写出类型相当安全的程序，相反的，差一点儿的程序员可能使用类型相当安全的语言写出类型不太安 全的程序。绝对类型安全的编程语言暂时还没有。

<https://blog.csdn.net/a3192048/article/details/82499164>

### 什么情况下会调用拷贝构造函数

用类的一个实例化对象去初始化另一个对象的时候函数的参数是类的对象时（非引用传递）

函数的返回值是函数体内局部对象的类的对象时 ,此时虽然发生（Named return Value优化）NRV

优化，但是由于返回方式是值传递，所以会在返回值的地方调用拷贝构造函数

<https://blog.csdn.net/weixin_41066529/article/details/89671077>

### 235、C++中NULL和nullptr区别

算是为了与C语言进行兼容而定义的一个问题吧

NULL来自C语言，一般由宏定义实现，而 nullptr 则是C++11的新增关键字。在C语言中，NULL被定义为(void\*)0,而在C++语言中，NULL则被定义为整数0

<https://blog.csdn.net/qq_39380590/article/details/82563571>

### 简要说明C++的内存分区

代码区：存放函数体的二进制代码，由操做系统进行管理；

全局区：存放全局变量和静态变量以及常量；

栈区:由编译器自动分配释放，存放函数的参数值、局部变量等；

堆区：由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收。 《C/C++内存管理详解》：

https://chenqx.github.io/2014/09/25/Cpp-Memory-Management/

<https://blog.csdn.net/qq_42049394/article/details/132588603>

### 236、移动构造函数

1) 我们用对象a初始化对象b，后对象a我们就不在使用了，但是对象a的空间还在呀（在析构之前）， 既然拷贝构造函数，实际上就是把a对象的内容复制一份到b中，那么为什么我们不能直接使用a的空间 呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷；

2) 拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用 浅层复制。浅层复制之所以危险，是因为两个指针共同指向一片内存空间，若第一个指针将其释放，另 一个指针的指向就不合法了。

所以我们只要避免第一个指针释放空间就可以了。避免的方法就是将第一个指针（比如a->value）置为

NULL，这样在调用析构函数的时候，由于有判断是否为NULL的语句，所以析构a的时候并不会回收a-

>value指向的空间；

1. 移动构造函数的参数和拷贝构造函数不同，拷贝构造函数的参数是一个左值引用，但是移动构造函 数的初值是一个右值引用。意味着，移动构造函数的参数是一个右值或者将亡值的引用。也就是说，只 用用一个右值，或者将亡值初始化另一个对象的时候，才会调用移动构造函数。而那个move语句，就 是将一个左值变成一个将亡值。

### 237、C++中将临时变量作为返回值时的处理过程

首先需要明白一件事情，临时变量，在函数调用过程中是被压到程序进程的栈中的，当函数退出时，临 时变量出栈，即临时变量已经被销毁，临时变量占用的内存空间没有被清空，但是可以被分配给其他变 量，所以有可能在函数退出时，该内存已经被修改了，对于临时变量来说已经是没有意义的值了

C语言里规定：16bit程序中，返回值保存在ax寄存器中，32bit程序中，返回值保持在eax寄存器中，如 果是64bit返回值，edx寄存器保存高32bit，eax寄存器保存低32bit

由此可见，函数调用结束后，返回值被临时存储到寄存器中，并没有放到堆或栈中，也就是说与内存没 有关系了。当退出函数的时候，临时变量可能被销毁，但是返回值却被放到寄存器中与临时变量的生命 周期没有关系

如果我们需要返回值，一般使用赋值语句就可以了。

<https://www.wandouip.com/t5i204349/>

### 238、C++中将临时变量作为返回值时的处理过程

首先需要明白一件事情，临时变量，在函数调用过程中是被压到程序进程的栈中的，当函数退出时，临时变量出栈，即临时变量已经被销毁，临时变量占用的内存空间没有被清空，但是可以被分配给其他变 量，所以有可能在函数退出时，该内存已经被修改了，对于临时变量来说已经是没有意义的值了

C语言里规定：16bit程序中，返回值保存在ax寄存器中，32bit程序中，返回值保持在eax寄存器中，如 果是64bit返回值，edx寄存器保存高32bit，eax寄存器保存低32bit

由此可见，函数调用结束后，返回值被临时存储到寄存器中，并没有放到堆或栈中，也就是说与内存没 有关系了。当退出函数的时候，临时变量可能被销毁，但是返回值却被放到寄存器中与临时变量的生命 周期没有关系如果我们需要返回值，一般使用赋值语句就可以了。

在C++中，当函数返回一个临时变量时，编译器会执行返回值优化（Return Value Optimization，简称RVO）或者使用移动语义（Move Semantics）来避免不必要的复制操作，从而提高性能和减少内存消耗。

1. \*\*返回值优化（RVO）\*\*：

- 返回值优化是一种编译器优化技术，它通过将函数内创建的临时对象直接构造在函数调用方的目标对象中，避免了临时对象的复制和析构操作。这种优化可以在编译阶段进行，不会改变代码的行为，但能够显著提高性能。

- 示例：

```cpp

std::string createString() {

std::string str = "Hello";

return str; // 返回值优化：将str直接构造在调用方的目标对象中

}

std::string result = createString(); // 返回值直接构造在result中

```

2. \*\*移动语义\*\*：

- 如果返回的临时对象无法进行返回值优化（例如在构造函数中有副作用），或者编译器不支持返回值优化，那么就会使用移动语义来提高效率。移动语义利用了右值引用（rvalue reference）的特性，在对象移动时将资源的所有权转移给目标对象，避免了不必要的复制开销。

- 示例：

```cpp

std::vector<int> createVector() {

std::vector<int> vec = {1, 2, 3, 4, 5};

return std::move(vec); // 使用std::move进行移动语义

}

std::vector<int> result = createVector(); // vec的内容被移动到result中

```

总的来说，无论是返回值优化还是移动语义，编译器都会尽力避免不必要的对象复制和析构操作，以提高程序的性能和效率。因此，在编写函数时，可以放心返回临时对象，而不必担心性能损失。

### 239、智能指针的循环引用

C++11引入了智能指针unique\_ptr，shared\_ptr，及weak\_ptr来帮助用户管理动态分配的内存，其本质是基于RAII原理，即当程序运行超过某个作用域("{ }"包含的区域)时，在该作用域内声明定义的对象会自动调用自己的析构函数析构。当智能指针对象析构时，会将它所指向的对象的资源释放掉，从而不用用户自己手动的调用delete释放。

对于shared\_ptr，它内部有一个引用计数变量，每当它被拷贝一次，这个变量就加1。同样地，每当它被析构，变量就减1。当引用计数变量变为0时，就将它所指向的对象的资源释放掉。这样就有可能出现“循环引用”的问题，导致资源无法释放。

class B;

class A

{

public:

A()

{

std::cout << "A ctr" << std::endl;

}

~A()

{

std::cout << "A dtr" << std::endl;

}

std::shared\_ptr<B> pb;

};

class B

{

public:

B()

{

std::cout << "B ctr" << std::endl;

}

~B()

{

std::cout << "B dtr" << std::endl;

}

std::shared\_ptr<A> pa;

};

int main()

{

std::shared\_ptr<A> a(new A);

std::shared\_ptr<B> b(new B);

a->pb = b;

b->pa = a;

std::cout << "A use count: " << a.use\_count() << std::endl;

std::cout << "B use count: " << b.use\_count() << std::endl;

}

从运行结果可知，程序结束前，a和b的引用计数都是2。当程序运行超出main函数的作用域时，a和b的引用计数减1，但是减完后引用计数还是不为0，所以不会调用A和B的析构函数，这就造成了内存泄露。

从图中可以看出，虽然程序结束后，a和b已经不存在了，它们已经不再指向A和B的对象。但是先前创建的A和B对象内部的pa和pb还在互相指向着，所以A和B对象就一直存在着，无法释放。

解决循环引用的办法：

使用弱引用的智能指针打破这种循环引用。

当parent的生存期超过children的生存期的时候，children改为使用一个普通指针指向parent。

当只剩下最后一个引用的时候需要手动打破循环引用释放对象。

虽然这三种方法都可行，但方法2和方法3都需要程序员手动控制，麻烦且容易出错。

使用弱引用智能指针weak\_ptr：

弱引用计数不会影响对象是否析构，对象的析构由强引用计数决定，即shared\_ptr。如果强引用计数不为0，就算弱引用计数为0，对象也不会析构；如果强引用计数为0，就算弱引用计数不为0，对象也会析构。

class B;

class A

{

public:

A()

{

std::cout << "A ctr" << std::endl;

}

~A()

{

std::cout << "A dtr" << std::endl;

}

std::weak\_ptr<B> pb;

};

class B

{

public:

B()

{

std::cout << "B ctr" << std::endl;

}

~B()

{

std::cout << "B dtr" << std::endl;

}

std::weak\_ptr<A> pa;

};

int main()

{

std::shared\_ptr<A> a(new A);

std::shared\_ptr<B> b(new B);

a->pb = b;

b->pa = a;

std::cout << "A use count: " << a.use\_count() << std::endl;

std::cout << "B use count: " << b.use\_count() << std::endl;

}

使用weak\_ptr之后，可以看到循环引用问题解决了，当程序运行结束后，A和B对象可以正常析构了

<https://blog.csdn.net/zqxf123456789/article/details/108420964>

240、全局变量和局部变量有什么区别

生命周期不同：全局变量随主程序创建和创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至 局部循环体等内部存在，退出就不存在；

使用方式不同：通过声明后全局变量在程序的各个部分都可以用到；局部变量分配在堆栈区，只能在局 部使用。

操作系统和编译器通过内存分配的位置可以区分两者，全局变量分配在全局数据段并且在程序开始运行 的时候被加载。局部变量则分配在堆栈里面 。

<https://www.cnblogs.com/yjd_hycf_space/p/7495640.html>

1. int a[10000000]会有什么问题？

警告 C6262：函数使用了堆栈中的 <constant> 个字节: 超过了 /analyze:stacksize<constant>。<https://learn.microsoft.com/zh-cn/previous-versions/visualstudio/visual-studio-2008/7yhee2f0(v=vs.90)>

1. malloc和new的区别，能否malloc(1.2G)

能，malloc 分配的是虚拟内存

### C++中有几种类型的new

在C++中，new有三种典型的使用方法：plain new， nothrow new和placement new

<https://www.jianshu.com/p/9b57e769c3cb>

[C++中有几种类型的new（plain new\nothrow new\placement new)\_c++ new 类名-CSDN博客](https://blog.csdn.net/LB_AUTO/article/details/119978436)

### malloc 和 new 失败会返回什么

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

<https://blog.csdn.net/qq_53982713/article/details/128627648>

<https://blog.csdn.net/weixin_58163925/article/details/129420071>

### char arr[20]和char \*p = new char[20]的区别？初始化和未初始化的情况？放在内存的那个位置？

`char arr[20]` 和 `char \*p = new char[20]` 的区别主要在于内存管理方式、初始化方式以及内存位置。

1. \*\*内存管理方式\*\*：

- `char arr[20]` 是在栈上分配内存，编译器在编译时就知道数组的大小和位置，在函数作用域结束时会自动释放内存。

- `char \*p = new char[20]` 是在堆上动态分配内存，需要使用 `new` 运算符手动分配内存，而且需要使用 `delete[]` 运算符手动释放内存，否则会导致内存泄漏。

2. \*\*初始化情况\*\*：

- 对于 `char arr[20]`，在定义时可以选择进行初始化，例如 `char arr[20] = {'a', 'b', 'c', ...}` 或者 `char arr[20] = "Hello"`，也可以不进行初始化，此时数组中的内容为未定义的随机值。

- 对于 `char \*p = new char[20]`，动态分配的内存中的内容不会被初始化，需要手动进行初始化。如果需要初始化，可以使用循环或者 `memset` 等方式对内存进行初始化。

3. \*\*内存位置\*\*：

- `char arr[20]` 分配的内存位于栈上，随着函数的调用而分配和释放，具有固定的生命周期和作用域。

- `char \*p = new char[20]` 分配的内存位于堆上，由程序员手动控制其生命周期，直到使用 `delete[]` 明确释放内存为止。

总的来说，`char arr[20]` 是静态分配的数组，其大小和位置在编译时确定，作用域结束时自动释放内存；而 `char \*p = new char[20]` 是动态分配的数组，需要程序员手动管理内存的分配和释放，并且内存的内容需要手动初始化。

### 246、 C++内存分配有哪几种方式？画出C++内存布局图？

****1.栈，****就是那些由[编译器](https://so.csdn.net/so/search?q=%E7%BC%96%E8%AF%91%E5%99%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/BullJordan23/article/details/_blank)在需要的时候分配，在不需要的时候自动清楚的变量的存储区。里面的变量通常是局部变量、函数参数等。  
****2.堆，****就是那些由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。  
****3.自由存储区，****就是那些由malloc等分配的内存块，他和堆是十分相似的，不过它是用free来结束自己的生命的。  
****4.全局/静态存储区****，[全局变量](https://so.csdn.net/so/search?q=%E5%85%A8%E5%B1%80%E5%8F%98%E9%87%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/BullJordan23/article/details/_blank)和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区。  
****5.常量存储区，****这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改）

<https://blog.csdn.net/BullJordan23/article/details/7903730>

### 247、全局变量存在内存哪个区？（C++内存管理）

在C++中，全局变量存在于程序的数据段（data segment）中。数据段是用于存储全局变量和静态变量的内存区域，在程序运行期间始终存在。

数据段通常被划分为以下几个部分：

1. \*\*全局初始化数据段（.data）\*\*：

- 用于存储已初始化的全局变量和静态变量。

- 在程序启动前，这些变量的初始值就已经被存储在这个段中，并且在整个程序的执行过程中保持不变。

2. \*\*全局未初始化数据段（.bss）\*\*：

- 用于存储未初始化的全局变量和静态变量。

- 在程序启动前，这个段中的变量都被初始化为0或者空指针（nullptr）。

下面是一个简化的C++内存布局图，展示了全局变量在数据段中的位置：

```

+----------------------+

| |

| 代码段 |

| |

+----------------------+

| |

| 数据段 |

| |

+----------------------+

| 全局初始化数据段 |

| |

| 全局未初始化数据段 |

+----------------------+

| |

| 堆 |

| |

+----------------------+

| |

| 栈 |

| |

+----------------------+

| |

| 未使用 |

| |

+----------------------+

```

需要注意的是，这只是一个简化的内存布局图，实际的内存布局可能因操作系统和编译器的不同而有所差异。此外，还有其他的内存区域，如堆（heap）和栈（stack），用于动态分配内存和存储局部变量等。

<https://blog.csdn.net/u010843408/article/details/132590708>

### 248、怎么看堆区内存泄漏，栈溢出呢

检测堆区内存泄漏和栈溢出是编程中常见的任务，它们涉及到程序在运行时内存的分配和释放。以下是两者的检测方法：

### 检测堆区内存泄漏：

1. \*\*使用内存检测工具\*\*：

- 工具如Valgrind、AddressSanitizer等可以帮助检测程序中的内存泄漏问题。它们会在程序运行时跟踪内存分配和释放情况，并报告任何未释放的内存块。

2. \*\*编码规范和检查\*\*：

- 编码规范应包括内存分配和释放的一致性。通过审查代码并确保每次分配内存后都有相应的释放操作，可以帮助避免内存泄漏。

3. \*\*手动检查\*\*：

- 在程序中添加记录内存分配和释放的日志，并在程序结束时检查是否有未释放的内存块。

### 检测栈溢出：

1. \*\*使用调试工具\*\*：

- 调试器如GDB、LLDB等可以在程序崩溃时提供堆栈跟踪信息，帮助定位栈溢出的位置。

2. \*\*增加栈空间大小\*\*：

- 如果发现栈溢出问题，可以尝试增加栈空间的大小，以便容纳更多的局部变量和函数调用。

3. \*\*静态代码分析\*\*：

- 使用静态代码分析工具来识别潜在的栈溢出风险，例如递归调用层数过深或者过多局部变量导致栈空间不足等情况。

在编写程序时，应该注意内存的合理管理，包括避免内存泄漏和栈溢出等问题。通过使用工具、遵循编码规范和进行适当的测试，可以帮助及早发现和解决这些问题，提高程序的稳定性和可靠性。

1. 类的对象存储空间

非静态成员的数据类型大小之和。

编译器加入的额外成员变量（如指向虚函数表的指针）。为了边缘对齐优化加入的padding。

空类(无非静态数据成员)的对象的size为1, 当作为基类时, size为0.

1. const的含义及实现机制,比如: const int 1,是怎么做到i只可读的?

const变量的内存位于栈区或者静态存储区，不在符号表(常量表)中：

关于网上所说const修饰的变量存储在符号表中，这个并不完全是对的，const变量的内存位于C++的5大内存中的栈区或者静态存储区。在编译的时候，对于不试图通过内存来修改const变量值的，编译器统统将const变量存放在编译器内部产生的临时列表中，也就是所谓的符号表，该符号表与目标文件连接用的符号表是两个完全不同的东西。此临时符号表的作用就是提高效率，是编译器优化形成了，所以大家不必过多纠结const变量内存存放位置了，它的内存就是位于栈区或者静态存储区.

<https://blog.csdn.net/qq_43152052/article/details/99306967>

1. static存放在哪个区？

静态数据（即使是函数内部的静态局部变量）也存放在全局数据区，初始化不为0的在data段，初始化为0和未初始化的在bss段

Static分配内存空间的时机

需要注意的是，static 成员变量是在初始化时分配内存的。 不是在类声明时，也不是在创建对象时。因此，没有初始化的static 成员变量无法使用，因为还没分配内存。

<https://blog.csdn.net/qq_36908789/article/details/107122955>

252、静态变量在哪儿初始化的

说一下编译的过程，那静态变量是在哪个阶段初始化的?

<https://blog.csdn.net/qq_34139994/article/details/105157313>

253、Static分配内存空间的时机

需要注意的是，static 成员变量是在初始化时分配内存的。 不是在类声明时，也不是在创建对象时。因此，没有初始化的static 成员变量无法使用，因为还没分配内存。

<https://blog.csdn.net/qq_36908789/article/details/107122955>

254、对C++的泛型了解吗，底层如何实现？

C++泛型编程的底层实现主要依赖于模板元编程（Template metaprogramming）和类型擦除（Type erasure）两种技术。

1. \*\*模板元编程\*\*：

模板元编程是C++泛型编程的基础，它允许在编译期间进行类型检查和计算。通过使用模板（template）和元编程技术，可以实现在编译期间生成代码的功能。模板元编程的主要思想是利用模板实例化的特性，在编译期间生成代码，并通过递归、模板特化、模板偏特化等技术实现各种复杂的类型操作和计算。

2. \*\*类型擦除\*\*：

类型擦除是为了在运行时处理泛型类型而进行的一种技术。在C++中，模板实例化是在编译期间完成的，因此生成的代码会针对特定的模板参数类型进行优化。但有时候需要在运行时处理不同类型的对象，这时就需要使用类型擦除技术。类型擦除的基本思想是将泛型类型的具体实现隐藏起来，使用统一的接口来处理不同类型的对象。常见的类型擦除技术包括虚函数、抽象基类、模板类实例化等。

在底层实现中，C++编译器会根据模板的实例化情况生成对应的代码，这些代码会被插入到程序的各个位置，从而实现泛型编程的功能。在运行时，如果涉及到类型擦除，编译器会生成相应的类型信息，以便在运行时进行动态分派和类型转换。

总的来说，C++泛型编程的底层实现依赖于编译器对模板的实例化和类型擦除的支持，通过这些技术，可以实现高效、灵活的泛型编程功能。

1. 元编程（meta programming)

元编程（Meta Programming）是一种编程范式，指的是在编写程序时操作程序本身的能力。元编程允许程序在编译期间或运行时创建、检查和修改程序的结构，以实现自动生成代码、模板化、泛型编程等功能。在C++中，元编程通常与模板元编程（Template Metaprogramming）密切相关，因为C++的模板系统提供了强大的元编程能力。

元编程的主要目的是增强程序的灵活性、可重用性和可维护性，同时提高代码的效率和性能。元编程可以用于解决许多常见的编程问题，如代码生成、类型推导、优化、序列化等。

在C++中，模板元编程是一种利用模板特化、模板偏特化、递归模板等技术，在编译期间生成代码的技术。通过模板元编程，可以实现在编译期间进行类型检查、计算、代码生成等功能，从而提高程序的性能和灵活性。模板元编程通常使用递归和模板特化来处理各种复杂的类型操作和计算，如递归计算斐波那契数列、递归展开模板参数包等。

总的来说，元编程是一种强大的编程技术，可以在编译期间或运行时操作程序的结构和行为，从而实现更灵活、高效的编程。在C++中，模板元编程是元编程的重要组成部分，为实现泛型编程和高性能代码提供了有力的支持。

256、模板实例化的优先级

在C++中，模板实例化的优先级是根据特化的规则来确定的，具体优先级如下：

1. \*\*非模板特化\*\*：

- 非模板特化的函数或类模板是优先级最低的，当没有更具体的特化版本可用时，会选择非模板特化版本进行实例化。

2. \*\*模板偏特化\*\*：

- 模板偏特化是对模板参数进行部分特化的机制。当有多个模板特化版本可用时，编译器会根据模板参数的匹配程度来选择最合适的模板偏特化版本进行实例化。

3. \*\*模板特化\*\*：

- 模板特化是对特定模板参数的完全特化。当有多个模板特化版本可用时，编译器会选择最精确匹配的模板特化版本进行实例化。

4. \*\*函数重载\*\*：

- 如果存在多个函数模板或函数重载，编译器会根据传入的参数类型选择最匹配的函数进行实例化。

需要注意的是，当存在多个模板特化时，编译器会选择最具体的特化版本进行实例化。如果存在多个特化版本的匹配程度相同，或者无法找到最匹配的特化版本，将会引发编译错误。

此外，模板实例化的优先级还受到其他因素的影响，如函数调用的隐式类型转换、函数模板的默认模板参数等。这些因素可能会影响编译器选择的实例化版本。因此，在编写模板代码时，需要注意特化规则和函数重载的优先级，以确保正确的实例化和函数调用。

257 、模板是在什么时候进行实例化

**①**当使用了类模板实例的名字，并且上下文环境要求存在类的定义时。

**②**对象类型是一个类模板实例，当对象被定义时。此点被称作类的**实例化点**。

**③**一个指针或引用指向一个类模板实例，当检查这个指针或引用所指的对象时。

[C++ 模板何时被实例化 - 城市\*斗士（技术） - 博客园 (cnblogs.com)](https://www.cnblogs.com/live-in-city/p/3420577.html)

258、模板和override的关系

模板和override是C++中两个不同的概念，它们在语义和用法上有所不同。

1. \*\*模板\*\*：

- 模板是一种通用代码的机制，可以根据不同的类型参数生成特定的代码。通过模板，可以实现泛型编程，使代码更具通用性和重用性。模板使用关键字`template`定义，并在使用时通过提供类型参数来实例化模板。模板的特化机制允许对特定的类型进行更具体的处理。

2. \*\*override\*\*：

- override是C++11引入的关键字，用于在派生类中显式地重写（覆盖）基类的虚函数。当派生类中的函数与基类中的虚函数具有相同的函数签名时，使用override关键字可以告诉编译器，该函数是打算重写基类函数的。这样可以提高代码的可读性，并帮助检测潜在的错误，例如函数签名错误、基类函数非虚函数等。

在某些情况下，模板和override可以结合使用。例如，在使用模板实现泛型的继承关系时，可以在派生类中使用override关键字来明确地重写基类的虚函数。这样可以确保派生类中的函数与基类中的虚函数具有相同的函数签名，并提供更好的代码可读性和安全性。

总的来说，模板和override是C++中两个不同的概念，模板用于实现通用的泛型编程，而override用于显式地重写基类的虚函数。它们可以在一些场景中结合使用，提供更好的代码可读性和类型安全性。

1. 函数模版，类模版，模版底层怎么实现的

模板类（Template Class）是C++中一种特殊的类模板，它使用模板来定义通用的类结构，使得类中的成员或成员函数能够适用于多种不同的数据类型。模板类的实现、用法和使用场景如下：

### 实现：

模板类的实现方式与一般的类定义类似，只是在类定义中使用了模板参数。通常，模板类的声明和定义会分为两部分，分别放在头文件（.h或.hpp）和源文件（.cpp）中，以便实现代码重用和编译优化。

```cpp

// 模板类的声明（在头文件中）

template<typename T>

class MyTemplate {

private:

T data;

public:

MyTemplate(const T& d) : data(d) {}

T getData() const { return data; }

};

// 模板类的定义（在源文件中）

// 由于模板类的定义和实现都在头文件中，无需额外的源文件定义

```

### 用法：

使用模板类时，需要在实例化时提供具体的数据类型作为模板参数。可以像使用普通类一样来创建模板类的对象，并调用其成员函数。

```cpp

#include "MyTemplate.h"

int main() {

MyTemplate<int> intObj(5);

MyTemplate<double> doubleObj(3.14);

int intValue = intObj.getData();

double doubleValue = doubleObj.getData();

return 0;

}

```

### 使用场景：

1. \*\*泛型编程\*\*：模板类可以用于实现泛型数据结构，如容器类（例如`std::vector`、`std::list`等）和算法类（例如排序算法、搜索算法等），以适用于不同类型的数据。

2. \*\*类型安全\*\*：模板类可以在编译期间进行类型检查，从而提高代码的类型安全性。通过模板参数的类型推导和类型检查，可以避免在运行时出现类型不匹配的错误。

3. \*\*代码重用\*\*：模板类可以通过参数化来重用代码，减少代码的重复编写。通过定义通用的模板类，可以在不同的项目或场景中重复使用，提高代码的可维护性和可重用性。

4. \*\*性能优化\*\*：模板类可以根据具体的数据类型生成高效的代码，从而提高程序的性能。通过模板特化和内联优化等技术，可以针对不同的数据类型生成高效的代码实现。

总的来说，模板类是C++中一种强大的编程工具，可以实现泛型编程、提高代码的类型安全性、重用代码和优化性能等目的。在实际项目中，模板类通常用于实现通用的数据结构和算法，以满足不同数据类型的需求。

260、C++模板是什么，底层怎么实现的？

C++模板是一种通用编程技术，用于实现泛型编程。它允许在编译时生成适用于不同类型的代码，以提高代码的重用性和灵活性。模板是C++语言的核心特性之一，支持函数模板和类模板两种形式。

C++模板的底层实现涉及两个主要的过程：模板定义和模板实例化。

1. \*\*模板定义\*\*：

- 模板定义是指在源代码中使用`template`关键字定义一个通用的代码模板。模板中可以包含模板参数，这些参数可以是类型参数、非类型参数或模板参数本身。模板定义中可以包含函数、类、成员函数、成员变量等代码，用于实现通用的功能或数据结构。在编译过程中，编译器会对模板定义进行语法和语义检查，确保模板定义的正确性。

2. \*\*模板实例化\*\*：

- 模板实例化是指在编译时根据具体的模板参数生成特定类型的代码。在程序中使用模板时，编译器会根据模板参数的具体类型进行模板实例化。编译器会根据模板定义中的代码和模板参数的具体类型，生成对应的函数或类的定义和实现。在模板实例化过程中，编译器会进行类型推导和类型检查，以确保生成的代码与模板参数相匹配。

3. \*\*模板的使用\*\*：

- 使用模板时，需要提供具体的模板参数，以告诉编译器实例化模板。对于函数模板，可以通过函数调用时的实参类型来推导模板参数类型；对于类模板，则需要显式提供模板参数类型。编译器会根据提供的模板参数类型进行模板实例化，生成特定类型的代码。通过模板的实例化，可以在不同的上下文中使用通用的模板代码，以适用于不同的类型。

总的来说，C++模板是一种通用编程技术，用于实现泛型编程。模板的底层实现涉及模板定义和模板实例化两个过程，通过模板参数的具体类型，编译器可以生成适用于不同类型的代码。模板的使用可以提高代码的重用性、灵活性和性能。

261、C++的多态如何实现

C++的多态性，一言以蔽之就是：

在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据所指对象的实际类型来 调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的函数。

<https://www.cnblogs.com/crbhf/p/14924629.html>

1. C++有哪几种的构造函数

C++中的构造函数可以分为4类：

默认构造函数

初始化构造函数（有参数） 拷贝构造函数

移动构造函数（move和右值引用） 委托构造函数

转换构造函数

<https://blog.csdn.net/zxc024000/article/details/51153743>

1. C++函数调用的压栈过程

C++函数调用的压栈过程

C++函数调用的压栈过程是指在函数调用时，将函数的相关信息和局部变量等数据存储在栈（Stack）上的过程。下面是C++函数调用的一般压栈过程：

压入返回地址：

在函数调用之前，将下一条指令的地址（函数调用后执行的下一条指令）压入栈中，以便函数执行完后能够返回到正确的位置。

压入参数：

将函数调用时传递的参数按照从右到左的顺序压入栈中，以便在函数内部能够访问这些参数。

压入返回值地址（仅适用于有返回值的函数）：

如果函数有返回值，则在调用函数之前会为返回值分配一块内存，并将其地址压入栈中，以便函数返回后将结果存储到返回值地址所指向的位置。

分配局部变量空间：

在函数调用时，会为局部变量分配内存空间。这些局部变量的存储空间通常位于栈帧（Stack Frame）中，栈帧是每个函数调用所使用的栈空间。

执行函数调用：

跳转到被调用函数的入口点，开始执行函数体。

<https://blog.csdn.net/weixin_42492218/article/details/131449215>

264、怎么看堆区内存泄漏，栈溢出呢

内存泄漏：

内存泄漏（memory leak）是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理意义上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的分类：

（1）堆内存泄漏（heap leak）：堆内存指的是程序在运行中根据通过malloc/new等从堆中分配的一块内存，使用完成后必须通过调用相对应的free/delete释放掉。如果程序设计的错误导致这部分内存没有被释放掉，那么此后这块内存将不会被使用，就会产生堆内存泄漏。

（2）系统资源泄漏（resource leak）：主要指程序使用系统分配的资源比如bitmap、handle、socket等没有使用相应的函数释放掉，导致系统资源的浪费，严重时可导致系统性能降低，运行不稳定。

（3）没有将基类的析构函数定义为虚函数。当基类指针指向派生类的对象时，如果基类的析构函数不是虚函数，那么子类的析构函数将不会被调用，子类的资源没有被正确释放掉，因此造成内存泄漏。

如何判断内存泄漏？

内存泄漏通常是由于调用了malloc/new等申请内存的操作，但是缺少了对应的free/delete操作。为了判断内存泄漏，我们可以使用linux下的内存泄漏检查工具来判断内存是否泄漏，除此之外，我们可以在写代码时添加内存申请和释放的统计功能，来统计当前申请和释放的内存是否一致，从而来判断内存是否泄漏。

栈溢出

一般通俗来说，栈溢出就是由于递归或循环嵌套层次太多造成的。在平时的编程中，造成栈溢出的现象主要有以下几种：

（1）局部数组过大。当函数内部的数组过大时，有可能导致栈溢出。

（2）递归调用层次太多。递归函数在运行时会执行压栈操作，当压栈次数太多时，也会导致栈溢出。例如，在解决斐波那契数列时，采用递归法求第1000项的斐波那契数时，往往会造成栈溢出。

（3）指针或数组越界。这种情况最常见，例如进行字符串拷贝，或处理用户输入等等。

<https://blog.csdn.net/zhuguanlin121/article/details/119456778>

265、c++中new一个对象会发生什么

C++

1.分配内存空间

对于全局对象，静态对象以及分配在栈区域内的对象，对它们的内存分配是在编译阶段就完成了，

而对于分配在堆区域内的对象，它们的分配是在程序运行阶段完成的。

内存空间的分配过程中需要确定分配空间的大小，即类对象的大小，这个问题是编译器根据类数据成员来进行分配。

是否有足够的内存空间来满足分配，对于不同的情况我们需要具体问题具体分析：全局对象和静态对象。编译器会为他们划分一个独立的段(全局段)为他们分配足够的空间，一般不会涉及到内存空间不够的问题。分配在栈区域的对象。栈区域的大小由编译器的设置决定，不管具体的设置怎样，总归它是有一个具体的值，所以栈空间是有限的，在栈区域内同时分配超过空间大小的对象会导致栈区域溢出，由于栈区域的分配是在编译阶段完成的，所以在栈区域溢出的时候会抛出编译阶段的异常。分配在堆区域的对象。堆内存空间的分配是在运行是进行的，由于堆空间也是有限的，在栈区域内试图同时分配大量的对象会导致分配失败，通常情况会抛出运行时异常或者返回一个没有意义的值(通常是0)。

2.初始化成员变量

初始化列表

这一阶段是对象创建过程中最神秘的一个阶段，也是最容易被忽视的一个阶段。要想知道这一阶段具体完成那些任务，关键是要区分两个容易混淆的概念：初始化 (Initialization)和赋值(Assignment)。初始化早于赋值，它是随着对象的诞生一起进行的。而赋值是在对象诞生以后又给予它一个新的值。这里我想到了一个很好的例子：任何一个在医院诞生的婴儿，在它诞生的同时医院会给它一个标识，以防止和其他的婴儿混淆，这个标识通常是婴儿母亲所在床铺的编号，医院给婴儿一个标识的过程可以看作是初始化。当然当婴儿的父母拿到他们会为他们起个名字，起名字的过程就可以看作是赋值。经过初始化和赋值后，其他人就可以通过名字来标识他们的身份了。区分了这两个概念后，我们再转到对对象初始化的分析上。对类对象的初始化，实际上是对类对象内的所有数据成员进行初始化。C++已经为我们提供了对类对象进行初始化的能力，我们可以通过实现构造函数的初始化列表(memberinitialization list)来实现。

初始化列表先于构造函数体内的代码执行；

赋值(Assignment)--------------构造函数函数体内赋值实现

对象经过初始化以后，我们仍然可以对其进行赋值。我们可以通过构造函数的实现体(即构造函数中由"{}"包裹的部分)来实现。

由以上的分析可以看出，构造函数实现了对象的初始化和赋值两个过程：对象的初始化是通过初始化列表来完成，而对象的赋值则才是通过构造函数，或者更准确的说应该是构造函数的实现体。

3.调用构造方法

•构造函数重载

–构造函数也是函数，拥有重载的特征

–重载的构造函数在构造对象时根据参数自动选择

•构造函数可以使用默认参数

–构造函数也拥有函数参数默认值的特性

–使用默认值可以减少构造函数的个数

•构造函数的初始化列表

–初始化列表可以让构造函数在被调用之前进行初始化工作

–如果类的成员变量是const或引用类型，必须使用初始化列表初始化

在什么情况下必须使用初始化列表来初始化成员变量而不能使用构造函数赋值来实现？

当类的成员变量是const 或者 引用类型时，必须使用初始化列表进行初始化，原因是：

1. const数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。因为类可以创建多个对象，不同的对象其const数据成员的值可以不同。所以不能在类的声明中初始化const数据成员，因为类的对象没被创建时，编译器不知道const数据成员的值是什么。

2. const数据成员的初始化只能在类的构造函数的初始化列表中进行。要想建立在整个类中都恒定的常量，应该用类中的枚举常量来实现，或者static cosnt。

3. 引用的指向只能初始化，不能修改其指向，赋值过程是在修改其指向，其实这就是赋值和初始化的本质区别

总结: 1 . 对象的创建时先分配对象的存储，然后调用构造函数（调用代码段的构造函数并传递刚才开辟的内存空间this指针给该函数），有初始化列表的先初始化列表然后再执行构造函数中的函数体复制操作

2. 成员变量里面有引用类型的变量 或者 是const常量类型的变量 那就必须要使用初始化列表来初始化成员变量。

<https://blog.csdn.net/qq_37050329/article/details/104945157>

1. C++中的内存分区？bss段了解过吗？未初始化的全局变量和初始化的全局变量放在哪里？

在C++中，内存通常被划分为几个不同的段，其中包括堆、栈、全局/静态存储区和常量存储区等。这些段在程序运行时分别用于存储不同类型的数据。

1. \*\*堆（Heap）\*\*：

- 堆是动态分配的内存区域，用于存储程序运行时动态分配的内存。在堆上分配的内存需要手动管理，通过`new`和`delete`操作符进行分配和释放。堆上的内存分配和释放由程序员控制，因此存在内存泄漏和内存碎片化的风险。

2. \*\*栈（Stack）\*\*：

- 栈是一种静态分配的内存区域，用于存储函数调用时的局部变量、函数参数和函数返回地址等信息。栈的内存分配和释放由编译器自动管理，通过在函数调用和返回时进行压栈和出栈操作来实现。栈上的内存分配和释放是自动的，因此不存在内存泄漏的风险。

3. \*\*全局/静态存储区\*\*：

- 全局/静态存储区用于存储全局变量和静态变量，它们在程序运行期间始终存在，直到程序结束才被销毁。全局变量和静态变量在内存中的位置由编译器决定，通常包括以下几个部分：

- \*\*已初始化的全局/静态变量\*\*：已初始化的全局变量和静态变量存储在数据段（Data Segment）中，也称为`data`段。这些变量在编译时已经赋予了初始值，因此存储在程序的可执行文件中，并在程序加载时被初始化。

- \*\*未初始化的全局/静态变量\*\*：未初始化的全局变量和静态变量存储在BSS段（Block Started by Symbol），也称为`bss`段。这些变量在编译时未被赋予初始值，因此在程序加载时会被自动初始化为零或空值。

4. \*\*常量存储区\*\*：

- 常量存储区用于存储程序中的常量数据，包括字符串常量和全局常量等。常量数据通常存储在只读段（Read-Only Segment）中，以防止被修改。这些数据在程序加载时被加载到内存中，并保持不变。

关于`bss`段，它是一种特殊的内存段，用于存储未初始化的全局/静态变量。未初始化的全局变量和静态变量不占用实际的存储空间，它们只是在程序加载时占用了一些内存地址的空间，因此将它们放在`bss`段中可以节省可执行文件的空间。在程序加载时，未初始化的全局/静态变量会被自动初始化为零或空值。

1. 结构体struct和联合体union的区别

一个struct变量的总长度等于所有成员长度之和，遵从字节对其原则;

Union变量的长度等于最长的成员的长度。

<https://blog.csdn.net/weixin_52273949/article/details/130095755>

268、 怎样判断两个浮点数是否相等

对两个浮点数判断大小和是否相等不能直接用==来判断，会出错！明明相等的两个数比较反而是不相 等！对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值！浮点数与0的比较 也应该注意。与浮点数的表示方式有关。

269、C++中的指针参数传递和引用参数传递有什么区别？底层原理你知道吗？

在C++中，指针参数传递和引用参数传递是两种不同的函数参数传递机制，它们在底层原理和用法上有以下区别：

### 指针参数传递

1. \*\*传递方式\*\*：指针参数传递是通过将变量的地址传递给函数来实现的。

2. \*\*修改原始值\*\*：通过指针参数，可以在函数内部修改原始变量的值。

3. \*\*空值传递\*\*：可以传递空指针（nullptr），表示没有有效的对象地址。

4. \*\*多级指针\*\*：可以使用多级指针（如指针的指针）。

5. \*\*内存分配\*\*：指针本身可以在栈上或堆上分配内存。

6. \*\*底层原理\*\*：在底层，指针参数传递实际上是将指针变量的值（即地址）复制到函数的参数栈帧中。因此，函数内部获得的指针指向原始数据。

7. \*\*使用`\*`操作符\*\*：在函数内部，需要使用`\*`操作符来解引用指针以访问原始数据。

```cpp

void func(int \*p) {

if (p) { // 检查是否为空指针

\*p = 10; // 通过解引用修改原始数据

}

}

int main() {

int x = 5;

func(&x); // 传递地址

return 0;

}

```

### 引用参数传递

1. \*\*传递方式\*\*：引用参数传递是通过创建一个原始变量的别名来实现的。

2. \*\*修改原始值\*\*：通过引用参数，可以在函数内部修改原始变量的值。

3. \*\*初始化限制\*\*：引用在定义时必须初始化，并且一旦绑定到一个对象，就不能更改引用。

4. \*\*不能为空\*\*：引用不能为空，它总是指向有效的对象。

5. \*\*单级引用\*\*：C++不支持引用的引用。

6. \*\*内存分配\*\*：引用本身不分配额外的内存，它只是原始变量的一种别名。

7. \*\*底层原理\*\*：在底层，引用参数传递实际上是将引用变量作为原始变量的别名，编译器将引用替换为它所引用的对象的地址。函数内部操作的是原始数据。

8. \*\*不需要解引用\*\*：在函数内部，可以直接使用引用变量，不需要特殊的解引用操作符。

```cpp

void func(int &r) {

r = 10; // 直接修改原始数据

}

int main() {

int x = 5;

func(x); // 传递引用

return 0;

}

```

总的来说，指针参数传递提供了更多的灵活性，包括传递空值和多层指针的能力，但需要额外的解引用操作，并且可能导致内存管理上的复杂性。而引用参数传递更简洁，直接操作原始数据，但限制了传递空值和使用多级引用的能力。在函数参数传递时，选择指针还是引用，取决于具体的需求和上下文。

270、类如何实现只能静态分配和只能动态分配

在C++中，可以通过限制类的构造函数和析构函数的访问权限来实现只能静态分配和只能动态分配的效果。

1. \*\*只能静态分配\*\*：如果希望类只能通过静态分配（在栈上或全局/静态存储区域分配内存）来创建对象，可以将构造函数声明为私有，并提供一个静态的公有成员函数来创建对象。

```cpp

class OnlyStaticAllocation {

private:

OnlyStaticAllocation() {} // 私有构造函数

public:

static OnlyStaticAllocation createObject() {

return OnlyStaticAllocation(); // 静态成员函数创建对象

}

void foo() {

// 类的其他成员函数

}

};

int main() {

// 静态分配

OnlyStaticAllocation obj1 = OnlyStaticAllocation::createObject();

// 以下代码将无法编译通过，因为构造函数是私有的

// OnlyStaticAllocation obj2;

return 0;

}

```

2. \*\*只能动态分配\*\*：如果希望类只能通过动态分配（在堆上分配内存）来创建对象，可以将构造函数声明为私有，并提供一个静态的公有成员函数来创建对象，并返回指向该对象的指针。

```cpp

class OnlyDynamicAllocation {

private:

OnlyDynamicAllocation() {} // 私有构造函数

public:

static OnlyDynamicAllocation\* createObject() {

return new OnlyDynamicAllocation(); // 返回对象指针

}

void foo() {

// 类的其他成员函数

}

};

int main() {

// 动态分配

OnlyDynamicAllocation\* obj1 = OnlyDynamicAllocation::createObject();

// 以下代码将无法编译通过，因为构造函数是私有的

// OnlyDynamicAllocation obj2;

return 0;

}

```

需要注意的是，只能静态分配和只能动态分配是为了限制对象的创建方式，这样可以在一定程度上控制对象的生命周期和内存管理。但在使用时需要注意释放动态分配的对象以避免内存泄漏。

1. 什么情况会自动生成默认构造函数？

1) 带有默认构造函数的类成员对象，如果一个类没有任何构造函数，但它含有一个成员对象，而后者有 默认构造函数，那么编译器就为该类合成出一个默认构造函数。

不过这个合成操作只有在构造函数真正被需要的时候才会发生；

如果一个类A含有多个成员类对象的话，那么类A的每一个构造函数必须调用每一个成员对象的默认构造 函数而且必须按照类对象在类A中的声明顺序进行；

2) 带有默认构造函数的基类，如果一个没有任务构造函数的派生类派生自一个带有默认构造函数基类， 那么该派生类会合成一个构造函数调用上一层基类的默认构造函数；

3) 带有一个虚函数的类

4) 带有一个虚基类的类

5) 合成的默认构造函数中，只有基类子对象和成员类对象会被初始化。所有其他的非静态数据成员都不会被初始化。

272、抽象基类为什么不能创建对象？

抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类往往用来表征对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。

比如，在一个图形编辑软件的分析设计过程中，就会发现问题领域存在着圆、三角形这样一些具体概念，它们是不同的，但是它们又都属于形状这样一个概念，形状这个概念在问题领域并不是直接存在的，它就是一个抽象概念。而正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的。

比较区别

与具体类比较

抽象类不能直接实例化，并且对抽象类使用 new 运算符会导致编译时错误。虽然一些变量和值在编译时的类型可以是抽象的，但是这样的变量和值必须或者为 null，或者含有对非抽象类的实例的引用（此非抽象类是从抽象类派生的）。

允许（但不要求）抽象类包含抽象成员。

抽象类不能被密封。

与接口比较

抽象类表示该类中可能已经有一些方法的具体定义，但是接口就仅仅只能定义各个方法的界面（方法名，参数列表，返回类型），并不关心具体细节。

接口是引用类型的，和抽象类的相似之处有三点：

不能实例化；

包含未实现的方法声明；

派生类必须实现未实现的方法，抽象类是抽象方法，接口则是所有成员（不仅是方法包括其他成员）。 [2]

抽象类与接口紧密相关。然而接口又比抽象类更抽象，这主要体现在它们的差别上：

类可以实现无限个接口，但仅能从一个抽象（或任何其他类型）类继承，从抽象类派生的类仍可实现接口，从而得出接口是用来解决多重继承问题的。

抽象类当中可以存在非抽象的方法，可接口不能，且它里面的方法只是一个声明必须用public来修饰没有具体实现的方法。

抽象类中的成员变量可以被不同的修饰符来修饰，可接口中的成员变量默认的都是静态常量（static final）。

抽象类是对象的抽象，然而接口是一种行为规范。

创建对象

抽象类尚未实现方法，所以不能创建对象。

创建对象去调用方法是指做明确的事情，而这个抽象方法并不明确，所以只有继承抽象类去实现抽象方法才可以。

<https://blog.csdn.net/qq_43410906/article/details/107435448>

1. 继承机制中对象之间如何转换？指针和引用之间如何转换

1) 向上类型转换

将派生类指针或引用转换为基类的指针或引用被称为向上类型转换，向上类型转换会自动进行，而且向 上类型转换是安全的。

2) 向下类型转换

将基类指针或引用转换为派生类指针或引用被称为向下类型转换，向下类型转换不会自动进行，因为一 个基类对应几个派生类，所以向下类型转换时不知道对应哪个派生类，所以在向下类型转换时必须加动 态类型识别技术。RTTI技术，用dynamic\_cast进行向下类型转换。

1. C++类型转换机制
2. 类的类型转换
3. C++ 中的哪些cast发生在运行时？
4. dynamic\_cast的效率问题 ？ 请问dynamic\_cast怎么实现的？
5. 强制转换怎么做的，例如const\_cast原理是啥
6. 有没有其他方法判断是父类还是子类的对象
7. 纯虚函数的实现原理
8. 纯虚函数？使用场景有哪些？
9. 虚继承解决什么问题
10. 模板函数可以是虚函数吗？
11. 基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间
12. 虚函数内部调用非虚函数是调用指针类还是对象类
13. 虚函数实现机制
14. 虚函数表，是对象拥有还是类拥有

## 2、数据结构与算法（以刷题为主，如果实在没时间就略过）

说实话，算法这种东西没得快速提升，算法能力的提升需要日积月累慢慢累积而成的。

在互联网招聘中，不管是笔试还是面试中的手撕算法，可以考察的算法题简直不要太多。比如链表、 树、数组、动态规划、回溯算法、贪心算法、甚至是拓扑都有可能考察到。

而一般说来笔试的难度是比面试稍微高一些的，面试中的手撕算法难度一般是力扣的 medium 水平， 也有一些 easy 的，而笔试至少都是力扣 medium 难度以上的。

### 1、合并有序链表

将两个有序的链表合并为一个新链表，要求新的链表是通过拼接两个链表的节点来生成的。 输入：1->2->4, 1->3->4

输出：1->1->2->3->4->4

力扣链接：<https://leetcode-cn.com/problems/he-bing-liang-ge-pai-xu-de-lian-biao-lcof/>

#include <iostream>

using namespace std;

struct myList {

int val;

myList\* next;

myList(int \_val) :val(\_val), next(nullptr) {}

};

myList\* merge(myList\* l1, myList\* l2) {

if (l1 == nullptr) return l2;

if (l2 == nullptr) return l1;

myList head(0);

myList\* node = &head;

while (l1 != nullptr && l2 != nullptr) {

if (l1->val < l2->val) {

node->*next* = l1;

l1 = l1->*next*;

}

else {

node->*next* = l2;

l2 = l2->*next*;

}

node = node->*next*;

}

if (l1 == nullptr)

node->*next* = l2;

if (l2 == nullptr)

node->*next* = l1;

return head.*next*;

};

int main(void) {

myList\* node0 = new myList(0);

myList\* node1 = new myList(1);

myList\* node2 = new myList(2);

myList\* node3 = new myList(3);

myList\* node4 = new myList(1);

myList\* node5 = new myList(4);

node0->*next* = node1;

node1->*next* = node2;

node2->*next* = node3;

node3->*next* = nullptr;

node4->*next* = node5;

node5->*next* = nullptr;

auto node = merge(node0, node4);

while (node != nullptr) {

cout << node->val << *endl*;

node = node->*next*;

}

return 0;

}

### 2、反转链表

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。 输入: 1->2->3->4->5->NULL

输出: 5->4->3->2->1->NULL

##### 第一种做法

#include<algorithm>

#include<unordered\_map>

#include <iostream>

#include<vector>

using namespace std;

struct node {

int  data;

struct node\* next;

node(int \_data) :data(\_data), next(nullptr) {

}

};

struct node\* init() {

node\* head = new node(1);

node\* node1 = new node(2);

node\* node2 = new node(3);

node\* node3 = new node(4);

node\* node4 = new node(5);

head->*next* = node1;

node1->*next* = node2;

node2->*next* = node3;

node3->*next* = node4;

node4->*next* = nullptr;

return head;

}

struct node\* reverse(node\* head) {

struct node\* pre = new node(-1);

struct node\* temp = new node(-1);

pre = head;

temp = head->*next*;

pre->*next* = nullptr;

struct node\* cur = new node(-1);

cur = temp;

while (cur != nullptr) {

temp = cur;

cur = cur->*next*;

temp->*next* = pre;

pre = temp;

}

return pre;

}

int main() {

auto head = init();

head = reverse(head);

while (head != nullptr) {

cout << head->*data* << *endl*;

head = head->*next*;

}

return 0;

}

##### 第二种做法

|  |
| --- |
| 1. //头插法来做，将元素开辟在栈上，这样会避免内存泄露 2. ListNode\* ReverseList(ListNode\* pHead) { 3 3. // 头插法 4. if (pHead == nullptr || pHead->next == nullptr) return pHead; 5. ListNode dummyNode = ListNode(0); 6. ListNode\* pre = &(dummyNode); 7. pre->next = pHead; 8. ListNode\* cur = pHead->next; 9. pHead->next = nullptr; 10. //pre = cur; 11. ListNode\* temp = nullptr; 12. while (cur != nullptr) { 13. temp = cur; 14. cur = cur->next; 15. temp->next = pre->next; 16. pre->next = temp; 18 }   19 return dummyNode.next; 20  21 }  22 |

### 3、单例模式

**恶汉模式**

|  |
| --- |
| 1. class singlePattern { 2. private: 3. singlePattern() {}; 4. static singlePattern\* p; 5. public: 6. static singlePattern\* instacne(); 7 7. class CG { 8. public:   10 ~CG() {   1. if (singlePattern::p != nullptr) { 2. delete singlePattern::p; 3. singlePattern::p = nullptr; 14 }   15 }  16 };  17 };  18   1. singlePattern\* singlePattern::p = new singlePattern(); 2. singlePattern\* singlePattern::instacne() { 3. return p; 22 } |

**懒汉模式**

|  |
| --- |
| 1. class singlePattern { 2. private: 3. static singlePattern\* p; 4. singlePattern(){} 5. public: 6. static singlePattern\* instance(); 7. class CG { 8. public:   9 ~CG() {   1. if (singlePattern::p != nullptr) { 2. delete singlePattern::p; 3. singlePattern::p = nullptr; 13 }   14 }  15 };  16 };   1. singlePattern\* singlePattern::p = nullptr; 2. singlePattern\* singlePattern::instance() { 3. if (p == nullptr) { 4. return new singlePattern(); 21 }   22 return p; 23 } |

### 4、简单工厂模式

|  |
| --- |
| 1 typedef enum productType { |

1. TypeA,
2. TypeB,
3. TypeC
4. } productTypeTag; 6

7 class Product { 8

9 public:

1. virtual void show() = 0;
2. virtual ~Product() = 0; 12 };

13

1. class ProductA :public Product {
2. public:
3. void show() {
4. cout << "ProductA" << endl; 18 }
5. ~ProductA() {
6. cout << "~ProductA" << endl; 21 }

22 };

23

1. class ProductB :public Product {
2. public:
3. void show() {
4. cout << "ProductB" << endl; 28 }
5. ~ProductB() {
6. cout << "~ProductB" << endl; 31 }

32 };

33

1. class ProductC :public Product {
2. public:
3. void show() {
4. cout << "ProductC" << endl; 38 }
5. ~ProductC() {
6. cout << "~ProductC" << endl; 41 }

42 };

43

44 class Factory { 45

1. public:
2. Product\* createProduct(productType type) {
3. switch (type) {
4. case TypeA:
5. return new ProductA();
6. case TypeB:
7. return new ProductB();
8. case TypeC:
9. return new ProductC();
10. default:
11. return nullptr; 57 }

58 }

59 };

### 5、快排排序

|  |
| --- |
| 1. void quickSort(vector<int>& data, int low, int high) { 2. //for\_each(data.begin(), data.end(), [](const auto a) {cout << a << " "; }); 3. //cout << endl; 4. if (low >= high) return; 5. int key = data[low], begin = low, end = high; 6. while (begin < end) { 7. while (begin<end && data[end]>key) { 8 end--;   9 }  10 if (begin < end) data[begin++] = data[end]; 11   1. while (begin<end && data[begin]<= key) { 2. begin++;   14 }  15 if (begin < end) data[end--] = data[begin]; 16  17  18 }  19   1. data[begin] = key; 2. quickSort(data, low, begin - 1); 3. quickSort(data, begin + 1,high); 23 } |

### 6、归并排序

void mergeSort(*vector*<int>& data, *vector*<int>& copy, int begin, int end) {

if (begin >= end) return;

int mid = begin + (end - begin) / 2;

int low1 = begin, high1 = mid, low2 = mid + 1, high2 = end, index =

begin;

mergeSort(copy, data, low1, high1);

mergeSort(copy, data, low2, high2);

while (low1 <= high1 && low2 <= high2) {

copy[index++] = data[low1] < data[low2] ? data[low1++] :

data[low2++];

}

while (low1 <= high1) {

copy[index++] = data[low1++];

}

while (low2 <= high2) {

copy[index++] = data[low2++];

}

}

void mergeTest() {

*vector*<int> nums = { -5,-10,6,5,12,96,1,2,3 };

*vector*<int> copy(nums);

mergeSort(nums, copy, 0, nums.*size*() - 1);

nums.*assign*(copy.*begin*(), copy.*end*());

*for\_each*(nums.*begin*(), nums.*end*(), [](const auto& a) {cout << a << " ";

});

}

### 7、设计LRU缓存

设计和构建一个“最近最少使用”缓存，该缓存会删除最近最少使用的项目。缓存应该从键映射到值(允许你插入和检索特定键对应的值)，并在初始化时指定最大容量。当缓存被填满时，它应该删除最近最少使用的项目。

它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

|  |
| --- |
| 1 LRUCache cache = new LRUCache( 2 /\* 缓存容量 \*/ ); 2   1. cache.put(1, 1); 2. cache.put(2, 2); 3. cache.get(1); // 返回 1 4. cache.put(3, 3); // 该操作会使得密钥 2 作废 5. cache.get(2); // 返回 -1 (未找到) 6. cache.put(4, 4); // 该操作会使得密钥 1 作废 7. cache.get(1); // 返回 -1 (未找到) 8. cache.get(3); // 返回 3 9. cache.get(4); // 返回 4 |

链接：<https://leetcode-cn.com/problems/lru-cache-lcci>

struct DoubleList {

int key, val;

DoubleList\* pre, \* next;

DoubleList(int \_key, int

\_val) :key(\_key), val(\_val), pre(nullptr), next(nullptr) { }

};

class LRU {

private:

int capacity;

DoubleList\* head, \* tail;

*unordered\_map*<int, DoubleList\*> memory;

public:

LRU(int \_capacity) {

this->capacity = \_capacity;

head = new DoubleList(-1, -1);

tail = new DoubleList(-1, -1);

head->next = tail;

tail->pre = head;

}

~LRU() {

if (head != nullptr) {

delete head;

head = nullptr;

}

if (tail != nullptr) {

delete tail;

tail = nullptr;

}

for (auto& a : memory) {

if (a.*second* != nullptr) {

delete a.*second*;

a.*second* = nullptr;

}

}

}

void set(int \_key, int \_val) {

if (memory.*find*(\_key) != memory.*end*()) {

DoubleList\* node = memory[\_key];

removeNode(node);

node->val = \_val;

pushNode(node);

return;

}

if (memory.*size*() == this->capacity) {// 这里很重要，也很爱错，千万记得更

新memory

int topKey = head->next->key;//取得key值，方便在后面删除

removeNode(head->next);//移除头部的下一个

memory.*erase*(topKey);//在memory中删除当前头部的值

}

DoubleList\* node = new DoubleList(\_key, \_val);//新增node

pushNode(node);//放在尾部

memory[\_key] = node;//记得在memory中添加进去

}

int get(int \_key) {

if (memory.*find*(\_key) != memory.*end*()) {

DoubleList\* node = memory[\_key];

removeNode(node);

pushNode(node);

return node->val;

}

return -1;

}

void removeNode(DoubleList\* node) {

node->pre->next = node->next;

node->next->pre = node->pre;

}

void pushNode(DoubleList\* node) {

tail->pre->next = node;

node->pre = tail->pre;

node->next = tail;

tail->pre = node;

}

};

### 8、重排链表

给定一个单链表 *L*：*L*0→*L*1→…→*L\*\*n*-1→*L*n ，

将其重新排列后变为： *L*0→*L\*\*n*→*L*1→*L\*\*n*-1→*L*2→*L\*\*n*-2→…

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

|  |
| --- |
| 1 示例 1:  2 给定链表 1->2->3->4, 重新排列为 1->4->2->3.  3 示例 2:  4 给定链表 1->2->3->4->5, 重新排列为 1->5->2->4->3. |

力扣链接：<https://leetcode-cn.com/problems/reorder-list/>

#include<iostream>

#include<string>

#include<vector>

#include<algorithm>

#include<unordered\_map>

using namespace std;

struct ListNode {

int val;

ListNode\* next;

ListNode(int \_val) :val(\_val), next(nullptr) {}

};

ListNode\* myReverseList(ListNode\* head) {

if (head == nullptr || head->*next* == nullptr) return head;

ListNode dumyhead(0);

ListNode\* pre = &dumyhead;

pre->*next* = head;

ListNode\* cur = head->*next*;

head->*next* = nullptr;

ListNode\* node = new ListNode(-1);

while (cur != nullptr) {

node = cur;

cur = cur->*next*;

node->*next* = pre->*next*;

pre->*next* = node;

}

return dumyhead.*next*;

}

ListNode\* myMerge(ListNode\* p1, ListNode\* p2) {

if (p1 == nullptr) return p2;

if (p2 == nullptr) return p1;

ListNode dumyhead(0);

ListNode\* pre = &dumyhead;

while (p1 != nullptr && p2 != nullptr) {

pre->*next* = p1;

p1 = p1->*next*;

pre = pre->*next*;

pre->*next* = p2;

p2 = p2->*next*;

pre = pre->*next*;

}

if (p1 != nullptr) pre->*next* = p1;

return dumyhead.*next*;

}

ListNode\* myReverOrderList(ListNode\* head) {

if (head == nullptr || head->*next* == nullptr) return head;

ListNode\* slow = head, \* fast = head->*next*;

while (fast != nullptr && fast->*next* != nullptr) {

slow = slow->*next*;

fast = fast->*next*->*next*;

}

ListNode\* second = slow->*next*;

slow->*next* = nullptr;

second = myReverseList(second);

head = myMerge(head, second);

return head;

}

int  main() {

ListNode\* head = new ListNode(1);

ListNode\* node1 = new ListNode(2);

ListNode\* node2 = new ListNode(3);

ListNode\* node3 = new ListNode(4);

ListNode\* node4 = new ListNode(5);

ListNode\* node5 = new ListNode(6);

head->*next* = node1;

node1->*next* = node2;

node2->*next* = node3;

node3->*next* = node4;

node4->*next* = node5;

node5->*next* = nullptr;

head = myReverOrderList(head);

while (head != nullptr) {

cout << head->val << *endl*;

head = head->*next*;

}

return 0;

}

### 9、奇偶链表

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指 的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

|  |
| --- |
| 1 示例 1:  2 输入: 1->2->3->4->5->NULL  3 输出: 1->3->5->2->4->NULL  4 示例 2:  5 输入: 2->1->3->5->6->4->7->NULL  6 输出: 2->3->6->7->1->5->4->NULL |

说明:

应当保持奇数节点和偶数节点的相对顺序。

链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

力扣链接：<https://leetcode-cn.com/problems/odd-even-linked-list/>

##### 第一种解法

|  |
| --- |
| 1. ListNode\* oddEvenList(ListNode\* head) { 2. if(head==NULL || head->next==NULL) 3 {   4 return head;  5 }   1. ListNode\* first=head;//奇链表头结点 2. ListNode\* second=head->next;//偶链表头结点 3. ListNode\* cur=second;//保存偶链表头结点 4. while(second != nullptr && second->next != nullptr) 10 { 5. first->next=second->next; 6. second->next=first->next->next; 7. first=first->next; 8. second=second->next; 15 } 9. first->next=cur; 10. return head; 18 }   19 |

##### 第二种解法

ListNode\* oddEvenList(ListNode\* head)

{

if (head == *NULL*)

{

return head;

}

ListNode\* p = head;

ListNode\* q = head->*next*;

ListNode\* evenhead = q;

while (q != *NULL* && q->*next* != *NULL*)

{

p->*next* = p->*next*->*next*;

p = p->*next*;

q->*next* = q->*next*->*next*;

q = q->*next*;

}

p->*next* = evenhead;

return head;

}

### 10 快排的优化

快速排序是一种常见的排序算法，它在大多数情况下都具有良好的性能，但在某些特殊情况下，其效率可能会下降。以下是一些优化快速排序的方法：  
  
1. 三数取中法  
  
在选择枢轴元素时，可以使用三数取中法，即从待排序数组的左端、右端和中心位置选择三个元素，然后选择它们的中位数作为枢轴元素。这样可以获得更好的分割结果，从而提高快速排序的效率。  
  
2. 插入排序优化  
  
在数据规模较小时，插入排序的性能通常比快速排序的好。因此，可以在快速排序中添加插入排序优化，当数组长度小于某个阈值时，使用插入排序进一步优化排序效率。  
  
3. 尾递归优化  
  
尾递归优化是一种减少函数调用栈空间的方法，在快速排序过程中，使用递归调用时会不断调用新的栈帧，导致栈空间的浪费。使用尾递归优化可以将递归调用转换为循环调用，从而减少栈空间的使用。  
  
4. 随机化优化  
  
在选择枢轴元素时，如果只是选择待排序数组的第一个元素，则在某些情况下会导致快速排序的效率下降。因此，可以使用随机化的方法，从待排序数组中随机选择一个元素作为枢轴元素，以避免上述问题。  
  
通过以上优化方法，可以提高C++快速排序的效率，减少时间和空间成本。

### 11、字典树

/\*\*

\* 前缀树/字典树

\*

\* @author snailmann

\*/

public class TrieTree {

/\*\*

\* root node，meaningless dummy node

\*/

private TrieNode root;

public TrieTree() {

root = new TrieNode('\*');

}

/\*\*

\* 插入一个单词 [a-z]

\*

\* @param word 单词

\*/

public void insert(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++) {

int index = word.charAt(i) - 'a';

if (node.next[index] == null) {

node.next[index] = new TrieNode(word.charAt(i));

}

node = node.next[index];

}

node.end = true;

++node.count;

}

/\*\*

\* 是否包含该单词

\*

\* @param word 单词

\* @return 真假

\*/

public boolean contains(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++) {

int index = word.charAt(i) - 'a';

if (node.next[index] == null) {

return false;

}

node = node.next[index];

}

return node.end;

}

/\*\*

\* 是否有以 prefix 为前缀的单词

\*

\* @param prefix 前缀

\* @return 真假

\*/

public boolean startsWith(String prefix) {

TrieNode node = root;

for (int i = 0; i < prefix.length(); i++) {

int index = prefix.charAt(i) - 'a';

if (node.next[index] == null) {

return false;

}

node = node.next[index];

}

return true;

}

/\*\*

\* 词频统计

\*

\* @param word 单词

\* @return 单词出现的次数

\*/

public int count(String word) {

TrieNode node = tailNode(word);

if (node == null) {

return 0;

}

return node.count;

}

/\*\*

\* 单词联想，通过多路递归实现

\*

\* @param prefix 前缀

\* @return 联想词集合

\*/

public List<String> search(String prefix) {

TrieNode node = tailNode(prefix);

if (node == null) {

return Collections.emptyList();

}

List<String> words = new ArrayList<>();

for (TrieNode next : node.next) {

if (next != null) {

String word = prefix + next.data;

if (next.end) {

words.add(word);

}

List<String> relates = search(word);

if (relates != null && relates.size() != 0) {

words.addAll(relates);

}

}

}

return words;

}

/\*\*

\* 获取单词的尾部节点

\*

\* @param word 单词

\* @return 如果不存在，则返回 null

\*/

private TrieNode tailNode(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++) {

int index = word.charAt(i) - 'a';

if (node.next[index] == null) {

return null;

}

node = node.next[index];

}

return node;

}

@Override

public String toString() {

String prefix = "";

TrieNode node = root;

if (node == null) {

return "";

}

return search(prefix).toString();

}

public static class TrieNode {

/\*\*

\* 是否构成单词

\*/

private boolean end;

/\*\*

\* 字符数据

\*/

private char data;

/\*\*

\* 单词次数

\*/

private int count;

/\*\*

\* 26 棵子树 [a-z]

\*/

private TrieNode[] next = new TrieNode[26];

public TrieNode(char data) {

this.data = data;

}

@Override

public String toString() {

return "TrieNode{" +

"end=" + end +

", data=" + data +

", num=" + count +

", next=" + Arrays.toString(next) +

'}';

}

}

public static void main(String[] args) {

TrieTree trieTree = new TrieTree();

trieTree.insert("helloworld");

trieTree.insert("helloworlde");

trieTree.insert("helloworldas");

trieTree.insert("hellocool");

trieTree.insert("sdf");

trieTree.insert("ab");

trieTree.insert("abc");

trieTree.insert("abd");

System.out.println(trieTree.contains("helloworld"));

System.out.println(trieTree.startsWith("hello"));

System.out.println(trieTree.count("helloworld"));

System.out.println(trieTree.count("hello"));

System.out.println(trieTree);

}

<https://blog.csdn.net/SnailMann/article/details/115493906>

### 12、kmp算法

#include <iostream>

#include <string>

using namespace std;

const int N = 1e5 + 10;

int nxt[N];

int kmp(const string &s, const string &t) {

int i = 0, j = 0, k = -1;

nxt[0] = -1;

while (i < t.size() - 1) {

if (k == -1 || t[k] == t[i]) nxt[++i] = ++k;

else k = nxt[k];

}

i = 0;

while (i < s.size() && j < (int) t.size()) {

if (j == -1 || s[i] == t[j]) i++, j++;

else j = nxt[j];

}

if (j == t.size()) return i - j;

else return -1;

}

int main() {

string s, t;

cin >> s >> t;

cout << kmp(s, t) << endl;

return 0;

}

<https://blog.csdn.net/raelum/article/details/128823560>

1. 排序算法，时间复杂度，空间复杂度



<https://blog.csdn.net/cativen/article/details/124609034>

14、归并排序求逆序对

<https://blog.csdn.net/zy98zy998/article/details/122745998>

### 15、归并排序的原理

归并排序是一种经典的排序算法，基于分治法的思想。它的原理可以简单概括为以下几个步骤：

1. \*\*分解（Divide）\*\*：将待排序的序列分成两个子序列，直到每个子序列只有一个元素为止。这个过程是通过递归实现的。

2. \*\*解决（Conquer）\*\*：对每个子序列进行排序。如果序列长度为1，则认为它已经是有序的。

3. \*\*合并（Merge）\*\*：将排好序的子序列合并成一个有序序列。这个过程是归并排序的核心步骤。

在合并过程中，需要比较两个子序列的元素，并按照顺序合并到一个新的序列中。具体步骤如下：

- 首先，比较两个子序列的第一个元素，将较小（或较大）的元素放入新序列中。

- 然后，从剩余的元素中继续比较并选择较小（或较大）的元素放入新序列中。

- 重复以上步骤，直到两个子序列中的所有元素都放入了新序列中。

最终，通过不断合并子序列，得到一个完全有序的序列，完成了整个排序过程。

归并排序的时间复杂度是O(nlogn)，其中n是待排序序列的长度。这是因为归并排序每次将序列分成两半，并且需要O(n)的时间将两个有序子序列合并成一个有序序列。因此，总的时间复杂度可以表示为logn次分解和每次合并O(n)的时间复杂度。

### 红黑树

红黑树（Red-Black Tree）是一种自平衡的二叉搜索树，它在插入和删除节点时通过一系列的旋转和变色操作来保持树的平衡。红黑树具有以下特点：

1. \*\*节点颜色\*\*：每个节点被标记为红色或黑色。

2. \*\*根节点\*\*：根节点是黑色的。

3. \*\*叶子节点\*\*：叶子节点（NIL节点或空节点）是黑色的。

4. \*\*颜色规则\*\*：每个红色节点的子节点都是黑色的。

5. \*\*黑高度规则\*\*：从任意节点出发，到达其每个叶子节点的路径上，经过的黑色节点数量是相同的。也就是说，红黑树的黑高度是指从根节点到叶子节点的最长路径上的黑色节点数量。

红黑树的插入和删除操作需要维护上述特性，以保证树的平衡性。以下是红黑树的插入和删除节点的基本操作：

\*\*插入节点操作\*\*：

1. 将新节点插入红黑树中，将其颜色设置为红色。

2. 根据红黑树的性质，可能需要进行旋转和颜色调整的操作，以保持树的平衡性。

\*\*删除节点操作\*\*：

1. 删除节点并调整红黑树，保持树的平衡性。

2. 如果被删除节点有一个子节点，则用子节点替代被删除节点。

3. 如果被删除节点有两个子节点，则找到它的后继节点（即比它大的最小节点），将后继节点的值复制到被删除节点，并删除后继节点。

4. 根据红黑树的性质，可能需要进行旋转和颜色调整的操作，以保持树的平衡性。

红黑树的自平衡性质保证了其操作的时间复杂度为O(logn)，其中n是树中节点的数量。这使得红黑树在实际中被广泛应用于需要高效插入、删除和查找的场景，例如在C++的STL中的`map`和`set`容器的实现中。

### b+树, B+树增加一个节点会怎么样

B+树是一种常用的多叉树结构，用于实现关联数组和排序集合等数据结构。在B+树中，每个节点可以存储多个关键字和对应的指针，而且具有以下特点：

1. 内部节点存储关键字和指向子节点的指针，叶子节点存储关键字和对应的值或数据记录。

2. 所有叶子节点通过指针连接成一个有序链表，便于范围查询和遍历。

3. 内部节点的关键字按照升序排列，且每个关键字对应的指针指向的子树中的所有关键字都大于等于（或小于等于）该关键字。

当向B+树中增加一个节点时，可能会触发以下情况：

1. \*\*分裂节点\*\*：如果插入新节点后，该节点的关键字数量超过了B+树节点的容量限制，那么就需要将该节点进行分裂。分裂操作会将节点分成两个节点，并将其中一部分关键字移动到新的节点中。

2. \*\*向父节点插入新关键字\*\*：如果分裂节点导致父节点关键字数量超过了容量限制，那么就需要向父节点插入新的关键字，并调整指针指向子节点。

3. \*\*调整树的高度\*\*：如果分裂操作导致根节点被分裂，那么就需要创建一个新的根节点，并将原根节点和新分裂出来的节点作为子节点插入新的根节点中，从而增加树的高度。

4. \*\*更新叶子节点链表\*\*：如果新插入的节点是叶子节点，并且需要调整叶子节点之间的连接关系，以确保有序链表的正确性。

总的来说，增加一个节点可能会引起节点的分裂、父节点的调整、树高的增加以及叶子节点链表的更新等操作，以保持B+树的平衡性和有序性。

### 如何减少哈希冲突

减少哈希冲突是提高哈希表性能和效率的关键之一。以下是一些常见的方法：

1. 良好的哈希函数设计：

选择一个好的哈希函数至关重要。好的哈希函数应该能够将输入的键均匀地分散到哈希表的不同位置上，避免将大量键映射到同一个槽位，从而减少冲突的可能性。通常，好的哈希函数会考虑到键的特征，并尽可能将它们映射到不同的位置。

1. 调整哈希表大小：

当哈希表中的负载因子（已存储元素数量与哈希表容量的比值）过高时，哈希冲突的概率会增加。通过增加哈希表的大小，可以减少每个槽位上的元素数量，从而降低冲突的可能性。一般来说，当负载因子超过某个阈值时，就应该考虑扩大哈希表的容量。

1. \*\*使用链地址法（Separate Chaining）\*\*：

链地址法是一种解决哈希冲突的常见方法。它将哈希表的每个槽位都连接一个链表，当发生冲突时，将冲突的元素添加到对应槽位的链表中。链地址法能够保证每个元素总是能够插入到哈希表中，尽管可能需要顺序查找链表来找到目标元素。

1. 使用开放地址法（Open Addressing）：

开放地址法是另一种解决哈希冲突的方法。它尝试将冲突的元素插入到哈希表的其他槽位中，而不是使用链表。常见的开放地址法包括线性探测、二次探测和双重哈希等技术。

5. 再哈希（Rehashing）：再哈希是一种动态调整哈希函数的方法。当发生冲突时，可以通过再哈希来选择一个不同的哈希函数，尽量将元素映射到不同的槽位中，从而减少冲突的发生。

通过以上方法的组合或选择，可以有效地减少哈希冲突，提高哈希表的性能和效率。选择适合具体应用场景的方法可以最大程度地减少冲突，提高哈希表的操作效率。

### 19、排序算法有哪些？冒泡和快排的区别

常见的排序算法包括：

1. 冒泡排序（Bubble Sort）

2. 选择排序（Selection Sort）

3. 插入排序（Insertion Sort）

4. 归并排序（Merge Sort）

5. 快速排序（Quick Sort）

6. 堆排序（Heap Sort）

7. 计数排序（Counting Sort）

8. 桶排序（Bucket Sort）

9. 基数排序（Radix Sort）

冒泡排序和快速排序是两种常见的排序算法，它们的主要区别在于排序的方式和时间复杂度。

冒泡排序（Bubble Sort）：

- 冒泡排序是一种简单直观的排序算法，它重复地遍历要排序的列表，一次比较两个相邻的元素，并且如果它们的顺序错误就交换它们。

- 每次遍历过程中，都会将最大（或最小）的元素移动到列表的末尾，因此在每次遍历后，未排序部分的最大（或最小）元素会被固定在正确的位置上。

- 时间复杂度为$O(n^2)$，其中$n$是要排序的元素数量。

快速排序（Quick Sort）：

- 快速排序是一种分治算法，它将一个列表分成两个子列表，然后递归地对子列表进行排序。

- 在快速排序中，选择一个基准元素，然后将列表中的元素分为两部分，左边的元素都小于基准元素，右边的元素都大于基准元素。

- 然后对左右两部分分别递归地进行快速排序，直到整个列表有序。

- 快速排序的平均时间复杂度为$O(n \log n)$，最坏情况下的时间复杂度为$O(n^2)$，其中$n$是要排序的元素数量。

区别：

1. 算法思想：

冒泡排序是一种比较简单的交换排序算法，每次通过比较相邻元素的大小来进行排序；而快速排序则是一种分治算法，通过选取基准元素将列表分为两部分，并递归地对子列表进行排序。

2. 时间复杂度：快速排序的平均时间复杂度要优于冒泡排序，尤其在大规模数据的情况下，快速排序通常表现更好。

3. 稳定性：冒泡排序是一种稳定的排序算法，而快速排序在不稳定的情况下可能改变相同元素的相对位置。

综上所述，虽然冒泡排序和快速排序都是常见的排序算法，但它们的实现思想、性能和稳定性有很大的差异。在选择排序算法时，需要根据具体的需求和数据规模来进行选择。

### B树

### 数据机构 给出100万个字符串，统计相同字符串出现的次数，多种方式？

|  |
| --- |
| // 获得本地系统的主机名和IP地址  hostent\* localHost = gethostbyname("");  char\* localIP = inet\_ntoa(\*(struct in\_addr\*) \* localHost->h\_addr\_list); |

### 数据结构 哈希冲突的解决方式，搜索的时候怎么办？

### 数据结构 知道跳跃表嘛，跳跃表的优缺点

### 十亿QQ号，查找某个QQ号在不在里面

### 40亿个QQ号怎么存储

### 在数组n中找第k大的数

### 哈希实现的方法有哪些，解决哈希冲突的方法有哪些

### 目前效率最好的内部排序是什么

### Top K海量数据查找

### 描述下快速排序的思想，时间复杂度？什么场景对应着最坏情况？

### 打印回环数组

### B+树索引和Hash索引的区别？

### 5 二分查找（注意边界条件）实现思路空间复杂度

### 6 常见排序算法的实现以及稳定性（快排跟归并考的很多）

### 7 字符串翻转（O(n)）、匹配（KMP算法）

### 红黑树的性质以及插入和删除

### 卡特兰数以及公式推导（应多很多）

### B+树，B树，红黑树，二叉搜索树

### 红黑树和一般的平衡二叉树，增、删、改、查的过程和效率、时间复杂度

### 二叉树各层节点数，递归、非递归，时间、空间复杂度

### unordered\_map和map区别

### 判断一棵树是否为二叉搜索树，首先用递归判断实现的，后来有问不递归怎么实现（判断每一个点的中序后继节点是否大于该点），如何空间复杂度O（1），讲了一下morris遍历的原理

### 顺着项目问：如果要做一个栈，用数组的结构还是链表？（提示）数组和链表哪一个遍历的效率高？（解答：数组。考虑cache缓存，所以是一组一组读进去的，而链表肯定是一个一个读的）

### 多个有序数组排序 vector fun( vector<vector > )

### 100G的文件，放的整数，怎么排序

### 讲讲贪心算法

### 设计DNS服务器中cache的数据结构。

### 双向链表的实现

### 二叉树遍历查找

### 链表和数据的差异

### 二叉搜索树与红黑树的区别

### 动态规划原理，使用方法，怎么实现规划?

### 递归的好处和缺点

### 大文件如何排序

### hash冲突了如何拓展，如何重构，在不影响使用的前提下

### kmp的复杂度，strstr内置算法都是用的什么

### 5T大文件，其中有一对数字相等（比如：1, 1, 2, 3, 6, 10, -1, ..., 7），给定有限内存和CPU，如何找出这一对数字

### 平衡二叉树的查找时间复杂度

### 描述一下堆排序和快速排序

### 如何让哈希写入的时间复杂度保证是o（1）

### 说说AVL树？

### 哈希表中负载因子过大，性能下降，要怎样扩容？

### bfs思路

### 图的数据结构；

### 迪杰斯特拉算法；

### 回溯算法是什么？什么情况下使用

### 刚才说到了递归的思想，讲讲递归算法有哪几个环节？

### 字典树 在字典树和基数树中插入alibaba，然后又插入alimama，第二次插入的时候有什么区别

### 一个大文件中，每一行有一个整数，怎么找第100大的数？

### 一个大文件中，每一行有一个整数，怎么找中位数？

### "如果有一个很大的文件，TB级别，文件里是乱序的数字，如何排序？mapreduce如何实现？

### 排序过程中的归并排序，请描述一下其过程？时间复杂度"

### 介绍背包问题的原理

### 千万级数组如何求最大k个数？（用最小堆反之最大堆）

### 千万数据范围有限，0到1000，有很多重复的，按频率排序怎么处理？

### 二分法

### 堆排序解释，删除插入操作怎么实现。

### 算法/设计题 设计文件描述符分配机制

### 1000个球分成五堆一共有多少种分堆方式

### 循环队列

### 做完这些还剩3分钟，面试官问有一个每秒钟都有新数字进来的序列，需要随时可以查询它们的中位数，问我用什么数据结构存比较合适

### 解释ChainBuffer RingBuffer

### 讲一下基于RingBuffer和基于ChainBuffer的无锁异步日志的实现（前面总共30min）

### 在圆形桌子上交替下棋的必胜策略

### 介绍一下跳表及其插入、查找、删除的时间复杂度（均为 O(logn)）

### 给定一块够的内存，对一个非常大的无重复的数组排序，要求稳定高效

### 背包问题如何用线性规划的思路建立求解模型，非常详细的讲每一步分析过程

### 生成10000个随机数，要求无重复，如何设计

### 一个公交站一小时通过公交车的概率是96%，那么半小时内有公交车通过几率是多少

### 完全二叉树和平衡二叉树

### 说说并查集

### 图

### 差分数组好处？

### 快排和堆排的时间复杂度，以及最坏情况

### 解释快速排序，时空复杂度，是否稳定

### 解释哈希表如何缓解碰撞带来的性能下降

### 给定一个字符串，要求把字符串中偶数位数的子回文串删掉；注意，回文串删除后生成新偶数位数的子回文串也需要被删掉。返回最终删除后的字符串。

### 给定一个二维数组，已知两个节点x y，二维数组的每一行的起点终点均为x y，二维数组的每一行代表了x到y的每一条路线。因为路线中间可能有一些点重合，所以x到y的路线数肯定比行数要多。求x到y的路线数

### 哈希表 为什么要设计为数组+链表的形式

### 讲一下跳表

### 了解Trie树吗

### 数组和哈希区别（面试官说主要是寻址方式区别），内存分段布局

### 递归和迭代的却别,所以递归可转成迭代,所有迭代可转成递归吗

### "哈夫曼编码

### 前缀表达式"

### 实现循环队列（数组和双链表的区别，答出来了三点还不够）

### 问一亿个数怎么找出最大的k个，说了后让写个小顶堆（垃圾赛码网，怎么都编译不过，难受的是看不懂报错问题，完了在我电脑上同样的代码啥事没有）

### malloc的内部实现细节，涉及malloc如何分配内存，free函数如何得到释放内存的长度？具体的设计思路应该是怎么样的？

### 链表增删快，那如何提高其查询效率，有没有什么想法？

### 大顶堆、小顶堆了解吗？

### LRU算法了解吗？你会如何实现它？这个算法可以应用在哪些场景下？

### 数据结构：排序算法的复杂度稳定性，图的深度广度遍历

### 一个树的广义表达A(B(E,F,G)，C,D)怎么转换为二叉树。

### hash算法，α是啥，碰撞和n有没关系

### hash，线性探测再散列算法

### 终极位运算，怎么找规律。197&0x5555555 + (n>>1)&0x5555555. 好多个，算一个硬算可以，好几个不知道规律是啥。

### 路径规划（dijstra、A\*）

### 红黑树时间复杂度？

### LFU和LRU相关？

### 跳表结构？在哪些数据库有用到

### 常用的哈希函数

### 二叉搜索树的第K大元素（要分析复杂度，要最优）

### 求两个不同大小set的交集，如何使时间复杂度最低？能否给出证明？

### 考到了kmp(问空间复杂度）、循环队列，

### 有森林转二叉树的题

### 有找出无向图邻接矩阵，

### 1-100，100个数，随意添一个重复的数，怎么快速求出这个数

### 一个场景，有1000个文件，每个文件有1000w行，每一行是一个数，求所有数据的中位数（我只说了用排序算法，没回答到点上）

### "9. 算法题：查找范围内不存在的一个数，

### 1. 解法一：异或，第一想法

### 2. 还让说别的方法，我说了用哈希表存，然后匹配，显然不合适

### 3. 解法二：求和相减"

### 用过双向dijkstra吗

### "有一个无序数组，如何构建二叉查找树

### 极端情况下退化成链表，怎么解决（平衡树：AVL或者红黑树）

### 你能用代码实现平衡二叉树吗（直接无语，撸了半天代码把除了旋转之外的代码全部写好了，然后开摆）

### 口述AVL的单旋与双旋

### 口述如何层序遍历，讲了半天没讲明白，直接撸代码

### 有其他方法层序遍历吗"

### c++实现全排列都有哪些方法（只回答了遍历和回溯，应该是想听到next\_permutation()系统函数的使用吧）

### 堆排序，建堆的时间复杂度

### "跳表的数据结构，查询复杂度

### 是否可以适用跳表来作为内存索引"

### 最左前缀了解吗？忘了，我说我听过最左优先匹配

### 万字长文！滑动窗口看这篇就够了！

### 扔鸡蛋问题

## 3、操作系统

### 1、进程、线程和协程的区别和联系

|  |  |  |  |
| --- | --- | --- | --- |
|  | **进程** | **线程** | **协程** |
| 定义 | 资源分配和拥有的基本单位 | 程序执行的基本单位 | 用户态的轻量级线程，线程内部调度的基本单位 |
| 切换情况 | 进程CPU环境(栈、寄存器、页表和文件句柄等)的保存以及新调度的进程CPU环境的设置 | 保存和设置程序计数器、少量寄存器和栈的内容 | 先将寄存器上下文和栈保存，等切换回来的时候再进行恢复 |
| 切换者 | 操作系统 | 操作系统 | 用户 |
| 切换过程 | 用户态->内核态->用户态 | 用户态->内核态-> 用户态 | 用户态(没有陷入内核) |
| 调用栈 | 内核栈 | 内核栈 | 用户栈 |
| 拥有资源 | CPU资源、内存资源、文件资源和句柄等 | 程序计数器、寄存器、栈和状态字 | 拥有自己的寄存器上下文和栈 |
| 并发性 | 不同进程之间切换实现并发， 各自占有CPU实现并行 | 一个进程内部的多个线程并发执行 | 同一时间只能执行一个协程，而其他协程处于休眠状态，适合对任务进行分时处理 |
| 系统开销 | 切换虚拟地址空间，切换内核栈和硬件上下文，CPU高速缓存失效、页表切换，开销很大 | 切换时只需保存和设置少量寄存器内容，因此开销很小 | 直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快 |
| 通信方面 | 进程间通信需要借助操作系统 | 线程间可以直接读写进程数据段  (如全局变量)来进行通信 | 共享内存、消息队列 |

1、进程是资源调度的基本单位，运行一个可执行程序会创建一个或多个进程，进程就是运行起来的可执行程序

2、线程是程序执行的基本单位，是轻量级的进程。每个进程中都有唯一的主线程，且只能有一个，主线程和进程是相互依存的关系，主线程结束进程也会结束。多提一句：协程是用户态的轻量级线程，线程内部调度的基本单位

### 2、线程与进程的比较

1、线程启动速度快，轻量级

2、线程的系统开销小

3、线程使用有一定难度，需要处理数据一致性问题

4、同一线程共享的有堆、全局变量、静态变量、指针，引用、文件等，而独自占有

### 3、一个进程可以创建多少线程，和什么有关？

理论上，一个进程可用虚拟空间是2G，默认情况下，线程的栈的大小是1MB，所以理论上最多只能创 建2048个线程。如果要创建多于2048的话，必须修改编译器的设置。

因此，一个进程可以创建的线程数由可用虚拟空间和线程的栈的大小共同决定，只要虚拟空间足够，那 么新线程的建立就会成功。如果需要创建超过2K以上的线程，减小你线程栈的大小就可以实现了，虽然 在一般情况下，你不需要那么多的线程。过多的线程将会导致大量的时间浪费在线程切换上，给程序运 行效率带来负面影响。

《一个进程到底能创建多少线程》：<https://www.cnblogs.com/Leo_wl/p/5969621.html>

### 4、外中断和异常有什么区别？

外中断是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

而异常时由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

### 5、进程线程模型你知道多少？

对于进程和线程的理解和把握可以说基本奠定了对系统的认知和把控能力。其核心意义绝不仅仅是“线 程是调度的基本单位，进程是资源分配的基本单位”这么简单。

##### 多线程

我们这里讨论的是用户态的多线程模型，同一个进程内部有多个线程，所有的线程共享同一个进程的内 存空间，进程中定义的全局变量会被所有的线程共享，比如有全局变量int i = 10，这一进程中所有并发运行的线程都可以读取和修改这个i的值，而多个线程被CPU调度的顺序又是不可控的，所以对临界资源 的访问尤其需要注意安全。

我们必须知道，**做一次简单的i = i + 1在计算机中并不是原子操作，涉及内存取数，计算和写入内存几个环节，**而线程的切换有可能发生在上述任何一个环节中间，所以不同的操作顺序很有可能带来意想不 到的结果。

但是，虽然线程在安全性方面会引入许多新挑战，但是线程带来的好处也是有目共睹的。首先，原先顺 序执行的程序（暂时不考虑多进程）可以被拆分成几个独立的逻辑流，这些逻辑流可以独立完成一些任 务（最好这些任务是不相关的）。

比如 QQ 可以一个线程处理聊天一个线程处理上传文件，两个线程互不干涉，在用户看来是同步在执行两个任务，试想如果线性完成这个任务的话，在数据传输完成之前用户聊天被一直阻塞会是多么尴尬的 情况。

对于线程，我认为弄清以下两点非常重要：

线程之间有无先后访问顺序（线程依赖关系） 多个线程共享访问同一变量（同步互斥问题）

另外，我们通常只会去说同一进程的多个线程共享进程的资源，但是每个线程特有的部分却很少提及， 除了标识线程的tid，每个线程还有自己独立的栈空间，线程彼此之间是无法访问其他线程栈上内容的。

而作为处理机调度的最小单位，线程调度只需要保存线程栈、寄存器数据和PC即可，相比进程切换开销 要小很多。

线程相关接口不少，主要需要了解各个参数意义和返回值意义。

1. 线程创建和结束

背景知识：

在一个文件内的多个函数通常都是按照main函数中出现的顺序来执行，但是在分时系统下， 我们可以让每个函数都作为一个逻辑流并发执行，最简单的方式就是采用多线程策略。在main函数中调用多线程接口创建线程，每个线程对应特定的函数（操作），这样就可以不按 照main函数中各个函数出现的顺序来执行，避免了忙等的情况。线程基本操作的接口如下。

相关接口：

创建线程：int pthread\_create(pthread\_t \*pthread, const pthread\_attr\_t \*attr, void

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

创建一个新线程，pthread和start\_routine不可或缺，分别用于标识线程和执行体入 口，其他可以填NULL。

pthread：用来返回线程的tid，\*pthread值即为tid，类型pthread\_t == unsigned long int 。attr：指向线程属性结构体的指针，用于改变所创线程的属性，填NULL使用默认 值。

start\_routine：线程执行函数的首地址，传入函数指针。 arg：通过地址传递来传递函数参数，这里是无符号类型指针，可以传任意类型变 量的地址，在被传入函数中先强制类型转换成所需类型即可。

获得线程ID：pthread\_t pthread\_self();

调用时，会打印线程ID。

等待线程结束：int pthread\_join(pthread\_t tid, void\*\* retval);

主线程调用，等待子线程退出并回收其资源，类似于进程中wait/waitpid回收僵尸进 程，调用pthread\_join的线程会被阻塞。

tid：创建线程时通过指针得到tid值。retval：指向返回值的指针。

结束线程：pthread\_exit(void \*retval);

子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过

pthread\_join获得。

retval：同上。

分离线程：int pthread\_detach(pthread\_t tid);

主线程、子线程均可调用。主线程中pthread\_detach(tid)，子线程中pthread\_detach(pthread\_self())，调用后和主线程分离，子线程结束时自己立即回收资 源。

tid：同上。

1. 线程属性值修改

背景知识：

线程属性对象类型为pthread\_attr\_t，结构体定义如下：

size\_t stacksize; // 线程栈大小

}pthread\_arrt\_t;

stackaddr; // 线程栈的位置

void \*

// 以下为线程栈的设置

size\_t guardsize; // 线程栈末尾警戒缓冲大小

int stackaddr\_set; // 线程的栈设置

// 线程的作用域

int scope;

// 线程的调度参数

struct sched\_param schedparam;

int inheritsched; // 线程的继承性

// 线程分离的状态

// 线程调度策略

typedef struct{ int etachstate;

int schedpolicy;

1

2

3

4

5

6

7

8

9

10

11

12

13

##### 多进程

相关接口：

对上述结构体中各参数大多有：pthread\_attr\_get()和pthread\_attr\_set()系统调用函数来设置 和获取。这里不一一罗列。

每一个进程是资源分配的基本单位。

进程结构由以下几个部分组成：代码段、堆栈段、数据段。代码段是静态的二进制代码，多个程序可以 共享。

实际上在父进程创建子进程之后，父、子进程除了pid外，几乎所有的部分几乎一样。

父、子进程共享全部数据，但并不是说他们就是对同一块数据进行操作，子进程在读写数据时会通过写 时复制机制将公共的数据重新拷贝一份，之后在拷贝出的数据上进行操作。

如果子进程想要运行自己的代码段，还可以通过调用execv()函数重新加载新的代码段，之后就和父进程 独立开了。

我们在shell中执行程序就是通过shell进程先fork()一个子进程再通过execv()重新加载新的代码段的过 程。

1. 进程创建与结束

背景知识：

进程有两种创建方式，一种是操作系统创建的一种是父进程创建的。从计算机启动到终端执 行程序的过程为：0号进程 -> 1号内核进程 -> 1号用户进程(init进程) -> getty进程 -> shell进程 -> 命令行执行进程。所以我们在命令行中通过 ./program执行可执行文件时，所有创建的进程都是shell进程的子进程，这也就是为什么shell一关闭，在shell中执行的进程都自动被关 闭的原因。从shell进程到创建其他子进程需要通过以下接口。

相关接口：

创建进程：pid\_t fork(void);

返回值：出错返回-1；父进程中返回pid > 0；子进程中pid == 0

结束进程：void exit(int status);

status是退出状态，保存在全局变量中S?，通常0表示正常退出。获得PID：pid\_t getpid(void);

返回调用者pid。

获得父进程PID：pid\_t getppid(void);

返回父进程pid。

其他补充：

正常退出方式：exit()、\_exit()、return（在main中）。

exit()和*exit()*区别：*exit()*是对*\_*exit()的封装，都会终止进程并做相关收尾工作，最主要的 区别是\_exit()函数关闭全部描述符和清理函数后不会刷新流，但是exit()会在调用\_exit() 函数前刷新数据流。

return和exit()区别：exit()是函数，但有参数，执行完之后控制权交给系统。return若 是在调用函数中，执行完之后控制权交给调用进程，若是在main函数中，控制权交给系 统。

异常退出方式：abort()、终止信号。

1. Linux进程控制

进程地址空间（地址空间）

虚拟存储器为每个进程提供了独占系统地址空间的假象。

尽管每个进程地址空间内容不尽相同，但是他们的都有相似的结构。X86 Linux进程的地址空间底部是保留给用户程序的，包括文本、数据、堆、栈等，其中文本区和数据区是通过存储器映射方式 将磁盘中可执行文件的相应段映射至虚拟存储器地址空间中。

有一些"敏感"的地址需要注意下，对于32位进程来说，代码段从0x08048000开始。从

0xC0000000开始到0xFFFFFFFF是内核地址空间，通常情况下代码运行在用户态（使用

0x00000000 ~ 0xC00000000的用户地址空间），当发生系统调用、进程切换等操作时CPU控制寄存器设置模式位，进入内和模式，在该状态（超级用户模式）下进程可以访问全部存储器位置和 执行全部指令。

也就说32位进程的地址空间都是4G，但用户态下只能访问低3G的地址空间，若要访问3G ~ 4G的地址空间则只有进入内核态才行。

进程控制块（处理机）

进程的调度实际就是内核选择相应的进程控制块，被选择的进程控制块中包含了一个进程基本的信 息。

上下文切换

内核管理所有进程控制块，而进程控制块记录了进程全部状态信息。每一次进程调度就是一次上下 文切换，所谓的上下文本质上就是当前运行状态，主要包括通用寄存器、浮点寄存器、状态寄存 器、程序计数器、用户栈和内核数据结构（页表、进程表、文件表）等。

进程执行时刻，内核可以决定抢占当前进程并开始新的进程，这个过程由内核调度器完成，当调度 器选择了某个进程时称为该进程被调度，该过程通过上下文切换来改变当前状态。

一次完整的上下文切换通常是进程原先运行于用户态，之后因系统调用或时间片到切换到内核态执 行内核指令，完成上下文切换后回到用户态，此时已经切换到进程B。

### 6、进程调度算法你了解多少？

##### 1、 先来先服务 ﬁrst-come ﬁrst-serverd（FCFS）

非抢占式的调度算法，按照请求的顺序进行调度。

有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业 又需要执行很长时间，造成了短作业等待时间过长。

##### 2、 短作业优先 shortest job ﬁrst（SJF）

非抢占式的调度算法，按估计运行时间最短的顺序进行调度。

长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业 永远得不到调度。

##### 3、最短剩余时间优先 shortest remaining time next（SRTN）

最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。 当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。

如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。

4、**时间片轮转**

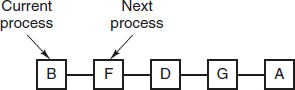
将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。

当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末 尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：

因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太 频繁，在进程切换上就会花过多时间。

而如果时间片过长，那么实时性就不能得到保证。



5、**优先级调度**

为每个进程分配一个优先级，按优先级进行调度。

为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

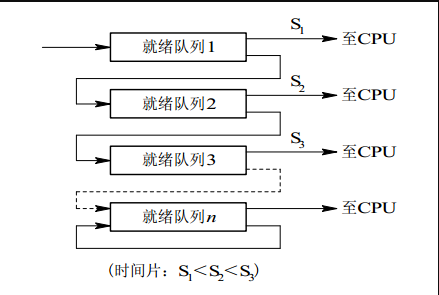
6、**多级反馈队列**

一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。

多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不 同，例如 1,2,4,8,..。进程在第一个队列没执行完，就会被移到下一个队列。

这种方式下，之前的进程只需要交换 7 次。每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。



### 7、Linux下进程间通信方式？

管道：

无名管道（内存文件）：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具 有亲缘关系的进程之间使用。进程的亲缘关系通常是指父子进程关系。

有名管道（FIFO文件，借助文件系统）：有名管道也是半双工的通信方式，但是允许在没有 亲缘关系的进程之间使用，管道是先进先出的通信方式。

共享内存：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但 多个进程都可以访问。共享内存是最快的IPC方式，它是针对其他进程间通信方式运行效率低而专 门设计的。它往往与信号量，配合使用来实现进程间的同步和通信。

消息队列：消息队列是有消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信 号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

套接字：适用于不同机器间进程通信，在本地也可作为两个进程通信的方式。

信号：用于通知接收进程某个事件已经发生，比如按下ctrl + C就是信号。

信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制， 实现进程、线程的对临界区的同步及互斥访问。

### 8、Linux下同步机制？

POSIX信号量：可用于进程同步，也可用于线程同步。POSIX互斥锁 + 条件变量：只能用于线程同步。

1. 线程和进程的区别？

调度：线程是调度的基本单位（PC，状态码，通用寄存器，线程栈及栈指针）；进程是拥有 资源的基本单位（打开文件，堆，静态区，代码段等）。

并发性：一个进程内多个线程可以并发（最好和CPU核数相等）；多个进程可以并发。

拥有资源：线程不拥有系统资源，但一个进程的多个线程可以共享隶属进程的资源；进程是 拥有资源的独立单位。

系统开销：线程创建销毁只需要处理PC值，状态码，通用寄存器值，线程栈及栈指针即可； 进程创建和销毁需要重新分配及销毁task\_struct结构。

### 9、如果系统中具有快表后，那么地址的转换过程变成什么样了？

①CPU给出逻辑地址，由某个硬件算得页号、页内偏移量，将页号与快表中的所有页号进行比 较。②如果找到匹配的页号，说明要访问的页表项在快表中有副本，则直接从中取出该页对应的 内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单 元。因此，若快表命中，则访问某个逻辑地址仅需一次访存即可。

③如果没有找到匹配的页号，则需要访问内存中的页表，找到对应页表项，得到页面存放的内存 块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。 因此,若快表未命中，则访问某个逻辑地址需要两次访存(注意:在找到页表项后，应同时将其存入 快表,以便后面可能的再次访问。但若快表已满，则必须按照-定的算法对旧的页表项进行替换)

由于查询快表的速度比查询页表的速度快很多，因此只要快表命中，就可以节省很多时间。 因为局部性原理，–般来说快表的命中率可以达到90%以上。

例:某系统使用基本分页存储管理，并采用了具有快表的地址变换机构。访问- -次快表耗时1us， 访问一次内存耗时100us。若快表的命中率为90%，那么访问一个逻辑地址的平均耗时是多少?

(1+100) \* 0.9 + (1+100+100) \* 0.1 = 111 us

有的系统支持快表和慢表同时查找，如果是这样，平均耗时应该是(1+100) \* 0.9+ (100+100)

\*0.1=110.9 us

若未采用快表机制，则访问一个逻辑地址需要100+100 = 200us

显然，引入快表机制后，访问一个逻辑地址的速度快多了。

### 10、内存交换和覆盖有什么区别？

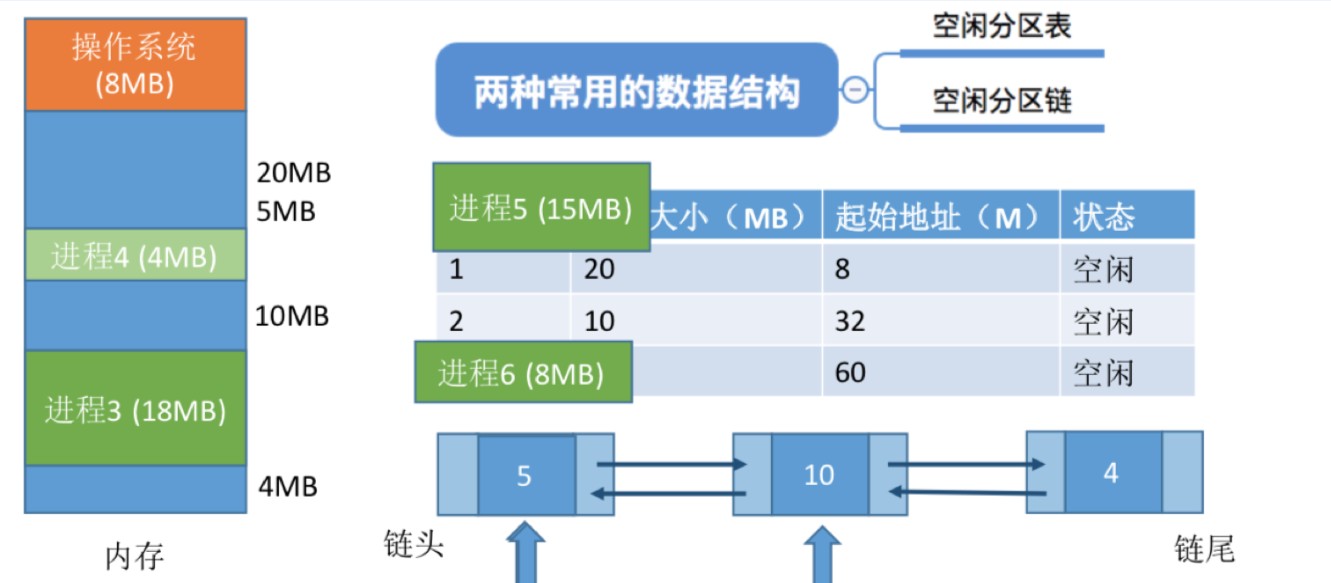
交换技术主要是在不同进程（或作业）之间进行，而覆盖则用于同一程序或进程中。

### 11、动态分区分配算法有哪几种？可以分别说说吗？

##### 1、首次适应算法

算法思想：每次都从低地址开始查找，找到第–个能满足大小的空闲分区。

如何实现：空闲分区以地址递增的次序排列。每次分配内存时顺序查找空闲分区链( 或空闲分[表)，找到大小能满足要求的第-一个空闲分区。



##### 2、最佳适应算法

算法思想:由于动态分区分配是一种连续分配方式，为各进程分配的空间必须是连续的一整片区域。因此 为了保证当“大进程”到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区,即，优先使用更小 的空闲区。

如何实现:空闲分区按容量递增次序链接。每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大 小能满足要求的第-一个空闲分区。



##### 3、最坏适应算法

又称最大适应算法(Largest Fit)

算法思想:为了解决最佳适应算法的问题—即留下太多难以利用的小碎片，可以在每次分配时优先使用最 大的连续空闲区，这样分配后剩余的空闲区就不会太小，更方便使用。

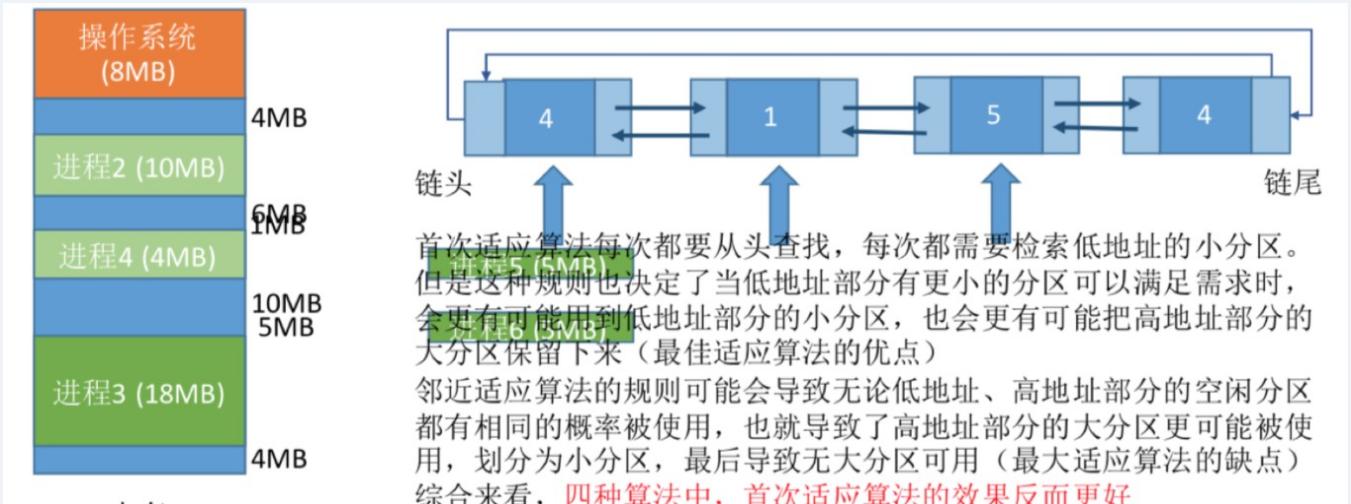
如何实现:空闲分区按容量递减次序链接。每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大 小能满足要求的第-一个空闲分区。



##### 4、邻近适应算法

算法思想：首次适应算法每次都从链头开始查找的。这可能会导致低地址部分出现很多小的空闲分区， 而每次分配查找时，都要经过这些分区，因此也增加了查找的开销。如果每次都从上次查找结束的位置 开始检索，就能解决上述问题。

如何实现：空闲分区以地址递增的顺序排列(可排成-一个循环链表)。每次分配内存时从上次查找结束的 位置开始查找空闲分区链(或空闲分区表)，找到大小能满足要求的第一个空闲分区。



##### 5、总结

首次适应不仅最简单，通常也是最好最快，不过首次适应算法会使得内存低地址部分出现很多小的空闲 分区，而每次查找都要经过这些分区，因此也增加了查找的开销。邻近算法试图解决这个问题，但实际 上，它常常会导致在内存的末尾分配空间分裂成小的碎片，它通常比首次适应算法结果要差。

最佳导致大量碎片，最坏导致没有大的空间。

进过实验，首次适应比最佳适应要好，他们都比最坏好。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **算法** | **算法思想** | **分区排列顺序** | **优点** | **缺点** |
| 首次适应 | 从头到尾找适合的分区 | 空闲分区以地址递增次序排列 | 综合看性能最好。**算法开销小**，回收分区后一.般不需要对空闲分区队列重新排序 |  |
| 最佳适应 | 优先使用更小的分区，以保留更多大分区 | 空闲分区以容量递增次序排列 | 会有更多的大分区被保留下来，更能满足大进程需求 | 会产生很多太小的、难以利用的碎片;**算法开销大**，回收分区后可能需要对空闲分区队列重新排序 |
| 最坏适应 | 优先使用更大的分区，以防止产生太小的不可用的碎片 | 空闲分区以容量递减次序排列 | 可以减少难以利用的小碎片 | 大分区容易被用完，不利于大进程;**算法开销大**(原因同上) |
| 邻近适应 | 由首次适应演变而来，每次从上次查找结束位置开始查找 | 空闲分区以地址递增次序排列(可排列成循环链表) | 不用每次都从低地址的小分区开始检索。**算法开销小**(原因同首次适应算法) | 会使高地址的大分区也被用完 |

### 12、虚拟技术你了解吗？

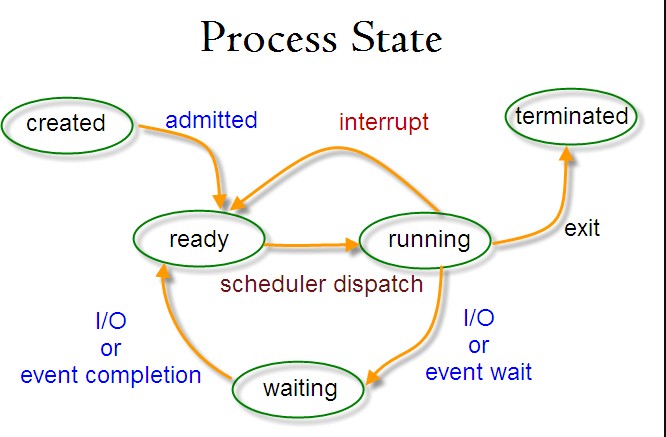
虚拟技术把一个物理实体转换为多个逻辑实体。

主要有两种虚拟技术：时（时间）分复用技术和空（空间）分复用技术。

多进程与多线程：多个进程能在同一个处理器上并发执行使用了时分复用技术，让每个进程轮流占用处 理器，每次只执行一小个时间片并快速切换。

虚拟内存使用了空分复用技术，它将物理内存抽象为地址空间，每个进程都有各自的地址空间。地址空 间的页被映射到物理内存，地址空间的页并不需要全部在物理内存中，当使用到一个没有在物理内存的 页时，执行页面置换算法，将该页置换到内存中。

### 13、进程状态的切换你知道多少？



就绪状态（ready）：等待被调度运行状态（running）

阻塞状态（waiting）：等待资源

应该注意以下内容：

只有就绪态和运行态可以相互转换，其它的都是单向转换。就绪状态的进程通过调度算法从而获得CPU 时间，转为运行状态；而运行状态的进程，在分配给它的 CPU 时间片用完之后就会转为就绪状态，等待下一次调度。

阻塞状态是缺少需要的资源从而由运行状态转换而来，但是该资源不包括 CPU 时间，缺少 CPU 时间会从运行态转换为就绪态。

### 14、一个程序从开始运行到结束的完整过程，你能说出来多少？

四个过程：

##### 预编译

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下

1、删除所有的#deﬁne，展开所有的宏定义。

2、处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。

3、处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他 文件。

4、删除所有的注释，“//”和“/\*\*/”。

5、保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重

复引用。

6、添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是 能够显示行号。

##### 编译

把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相 应

的汇编代码文件。

1、词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分 割成一系列的记号。

2、语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的 语法树是一种以表达式为节点的树。

3、语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进 行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定 的语义。

4、优化：源代码级别的一个优化过程。

5、目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语 言

表示。

6、目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移 来替代乘法运算、删除多余的指令等。

##### 汇编

将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过 来，汇编过程有汇编器as完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Linux 下)、xxx.obj(Windows下)。

##### 链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链 接：

1、静态链接：

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库 中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个 目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本；

更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西， 在执行的时候运行速度快。

2、动态链接：

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形 成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多份副 本，而是这多个程序在执行时共享同一份副本；

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运 行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损 失。

[《操作系统（三）》：https://www.nowcoder.com/tutorial/93/675fd4af3ab34b2db0ae65085 5aa52d5](https://www.nowcoder.com/tutorial/93/675fd4af3ab34b2db0ae650855aa52d5)

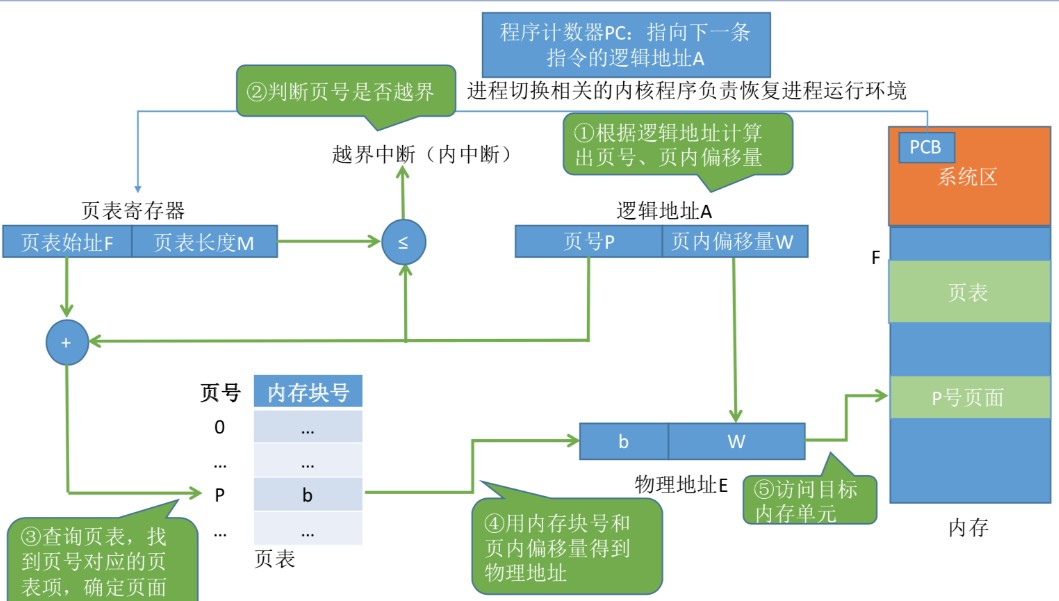
### 15、通过例子讲解逻辑地址转换为物理地址的基本过程

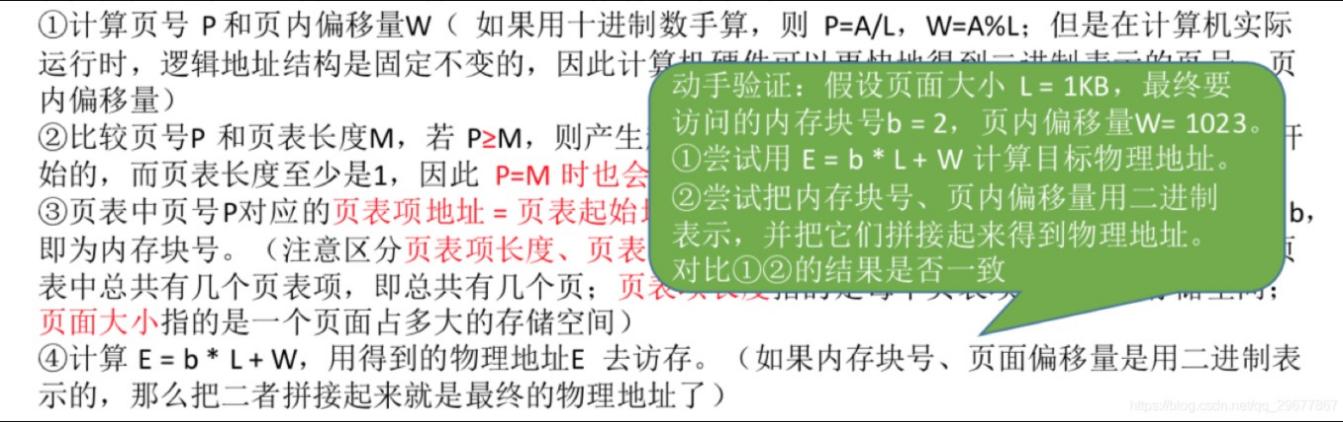
可以借助进程的页表将逻辑地址转换为物理地址。

通常会在系统中设置一个页表寄存器(PTR)，存放页表在内存中的起始地址F和页表长度M。进程未执行 时，页表的始址和页表长度放在进程控制块(PCB) 中，当进程被调度时，操作系统内核会把它们放到页表寄存器中。

注意:页面大小是2的整数幂

设页面大小为L，逻辑地址A到物理地址E的变换过程如下:





例:若页面大小L为1K字节，页号2对应的内存块号b=8，将逻辑地址A=2500转换为物理地址E。

等价描述：某系统按字节寻址，逻辑地址结构中，页内偏移量占10位(说明一个页面的大小为2^10B = 1KB)，页号2对应的内存块号 b=8，将逻辑地址A=2500转换为物理地址E。

①计算页号、页内偏移量

页号P=A/L = 2500/1024 = 2; 页内偏移量W= A%L = 2500%1024 = 452

②根据题中条件可知，页号2没有越界，其存放的内存块号b=8

③物理地址E=b\*L+W=8 \* 1024+ 425 = 8644

在分页存储管理(页式管理)的系统中，只要确定了每个页面的大小，逻辑地址结构就确定了。因 此，页式管理中地址是-维的。即，只要给出一个逻辑地址，系统就可以自动地算出页号、页内偏 移量两个部分，并不需要显式地告诉系统这个逻辑地址中，页内偏移量占多少位。

### 16、进程同步的四种方法？

##### 临界区

对临界资源进行访问的那段代码称为临界区。

为了互斥访问临界资源，每个进程在进入临界区之前，需要先进行检查。

|  |
| --- |
| 1. // entry section 2. // critical section; 3. // exit section 4 |

##### 同步与互斥

同步：多个进程因为合作产生的直接制约关系，使得进程有一定的先后执行关系。 互斥：多个进程在同一时刻只有一个进程能进入临界区。

##### 信号量

信号量（Semaphore）是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。

**down** : 如果信号量大于 0 ，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；

**up** ：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。

down 和 up 操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为 0 或者 1，那么就成为了 **互斥量（Mutex）** ，0 表示临界区已经加锁，1 表示临界区解锁。

|  |
| --- |
| 1. typedef int semaphore; 2. semaphore mutex = 1; 3. void P1() { 4. down(&mutex); 5. // 临界区 6. up(&mutex); 7 }   8  9 void P2() {   1. down(&mutex); 2. // 临界区 3. up(&mutex); 13 }   14 |

### 17 使用信号量实现生产者-消费者问题

问题描述：使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为 空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量 mutex 来控制对缓冲区的互斥访问。

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这 里需要使用两个信号量：empty 记录空缓冲区的数量，full 记录满缓冲区的数量。

其中，empty 信号量是在生产者进程中使用，当 empty 不为 0 时，生产者才可以放入物品；full 信号量是在消费者进程中使用，当 full 信号量不为 0 时，消费者才可以取走物品。

**注意**，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 down(mutex) 再执行down(empty)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行down(empty) 操作，发现 empty = 0，此时生产者睡眠。

消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 up(empty) 操作，empty 永远都为 0，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

#define N 100

typedef int semaphore;

semaphore mutex = 1;

semaphore empty = N;

semaphore full = 0;

void producer() {

while (*TRUE*) {

int item = produce\_item();

down(&empty);

down(&mutex);

insert\_item(item);

up(&mutex);

up(&full);

}

}

void consumer() {

while (*TRUE*) {

down(&full);

down(&mutex);

int item = remove\_item();

consume\_item(item);

up(&mutex);

up(&empty);

}

}

1. 管程

使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来， 不仅不容易出错，也使得客户端代码调用更容易。

c 语言不支持管程，下面的示例代码使用了类 Pascal 语言来描述管程。示例代码的管程提供了 insert()

和 remove() 方法，客户端代码通过调用这两个方法来解决生产者-消费者问题。

|  |
| --- |
| 1. monitor ProducerConsumer 2. integer i; 3. condition c; 4 4. procedure insert(); 5. begin   7 // ...  8 end;  9   1. procedure remove(); 2. begin   12 // ...   1. end; 2. end monitor; 15 |

管程有一个重要特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用 管程，否则其它进程永远不能使用管程。

管程引入了 **条件变量** 以及相关的操作：**wait()** 和 **signal()** 来实现同步操作。对条件变量执行 wait()

操作会导致调用进程阻塞，把管程让出来给另一个进程持有。signal() 操作用于唤醒被阻塞的进程。

### 18、操作系统在对内存进行管理的时候需要做些什么?

操作系统负责内存空间的分配与回收。

操作系统需要提供某种技术从逻辑上对内存空间进行扩充。

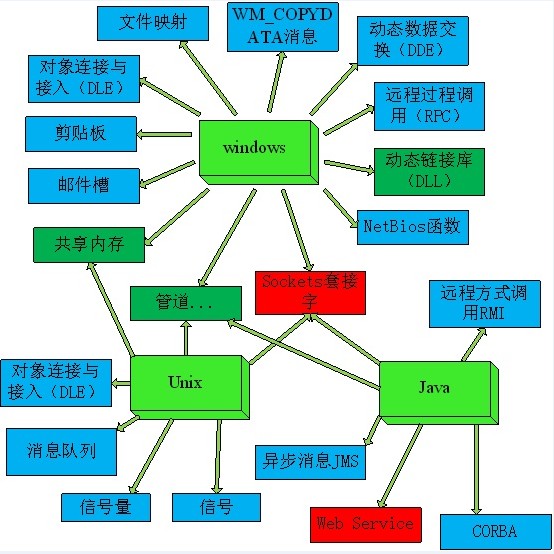
操作系统需要提供地址转换功能，负责程序的逻辑地址与物理地址的转换。

操作系统需要提供内存保护功能。保证各进程在各自存储空间内运行，互不干扰

### 19、进程通信方法（Linux和windows下），线程通信方法（Linux和windows

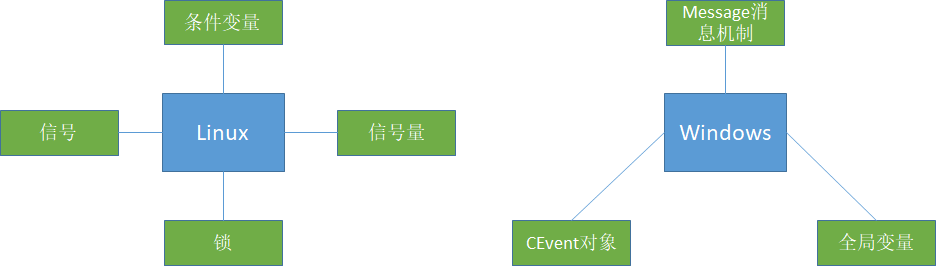
**下）**

##### 进程通信方法



|  |
| --- |
| **名称及方式** |
| 管道(pipe)：允许一个进程和另一个与它有共同祖先的进程之间进行通信 |
| 命名管道(FIFO)：类似于管道，但是它可以用于任何两个进程之间的通信，命名管道在文件系统中 有对应的文件名。命名管道通过命令mkﬁfo或系统调用mkﬁfo来创建 |
| 消息队列(MQ)：消息队列是消息的连接表，包括POSIX消息对和System V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信 号承载信息量少，管道只能成该无格式字节流以及缓冲区大小受限等缺点； |
| 信号量(semaphore)：信号量主要作为进程间以及同进程不同线程之间的同步手段； |
| 共享内存(shared memory)：它使得多个进程可以访问同一块内存空间，**是最快的可用IPC形式。**这是针对其他通信机制运行效率较低而设计的。它往往与其他通信机制，如信号量结合使用，以达 到进程间的同步及互斥 |
| 信号(signal)：信号是比较复杂的通信方式，用于通知接收进程有某种事情发生，除了用于进程间 通信外，进程还可以发送信号给进程本身 |
| 内存映射(mapped memory)：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它 |
| Socket：它是更为通用的进程间通信机制，可用于不同机器之间的进程间通信 |

**线程通信方法**



|  |
| --- |
| **名称及含义** |
| **Linux：** |
| 信号：类似进程间的信号处理 |
| 锁机制：互斥锁、读写锁和自旋锁 |
| 条件变量：使用通知的方式解锁，与互斥锁配合使用 |
| 信号量：包括无名线程信号量和命名线程信号量 |
| **Windows：** |
| 全局变量：需要有多个线程来访问一个全局变量时，通常我们会在这个全局变量前加上volatile声 明，以防编译器对此变量进行优化 |
| Message消息机制：常用的Message通信的接口主要有两个：PostMessage和PostThreadMessage，PostMessage为线程向主窗口发送消息。而PostThreadMessage是任意两 个线程之间的通信接口。 |
| CEvent对象：CEvent为MFC中的一个对象，可以通过对CEvent的触发状态进行改变，从而实现线 程间的通信和同步，这个主要是实现线程直接同步的一种方法。 |

### 20、进程间通信有哪几种方式？把你知道的都说出来

Linux几乎支持全部UNIX进程间通信方法，包括管道（有名管道和无名管道）、消息队列、共享内存、 信号量和套接字。其中前四个属于同一台机器下进程间的通信，套接字则是用于网络通信。

##### 管道

无名管道

无名管道特点：

无名管道是一种特殊的文件，这种文件只存在于内存中。

无名管道只能用于父子进程或兄弟进程之间，必须用于具有亲缘关系的进程间的通信。 无名管道只能由一端向另一端发送数据，是半双工方式，如果双方需要同时收发数据需 要两个管道。

相关接口：

int pipe(int fd[2]);

fd[2]：管道两端用fd[0]和fd[1]来描述，读的一端用fd[0]表示，写的一端用fd[1]表 示。通信双方的进程中写数据的一方需要把fd[0]先close掉，读的一方需要先把fd[1]给close掉。

有名管道：

有名管道特点：

有名管道是FIFO文件，存在于文件系统中，可以通过文件路径名来指出。有名管道可以在不具有亲缘关系的进程间进行通信。

相关接口：

int mkﬁfo(const char \*pathname, mode\_t mode);

pathname：即将创建的FIFO文件路径，如果文件存在需要先删除。mode：和open()中的参数相同。

##### 消息队列

相比于 FIFO，消息队列具有以下优点：

消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；

避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；

读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。

##### 共享内存

进程可以将同一段共享内存连接到它们自己的地址空间，所有进程都可以访问共享内存中的地址，如果 某个进程向共享内存内写入数据，所做的改动将立即影响到可以访问该共享内存的其他所有进程。

相关接口

创建共享内存：int shmget(key\_t key, int size, int ﬂag);

成功时返回一个和key相关的共享内存标识符，失败范湖范围-1。

key：为共享内存段命名，多个共享同一片内存的进程使用同一个key。size：共享内存容量。

ﬂag：权限标志位，和open的mode参数一样。

连接到共享内存地址空间：void \*shmat(int shmid, void \*addr, int ﬂag);

返回值即共享内存实际地址。

shmid：shmget()返回的标识。addr：决定以什么方式连接地址。ﬂag：访问模式。

从共享内存分离：int shmdt(const void \*shmaddr);

调用成功返回0，失败返回-1。

shmaddr：是shmat()返回的地址指针。

其他补充

共享内存的方式像极了多线程中线程对全局变量的访问，大家都对等地有权去修改这块内存的值， 这就导致在多进程并发下，最终结果是不可预期的。所以对这块临界区的访问需要通过信号量来进 行进程同步。

但共享内存的优势也很明显，首先可以通过共享内存进行通信的进程不需要像无名管道一样需要通 信的进程间有亲缘关系。其次内存共享的速度也比较快，不存在读取文件、消息传递等过程，只需 要到相应映射到的内存地址直接读写数据即可。

##### 信号量

在提到共享内存方式时也提到，进程共享内存和多线程共享全局变量非常相似。所以在使用内存共享的 方式是也需要通过信号量来完成进程间同步。多线程同步的信号量是POSIX信号量，而在进程里使用SYSTEM V信号量。

相关接口

创建信号量：int semget(key\_t key, int nsems, int semﬂag);

创建成功返回信号量标识符，失败返回-1。

key： 进 程 pid 。 nsems：创建信号量的个数。semﬂag：指定信号量读写权限。

改变信号量值：int semop(int semid, struct sembuf \*sops, unsigned nsops);

我们所需要做的主要工作就是串讲sembuf变量并设置其值，然后调用semop，把设置好的

sembuf变量传递进去。

struct sembuf结构体定义如下：

1. struct sembuf{
2. short sem\_num;
3. short sem\_op;
4. short sem\_flg; 5 };

成功返回信号量标识符，失败返回-1。

semid：信号量集标识符，由semget()函数返回。

sops：指向struct sembuf结构的指针，先设置好sembuf值再通过指针传递。nsops：进行操作信号量的个数，即sops结构变量的个数，需大于或等于1。最常见设 置此值等于1，只完成对一个信号量的操作。

直接控制信号量信息：int semctl(int semid, int semnum, int cmd, union semun arg);

semid： 信 号 量 集 标 识 符 。 semnum：信号量集数组上的下标，表示某一个信号量。arg：union semun类型。

##### 辅助命令

ipcs命令用于报告共享内存、信号量和消息队列信息。

ipcs -a：列出共享内存、信号量和消息队列信息。ipcs -l：列出系统限额。

ipcs -u：列出当前使用情况。

##### 套接字

与其它通信机制不同的是，它可用于不同机器间的进程通信。

### 21、虚拟内存的目的是什么？

虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。

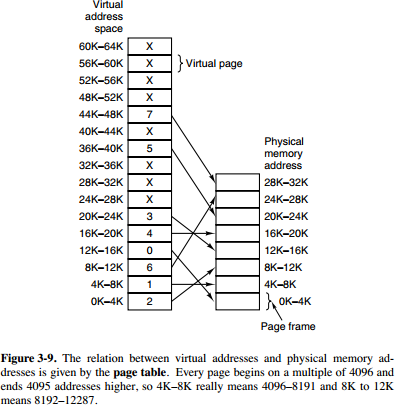
为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间 被分割成多个块，每一块称为一页。

这些页被映射到物理内存，但不需要映射到连续的物理内存，也不需要所有页都必须在物理内存中。当 程序引用到不在物理内存中的页时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失 败的指令。

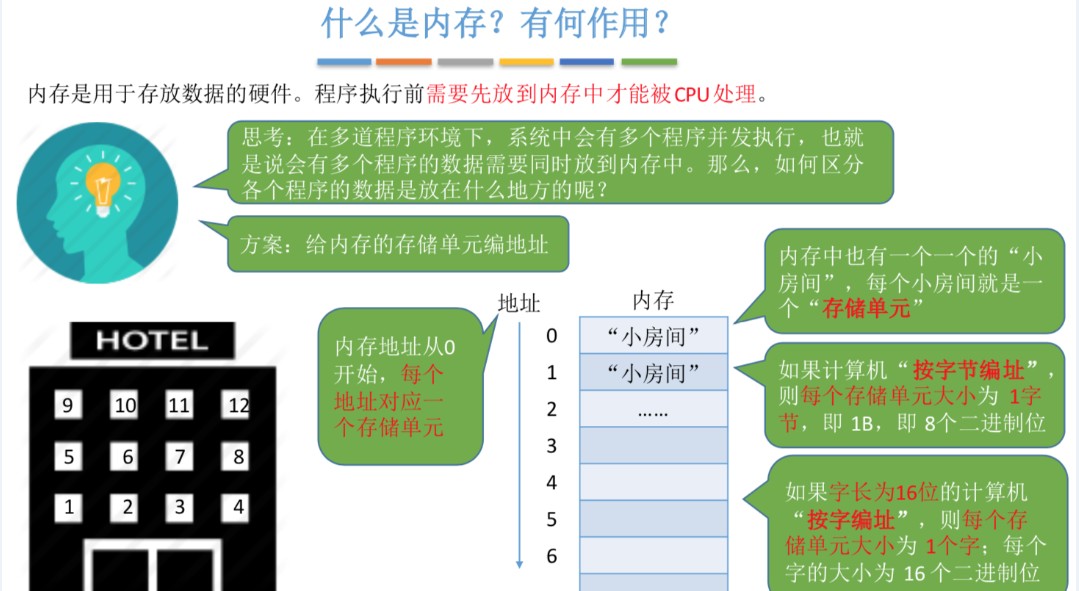
从上面的描述中可以看出，虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说 一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。

例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB

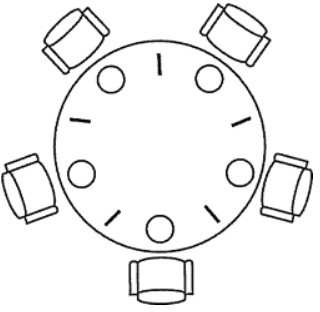
的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。



### 22、说一下你理解中的内存？他有什么作用呢？



### 23、操作系统经典问题之哲学家进餐问题



五个哲学家围着一张圆桌，每个哲学家面前放着食物。哲学家的生活有两种交替活动：吃饭以及思考。 当一个哲学家吃饭时，需要先拿起自己左右两边的两根筷子，并且一次只能拿起一根筷子。

下面是一种错误的解法，如果所有哲学家同时拿起左手边的筷子，那么所有哲学家都在等待其它哲学家 吃完并释放自己手中的筷子，导致死锁。

|  |
| --- |
| 1 #define N 5 2   1. void philosopher(int i) { 2. while(TRUE) { 3. think(); 4. take(i); // 拿起左边的筷子 5. take((i+1)%N); // 拿起右边的筷子 6. eat(); 7. put(i);   10 put((i+1)%N);  11 }  12 } |

为了防止死锁的发生，可以设置两个条件： 必须同时拿起左右两根筷子；

只有在两个邻居都没有进餐的情况下才允许进餐。

#define N 5

#define LEFT (i + N - 1) % N // 左邻居

#define RIGHT (i + 1) % N    // 右邻居

#define THINKING 0

#define HUNGRY   1

#define EATING   2

typedef int semaphore;

int state[N];                // 跟踪每个哲学家的状态

semaphore mutex = 1;         // 临界区的互斥，临界区是 state 数组，对其修改需要互斥

semaphore s[N];              // 每个哲学家一个信号量

void philosopher(int i) {

while (*TRUE*) {

think(i);

take\_two(i);

eat(i);

put\_two(i);

}

}

void take\_two(int i) {

down(&mutex);

state[i] = HUNGRY;

check(i);

up(&mutex);

down(&s[i]); // 只有收到通知之后才可以开始吃，否则会一直等下去

}

void put\_two(i) {

down(&mutex);

state[i] = THINKING;

check(LEFT); // 尝试通知左右邻居，自己吃完了，你们可以开始吃了

check(RIGHT);

up(&mutex);

}

void eat(int i) {

down(&mutex);

state[i] = EATING;

up(&mutex);

}

// 检查两个邻居是否都没有用餐，如果是的话，就 up(&s[i])，使得 down(&s[i]) 能够得到通知

并继续执行

void check(i) {

if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT]

!= EATING) {

state[i] = EATING;

up(&s[i]);

}

}

### 24、操作系统经典问题之读者-写者问题

允许多个进程同时对数据进行读操作，但是不允许读和写以及写和写操作同时发生。

一个整型变量 count 记录在对数据进行读操作的进程数量，一个互斥量 count\_mutex 用于对 count 加锁，一个互斥量 data\_mutex 用于对读写的数据加锁。

|  |
| --- |
| 1. typedef int semaphore; 2. semaphore count\_mutex = 1; 3. semaphore data\_mutex = 1; 4. int count = 0; 5 5. void reader() { 6. while(TRUE) { 7. down(&count\_mutex); 8. count++; |

|  |
| --- |
| 1. if(count == 1) down(&data\_mutex); // 第一个读者需要对数据进行加锁，防止写进程访问 2. up(&count\_mutex); 3. read(); 4. down(&count\_mutex); 5. count--; 6. if(count == 0) up(&data\_mutex);//最后一个读者要对数据进行解锁，防止写进程无法访问 7. up(&count\_mutex); 17 }   18 }  19   1. void writer() { 2. while(TRUE) { 3. down(&data\_mutex); 4. write(); 5. up(&data\_mutex); 25 }   26 }  27  28 |

### 25、介绍一下几种典型的锁

##### 读写锁

多个读者可以同时进行读

写者必须互斥（只允许一个写者写，也不能读者写者同时进行）

写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

##### 互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责 线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以 互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线 程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

##### 条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个 线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足 时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变 量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说**互斥锁是线程间互 斥的机制，条件变量则是同步机制。**

##### 自旋锁

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。 如果别的线程长时期占有锁，那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短 的场景，这个时候效率比较高。

[《互斥锁、读写锁、自旋锁、条件变量的特点总结》：https://blog.csdn.net/RUN32875094/arti cle/details/80169978](https://blog.csdn.net/RUN32875094/article/details/80169978)

* 1. 线程（POSIX）锁有哪些？ 互斥锁（mutex）

互斥锁属于sleep-waiting类型的锁。例如在一个双核的机器上有两个线程A和B，它们 分别运行在core 0和core 1上。假设线程A想要通过pthread\_mutex\_lock操作去得到一个临界区的锁，而此时这个锁正被线程B所持有，那么线程A就会被阻塞，此时会通过上 下文切换将线程A置于等待队列中，此时core 0就可以运行其他的任务（如线程C）。

条件变量(cond)

自旋锁(spin)

自旋锁属于busy-waiting类型的锁，如果线程A是使用pthread\_spin\_lock操作去请求 锁，如果自旋锁已经被线程B所持有，那么线程A就会一直在core 0上进行忙等待并不停的进行锁请求，检查该自旋锁是否已经被线程B释放，直到得到这个锁为止。因为自旋 锁不会引起调用者睡眠，所以自旋锁的效率远高于互斥锁。

虽然它的效率比互斥锁高，但是它也有些不足之处：

自旋锁一直占用CPU，在未获得锁的情况下，一直进行自旋，所以占用着CPU，如 果不能在很短的时间内获得锁，无疑会使CPU效率降低。

在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁。

自旋锁只有在内核可抢占式或SMP的情况下才真正需要，在单CPU且不可抢占式的内核 下，自旋锁的操作为空操作。自旋锁适用于锁使用者保持锁时间比较短的情况下。

### 26、逻辑地址VS物理地址

Eg:编译时只需确定变量x存放的相对地址是100 ( 也就是说相对于进程在内存中的起始地址而言的地址)。CPU想要找到x在内存中的实际存放位置，只需要用进程的起始地址+100即可。

相对地址又称逻辑地址，绝对地址又称物理地址。

### 27、怎么回收线程？有哪几种方法？

等待线程结束：int pthread\_join(pthread\_t tid, void\*\* retval);

主线程调用，等待子线程退出并回收其资源，类似于进程中wait/waitpid回收僵尸进程，调用

pthread\_join的线程会被阻塞。

tid：创建线程时通过指针得到tid值。retval：指向返回值的指针。

结束线程：pthread\_exit(void \*retval);

子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过pthread\_join获得。

retval：同上。

分离线程：int pthread\_detach(pthread\_t tid);

主线程、子线程均可调用。主线程中pthread\_detach(tid)，子线程中

pthread\_detach(pthread\_self())，调用后和主线程分离，子线程结束时自己立即回收资源。

tid：同上。

### 28、内存的覆盖是什么？有什么特点？

由于程序运行时并非任何时候都要访问程序及数据的各个部分（尤其是大程序），因此可以把用户空间 分成为一个固定区和若干个覆盖区。将经常活跃的部分放在固定区，其余部分按照调用关系分段，首先 将那些即将要访问的段放入覆盖区，其他段放在外存中，在需要调用前，系统将其调入覆盖区，替换覆 盖区中原有的段。

覆盖技术的特点：是打破了必须将一个进程的全部信息装入内存后才能运行的限制，但当同时运行程序 的代码量大于主存时仍不能运行，再而，大家要注意到，内存中能够更新的地方只有覆盖区的段，不在 覆盖区的段会常驻内存。

### 29、内存交换是什么？有什么特点？

**交换(对换)技术的设计思想**：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已 具备运行条件的进程换入内存(进程在内存与磁盘间动态调度)

换入：把准备好竞争CPU运行的程序从辅存移到内存。

换出：把处于等待状态（或CPU调度原则下被剥夺运行权力）的程序从内存移到辅存，把内存空间腾出 来。

### 30、什么时候会进行内存的交换？

内存交换通常在许多进程运行且内存吃紧时进行，而系统负荷降低就暂停。例如:在发现许多进程运行时 经常发生缺页，就说明内存紧张，此时可以换出一些进程;如果缺页率明显下降，就可以暂停换出。

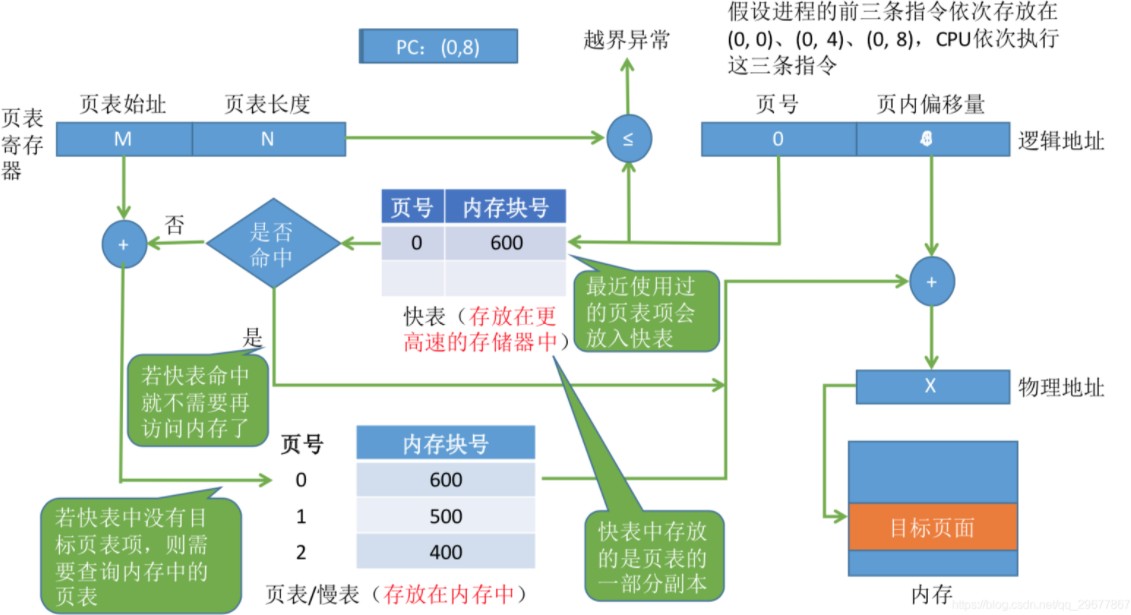
### 31、终端退出，终端运行的进程会怎样

终端在退出时会发送SIGHUP给对应的bash进程，bash进程收到这个信号后首先将它发给session下面 的进程，如果程序没有对SIGHUP信号做特殊处理，那么进程就会随着终端关闭而退出

[《linux终端关闭时为什么会导致在其上启动的进程退出？》：https://blog.csdn.net/QFire/articl e/details/80112701](https://blog.csdn.net/QFire/article/details/80112701)

### 32、什么是快表，你知道多少关于快表的知识？

快表，又称联想寄存器(TLB) ，是一种访问速度比内存快很多的高速缓冲存储器，用来存放当前访问的若干页表项，以加速地址变换的过程。与此对应，内存中的页表常称为慢表。



### 33、地址变换中，有快表和没快表，有什么区别？

|  |  |  |
| --- | --- | --- |
|  | **地址变换过程** | **访问一个逻辑地址的访存次数** |
| 基本地址变换机构 | ①算页号、页内偏移量 ②检查页号合法性 ③查页表，找到页面存放的内存块号 ④根据内存块号与页内偏移量得到物理地址 ⑤访问目标内存单元 | 两次访存 |
| 具有快表的地址变换机构 | ①算页号、页内偏移量 ②检查页号合法性 ③查快表。若命中，即可知道页面存放的内存块号，可直接进行⑤;若未命中则进行④ ④查页表，找到页面存放的内存块号，并且将页表项复制到快表中 ⑤根据内存块号与页内偏移量得到物理地址 ⑥访问目标内存单元 | 快表命中， 只需一次访存 快表未命中，需要两次访存 |

### 34、 守护进程、僵尸进程和孤儿进程

##### 守护进程

指在后台运行的，没有控制终端与之相连的进程。它独立于控制终端，周期性地执行某种任务。Linux 的大多数服务器就是用守护进程的方式实现的，如web服务器进程http等

创建守护进程要点：

1. 让程序在后台执行。方法是调用fork（）产生一个子进程，然后使父进程退出。
2. 调用setsid（）创建一个新对话期。控制终端、登录会话和进程组通常是从父进程继承下来的，守 护进程要摆脱它们，不受它们的影响，方法是调用setsid（）使进程成为一个会话组长。setsid（）调 用成功后，进程成为新的会话组长和进程组长，并与原来的登录会话、进程组和控制终端脱离。
3. 禁止进程重新打开控制终端。经过以上步骤，进程已经成为一个无终端的会话组长，但是它可以 重新申请打开一个终端。为了避免这种情况发生，可以通过使进程不再是会话组长来实现。再一次通过fork（）创建新的子进程，使调用fork的进程退出。
4. 关闭不再需要的文件描述符。子进程从父进程继承打开的文件描述符。如不关闭，将会浪费系统 资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。首先获得最高文件描述符值，然后 用一个循环程序，关闭0到最高文件描述符值的所有文件描述符。
5. 将当前目录更改为根目录。
6. 子进程从父进程继承的文件创建屏蔽字可能会拒绝某些许可权。为防止这一点，使用

unmask（0）将屏蔽字清零。

1. 处理SIGCHLD信号。对于服务器进程，在请求到来时往往生成子进程处理请求。如果子进程等待 父进程捕获状态，则子进程将成为僵尸进程（zombie），从而占用系统资源。如果父进程等待子进程 结束，将增加父进程的负担，影响服务器进程的并发性能。在Linux下可以简单地将SIGCHLD信号的操 作设为SIG\_IGN。这样，子进程结束时不会产生僵尸进程。

##### 孤儿进程

如果父进程先退出，子进程还没退出，那么子进程的父进程将变为init进程。（注：任何一个进程都必 须有父进程）。

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被

init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

##### 僵尸进程

如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结 束，否则这个时候子进程就成为僵尸进程。

设置**僵尸进程的目**的是维护子进程的信息，以便父进程在以后某个时候获取。这些信息至少包括进程 ID，进程的终止状态，以及该进程使用的CPU时间，所以当终止子进程的父进程调用wait或waitpid时 就可以得到这些信息。如果一个进程终止，而该进程有子进程处于僵尸状态，那么它的所有僵尸子进程 的父进程ID将被重置为1（init进程）。继承这些子进程的init进程将清理它们（也就是说init进程将wait 它们，从而去除它们的僵尸状态）。

### 35、如何避免僵尸进程？

通过signal(SIGCHLD, SIG\_IGN)通知内核对子进程的结束不关心，由内核回收。如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG\_IGN);表示父进程忽略SIGCHLD信号， 该信号是子进程退出的时候向父进程发送的。

父进程调用wait/waitpid等函数等待子进程结束，如果尚无子进程退出wait会导致父进程阻塞。

waitpid可以通过传递WNOHANG使父进程不阻塞立即返回。

如果父进程很忙可以用signal注册信号处理函数，在信号处理函数调用wait/waitpid等待子进程退 出。

通过两次调用fork。父进程首先调用fork创建一个子进程然后waitpid等待子进程退出，子进程再fork一个孙进程后退出。这样子进程退出后会被父进程等待回收，而对于孙子进程其父进程已经退 出所以孙进程成为一个孤儿进程，孤儿进程由init进程接管，孙进程结束后，init会等待回收。

第一种方法忽略SIGCHLD信号，这常用于并发服务器的性能的一个技巧因为并发服务器常常fork很多子 进程，子进程终结之后需要服务器进程去wait清理资源。如果将此信号的处理方式设为忽略，可让内核 把僵尸子进程转交给init进程去处理，省去了大量僵尸进程占用系统资源。

[《Linux系统下创建守护进程(Daemon)》：https://blog.csdn.net/linkedin\_35878439/article/de tails/81288889](https://blog.csdn.net/linkedin_35878439/article/details/81288889)

《01\_fork()的使用》：<https://blog.csdn.net/WUZHU2017/article/details/81636851>

### 36、父进程、子进程、进程组、作业和会话

##### 父进程

已创建一个或多个子进程的进程

##### 子进程

由fork创建的新进程被称为子进程（child process）。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是新进程（子进程）的进程 id。将子进程id返回给父进程的理由是：因为一个进程的子进程可以多于一个，没有一个函数使一个进程可以获得其所有子进程的 进程id。对子进程来说，之所以fork返回0给它，是因为它随时可以调用getpid()来获取自己的pid；也 可以调用getppid()来获取父进程的id。(进程id 0总是由交换进程使用，所以一个子进程的进程id不可能为0 )。

fork之后，操作系统会复制一个与父进程完全相同的子进程，虽说是父子关系，但是在操作系统看来， 他们更像兄弟关系，这2个进程共享代码空间，但是数据空间是互相独立的，子进程数据空间中的内容 是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行到的位置（两进程的程序计数 器pc值相同，也就是说，子进程是从fork返回处开始执行的），但有一点不同，如果fork成功，子进程

中fork的返回值是0，父进程中fork的返回值是子进程的进程号，如果fork不成功，父进程会返回错误。

子进程从父进程继承的有：1.进程的资格(真实(real)/有效(eﬀective)/已保存(saved)用户号(UIDs)和组号

(GIDs))2.环境(environment)3.堆栈4.内存5.进程组号

独有：1.进程号；2.不同的父进程号(译者注：即子进程的父进程号与父进程的父进程号不同， 父进程号可由getppid函数得到)；3.资源使用(resource utilizations)设定为0

##### 进程组

进程组就是多个进程的集合，其中肯定有一个组长，其进程PID等于进程组的PGID。只要在某个进程组 中一个进程存在，该进程组就存在，这与其组长进程是否终止无关。

##### 作业

shell分前后台来控制的不是进程而是作业（job）或者进程组（Process Group）。

一个前台作业可以由多个进程组成，一个后台也可以由多个进程组成，shell可以运行一个前台作业和任 意多个后台作业，这称为作业控制

##### 为什么只能运行一个前台作业？

答：当我们在前台新起了一个作业，shell就被提到了后台，因此shell就没有办法再继续接受我们的指 令并且解析运行了。 但是如果前台进程退出了，shell就会有被提到前台来，就可以继续接受我们的命令并且解析运行。

作业与进程组的区别：如果作业中的某个进程有创建了子进程，则该子进程是不属于该作业的。

一旦作业运行结束，shell就把自己提到前台（子进程还存在，但是子进程不属于作业），如果原来的前 台进程还存在（这个子进程还没有终止），他将自动变为后台进程组

##### 会话

会话（Session）是一个或多个进程组的集合。一个会话可以有一个控制终端。在xshell或者WinSCP中 打开一个窗口就是新建一个会话。

### 37、进程终止的几种方式

1、main函数的自然返回， return

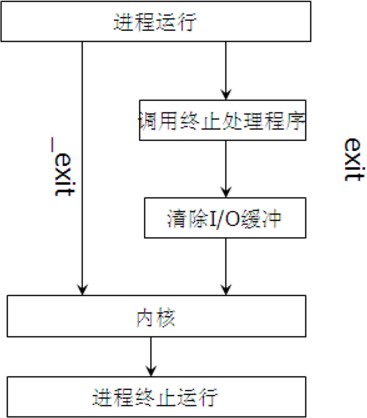
2、调用 exit 函数，属于c的函数库

3、调用 \_exit 函数，属于系统调用

4、调用 abort 函数，异常程序终止，同时发送SIGABRT信号给调用进程。

5、接受能导致进程终止的信号：ctrl+c (^C)、SIGINT(SIGINT中断进程)

##### exit和\_exit的区别

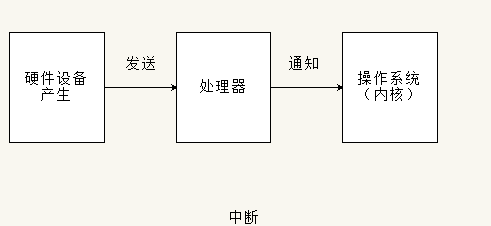


《学习笔记]进程终止的5种方式》：<https://www.cnblogs.com/shichuan/p/4432503.html>

### 38、Linux中异常和中断的区别

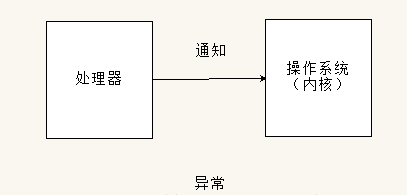
##### 中断

大家都知道，当我们在敲击键盘的同时就会产生中断，当硬盘读写完数据之后也会产生中断，所以，我 们需要知道，中断是由硬件设备产生的，而它们从物理上说就是电信号，之后，它们通过中断控制器发 送给CPU，接着CPU判断收到的中断来自于哪个硬件设备（这定义在内核中），最后，由CPU发送给内 核，有内核处理中断。下面这张图显示了中断处理的流程：



##### 异常

我们在学习《计算机组成原理》的时候会知道两个概念，CPU处理程序的时候一旦程序不在内存中，会 产生缺页异常；当运行除法程序时，当除数为0时，又会产生除0异常。所以，大家也需要记住的是，**异 常是由CPU产生的，同时，它会发送给内核，要求内核处理这些异常**，下面这张图显示了异常处理的流 程：



##### 相同点

最后都是由CPU发送给内核，由内核去处理处理程序的流程设计上是相似的

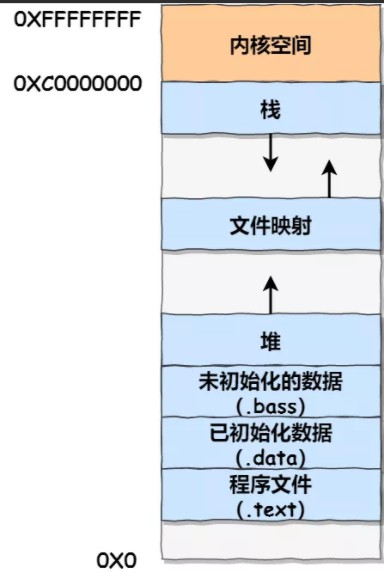
##### 不同点

产生源不相同，异常是由CPU产生的，而中断是由硬件设备产生的内核需要根据是异常还是中断调用不同的处理程序

中断不是时钟同步的，这意味着中断可能随时到来；异常由于是CPU产生的，所以它是时钟同步的 当处理中断时，处于中断上下文中；处理异常时，处于进程上下文中

[《Linux内核--异常和中断的区别》：https://blog.csdn.net/u011068464/article/details/102847 41](https://blog.csdn.net/u011068464/article/details/10284741)

### 39、Windows和Linux环境下内存分布情况



通过这张图你可以看到，用户空间内存，从**低到高**分别是 7 种不同的内存段：

程序文件段，包括二进制可执行代码； 已初始化数据段，包括静态常量；

未初始化数据段，包括未初始化的静态变量；

堆段，包括动态分配的内存，从低地址开始向上增长；

文件映射段，包括动态库、共享内存等，从低地址开始向上增长（跟硬件和内核版本有关）

栈段，包括局部变量和函数调用的上下文等。栈的大小是固定的，一般是 8 MB 。当然系统也提供了参数，以便我们自定义大小；

### 40、一个由C/C++编译的程序占用的内存分为哪几个部分？

1、栈区（stack）— 地址向下增长，由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的队列，先进后出。

2、堆区（heap）— 地址向上增长，一般由程序员分配释放，若程序员不释放，程序结束时可能由OS 回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3、全局区（静态区）（static）—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，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。 - 程序结束后有系统释放

4、文字常量区 —常量字符串就是放在这里的。程序结束后由系统释放

5、程序代码区(text)—存放函数体的二进制代码。

### 41、一般情况下在Linux/windows平台下栈空间的大小

Linux环境下有操作系统决定，一般是8KB，8192kbytes，通过ulimit命令查看以及修改Windows环境下由编译器决定，VC++6.0一般是1M

##### Linux

linux下非编译器决定栈大小，而是由操作系统环境决定，默认是8192KB（8M）；而在Windows平台 下栈的大小是被记录在可执行文件中的（由编译器来设置)，即：windows下可以由编译器决定栈大 小，而在Linux下是由系统环境变量来控制栈的大小的。

在Linux下通过如下命令可查看和设置栈的大小：

|  |
| --- |
| 1. $ ulimit -a # 显示当前栈的大小 （ulimit为系统命令，非编译器命令） 2. $ ulimit -s 32768 # 设置当前栈的大小为32M 3   4 |

##### Windows

下程序栈空间的大小，VC++ 6.0 默认的栈空间是1M。

VC6.0中修改堆栈大小的方法：

选择 "Project->Setting"

选择 "Link"

选择 "Category"中的 "Output"

在 "Stack allocations"中的"Reserve:"中输栈的大小

《Linux/windows栈大小》：<https://blog.csdn.net/HQ354974212/article/details/76087676>

### 42、程序从堆中动态分配内存时，虚拟内存上怎么操作的

页表：是一个存放在物理内存中的数据结构，它记录了虚拟页与物理页的映射关系

在进行动态内存分配时，例如malloc()函数或者其他高级语言中的new关键字，操作系统会在硬盘中创 建或申请一段虚拟内存空间，并更新到页表（分配一个页表条目（PTE），使该PTE指向硬盘上这个新 创建的虚拟页），通过PTE建立虚拟页和物理页的映射关系。

### 43、交换空间与虚拟内存的关系

##### 交换空间

Linux 中的交换空间（Swap space）在**物理内存**（RAM）被充满时被使用。如果系统需要更多的内存资源，而物理内存已经充满，内存中不活跃的页就会被移到交换空间去。虽然交换空间可以为带有少量 内存的机器提供帮助，但是这种方法不应该被当做是对内存的取代。交换空间位于硬盘驱动器上，它比 进入物理内存要慢。

交换空间可以是一个专用的交换分区（推荐的方法），交换文件，或两者的组合。

交换空间的总大小应该相当于你的计算机内存的两倍和 32 MB这两个值中较大的一个，但是它不能超过2048MB（2 GB）。

##### 虚拟内存

虚拟内存是文件数据交叉链接的活动文件。是WINDOWS目录下的一个"WIN386.SWP"文件，这个文件 会不断地扩大和自动缩小。

就速度方面而言,CPU的L1和L2缓存速度最快，内存次之，硬盘再次之。但是**虚拟内存使用的是硬盘的 空间**，为什么我们要使用速度最慢的硬盘来做 为虚拟内存呢？因为电脑中所有运行的程序都需要经过内存来执行，如果执行的程序很大或很多，就会导致我们只有可怜的256M/512M内存消耗殆尽。而硬盘 空间动辄几十G上百G，为了解决这个问题，Windows中运用了虚拟内存技术，即拿出一部分硬盘空间 来充当内存使用。

《交换空间和虚拟内存的区别》：<https://blog.csdn.net/qsd007/article/details/1567955>

### 44、从堆和栈上建立对象哪个快？（考察堆和栈的分配效率比较）

从两方面来考虑：

分配和释放，堆在分配和释放时都要调用函数（malloc,free)，比如分配时会到堆空间去寻找足够 大小的空间（因为多次分配释放后会造成内存碎片），这些都会花费一定的时间，具体可以看看malloc和free的源代码，函数做了很多额外的工作，而栈却不需要这些。

访问时间，访问堆的一个具体单元，需要两次访问内存，第一次得取得指针，第二次才是真正的数 据，而栈只需访问一次。另外，堆的内容被操作系统交换到外存的概率比栈大，栈一般是不会被交 换出去的。

### 45、常见内存分配方式有哪些？

##### 内存分配方式

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我们决定，使用非常灵活，但问题也 最多。

### 46、常见内存分配内存错误

1. 内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检 查指针是否为NULL。如果指针p是函数的参数，那么在函数的入口处用assert(p!=NULL)进行检查。如 果是用malloc或new来申请内存，应该用if(p==NULL) 或if(p!=NULL)进行防错处理。

1. 内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初 值错误（例如数组）。内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可 信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要 嫌麻烦。

1. 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在for循环语句中，循环次数很容易搞 错，导致数组操作越界。

1. 忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一 次程序突然挂掉，系统出现提示：内存耗尽。动态内存的申请与释放必须配对，程序中malloc与free的 使用次数一定要相同，否则肯定有错误（new/delete同理）。

1. 释放了内存却继续使用它。常见于以下有三种情况：

程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重 新设计数据结构，从根本上解决对象管理的混乱局面。

函数的return语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体 结束时被自动销毁。

使用free或delete释放了内存后，没有将指针设置为NULL。导致产生“野指针”。

[《内存分配方式及常见错误》：https://www.cnblogs.com/skynet/archive/2010/12/03/189504 5.html](https://www.cnblogs.com/skynet/archive/2010/12/03/1895045.html)

应该在外存(磁盘)的什么位置保存被换出的进程?

### 47、ASCII、Unicode和UTF-8编码的区别？

##### ASCII

ASCII 只有127个字符，表示英文字母的大小写、数字和一些符号，但由于其他语言用ASCII 编码表示字节不够，例如：常用中文需要两个字节，且不能和ASCII冲突，中国定制了GB2312编码格式，相同的， 其他国家的语言也有属于自己的编码格式。

##### Unicode

由于每个国家的语言都有属于自己的编码格式，在多语言编辑文本中会出现乱码，这样Unicode应运而 生，Unicode就是将这些语言统一到一套编码格式中，通常两个字节表示一个字符，而ASCII是一个字节 表示一个字符，这样如果你编译的文本是全英文的，用Unicode编码比ASCII编码需要多一倍的存储空 间，在存储和传输上就十分不划算。

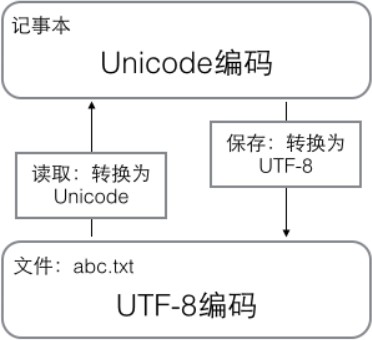
##### UTF-8

为了解决上述问题，又出现了把Unicode编码转化为“**可变长编码**”UTF-8编码，UTF-8编码将Unicode字 符按数字大小编码为1-6个字节，英文字母被编码成一个字节，常用汉字被编码成三个字节，如果你编 译的文本是纯英文的，那么用UTF-8就会非常节省空间，并且ASCII码也是UTF-8的一部分。

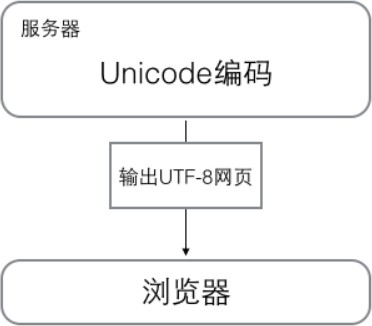
##### 三者之间的联系

搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方 式：

1. 在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8 编码
2. 用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存 的时候再把Unicode转换为UTF-8保存到文件。如下图（截取他人图片）



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



[《字符编码中ASCII、Unicode和UTF-8的区别》：https://www.cnblogs.com/moumoon/p/1098 8234.html](https://www.cnblogs.com/moumoon/p/10988234.html)

### 48、原子操作的是如何实现的

**处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。**首先处理器会自动保证 基本的内存操作的原子性。处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处 理器读取一个字节时，其他处理器不能访问这个字节的内存地址。Pentium 6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器是不能自动 保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。但是，处理器提供总线锁定和缓存 锁定两个机制来保证复杂内存操作的原子性。

1. 使用总线锁保证原子性

第一个机制是通过总线锁保证原子性。如果多个处理器同时对共享变量进行读改写操作（i++就是经典 的读改写操作），那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作 完之后共享变量的值会和期望的不一致。举个例子，如果i=1，我们进行两次i++操作，我们期望的结果 是3，但是有可能结果是2，如图下图所示。

|  |
| --- |
| 1 CPU1 CPU2  2 i=1 i=1  3 i+1 i+1  4 i=2 i=2  5 |

原因可能是多个处理器同时从各自的缓存中读取变量i，分别进行加1操作，然后分别写入系统内存中。 那么，想要保证读改写共享变量的操作是原子的，就必须保证CPU1读改写共享变量的时候，CPU2不能 操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。**所谓总线锁就是使用处理器提供的一个LOCK＃信号，当一 个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。**

1. 使用缓存锁保证原子性

第二个机制是通过缓存锁定来保证原子性。在同一时刻，我们只需保证对某个内存地址的操作是原子性 即可，但**总线锁定把CPU和内存之间的通信锁住了**，这使得锁定期间，其他处理器不能操作其他内存地 址的数据，所以总线锁定的开销比较大，目前处理器在某些场合下使用缓存锁定代替总线锁定来进行优

化。

频繁使用的内存会缓存在处理器的L1、L2和L3高速缓存里，那么原子操作就可以直接在处理器内部缓存 中进行，并不需要声明总线锁，在Pentium 6和目前的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。

所谓“缓存锁定”是指内存区域如果被缓存在处理器的缓存行中，并且在Lock操作期间被锁定，那么当它 执行锁操作回写到内存时，处理器不在总线上声言LOCK＃信号，而是修改内部的内存地址，并允许它 的缓存一致性机制来保证操作的原子性，因为**缓存一致性机制会阻止同时修改由两个以上处理器缓存的 内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效，在如上图所示的例子 中，当CPU1修改缓存行中的i时使用了缓存锁定，那么CPU2就不能使用同时缓存i的缓存行。**

但是有两种情况下处理器不会使用缓存锁定。

第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line） 时，则处理器会调用总线锁定。

第二种情况是：有些处理器不支持缓存锁定。对于Intel 486和Pentium处理器，就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

《原子操作的实现原理》：<https://blog.csdn.net/zxx901221/article/details/83033998>

### 49、内存交换你知道有哪些需要注意的关键点吗？

* 1. 交换需要备份存储，通常是快速磁盘，它必须足够大，并且提供对这些内存映像的直接访问。
  2. 为了有效使用CPU，需要每个进程的执行时间比交换时间长，而影响交换时间的主要是转移时间， 转移时间与所交换的空间内存成正比。
  3. 如果换出进程，比如确保该进程的内存空间成正比。
  4. 交换空间通常作为磁盘的一整块，且独立于文件系统，因此使用就可能很快。
  5. 交换通常在有许多进程运行且内存空间吃紧时开始启动，而系统负荷降低就暂停。
  6. 普通交换使用不多，但交换的策略的某些变种在许多系统中（如UNIX系统）仍然发挥作用。

### 50、系统并发和并行，分得清吗？

并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。 并行需要硬件支持，如多流水线、多核处理器或者分布式计算系统。

操作系统通过引入进程和线程，使得程序能够并发运行。

### 51、页面置换算法有哪些？

##### 1、最佳置换法(OPT)

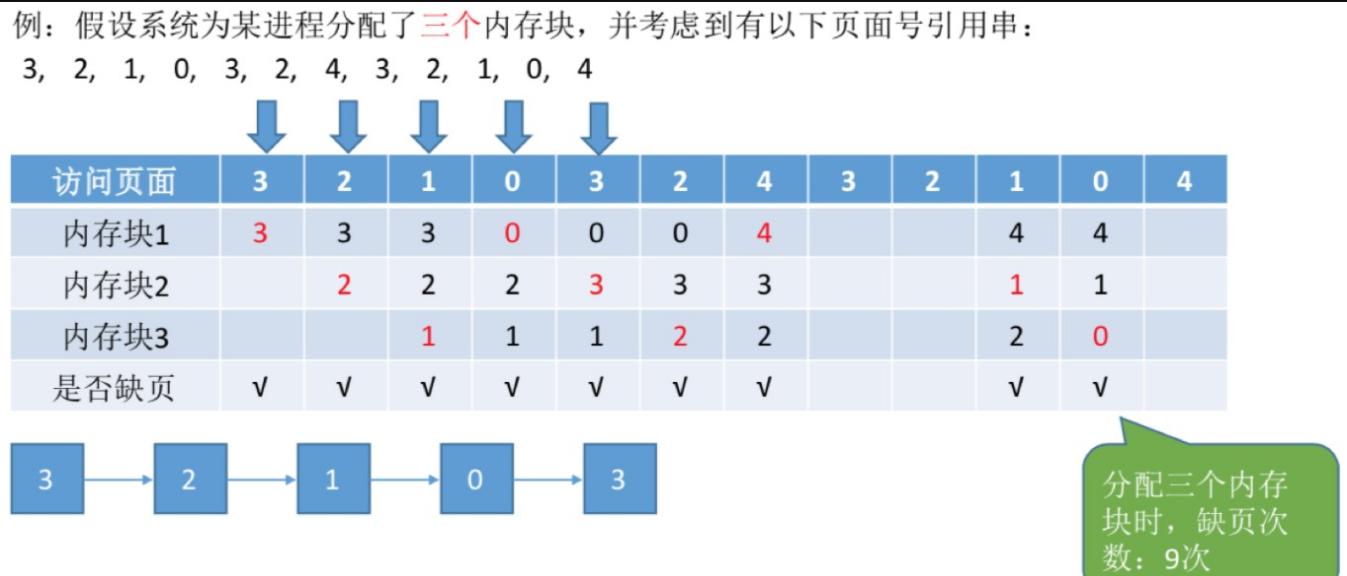
最佳置换算法(OPT，Optimal) :每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。

最佳置换算法可以保证最低的缺页率，但实际上，只有在进程执行的过程中才能知道接下来会访问到的 是哪个页面。操作系统无法提前预判页面访问序列。因此，最佳置换算法是无法实现的

##### 2、先进先出置换算法(FIFO)

先进先出置换算法(FIFO) :每次选择淘汰的页面是最早进入内存的页面

实现方法:把调入内存的页面根据调入的先后顺序排成一个队列，需要换出页面时选择队头页面队列的最 大长度取决于系统为进程分配了多少个内存块。





Belady异常—当为进程分配的物理块数增大时，缺页次数不减反增的异常现象。

只有FIFO算法会产生Belady异常，而LRU和OPT算法永远不会出现Belady异常。另外，FIFO算法虽然实 现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因 此，算法性能差

FIFO的性能较差，因为较早调入的页往往是经常被访问的页，这些页在FIFO算法下被反复调入和调出， 并且有Belady现象。所谓Belady现象是指：采用FIFO算法时，如果对—个进程未分配它所要求的全部 页面，有时就会出现分配的页面数增多但缺页率反而提高的异常现象。

##### 3、最近最久未使用置换算法(LRU)

最近最久未使用置换算法(LRU，least recently used) :每次淘汰的页面是最近最久未使用的页面

实现方法:赋予每个页面对应的页表项中，用访问字段记录该页面自.上次被访问以来所经历的时间t(该算 法的实现需要专门的硬件支持，虽然算法性能好，但是实现困难，开销大)。当需要淘汰一个页面时，选 择现有页面中t值最大的，即最近最久未使用的页面。

LRU性能较好，但需要寄存器和栈的硬件支持。LRU是堆栈类算法，理论上可以证明，堆栈类算法不可 能出现Belady异常。

在手动做题时，若需要淘汰页面，可以逆向检查此时在内存中的几个页面号。在逆向扫描过程中最后一 个出现的页号就是要淘汰的页面。

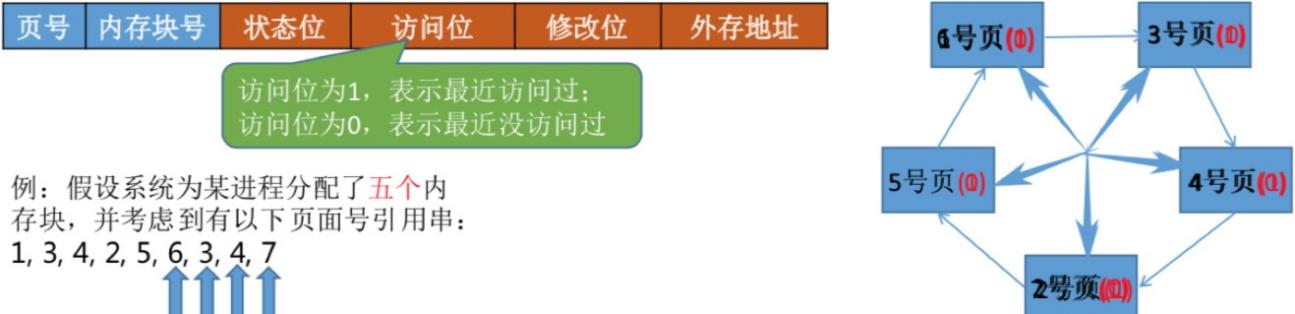
##### 4、时钟置换算法(CLOCK)

最佳置换算法性OPT能最好，但无法实现；先进先出置换算法实现简单，但算法性能差；最近最久未使 用置换算法性能好，是最接近OPT算法性能的，但是实现起来需要专门的硬件支持，算法开销大。

所以操作系统的设计者尝试了很多算法，试图用比较小的开销接近LRU的性能，这类算法都是CLOCK算 法的变体，因为算法要循环扫描缓冲区像时钟一样转动。所以叫clock算法。

时钟置换算法是一种性能和开销较均衡的算法，又称CLOCK算法，或最近未用算法(NRU，Not Recently Used)

简单的CLOCK算法实现方法:为每个页面设置一个访问位，再将内存中的页面都通过链接指针链接成一 个循环队列。当某页被访问时，其访问位置为1。当需要淘汰-一个页面时，只需检查页的访问位。如果 是0，就选择该页换出;如果是1，则将它置为0，暂不换出，继续检查下一个页面，若第- - ~轮扫描中所有页面都是1，则将这些页面的访问位依次置为0后，再进行第二轮扫描(第二轮扫描中一定会有访问位 为0的页面，因此简单的CLOCK算法选择–个淘汰页面最多会经过两轮扫描)



##### 5、改进型的时钟置换算法

简单的时钟置换算法仅考虑到一个页面最近是否被访问过。事实上，如果被淘汰的页面没有被修改过,就 不需要执行I/O操作写回外存。只有被淘汰的页面被修改过时，才需要写回外存。

因此，除了考虑一个页面最近有没有被访问过之外，操作系统还应考虑页面有没有被修改过。在其他条 件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。这就是改进型的时钟置换算法的思想。修 改位=0，表示页面没有被修改过;修改位=1，表示页面被修改过。

为方便讨论，用(访问位，修改位)的形式表示各页面状态。如(1, 1)表示一个页面近期被访问过，且被修改过。

改进型的Clock算法需要综合考虑某一内存页面的访问位和修改位来判断是否置换该页面。在实际编写 算法过程中，同样可以用一个等长的整型数组来标识每个内存块的修改状态。访问位A和修改位M可以 组成一下四种类型的页面。

算法规则:将所有可能被置换的页面排成–个循环队列

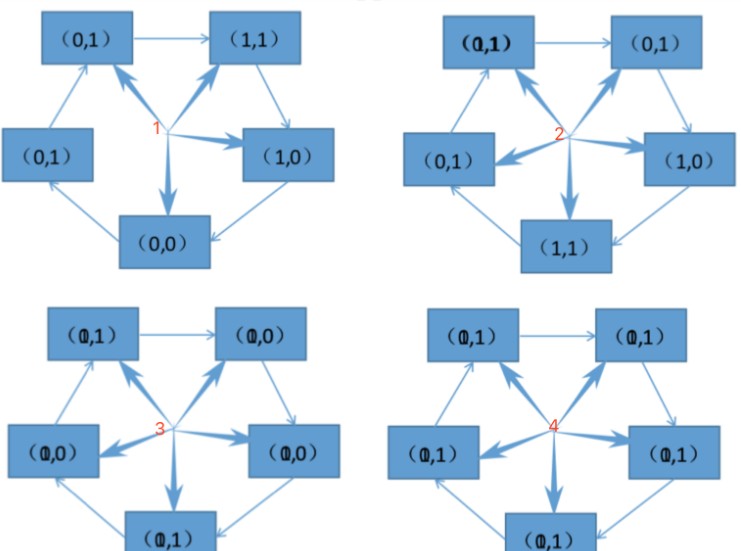
第一轮:从当前位置开始扫描到第一个(A =0, M = 0)的帧用于替换。表示该页面最近既未被访问， 又未被修改，是最佳淘汰页

第二轮:若第一轮扫描失败，则重新扫描，查找第一个(A =1, M = 0)的帧用于替换。本轮将所有扫描过的帧访问位设为0。表示该页面最近未被访问，但已被修改，并不是很好的淘汰页。

第三轮:若第二轮扫描失败，则重新扫描，查找第一个(A =0, M = 1)的帧用于替换。本轮扫描不修改任何标志位。表示该页面最近已被访问，但未被修改，该页有可能再被访问。

第四轮:若第三轮扫描失败，则重新扫描，查找第一个A =1, M = 1)的帧用于替换。表示该页最近已被访问且被修改，该页可能再被访问。

由于第二轮已将所有帧的访问位设为0，因此经过第三轮、第四轮扫描一定会有一个帧被选中，因此改 进型CLOCK置换算法选择- -个淘汰页面最多会进行四轮扫描



算法规则：将所有可能被置换的页面排成一个循环队列

第一轮:从当前位置开始扫描到第-一个(0, 0)的帧用于替换。本轮扫描不修改任何标志位。(第一优先级:

最近没访问，且没修改的页面)

第二轮:若第一轮扫描失败，则重新扫描，查找第一个(0, 1)的帧用于替换。本轮将所有扫描过的帧访问

位设为0

(第二优先级: 最近没访问，但修改过的页面)

第三轮:若第二轮扫描失败，则重新扫描，查找第一个(0, 0)的帧用于替换。本轮扫描不修改任何标志位(第三优先级:最近访问过，但没修改的页面)

第四轮:若第三轮扫描失败，则重新扫描，查找第一个(0, 1)的帧用于替换。(第四优先级:最近访问过，且修改过的页面)

由于第二轮已将所有帧的访问位设为0，因此经过第三轮、第四轮扫描一定会有一个帧被选中，因此改 进型CLOCK置换算法选择一个淘汰页面最多会进行四轮扫描

##### 6、总结

|  |  |  |
| --- | --- | --- |
|  | **算法规则** | **优缺点** |
| OPT | 优先淘汰最长时间内不会被访问的页面 | 缺页率最小，性能最好;但无法实现 |
| FIFO | 优先淘汰最先进入内存的页面 | 实现简单;但性能很差，可能出现Belady异常 |
| LRU | 优先淘汰最近最久没访问的页面 | 性能很好;但需要硬件支持，算法开销大 |
| CLOCK (NRU) | 循环扫描各页面 第一轮淘汰访问位=0的，并将扫描过的页面访问位改为1。若第-轮没选中，则进行第二轮扫描。 | 实现简单，算法开销小;但未考虑页面是否被修改过。 |
| 改进型  CLOCK  (改进型NRU) | 若用(访问位，修改位)的形式表述，则 第一轮:淘汰(0,0) 第二轮:淘汰(O,1)，并将扫描过的页面访问位都置为0 第三轮: 淘汰(O, 0) 第四轮:淘汰(0, 1) | 算法开销较小，性能也不错 |

**52、共享是什么？**

共享是指系统中的资源可以被多个并发进程共同使用。有两种共享方式：互斥共享和同时共享。

互斥共享的资源称为临界资源，例如打印机等，在同一时刻只允许一个进程访问，需要用同步机制来实 现互斥访问。

### 52、死锁相关问题大总结，超全！

##### 死锁是指两个（多个）线程相互等待对方数据的过程，死锁的产生会导致程序卡死，不解锁程序将永远 无法进行下去。

**1、死锁产生原因**

举个例子：两个线程A和B，两个数据1和2。线程A在执行过程中，首先对资源1加锁，然后再去给资源2 加锁，但是由于线程的切换，导致线程A没能给资源2加锁。线程切换到B后，线程B先对资源2加锁，然 后再去给资源1加锁，由于资源1已经被线程A加锁，因此线程B无法加锁成功，当线程切换为A时，A也 无法成功对资源2加锁，由此就造成了线程AB双方相互对一个已加锁资源的等待，死锁产生。

理论上认为死锁产生有以下四个必要条件，缺一不可：

1. **互斥条件**：进程对所需求的资源具有排他性，若有其他进程请求该资源，请求进程只能等待。
2. **不剥夺条件**：进程在所获得的资源未释放前，不能被其他进程强行夺走，只能自己释放。
3. **请求和保持条件**：进程当前所拥有的资源在进程请求其他新资源时，由该进程继续占有。
4. **循环等待条件**：存在一种进程资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程 所请求。

##### 2、死锁演示

通过代码的形式进行演示，需要两个线程和两个互斥量。

#include <iostream>

#include <vector>

#include <list>

#include <thread>

#include <mutex>  //引入互斥量头文件

using namespace std;

class A {

public:

//插入消息，模拟消息不断产生

void insertMsg() {

for (int i = 0; i < 100; i++) {

cout << "插入一条消息:" << i << *endl*;

my\_mutex1.*lock*(); //语句1

my\_mutex2.*lock*(); //语句2

Msg.*push\_back*(i);

my\_mutex2.*unlock*();

my\_mutex1.*unlock*();

}

}

//读取消息

void readMsg() {

int MsgCom;

for (int i = 0; i < 100; i++) {

MsgCom = MsgLULProc(i);

if (MsgLULProc(MsgCom)) {

//读出消息了

cout << "消息已读出" << MsgCom << *endl*;

}

else {

//消息暂时为空

cout << "消息为空" << *endl*;

}

}

}

//加解锁代码

bool MsgLULProc(int& command) {

int curMsg;

my\_mutex2.*lock*();   //语句3

my\_mutex1.*lock*();   //语句4

if (!Msg.*empty*()) {

//读取消息，读完删除

command = Msg.*front*();

Msg.*pop\_front*();

my\_mutex1.*unlock*();

my\_mutex2.*unlock*();

return true;

}

my\_mutex1.*unlock*();

my\_mutex2.*unlock*();

return false;

}

private:

std::*list*<int> Msg;  //消息变量

std::*mutex* my\_mutex1; //互斥量对象1

std::*mutex* my\_mutex2; //互斥量对象2

};

int main() {

A a;

//创建一个插入消息线程

std::*thread* insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象

//创建一个读取消息线程

std::*thread* readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象

insertTd.*join*();

readTd.*join*();

return 0;

}

语句1和语句2表示线程A先锁资源1，再锁资源2，语句3和语句4表示线程B先锁资源2再锁资源1，具备 死锁产生的条件。

##### 3、死锁的解决方案

**保证上锁的顺序一致。**

**4、死锁必要条件**

互斥条件：进程对所需求的资源具有排他性，若有其他进程请求该资源，请求进程只能等待。 不剥夺条件：进程在所获得的资源未释放前，不能被其他进程强行夺走，只能自己释放

请求和保持条件：进程当前所拥有的资源在进程请求其他新资源时，由该进程继续占有。

循环等待条件：存在一种进程资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程 所请求。

##### 5、处理方法

主要有以下四种方法： 鸵鸟策略

死锁检测与死锁恢复死锁预防

死锁避免

##### 鸵鸟策略

把头埋在沙子里，假装根本没发生问题。

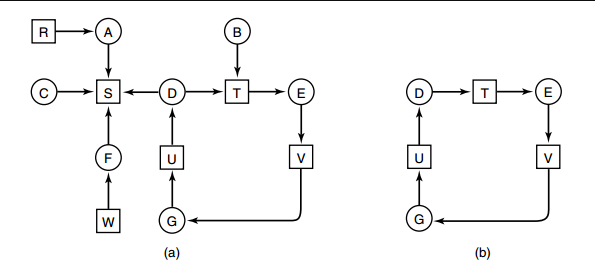
因为解决死锁问题的代价很高，因此鸵鸟策略这种不采取任务措施的方案会获得更高的性能。 当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。

大多数操作系统，包括 Unix，Linux 和 Windows，处理死锁问题的办法仅仅是忽略它。

##### 死锁检测与死锁恢复

不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复。

1、每种类型一个资源的死锁检测

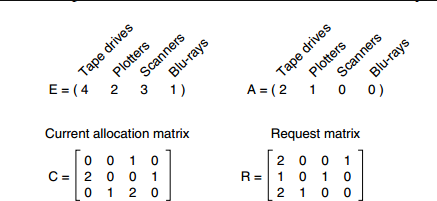


上图为资源分配图，其中方框表示资源，圆圈表示进程。资源指向进程表示该资源已经分配给该进程， 进程指向资源表示进程请求获取该资源。

图 a 可以抽取出环，如图 b，它满足了环路等待条件，因此会发生死锁。

每种类型一个资源的死锁检测算法是通过检测有向图是否存在环来实现，从一个节点出发进行深度优先 搜索，对访问过的节点进行标记，如果访问了已经标记的节点，就表示有向图存在环，也就是检测到死 锁的发生。

2、每种类型多个资源的死锁检测



上图中，有三个进程四个资源，每个数据代表的含义如下：

E 向量：资源总量

A 向量：资源剩余量

C 矩阵：每个进程所拥有的资源数量，每一行都代表一个进程拥有资源的数量

R 矩阵：每个进程请求的资源数量

进程 P1 和 P2 所请求的资源都得不到满足，只有进程 P3 可以，让 P3 执行，之后释放 P3 拥有的资源， 此时 A = (2 2 2 0)。P2 可以执行，执行后释放 P2 拥有的资源，A = (4 2 2 1) 。P1 也可以执行。所有进程都可以顺利执行，没有死锁。

算法总结如下：

每个进程最开始时都不被标记，执行过程有可能被标记。当算法结束时，任何没有被标记的进程都是死 锁进程。

1. 寻找一个没有标记的进程 Pi，它所请求的资源小于等于 A。
2. 如果找到了这样一个进程，那么将 C 矩阵的第 i 行向量加到 A 中，标记该进程，并转回 1。
3. 如果没有这样一个进程，算法终止。

##### 6、死锁恢复

利用抢占恢复利用回滚恢复

通过杀死进程恢复

##### 7、死锁预防

在程序运行之前预防发生死锁。

1. 破坏互斥条件

例如假脱机打印机技术允许若干个进程同时输出，唯一真正请求物理打印机的进程是打印机守护进程。

1. 破坏请求和保持条件

一种实现方式是规定所有进程在开始执行前请求所需要的全部资源。

1. 破坏不剥夺条件允许抢占资源
2. 破坏循环请求等待

给资源统一编号，进程只能按编号顺序来请求资源。

##### 8、死锁避免

在程序运行时避免发生死锁。

##### 安全状态

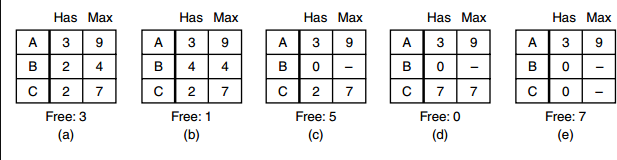
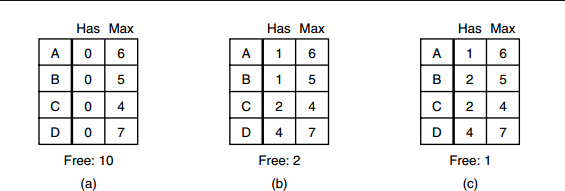


图 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，因此可以称图 a 所示的状态时安全的。

定义：如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能 够使得每一个进程运行完毕，则称该状态是安全的。

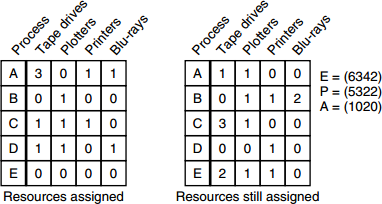
安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检 测算法非常类似，可以结合着做参考对比。

##### 单个资源的银行家算法

一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，算法要做的是判断对请求的满足是否 会进入不安全状态，如果是，就拒绝请求；否则予以分配。

上图 c 为不安全状态，因此算法会拒绝之前的请求，从而避免进入图 c 中的状态。

##### 多个资源的银行家算法



上图中有五个进程，四个资源。左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右 边的 E、P 以及 A 分别表示：总资源、已分配资源以及可用资源，注意这三个为向量，而不是具体数值，例如 A=(1020)，表示 4 个资源分别还剩下 1/0/2/0。

##### 4、检查一个状态是否安全的算法如下：

查找右边的矩阵是否存在一行小于等于向量 A。如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。

假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。重复以上两步，直到所有进程都标记为终止，则状态时安全的。

如果一个状态不是安全的，需要拒绝进入这个状态。

### 53、为什么分段式存储管理有外部碎片而无内部碎片？为什么固定分区分配有内部碎片而不会有外部碎片？

分段式分配是按需分配，而固定式分配是固定分配的方式

### 54、内部碎片与外部碎片

内碎片：分配给某些进程的内存区域中有些部分没用上，常见于固定分配方式

内存总量相同，100M

固定分配，将100M分割成10块，每块10M，一个程序需要45M，那么需要分配5块，第五块只用了

5M，剩下的5M就是内部碎片；

分段式分配，按需分配，一个程序需要45M，就给分片45MB，剩下的55M供其它程序使用，不存在内 部碎片。

外碎片：内存中某些空闲区因为比较小，而难以利用上，一般出现在内存动态分配方式中

分段式分配：内存总量相同，100M，比如，内存分配依次5M，15M，50M，25M，程序运行一段时间 之后，5M，15M的程序运行完毕，释放内存，其他程序还在运行，再次分配一个10M的内存供其它程 序使用，只能从头开始分片，这样，就会存在10M+5M的外部碎片

### 55、如何消除碎片文件

对于外部碎片，通过**紧凑技术**消除，就是操作系统不时地对进程进行移动和整理。但是这需要动态重定 位寄存器地支持，且相对费时。紧凑地过程实际上类似于Windows系统中地磁盘整理程序，只不过后者 是对外存空间地紧凑

解决外部内存碎片的问题就是**内存交换**。

可以把音乐程序占用的那 256MB 内存写到硬盘上，然后再从硬盘上读回来到内存里。不过再读回的时候，我们不能装载回原来的位置，而是紧紧跟着那已经被占用了的 512MB 内存后面。这样就能空缺出连续的 256MB 空间，于是新的 200MB 程序就可以装载进来。

回收内存时要尽可能地将相邻的空闲空间合并。

### 56、进程与线程的区别，共享的数据

进程（Process）和线程（Thread）是操作系统中用于实现并发执行的两种基本单位，它们之间有以下区别：

1. 定义：

- 进程是程序的一次执行过程，是系统资源分配和调度的基本单位，具有独立的地址空间、独立的内存空间和独立的执行序列。

- 线程是进程中的一个执行单元，是操作系统能够进行运算调度的最小单位，线程间共享进程的资源。

2. 资源占用：

- 进程拥有独立的地址空间和系统资源，包括内存、文件描述符、安全权限等，因此创建和销毁进程的开销比较大。

- 线程是进程内的轻量级执行单元，线程间共享进程的资源，如内存空间、文件描述符等，因此创建和销毁线程的开销较小。

3. 并发性：

- 进程是独立运行的，每个进程有自己的执行序列，因此进程间的通信和数据共享需要特殊的机制，如管道、消息队列等。

- 线程是在同一个进程内并发执行的，线程间可以直接共享进程的资源，因此线程间的通信和数据共享比较容易实现。

4. 切换开销：

- 由于进程拥有独立的地址空间和系统资源，进程切换的开销比较大，涉及到上下文切换和内存状态保存等操作。

- 线程是共享进程资源的执行单元，线程切换的开销比进程切换小很多，通常只需保存和恢复线程的上下文信息。

5. 错误影响范围：

- 进程间的错误一般不会相互影响，因为它们拥有独立的地址空间和系统资源。

- 线程间共享进程资源，一个线程的错误可能会影响到其他线程和整个进程的稳定性。

关于共享的数据，进程间的数据通常通过进程间通信（Inter-Process Communication，IPC）来实现，如管道、消息队列、共享内存等；而线程间共享的数据则直接使用进程的共享内存区域或全局变量来实现。需要注意的是，线程间共享数据时需要考虑同步和互斥的问题，以避免数据竞争和不一致的情况发生。

### 57、进程和线程、上下文切换了什么、共享了什么

进程和线程是操作系统中用于实现并发执行的两种基本单位，它们之间的上下文切换和共享的内容如下：

上下文切换：

- 上下文是指进程或线程在执行过程中所需的状态信息，包括程序计数器、寄存器、栈指针等。

- 当操作系统决定切换到另一个进程或线程时，需要保存当前进程或线程的上下文，并恢复下一个要执行的进程或线程的上下文，这个过程称为上下文切换。

- 上下文切换的开销较高，因为需要保存和恢复大量的状态信息，包括寄存器值、内存映射等。

\*进程的上下文切换：

- 进程的上下文切换涉及到保存和恢复的内容较多。

- 在进程切换时，操作系统会保存当前进程的程序计数器、寄存器、内存映射、打开的文件等状态信息，然后加载下一个要执行的进程的相应状态信息。

线程的上下文切换：

- 线程的上下文切换相对较轻量级，因为线程共享进程的地址空间和系统资源。

- 在线程切换时，只需要保存和恢复线程的程序计数器、寄存器等少量的状态信息即可。

共享的内容：

- 进程间的共享主要是通过进程间通信（IPC）来实现，如管道、消息队列、共享内存等机制。

- 线程间共享的内容包括进程的地址空间、全局变量、堆内存等。线程可以直接读取和修改这些共享的数据。

- 需要注意的是，线程间共享数据时需要考虑同步和互斥的问题，以避免数据竞争和不一致的情况发生。常见的同步机制包括互斥锁、条件变量、信号量等。

总结起来，进程和线程的上下文切换涉及保存和恢复执行状态的信息，进程间的共享主要通过IPC机制实现，而线程间共享进程的资源，如地址空间、全局变量、堆内存等。

### 线程独占什么、切换时候内核做了什么

### 线程独占的资源和内核的操作

线程独占的资源指的是在执行过程中线程所需的一些系统资源，以及在切换时内核所执行的操作。下面详细说明：

#### 线程独占的资源：

1. \*\*CPU时间\*\*：线程在运行时，会占用CPU时间片来执行任务。

2. \*\*寄存器状态\*\*：线程在执行时会使用CPU的寄存器，这些寄存器的状态属于线程独占的资源。当线程切换时，当前线程的寄存器状态需要保存，下一个线程的寄存器状态需要加载。

3. \*\*栈空间\*\*：每个线程都有自己的栈空间，用于存储局部变量、函数调用信息等。线程独占的资源之一就是其所使用的栈空间。

4. \*\*线程本地存储（Thread Local Storage，TLS）\*\*：某些线程可能会使用到线程本地存储，这也属于线程独占的资源。

#### 内核切换线程时的操作：

1. \*\*保存上下文\*\*：内核在切换线程之前会保存当前线程的上下文信息，包括寄存器状态、程序计数器等。这些信息通常保存在线程的控制块（Thread Control Block，TCB）中。

2. \*\*选择下一个线程\*\*：内核会选择下一个要执行的线程，这通常由调度算法决定。

3. \*\*加载下一个线程的上下文\*\*：内核根据下一个线程的控制块中保存的信息，加载其上下文状态到CPU寄存器和其他相关硬件中。

4. \*\*切换页表\*\*：如果线程之间使用了不同的地址空间（比如不同的进程），内核可能需要切换页表以加载正确的内存映射关系。

5. \*\*更新调度信息\*\*：内核可能会更新线程的调度信息，如更新时间片、更新等待队列等。

总之，线程独占的资源包括CPU时间、寄存器状态、栈空间等；而内核在切换线程时会保存当前线程的上下文，并加载下一个线程的上下文，还可能涉及到页表切换和调度信息的更新。这些操作保证了线程切换的正确性和高效性。

### 进程上下文切换开销 (一直问 还有吗 还有吗 还有吗)

进程上下文切换的开销较高，主要由以下几个方面组成：

1. \*\*保存和恢复寄存器状态\*\*：在进行进程切换时，需要保存当前进程的寄存器状态，包括通用寄存器、程序计数器（PC）、堆栈指针（SP）等。这些寄存器中存储了当前进程的执行状态，因此在切换到下一个进程时需要将这些寄存器状态恢复到正确的值。

2. \*\*内存映射切换\*\*：每个进程都有自己的地址空间，包括代码段、数据段、堆、栈等。当进程切换时，操作系统需要切换内存映射关系，以确保新进程能够正确访问其地址空间中的数据和指令。这涉及到页表的更新和切换，可能需要更新内存管理单元（MMU）中的页表基地址等信息。

3. \*\*I/O状态保存和恢复\*\*：如果进程在执行过程中涉及到了I/O操作，比如读写文件、网络通信等，那么在进行进程切换时，可能需要保存和恢复相关的I/O状态，以确保I/O操作的正确性和连续性。这可能涉及到文件描述符、I/O缓冲区等的保存和恢复。

4. \*\*上下文切换代码执行\*\*：实际进行进程切换时，需要执行一系列的上下文切换代码，包括保存当前进程的状态、加载下一个进程的状态、更新调度信息等操作。这些代码的执行也会带来一定的开销。

5. \*\*缓存刷新\*\*：进程切换可能导致CPU缓存中的数据失效，因此在切换过程中可能需要刷新缓存，以确保新进程能够获得正确的数据。

综上所述，进程上下文切换的开销主要包括保存和恢复寄存器状态、内存映射切换、I/O状态保存和恢复、上下文切换代码执行以及缓存刷新等方面的开销。这些开销会影响系统的性能和响应速度，因此在设计和优化系统时需要充分考虑进程切换的开销。

### 进程的内存空间

进程的内存空间是指操作系统为每个进程分配的地址空间，用于存储程序代码、数据、堆和栈等信息。这个地址空间通常被划分为几个部分，每个部分用于不同的目的。

以下是进程的内存空间通常包含的几个部分：

1. \*\*代码段（Text Segment）\*\*：也称为只读段，存储程序的机器代码。这部分内存通常是只读的，用于存放程序的指令。

2. \*\*数据段（Data Segment）\*\*：存储程序中已初始化的全局变量和静态变量。这些变量在程序执行之前就已经被分配了内存空间，并且有初始值。

3. \*\*未初始化数据段（BSS Segment）\*\*：也称为未初始化数据段，存储程序中未初始化的全局变量和静态变量。这些变量在程序执行之前并没有被初始化，因此占用的内存空间会被初始化为0或空指针。

4. \*\*堆（Heap）\*\*：用于动态分配内存空间，例如通过malloc()、calloc()等函数分配的内存空间。堆的大小通常是不固定的，可以动态地扩展和收缩。

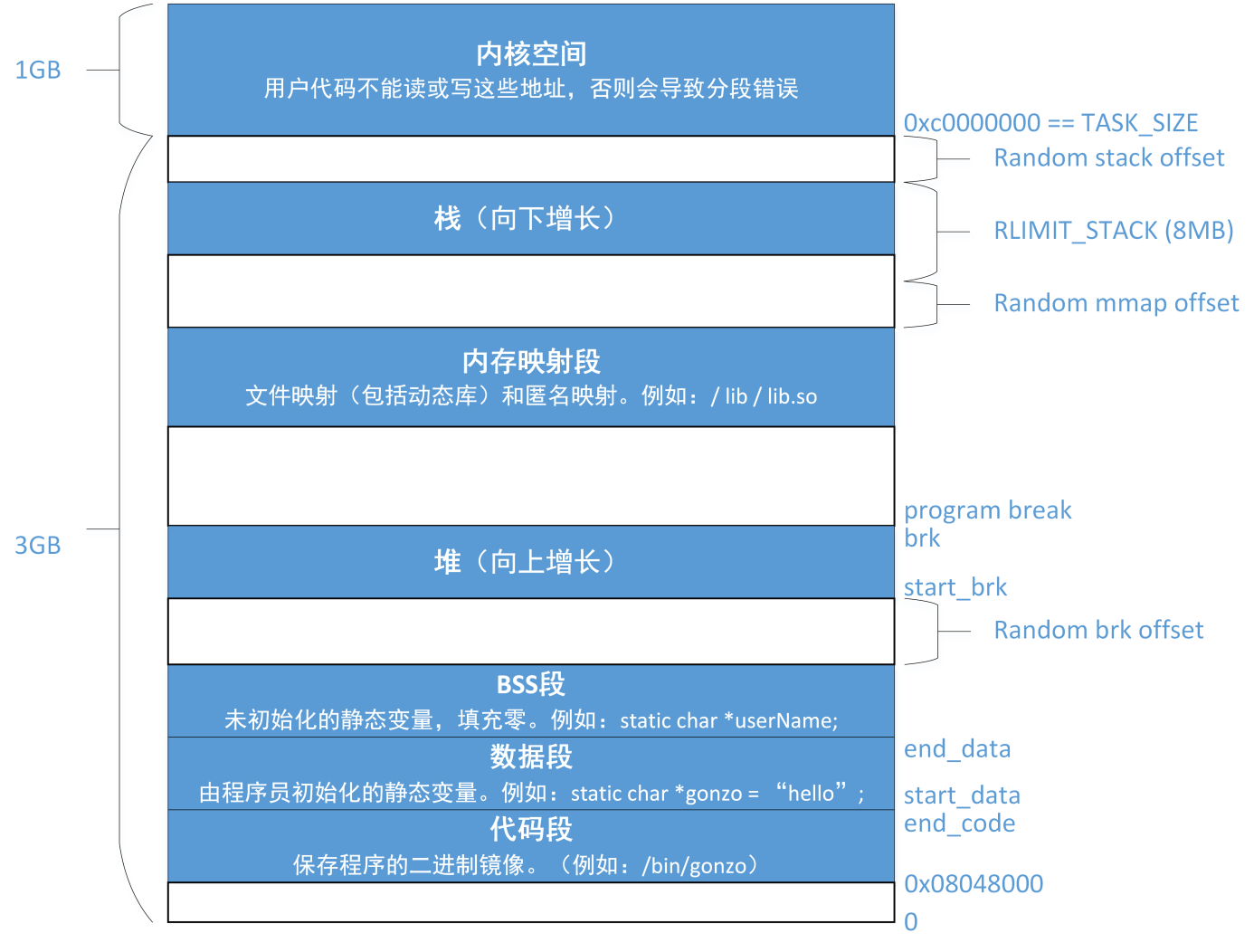
5. \*\*栈（Stack）\*\*：用于存储函数的局部变量、函数的参数值以及函数调用时的返回地址等信息。栈是一种后进先出（LIFO）的数据结构，每次函数调用时会在栈上分配一块内存空间，函数返回时会释放这些内存空间。

6. \*\*环境变量（Environment Variables）\*\*：存储进程运行时的环境变量信息，例如PATH、HOME等。

7. \*\*命令行参数（Command Line Arguments）\*\*：存储进程启动时传递的命令行参数信息。

每个进程的内存空间都是独立的，彼此之间不会相互干扰。操作系统通过虚拟内存管理来实现进程的内存空间隔离和保护，使得每个进程都认为自己拥有整个系统的内存空间。

<https://blog.csdn.net/cztqwan/article/details/80248479>



### 一个可执行文件如何变成进程的？

当一个可执行文件被执行时，它会通过以下步骤变成一个进程：

1. \*\*加载可执行文件\*\*：操作系统会将可执行文件从磁盘加载到内存中。这个过程称为加载（loading）或装载（loading）。加载过程包括将可执行文件的代码段、数据段和其他相关资源加载到内存的适当位置。

2. \*\*创建进程控制块\*\*：操作系统会为该进程创建一个数据结构，称为进程控制块（Process Control Block，PCB）。PCB中保存了进程的各种属性和状态信息，如进程ID、程序计数器、寄存器状态、内存映射关系、打开的文件描述符等。

3. \*\*分配内存空间\*\*：操作系统为进程分配内存空间，这个内存空间就成为进程的地址空间。地址空间是进程独立的，用于存储进程的代码、数据、堆、栈等信息。

4. \*\*设置初始状态\*\*：操作系统会设置进程的初始状态，包括将程序计数器设置为可执行文件的起始地址，将堆栈指针设置为栈的初始位置等。这样，当进程开始执行时，它会从可执行文件的起始地址开始执行。

5. \*\*开始执行\*\*：操作系统会将进程的控制权交给该进程，使其开始执行。进程会按照程序指令的顺序执行，读取数据、进行计算等操作。

6. \*\*执行完毕或被中断\*\*：进程会一直执行，直到完成任务或被中断。当进程完成任务时，它会释放占用的资源并通知操作系统进程的结束。当进程被中断时，它可能会暂停执行，并将控制权交还给操作系统。

通过以上步骤，一个可执行文件就变成了一个进程。操作系统负责管理和调度进程，确保它们能够正常执行和相互隔离。

### CPU调度基本单位？

CPU调度的基本单位是进程或线程。进程是资源分配的最小单位，线程是CPU调度的最小单位。在操作系统中，CPU调度算法决定了如何分配和管理CPU的执行时间，以使得多个进程或线程能够公平地共享CPU资源。

当多个进程或线程同时处于就绪状态时，操作系统会根据调度算法选择一个进程或线程，并将CPU的控制权交给选中的进程或线程，使其开始执行。被选中的进程或线程会在CPU上运行一段时间，直到完成任务、被中断或时间片用尽。

CPU调度的基本单位可以是进程或线程，具体取决于操作系统的实现和调度策略。在传统的操作系统中，CPU调度的基本单位是进程。每个进程拥有自己的地址空间、资源和执行上下文。而在一些现代操作系统中，引入了线程级别的调度，即将CPU调度的基本单位缩小为线程。线程是进程的执行单元，多个线程可以共享同一个进程的资源和地址空间。

不管是以进程还是线程为基本单位，CPU调度的目标都是提高系统的性能和资源利用率，以及确保进程或线程的公平性和响应性。调度算法的选择和优化对于系统的性能、吞吐量和响应时间有重要影响。

### 进程和线程的优缺点，各适用于什么场合？

进程和线程各自有其优缺点，适用于不同的场合。

进程的优缺点

优点：

独立性：每个进程都有独立的地址空间，一个进程崩溃不会影响其他进程。

隔离性：进程间通信需要通过操作系统提供的机制（如管道、消息队列、共享内存等），这有助于保护进程间的数据不被非法访问。

稳定性：由于进程间相对独立，一个进程的问题通常不会导致整个系统崩溃。

多CPU利用：多个进程可以运行在不同的CPU上，充分利用多核处理器的性能。

缺点

创建和销毁开销大：进程的创建和销毁需要较多的系统资源。

通信开销大：进程间通信需要通过操作系统，这通常比线程间通信要慢。

上下文切换开销大：进程的上下文切换比线程的上下文切换开销大。

适用场合

需要较高稳定性和隔离性的场合，如服务器软件、大型应用程序。

需要充分利用多CPU或多核心的场合。

线程的优缺点

优点

创建和销毁开销小：线程的创建和销毁通常比进程快。

通信开销小：同一进程内的线程可以直接读写共享内存，通信效率高。

上下文切换开销小：线程的上下文切换比进程快。

共享资源：线程可以共享进程的地址空间和资源，如打开的文件描述符、数据库连接等。

缺点

不稳定：一个线程的崩溃可能影响到同一进程内的其他线程。

数据同步问题：由于线程共享地址空间，需要额外的同步机制（如互斥锁、条件变量等）来保护共享数据。

受限于地址空间：所有线程共享同一地址空间，受限于地址空间的限制。

适用场合

需要快速并发执行的任务，如GUI应用程序、Web服务器。

需要大量并发执行且通信频繁的任务。  
总的来说，选择进程还是线程，需要根据具体的应用场景和需求来决定。在实际的应用程序设计中，经常会结合使用进程和线程，以达到最佳的并发性能和资源利用率。

### fork的作用是什么？为什么不用system

fork 是 Unix 和类 Unix 系统中的一个系统调用，其主要作用是创建一个与当前进程几乎完全相同的子进程。以下是 fork 的一些特点和它与传统 system 调用的区别：

fork 的作用：

创建子进程：fork 创建一个子进程，该子进程是调用 fork 的进程的副本，包括代码段、数据段、堆、栈等。

返回值：fork 在父进程和子进程中都有返回值。在父进程中，返回子进程的进程 ID；在子进程中，返回 0。

执行流分叉：fork 之后，父进程和子进程的执行流开始分开，它们可以独立执行不同的代码段。

为什么使用 fork 而不使用 system：

性能：fork 比起 system 来说，性能更高，因为 system 调用会创建一个 shell 进程来执行命令，而 fork 直接创建一个新的进程，避免了额外的进程开销。

控制：fork 提供了更细粒度的控制。在子进程中，你可以执行任何你想要的操作，而不需要依赖 shell。

安全性：使用 fork 可以避免由于 shell 解释命令带来的潜在安全问题，例如注入攻击。

资源使用：fork 允许子进程共享父进程的资源，如文件描述符、内存等，而 system 创建的进程是独立的。

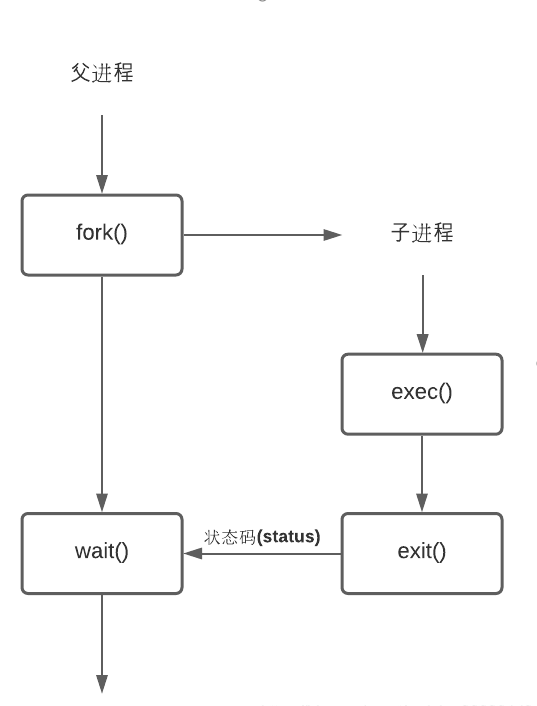
同步：fork 允许父进程和子进程之间进行同步操作，比如使用 wait 或 waitpid 系统调用来等待子进程结束。

system 的缺点：

system 调用执行一个 shell 命令，并且等待命令完成，这涉及到创建一个新的 shell 进程和执行命令的进程，增加了开销。

system 调用需要传递命令字符串，这可能存在注入风险，如果命令字符串来源于不可信的输入。

system 调用的执行流程不可控，因为它依赖于 shell 的行为。  
因此，当需要精细控制进程创建和执行流程时，通常选择 fork 而不是 system。然而，如果只需要执行一个简单的命令，并不关心其执行细节，system 仍然是一个方便的选择。



### 锁机制，信号量

锁机制和信号量是操作系统中用于同步和互斥的两种常见并发控制工具。它们在多线程或多进程环境中确保共享资源的正确访问和防止竞态条件。

锁机制（Lock）

锁是一种保证在任一时刻只有一个线程可以访问共享资源的同步机制。

特点：

互斥：确保同一时间只有一个线程可以持有锁。

原子操作：锁的操作通常是原子的，即不可中断的。

资源保护：锁用于保护临界区，防止多个线程同时修改共享资源。

类型：

自旋锁：当锁不可用时，线程会循环检查锁是否可用，适用于锁被占用时间短的场景。

递归锁：允许同一线程多次获取同一把锁，常用于递归调用。

读写锁：允许多个读操作同时进行，但写操作是互斥的。

使用场合：

当需要保护共享数据结构时，如链表、树、哈希表等。

在多线程环境中，需要确保代码的原子性时。

信号量（Semaphore）

信号量是一个更高级的同步机制，它是一个整数变量，可以执行两种操作：P（等待）和V（信号）。

特点：

计数信号量：可以用来控制对某一资源的访问数量。

二进制信号量：特殊情况下，信号量的值只能是 0 或 1，这时它就退化成普通的锁。

系统调用：信号量通常是通过操作系统提供的系统调用来实现的。

使用场合：

当需要限制对某一资源的并发访问数量时，如限制同时访问数据库的连接数。

在生产者-消费者模型中，用来同步生产者和消费者的操作。

对比：

锁：通常用于简单的互斥操作，实现起来相对简单，适用于轻量级的同步需求。

信号量：提供了更复杂的同步功能，如资源计数，适用于需要精确控制资源访问的场景。  
在实际应用中，锁和信号量可以结合使用，以解决复杂的并发控制问题。例如，在实现一个进程池或线程池时，可能会使用信号量来控制同时运行的任务数量，同时使用锁来保护任务队列的访问。

### 66、通知其他线程等问题

### 67、内核创建进程时会自动打开哪几个文件。

### 68、进程控制块

### 70、Linux fork fork 原理。

### 71、什么情况下需要多线程频繁切换？

### 72、僵尸进程是什么，怎么处理

### 73、threadlocal的实现，原理，业务用来做什么？

### 74、程序什么时候应该使用线程，什么时候单线程效率高

### 75、cpu怎么调度进程？调度算法？

<https://blog.csdn.net/sandalphon4869/article/details/104984046>

### 76、多线程好处

### 77、进程调度，CFS。

### 78、进程怎么实现访问隔离

### 79、线程有什么数据是自己私有，哪些是共享的（虚拟处理器、指令计数器）

### 80、io密集型和计算密集型分别适合多线程还是多进程

### 81、线程的堆栈，全局静态变量，哪些对象共享内存

### 82、如何理解协程不被操作系统内核管理，而完全是由程序控制？

### 83、进程a，hello world，把a的内存发送给了进程b，进程b可以访问这段内存吗

### 84、你知道的进程调度算法有哪些？详细介绍一下？

### 85、用户级线程和内核级线程的区别

### 86、进程有哪些状态，怎么转换的

### 87、线程状态及转化

### 88、进程的创建需要系统分配什么资源

### 89、讲讲进程管理 你平时自己有过进程管理吗

### 90、进程调度算法有哪些，最短作业优先怎么样的，有什么弊端

### 91、什么是僵尸进程和孤儿进程 具体怎么查看

### 92、进程、线程、协程、线程池，线程切换的资源消耗和时间消耗

### 93、守护进程是什么？为什么

### 95、进程可以忽视信号吗（可以，不过看级别）

### 96、线程如何切换

### 97、知道协程吗？协程是在生命周期的哪一步？

### 98、那你知道什么进程之间调度？的算法吗

### 99、什么情况适合用协程池，什么情况适合用线程池（IO密集协程，资源多线程）

### 100、"fork会拷贝页表吗

### 101、fork的读时共享，写时拷贝

### 102、fork在写时拷贝之前，直接拷贝的内容"

### 进程间通信

### 进程间通信的最快方式？

### 进程通信有哪些方式方式 共享内存是啥

### 异步通信和同步通信的区别？

### 进程间通信的最快方式？

### 使用共享内存通信有什么缺点？

### 线程同步的方法；进程通信七种方法。

### 进程通信那种速度最快

### 共享内存有了解吗，不同进程的地址是不是一样呢，原因

### 进程同步方式

### 进程间通信？说一下原理?共享内存是如何确定物理地址的？

### 生产者消费者，信号量的使用

### 信号量和自旋锁的区别

### 线程的内存空间

### 多线程栈之间传递数据

### Linux 文件标识符，父进程fork子进程后，子进程能不能修改父进程的文件还是啥（我打能，对了，原因打得不好）

### 项目用过消息队列吗，为什么不消息队列选择用信号量

### 管道用过吗，会溢出吗，为什么会

### 如何限制一个进程能够使用的线程等资源

### 线程之间如何通信

### 线程的状态

### 操作系统中是如何实现线程的（不知道该怎么答）

### PCB了解吗？

### "进程之间的通信方式；

### 线程之间的通信方式；

### 线程之间的数据同步、数据竞争；"

### 操作系统最小是线程，那协程怎么实现

### 说说多个线程对多个协程原理设计

### 同一台机器，同一个应用的线程，不同机器之间如何通信

### 时间片轮转算法详细说一下

### 共享内存，如何控制读取写入的一致性（信号量）

### 不同进程之间是否可以共享内存？父子进程也不可以吗？（读时共享，写时拷贝）

### 一句话概括信号量的作用

### 管道怎么用，数据写到哪里（内核）

### 线程之间通信方式

### "孤儿进程跟僵尸进程

### 父进程怎么回收僵尸进程的资源"

### 死锁是什么，死锁发生了怎么办

### 用互斥锁实现读写锁，写者优先

### 锁机制，信号量，开锁时如何通知其他线程等问题

### 什么是死锁，死锁的条件？怎么解决？

### 产生死锁的原因？

### 如何预防死锁，！避免死锁，死锁发生的检查？

### 项目中多线程的应用，多线程要注意那些点？

### 生产者消费者模式

### 锁有哪几种，自旋锁和互斥锁的区别

### 内存屏障分几种？

### 同步互斥、信号量

### 条件变量有什么缺点 为什么要和mutex搭配使用

### 多线程编程要注意些什么

### 操作系统中的原子操作是怎么实现的？

### 在线程从任务队列中取任务时，有没有办法不适用锁?

### 线程同步的方式

### 线性一致性

### 用锁的性能会不会降低

### 死锁的发生、解决、检查；

### 模拟死锁

### 如何判断多线程内存冲突

### CAS中有一个ABA问题，你能介绍一下吗？怎么解决CAS中的ABA问题？

### CAS有哪些问题？

### 原子性、锁，还有硬件如何实现（原子硬件我蒙的指令集，蒙对了，锁没答上来，不同处理器架构可能有不同的锁实现方式，他当时问的我英特尔的芯片是如何实现的）

### 并发和并行

### pthread函数有几个参数和状态。

### 多线程相关的问题 锁 cas aqs等等

### 解释读写锁

### 为什么用线程池不用进程池（多线程是否能完全取代多进程）

### 什么是线程的同步，什么是线程的异步，分别有什么样的使用场景

### 用过读写锁吗

### 使用条件变量是需要注意什么

### 锁的底层原理

### 锁的种类和分别的作用（互斥锁、自旋锁、读写锁）

### sleep和wait方法的区别

### 如何避免互斥发生

### epoll实现机制，为什么返回的都是就绪的，和select的区别，为什么select不能做到

### 讲一下select，poll , epoll区别

### epoll水平触发和边缘触发

### epoll的边缘触发，触发没有处理，后面怎么办（事件是不是都在队列不处理了？）？如果后面又处理这个事件，会丢掉吗还是一直都在队列里面了？

### recv函数的返回值及其意义、read 函数的返回值你了解吗

### socket.accept()函数对应着TCP三次握手中哪一次握手，哪个函数代表连接建立完毕，四次挥手的过程？

### 一台机器最多可以建立多少个tcp连接，client端，server端，超过了怎么办

### socket编程服务端和客户端调用函数的流程

### accept函数的参数和返回值分别是什么

### 介绍第四个项目；reactor 和 proactor 的区别，知道哪些项目是基于proactor 的吗

### 如果不用io多路复用该用什么

### listen参数block 的意义

### 介绍五大 IO 模型

### 为什么要使用Reactor模式, 为什么不直接用多线程

### epoll\_wait 是怎么处理的，time\_out 值等

### socket编程，如果客户端崩溃服务器会怎么样

### window下的IOCP，完成端口

### reactor和proactor的优缺点了解过吗？使用场景

### Eventloop是怎么实现的

### poll和select除了参数上还有什么区别

### 一个线程会发生死锁吗（比如：多次尝试同一个锁、递归获取锁）

### Nio

### 用过select水平触发吗，讲讲epoll的水平触发和定时，怎么实现定时

### socket通信里write()的返回值

### epoll\_wait怎么获取事件的

### socket通讯的流程，深入到kernel层面

### Epoll 详解，Epoll惊群避免

### 如何修改socket接收缓冲区大小

### NIO、BIO、同步IO、异步IO，以及它们与IO多路复用的区别和联系

### 介绍一下字节序

### 项目中使用了哪些api，详细说说setsocketopt和fcntl的作用；

### listen参数，listen和accept返回值有什么不同？

### listen 函数第二个参数 backlog 参数作用

### 寄存器/cache优化

### cache和内存关系？

### cache line是什么

### 虚拟内存寻址空间大小

### 虚拟内存中 堆和栈的区别

### 内存分配，从代码段讲到栈了 问我栈上面是什么

### 操作系统执行可执行程序时，内存分配是怎样的？

### 堆和栈的区别？

### page fault

### 什么是虚拟内存? 什么是物理内存？

### 缺页了怎么办，物理内存如何分配的

### 一般程序中栈的大小

### 内存分配原理。

### new 和 malloc，malloc如果越界访问会怎么样？free怎么实现的？为什么不用指定大小？

### 内存的分配方式有几种?

### TLB是什么

### mmap

### 知道有什么硬件存储吗？

### 虚拟内存，页表，TLB；

### 内存池怎么可以保证不会产生碎片

### 内存碎片解决Linux系统还是Windows的，为什么操作系统设计这种产生内存碎片的机制

### malloc函数的原型是什么？

### c和c++里如何动态申请一块内存？怎么判断它们有没有申请成功呢？

### 有哪些情况可能会申请内存失败？

### Linux虚拟内存，为什么用虚拟内存，为什么使用逻辑地址不使用物理地址

### Linux copyonwrite

### 这个物理页面大小怎么选择的？为什么要选择4KB大小的页面？

### 虚拟地址是怎么转化到物理地址的？页表的构成？mmu了解过吗？

### 虚拟内存有哪些区域

### 虚拟内存怎么映射到物理内存 转换用到的媒介是什么

### 内存回收

### 内存碎片

### 如果频繁进行内存的分配释放会有什么问题吗？

### 如果频繁分配释放的内存很大（>128k）,怎么处理？

### 操作系统对内存管理的两种方式是什么？

### 非连续内存管理你了解几种方式？

### 如何保证内存安全（保证内存不泄露）？

### 虚拟地址是怎么转换的

### 为什么会有栈溢出，为什么栈有设置容量

### 页表用的什么数据结构 ？

### linux分配内存 ？（物理内存 slab + buddy算法）

### 分页有什么好处，为什么要分页，为什么需要虚拟内存，虚拟内存有什么用，虚拟内存如何映射到物理内存，虚拟地址怎么转换为物理地址。页面置换算法有哪几种。怎么实现的。

### 如果1g的内存，能否申请2g的空间，8g呢？

### malloc线程安全吗，多线程malloc系统怎么设计

### 怎么优化内存使用率，减少内存碎片

### 内存为什么设计栈和堆，不能都在堆上分配吗

### 内存模型（不知道指的是啥，问是不是堆栈，面试官说不是，就说不问这个了...）

### malloc是如何跟物理地址关联上的？

### 碎片管理 swap做什么的？

### 操作系统的内存管理的页面淘汰 算法 ，介绍下LRU（最近最少使用算法 ）

### 怎么建立虚拟地址和物理地址之间的映射

### 如果要映射一个4g的空间，需要多大的物理地址？

### 有内存泄漏发生如何查找解决

### 怎么避免内存颠簸(不会)

### 给一个虚拟地址，具体怎么寻址，介绍页式、段页式

### 如何实现对象持久化存储（存储到磁盘）mmap原理

### 内存泄露原因及如何检测与预防，使用过哪些工具，原理是

### 堆内存和栈内存的区别

### 堆内存和栈内存的性能差距

### 一直频繁操作堆内存会发生什么

### 虚拟内存的大小有什么限制，举个例子

### 开发时如何避免内存泄漏

### "百万级行代码项目出现内存泄漏如何定位，答：第三方库

### 这个第三方库的实现原理知道吗(自己给自己挖坑😭)，如果让你自己设计一个内存泄漏检测工具，你会怎么做(会个der)"

### 内存置换在什么场景下使用LRU算法

### "malloc分配方式

### 具体怎么分配的，有啥缺点。

### 传一个指针进来，它如何确定具体要清理多少空间？"

### "内存泄露如何定位？答了gdb和valgrind(memcheck)

### 问:那你知道memcheck它具体怎么实现的吗？"

### 文件系统，创建文件时发生了啥

### Linux 还问了文件系统

### 中断

### 什么是系统调用

### 什么是阻塞调用

### 用过那些异步库，同步和异步的区别

### 中断的原理及实现

### 硬中断和软中断

### linux的信号底层原理（在内核怎么处理，面试官提示：软中断）

### 讲一下知道哪些类型的中断

### 讲一下内核态软件中断和硬件中断处理流程有什么不同

### 系统调用过程发生了什么

### 内核态和用户态转换发生了什么

### 为什么分用户态内核态，有什么区别（cpu权限）？

### 怎么减少用户态内核态切换？

### 中断是什么？有哪些流程

### 缺页中断，分别说程序和系统层面怎么减少缺页中断

### 用户程序想访问硬件资源应怎么办

### 读取到用户空间一块地址不在物理内存，怎么处理

### 什么时候会从用户态陷入内核态

### 用户态和内核态？怎么切换？为什么要有内核态？内核态可以进行哪些操作，举具体例子？

### 平时怎么调试，内存不断增长的话怎么判断哪里出了问题，调试方面问了很多，用了什么调试方式

### 内存泄露、内存泄露检测工具

### 递归程序的缺点怎么维护

### CPU的核心怎么分配 怎么保证不同的核心获取的数据不冲突

### 内存泄露检测工具 内存泄漏怎么处理

### 线上机器cpu100%你怎么处理

### cpu利用率暴涨怎么排查怎么处理

### 你在项目中碰到什么比较棘手的问题？怎么解决的？

### gdb调试器怎么调试多线程，怎么查看堆栈

### 调试方法有哪些？说出优缺点

### 如何调试coredump文件？

### 内存溢出和内存泄漏是什么？

### gdb调试，指定线程，条件断点

### 断点怎么实现，为什么到这里就停了

### dump出来的文件一般多大呢？（G） 用什么东西dump出来嗯

### gdb中，如何查看每个线程相关的信息

### 如何分析报错生成的core文件

### 如何使用 gdb 排查多线程中哪个函数出现了死锁（info thread & thread 2 & bt）

### cpu密集型和i/o密集型什么意思

### 使用过perf吗

### 怎么查看tlb命中率

### 如何查看进程的线程数

### 你是怎么debug并发程序的

### coredump还原

### gdb断点

### gdb操作, gdb调试多线程和死锁

### 如何监控服务器cpu信息，内存使用率，io等

### segment error的原因都有哪些

### 动态编链和静态编链

### 动态库 静态库

### 编译器为什么要按字节对齐

### 什么时字节序 为什么需要字节对齐，什么因素会影响字节对齐。可以让字节以1对齐吗？

### 为什么通过一个宏就能改变内存字节对齐方式

### makefile怎么构建

### cmake

### cmake输出可执行文件的指令

### 赋值操作中，cpu层面的具体流程

### 源码到可执行文件的过程 链接在做什么？

### gcc生成可执行文件的原生指令

### 你什么时候用到了CMake/Makefile

### zero copy

### nio

### 计算机启动

### Ctrl C Ctrl Z 实现

### 超线程是什么？知道吗？

### 64位和32位的区别

### 内核有哪几个子系统

### cpu由哪些组成

### BP SP寄存器是什么

### 你了解哪些寄存器

### dma

### 零拷贝

### CPU缓存

### CPU亲和型

### 阻塞性IO好处(bio)？

### PageCache

### 阻塞IO、非阻塞IO

### 磁盘IO？什么情况下会发生磁盘IO

### Direct IO

### buffer read

### 问个机组的问题吧,最后一个,计算机IO输入输出的时候硬链接和软链接在这里发挥的作用,整个流程你说一下吧

### Linux里kernel里的buffer如何通知user的

### 敲键盘发生的中断

### 非阻塞IO

### 异步IO和非阻塞IO

### eflag寄存器作用

### 优化程序，从CPU和内存考虑：主要答了核亲和性绑定以及内存池技术

### eBPF技术还有哪些方面可以提升？

### 如何实现cpu的虚拟化

### cpu访问磁盘数据和内存数据性能对比

### arm架构和x86架构区别

### 软链接和硬链接的区别（软链接文件与源文件的inode号不同，类似与源文件的快捷方式；硬链接的inode号与源文件相同，相当于源文件的另一个文件名）；

### 一次IO操作的完整流程

### 是否了解cpu catch？

### 你怎么理解异步的？怎么实现的？

### 零拷贝

### "数据类型转换方式

### 计算机存储类型（寄存器，内存，磁盘）一个程序中那些地方需要访问寄存器，内存，磁盘"

### cache是干嘛的，那些地方用到过

### 手机cpu和计算机cpu有啥区别

### cas的ABA问题怎么解决

### 系统调用实现过程

### 网卡是干啥的，网卡收发数据是通过什么实现的

### 虚拟化实现的方式（软件实现，硬件实现，容器）以及硬件如何实现

### 虚拟地址和物理地址转化

## 4、计算机网络

### 1、OSI 的七层模型分别是？各自的功能是什么？

简要概括

物理层：底层数据传输，如网线；网卡标准。

数据链路层：定义数据的基本格式，如何传输，如何标识；如网卡MAC地址。网络层：定义IP编址，定义路由功能；如不同设备的数据转发。

传输层：端到端传输数据的基本功能；如 TCP、UDP。

会话层：控制应用程序之间会话能力；如不同软件数据分发给不同软件。表示层：数据格式标识，基本压缩加密功能。

应用层：各种应用软件，包括 Web 应用。

说明：

在四层，既传输层数据被称作**段**（Segments）； 三层网络层数据被称做**包**（Packages）；

二层数据链路层时数据被称为**帧**（Frames）； 一层物理层时数据被称为**比特流**（Bits）。

总结

网络七层模型是一个标准，而非实现。网络四层模型是一个实现的应用模型。网络四层模型由七层模型简化合并而来。

### 2、说一下一次完整的HTTP请求过程包括哪些内容？

第一种回答

建立起客户机和服务器连接。

建立连接后，客户机发送一个请求给服务器。服务器收到请求给予响应信息。

客户端浏览器将返回的内容解析并呈现，断开连接。

第二种回答

域名解析 --> 发起TCP的3次握手 --> 建立TCP连接后发起http请求 --> 服务器响应http请求，浏览器得到html代码 --> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） --> 浏览器对页面进行渲染呈现给用户。

### 7、HTTP长连接和短连接的区别

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接， 任务结束就中断连接。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。

### 8、什么是TCP粘包/拆包？发生的原因？

一个完整的业务可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发 送，这个就是TCP的拆包和粘包问题。

原因

1、应用程序写入数据的字节大小大于套接字发送缓冲区的大小.

2、进行MSS大小的TCP分段。( MSS=TCP报文段长度-TCP首部长度)

3、以太网的payload大于MTU进行IP分片。（ MTU指：一种通信协议的某一层上面所能通过的最大数据包大小。）

解决方案

1、消息定长。

2、在包尾部增加回车或者空格符等特殊字符进行分割

3. 将消息分为消息头和消息尾。

4. 使用其它复杂的协议，如RTMP协议等。

### 10、HTTP请求方法你知道多少？

客户端发送的 **请求报文** 第一行为请求行，包含了方法字段。根据 HTTP 标准，HTTP 请求可以使用多种请求方法。HTTP1.0 定义了三种请求方法： GET, POST 和 HEAD方法。

HTTP1.1 新增了六种请求方法：OPTIONS、PUT、PATCH、DELETE、TRACE 和 CONNECT 方法。

|  |  |  |
| --- | --- | --- |
| **序号** | **方法** | **描述** |
| 1 | GET | 请求指定的页面信息，并返回实体主体。 |
| 2 | HEAD | 类似于 GET 请求，只不过返回的响应中没有具体的内容，用于获取报头 |
| 3 | POST | 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被 包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。 |
| 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容。 |
| 5 | DELETE | 请求服务器删除指定的页面。 |
| 6 | CONNECT | HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。 |
| 7 | OPTIONS | 允许客户端查看服务器的性能。 |
| 8 | TRACE | 回显服务器收到的请求，主要用于测试或诊断。 |
| 9 | PATCH | 是对 PUT 方法的补充，用来对已知资源进行局部更新 。 |

### 11、GET 和 POST 的区别，你知道哪些？

* 1. get是获取数据，post是修改数据
  2. get把请求的数据放在url上， 以?分割URL和传输数据，参数之间以&相连，所以get不太安全。而

post把数据放在HTTP的包体内（requrest body）

* 1. get提交的数据最大是2k（ 限制实际上取决于浏览器）， post理论上没有限制。
  2. GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据); POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。
  3. GET请求会被浏览器主动缓存，而POST不会，除非手动设置。
  4. 本质区别：GET是幂等的，而POST不是幂等的

这里的幂等性：幂等性是指一次和多次请求某一个资源应该具有同样的副作用。简单来说意 味着对同一URL的多个请求应该返回同样的结果。

正因为它们有这样的区别，所以不应该且**不能用get请求做数据的增删改这些有副作用的操作**。因为get 请求是幂等的，**在网络不好的隧道中会尝试重试**。如果用get请求增数据，会有**重复操作**的风险，而这 种重复操作可能会导致副作用（浏览器和操作系统并不知道你会用get请求去做增操作）。

### 12、一个TCP连接可以对应几个HTTP请求？

如果维持连接，一个 TCP 连接是可以发送多个 HTTP 请求的。

### 13、一个 TCP 连接中 HTTP 请求发送可以一起发送么（比如一起发三个请求，再三个响应一起接收）？

HTTP/1.1 存在一个问题，单个 TCP 连接在同一时刻只能处理一个请求，意思是说：两个请求的生命周期不能重叠，任意两个 HTTP 请求从开始到结束的时间在同一个 TCP 连接里不能重叠。

在 HTTP/1.1 存在 Pipelining 技术可以完成这个多个请求同时发送，但是由于浏览器默认关闭，所以可以认为这是不可行的。在 HTTP2 中由于 Multiplexing 特点的存在，多个 HTTP 请求可以在同一个 TCP 连接中并行进行。

那么在 HTTP/1.1 时代，浏览器是如何提高页面加载效率的呢？主要有下面两点： 维持和服务器已经建立的 TCP 连接，在同一连接上顺序处理多个请求。

和服务器建立多个 TCP 连接。

### 14、浏览器对同一 Host 建立 TCP 连接到数量有没有限制？

假设我们还处在 HTTP/1.1 时代，那个时候没有多路传输，当浏览器拿到一个有几十张图片的网页该怎么办呢？肯定不能只开一个 TCP 连接顺序下载，那样用户肯定等的很难受，但是如果每个图片都开一个TCP 连接发 HTTP 请求，那电脑或者服务器都可能受不了，要是有 1000 张图片的话总不能开 1000 个TCP 连接吧，你的电脑同意 NAT 也不一定会同意。

有。Chrome 最多允许对同一个 Host 建立六个 TCP 连接。不同的浏览器有一些区别。

如果图片都是 HTTPS 连接并且在同一个域名下，那么浏览器在 SSL 握手之后会和服务器商量能不能用HTTP2，如果能的话就使用 Multiplexing 功能在这个连接上进行多路传输。不过也未必会所有挂在这个域名的资源都会使用一个 TCP 连接去获取，但是可以确定的是 Multiplexing 很可能会被用到。

如果发现用不了 HTTP2 呢？或者用不了 HTTPS（现实中的 HTTP2 都是在 HTTPS 上实现的，所以也就是只能使用 HTTP/1.1）。那浏览器就会在一个 HOST 上建立多个 TCP 连接，连接数量的最大限制取决于浏览器设置，这些连接会在空闲的时候被浏览器用来发送新的请求，如果所有的连接都正在发送请求 呢？那其他的请求就只能等等了。

### 15、在浏览器中输入url地址后显示主页的过程?

根据域名，进行DNS域名解析；

拿到解析的IP地址，建立TCP连接； 向IP地址，发送HTTP请求；

服务器处理请求； 返回响应结果； 关闭TCP连接；

浏览器解析HTML； 浏览器布局渲染；

### 16、在浏览器地址栏输入一个URL后回车，背后会进行哪些技术步骤？

第一种回答

1、查浏览器缓存，看看有没有已经缓存好的，如果没有 2 、检查本机host文件，

3、调用API，Linux下Scoket函数 gethostbyname

4、向DNS服务器发送DNS请求，查询本地DNS服务器，这其中用的是UDP的协议

6、如果在一个子网内采用ARP地址解析协议进行ARP查询如果不在一个子网那就需要对默认网关进行DNS查询，如果还找不到会一直向上找根DNS服务器，直到最终拿到IP地址（全球好像一共有13台根服 务器）

7、这个时候我们就有了服务器的IP地址 以及默认的端口号了，http默认是80 https是 443 端口号， 会，首先尝试http然后调用Socket建立TCP连接，

8、经过三次握手成功建立连接后，开始传送数据，如果正是http协议的话，就返回就完事了，

9、如果不是http协议，服务器会返回一个5开头的的重定向消息，告诉我们用的是https，那就是说IP

没变，但是端口号从80变成443了，好了，再四次挥手，完事，

10、再来一遍，这次除了上述的端口号从80变成443之外，还会采用SSL的加密技术来保证传输数据的 安全性，保证数据传输过程中不被修改或者替换之类的，

11、这次依然是三次握手，沟通好双方使用的认证算法，加密和检验算法，在此过程中也会检验对方的 CA安全证书。

12、确认无误后，开始通信，然后服务器就会返回你所要访问的网址的一些数据，在此过程中会将界面 进行渲染，牵涉到ajax技术之类的，直到最后我们看到色彩斑斓的网页

第二种回答

浏览器检查域名是否在缓存当中（要查看 Chrome 当中的缓存， 打开 chrome://net- internals/#dns）。

如果缓存中没有，就去调用 库函数（操作系统不同函数也不同）进行查询。

gethostbyname

gethostbyname 函数在试图进行DNS解析之前首先检查域名是否在本地 Hosts 里，Hosts 的位置 [不同的操作系统有所不同]

(https://en.wikipedia.org/wiki/Hosts\_(file)#Location\_in\_the\_file\_system)

如果 gethostbyname 没有这个域名的缓存记录，也没有在 hosts` 里找到，它将会向 DNS 服务器发送一条

DNS 查询请求。DNS 服务器是由网络通信栈提供的，通常是本地路由器或者 ISP 的缓存 DNS 服务器。

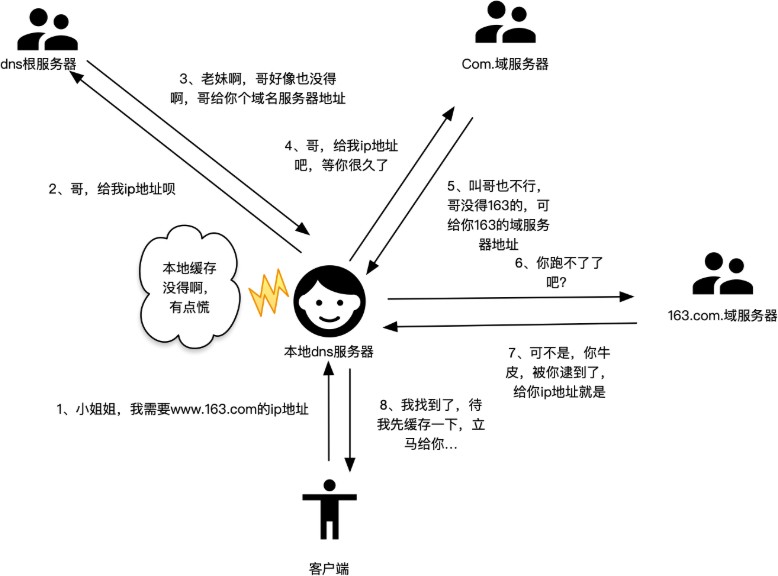
查询本地 DNS 服务器

如果 DNS 服务器和我们的主机在同一个子网内，系统会按照下面的 ARP 过程对 DNS 服务器进行 ARP

查询

如果 DNS 服务器和我们的主机在不同的子网，系统会按照下面的 ARP 过程对默认网关进行查询参考：<https://www.zhihu.com/question/34873227/answer/518086565>

### 17、谈谈DNS解析过程，具体一点



请求一旦发起，若是chrome浏览器，先在浏览器找之前**有没有缓存过的域名所对应的ip地址**，有 的话，直接跳过dns解析了，若是没有，就会**找硬盘的hosts文件**，看看有没有，有的话，直接找 到hosts文件里面的ip

如果本地的hosts文件没有能的到对应的ip地址，浏览器会发出一个**dns请求到本地dns服务器**，**本 地dns服务器一般都是你的网络接入服务器商提供**，比如中国电信，中国移动等。

查询你输入的网址的DNS请求到达本地DNS服务器之后，**本地DNS服务器会首先查询它的缓存记 录**，如果缓存中有此条记录，就可以直接返回结果，此过程是**递归的方式进行查询**。如果没有，本 地DNS服务器还要向**DNS根服务器**进行查询。

本地DNS服务器继续向域服务器发出请求，在这个例子中，请求的对象是.com域服务器。.com域 服务器收到请求之后，也不会直接返回域名和IP地址的对应关系，而是告诉本地DNS服务器，你的 域名的解析服务器的地址。

最后，本地DNS服务器向**域名的解析服务器**发出请求，这时就能收到一个域名和IP地址对应关系， 本地DNS服务器不仅要把IP地址返回给用户电脑，还要把这个对应关系保存在缓存中，以备下次别 的用户查询时，可以直接返回结果，加快网络访问。

### 18、DNS负载均衡是什么策略？

当一个网站有足够多的用户的时候，假如每次请求的资源都位于同一台机器上面，那么这台机器随时可 能会蹦掉。处理办法就是用DNS负载均衡技术，它的原理是在**DNS服务器中为同一个主机名配置多个IP 地址,在应答DNS查询时,DNS服务器对每个查询将以DNS文件中主机记录的IP地址按顺序返回不同的解 析结果,将客户端的访问引导到不同的机器上去,使得不同的客户端访问不同的服务器**,从而达到负载均衡 的目的｡例如可以根据每台机器的负载量，该机器离用户地理位置的距离等等。

### 19、HTTPS和HTTP的区别

1、HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安 全， HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

2、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

参考：<https://www.cnblogs.com/wqhwe/p/5407468.html>

### 20、什么是SSL/TLS ？

SSL代表安全套接字层。它是一种用于加密和验证应用程序（如浏览器）和Web服务器之间发送的数据 的协议。 身份验证 ， 加密Https的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机

制。

SSL/TLS协议作用：认证用户和服务，加密数据，维护数据的完整性的应用层协议加密和解密需要两个 不同的密钥，故被称为非对称加密；加密和解密都使用同一个密钥的

对称加密：优点在于加密、解密效率通常比较高 ，HTTPS 是基于非对称加密的， 公钥是公开的，

### 21、HTTPS是如何保证数据传输的安全，整体的流程是什么？（SSL是怎么工作保证安全的）

1. 客户端向服务器端发起SSL连接请求；
2. 服务器把公钥发送给客户端，并且服务器端保存着唯一的私钥
3. 客户端用公钥对双方通信的对称秘钥进行加密，并发送给服务器端
4. 服务器利用自己唯一的私钥对客户端发来的对称秘钥进行解密，
5. 进行数据传输，服务器和客户端双方用公有的相同的对称秘钥对数据进行加密解密，可以保证在 数据收发过程中的安全，即是第三方获得数据包，也无法对其进行加密，解密和篡改。

因为数字签名、摘要是证书防伪非常关键的武器。 “摘要”就是对传输的内容，通过hash算法计算出一段固定长度的串。然后，在通过CA的私钥对这段摘要进行加密，加密后得到的结果就是“数字签名”

SSL/TLS协议的基本思路是采用公钥加密法，也就是说，客户端先向服务器端索要公钥，然后用公钥加 密信息，服务器收到密文后，用自己的私钥解密。

### 22、如何保证公钥不被篡改？

将公钥放在数字证书中。只要证书是可信的，公钥就是可信的。公钥加密计算量太大，如何减少耗用的时间？

每一次对话（session），客户端和服务器端都生成一个"对话密钥"（session key），用它来加密信息。由于"对话密钥"是对称加密，所以运算速度非常快，而服务器公钥只用于加密"对话密钥"本身，这 样就减少了加密运算的消耗时间。

1. 客户端向服务器端索要并验证公钥。
2. 双方协商生成"对话密钥"。
3. 双方采用"对话密钥"进行加密通信。上面过程的前两步，又称为"握手阶段"（handshake）。

### 23、HTTP请求和响应报文有哪些主要字段？

请求报文

简单来说：

请求行：Request Line

请求头：Request Headers

请求体：Request Body

响应报文

简单来说：

状态行：Status Line

响应头：Response Headers

响应体：Response Body

### 24、Cookie是什么？

HTTP 协议是**无状态**的，主要是为了让 HTTP 协议尽可能简单，使得它能够处理大量事务，HTTP/1.1 引入 Cookie 来保存状态信息。

Cookie 是**服务器发送到用户浏览器并保存在本地的一小块数据**，它会在浏览器之后向同一服务器再次发起请求时被携带上，用于告知服务端两个请求是否来自同一浏览器。由于之后每次请求都会需要携带Cookie 数据，因此会带来额外的性能开销（尤其是在移动环境下）。

Cookie 曾一度用于客户端数据的存储，因为当时并没有其它合适的存储办法而作为唯一的存储手段， 但现在随着现代浏览器开始支持各种各样的存储方式，Cookie 渐渐被淘汰。

新的浏览器 API 已经允许开发者直接将数据存储到本地，如使用 Web storage API（本地存储和会话存储）或 IndexedDB。

cookie 的出现是因为 HTTP 是无状态的一种协议，换句话说，服务器记不住你，可能你每刷新一次网页，就要重新输入一次账号密码进行登录。这显然是让人无法接受的，cookie 的作用就好比服务器给你贴个标签，然后你每次向服务器再发请求时，服务器就能够 cookie 认出你。

抽象地概括一下：一个 cookie 可以认为是一个「变量」，形如 name=value，存储在浏览器；一个session 可以理解为一种数据结构，多数情况是「映射」（键值对），存储在服务器上。

### 25、Cookie有什么用途？用途

会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息） 个性化设置（如用户自定义设置、主题等）

浏览器行为跟踪（如跟踪分析用户行为等）

### 26、Session知识大总结

除了可以将用户信息通过 Cookie 存储在用户浏览器中，也可以利用 Session 存储在服务器端，存储在服务器端的信息更加安全。

Session 可以存储在服务器上的文件、数据库或者内存中。也可以将 Session 存储在 Redis 这种内存型数据库中，效率会更高。

使用 Session 维护用户登录状态的过程如下：

* 1. 用户进行登录时，用户提交包含用户名和密码的表单，放入 HTTP 请求报文中；
  2. 服务器验证该用户名和密码，如果正确则把用户信息存储到 Redis 中，它在 Redis 中的 Key 称为

Session ID；

* 1. 服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie 值存入浏览器中；
  2. 客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID， 从 Redis 中取出用户信息，继续之前的业务操作。

注意：Session ID 的安全性问题，不能让它被恶意攻击者轻易获取，那么就不能产生一个容易被猜到的 Session ID 值。此外，还需要经常重新生成 Session ID。在对安全性要求极高的场景下， 例如转账等操作，除了使用 Session 管理用户状态之外，还需要对用户进行重新验证，比如重新输入密码，或者使用短信验证码等方式。

### 27、Session 的工作原理是什么？

session 的工作原理是客户端登录完成之后，服务器会创建对应的 session，session 创建完之后，会把session 的 id 发送给客户端，客户端再存储到浏览器中。这样客户端每次访问服务器时，都会带着sessionid，服务器拿到 sessionid 之后，在内存找到与之对应的 session 这样就可以正常工作了。

### 28、Cookie与Session的对比

HTTP作为无状态协议，必然需要在某种方式保持连接状态。这里简要介绍一下Cookie和Session。

Cookie

Cookie是客户端保持状态的方法。

Cookie简单的理解就是存储由服务器发至客户端并由客户端保存的一段字符串。为了保持会话， 服务器可以在响应客户端请求时将Cookie字符串放在Set-Cookie下，客户机收到Cookie之后保存 这段字符串，之后再请求时候带上Cookie就可以被识别。

除了上面提到的这些，Cookie在客户端的保存形式可以有两种，一种是会话Cookie一种是持久Cookie，会话Cookie就是将服务器返回的Cookie字符串保持在内存中，关闭浏览器之后自动销 毁，持久Cookie则是存储在客户端磁盘上，其有效时间在服务器响应头中被指定，在有效期内， 客户端再次请求服务器时都可以直接从本地取出。需要说明的是，存储在磁盘中的Cookie是可以 被多个浏览器代理所共享的。

Session

Session是服务器保持状态的方法。

首先需要明确的是，Session保存在服务器上，可以保存在数据库、文件或内存中，每个用户有独 立的Session用户在客户端上记录用户的操作。我们可以理解为每个用户有一个独一无二的Session ID作为Session文件的Hash键，通过这个值可以锁定具体的Session结构的数据，这个Session结构中存储了用户操作行为。

当服务器需要识别客户端时就需要结合Cookie了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用Cookie来实现Session跟踪的，第一次创建Session 的时候，服务端会在HTTP协议中告诉客户端，需要在Cookie里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了。如果客户端的浏览器禁用了Cookie，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如sid=xxxxx这样的 参数，服务端据此来识别用户，这样就可以帮用户完成诸如用户名等信息自动填入的操作了。

### 29、SQL注入攻击了解吗？

攻击者在HTTP请求中注入恶意的SQL代码，服务器使用参数构建数据库SQL命令时，恶意SQL被一起构 造，并在数据库中执行。

用户登录，输入用户名 lianggzone，密码 ‘ or ‘1’=’1 ，如果此时使用参数构造的方式，就会出现

select \* from user where name = ‘lianggzone’ and password = ‘’ or ‘1’=‘1’

不管用户名和密码是什么内容，使查询出来的用户列表不为空。如何防范SQL注入攻击使用预编译的

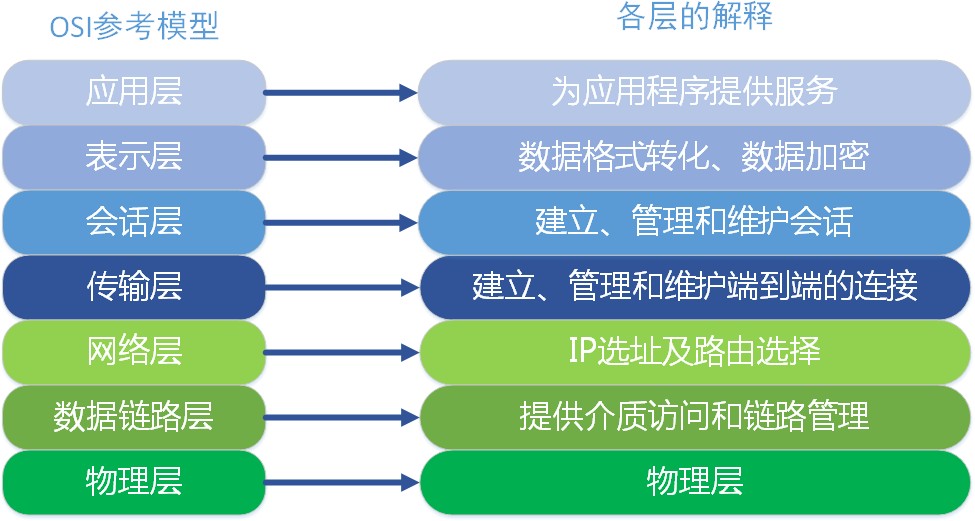
PrepareStatement是必须的，但是一般我们会从两个方面同时入手。Web端

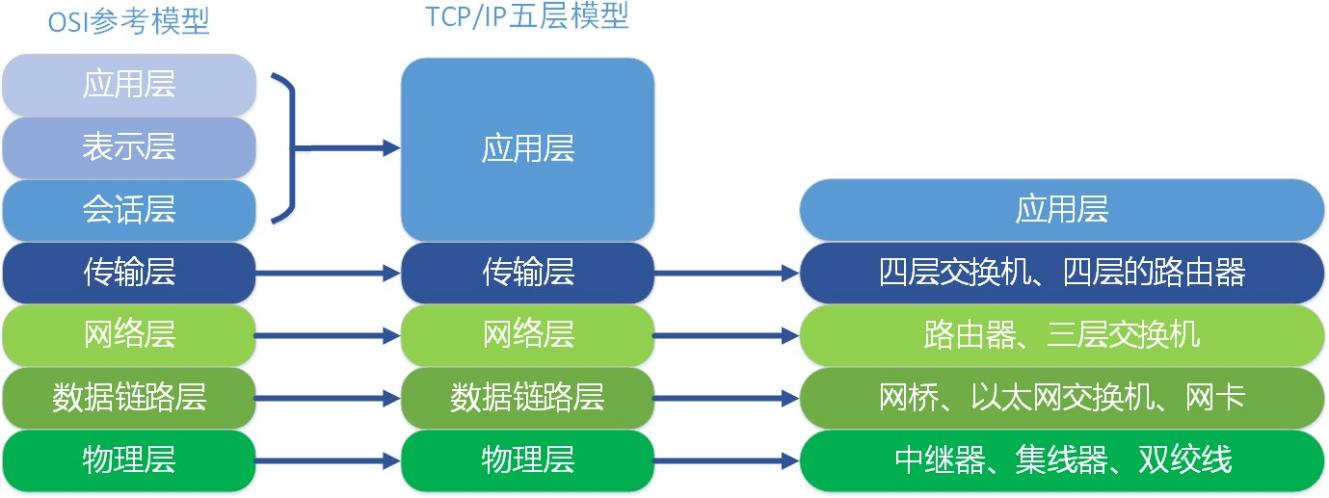
1. 有效性检验。
2. 限制字符串输入的长度。服务端

1）不用拼接SQL字符串。 2）使用预编译的PrepareStatement。

1. 有效性检验。(为什么服务端还要做有效性检验？第一准则，外部都是不可信的，防止攻击者绕过Web端请求)
2. 过滤SQL需要的参数中的特殊字符。比如单引号、双引号。

### 30、网络的七层模型与各自的功能（图片版）





**31、什么是RARP？工作原理**

概括： 反向地址转换协议，网络层协议，RARP与ARP工作方式相反。 RARP使只知道自己硬件地址的主机能够知道其IP地址。RARP发出要反向解释的物理地址并希望返回其IP地址，应答包括能够提供所需 信息的RARP服务器发出的IP地址。

原理：

1. 网络上的每台设备都会有一个独一无二的硬件地址，通常是由设备厂商分配的MAC地址。主机从网卡 上读取MAC地址，然后在网络上发送一个RARP请求的广播数据包，请求RARP服务器回复该主机的IP地 址。
2. RARP服务器收到了RARP请求数据包，为其分配IP地址，并将RARP回应发送给主机。(3)PC1收到RARP回应后，就使用得到的IP地址进行通讯。

### 32、端口有效范围是多少到多少？

0-1023为知名端口号，比如其中HTTP是80，FTP是20（数据端口）、21（控制端口）

UDP和TCP报头使用两个字节存放端口号，所以端口号的有效范围是从0到65535。动态端口的范围是从1024到65535

### 33、为何需要把 TCP/IP 协议栈分成 5 层（或7层）？开放式回答。

答：ARPANET 的研制经验表明，对于复杂的计算机网络协议，其结构应该是层次式的。分层的好处：

①隔层之间是独立的

②灵活性好

③结构上可以分隔开

④易于实现和维护

⑤能促进标准化工作。

### 34、DNS查询方式有哪些？

递归解析

当局部DNS服务器自己不能回答客户机的DNS查询时，它就需要向其他DNS服务器进行查询。此时有两 种方式。**局部DNS服务器自己负责向其他DNS服务器进行查询，一般是先向该域名的根域服务器查询， 再由根域名服务器一级级向下查询**。最后得到的查询结果返回给局部DNS服务器，再由局部DNS服务器 返回给客户端。

迭代解析

当局部DNS服务器自己不能回答客户机的DNS查询时，也可以通过迭代查询的方式进行解析。局部DNS 服务器不是自己向其他DNS服务器进行查询，**而是把能解析该域名的其他DNS服务器的IP地址返回给客 户端DNS程序**，客户端DNS程序再继续向这些DNS服务器进行查询，直到得到查询结果为止。也就是 说，迭代解析只是帮你找到相关的服务器而已，而不会帮你去查。比如说：baidu.com的服务器ip地址 在192.168.4.5这里，你自己去查吧，本人比较忙，只能帮你到这里了。

### 35、HTTP中缓存的私有和共有字段？知道吗？

private 指令规定了将资源作为私有缓存，只能被单独用户使用，一般存储在用户浏览器中。

|  |
| --- |
| 1 Cache-Control: private |

public 指令规定了将资源作为公共缓存，可以被多个用户使用，一般存储在代理服务器中。

|  |
| --- |
| 1 Cache-Control: public |

### 36、GET 方法参数写法是固定的吗？

在约定中，我们的参数是写在 ? 后面，用 & 分割。

我们知道，解析报文的过程是通过获取 TCP 数据，用正则等工具从数据中获取 Header 和 Body，从而提取参数。

比如header请求头中添加token，来验证用户是否登录等权限问题。

也就是说，我们可以自己约定参数的写法，只要服务端能够解释出来就行，万变不离其宗。

### 37、GET 方法的长度限制是怎么回事？

网络上都会提到浏览器地址栏输入的参数是有限的。

首先说明一点，HTTP 协议没有 Body 和 URL 的长度限制，对 URL 限制的大多是浏览器和服务器的原因。

浏览器原因就不说了，服务器是因为处理长 URL 要消耗比较多的资源，为了性能和安全（防止恶意构造长 URL 来攻击）考虑，会给 URL 长度加限制。

### 38、POST 方法比 GET 方法安全？

有人说POST 比 GET 安全，因为数据在地址栏上不可见。

然而，从传输的角度来说，他们都是不安全的，因为 HTTP 在网络上是明文传输的，只要在网络节点上捉包，就能完整地获取数据报文。

要想安全传输，就只有加密，也就是 HTTPS。

### 39、POST 方法会产生两个 TCP 数据包？你了解吗？

有些文章中提到，POST 会将 header 和 body 分开发送，先发送 header，服务端返回 100 状态码再发送 body。

HTTP 协议中没有明确说明 POST 会产生两个 TCP 数据包，而且实际测试(Chrome)发现，header 和

body 不会分开发送。

所以，header 和 body 分开发送是部分浏览器或框架的请求方法，不属于 post 必然行为。

### 40、Session是什么？

除了可以将用户信息通过 Cookie 存储在用户浏览器中，也可以利用 Session 存储在服务器端，存储在服务器端的信息更加安全。

Session 可以存储在服务器上的文件、数据库或者内存中。也可以将 Session 存储在 Redis 这种内存型数据库中，效率会更高。

### 41、使用 Session 的过程是怎样的？

过程如下：

用户进行登录时，用户提交包含用户名和密码的表单，放入 HTTP 请求报文中；

服务器验证该用户名和密码，如果正确则把用户信息存储到 Redis 中，它在 Redis 中的 Key 称为

Session ID；

服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie 值存入浏览器中；

客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID， 从 Redis 中取出用户信息，继续之前的业务操作。

**注意**：Session ID 的安全性问题，不能让它被恶意攻击者轻易获取，那么就不能产生一个容易被猜到的Session ID 值。此外，还需要经常重新生成 Session ID。在对安全性要求极高的场景下，例如转账等操作，除了使用 Session 管理用户状态之外，还需要对用户进行重新验证，比如重新输入密码，或者使用短信验证码等方式。

### 42、Session和cookie应该如何去选择（适用场景）？

Cookie 只能存储 ASCII 码字符串，而 Session 则可以存储任何类型的数据，因此在考虑数据复杂性时首选 Session；

Cookie 存储在浏览器中，容易被恶意查看。如果非要将一些隐私数据存在 Cookie 中，可以将

Cookie 值进行加密，然后在服务器进行解密；

对于大型网站，如果用户所有的信息都存储在 Session 中，那么开销是非常大的，因此不建议将所有的用户信息都存储到 Session 中。

### 43、Cookies和Session区别是什么？

Cookie和Session都是客户端与服务器之间保持状态的解决方案 1，存储的位置不同，cookie：存放在客户端，session：存放在服务端。Session存储的数据比较安全2，存储的数据类型不同

两者都是key-value的结构，但针对value的类型是有差异的cookie：value只能是字符串类型，session：value是Object类型3，存储的数据大小限制不同

cookie：大小受浏览器的限制，很多是是4K的大小， session：理论上受当前内存的限制， 4，生命周期的控制

cookie的生命周期当浏览器关闭的时候，就消亡了

(1)cookie的生命周期是累计的，从创建时，就开始计时，20分钟后，cookie生命周期结束， (2)session的生命周期是间隔的，从创建时，开始计时如在20分钟，没有访问session，那么session生 命周期被销毁

### 44、DDos 攻击了解吗？

客户端向服务端发送请求链接数据包，服务端向客户端发送确认数据包，客户端不向服务端发送确认数 据包，服务器一直等待来自客户端的确认

没有彻底根治的办法，除非不使用TCP DDos 预防：

1. 限制同时打开SYN半链接的数目2）缩短SYN半链接的Time out 时间3）关闭不必要的服务

### 45、MTU和MSS分别是什么？

MTU：maximum transmission unit，最大传输单元，由硬件规定，如以太网的MTU为1500字节。

MSS：maximum segment size，最大分节大小，为TCP数据包每次传输的最大数据分段大小，一般由发送端向对端TCP通知对端在每个分节中能发送的最大TCP数据。MSS值为MTU值减去IPv4 Header（20 Byte）和TCP header（20 Byte）得到。

### 46、HTTP中有个缓存机制，但如何保证缓存是最新的呢？（缓存过期机制）

max-age 指令出现在请求报文，并且缓存资源的缓存时间小于该指令指定的时间，那么就能接受该缓存。

max-age 指令出现在响应报文，表示缓存资源在缓存服务器中保存的时间。

|  |
| --- |
| 1 Cache-Control: max-age=31536000 |

Expires 首部字段也可以用于告知缓存服务器该资源什么时候会过期。

|  |
| --- |
| 1 Expires: Wed, 04 Jul 2012 08:26:05 GMT |

在 HTTP/1.1 中，会优先处理 max-age 指令； 在 HTTP/1.0 中，max-age 指令会被忽略掉。

### 47、TCP头部中有哪些信息？

序号（32bit）：传输方向上字节流的字节编号。初始时序号会被设置一个随机的初始值（ISN）， 之后每次发送数据时，序号值 = ISN + 数据在整个字节流中的偏移。假设A -> B且ISN = 1024，第一段数据512字节已经到B，则第二段数据发送时序号为1024 + 512。用于解决网络包乱序问题。

确认号（32bit）：接收方对发送方TCP报文段的响应，其值是收到的序号值 + 1。首部长（4bit）：标识首部有多少个4字节 \* 首部长，最大为15，即60字节。

标志位（6bit）：

URG：标志紧急指针是否有效。 ACK：标志确认号是否有效（确认报文段）。用于解决丢包问题。PSH：提示接收端立即从缓冲读走数据。 RST：表示要求对方重新建立连接（复位报文段）。 SYN：表示请求建立一个连接（连接报文段）。 FIN：表示关闭连接（断开报文段）。

窗口（16bit）：接收窗口。用于告知对方（发送方）本方的缓冲还能接收多少字节数据。用于解 决流控。

校验和（16bit）：接收端用CRC检验整个报文段有无损坏。

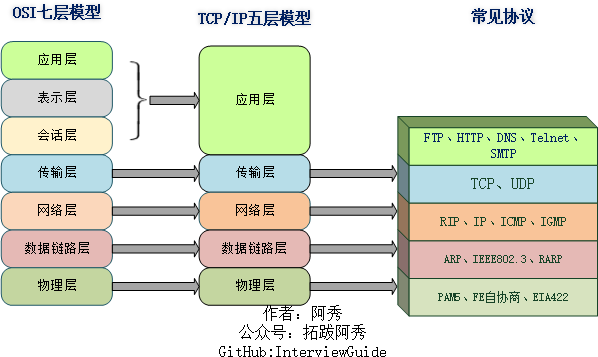
### 48、常见TCP的连接状态有哪些？

CLOSED： 初 始 状 态 。 LISTEN：服务器处于监听状态。

SYN\_SEND：客户端socket执行CONNECT连接，发送SYN包，进入此状态。 SYN\_RECV：服务端收到SYN包并发送服务端SYN包，进入此状态。 ESTABLISH：表示连接建立。客户端发送了最后一个ACK包后进入此状态，服务端接收到ACK包后 进入此状态。

FIN\_WAIT\_1：终止连接的一方（通常是客户机）发送了FIN报文后进入。等待对方FIN。CLOSE\_WAIT：（假设服务器）接收到客户机FIN包之后等待关闭的阶段。在接收到对方的FIN包之 后，自然是需要立即回复ACK包的，表示已经知道断开请求。但是本方是否立即断开连接（发送 FIN包）取决于是否还有数据需要发送给客户端，若有，则在发送FIN包之前均为此状态。FIN\_WAIT\_2：此时是半连接状态，即有一方要求关闭连接，等待另一方关闭。客户端接收到服务 器的ACK包，但并没有立即接收到服务端的FIN包，进入FIN\_WAIT\_2状态。 LAST\_ACK：服务端发动最后的FIN包，等待最后的客户端ACK响应，进入此状态。 TIME\_WAIT：客户端收到服务端的FIN包，并立即发出ACK包做最后的确认，在此之后的2MSL时 间称为TIME\_WAIT状态。

### 49、网络的七层/五层模型主要的协议有哪些？



**50、TCP是什么？**

TCP（Transmission Control Protocol 传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议。

### 51、TCP头部报文字段介绍几个？各自的功能？

source port 和 destination port

两者分别为「源端口号」和「目的端口号」。源端口号就是指本地端口，目的端口就是远程端 口。

可以这么理解，我们有很多软件，每个软件都对应一个端口，假如，你想和我数据交互，咱们得互相知 道你我的端口号。

再来一个很官方的：

扩展：应用程序的端口号和应用程序所在主机的 IP 地址统称为 socket（套接字），IP:端口号, 在互联网上 socket 唯一标识每一个应用程序，源端口+源IP+目的端口+目的IP称为”套接字对“，一对套接字就是一个连接，一个客户端与服务器之间的连接。

Sequence Number

称为「序列号」。用于 TCP 通信过程中某一传输方向上字节流的每个字节的编号，为了确保数据通信的有序性，避免网络中乱序的问题。接收端根据这个编号进行确认，保证分割的数据段在原 始数据包的位置。初始序列号由自己定，而后绪的序列号由对端的 ACK 决定：SN\_x = ACK\_y (x 的序列号 = y 发给 x 的 ACK)。

说白了，类似于身份证一样，而且还得发送此时此刻的所在的位置，就相当于身份证上的地址一样。

Acknowledge Number

称为「确认序列号」。确认序列号是接收确认端所期望收到的下一序列号。确认序号应当是上次 已成功收到数据字节序号加1，只有当标志位中的 ACK 标志为 1 时该确认序列号的字段才有效。主要用来解决不丢包的问题。

TCP Flag

TCP 首部中有 6 个标志比特，它们中的多个可同时被设置为 1 ，主要是用于操控 TCP 的状态机的， 依次为 URG，ACK，PSH，RST，SYN，FIN 。

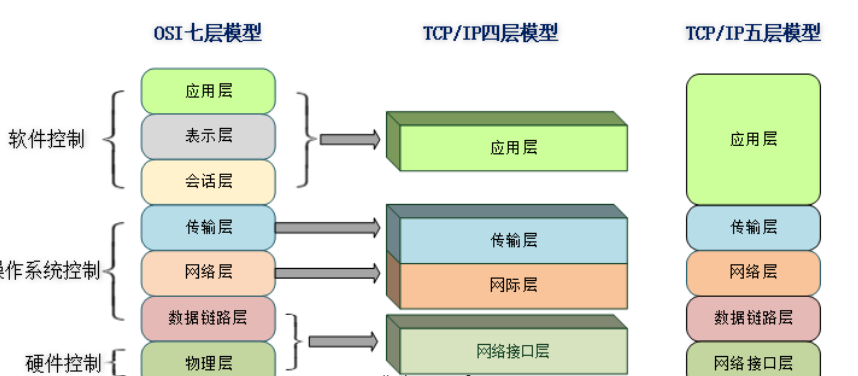
当然只介绍三个：

* 1. **ACK**：这个标识可以理解为发送端发送数据到接收端，发送的时候 ACK 为 0，标识接收端还未应答，一旦接收端接收数据之后，就将 ACK 置为 1，发送端接收到之后，就知道了接收端已经接收了数据。
  2. **SYN**：表示「同步序列号」，是 TCP 握手的发送的第一个数据包。用来建立 TCP 的连接。SYN 标志位和 ACK 标志位搭配使用，当连接请求的时候，SYN=1，ACK=0连接被响应的时候，SYN=1， ACK=1；这个标志的数据包经常被用来进行端口扫描。扫描者发送一个只有 SYN 的数据包，如果对方主机响应了一个数据包回来 ，就表明这台主机存在这个端口。
  3. **FIN**：表示发送端已经达到数据末尾，也就是说双方的数据传送完成，没有数据可以传送了，发送FIN标志位的 TCP 数据包后，连接将被断开。这个标志的数据包也经常被用于进行端口扫描。发送端只剩最后的一段数据了，同时要告诉接收端后边没有数据可以接受了，所以用FIN标识一下，接 收端看到这个FIN之后，哦！这是接受的最后的数据，接受完就关闭了；**TCP四次分手必然问**。

Window size

称为滑动窗口大小。所说的滑动窗口，用来进行流量控制。

### 52、OSI 的七层模型的主要功能？

物理层：利用传输介质为数据链路层提供物理连接，实现比特流的透明传输。 数据链路层：接收来自物理层的位流形式的数据，并封装成帧，传送到上一层

网络层：将网络地址翻译成对应的物理地址，并通过路由选择算法为分组通过通信子网选择最适当的路 径。

传输层：在源端与目的端之间提供可靠的透明数据传输

会话层：负责在网络中的两节点之间建立、维持和终止通信

表示层：处理用户信息的表示问题，数据的编码，压缩和解压缩，数据的加密和解密 应用层：为用户的应用进程提供网络通信服务

### 53、应用层常见协议知道多少？了解几个？

|  |  |  |  |
| --- | --- | --- | --- |
| **协议** | **名称** | **默认端口** | **底层协议** |
| HTTP | 超文本传输协议 | 80 | TCP |
| HTTPS | 超文本传输安全协议 | 443 | TCP |
| Telnet | 远程登录服务的标准协议 | 23 | TCP |
| FTP | 文件传输协议 | 20传输和21连接 | TCP |
| TFTP | 简单文件传输协议 | 21 | UDP |
| SMTP | 简单邮件传输协议（发送用） | 25 | TCP |
| POP | 邮局协议（接收用） | 110 | TCP |
| DNS | 域名解析服务 | 53 | 服务器间进行域传输的时候用  TCP  客户端查询DNS服务器时用  UDP |

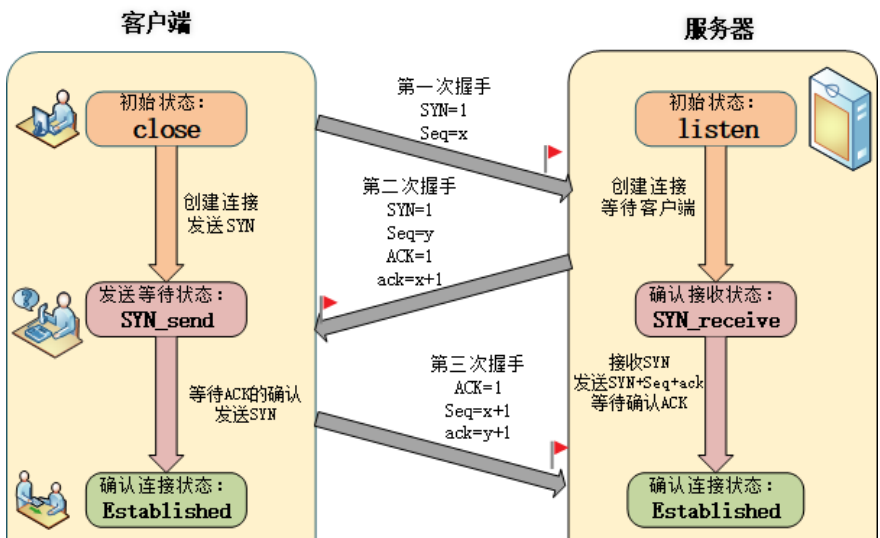
**54、浏览器在与服务器建立了一个 TCP 连接后是否会在一个 HTTP 请求完成后断开？什么情况下会断开？**

在 HTTP/1.0 中，一个服务器在发送完一个 HTTP 响应后，会断开 TCP 链接。但是这样每次请求都会重新建立和断开 TCP 连接，代价过大。所以虽然标准中没有设定，**某些服务器对 Connection: keep- alive 的 Header 进行了支持**。意思是说，完成这个 HTTP 请求之后，不要断开 HTTP 请求使用的 TCP 连接。这样的好处是连接可以被重新使用，之后发送 HTTP 请求的时候不需要重新建立 TCP 连接，以及如果维持连接，那么 SSL 的开销也可以避免。

**持久连接**：既然维持 TCP 连接好处这么多，HTTP/1.1 就把 Connection 头写进标准，并且默认开启持久连接，除非请求中写明 Connection: close，那么浏览器和服务器之间是会维持一段时间的 TCP 连接，不会一个请求结束就断掉。

默认情况下建立 TCP 连接不会断开，只有在请求报头中声明 Connection: close 才会在请求完成后关闭连接。

### 55、三次握手相关内容



三次握手（Three-way Handshake）其实就是指建立一个TCP连接时，需要客户端和服务器总共发送3 个包。进行三次握手的主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化 序列号为后面的可靠性传送做准备。实质上其实就是连接服务器指定端口，建立TCP连接，并同步连接 双方的序列号和确认号，交换 TCP窗口大小 信息。

第一种回答

刚开始客户端处于 Closed 的状态，服务端处于 Listen 状态，进行三次握手：

第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 ISN(c)。此时客户

端处于 状态。

SYN\_SEND

首部的同步位SYN=1，初始序号seq=x，SYN=1的报文段不能携带数据，但要消耗掉一个序号。

第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN(s)。同时会把客户端的 ISN + 1 作为ACK 的值，表示自己已经收到了客

户端的 SYN，此时服务器处于 的状态。

SYN\_RCVD

在确认报文段中SYN=1，ACK=1，确认号ack=x+1，初始序号seq=y。

第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN

+ 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 状态。

ESTABLISHED

服务器收到 ACK 报文之后，也处于 状态，此时，双方已建立起了连接。

ESTABLISHED

确认报文段ACK=1，确认号ack=y+1，序号seq=x+1（初始为seq=x，第二个报文段所以要+1），

ACK报文段可以携带数据，不携带数据则不消耗序号。

发送第一个SYN的一端将执行主动打开（active open），接收这个SYN并发回下一个SYN的另一端执行被动打开（passive open）。

在socket编程中，客户端执行connect()时，将触发三次握手。

第二种回答

**初始状态**：客户端处于 closed(关闭) 状态，服务器处于 listen(监听) 状态。

**第一次握手**：客户端发送请求报文将 SYN = 1 同步序列号和初始化序列号 seq = x 发送给服务端，发送完之后客户端处于 SYN\_Send 状态。（验证了客户端的发送能力和服务端的接收能力） **第二次握手**：服务端受到 SYN 请求报文之后，如果同意连接，会以自己的同步序列号 SYN(服务端) = 1 、初始化序列号 seq = y 和确认序列号（期望下次收到的数据包） ack = x+ 1 以及确认号 ACK = 1 报文作为应答，服务器为 SYN\_Receive 状态。（问题来了，两次握手之后，站在客户端角度上思考：我发送和接收都ok，服务端的发送和接收也都ok。但是站在服务端的角度思

考：哎呀，我服务端接收ok，但是我不清楚我的发送ok不ok呀，而且我还不知道你接受能力如何 呢？所以老哥，你需要给我三次握手来传个话告诉我一声。你要是不告诉我，万一我认为你跑了， 然后我可能出于安全性的考虑继续给你发一次，看看你回不回我。）

**第三次握手**： 客户端接收到服务端的 SYN + ACK 之后，知道可以下次可以发送了下一序列的数据

包了，然后发送同步序列号 ack = y + 1 和数据包的序列号 seq = x + 1 以及确认号 ACK = 1

确认包作为应答，客户端转为 established 状态。（分别站在双方的角度上思考，各自ok）

### 56、为什么需要三次握手，两次不行吗？

弄清这个问题，我们需要先弄明白三次握手的目的是什么，能不能只用两次握手来达到同样的目的。

第一次握手：客户端发送网络包，服务端收到了。 这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。

第二次握手：服务端发包，客户端收到了。 这样客户端就能得出结论：服务端的接收、发送能

力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常。 第三次握手：客户端发包，服务端收到了。 这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常。

因此，需要三次握手才能确认双方的接收与发送能力是否正常。试想如果是用两次握手，则会出现下面这种情况：

如客户端发出连接请求，但因连接请求报文丢失而未收到确认，于是客户端再重传一次连接请 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，客户端共发出了两个连接 请求报文段，其中第一个丢失，第二个到达了服务端，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服务端**，此时服务端误认为客户端又 发出一次新的连接请求，于是就向客户端发出确认报文段，同意建立连接，不采用三次握手，只 要服务端发出确认，就建立新的连接了，此时客户端忽略服务端发来的确认，也不发送数据，则 服务端一致等待客户端发送数据，浪费资源。

### 57、什么是半连接队列？

服务器第一次收到客户端的 SYN 之后，就会处于 SYN\_RCVD 状态，此时双方还没有完全建立其连接， 服务器会把此种状态下请求连接放在一个**队列**里，我们把这种队列称之为**半连接队列**。

当然还有一个**全连接队列**，就是已经完成三次握手，建立起连接的就会放在全连接队列中。如果队列满 了就有可能会出现丢包现象。

这里在补充一点关于**SYN-ACK 重传次数**的问题： 服务器发送完SYN-ACK包，如果未收到客户确认包， 服务器进行首次重传，等待一段时间仍未收到客户确认包，进行第二次重传。如果重传次数超过系统规 定的最大重传次数，系统将该连接信息从半连接队列中删除。 注意，每次重传等待的时间不一定相同， 一般会是指数增长，例如间隔时间为 1s，2s，4s，8s......

### 58、 ISN(Initial Sequence Number)是固定的吗？

当一端为建立连接而发送它的SYN时，它为连接选择一个初始序号。ISN随时间而变化，因此每个连接 都将具有不同的ISN。ISN可以看作是一个32比特的计数器，每4ms加1 。这样选择序号的目的在于防止在网络中被延迟的分组在以后又被传送，而导致某个连接的一方对它做错误的解释。

三次握手的其中一个重要功能是客户端和服务端交换 ISN(Initial Sequence Number)，以便让对方知道接下来接收数据的时候如何按序列号组装数据。如果 ISN 是固定的，攻击者很容易猜出后续的确认号，因此 ISN 是动态生成的。

**59、 三次握手过程中可以携带数据吗？**

其实第三次握手的时候，是可以携带数据的。但是，**第一次、第二次握手不可以携带数据**

为什么这样呢?大家可以想一个问题，假如第一次握手可以携带数据的话，如果有人要恶意攻击服务 器，那他每次都在第一次握手中的 SYN 报文中放入大量的数据。因为攻击者根本就不理服务器的接收、发送能力是否正常，然后疯狂着重复发 SYN 报文的话，这会让服务器花费很多时间、内存空间来接收这些报文。

也就是说，第一次握手不可以放数据，其中一个简单的原因就是会让服务器更加容易受到攻击了。而对 于第三次的话，此时客户端已经处于 ESTABLISHED 状态。对于客户端来说，他已经建立起连接了，并且也已经知道服务器的接收、发送能力是正常的了，所以能携带数据也没啥毛病。

**60、SYN攻击是什么？**

**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。SYN 攻击是一种典型的 DoS/DDoS 攻击。

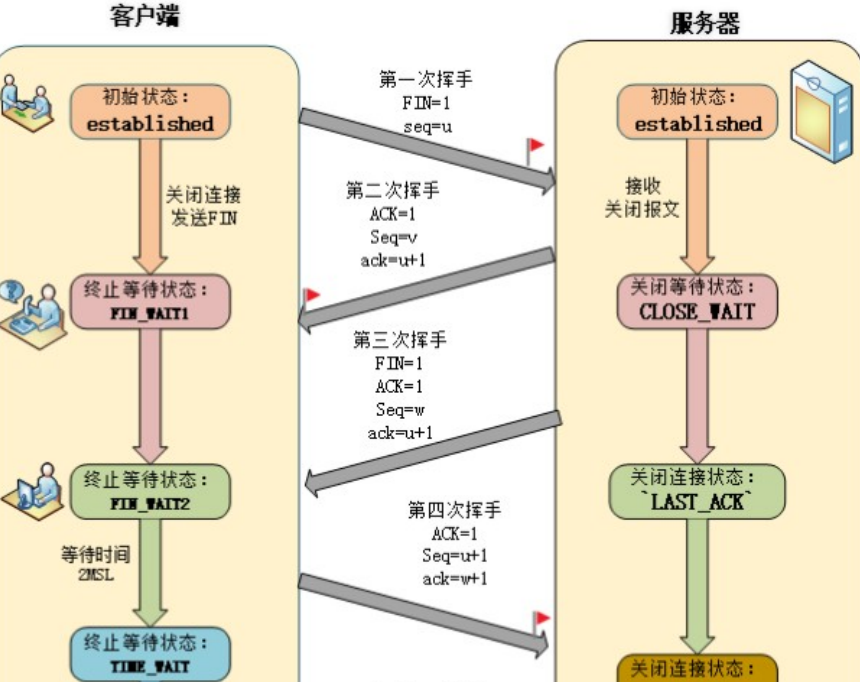
检测 SYN 攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击。在 Linux/Unix 上可以使用系统自带的 netstats 命令来检测 SYN 攻击。

|  |
| --- |
| 1. netstat -n -p TCP | grep SYN\_RECV 2. 复制代码 |

常见的防御 SYN 攻击的方法有如下几种： 缩短超时（SYN Timeout）时间

增加最大半连接数过 滤 网 关 防 护 SYN cookies技术

### 61、 四次挥手相关内容



建立一个连接需要三次握手，而终止一个连接要经过四次挥手（也有将四次挥手叫做四次握手的）。这 由TCP的**半关闭**（half-close）造成的。所谓的半关闭，其实就是TCP提供了连接的一端在结束它的发送 后还能接收来自另一端数据的能力。

TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，客户端或服务器均可主动发起挥手动作。

第一种回答

刚开始双方都处于 ESTABLISHED 状态，假如是客户端先发起关闭请求。四次挥手的过程如下： 第一次挥手：客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于

FIN\_WAIT1

状态。 即发出**连接释放报文段**（FIN=1，序号seq=u），并停止再发送数据，主动关闭TCP连接， 进入FIN\_WAIT1（终止等待1）状态，等待服务端的确认。

第二次挥手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 +1 作为 ACK 报文

的序列号值，表明已经收到客户端的报文了，此时服务端处于 状态。 即服务端收到

CLOSE\_WAIT

连接释放报文段后即发出**确认报文段**（ACK=1，确认号ack=u+1，序号seq=v），服务端进入CLOSE\_WAIT（关闭等待）状态，此时的TCP处于半关闭状态，客户端到服务端的连接释放。客户 端收到服务端的确认后，进入FIN\_WAIT2（终止等待2）状态，等待服务端发出的连接释放报文 段。

第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定

一个序列号。此时服务端处于 的状态。 即服务端没有要向客户端发出的数据，服务端

LAST\_ACK

发出**连接释放报文段**（FIN=1，ACK=1，序号seq=w，确认号ack=u+1），服务端进入

LAST\_ACK（最后确认）状态，等待客户端的确认。

第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 +1

作为自己 ACK 报文的序列号值，此时客户端处于 状态。需要过一阵子以确保服务端

TIME\_WAIT

收到自己的 ACK 报文之后才会进入 CLOSED 状态，服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。 即客户端收到服务端的连接释放报文段后，对此发出**确认报文段**

（ACK=1，seq=u+1，ack=w+1），客户端进入TIME\_WAIT（时间等待）状态。此时TCP未释放 掉，需要经过时间等待计时器设置的时间2MSL后，客户端才进入CLOSED状态。

收到一个FIN只意味着在这一方向上没有数据流动。**客户端执行主动关闭并进入TIME\_WAIT是正常的， 服务端通常执行被动关闭，不会进入TIME\_WAIT状态。**

在socket编程中，任何一方执行close()操作即可产生挥手操作。

第二种回答

**初始化状态**：客户端和服务端都在连接状态，接下来开始进行四次分手断开连接操作。 **第一次分手**：第一次分手无论是客户端还是服务端都可以发起，因为 TCP 是全双工的。

假如客户端发送的数据已经发送完毕，发送FIN = 1 **告诉服务端，客户端所有数据已经全发完了**， **服务端你可以关闭接收了**，但是如果你们服务端有数据要发给客户端，客户端照样可以接收的。 此时客户端处于FIN = 1等待服务端确认释放连接状态。

**第二次分手**：服务端接收到客户端的释放请求连接之后，**知道客户端没有数据要发给自己了**，**然后 服务端发送ACK = 1告诉客户端收到你发给我的信息**，此时服务端处于 CLOSE\_WAIT 等待关闭状态。（服务端先回应给客户端一声，我知道了，但服务端的发送数据能力即将等待关闭，于是接下 来第三次就来了。）

**第三次分手**：此时服务端向客户端把所有的数据发送完了，然后发送一个FIN = 1，**用于告诉客户端，服务端的所有数据发送完毕**，**客户端你也可以关闭接收数据连接了**。此时服务端状态处于LAST\_ACK状态，来等待确认客户端是否收到了自己的请求。（服务端等客户端回复是否收到呢， 不收到的话，服务端不知道客户端是不是挂掉了还是咋回事呢，所以服务端不敢关闭自己的接收能 力，于是第四次就来了。）

**第四次分手**：此时如果客户端收到了服务端发送完的信息之后，就发送ACK = 1，告诉服务端，客户端已经收到了你的信息。**有一个 2 MSL 的延迟等待**。

### 62、挥手为什么需要四次？

第一种回答

因为当服务端收到客户端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中**ACK报文是用来应 答的，SYN报文是用来同步的**。但是关闭连接时，当服务端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉客户端，"你发的FIN报文我收到了"。只有等到我服务端 所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四次挥手。

第二种回答

任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没 有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了TCP连接。举个例子：A 和 B 打电话，通话即将结束后，A 说“我没啥要说的了”，B回答“我知道了”，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话，于是 B 可能又巴拉巴拉说了一通，最后 B 说“我说完了”，A 回答

“知道了”，这样通话才算结束。

### 63、2MSL等待状态？

TIME\_WAIT状态也成为2MSL等待状态。每个具体TCP实现必须选择一个报文段最大生存时间MSL（Maximum Segment Lifetime），它是任何报文段被丢弃前在网络内的最长时间。这个时间是有限的，因为TCP报文段以IP数据报在网络内传输，而IP数据报则有限制其生存时间的TTL字段。

对一个具体实现所给定的MSL值，处理的原则是：当TCP执行一个主动关闭，并发回最后一个ACK，该 连接必须在TIME\_WAIT状态停留的时间为2倍的MSL。这样可让TCP再次发送最后的ACK以防这个ACK丢 失（另一端超时并重发最后的FIN）。

这种2MSL等待的另一个结果是这个TCP连接在2MSL等待期间，定义这个连接的插口（客户的IP地址和 端口号，服务器的IP地址和端口号）不能再被使用。这个连接只能在2MSL结束后才能再被使用。

### 64、四次挥手释放连接时，等待2MSL的意义?

**MSL**是Maximum Segment Lifetime的英文缩写，可译为“最长报文段寿命”，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

为了保证客户端发送的最后一个ACK报文段能够到达服务器。因为这个ACK有可能丢失，从而导致处在LAST-ACK状态的服务器收不到对FIN-ACK的确认报文。服务器会超时重传这个FIN-ACK，接着客户端再 重传一次确认，重新启动时间等待计时器。最后客户端和服务器都能正常的关闭。假设客户端不等待2MSL，而是在发送完ACK之后直接释放关闭，一但这个ACK丢失的话，服务器就无法正常的进入关闭连 接状态。

两个理由

1. 保证客户端发送的最后一个ACK报文段能够到达服务端。 这个ACK报文段有可能丢失，使得处于LAST-ACK状态的B收不到对已发送的FIN+ACK报文段的确认，服务端超时重传FIN+ACK报文段，而 客户端能在2MSL时间内收到这个重传的FIN+ACK报文段，接着客户端重传一次确认，重新启动2MSL计时器，最后客户端和服务端都进入到CLOSED状态，若客户端在TIME-WAIT状态不等待一 段时间，而是发送完ACK报文段后立即释放连接，则无法收到服务端重传的FIN+ACK报文段，所以 不会再发送一次确认报文段，则服务端无法正常进入到CLOSED状态。
2. 防止“已失效的连接请求报文段”出现在本连接中。 客户端在发送完最后一个ACK报文段后，再经过2MSL，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，使下一个新的连接中 不会出现这种旧的连接请求报文段。

### 65、为什么TIME\_WAIT状态需要经过2MSL才能返回到CLOSE状态？

第一种回答

理论上，四个报文都发送完毕，就可以直接进入CLOSE状态了，但是可能网络是不可靠的，有可能最后 一个ACK丢失。所以**TIME\_WAIT状态就是用来重发可能丢失的ACK报文**。

第二种回答

对应这样一种情况，最后客户端发送的ACK = 1给服务端的**过程中丢失**了，服务端没收到，服务端怎么认为的？我已经发送完数据了，怎么客户端没回应我？是不是中途丢失了？然后服务端再次发起断开连 接的请求，一个来回就是2MSL。

客户端给服务端发送的ACK = 1丢失，**服务端等待 1MSL没收到**，**然后重新发送消息需要1MSL**。如果再次接收到服务端的消息，则**重启2MSL计时器**，**发送确认请求**。客户端只需等待2MSL，如果没有再次收 到服务端的消息，就说明服务端已经接收到自己确认消息；此时双方都关闭的连接，TCP 四次分手完毕

### 66、TCP粘包问题是什么？你会如何去解决它？

**TCP粘包**是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧 接着前一包数据的尾。

由TCP**连接复用**造成的粘包问题。

因为TCP默认会使用**Nagle算法**，此算法会导致粘包问题。

只有上一个分组得到确认，才会发送下一个分组； 收集多个小分组，在一个确认到来时一起发送。

**数据包过大**造成的粘包问题。

流量控制，**拥塞控制**也可能导致粘包。

接收方不及时接收缓冲区的包，造成多个包接收解决：

1. **Nagle算法**问题导致的，需要结合应用场景适当关闭该算法
2. 尾部标记序列。通过特殊标识符表示数据包的边界，例如\n\r，\t，或者一些隐藏字符。
3. 头部标记分步接收。在TCP报文的头部加上表示数据长度。
4. 应用层发送数据时**定长**发送。

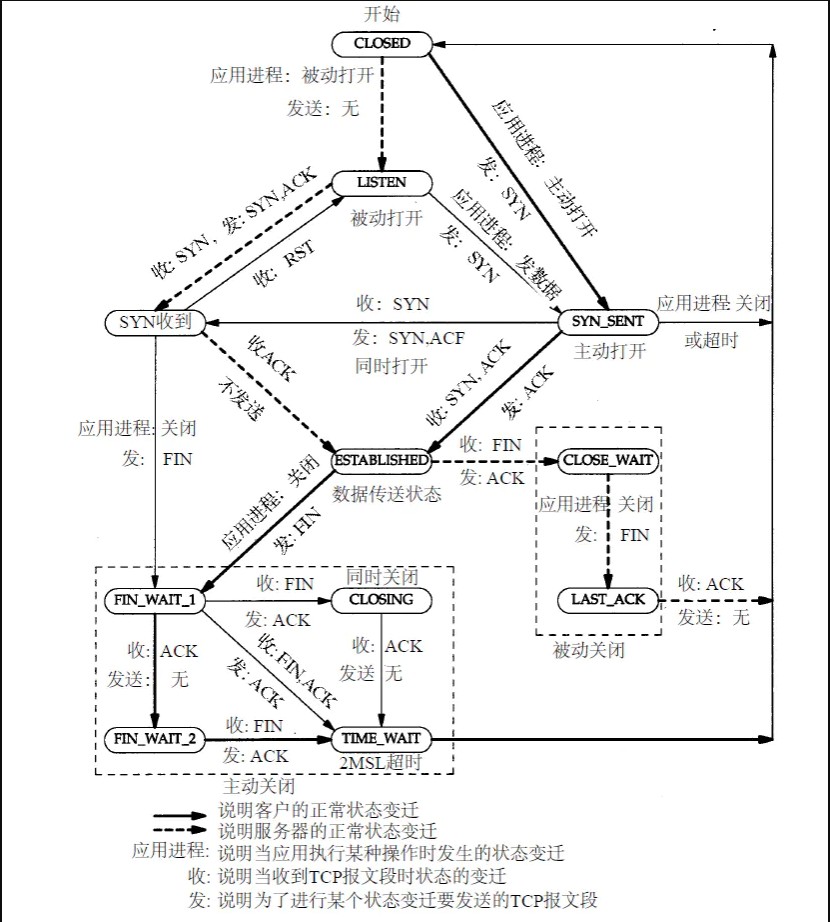
### 67、OSI七层模型中表示层和会话层功能是什么？

表示层：图像、视频编码解，数据加密。

会话层：建立会话，如session认证、断点续传。

### 68、三次握手四次挥手的变迁图

《TCP/IP详解 卷1:协议》有一张TCP状态变迁图，很具有代表性，有助于大家理解三次握手和四次挥手的状态变化。如下图所示，粗的实线箭头表示正常的客户端状态变迁，粗的虚线箭头表示正常的服务器 状态变迁。



### 69、对称密钥加密的优点缺点？

对称密钥加密（Symmetric-Key Encryption），加密和解密使用同一密钥。优点：运算速度快

缺点：无法安全地将密钥传输给通信方

### 70、非对称密钥加密你了解吗？优缺点？

非对称密钥加密，又称公开密钥加密（Public-Key Encryption），加密和解密使用不同的密钥。

公开密钥所有人都可以获得，**通信发送方获得接收方的公开密钥之后，就可以使用公开密钥进行加密**，

接收方收到通信内容后使用私有密钥解密。

非对称密钥除了用来加密，还可以用来进行签名。因为私有密钥无法被其他人获取，因此通信发送方使 用其私有密钥进行签名，通信接收方使用发送方的公开密钥对签名进行解密，就能判断这个签名是否正 确。

优点：可以更安全地将公开密钥传输给通信发送方； 缺点：运算速度慢。

### 71、HTTPS是什么

HTTPS 并不是新协议，而是让 **HTTP 先和 SSL（Secure Sockets Layer）通信，再由 SSL 和 TCP 通信，也就是说 HTTPS 使用了隧道进行通信**。通过使用 SSL，HTTPS 具有了加密（防窃听）、认证（防伪装）和完整性保护（防篡改）。

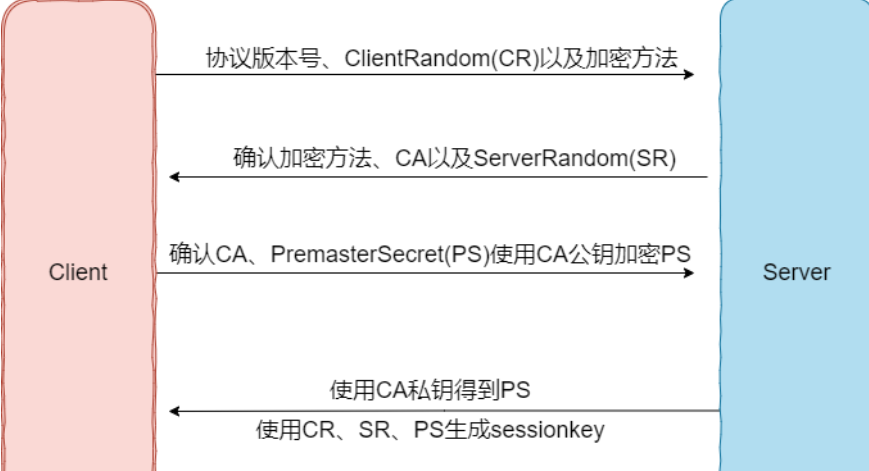
### 72、HTTP的缺点有哪些？

使用明文进行通信，内容可能会被窃听；

不验证通信方的身份，通信方的身份有可能遭遇伪装； 无法证明报文的完整性，报文有可能遭篡改。

### 73、HTTPS采用的加密方式有哪些？是对称还是非对称？

HTTPS 采用混合的加密机制，使用**非对称密钥加密用于传输对称密钥来保证传输过程的安全性**，之后使用**对称密钥加密进行通信来保证通信过程的效率**。

\

确保传输安全过程（其实就是rsa原理）：

1. Client给出协议版本号、一个客户端生成的随机数（Client random），以及客户端支持的加密方法。
2. Server确认双方使用的加密方法，并给出数字证书、以及一个服务器生成的随机数（Server random）。
3. Client确认数字证书有效，然后生成呀一个新的随机数（Premaster secret），并使用数字证书中的公钥，加密这个随机数，发给Server。
4. Server使用自己的私钥，获取Client发来的随机数（Premaster secret）。
5. Client和Server根据约定的加密方法，使用前面的三个随机数，生成”对话密钥”（session key）， 用来加密接下来的整个对话过程。

### 74、为什么有的时候刷新页面不需要重新建立 SSL 连接？

TCP 连接有的时候会被浏览器和服务端维持一段时间，TCP 不需要重新建立，SSL 自然也会用之前的。

### 75、SSL中的认证中的证书是什么？了解过吗？

通过使用 **证书** 来对通信方进行认证。

数字证书认证机构（CA，Certiﬁcate Authority）是客户端与服务器双方都可信赖的第三方机构。

服务器的运营人员向 CA 提出公开密钥的申请，CA 在判明提出申请者的身份之后，会对已申请的公开密钥做数字签名，然后分配这个已签名的公开密钥，并将该公开密钥放入公开密钥证书后绑定在一起。

进行 HTTPS 通信时，服务器会把证书发送给客户端。客户端取得其中的公开密钥之后，先使用数字签名进行验证，如果验证通过，就可以开始通信了。

**76、HTTP如何禁用缓存？如何确认缓存？** HTTP/1.1 通过 Cache-Control 首部字段来控制缓存。**禁止进行缓存**

no-store 指令规定不能对请求或响应的任何一部分进行缓存。

|  |
| --- |
| 1 Cache-Control: no-store |

强制确认缓存

no-cache 指令规定缓存服务器需要先向源服务器验证缓存资源的有效性，只有当缓存资源有效时才能使用该缓存对客户端的请求进行响应。

|  |
| --- |
| 1 Cache-Control: no-cache |

### 77、GET与POST传递数据的最大长度能够达到多少呢？

get 是通过URL提交数据，因此GET可提交的数据量就跟URL所能达到的最大长度有直接关系。

很多文章都说GET方式提交的数据最多只能是1024字节，而实际上，URL不存在参数上限的问题，

HTTP协议规范也没有对URL长度进行限制。

这个限制是特定的浏览器及服务器对它的限制，比如IE对URL长度的限制是2083字节(2K+35字节)。对 于其他浏览器，如FireFox，Netscape等，则没有长度限制，这个时候其限制取决于服务器的操作系 统；即如果url太长，服务器可能会因为安全方面的设置从而拒绝请求或者发生不完整的数据请求。

post 理论上讲是没有大小限制的，HTTP协议规范也没有进行大小限制，但实际上post所能传递的数据量大小取决于服务器的设置和内存大小。

因为我们一般post的数据量很少超过MB的，所以我们很少能感觉的到post的数据量限制，但实际中如 果你上传文件的过程中可能会发现这样一个问题，即上传个头比较大的文件到服务器时候，可能上传不 上去。

以php语言来说，查原因的时候你也许会看到有说PHP上传文件涉及到的参数PHP默认的上传有限定， 一般这个值是2MB，更改这个值需要更改php.conf的post\_max\_size这个值。这就很明白的说明了这个 问题了。

### 78、网络层常见协议？可以说一下吗？

|  |  |  |
| --- | --- | --- |
| **协议** | **名称** | **作用** |
| IP | 网际协议 | IP协议不但定义了数据传输时的基本单元和格式，还定义了数据报的递交方法和路由选择 |
| ICMP | 超文本传输安全协议 | ICMP就是一个“错误侦测与回报机制”，其目的就是让我们能够检测网路的连线状况﹐也能确保连线的准确性，是ping和traceroute的工作协议 |
| RIP | 路由信息协议 | 使用“跳数”(即metric)来衡量到达目标地址的路由距离 |
| IGMP | Internet 组管理协议 | 用于实现组播、广播等通信 |

**79、TCP四大拥塞控制算法总结？（极其重要）**

四大算法

拥塞控制主要是四个算法：1）慢启动，2）拥塞避免，3）拥塞发生，4）快速恢复。这四个算法不是一 天都搞出来的，这个四算法的发展经历了很多时间，到今天都还在优化中。

慢热启动算法 – Slow Start

所谓慢启动，也就是TCP连接刚建立，一点一点地提速，试探一下网络的承受能力，以免直接扰乱了 网络通道的秩序。

慢启动算法：

1. 连接建好的开始先初始化拥塞窗口cwnd大小为1，表明可以传一个MSS大小的数据。
2. 每当收到一个ACK，cwnd大小加一，呈线性上升。
3. 每当过了一个往返延迟时间RTT(Round-Trip Time)，cwnd大小直接翻倍，乘以2，呈指数让升。
4. 还有一个ssthresh（slow start threshold），是一个上限，当cwnd >= ssthresh时，就会进入“拥塞避免算法”（后面会说这个算法）

拥塞避免算法 – Congestion Avoidance

如同前边说的，当拥塞窗口大小cwnd大于等于慢启动阈值ssthresh后，就进入拥塞避免算法。算法 如下：

1. 收到一个ACK，则cwnd = cwnd + 1 / cwnd
2. 每当过了一个往返延迟时间RTT，cwnd大小加一。

过了慢启动阈值后，拥塞避免算法可以避免窗口增长过快导致窗口拥塞，而是缓慢的增加调整到网络 的最佳值。

拥塞发生状态时的算法

一般来说，TCP拥塞控制默认认为网络丢包是由于网络拥塞导致的，所以一般的TCP拥塞控制算法以 丢包为网络进入拥塞状态的信号。对于丢包有两种判定方式，一种是超时重传RTO[Retransmission Timeout]超时，另一个是收到三个重复确认ACK。

超时重传是TCP协议保证数据可靠性的一个重要机制，其原理是在发送一个数据以后就开启一个计时 器，在一定时间内如果没有得到发送数据报的ACK报文，那么就重新发送数据，直到发送成功为止。

但是如果发送端接收到3个以上的重复ACK，TCP就意识到数据发生丢失，需要重传。这个机制不需要 等到重传定时器超时，所以叫

做快速重传，而快速重传后没有使用慢启动算法，而是拥塞避免算法，所以这又叫做快速恢复算法。

超时重传RTO[Retransmission Timeout]超时，TCP会重传数据包。TCP认为这种情况比较糟糕，反应也比较强烈：

由于发生丢包，将慢启动阈值ssthresh设置为当前cwnd的一半，即ssthresh = cwnd / 2. cwnd重置为1

进入慢启动过程

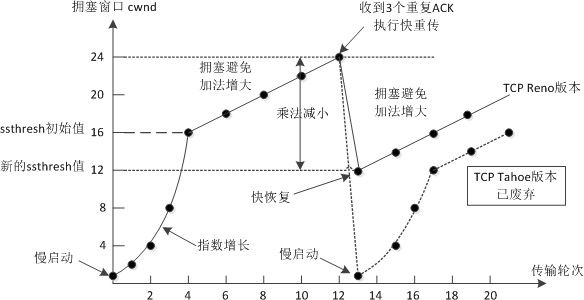
最为早期的TCP Tahoe算法就只使用上述处理办法，但是由于一丢包就一切重来，导致cwnd又重置为1，十分不利于网络数据的稳定传递。

所以，TCP Reno算法进行了优化。当收到三个重复确认ACK时，TCP开启快速重传Fast Retransmit

算法，而不用等到RTO超时再进行重传：

cwnd大小缩小为当前的一半ssthresh设置为缩小后的cwnd大小

然后进入快速恢复算法Fast Recovery。



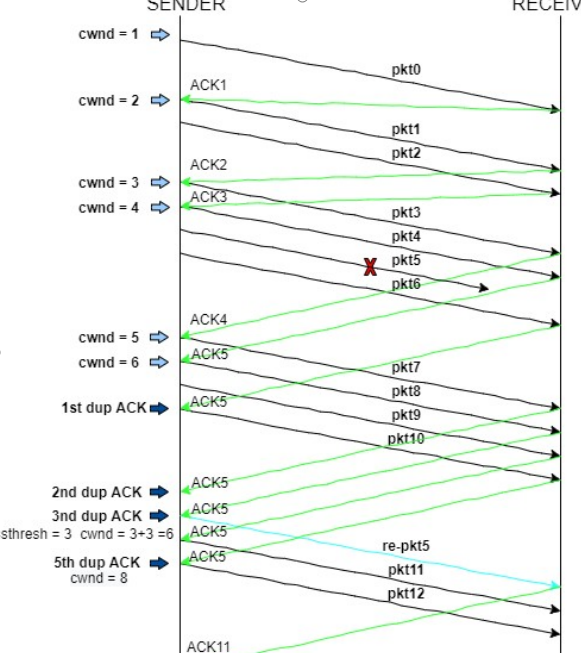
快速恢复算法 – Fast Recovery

TCP Tahoe是早期的算法，所以没有快速恢复算法，而Reno算法有。在进入快速恢复之前，cwnd和ssthresh已经被更改为原有cwnd的一半。快速恢复算法的逻辑如下：

cwnd = cwnd + 3 *MSS*，加*3* MSS的原因是因为收到3个重复的ACK。重传DACKs指定的数据包。

如果再收到DACKs，那么cwnd大小增加一。

如果收到新的ACK，表明重传的包成功了，那么退出快速恢复算法。将cwnd设置为ssthresh，然 后进入拥塞避免算法。



如图所示，第五个包发生了丢失，所以导致接收方接收到三次重复ACK，也就是ACK5。所以将ssthresh设置当当时cwnd的一半，也就是6/2 = 3，cwnd设置为3 + 3 = 6。然后重传第五个包。当收到新的ACK时，也就是ACK11，则退出快速恢复阶段，将cwnd重新设置为当前的ssthresh，也就是3，然 后进入拥塞避免算法阶段。

《TCP 拥塞控制算法简介》：<https://yq.aliyun.com/articles/691978>

### 80、为何快速重传是选择3次ACK？

主要的考虑还是要区分包的丢失是由于链路故障还是乱序等其他因素引发。

两次duplicated ACK时很可能是乱序造成的！三次duplicated ACK时很可能是丢包造成的！四次duplicated ACK更更更可能是丢包造成的，但是这样的响应策略太慢。丢包肯定会造成三次duplicated ACK!综上是选择收到三个重复确认时窗口减半效果最好，这是实践经验。

在没有fast retransmit / recovery 算法之前，重传依靠发送方的retransmit timeout，就是在timeout

内如果没有接收到对方的ACK，默认包丢了，发送方就重传，包的丢失原因

1. 包checksum 出错
2. 网络拥塞
3. 网络断，包括路由重收敛，但是发送方无法判断是哪一种情况，于是采用最笨的办法，就是将自己 的发送速率减半，即CWND 减为1/2，这样的方法对2是有效的，可以缓解网络拥塞，3则无所谓，反正网络断了，无论发快发慢都会被丢；但对于1来说，丢包是因为偶尔的出错引起，一丢包就对半减速不 合理。

于是有了fast retransmit 算法，基于在反向还可以接收到ACK，可以认为网络并没有断，否则也接收不到ACK，如果在timeout 时间内没有接收到> 2 的duplicated ACK，则概率大事件为乱序，乱序无需重传，接收方会进行排序工作；

而如果接收到三个或三个以上的duplicated ACK，则大概率是丢包，可以逻辑推理，发送方可以接收ACK，则网络是通的，可能是1、2造成的，先不降速，重传一次，如果接收到正确的ACK，则一切OK， 流速依然（包出错被丢）。

而如果依然接收到duplicated ACK，则认为是网络拥塞造成的，此时降速则比较合理。

[《TCP快速重传为什么是三次冗余ack，这个三次是怎么定下来的？》：https://blog.csdn.net/u0 10202588/article/details/54563648](https://blog.csdn.net/u010202588/article/details/54563648)

### 81、对于FIN\_WAIT\_2，CLOSE\_WAIT状态和TIME\_WAIT状态？你知道多少?

FIN\_WAIT\_2：

半关闭状态。

发送断开请求一方还有接收数据能力，但已经没有发送数据能力。

CLOSE\_WAIT状态：

被动关闭连接一方接收到FIN包会立即回应ACK包表示已接收到断开请求。 被动关闭连接一方如果还有剩余数据要发送就会进入CLOSED\_WAIT状态。

TIME\_WAIT状态：

又叫2MSL等待状态。

如果客户端直接进入CLOSED状态，如果服务端没有接收到最后一次ACK包会在超时之后重新 再发FIN包，此时因为客户端已经CLOSED，所以服务端就不会收到ACK而是收到RST。所以TIME\_WAIT状态目的是防止最后一次握手数据没有到达对方而触发重传FIN准备的。

在2MSL时间内，同一个socket不能再被使用，否则有可能会和旧连接数据混淆（如果新连接 和旧连接的socket相同的话）。

### 82、你了解流量控制原理吗？

目的是接收方通过TCP头窗口字段告知发送方本方可接收的最大数据量，用以解决发送速率过快导 致接收方不能接收的问题。所以流量控制是点对点控制。

TCP是双工协议，双方可以同时通信，所以发送方接收方各自维护一个发送窗和接收窗。

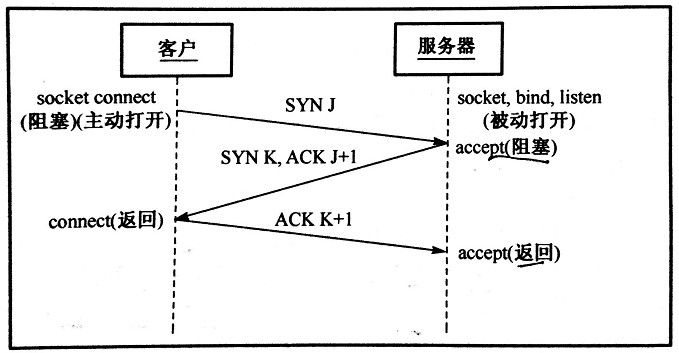
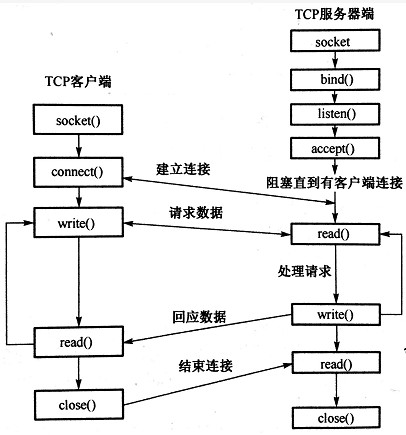
发送窗：用来限制发送方可以发送的数据大小，其中发送窗口的大小由接收端返回的TCP报 文段中窗口字段来控制，接收方通过此字段告知发送方自己的缓冲（受系统、硬件等限制） 大小。

接收窗：用来标记可以接收的数据大小。

TCP是流数据，发送出去的数据流可以被分为以下四部分：已发送且被确认部分 | 已发送未被确认部分 | 未发送但可发送部分 | 不可发送部分，其中发送窗 = 已发送未确认部分 + 未发但可发送部分。接收到的数据流可分为：已接收 | 未接收但准备接收 | 未接收不准备接收。接收窗 = 未接收但准备接收部分。

发送窗内数据只有当接收到接收端某段发送数据的ACK响应时才移动发送窗，左边缘紧贴刚被确认 的数据。接收窗也只有接收到数据且最左侧连续时才移动接收窗口。

### 83、建立TCP服务器的各个系统调用过程是怎样的？



服务器：

创建socket -> int socket(int domain, int type, int protocol);

domain：协议域，决定了socket的地址类型，IPv4为AF\_INET。type：指定socket类型，SOCK\_STREAM为TCP连接。

protocol：指定协议。IPPROTO\_TCP表示TCP协议，为0时自动选择type默认协议。绑定socket和端口号 -> int bind(int sockfd, const struct sockaddr \*addr, socklen\_t

addrlen);

sockfd：socket返回的套接字描述符，类似于文件描述符fd。addr：有个sockaddr类型数据的指针，指向的是被绑定结构变量。

};

struct in\_addr { uint32\_t s\_addr;

}

// IP地址

struct in\_addr sin\_addr;

in\_port\_t sin\_port; // 端口号

// 协议类型，AF\_INET

// IPv4的sockaddr地址结构

struct sockaddr\_in { sa\_family\_t sin\_family;

1

2

3

4

5

6

7

8

9

addrlen：地址长度。

监听端口号 -> int listen(int sockfd, int backlog);

sockfd：要监听的sock描述字。backlog：socket可以排队的最大连接数。

接收用户请求 -> int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

sockfd：服务器socket描述字。addr：指向地址结构指针。addrlen：协议地址长度。

注：一旦accept某个客户机请求成功将返回一个全新的描述符用于标识具体客户的TCP 连接。

从socket中读取字符 -> ssize\_t read(int fd, void \*buf, size\_t count);

fd：连接描述字。buf：缓冲区buf。count：缓冲区长度。

注：大于0表示读取的字节数，返回0表示文件读取结束，小于0表示发生错误。 关闭socket -> int close(int fd);

fd：accept返回的连接描述字，每个连接有一个，生命周期为连接周期。

注：sockfd是监听描述字，一个服务器只有一个，用于监听是否有连接；fd是连接描述 字，用于每个连接的操作。

客户机：

创建socket -> int socket(int domain, int type, int protocol);

连接指定计算机 -> int connect(int sockfd, struct sockaddr\* addr, socklen\_t addrlen);

sockfd客户端的sock描述字。addr：服务器的地址。addrlen：socket地址长度。

向socket写入信息 -> ssize\_t write(int fd, const void \*buf, size\_t count);

fd、buf、count：同read中意义。

大于0表示写了部分或全部数据，小于0表示出错。关闭oscket -> int close(int fd);

fd：同服务器端fd。

### 84、TCP 协议如何保证可靠传输？

第一种回答

**确认和重传**：接收方收到报文就会确认，发送方发送一段时间后没有收到确认就会重传。

**数据校验**：TCP报文头有校验和，用于校验报文是否损坏。

**数据合理分片和排序**：tcp会按最大传输单元(MTU)合理分片，接收方会缓存未按序到达的数据， 重新排序后交给应用层。而UDP：IP数据报大于1500字节，大于MTU。这个时候发送方的IP层就 需要分片，把数据报分成若干片，是的每一片都小于MTU。而接收方IP层则需要进行数据报的重 组。由于UDP的特性，某一片数据丢失时，接收方便无法重组数据报，导致丢弃整个UDP数据 报。

**流量控制**：当接收方来不及处理发送方的数据，能通过滑动窗口，提示发送方降低发送的速率，防 止包丢失。

**拥塞控制**：当网络拥塞时，通过拥塞窗口，减少数据的发送，防止包丢失。

第二种回答

建立连接（标志位）：通信前确认通信实体存在。

序号机制（序号、确认号）：确保了数据是按序、完整到达。数据校验（校验和）：CRC校验全部数据。

超时重传（定时器）：保证因链路故障未能到达数据能够被多次重发。窗口机制（窗口）：提供流量控制，避免过量发送。

拥塞控制：同上。

第三种回答

首部校验

这个校验机制能够确保数据传输不会出错吗？ 答案是不能。

原因

TCP协议中规定，TCP的首部字段中有一个字段是校验和，发送方将伪首部、TCP首部、TCP数据使用累 加和校验的方式计算出一个数字，然后存放在首部的校验和字段里，接收者收到TCP包后重复这个过

程，然后将计算出的校验和和接收到的首部中的校验和比较，如果不一致则说明数据在传输过程中出 错。

这就是TCP的数据校验机制。 但是这个机制能够保证检查出一切错误吗？**显然不能**。

因为这种校验方式是累加和，也就是将一系列的数字（TCP协议规定的是数据中的每16个比特位数据作 为一个数字）求和后取末位。 但是小学生都知道A+B=B+A，假如在传输的过程中有前后两个16比特位的数据前后颠倒了（至于为什么这么巧合？我不知道，也许路由器有bug？也许是宇宙中的高能粒子击 中了电缆？反正这个事情的概率不为零，就有可能会发生），那么校验和的计算结果和颠倒之前是一样 的，那么接收端肯定无法检查出这是错误的数据。

解决方案

传输之前先使用MD5加密数据获得摘要，跟数据一起发送到服务端，服务端接收之后对数据也进行

MD5加密，如果加密结果和摘要一致，则认为没有问题

《TCP新手误区--数据校验的意义》：<https://blog.csdn.net/bjrxyz/article/details/75194716>

### 85、UDP是什么

提供**无连接**的，尽最大努力的数据传输服务（**不保证数据传输的可靠性**）。

### 86、TCP和UDP的区别

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到 达;UDP尽最大努力交付，即不保证可靠交付

3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的

UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话， 实时视频会议等）

4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

5、TCP首部开销20字节;UDP的首部开销小，只有8个字节

6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

7、UDP是面向报文的，发送方的UDP对应用层交下来的报文，不合并，不拆分，只是在其上面加上首 部后就交给了下面的网络层，论应用层交给UDP多长的报文，它统统发送，一次发送一个。而对接收 方，接到后直接去除首部，交给上面的应用层就完成任务了。因此，它需要应用层控制报文的大小

TCP是面向字节流的，它把上面应用层交下来的数据看成无结构的字节流会发送，可以想象成流水形式 的，发送方TCP会将数据放入“蓄水池”（缓存区），等到可以发送的时候就发送，不能发送就等着TCP会 根据当前网络的拥塞状态来确定每个报文段的大小。

[《TCP数据段格式+UDP数据段格式详解》：https://www.cnblogs.com/love-jelly-pig/p/847118 1.html](https://www.cnblogs.com/love-jelly-pig/p/8471181.html)

### 87、UDP的特点有哪些（附赠TCP的特点）？

UDP是**无连接的**；

UDP使用**尽最大努力交付**，即不保证可靠交付，因此主机不需要维持复杂的链接状态（这里面有 许多参数）；

UDP是**面向报文**的；

UDP**没有拥塞控制**，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电 话，实时视频会议等）；

UDP**支持一对一、一对多、多对一和多对多**的交互通信；

UDP的**首部开销小**，只有8个字节，比TCP的20个字节的首部要短。

那么，再说一次TCP的特点：

**TCP是面向连接的**。（就好像打电话一样，通话前需要先拨号建立连接，通话结束后要挂机释放连 接）；

每一条TCP连接只能有两个端点，每一条TCP连接只能是点对点的（**一对一**）；

TCP**提供可靠交付的服务**。通过TCP连接传送的数据，无差错、不丢失、不重复、并且按序到达； TCP**提供全双工通信**。TCP允许通信双方的应用进程在任何时候都能发送数据。TCP连接的两端都 设有发送缓存和接收缓存，用来临时存放双方通信的数据；

**面向字节流**。TCP中的“流”（stream）指的是流入进程或从进程流出的字节序列。“面向字节流”的 含义是：虽然应用程序和TCP的交互是一次一个数据块（大小不等），但TCP把应用程序交下来的 数据仅仅看成是一连串的无结构的字节流。

### 88、TCP对应的应用层协议

FTP：定义了文件传输协议，使用21端口. Telnet：它是一种用于远程登陆的端口,23端口 SMTP：定义了简单邮件传送协议，服务器开放的是25号端口。POP3：它是和SMTP对应，POP3用于接收邮件。

### 89、UDP对应的应用层协议

DNS：用于域名解析服务，用的是53号端口SNMP：简单网络管理协议，使用161号端口

TFTP(Trival File Transfer Protocal)：简单文件传输协议，69

### 90、数据链路层常见协议？可以说一下吗？

|  |  |  |
| --- | --- | --- |
| **协议** | **名称** | **作用** |
| ARP | 地址解析协议 | 根据IP地址获取物理地址 |
| RARP | 反向地址转换协议 | 根据物理地址获取IP地址 |
| PPP | 点对点协议 | 主要是用来通过拨号或专线方式建立点对点连接发送数据，使其成为各种主机、网桥和路由器之间简单连接的一种共通的解决方案 |

《OSI七层模型与TCP/IP五层模型》：<https://www.cnblogs.com/qishui/p/5428938.html>

### 91、Ping命令基于哪一层协议的原理是什么？

ping命令基于网络层的命令，是基于ICMP协议工作的。

### 92、在进行UDP编程的时候，一次发送多少bytes好?

当然,这个没有唯一答案，相对于不同的系统,不同的要求,其得到的答案是不一样的。

我这里仅对像ICQ一类的发送聊天消息的情况作分析，对于其他情况，你或许也能得到一点帮助:首先,我 们知道,TCP/IP通常被认为是一个四层协议系统,包括链路层,网络层,运输层,应用层.UDP属于运输层,

下面我们由下至上一步一步来看:以太网(Ethernet)数据帧的长度必须在46-1500字节之间,这是由以太网 的物理特性决定的.这个1500字节被称为链路层的MTU(最大传输单元).但这并不是指链路层的长度被限 制在1500字节,其实这这个MTU指的是链路层的数据区.并不包括链路层的首部和尾部的18个字节.

所以,事实上,这个1500字节就是网络层IP数据报的长度限制。因为IP数据报的首部为20字节,所以IP数据 报的数据区长度最大为1480字节.而这个1480字节就是用来放TCP传来的TCP报文段或UDP传来的UDP 数据报的.又因为UDP数据报的首部8字节,所以UDP数据报的数据区最大长度为1472字节.这个1472字节 就是我们可以使用的字节数。

当我们发送的UDP数据大于1472的时候会怎样呢？

这也就是说IP数据报大于1500字节,大于MTU.这个时候发送方IP层就需要分片(fragmentation). 把数据报分成若干片,使每一片都小于MTU.而接收方IP层则需要进行数据报的重组.

这样就会多做许多事情,而更严重的是,由于UDP的特性,当某一片数据传送中丢失时,接收方便 无法重组数据报.将导致丢弃整个UDP数据报。

因此,在普通的局域网环境下，我建议将UDP的数据控制在1472字节以下为好.

进行Internet编程时则不同,因为Internet上的路由器可能会将MTU设为不同的值.

如果我们假定MTU为1500来发送数据的,而途经的某个网络的MTU值小于1500字节,那么系统将会使用 一系列的机

制来调整MTU值,使数据报能够顺利到达目的地,这样就会做许多不必要的操作.

鉴于Internet上的标准MTU值为576字节,所以我建议在进行Internet的UDP编程时. 最好将UDP的数据长度控件在548字节(576-8-20)以内

[《TCP协议中的窗口机制 滑动窗口详解》：https://blog.csdn.net/m0\_37962600/article/det](https://blog.csdn.net/m0_37962600/article/details/79951780)

[ails/79951780](https://blog.csdn.net/m0_37962600/article/details/79951780)

### 93、TCP 利用滑动窗口实现流量控制的机制？

流量控制是为了控制发送方发送速率，保证接收方来得及接收。TCP 利用滑动窗口实现流量控制。

TCP 中采用滑动窗口来进行传输控制，滑动窗口的大小意味着**接收方还有多大的缓冲区可以用于接收数据**。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为 0 时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据。

例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个 1 字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

### 94、可以解释一下RTO，RTT和超时重传分别是什么吗？

超时重传：发送端发送报文后若长时间未收到确认的报文则需要重发该报文。可能有以下几种情 况：

发送的数据没能到达接收端，所以对方没有响应。

接收端接收到数据，但是ACK报文在返回过程中丢失。接收端拒绝或丢弃数据。

RTO：从上一次发送数据，因为长期没有收到ACK响应，到下一次重发之间的时间。就是重传间 隔。

通常每次重传RTO是前一次重传间隔的两倍，计量单位通常是RTT。例：1RTT，2RTT，

4RTT，8RTT......

重传次数到达上限之后停止重传。 RTT：数据从发送到接收到对方响应之间的时间间隔，即数据报在网络中一个往返用时。大小不稳 定。

### 95、拥塞控制原理听说过吗？

拥塞控制目的是防止数据被过多注网络中导致网络资源（路由器、交换机等）过载。因为拥塞控制 涉及网络链路全局，所以属于全局控制。控制拥塞使用拥塞窗口。

TCP拥塞控制算法：

慢开始 & 拥塞避免：先试探网络拥塞程度再逐渐增大拥塞窗口。每次收到确认后拥塞窗口翻倍，直到达到阀值ssthresh，这部分是慢开始过程。达到阀值后每次以一个MSS为单位增长 拥塞窗口大小，当发生拥塞（超时未收到确认），将阀值减为原先一半，继续执行线性增 加，这个过程为拥塞避免。

快速重传 & 快速恢复：略。

最终拥塞窗口会收敛于稳定值。

### 96、如何区分流量控制和拥塞控制？

流量控制属于通信双方协商；拥塞控制涉及通信链路全局。

流量控制需要通信双方各维护一个发送窗、一个接收窗，对任意一方，接收窗大小由自身决定，发 送窗大小由接收方响应的TCP报文段中窗口值确定；拥塞控制的拥塞窗口大小变化由试探性发送一 定数据量数据探查网络状况后而自适应调整。

实际最终发送窗口 = min{流控发送窗口，拥塞窗口}。

### 97、常见的HTTP状态码有哪些？

|  |  |  |
| --- | --- | --- |
| **状态码** | **类别** | **含义** |
| 1XX | Informational（信息性状态码） | 接收的请求正在处理 |
| 2XX | Success（成功状态码） | 请求正常处理完毕 |
| 3XX | Redirection（重定向状态码） | 需要进行附加操作以完成请求 |
| 4XX | Client Error（客户端错误状态码） | 服务器无法处理请求 |
| 5XX | Server Error（服务器错误状态码） | 服务器处理请求出 |

1xx 信息

**100 Continue** ：表明到目前为止都很正常，客户端可以继续发送请求或者忽略这个响应。

2xx 成功

**200 OK**

**204 No Content** ：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。

**206 Partial Content** ：表示客户端进行了范围请求，响应报文包含由 Content-Range 指定范围的实体内容。

3xx 重定向

1. **Moved Permanently** ：永久性重定向
2. **Found** ：临时性重定向
3. **See Other** ：和 302 有着相同的功能，但是 303 明确要求客户端应该采用 GET 方法获取资源。
4. **Not Modiﬁed** ：如果请求报文首部包含一些条件，例如：If-Match，If-Modiﬁed-Since，If-

None-Match，If-Range，If-Unmodiﬁed-Since，如果不满足条件，则服务器会返回 304 状态码。

**307 Temporary Redirect** ：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的 POST 方法改成 GET 方法。

4xx 客户端错误

1. **Bad Request** ：请求报文中存在语法错误。
2. **Unauthorized** ：该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。

**403 Forbidden** ：请求被拒绝。

404 Not Found 5xx 服务器错误

**500 Internal Server Error** ：服务器正在执行请求时发生错误。

**503 Service Unavailable** ：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。

## 设计模式

### 0 23种设计模式速记

设计模式的三大类

**创建型模式（Creational Pattern）：**对类的实例化过程进行了抽象，能够将软件模块中**对象的创建**和对象的使用分离。

（5种）工厂模式、抽象工厂模式、单例模式、建造者模式、原型模式

记忆口诀：创工原单建抽（创公园，但见愁）

**结构型模式（Structural Pattern）**：关注于对象的组成以及对象之间的依赖关系，描述如何将类或者对象结合在一起形成更大的结构，就像**搭积木**，可以通过简单积木的组合形成复杂的、功能更为强大的结构。

（7种）适配器模式、装饰者模式、代理模式、外观模式、桥接模式、组合模式、享元模式

记忆口诀：结享外组适代装桥（姐想外租，世代装桥）

**行为型模式（Behavioral Pattern）**：关注于对象的行为问题，是对在不同的对象之间划分责任和算法的抽象化；不仅仅关注类和对象的结构，而且重点关注它们之间的**相互作用**。

（11种）策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

记忆口诀：行状责中模访解备观策命迭（模仿这种形状，观测戒备鸣笛儿）

**23种设计模式的分类与特点概述**

**1. 创建型设计模式（Creational Patterns）：**

这些模式关注对象的创建机制，旨在提供一种更灵活、更复杂的对象创建方式，而不是直接实例化对象。常见的创建型模式包括：

单例模式（Singleton Pattern）： 确保一个类只有一个实例，并提供全局访问点。

工厂方法模式（Factory Method Pattern）： 定义一个创建对象的接口，但将对象的实际创建延迟到子类。

抽象工厂模式（Abstract Factory Pattern）： 提供一个创建相关对象家族的接口，而无需指定具体类。

建造者模式（Builder Pattern）： 将一个复杂对象的构建过程分解为多个步骤，以便更灵活地创建对象。

原型模式（Prototype Pattern）： 通过复制现有对象来创建新对象，而不是从头开始构建。

**2. 结构型设计模式（Structural Patterns）：**

这些模式关注如何组合类和对象以形成更大的结构，以解决系统中对象之间的关系。常见的结构型模式包括：

适配器模式（Adapter Pattern）： 允许将一个接口转换为另一个客户端期望的接口。

装饰者模式（Decorator Pattern）： 动态地为对象添加新的功能，而无需修改其源代码。

代理模式（Proxy Pattern）： 提供一个代理对象以控制对其他对象的访问。

组合模式（Composite Pattern）： 允许客户端以统一的方式处理单个对象和对象组合。

桥接模式（Bridge Pattern）： 将抽象部分与其实现分离，以便它们可以独立变化。

外观设计模式（Facade Design Pattern）： 外观模式提供了一个高级别的接口，将多个底层接口组合成一个更简单的接口，以供客户端使用。这有助于隔离客户端代码和复杂子系统之间的耦合关系。

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

### 1 谈谈你对简单工厂设计模式的理解；

简单工厂设计模式（Simple Factory Pattern）是一种创建型设计模式，它提供了一个工厂类，用于根据客户端的请求创建不同类型的对象，而客户端无需直接与具体类相互耦合。这个工厂类拥有一个方法，该方法接受客户端的参数，并根据参数的不同来创建并返回相应的对象。

核心思想： 将对象的创建过程封装在一个独立的工厂类中，客户端只需要通过工厂类来创建所需的对象，而不需要直接实例化具体的类。

简单工厂的结构：

工厂类（Factory）： 负责创建具体对象的类，通常包含一个或多个创建对象的方法。

产品类（Product）： 具体对象的抽象，由工厂类来创建，客户端与产品类进行交互。

示例：

假设你正在开发一个图形绘制应用程序，需要根据用户的选择创建不同类型的图形对象，如圆形（Circle）和矩形（Rectangle）。你可以使用简单工厂来实现这个场景。

|  |
| --- |
| // 产品类 - 图形  class Shape {  public:  virtual void draw() = 0;  };  // 具体产品类 - 圆形  class Circle : public Shape {  public:  void draw() override {  *std*::*cout* << "绘制圆形" << *std*::*endl*;  }  };  // 具体产品类 - 矩形  class Rectangle : public Shape {  public:  void draw() override {  *std*::*cout* << "绘制矩形" << *std*::*endl*;  }  };  // 工厂类  class ShapeFactory {  public:  // 根据类型创建图形对象  static Shape\* createShape(*std*::*string* **type**) {  if (**type** == "Circle") {  return new Circle();  }  else if (**type** == "Rectangle") {  return new Rectangle();  }  else {  return nullptr; // 可以根据需要添加其他图形类型的创建逻辑  }  }  }; |

应用场景：

**图形绘制应用程序**： 如上面的示例，根据用户选择的不同图形类型创建相应的图形对象。

**日志记录器**： 根据不同的日志级别（信息、警告、错误）创建不同的日志记录器对象。

**数据库连接管理**： 根据数据库类型（MySQL、PostgreSQL、Oracle）创建数据库连接对象。

**UI库控件**： 根据用户需求创建不同类型的UI控件（按钮、文本框、下拉菜单等）。

简单工厂模式的优点在于**封装了对象的创建过程，客户端代码与具体类解耦，降低了代码的维护复杂度。但缺点是当需要添加新的产品类型时，需要修改工厂类的代码，违反了开闭原则**。因此，如果需要频繁添加新的产品类型，可能需要考虑其他创建型设计模式，如工厂方法模式或抽象工厂模式。

### 2 单例设计模式的饿汉式和懒汉式的区别；

单例设计模式旨在确保一个类只有一个实例，并提供全局访问点以访问该实例。在单例模式中，有两种常见的实现方式，即饿汉式和懒汉式。它们的主要区别在于实例的创建时间和线程安全性。

|  |
| --- |
| class Singleton {  private:  static Singleton\* **instance**;  Singleton() {} // 私有构造函数，禁止外部实例化  public:  static Singleton\* getInstance() {  if (**instance** == nullptr) {  **instance** = new Singleton();  }  return **instance**;  }  }; |

在懒汉式中，实例在第一次被请求时创建。这意味着实例的创建是延迟的，直到需要使用它时才会创建。

1. **饿汉式单例模式**：

在饿汉式中，实例在类加载时就被创建，不管是否需要使用它。这意味着无论何时引用该类，都会返回相同的实例。

**主要特点**：

**线程安全**：在多线程环境中，由于实例在类加载时创建，所以不会存在竞态条件，是线程安全的。

简单：实现简单，但可能会浪费内存，因为无论是否需要，实例都会创建。

1. **懒汉式单例模式**：

**主要特点：**

**延迟加载**：实例的创建是延迟的，只有在需要时才会创建。

非线程安全：在多线程环境中，多个线程可能同时访问getInstance()方法，导致创建多个实例。需要额外的同步措施来保证线程安全。

**注意事项**：

在懒汉式中，需要考虑多线程环境下的线程安全问题。可以使用加锁或双重检查锁等方式来确保线程安全。

饿汉式的主要优点是简单和线程安全，但可能浪费内存。懒汉式则具有延迟加载的特点，但需要处理线程安全性。选择哪种方式取决于具体需求和性能考虑。如果不需要延迟加载，并且希望简单的实现，可以选择饿汉式。如果需要延迟加载，并且能够处理线程安全问题，可以选择懒汉式。

**3. 行为型设计模式（Behavioral Patterns）：**

这些模式关注对象之间的通信和职责分配，以帮助在系统中实现更松散耦合的对象之间的通信。常见的行为型模式包括：

模板方法（Template Method Pattern）是：定义一个算法的骨架，而将一些步骤的具体实现延迟到子类中。这种模式允许在不改变算法结构的情况下，通过子类重写或扩展某些步骤来定制算法的行为。

观察者模式（Observer Pattern）： 定义一种一对多的依赖关系，使得当一个对象状态发生变化时，所有依赖于它的对象都得到通知并自动更新。

策略模式（Strategy Pattern）： 定义一系列算法，将它们封装成对象，并使其可以互换使用，以便根据需要动态选择算法。

命令模式（Command Pattern）： 将请求封装成对象，以便可以参数化客户端对象，排队请求和记录日志，以及支持可撤销操作。

状态模式（State Pattern）： 允许对象在其内部状态发生变化时改变其行为。

责任链模式（Chain of Responsibility Pattern）： 将请求的发送者和接收者解耦，以便多个对象都有机会处理请求，将请求沿链传递，直到有一个对象处理它为止。

### 3 怎么实现懒汉式单例设计模式的线程安全？

双重检查锁定（Double-Checked Locking）是一种用于实现懒汉式单例模式的常见线程安全技术。它的目的是在多线程环境下延迟实例化对象，同时确保只有一个实例被创建。这种技术通常用于需要考虑性能的情况下，因为它在绝大多数情况下避免了不必要的锁定操作。

双重检查锁定的基本思想是：

首先检查实例是否已经创建，如果已经创建，则直接返回实例。

如果实例尚未创建，才进行锁定操作。

在锁定的情况下再次检查实例是否已经创建，以防止其他线程在等待锁定时已经创建了实例。

如果实例仍未创建，才创建并返回实例。

以下是一个使用双重检查锁定的懒汉式单例模式示例

|  |
| --- |
| class Singleton {  private:  static Singleton\* **instance**;  static *std*::*mutex* **mtx**; // 用于线程安全的锁  Singleton() {} // 私有构造函数，禁止外部实例化  public:  static Singleton\* getInstance() {  // 第一次检查，如果实例已经存在，直接返回  if (**instance** == nullptr) {  *std*::*lock\_guard*<*std*::*mutex*> **lock**(**mtx**); // 加锁  // 第二次检查，防止在等待锁期间其他线程已经创建实例  if (**instance** == nullptr) {  **instance** = new Singleton(); // 创建实例  }  }  return **instance**;  }  };  // 静态成员初始化  Singleton\* Singleton::**instance** = nullptr;  *std*::*mutex* Singleton::**mtx**; |

双重检查锁定确保只有在实例未创建时才会加锁和创建实例，从而提高了性能。

需要注意的是，双重检查锁定需要在多线程环境下使用，因为它涉及到线程同步。另外，C++11之后，还可以使用更方便的局部静态变量来实现懒汉式单例模式，因为C++11引入了线程安全的局部静态变量初始化。

### 4 聊聊你对抽象工厂设计模式的理解：

抽象工厂设计模式是一种创建型设计模式，它提供了一种创建一系列相关或相互依赖对象的接口，而无需指定其具体类。

抽象工厂设计模式通过定义一个抽象工厂接口，该接口声明了一组用于创建不同产品族的抽象方法。每个具体的工厂类实现这个接口，并负责创建特定的产品族。具体产品类实现了产品接口，每个工厂类创建的产品都属于同一个产品族，但是产品族之间有不同的实现。

抽象工厂模式的核心思想是通过创建一组相关的产品，而不是单个产品。它提供了一种封装对象创建的方式，使得客户端可以从具体实现中解耦。

下面是一个简单的例子来说明抽象工厂设计模式的实现：

|  |
| --- |
| #include <iostream>  // 抽象产品接口  class Button {  public:  virtual void render() = 0;  };  // 具体产品类A  class WindowsButton : public Button {  public:  void render() override {  *std*::*cout* << "Rendering a Windows button." << *std*::*endl*;  }  };  // 具体产品类B  class MacButton : public Button {  public:  void render() override {  *std*::*cout* << "Rendering a Mac button." << *std*::*endl*;  }  };  // 抽象工厂接口  class GUIFactory {  public:  virtual Button\* createButton() = 0;  };  // 具体工厂类A  class WindowsFactory : public GUIFactory {  public:  Button\* createButton() override {  return new WindowsButton();  }  };  // 具体工厂类B  class MacFactory : public GUIFactory {  public:  Button\* createButton() override {  return new MacButton();  }  };  int main() {  // 创建一个Windows风格的工厂  GUIFactory\* **windowsFactory** = new WindowsFactory();  // 创建一个Windows风格的按钮  Button\* **windowsButton** = **windowsFactory**->createButton();  // 渲染按钮  **windowsButton**->render(); // Output: Rendering a Windows button.  // 创建一个Mac风格的工厂  GUIFactory\* **macFactory** = new MacFactory();  // 创建一个Mac风格的按钮  Button\* **macButton** = **macFactory**->createButton();  // 渲染按钮  **macButton**->render(); // Output: Rendering a Mac button.  delete **windowsFactory**;  delete **windowsButton**;  delete **macFactory**;  delete **macButton**;  return 0;  } |

在上述代码中定义了一个抽象产品接口 Button，它声明了一个抽象方法 render。然后我们创建了两个具体产品类 WindowsButton 和 MacButton，它们分别实现了 Button 接口。

接着，我们定义了一个抽象工厂接口 GUIFactory，它声明了一个抽象方法 createButton。然后我们创建了两个具体工厂类 WindowsFactory 和 MacFactory，它们分别实现了 GUIFactory 接口，并分别负责创建对应的产品。

在 main 函数中，我们首先创建了一个 WindowsFactory 工厂对象，然后通过该工厂对象创建了一个 WindowsButton 按钮对象，并调用其 render 方法来渲染按钮。接着，我们创建了一个 MacFactory 工厂对象，通过该工厂对象创建了一个 MacButton 按钮对象，并渲染按钮。

通过抽象工厂设计模式，我们可以通过工厂来创建一组相关的产品，而无需关心具体的产品类。这样可以实现一系列产品的一致性和互换性，同时也符合了开闭原则，使得系统更加灵活和可扩展。

场景举例：

图形界面库：图形界面库通常需要创建一组相关的界面元素，如按钮、文本框、标签等。通过抽象工厂模式，可以创建不同风格的界面元素，如Windows风格、Mac风格等，确保界面元素之间的风格一致性。

数据库访问：在数据库访问中，可能需要使用不同的数据库引擎，如MySQL、Oracle、SQL Server等。通过抽象工厂模式，可以创建不同数据库引擎的连接对象、命令对象等，以便在运行时根据需要切换使用不同的数据库引擎。

操作系统适配：在开发跨平台的软件时，可能需要根据不同的操作系统提供不同的实现。通过抽象工厂模式，可以根据操作系统的不同创建对应的工厂对象，从而提供与操作系统相关的实现。

### 5 你对原型设计模式的理解：

原型设计模式是一种对象创建型模式，它通过**复制现有对象来创建新的对象，而无需显式地调用构造函数。简单来说，原型设计模式就是通过克隆已有对象来创建新对象**。

举一个通俗易懂的例子，假设我们有一个图形库，其中有一个基类 Shape 表示图形，它有一个纯虚函数 draw() 用于绘制图形。现在我们希望能够复制已有的图形对象来创建新的图形对象，而无需重新构造图形对象。

下面是一个使用C++代码示例来说明原型设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <string>  // 图形基类  class Shape {  public:  virtual \~Shape() {}  virtual void draw() = 0;  virtual Shape\* clone() = 0;  };  // 圆形类  class Circle : public Shape {  private:  *std*::*string* **color**;  int **radius**;  public:  Circle(*std*::*string* **color**, int **radius**) : **color**(**color**), **radius**(**radius**) {}  void draw() override {  *std*::*cout* << "Drawing a circle with color " << **color** << " and radius " << **radius** << *std*::*endl*;  }  Shape\* clone() override {  return new Circle(\*this);  }  };  int main() {  // 创建一个原型对象  Shape\* **circlePrototype** = new Circle("red", 5);  // 克隆原型对象来创建新对象  Shape\* **circle1** = **circlePrototype**->clone();  Shape\* **circle2** = **circlePrototype**->clone();  // 绘制图形  **circle1**->draw(); // Output: Drawing a circle with color red and radius 5  **circle2**->draw(); // Output: Drawing a circle with color red and radius 5  delete **circlePrototype**;  delete **circle1**;  delete **circle2**;  return 0;  } |

在上述代码中，我们定义了一个图形基类 Shape，它声明了纯虚函数 draw 和 clone。然后我们创建了一个具体的图形类 Circle，它继承自 Shape，并实现了 draw 和 clone 函数。

在 main 函数中，我们首先创建了一个原型对象 circlePrototype，然后通过调用 clone 方法来创建了两个克隆对象 circle1 和 circle2。最后，我们分别调用了 draw 方法来绘制图形。

通过原型设计模式，我们可以通过复制已有的图形对象来创建新的图形对象，而无需重新构造图形对象。这样可以提高对象创建的效率，并且可以动态地添加新的图形对象。

原型设计模式的应用场景通常包括以下情况：

1. 创建对象的过程比较复杂，且需要频繁地创建相似对象。原型设计模式可以通过复制已有对象来创建新对象，避免了复杂的创建过程，提高了创建对象的效率。

2. 需要动态地添加新的对象，而且这些对象的类型在运行时才能确定。原型设计模式通过克隆已有对象来创建新对象，可以在运行时动态地添加新的对象类型，而无需显式地调用构造函数。

典型应用场景的举例：

1. 图形编辑器：在图形编辑器中，可能需要频繁地创建相似的图形对象，如矩形、圆形等。使用原型设计模式，可以先创建一个原型对象，并通过克隆原型对象来创建新的图形对象，而无需重新构造对象。

2. 缓存系统：在缓存系统中，可能需要频繁地复制已有的缓存对象来创建新的缓存对象。使用原型设计模式，可以通过克隆已有的缓存对象来创建新的缓存对象，而无需重新从数据库或其他地方获取数据。

3. 原始对象的状态保存：有时候需要保存对象的某个特定状态，以便在需要时恢复回来。原型设计模式可以通过克隆对象来保存和恢复对象的状态，而不需要手动记录和恢复每个属性的值。

原型设计模式适用于需要频繁地创建相似对象或动态地添加新对象的场景。它可以通过复制已有对象来创建新对象，避免了复杂的创建过程，并且可以在运行时动态地添加新的对象类型。

### 6 建造者设计模式的理解：

建造者设计模式是一种创建型设计模式，旨在将**对象的构建过程与其表示分离**。该模式允许逐步构建复杂的对象，同时保持构建过程的灵活性。

建造者设计模式的核心思想是将一个复杂对象的构建过程分解为多个简单的步骤，每个步骤由一个具体的建造者类负责实现。这些步骤按照一定的顺序被调用，最终构建出一个完整的对象。

下面是一个简单的C++代码示例来说明建造者设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <string>  // 产品类  class Pizza {  public:  void setDough(const *std*::*string*& **dough**) {  **m\_dough** = **dough**;  }  void setSauce(const *std*::*string*& **sauce**) {  **m\_sauce** = **sauce**;  }  void setTopping(const *std*::*string*& **topping**) {  **m\_topping** = **topping**;  }  void showPizza() const {  *std*::*cout* << "Pizza with " << **m\_dough** << " dough, " << **m\_sauce** << " sauce, and " << **m\_topping** << " topping." << *std*::*endl*;  }  private:  *std*::*string* **m\_dough**;  *std*::*string* **m\_sauce**;  *std*::*string* **m\_topping**;  };  // 抽象建造者类  class PizzaBuilder {  public:  virtual \~PizzaBuilder() {}  virtual void buildDough() = 0;  virtual void buildSauce() = 0;  virtual void buildTopping() = 0;  virtual Pizza\* getPizza() = 0;  };  // 具体建造者类A  class HawaiianPizzaBuilder : public PizzaBuilder {  public:  void buildDough() override {  **m\_pizza**->setDough("cross");  }  void buildSauce() override {  **m\_pizza**->setSauce("mild");  }  void buildTopping() override {  **m\_pizza**->setTopping("ham and pineapple");  }  Pizza\* getPizza() override {  return **m\_pizza**;  }  private:  Pizza\* **m\_pizza** = new Pizza();  };  // 具体建造者类B  class SpicyPizzaBuilder : public PizzaBuilder {  public:  void buildDough() override {  **m\_pizza**->setDough("pan baked");  }  void buildSauce() override {  **m\_pizza**->setSauce("hot");  }  void buildTopping() override {  **m\_pizza**->setTopping("pepperoni and jalapeno");  }  Pizza\* getPizza() override {  return **m\_pizza**;  }  private:  Pizza\* **m\_pizza** = new Pizza();  };  // 指导者类  class PizzaDirector {  public:  void setPizzaBuilder(PizzaBuilder\* **builder**) {  **m\_builder** = **builder**;  }  void constructPizza() {  **m\_builder**->buildDough();  **m\_builder**->buildSauce();  **m\_builder**->buildTopping();  }  private:  PizzaBuilder\* **m\_builder**;  };  int main() {  PizzaDirector **director**;  HawaiianPizzaBuilder **hawaiianBuilder**;  **director**.setPizzaBuilder(&**hawaiianBuilder**);  **director**.constructPizza();  Pizza\* **hawaiianPizza** = **hawaiianBuilder**.getPizza();  **hawaiianPizza**->showPizza(); // .  SpicyPizzaBuilder **spicyBuilder**;  **director**.setPizzaBuilder(&**spicyBuilder**);  **director**.constructPizza();  Pizza\* **spicyPizza** = **spicyBuilder**.getPizza();  **spicyPizza**->showPizza(); //  delete **hawaiianPizza**;  delete **spicyPizza**;  return 0;  } |

在上述代码中，我们定义了一个产品类 Pizza，它具有三个属性：面团（dough）、酱料（sauce）和配料（topping）。然后我们定义了一个抽象建造者类 PizzaBuilder，它声明了一系列构建步骤的抽象方法，并提供了获取构建结果的方法 getPizza。

接下来，我们创建了两个具体建造者类 HawaiianPizzaBuilder 和 SpicyPizzaBuilder，它们分别实现了抽象建造者类中的构建步骤方法。这些方法负责设置不同的面团、酱料和配料。

然后我们定义了一个指导者类 PizzaDirector，它有一个成员变量指向具体的建造者对象。指导者类负责按照一定的顺序调用建造者对象的构建步骤方法来构建产品。

在 main 函数中，我们首先创建一个指导者对象，并将具体建造者对象 HawaiianPizzaBuilder 传递给指导者对象。然后通过指导者对象调用构建方法来构建产品。最后通过具体建造者对象的 getPizza 方法获取构建结果，并调用 showPizza 方法展示产品信息。

建造者设计模式的优点是可以将对象的构建过程与表示分离，使得构建过程更加灵活，同时可以避免构造函数的参数过多的问题。此外，通过使用建造者模式，可以更好地控制对象的创建过程，使得代码更加清晰易懂。

建造者设计模式的应用场景包括但不限于以下情况：

创建复杂的对象：当对象的构建过程比较复杂，涉及多个步骤、多个组件或者需要进行一些复杂的计算或处理时，可以使用建造者模式来将构建过程分解为多个简单的步骤，使得构建过程更加清晰和可控。

避免构造函数的参数过多：当一个对象的构造函数需要传递大量的参数时，使用构造函数来创建对象会导致代码难以理解和维护。而使用建造者模式，可以通过一系列方法来设置对象的属性，使得代码更加清晰易懂。

创建一系列相似的对象：当需要创建一系列相似但属性略有不同的对象时，使用建造者模式可以通过复用已有的建造者对象来构建这些对象，减少重复的代码。

控制对象的创建过程：建造者模式允许在构建过程中灵活地添加、修改或替换组件，从而更好地控制对象的创建过程。这样可以根据不同的需求，使用不同的建造者来构建具有不同特性的对象。

**应用场景**：

创建游戏角色：在游戏开发中，创建游戏角色通常涉及到多个属性的设置，例如角色的外观、能力、武器等。使用建造者模式可以将角色的构建过程分解为多个步骤，每个步骤由一个具体的建造者负责设置相应的属性。这样可以根据不同的建造者来构建不同类型的游戏角色。

构建网页：在网页设计中，创建网页通常需要设置不同的元素、样式和布局。使用建造者模式可以将网页的构建过程分解为多个步骤，例如设置标题、导航栏、内容区域等，每个步骤由一个具体的建造者负责设置相应的元素和样式。这样可以根据不同的建造者来构建不同风格和布局的网页。

创建菜单：在餐厅或者饮食应用中，创建菜单通常需要设置菜品的名称、描述、价格和配料等。使用建造者模式可以将菜单的构建过程分解为多个步骤，每个步骤由一个具体的建造者负责设置相应的属性。这样可以根据不同的建造者来构建不同类型和特色的菜单。

生成报告：在报告生成系统中，生成报告通常需要设置报告的标题、作者、日期、内容等。使用建造者模式可以将报告的构建过程分解为多个步骤，每个步骤由一个具体的建造者负责设置相应的属性。这样可以根据不同的建造者来构建不同类型和格式的报告。

### 7 享元设计模式的理解：

享元设计模式（Flyweight Design Pattern）是一种结构型设计模式，旨在**通过共享对象来最小化内存使用和提高性能**。该模式适用于需要创建大量细粒度对象的情况，通过共享相同的对象实例来减少内存占用。

在享元模式中，对象被分为两种类型：内部状态（Intrinsic State）和外部状态（Extrinsic State）。内部状态是对象共享的部分，不会随着外部环境的改变而改变；外部状态是对象特定的部分，会随着外部环境的改变而改变。

下面是一个简单的C++代码示例来说明享元设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <vector>  #include <unordered\_map>  // 抽象享元类  class Flyweight {  public:  virtual void operation(int **extrinsicState**) = 0;  };  // 具体享元类  class ConcreteFlyweight : public Flyweight {  public:  void operation(int **extrinsicState**) override {  *std*::*cout* << "具体享元对象，外部状态为：" << **extrinsicState** << *std*::*endl*;  }  };  // 享元工厂类  class FlyweightFactory {  private:  *std*::*unordered\_map*<int, Flyweight\*> **flyweights**;  public:  Flyweight\* getFlyweight(int **key**) {  if (**flyweights**.*find*(**key**) == **flyweights**.*end*()) {  **flyweights**[**key**] = new ConcreteFlyweight();  }  return **flyweights**[**key**];  }  };  int main() {  FlyweightFactory **factory**;  Flyweight\* **flyweight** = **factory**.getFlyweight(1);  **flyweight**->operation(100);  **flyweight** = **factory**.getFlyweight(2);  **flyweight**->operation(200);  return 0;  } |

在上面的示例中，Flyweight 是抽象享元类，定义了享元对象的操作方法。ConcreteFlyweight 是具体享元类，实现了抽象享元类的方法。FlyweightFactory 是享元工厂类，用于创建和管理享元对象。

在 main 函数中，我们首先创建了一个享元工厂对象 factory。然后通过 factory 获取具体享元对象，调用其操作方法来展示对象的外部状态。

享元设计模式的应用场景：

**大量细粒度对象**：当需要创建大量细粒度的对象，并且这些对象之间有很多共享的部分时，可以使用享元模式来共享相同的对象实例，减少内存占用。

**缓存**：在需要频繁访问的数据缓存中，使用享元模式可以减少重复创建对象的开销，提高访问速度。

**文字处理器**：在文字处理器中，每个字符都可以看作一个享元对象，通过共享相同的字符对象实例，可以减少内存使用量。

**游戏中的角色和地图**：在游戏中，角色和地图等对象可能会有多个实例，使用享元模式可以共享相同的对象实例，提高游戏性能。

总之，享元设计模式适用于需要创建大量细粒度对象，并且这些对象之间有很多共享部分的场景。通过共享相同的对象实例，可以减少内存占用和提高性能。

### 8 外观者设计模式的理解：

外观设计模式（Facade Design Pattern）是一种结构型设计模式，**旨在提供一个统一的接口，用于访问子系统中的一组接口，以简化客户端与子系统之间的交互**。它允许客户端通过与外观对象交互，而不必直接与子系统的复杂性相互作用，从而降低了客户端代码的复杂性和依赖性。

核心思想： 外观模式提供了一个高级别的接口，将多个底层接口组合成一个更简单的接口，以供客户端使用。这有助于隔离客户端代码和复杂子系统之间的耦合关系。

主要组成部分：

外观类（Facade）： 提供一个简单的接口，封装了对一个或多个子系统的复杂操作。它知道如何与子系统协调工作以完成特定任务。

子系统类（Subsystems）： 这些类包含了系统的具体实现，但对客户端是不可见的。客户端通过外观类来与这些子系统进行交互。

特点：

简化接口：外观模式提供了一个简化的接口，隐藏了子系统的复杂性，使客户端更容易使用。

降低耦合：客户端代码与子系统之间的依赖性降低，因为客户端只需与外观对象交互，而不需要直接与多个子系统交互。

改进可维护性：外观模式将系统的组件组织得更好，有助于维护和修改系统的不同部分，而不会影响客户端。

速记：  
去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。

示例：

假设你正在开发一个多媒体播放器应用程序，其中包含**音频播放、视频播放和字幕显示**等多个子系统。你可以使用外观模式来创建一个多媒体播放器的外观类，将这些子系统组合起来，并提供一个简化的接口供客户端使用：

|  |
| --- |
| // 外观类 - 多媒体播放器  class MultimediaPlayer {  private:  AudioPlayer **audioPlayer**;  VideoPlayer **videoPlayer**;  SubtitlePlayer **subtitlePlayer**;  public:  MultimediaPlayer() {  // 初始化子系统  }  // 提供一个播放多媒体文件的简化接口  void playMedia(const *std*::*string*& **mediaFilePath**) {  **audioPlayer**.playAudio(**mediaFilePath**);  **videoPlayer**.playVideo(**mediaFilePath**);  **subtitlePlayer**.displaySubtitles(**mediaFilePath**);  }  };  // 子系统类 - 音频播放器  class AudioPlayer {  public:  void playAudio(const *std*::*string*& **filePath**) {  *std*::*cout* << "播放音频：" << **filePath** << *std*::*endl*;  // 实际音频播放逻辑  }  };  // 子系统类 - 视频播放器  class VideoPlayer {  public:  void playVideo(const *std*::*string*& **filePath**) {  *std*::*cout* << "播放视频：" << **filePath** << *std*::*endl*;  // 实际视频播放逻辑  }  };  // 子系统类 - 字幕显示器  class SubtitlePlayer {  public:  void displaySubtitles(const *std*::*string*& **filePath**) {  *std*::*cout* << "显示字幕：" << **filePath** << *std*::*endl*;  // 实际字幕显示逻辑  }  }; |

在上面的示例中，MultimediaPlayer 类充当外观，隐藏了音频播放、视频播放和字幕显示等子系统的复杂性，提供了一个简化的接口供客户端使用。

应用场景：

外观模式通常在以下情况下使用：

当一个系统的多个复杂子系统需要与客户端交互，但客户端不需要了解这些子系统的具体工作方式时。

当你希望将系统的组件分层，使客户端可以轻松访问高级别的接口，而不必处理底层的复杂性。

当你希望隔离客户端代码和底层子系统，以提高代码的可维护性和可扩展性。 外观模式可以帮助简化大型系统的设计和开发，降低了系统的复杂性。

### 9 组合设计模式的理解：

组合设计模式（Composite Design Pattern）是一种结构型设计模式，旨在将对象组织成树形结构，以表示“整体-部分”的层次结构。该模式使得客户端可以一致地处理单个对象和组合对象，从而简化了客户端的代码。

在组合模式中，有两种基本类型的对象：叶子对象（Leaf）和组合对象（Composite）。叶子对象表示树中的最细粒度的对象，没有子对象；组合对象是由一个或多个叶子对象或其他组合对象组成的对象，可以拥有子对象。

下面是一个简单的C++代码示例来说明组合设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <vector>  // 抽象组件类  class Component {  public:  virtual void operation() = 0;  };  // 叶子组件类  class Leaf : public Component {  public:  void operation() override {  *std*::*cout* << "叶子组件的操作" << *std*::*endl*;  }  };  // 组合组件类  class Composite : public Component {  private:  *std*::*vector*<Component\*> **components**;  public:  void addComponent(Component\* **component**) {  **components**.*push\_back*(**component**);  }  void operation() override {  *std*::*cout* << "组合组件的操作" << *std*::*endl*;  for (Component\* **component** : **components**) {  **component**->operation();  }  }  };  int main() {  Component\* **leaf** = new Leaf();  Component\* **composite** = new Composite();  **composite**->addComponent(**leaf**);  **composite**->operation();  delete **composite**;  delete **leaf**;  return 0;  } |

在上面的示例中，Component 是抽象组件类，定义了组件对象的操作方法。Leaf 是叶子组件类，实现了抽象组件类的操作方法。Composite 是组合组件类，可持有多个叶子组件或其他组合组件，并实现了抽象组件类的操作方法。

在 main 函数中，我们首先创建了一个叶子组件对象 leaf 和一个组合组件对象 composite。然后通过 composite 的 addComponent 方法将叶子组件对象添加到组合组件对象中。最后调用组合组件对象的操作方法，会依次调用叶子组件对象的操作方法来展示整个组合对象的行为。

组合设计模式的应用场景：

**树形结构**：当需要表示对象的层次结构，且对象之间存在“整体-部分”的关系时，可以使用组合模式来构建树形结构，方便对整个结构进行统一操作。

**GUI界面中的布局**：在图形用户界面（GUI）的布局中，可以使用组合模式来构建复杂的布局结构，通过将容器组件和子组件组合起来，方便进行整体和部分的布局调整。

**文件系统的管理**：在文件系统中，文件和文件夹之间存在层次关系，可以使用组合模式来构建文件系统的层次结构，方便对文件和文件夹进行统一的管理和操作。

**组织机构的管理**：在组织机构中，部门和员工之间存在层次关系，可以使用组合模式来构建组织机构的层次结构，方便对整个组织进行统一的管理和操作。

总之，组合设计模式适用于需要**构建树形结构、处理整体和部分的关系**，以及统一操作整个结构的场景。通过将对象组织成树形结构，可以简化客户端的代码，使得客户端可以一致地处理单个对象和组合对象。

### 10 你对适配器设计模式的理解：

适配器设计模式（Adapter Design Pattern）是一种结构型设计模式，用于将一个类的接口转换为另一个类的接口，以满足客户端的需求。适配器模式允许不兼容的类能够合作，通过适配器将一个类的接口转换为另一个类的接口，使得两者可以无缝协同工作。

适配器模式通常包含以下三个角色：目标接口（Target）、适配器（Adapter）和被适配者（Adaptee）。目标接口是客户端所期望的接口形式；适配器是将被适配者的接口转换为目标接口的类；被适配者是需要被适配的类。

下面是一个简单的C++代码示例来说明适配器设计模式的实现

|  |
| --- |
| #include <iostream>  // 目标接口  class Target {  public:  virtual void request() = 0;  };  // 被适配者类  class Adaptee {  public:  void specificRequest() {  *std*::*cout* << "被适配者的特殊请求" << *std*::*endl*;  }  };  // 类适配器  class Adapter : public Target, private Adaptee {  public:  void request() override {  specificRequest();  }  };  int main() {  Target\* **adapter** = new Adapter();  **adapter**->request();  delete **adapter**;  return 0;  } |

在上面的示例中，Target 是目标接口，定义了客户端所期望的接口形式。Adaptee 是被适配者类，具有客户端无法直接使用的特殊接口。Adapter 是适配器类，通过继承被适配者类和实现目标接口的方法，将被适配者类的接口转换为目标接口。

在 main 函数中，我们创建了一个适配器对象 adapter，然后调用它的 request 方法来展示适配器将被适配者的特殊请求转换为目标接口的能力。

适配器设计模式的应用场景包括但不限于以下情况：

**老旧接口的使用**：当需要使用一些老旧接口的功能，但又希望与现有的代码协同工作时，可以使用适配器模式来将老旧接口转换为现有代码所期望的接口形式。

**类库的兼容性**：当需要使用某个类库，但该类库的接口与当前项目的接口不兼容时，可以使用适配器模式来将类库的接口转换为项目所期望的接口形式。

**接口的统一**：当多个类具有不同的接口，但需要以统一的方式进行操作时，可以使用适配器模式来将它们的接口转换为统一的接口形式。

**与第三方组件的集成**：当需要将第三方组件集成到自己的系统中，但第三方组件的接口与自己的系统不兼容时，可以使用适配器模式来将第三方组件的接口转换为自己系统所期望的接口形式。

总之，适配器设计模式可以帮助不兼容的类能够协同工作，通过适配器将一个类的接口转换为另一个类的接口。适配器模式在实际开发中经常用于兼容性问题，以及将现有类库或组件集成到系统中。

### 11 代理设计模式的理解：

代理设计模式（Proxy Design Pattern）是一种结构型设计模式，用于在访问对象时提供一种代理，以控制对对象的访问。**代理模式通过引入一个代理对象来替代原始对象，从而可以在不改变原始对象的情况下增加额外的功能或控制访问**。

代理模式通常包含以下三个角色：抽象主题（Subject）、真实主题（Real Subject）和代理（Proxy）。抽象主题定义了真实主题和代理的共同接口；真实主题是实际执行业务逻辑的对象；代理是一个中间类，通过持有真实主题的引用并实现抽象主题的方法，将客户端的请求委派给真实主题。

下面是一个简单的C++代码示例来说明代理设计模式的实现：

|  |
| --- |
| #include <iostream>  // 抽象主题  class Subject {  public:  virtual void request() = 0;  };  // 真实主题  class RealSubject : public Subject {  public:  void request() override {  *std*::*cout* << "真实主题的请求" << *std*::*endl*;  }  };  // 代理  class Proxy : public Subject {  private:  RealSubject\* **realSubject**;  public:  Proxy() {  **realSubject** = new RealSubject();  }  \~Proxy() {  delete **realSubject**;  }  void request() override {  // 在调用真实主题之前可以添加额外的逻辑  *std*::*cout* << "代理的请求" << *std*::*endl*;  // 委托给真实主题处理  **realSubject**->request();  }  };  int main() {  Subject\* **proxy** = new Proxy();  **proxy**->request();  delete **proxy**;  return 0;  } |

在上面的示例中，Subject 是抽象主题，定义了真实主题和代理的共同接口。RealSubject 是真实主题，实现了抽象主题的方法，用于执行实际的业务逻辑。Proxy 是代理类，持有真实主题的引用，并实现抽象主题的方法，通过委托真实主题来处理客户端的请求。

在 main 函数中，我们创建了一个代理对象 proxy，然后调用它的 request 方法来展示代理将请求委托给真实主题的能力。

代理设计模式的应用场景包括但不限于以下情况：

**远程代理**：当需要访问远程对象时，可以使用代理模式来隐藏底层的网络通信细节，客户端通过代理对象访问远程对象。

**虚拟代理**：当创建一个对象的成本很高时，可以使用代理模式来延迟对象的实际创建，只有在真正需要时才创建真实对象。

**安全代理**：当需要对对象的访问进行控制时，可以使用代理模式来添加额外的安全性检查，例如权限验证等。

**缓存代理**：当需要缓存对象的结果以提高性能时，可以使用代理模式来实现缓存功能，代理对象可以在调用真实对象之前检查缓存，并返回缓存结果。

**日志记录代理**：当需要记录对象的操作日志时，可以使用代理模式来在调用真实对象之前或之后添加日志记录的功能。

总之，代理设计模式提供了一种控制对对象访问的方式，通过引入一个代理对象来替代原始对象，可以增加额外的功能或控制访问。代理模式在实际开发中经常用于远程访问、延迟加载、安全性控制、性能优化等场景。

### 12 装饰者设计模式的理解：

装饰者设计模式（Decorator Design Pattern）是一种结构型设计模式，**用于动态地给对象添加额外的职责，同时又不改变其原始类的结构**。装饰者模式通过将对象包装在一个装饰者类中，然后逐层地添加装饰者，从而实现对对象的透明扩展。

装饰者模式通常包含以下几个角色：**抽象构件（Component）、具体构件（Concrete Component）、抽象装饰者（Decorator）和具体装饰者（Concrete Decorator）**。抽象构件定义了原始对象和装饰者共同的接口；具体构件是原始对象，实现了抽象构件的接口；抽象装饰者是装饰者的抽象类或接口，包含一个指向抽象构件的引用；具体装饰者是具体的装饰者类，继承自抽象装饰者，通过对抽象构件进行装饰。

下面是一个简单的C++代码示例来说明装饰者设计模式的实现：

|  |
| --- |
| #include <iostream>  // 抽象构件  class Component {  public:  virtual void operation() = 0;  };  // 具体构件  class ConcreteComponent : public Component {  public:  void operation() override {  *std*::*cout* << "执行具体构件的操作" << *std*::*endl*;  }  };  // 抽象装饰者  class Decorator : public Component {  protected:  Component\* **component**;  public:  Decorator(Component\* **component**) : **component**(**component**) {}  void operation() override {  if (**component** != nullptr) {  **component**->operation();  }  }  };  // 具体装饰者A  class ConcreteDecoratorA : public Decorator {  public:  ConcreteDecoratorA(Component\* **component**) : Decorator(**component**) {}  void operation() override {  addedBehavior();  Decorator::operation();  }  void addedBehavior() {  *std*::*cout* << "具体装饰者A的附加操作" << *std*::*endl*;  }  };  // 具体装饰者B  class ConcreteDecoratorB : public Decorator {  public:  ConcreteDecoratorB(Component\* **component**) : Decorator(**component**) {}  void operation() override {  addedBehavior();  Decorator::operation();  }  void addedBehavior() {  *std*::*cout* << "具体装饰者B的附加操作" << *std*::*endl*;  }  };  int main() {  Component\* **component** = new ConcreteComponent();  Component\* **decoratorA** = new ConcreteDecoratorA(**component**);  **decoratorA**->operation();  Component\* **decoratorB** = new ConcreteDecoratorB(**decoratorA**);  **decoratorB**->operation();  delete **decoratorB**;  delete **decoratorA**;  delete **component**;  return 0;  } |

在上面的示例中，Component 是抽象构件，定义了原始对象和装饰者共同的接口。ConcreteComponent 是具体构件，实现了抽象构件的接口，表示原始对象。Decorator 是抽象装饰者，包含一个指向抽象构件的引用，并实现了抽象构件的接口。ConcreteDecoratorA 和 ConcreteDecoratorB 是具体装饰者，继承自抽象装饰者，通过对抽象构件进行装饰。

在 main 函数中，我们创建了一个具体构件 component，然后分别用具体装饰者 ConcreteDecoratorA 和 ConcreteDecoratorB 对其进行装饰。最终，通过调用装饰者的 operation 方法，实现了对原始对象的透明扩展。

装饰者设计模式的应用场景包括但不限于以下情况：

1 动态增加功能：当需要动态地给对象添加额外的职责时，可以使用装饰者模式。通过创建不同的装饰者类，可以在不改变原始对象的情况下，透明地为对象添加新的功能。

2. 多层次的装饰：当需要对对象进行多层次的装饰时，可以使用装饰者模式。每个具体装饰者可以在上一层装饰的基础上添加新的功能。

3. 组合功能：当需要将多个不同的功能组合在一起使用时，可以使用装饰者模式。通过创建不同的具体装饰者类，可以按需组合功能。

4. 开放封闭原则：当希望在不修改现有代码的情况下扩展功能时，可以使用装饰者模式。通过创建新的具体装饰者类，可以在不修改原始对象的情况下添加新的功能。

5. 单一职责原则：当希望将不同的责任分离到不同的类中，并且可以动态地进行组合时，可以使用装饰者模式。

总之，装饰者设计模式提供了一种动态地给对象添加额外职责的方式，同时又不改变其原始类的结构。装饰者模式在实际开发中经常用于动态增加功能、多层次的装饰、组合功能等场景。

### 13 桥接设计模式的理解：

桥接设计模式（Bridge Design Pattern）是一种结构型设计模式，用于将抽象部分与其具体实现部分分离，使它们可以独立地变化。桥接模式通过将抽象和实现通过一个桥接接口进行连接，使得它们可以独立地扩展和变化，而不会相互影响。

桥接模式通常包含以下几个角色：抽象部分（Abstraction）、具体抽象部分（Concrete Abstraction）、实现部分（Implementor）和具体实现部分（Concrete Implementor）。抽象部分定义了抽象接口，并包含一个对实现部分的引用；具体抽象部分是抽象部分的具体实现；实现部分定义了实现接口，并提供基本的操作；具体实现部分是实现部分的具体实现。

下面是一个简单的C++代码示例来说明桥接设计模式的实现：

|  |
| --- |
| #include <iostream>  // 实现部分  class Implementor {  public:  virtual void operationImpl() = 0;  };  // 具体实现部分A  class ConcreteImplementorA : public Implementor {  public:  void operationImpl() override {  *std*::*cout* << "具体实现部分A的操作" << *std*::*endl*;  }  };  // 具体实现部分B  class ConcreteImplementorB : public Implementor {  public:  void operationImpl() override {  *std*::*cout* << "具体实现部分B的操作" << *std*::*endl*;  }  };  // 抽象部分  class Abstraction {  protected:  Implementor\* **implementor**;  public:  Abstraction(Implementor\* **implementor**) : **implementor**(**implementor**) {}  virtual void operation() = 0;  };  // 具体抽象部分A  class ConcreteAbstractionA : public Abstraction {  public:  ConcreteAbstractionA(Implementor\* **implementor**) : Abstraction(**implementor**) {}  void operation() override {  *std*::*cout* << "具体抽象部分A的操作" << *std*::*endl*;  **implementor**->operationImpl();  }  };  // 具体抽象部分B  class ConcreteAbstractionB : public Abstraction {  public:  ConcreteAbstractionB(Implementor\* **implementor**) : Abstraction(**implementor**) {}  void operation() override {  *std*::*cout* << "具体抽象部分B的操作" << *std*::*endl*;  **implementor**->operationImpl();  }  };  int main() {  Implementor\* **implementorA** = new ConcreteImplementorA();  Abstraction\* **abstractionA** = new ConcreteAbstractionA(**implementorA**);  **abstractionA**->operation();  Implementor\* **implementorB** = new ConcreteImplementorB();  Abstraction\* **abstractionB** = new ConcreteAbstractionB(**implementorB**);  **abstractionB**->operation();  delete **abstractionB**;  delete **implementorB**;  delete **abstractionA**;  delete **implementorA**;  return 0;  } |

在上面的示例中，Implementor 是实现部分，定义了实现的接口。ConcreteImplementorA 和 ConcreteImplementorB 是具体实现部分，分别实现了 Implementor 的接口。

Abstraction 是抽象部分，定义了抽象接口，并包含一个对实现部分的引用。ConcreteAbstractionA 和 ConcreteAbstractionB 是具体抽象部分，继承自 Abstraction，实现了抽象部分的接口。

在 main 函数中，我们创建了一个具体实现部分 ConcreteImplementorA，然后用具体抽象部分 ConcreteAbstractionA 对其进行桥接。最终，通过调用抽象部分的 operation 方法，实现了对实现部分的透明调用。

桥接设计模式的应用场景包括但不限于以下情况：

1 当需要在抽象部分和实现部分之间进行解耦，使它们可以独立地变化时，可以使用桥接模式。

2. 当需要在运行时切换不同的实现部分时，可以使用桥接模式。通过更换具体实现部分，可以动态改变系统的行为。

3. 当一个类存在多个变化维度时，可以使用桥接模式。通过将每个变化维度抽象成抽象部分和实现部分，可以灵活地组合不同的变化维度。

4. 当需要对抽象部分和实现部分进行独立扩展时，可以使用桥接模式。通过增加新的具体实现部分或具体抽象部分，可以扩展系统的功能。

总之，桥接设计模式**通过将抽象部分和实现部分分离，使它们可以独立地变化。桥接模式在实际开发中经常用于解耦、运行时切换实现、处理多个变化维度等场景**。

### 14 模板方法设计模式的概念

模板方法设计模式（Template Method Pattern）是一种行为型设计模式，用于定义一个算法的骨架，而将一些步骤的具体实现延迟到子类中。这种模式允许在不改变算法结构的情况下，通过子类重写或扩展某些步骤来定制算法的行为。

核心思想： 模板方法模式将一个算法的步骤模板化，定义在一个抽象的基类中，其中一些步骤由抽象方法或默认方法实现，而其他步骤则由具体子类实现。

主要组成部分：

**模板类（Abstract Class）**： 定义了算法的骨架，通常包含一个或多个抽象方法，代表算法中的不同步骤。同时，也可以包含具体的方法来处理一些通用的逻辑。

**具体子类（Concrete Subclasses）**： 继承自模板类，实现模板中的抽象方法，以完成特定步骤的具体实现。每个具体子类可以定制算法的不同部分。

特点：

定义了一个算法的框架，具体步骤由子类实现。

遵循开闭原则，即允许在不修改模板代码的情况下扩展和变化算法的某些部分。

促使代码重用，将通用行为提取到模板方法中。

示例：

假设你正在开发一个烹饪应用程序，其中有不同的食谱，如制作咖啡和茶。你可以使用模板方法模式来定义一个通用的烹饪算法

|  |
| --- |
| // 模板类 - 食谱  class Recipe {  public:  // 模板方法，定义烹饪算法的骨架  void cook() {  boilWater();  brew();  pourInCup();  if (customerWantsCondiments()) {  addCondiments();  }  }  // 抽象方法，由子类实现  virtual void brew() = 0;  virtual void addCondiments() = 0;  // 钩子方法，子类可以选择性地覆盖  virtual bool customerWantsCondiments() {  return true;  }  // 具体方法，通用逻辑  void boilWater() {  *std*::*cout* << "烧开水" << *std*::*endl*;  }  void pourInCup() {  *std*::*cout* << "倒入杯中" << *std*::*endl*;  }  };  // 具体子类 - 制作咖啡  class CoffeeRecipe : public Recipe {  public:  void brew() override {  *std*::*cout* << "冲泡咖啡" << *std*::*endl*;  }  void addCondiments() override {  *std*::*cout* << "加入糖和牛奶" << *std*::*endl*;  }  };  // 具体子类 - 制作茶  class TeaRecipe : public Recipe {  public:  void brew() override {  *std*::*cout* << "浸泡茶叶" << *std*::*endl*;  }  void addCondiments() override {  *std*::*cout* << "加入柠檬" << *std*::*endl*;  }  }; |

在上面的示例中，Recipe 类定义了一个通用的烹饪算法框架，而具体的烹饪步骤由子类（CoffeeRecipe 和 TeaRecipe）来实现。这使得你可以轻松添加新的食谱，而不必更改通用的烹饪算法。

**应用场景**：

模板方法模式通常在以下情况下使用：

当你有一个通用的算法骨架，但不同部分的具体实现可能会变化时。

当你想要确保一些步骤在整个算法中保持一致性，而其他步骤由子类提供不同实现时。

当你希望通过子类来定制算法的某些部分，同时保留算法的整体结构。

当你需要遵循开闭原则，以便在不修改现有代码的情况下添加新的算法变种。

### 15 访问者设计模式的理解：

访问者设计模式（Visitor Design Pattern）是一种行为型设计模式，用于将**数据结构与对数据的操作分离**。访问者模式允许在不更改数据结构的情况下定义新的操作。

访问者模式通常包含以下几个角色：访问者（Visitor）、具体访问者（Concrete Visitor）、元素（Element）、具体元素（Concrete Element）和对象结构（Object Structure）。

访问者模式的核心思想是将**数据结构的遍历和操作分离开来**。访问者可以根据需求定义不同的具体访问者，每个具体访问者实现不同的操作。元素是数据结构的抽象，定义了接受访问者的接口。具体元素是元素的具体实现。对象结构是数据结构的集合，负责遍历元素并将访问者应用于每个元素。、

下面是一个简单的C++代码示例来说明访问者设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <vector>  // 前向声明  class ConcreteElementA;  class ConcreteElementB;  // 访问者接口  class Visitor {  public:  virtual void visit(ConcreteElementA\* **element**) = 0;  virtual void visit(ConcreteElementB\* **element**) = 0;  };  // 具体访问者A  class ConcreteVisitorA : public Visitor {  public:  void visit(ConcreteElementA\* **element**) override {  *std*::*cout* << "具体访问者A访问具体元素A，执行操作A" << *std*::*endl*;  }  void visit(ConcreteElementB\* **element**) override {  *std*::*cout* << "具体访问者A访问具体元素B，执行操作B" << *std*::*endl*;  }  };  // 具体访问者B  class ConcreteVisitorB : public Visitor {  public:  void visit(ConcreteElementA\* **element**) override {  *std*::*cout* << "具体访问者B访问具体元素A，执行操作C" << *std*::*endl*;  }  void visit(ConcreteElementB\* **element**) override {  *std*::*cout* << "具体访问者B访问具体元素B，执行操作D" << *std*::*endl*;  }  };  // 元素接口  class Element {  public:  virtual void accept(Visitor\* **visitor**) = 0;  };  // 具体元素A  class ConcreteElementA : public Element {  public:  void accept(Visitor\* **visitor**) override {  **visitor**->visit(this);  }  };  // 具体元素B  class ConcreteElementB : public Element {  public:  void accept(Visitor\* **visitor**) override {  **visitor**->visit(this);  }  };  // 对象结构  class ObjectStructure {  private:  *std*::*vector*<Element\*> **elements**;  public:  void addElement(Element\* **element**) {  **elements**.*push\_back*(**element**);  }  void removeElement(Element\* **element**) {  // 移除元素的代码  }  void accept(Visitor\* **visitor**) {  for (Element\* **element** : **elements**) {  **element**->accept(**visitor**);  }  }  };  int main() {  ObjectStructure **objectStructure**;  **objectStructure**.addElement(new ConcreteElementA());  **objectStructure**.addElement(new ConcreteElementB());  Visitor\* **visitorA** = new ConcreteVisitorA();  Visitor\* **visitorB** = new ConcreteVisitorB();  **objectStructure**.accept(**visitorA**);  **objectStructure**.accept(**visitorB**);  delete **visitorA**;  delete **visitorB**;  return 0;  } |

在上面的示例中，Visitor 是访问者接口，定义了对具体元素的访问方法。ConcreteVisitorA 和 ConcreteVisitorB 是具体访问者，分别实现了对具体元素的不同操作。

Element 是元素接口，定义了接受访问者的方法。ConcreteElementA 和 ConcreteElementB 是具体元素，实现了元素接口。

ObjectStructure 是对象结构，负责维护元素集合，并提供遍历元素的方法。

在 main 函数中，我们创建了

一个对象结构 `objectStructure`，并向其中添加了具体元素 `ConcreteElementA` 和 `ConcreteElementB`。然后我们创建了具体访问者 `ConcreteVisitorA` 和 `ConcreteVisitorB`。

通过调用 `objectStructure` 的 `accept` 方法，并传入具体访问者，实现了对元素的遍历和对应操作的执行。

访问者设计模式的应用场景包括但不限于以下情况：

1. 当存在一个复杂的对象结构，且需要对该结构进行不同的操作时，可以使用访问者模式。通过将操作封装在具体访问者中，可以实现对不同操作的灵活扩展。

2. 当不希望在元素类中添加新的操作，或者需要对元素进行操作的算法具有不同的变化时，可以使用访问者模式。通过将操作封装在具体访问者中，可以在不更改元素类的情况下定义新的操作。

3. 当需要对元素进行组合访问，且元素类的结构稳定，但操作算法的变化频繁时，可以使用访问者模式。通过访问者模式，可以将不同的操作逻辑抽离出来，避免对元素类的修改。

总之，访问者设计模式通过将数据结构与对数据的操作分离，实现了数据结构和操作的解耦。访问者模式在实际开发中常用于处理复杂的对象结构，实现对不同操作的灵活扩展。

### 16 责任链设计模式的理解：

责任链设计模式（Chain of Responsibility Design Pattern）是一种行为型设计模式，用于将请求的发送者和接收者解耦，并使多个对象都有机会处理该请求。请求沿着对象链进行传递，直到有一个对象能够处理它为止。

责任链模式通常包含以下几个角色：抽象处理器（Handler）、具体处理器（Concrete Handler）和客户端（Client）。

责任链模式的核心思想是将请求发送者和接收者解耦，每个处理器都有一个指向下一个处理器的引用。当请求到达处理器时，它可以选择处理请求并结束，或者将请求传递给下一个处理器。这样，请求会依次在处理器链中传递，直到有一个处理器能够处理它。

下面是一个简单的C++代码示例来说明责任链设计模式的实现：

|  |
| --- |
| #include <iostream>  // 抽象处理器  class Handler {  protected:  Handler\* **nextHandler**;  public:  Handler() : **nextHandler**(nullptr) {}  void setNextHandler(Handler\* **handler**) {  **nextHandler** = **handler**;  }  virtual void handleRequest(int **request**) = 0;  };  // 具体处理器A  class ConcreteHandlerA : public Handler {  public:  void handleRequest(int **request**) override {  if (**request** >= 0 && **request** < 10) {  *std*::*cout* << "具体处理器A处理请求：" << **request** << *std*::*endl*;  }  else if (**nextHandler** != nullptr) {  **nextHandler**->handleRequest(**request**);  }  }  };  // 具体处理器B  class ConcreteHandlerB : public Handler {  public:  void handleRequest(int **request**) override {  if (**request** >= 10 && **request** < 20) {  *std*::*cout* << "具体处理器B处理请求：" << **request** << *std*::*endl*;  }  else if (**nextHandler** != nullptr) {  **nextHandler**->handleRequest(**request**);  }  }  };  // 具体处理器C  class ConcreteHandlerC : public Handler {  public:  void handleRequest(int **request**) override {  if (**request** >= 20 && **request** < 30) {  *std*::*cout* << "具体处理器C处理请求：" << **request** << *std*::*endl*;  }  else if (**nextHandler** != nullptr) {  **nextHandler**->handleRequest(**request**);  }  }  };  int main() {  Handler\* **handlerA** = new ConcreteHandlerA();  Handler\* **handlerB** = new ConcreteHandlerB();  Handler\* **handlerC** = new ConcreteHandlerC();  **handlerA**->setNextHandler(**handlerB**);  **handlerB**->setNextHandler(**handlerC**);  **handlerA**->handleRequest(5);  **handlerA**->handleRequest(15);  **handlerA**->handleRequest(25);  delete **handlerA**;  delete **handlerB**;  delete **handlerC**;  return 0;  } |

在上面的示例中，Handler 是抽象处理器，定义了处理请求的接口和一个指向下一个处理器的引用。ConcreteHandlerA、ConcreteHandlerB 和 ConcreteHandlerC 是具体处理器，实现了处理请求的具体逻辑。

在 main 函数中，我们创建了具体处理器 handlerA、handlerB 和 handlerC，并通过调用 setNextHandler 方法将它们链接起来形成责任链。然后我们通过调用 handleRequest 方法来发送请求，并观察请求是如何在责任链中传递和处理的。

责任链设计模式的应用场景包括但不限于以下情况：

当需要将请求发送者和接收者解耦，并且有多个对象都有机会处理请求时，可以使用责任链模式。通过建立一个处理器链，每个处理器可以选择处理请求或将请求传递给下一个处理器。

当希望动态地指定处理请求的对象集合时，可以使用责任链模式。通过动态地调整处理器链，可以实现灵活的请求处理。

当需要按顺序处理请求，并且每个处理器都有不同的处理逻辑时，可以使用责任链模式。

通过责任链模式，可以将请求沿着处理器链进行传递，每个处理器都有机会处理请求或者将其传递给下一个处理器。这样可以实现请求的逐级处理，直到有一个处理器能够处理该请求为止。

例如，在一个电商网站中，可以使用责任链模式来处理用户的退款请求。假设有三个处理器，分别是财务部门、客服部门和仓储部门。当用户发起退款请求时，请求会依次经过这三个处理器。财务部门负责审核退款申请，客服部门负责与用户沟通并确认退款原因，仓储部门负责处理退货物品。如果某个处理器无法处理该请求，它会将请求传递给下一个处理器，直到有一个处理器能够处理该请求或处理器链结束。

责任链设计模式的优点包括：

1. 解耦发送者和接收者：责任链模式使得发送者不需要知道请求将由哪个处理器来处理，将请求和处理器解耦。

2. 灵活性和可扩展性：可以动态地调整处理器链，增加或移除处理器，以满足不同的处理需求。

3. 可以避免请求的发送者与接收者之间的紧耦合关系：将请求发送给处理器链，每个处理器都有机会处理请求，避免了发送者和接收者之间的直接依赖关系。

总之，**责任链设计模式通过将请求发送者和接收者解耦**，实现了对请求的逐级处理。它在实际开发中常用于处理请求的多级处理场景，提高了代码的灵活性和可扩展性。

### 17 中介者设计模式的理解：

中介者设计模式（Mediator Design Pattern）是一种行为型设计模式，用于降低多个对象之间的耦合性。中介者模式通过引入一个中介者对象，将对象间的交互集中管理，从而避免了对象之间的直接通信。

中介者模式通常包含以下几个角色：中介者（Mediator）、具体中介者（Concrete Mediator）和同事类（Colleague）。

中介者模式的核心思想是将对象间的交互通过中介者来进行协调和控制。当一个对象需要与其他对象进行通信时，它不直接与其他对象进行交互，而是通过中介者来进行通信。中介者负责接收和分发消息，并协调各个对象之间的交互。

下面是一个简单的C++代码示例来说明中介者设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <string>  class Colleague;  // 中介者抽象类  class Mediator {  public:  virtual void sendMessage(const *std*::*string*& **message**, Colleague\* **colleague**) = 0;  };  // 具体中介者类  class ConcreteMediator : public Mediator {  private:  Colleague\* **colleague1**;  Colleague\* **colleague2**;  public:  void setColleague1(Colleague\* **colleague**) {  **colleague1** = **colleague**;  }  void setColleague2(Colleague\* **colleague**) {  **colleague2** = **colleague**;  }  void sendMessage(const *std*::*string*& **message**, Colleague\* **colleague**) override {  if (**colleague** == **colleague1**) {  **colleague2**->receiveMessage(**message**);  }  else if (**colleague** == **colleague2**) {  **colleague1**->receiveMessage(**message**);  }  }  };  // 同事类  class Colleague {  protected:  Mediator\* **mediator**;  public:  void setMediator(Mediator\* **mediator**) {  this->**mediator** = **mediator**;  }  virtual void sendMessage(const *std*::*string*& **message**) = 0;  virtual void receiveMessage(const *std*::*string*& **message**) = 0;  };  // 具体同事类A  class ConcreteColleagueA : public Colleague {  public:  void sendMessage(const *std*::*string*& **message**) override {  **mediator**->sendMessage(**message**, this);  }  void receiveMessage(const *std*::*string*& **message**) override {  *std*::*cout* << "同事类A收到消息：" << **message** << *std*::*endl*;  }  };  // 具体同事类B  class ConcreteColleagueB : public Colleague {  public:  void sendMessage(const *std*::*string*& **message**) override {  **mediator**->sendMessage(**message**, this);  }  void receiveMessage(const *std*::*string*& **message**) override {  *std*::*cout* << "同事类B收到消息：" << **message** << *std*::*endl*;  }  };  int main() {  Mediator\* **mediator** = new ConcreteMediator();  Colleague\* **colleague1** = new ConcreteColleagueA();  Colleague\* **colleague2** = new ConcreteColleagueB();  **mediator**->setColleague1(**colleague1**);  **mediator**->setColleague2(**colleague2**);  **colleague1**->setMediator(**mediator**);  **colleague2**->setMediator(**mediator**);  **colleague1**->sendMessage("Hello, Colleague B!");  **colleague2**->sendMessage("Hi, Colleague A!");  delete **mediator**;  delete **colleague1**;  delete **colleague2**;  return 0;  } |

在上面的示例中，Mediator 是中介者抽象类，定义了中介者的接口。ConcreteMediator 是具体中介者类，实现了中介者的具体逻辑。Colleague 是同事类，定义了同事类的接口。ConcreteColleagueA 和 ConcreteColleagueB 是具体同事类，实现了同事类的具体逻辑。

在 main 函数中，我们创建了一个具体中介者对象 mediator，以及两个具体同事类对象 colleague1 和 colleague2。然后通过调用相应的方法进行对象之间的交互。

中介者设计模式的应用场景包括但不限于以下情况：

1 . 当对象间的通信过程需要集中管理和协调时，可以使用中介者模式。中介者模式可以将对象间的交互逻辑集中在中介者中，简化对象之间的通信。

2. 当对象间存在循环依赖关系时，可以使用中介者模式。中介者模式可以解决对象间的循环依赖问题，将复杂的关系转化为中介者与各个对象之间的简单关系。

3 当希望通过一个共享的中介者来减少系统中对象的数量时，可以使用中介者模式。中介者模式将对象间的一对多关系转化为一对一关系，减少了对象之间的直接耦合。

总之，中介者设计模式通过引入一个中介者对象来降低对象间的耦合性，将对象间的交互集中管理。它在实际开发中常用于需要集中管理对象间交互的场景，例如聊天室、调停者模式等。通过使用中介者模式，可以简化对象之间的通信，提高系统的灵活性和可扩展性。

### 18 状态设计模式的理解：

状态设计模式（State Design Pattern）是一种行为型设计模式，用于根据对象的内部状态改变其行为。状态模式将对象的行为封装在不同的状态类中，对象在不同状态下具有不同的行为。

状态模式通常包含以下几个角色：上下文（Context）、状态（State）和具体状态（Concrete State）。

上下文是拥有状态的对象，它维护一个指向当前状态对象的引用，并将请求委托给当前状态处理。上下文可以根据内部状态的改变来改变行为。

状态是一个抽象类或接口，定义了一个或多个在特定状态下的方法。这些方法将在具体状态类中被实现。

具体状态是状态的具体实现类，实现了在特定状态下的行为。

下面是一个简单的C++代码示例来说明状态设计模式的实现：

|  |
| --- |
| #include <iostream>  class Context;  // 状态抽象类  class State {  public:  virtual void handle(Context\* **context**) = 0;  };  // 具体状态类A  class ConcreteStateA : public State {  public:  void handle(Context\* **context**) override;  };  // 具体状态类B  class ConcreteStateB : public State {  public:  void handle(Context\* **context**) override;  };  // 上下文类  class Context {  private:  State\* **currentState**;  public:  Context(State\* **initialState**) : **currentState**(**initialState**) {}  void setState(State\* **state**) {  **currentState** = **state**;  }  void request() {  **currentState**->handle(this);  }  };  void ConcreteStateA::handle(Context\* **context**) {  *std*::*cout* << "当前状态是A，执行操作A，切换到状态B" << *std*::*endl*;  **context**->setState(new ConcreteStateB());  }  void ConcreteStateB::handle(Context\* **context**) {  *std*::*cout* << "当前状态是B，执行操作B，切换到状态A" << *std*::*endl*;  **context**->setState(new ConcreteStateA());  }  int main() {  Context\* **context** = new Context(new ConcreteStateA());  **context**->request(); // 输出：当前状态是A，执行操作A，切换到状态B  **context**->request(); // 输出：当前状态是B，执行操作B，切换到状态A  delete **context**;  return 0;  } |

在上面的示例中，State 是状态抽象类，定义了状态的接口方法 handle。ConcreteStateA 和 ConcreteStateB 是具体状态类，分别实现了在不同状态下的行为。Context 是上下文类，拥有一个当前状态的引用，并将请求委托给当前状态处理。

在 main 函数中，我们创建了一个上下文对象 context，并将初始状态设置为 ConcreteStateA。然后通过调用 context 的 request 方法来触发状态的改变。

状态设计模式的应用场景包括但不限于以下情况：

1当一个对象的行为取决于其内部状态，并且该对象在运行时需要根据状态改变行为时，可以使用状态模式。

2当一个操作具有多个不同的实现方式，每个实现方式对应一个状态时，可以使用状态模式。状态模式可以将不同的操作实现分离到不同的状态类中，避免了使用大量的条件语句。

3 当对象的行为在不同状态下会发生变化，并且状态转换的规则较为复杂时，可以使用状态模式。状态模式将状态转换的逻辑封装在状态类中，使得状态转换的逻辑更加清晰、可维护。

总之，状态设计模式通过将对象的行为封装在不同的状态类中，根据对象的内部状态来改变其行为。它在实际开发中常用于需要根据对象的状态来进行不同操作的场景，例如订单状态变化、游戏；

4. 当一个对象在不同状态下需要执行不同的操作，并且需要动态地切换状态时，可以使用状态模式。状态模式可以使对象的状态转换更加灵活，可以根据需要随时切换状态。

5. 当需要添加新的状态时，状态模式具有良好的扩展性。通过添加新的具体状态类，可以很容易地扩展系统的行为。

总之，状态设计模式通过将对象的行为封装在不同的状态类中，根据对象的内部状态来改变其行为。它在实际开发中常用于需要根据对象的状态来进行不同操作的场景，例如订单状态变化、游戏角色状态转换等。通过使用状态模式，可以使代码更加清晰、可维护，并提高系统的灵活性和可扩展性。

### 19 观察者设计模式的理解：

观察者设计模式（Observer Design Pattern）是一种行为型设计模式，用于建立对象之间的一对多依赖关系，当一个对象的状态发生变化时，所有依赖它的对象都会得到通知并自动更新。

观察者模式通常包含以下几个角色：主题（Subject）、观察者（Observer）和具体观察者（Concrete Observer）。

主题是被观察的对象，它维护一个观察者列表，并提供添加、删除和通知观察者的方法。

观察者是一个抽象类或接口，定义了一个或多个接收主题通知的方法。

具体观察者是观察者的具体实现类，实现了接收通知并做出相应处理的方法。

下面是一个简单的C++代码示例来说明观察者设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <vector>  class Observer;  // 主题抽象类  class Subject {  public:  virtual void attach(Observer\* **observer**) = 0;  virtual void detach(Observer\* **observer**) = 0;  virtual void notify() = 0;  };  // 观察者抽象类  class Observer {  public:  virtual void update() = 0;  };  // 具体观察者类A  class ConcreteObserverA : public Observer {  public:  void update() override {  *std*::*cout* << "具体观察者A收到通知并作出响应" << *std*::*endl*;  }  };  // 具体观察者类B  class ConcreteObserverB : public Observer {  public:  void update() override {  *std*::*cout* << "具体观察者B收到通知并作出响应" << *std*::*endl*;  }  };  // 具体主题类  class ConcreteSubject : public Subject {  private:  *std*::*vector*<Observer\*> **observers**;  public:  void attach(Observer\* **observer**) override {  **observers**.*push\_back*(**observer**);  }  void detach(Observer\* **observer**) override {  for (auto **it** = **observers**.*begin*(); **it** != **observers**.*end*(); ++**it**) {  if (\***it** == **observer**) {  **observers**.*erase*(**it**);  break;  }  }  }  void notify() override {  for (auto **observer** : **observers**) {  **observer**->update();  }  }  };  int main() {  ConcreteSubject **subject**;  Observer\* **observerA** = new ConcreteObserverA();  Observer\* **observerB** = new ConcreteObserverB();  **subject**.attach(**observerA**);  **subject**.attach(**observerB**);  **subject**.notify(); // 输出：具体观察者A收到通知并作出响应，具体观察者B收到通知并作出响应  **subject**.detach(**observerA**);  **subject**.notify(); // 输出：具体观察者B收到通知并作出响应  delete **observerA**;  delete **observerB**;  return 0;  } |

在上面的示例中，Subject 是主题的抽象类，定义了添加、删除和通知观察者的方法。Observer 是观察者的抽象类，定义了接收通知的方法。ConcreteObserverA 和 ConcreteObserverB 是具体观察者类，分别实现了接收通知并作出响应的方法。ConcreteSubject 是具体主题类，维护了一个观察者列表，并在状态变化时通知观察者。

在 main 函数中，我们创建了一个具体主题对象 subject，并创建了两个具体观察者对象 observerA 和 observerB。然后通过调用 subject 的 attach 方法将观察者添加到观察者列表中，再调用 subject 的 notify 方法发送通知。观察者收到通知后会作出相应的响应。

观察者设计模式的应用场景包括但不限于以下情况：

当一个对象的改变需要通知其他对象，并且不希望这些对象耦合在一起时，可以使用观察者模式。观察者模式可以将主题和观察者解耦，使它们可以独立地进行扩展和维护。

2. 当一个对象的状态改变需要影响其他多个对象，并且这些对象的数量和关系可能动态变化时，可以使用观察者模式。观察者模式可以方便地管理和维护多个观察者对象，实现对象之间的松耦合。

3. 当一个对象需要在特定情况下通知其他对象并触发一系列操作时，可以使用观察者模式。观察者模式可以将通知和操作封装在具体观察者中，实现更加灵活的处理。

4. 当一个对象需要与多个对象进行交互，并且对交互的顺序和方式有特定要求时，可以使用观察者模式。观察者模式可以通过定义不同的具体观察者，灵活地管理对象之间的交互。

总之，观察者设计模式通过建立对象之间的一对多依赖关系，实现了对象之间的松耦合。它在实际开发中常用于需要在对象状态变化时通知其他对象并作出相应处理的场景，例如事件驱动编程、GUI界面开发、消息传递等。通过使用观察者模式，可以提高代码的可维护性和扩展性，并实现对象之间的解耦。

### 20 策略设计模式的理解：

策略设计模式（Strategy Design Pattern）是一种行为型设计模式，它定义了一系列算法，并将每个算法封装到具体的策略类中，使得它们可以互相替换。通过使用策略模式，可以在运行时动态地选择算法，而不需要修改调用算法的代码。

策略模式通常包含以下几个角色：上下文（Context）、策略（Strategy）和具体策略（Concrete Strategy）。

上下文是使用策略的对象，它将具体的策略类传给客户端，并在需要时调用策略的方法。

策略是一个抽象类或接口，定义了算法的公共接口。

具体策略是策略的具体实现类，实现了算法的具体逻辑。

下面是一个简单的C++代码示例来说明策略设计模式的实现

|  |
| --- |
| #include <iostream>  // 策略抽象类  class Strategy {  public:  virtual void execute() = 0;  };  // 具体策略类A  class ConcreteStrategyA : public Strategy {  public:  void execute() override {  *std*::*cout* << "使用策略A执行算法" << *std*::*endl*;  }  };  // 具体策略类B  class ConcreteStrategyB : public Strategy {  public:  void execute() override {  *std*::*cout* << "使用策略B执行算法" << *std*::*endl*;  }  };  // 上下文类  class Context {  private:  Strategy\* **strategy**;  public:  void setStrategy(Strategy\* **strategy**) {  this->**strategy** = **strategy**;  }  void executeStrategy() {  if (**strategy**) {  **strategy**->execute();  }  }  };  int main() {  Context **context**;  // 使用策略A执行算法  ConcreteStrategyA **strategyA**;  **context**.setStrategy(&**strategyA**);  **context**.executeStrategy();  // 使用策略B执行算法  ConcreteStrategyB **strategyB**;  **context**.setStrategy(&**strategyB**);  **context**.executeStrategy();  return 0;  } |

在上面的示例中，Strategy 是策略的抽象类，定义了算法的执行接口。ConcreteStrategyA 和 ConcreteStrategyB 是具体策略类，分别实现了不同的算法。Context 是上下文类，用于使用策略执行算法。

在 main 函数中，我们创建了一个上下文对象 context，并通过调用 setStrategy 方法将具体策略对象传给上下文。然后调用 executeStrategy 方法执行算法，实际执行的算法取决于具体策略对象。

策略设计模式的应用场景包括但不限于以下情况：

1 当一个系统需要在不同时间应用不同的算法，并且希望能够灵活地切换算法时，可以使用策略模式。策略模式允许在运行时动态地选择算法，而不需要修改调用算法的代码。

2 当一个系统需要根据不同的条件选择不同的算法，并且这些条件可能会发生变化时，可以使用策略模式。策略模式可以通过定义不同的具体策略类来适应不同的条件，实现算法的灵活选择。

3 当一个系统的算法逻辑复杂，需要将不同的算法进行解耦和封装时，可以使用策略模式。策略模式将算法封装在具体策略类中，使得每个策略类只需要关注自己的算法逻辑，提高了代码的可读性和可维护性。

总之，策略设计模式通过将算法封装到具体的策略类中，实现了算法的可替换和动态选择。它在实际开发中常用于需要根据不同条件选择不同算法的场景，例如排序算法、计算策略、日志记录等。通过使用策略模式，可以提高代码的灵活性、可扩展性和可维护性。

补充说明：

在上述示例中，我们可以在 `Context` 类中添加一个方法 `void changeStrategy(Strategy\* newStrategy)`，用于在运行时切换策略。这样，在程序运行过程中可以动态地改变算法的行为，而不需要修改 `Context` 类的代码。

另外，策略模式还可以与工厂模式结合使用，通过工厂类来创建具体策略对象，并将其传递给上下文类。这样可以进一步降低上下文类对具体策略类的依赖性，增加系统的灵活性和可拓展性。

总结一下，**策略设计模式通过将算法封装到具体的策略类中**，实现了算法的可替换和动态选择。它在实际开发中常用于需要根据不同条件选择不同算法的场景，通过使用策略模式，可以提高代码的灵活性、可扩展性和可维护性。

### 21 解释器设计模式的理解：

解释器设计模式（Interpreter Design Pattern）是一种行为型设计模式，它定义了一种语言和解释器的结构，用于解释和执行特定的语法规则。该模式将语言的表达式表示为一个抽象语法树，并定义了一组解释器来解释和执行这些语法规则。

解释器模式通常包含以下几个角色：抽象表达式（Abstract Expression）、终结符表达式（Terminal Expression）、非终结符表达式（Non-terminal Expression）和上下文（Context）。

抽象表达式是一个抽象类或接口，定义了解释器的抽象方法。

终结符表达式表示语言中的终结符，它实现了抽象表达式的解释方法。

非终结符表达式表示语言中的非终结符，它通常由多个终结符表达式组成，并实现了抽象表达式的解释方法。

上下文类保存了解释器的全局信息，并提供给解释器访问和操作的接口。

下面是一个简单的C++代码示例来说明解释器设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <unordered\_map>  // 上下文类  class Context {  private:  *std*::*unordered\_map*<*std*::*string*, bool> **variables**;  public:  bool getVariable(const *std*::*string*& **name**) {  return **variables**[**name**];  }  void setVariable(const *std*::*string*& **name**, bool **value**) {  **variables**[**name**] = **value**;  }  };  // 抽象表达式类  class Expression {  public:  virtual bool interpret(Context& **context**) = 0;  };  // 终结符表达式类  class TerminalExpression : public Expression {  private:  *std*::*string* **variable**;  public:  TerminalExpression(const *std*::*string*& **variable**) : **variable**(**variable**) {}  bool interpret(Context& **context**) override {  return **context**.getVariable(**variable**);  }  };  // 非终结符表达式类  class AndExpression : public Expression {  private:  Expression\* **expression1**;  Expression\* **expression2**;  public:  AndExpression(Expression\* **expression1**, Expression\* **expression2**)  : **expression1**(**expression1**), **expression2**(**expression2**) {}  bool interpret(Context& **context**) override {  return **expression1**->interpret(**context**) && **expression2**->interpret(**context**);  }  };  // 非终结符表达式类  class OrExpression : public Expression {  private:  Expression\* **expression1**;  Expression\* **expression2**;  public:  OrExpression(Expression\* **expression1**, Expression\* **expression2**)  : **expression1**(**expression1**), **expression2**(**expression2**) {}  bool interpret(Context& **context**) override {  return **expression1**->interpret(**context**) || **expression2**->interpret(**context**);  }  };  int main() {  Context **context**;  **context**.setVariable("A", true);  **context**.setVariable("B", false);  Expression\* **expression1** = new TerminalExpression("A");  Expression\* **expression2** = new TerminalExpression("B");  // A AND B  Expression\* **andExpression** = new AndExpression(**expression1**, **expression2**);  bool **result** = **andExpression**->interpret(**context**);  *std*::*cout* << "A AND B = " << **result** << *std*::*endl*;  // A OR B  Expression\* **orExpression** = new OrExpression(**expression1**, **expression2**);  **result** = **orExpression**->interpret(**context**);  *std*::*cout* << "A OR B = " << **result** << *std*::*endl*;  delete **expression1**;  delete **expression2**;  delete **andExpression**;  delete **orExpression**;  return 0;  } |

在上述示例中，我们创建了一个简单的布尔表达式语言，并实现了解释器模式来解释和执行这些表达式。

Context 类保存了表达式的上下文信息，包括变量和对应的值。

Expression 是抽象表达式类，定义了解释器的抽象方法 interpret。

TerminalExpression 是终结符表达式类，表示语言中的终结符，它根据上下文中的变量值来解释和执行。

AndExpression 和 OrExpression 是非终结符表达式类，表示语言中的非终结结符，它们通过组合终结符表达式来实现复杂的表达式解释和执行。

在 `main` 函数中，我们创建了一个上下文对象 `context`，并设置了变量 A 和 B 的值。然后，我们创建了终结符表达式对象 `expression1` 和 `expression2`，分别表示变量 A 和 B。接着，我们创建了非终结符表达式对象 `andExpression` 和 `orExpression`，分别表示 A AND B 和 A OR B 的表达式。

最后，我们调用 `interpret` 方法来解释和执行这些表达式，并输出结果。

解释器设计模式的应用场景包括但不限于以下情况：

1. 当需要解释和执行一种特定语言或表达式时，可以使用解释器模式。例如，当需要解析和执行一种自定义查询语言、配置文件或规则引擎时，解释器模式可以帮助我们实现解释器，并执行相应的语法规则。

2. 当需要灵活地扩展语言的语法规则时，可以使用解释器模式。解释器模式通过定义抽象表达式、终结符表达式和非终结符表达式，使得我们可以轻松地添加新的语法规则，并解释和执行这些规则。

3. 当需要对复杂的逻辑进行解析和执行时，可以使用解释器模式。解释器模式可以将复杂的逻辑分解为简单的语法规则，并通过组合和嵌套表达式来实现逻辑的解释和执行。

总之，解释器设计模式通过定义一种语言和解释器的结构，实现了对特定语法规则的解释和执行。它在需要解释和执行特定语言或表达式、扩展语言的语法规则以及对复杂逻辑进行解析和执行的场景中有广泛的应用。

### 22 备忘录设计模式的理解：

备忘录设计模式（Memento Design Pattern）是一种行为型设计模式，用于捕获和保存对象的内部状态，以便在需要时能够恢复到之前的状态。它将对象的状态封装在备忘录对象中，同时提供了对备忘录的创建、恢复和管理。

备忘录设计模式通常包含以下几个角色：原发器（Originator）、备忘录（Memento）、负责人（Caretaker）。

原发器是需要保存状态的对象，它可以创建备忘录对象，将当前状态保存到备忘录中，或从备忘录中恢复之前的状态。

备忘录是用于保存原发器状态的对象，它通常提供了访问原发器状态的接口，但不允许其他对象修改备忘录的状态。

负责人是用于管理备忘录对象的对象，它保存了多个备忘录对象，并提供了对备忘录的管理操作，例如保存备忘录、获取备忘录等。

下面是一个简单的C++代码示例来说明备忘录设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <string>  // 备忘录类  class Memento {  private:  *std*::*string* **state**;  public:  Memento(const *std*::*string*& **state**) : **state**(**state**) {}  *std*::*string* getState() const {  return **state**;  }  };  // 原发器类  class Originator {  private:  *std*::*string* **state**;  public:  void setState(const *std*::*string*& **state**) {  this->**state** = **state**;  }  *std*::*string* getState() const {  return **state**;  }  Memento\* createMemento() const {  return new Memento(**state**);  }  void restoreMemento(const Memento\* **memento**) {  **state** = **memento**->getState();  }  };  // 负责人类  class Caretaker {  private:  Memento\* **memento**;  public:  void saveMemento(Memento\* **memento**) {  this->**memento** = **memento**;  }  Memento\* getMemento() const {  return **memento**;  }  };  int main() {  Originator **originator**;  Caretaker **caretaker**;  // 设置初始状态  **originator**.setState("State 1");  *std*::*cout* << "Current state: " << **originator**.getState() << *std*::*endl*;  // 保存备忘录  **caretaker**.saveMemento(**originator**.createMemento());  // 修改状态  **originator**.setState("State 2");  *std*::*cout* << "Current state: " << **originator**.getState() << *std*::*endl*;  // 恢复到之前的状态  **originator**.restoreMemento(**caretaker**.getMemento());  *std*::*cout* << "Current state: " << **originator**.getState() << *std*::*endl*;  return 0;  } |

在上述示例中，我们创建了一个备忘录设计模式的简单实现。Originator 类表示原发器，它具有一个状态 state，并提供了设置状态、获取状态、创建备忘录和恢复备忘录的方法。

Memento 类表示备忘录，它保存了原发器的状态，并提供了获取状态的方法。

Caretaker 类表示负责人，它保存了备忘录对象，并提供了保存备忘录和获取备忘录的方法。

在 main 函数中，我们创建了原发器对象 originator 和负责人对象 caretaker。然后，我们设置原发器的初始状态，并输出当前状态。接着，我们保存备忘录，并修改原发器的状态，并输出当前状态。最后，我们恢复到之前的状态，并输出当前状态，验证备忘录的恢复功能。

备忘录设计模式的应用场景包括但不限于以下情况：

当需要保存和恢复对象的内部状态时，可以使用备忘录模式。例如，在文本编辑器中，我们可以使用备忘录模式来保存编辑器的撤销和恢复操作，以便用户可以回到之前的编辑状态。

2. 当需要实现快照功能，保存对象的某个状态，以便后续可以随时恢复到该状态时，可以使用备忘录模式。例如，在游戏中，我们可以使用备忘录模式来保存游戏的进度，以便玩家可以在需要时恢复到之前的进度。

3. 当需要实现事务的回滚功能，即将一系列操作封装成一个事务，并在失败或取消时能够回滚到事务开始前的状态时，可以使用备忘录模式。例如，在数据库管理系统中，我们可以使用备忘录模式来实现事务的回滚功能。

4. 当需要实现多级撤销和恢复操作时，可以使用备忘录模式。备忘录模式可以保存多个备忘录对象，以便实现多级撤销和恢复的功能。

总之，备忘录设计模式通过将对象的内部状态封装在备忘录对象中，提供了对状态的保存、恢复和管理。它适用于需要保存和恢复对象状态、实现快照功能、实现事务回滚、实现多级撤销和恢复等场景。

### 23 命令设计模式的理解：

命令设计模式（Command Design Pattern）是一种行为型设计模式，用于将请求封装成一个对象，从而可以在不同的上下文中使用、传递和操作请求。它将请求的发送者和接收者解耦，使得发送者只需知道如何发送请求，而不需要知道请求是如何被执行和处理的。

命令设计模式通常包含以下几个角色：命令接口（Command）、具体命令（ConcreteCommand）、请求者（Invoker）、接收者（Receiver）。

命令接口是一个抽象接口，用于声明执行命令的方法。

具体命令是命令接口的具体实现，它将一个接收者对象和一组操作绑定在一起，并实现了执行命令的方法。

请求者是负责发送命令的对象，它持有一个命令对象，并在需要时调用命令对象的执行方法。

接收者是执行命令的对象，它实现了具体的操作逻辑，并在接收到命令时执行相应的操作。

下面是一个简单的C++代码示例来说明命令设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <string>  // 命令接口  class Command {  public:  virtual \~Command() {}  virtual void execute() = 0;  };  // 具体命令  class ConcreteCommand : public Command {  private:  *std*::*string* **message**;  public:  ConcreteCommand(const *std*::*string*& **message**) : **message**(**message**) {}  void execute() override {  *std*::*cout* << "Executing command: " << **message** << *std*::*endl*;  // 执行命令的具体操作  }  };  // 请求者  class Invoker {  private:  Command\* **command**;  public:  void setCommand(Command\* **command**) {  this->**command** = **command**;  }  void executeCommand() {  **command**->execute();  }  };  // 接收者  class Receiver {  public:  void performAction() {  *std*::*cout* << "Performing action" << *std*::*endl*;  // 执行接收者的操作  }  };  int main() {  Invoker **invoker**;  Receiver **receiver**;  Command\* **command** = new ConcreteCommand("Do something");  // 设置命令  **invoker**.setCommand(**command**);  // 执行命令  **invoker**.executeCommand();  // 执行接收者的操作  **receiver**.performAction();  delete **command**;  return 0;  } |

在上述示例中，我们创建了一个命令设计模式的简单实现。Command 类表示命令接口，它声明了执行命令的方法。

ConcreteCommand 类表示具体命令，它实现了命令接口，并将一个接收者对象和一组操作绑定在一起。

Invoker 类表示请求者，它持有一个命令对象，并在需要时调用命令对象的执行方法。

Receiver 类表示接收者，它实现了具体的操作逻辑，并在接收到命令时执行相应的操作。

在 main 函数中，我们创建了请求者对象 invoker 和接收者对象 receiver。然后，我们创建了具体命令对象，并将其设置到请求者对象中。接着，我们调用请求者对象的执行命令方法，触发命令的执行。最后，我们调用接收者对象的执行操作方法，执行接收者的操作逻辑。

命令设计模式的应用场景包括但不限于以下情况：

当需要将请求发送者和接收者解耦，使得它们可以独立变化时，可以使用命令模式。命令模式将请求封装成一个对象，使得请求的发送者只需要关注如何发送请求，而不需要知道请求是如何被执行和处理的。

当需要实现撤销、重做、事务等功能时，可以使用命令模式。命令模式可以将每个命令封装成一个对象，并保存在历史记录中，以便可以撤销和重做命令，或者将多个命令组合成一个事务进行执行。

3. 当需要实现日志记录、审计或回放功能时，可以使用命令模式。命令模式可以将执行的命令保存下来，以便后续可以进行日志记录、审计或回放操作。

4. 当需要实现任务调度和命令队列时，可以使用命令模式。命令模式可以将命令对象放入队列中，按照顺序执行，从而实现任务调度和命令队列的功能。

总之，命令设计模式通过将请求封装成一个对象，解耦了请求的发送者和接收者，使得请求的发送者只需要关注如何发送请求，而不需要知道请求是如何被执行和处理的。它适用于需要解耦请求发送者和接收者、实现撤销、重做、事务等功能、实现日志记录、审计或回放、实现任务调度和命令队列等场景。

### 24 迭代器设计模式的理解：

迭代器设计模式（Iterator Design Pattern）是一种行为型设计模式，用于提供一种统一的方式来访问集合对象中的元素，而无需暴露集合的内部表示。

迭代器设计模式通常包含以下几个角色：迭代器接口（Iterator）、具体迭代器（ConcreteIterator）、聚合接口（Aggregate）和具体聚合（ConcreteAggregate）。

迭代器接口是一个抽象接口，定义了访问和遍历集合元素的方法。

具体迭代器是迭代器接口的具体实现，实现了迭代器接口中的方法，并维护了一个指向集合中当前元素的指针。

聚合接口是一个抽象接口，定义了获取迭代器的方法。

具体聚合是聚合接口的具体实现，实现了聚合接口中的方法，并返回一个具体迭代器的实例。

下面是一个简单的C++代码示例来说明迭代器设计模式的实现：

|  |
| --- |
| #include <iostream>  #include <vector>  // 迭代器接口  template<class T>  class Iterator {  public:  virtual bool hasNext() = 0;  virtual T next() = 0;  };  // 具体迭代器  template<class T>  class ConcreteIterator : public Iterator<T> {  private:  *std*::*vector*<T>& **collection**;  int **index** = 0;  public:  ConcreteIterator(*std*::*vector*<T>& **collection**) : **collection**(**collection**) {}  bool hasNext() override {  return **index** < **collection**.*size*();  }  T next() override {  return **collection**[**index**++];  }  };  // 聚合接口  template<class T>  class Aggregate {  public:  virtual Iterator<T>\* createIterator() = 0;  };  // 具体聚合  template<class T>  class ConcreteAggregate : public Aggregate<T> {  private:  *std*::*vector*<T> **collection**;  public:  void add(T **item**) {  **collection**.*push\_back*(**item**);  }  Iterator<T>\* createIterator() override {  return new ConcreteIterator<T>(**collection**);  }  };  int main() {  ConcreteAggregate<int> **aggregate**;  **aggregate**.add(1);  **aggregate**.add(2);  **aggregate**.add(3);  Iterator<int>\* **iterator** = **aggregate**.createIterator();  while (**iterator**->hasNext()) {  *std*::*cout* << **iterator**->next() << " ";  }  delete **iterator**;  return 0;  } |

在上述示例中，我们创建了一个迭代器设计模式的简单实现。Iterator 类表示迭代器接口，它定义了访问和遍历集合元素的方法。

ConcreteIterator 类表示具体迭代器，它实现了迭代器接口，并维护了一个指向集合中当前元素的指针。

Aggregate 类表示聚合接口，它定义了获取迭代器的方法。

ConcreteAggregate 类表示具体聚合，它实现了聚合接口，并返回一个具体迭代器的实例。

在 main 函数中，我们创建了具体聚合对象 aggregate 并添加了一些元素。然后，我们通过调用 createIterator 方法获取迭代器，并使用迭代器遍历集合中的元素并打印。

迭代器设计模式的应用场景包括但不限于以下情况：

1当需要统一访问和遍历集合对象中的元素时，可以使用迭代器模式。迭代器模式提供了一种统一的方式来访问集合对象中的元素，而无需关注集合的内部表示和实现方式。

2当需要隐藏集合对象的内部结构，并提供一种安全的方式来遍历集合中的元素时，可以使用迭代器模式。迭代器模式将集合的遍历操作封装在迭代器中，使得外部代码无法直接访问集合的内部结构。

3当需要实现不同类型的集合对象的统一遍历方式时，可以使用迭代器模式。迭代器模式可以为每种类型的集合对象提供一个具体的迭代器实现，使得不同类型的集合对象都可以通过相同的方式进行遍历。

4. 当需要支持多种遍历方式或遍历顺序时，可以使用迭代器模式。迭代器模式可以为同一个集合对象提供多个不同的迭代器实现，每个迭代器都可以以不同的方式或顺序遍历集合。

总之，迭代器设计模式通过提供一个统一的方式来访问和遍历集合对象中的元素，隐藏了集合的内部结构，提供了安全的遍历方式，并支持多种遍历方式或顺序。它适用于需要统一访问和遍历集合对象、隐藏集合的内部结构、支持多种遍历方式或遍历顺序的场景。

# QT篇

### ****1 讲述Qt信号槽机制与优势与不足****

优点： ①类型安全。需要关联的信号槽的签名必须是等同的。即信号的参数类型和参数个数同接受该信号的槽的参数类型和参数个数相同。若信号和槽签名不一致，编译器会报错。

②松散耦合。信号和槽机制减弱了Qt对象的耦合度。激发信号的Qt对象无需知道是那个对象的那个信号槽接收它发出的信号，它只需在适当的时间发送适当的信号即可，而不需要关心是否被接受和那个对象接受了。Qt就保证了适当的槽得到了调用，即使关联的对象在运行时被删除。程序也不会奔溃。

③灵活性。一个信号可以关联多个槽，或多个信号关联同一个槽。

不足：速度较慢。与回调函数相比，信号和槽机制运行速度比直接调用非虚函数慢10倍。

原因：①需要定位接收信号的对象。②安全地遍历所有关联槽。③编组、解组传递参数。④多线程的时候，信号需要排队等待。（然而，与创建对象的new操作及删除对象的delete操作相比，信号和槽的运行代价只是他们很少的一部分。信号和槽机制导致的这点性能损耗，对实时应用程序是可以忽略的。）

### 2 Qt信号和槽的本质是什么？

 回调函数。信号或是传递值，或是传递动作变化；槽函数响应信号或是接收值，或者根据动作变化来做出对应操作。

### 3 描述QT中的文件流(QTextStream)和数据流(QDataStream)的区别

文件流(QTextStream)。操作轻量级数据（int,double,QString）数据写入文本件中以后以文本的方式呈现。

数据流(QDataStream)。通过数据流可以操作各种数据类型，包括对象，存储到文件中数据为二进制。

文件流，数据流都可以操作磁盘文件，也可以操作内存数据。通过流对象可以将对象打包到内存，进行数据的传输。

### 4、多线程使用使用方法

方法一：①创建一个类从QThread类派生②在子线程类中重写 run 函数, 将处理操作写入该函数中 ③在主线程中创建子线程对象, 启动子线程, 调用start()函数

方法二：①将业务处理抽象成一个业务类, 在该类中创建一个业务处理函数②在主线程中创建一QThread类对象 ③在主线程中创建一个业务类对象 ④将业务类对象移动到子线程中 ⑤在主线程中启动子线程 ⑥通过信号槽的方式, 执行业务类中的业务处理函数

多线程使用注意事项:

\* 1. 业务对象, 构造的时候不能指定父对象

\* 2. 子线程中不能处理ui窗口(ui相关的类)

\* 3. 子线程中只能处理一些数据相关的操作, 不能涉及窗口

### 5、多线程下，QT信号槽分别在什么线程中执行，如何控制

可以通过connect的第五个参数进行控制信号槽执行时所在的线程

　　connect有几种连接方式，直接连接和队列连接、自动连接

　　直接连接（Qt::DirectConnection）：信号槽在信号发出者所在的线程中执行

　　队列连接 (Qt::QueuedConnection)：信号在信号发出者所在的线程中执行，槽函数在信号接收者所在的线程中执行

　　自动连接 (Qt::AutoConnection)：多线程时为队列连接函数，单线程时为直接连接函数。

### 6 你做过QT自定义控件吗？能举几个例子吗？

从外观设计上：QSS、继承绘制函数重绘、继承QStyle相关类重绘、组合拼装等等

从功能行为上：重写事件函数、添加或者修改信号和槽等等

### 7 什么是QSS？QSS平时使用的多吗？能举几个例子吗？都是如何使用？

1.将QSS统一写在一个文件中，通过程序给主窗口加载；

2.写成一个字符串中，通过程序给主窗口加载；

3.需要使用的地方，写一个字符串，加载给对象；

4.QT Designer中填写；

事件机制：

QT程序是事件驱动的，事件到处都可以遇到。能说说平时经常使用到哪些事件吗？

常见的QT事件类型如下:

键盘事件: 按键按下和松开 鼠标事件: 鼠标移动,鼠标按键的按下和松开

拖放事件: 用鼠标进行拖放 滚轮事件: 鼠标滚轮滚动

绘屏事件: 重绘屏幕的某些部分 定时事件: 定时器到时

焦点事件: 键盘焦点移动 进入和离开事件: 鼠标移入widget之内,或是移出

移动事件: widget的位置改变 大小改变事件: widget的大小改变

显示和隐藏事件: widget显示和隐藏 窗口事件: 窗口是否为当前窗口

### 8 知道QT事件机制有几种级别的事件过滤吗？能大致描述下吗？

根据对Qt事件机制的分析, 可以得到5种级别的事件过滤,处理办法. 以功能从弱到强, 排列如下:

1）重载特定事件处理函数.

最常见的事件处理办法就是重载象mousePressEvent(), keyPressEvent(), paintEvent() 这样的特定事件处理函数.

2）重载event()函数.

通过重载event()函数,我们可以在事件被特定的事件处理函数处理之前(象keyPressEvent())处理它. 比如, 当我们想改变tab键的默认动作时,一般要重载这个函数. 在处理一些不常见的事件(比如:LayoutDirectionChange)时,evnet()也很有用,因为这些函数没有相应的特定事件处理函数. 当我们重载event()函数时, 需要调用父类的event()函数来处理我们不需要处理或是不清楚如何处理的事件.

3） 在Qt对象上安装事件过滤器.

安装事件过滤器有两个步骤: (假设要用A来监视过滤B的事件)

首先调用B的installEventFilter( const QOject \*obj ), 以A的指针作为参数. 这样所有发往B的事件都将先由A的eventFilter()处理.

然后, A要重载QObject::eventFilter()函数, 在eventFilter() 中书写对事件进行处理的代码.

4） 给QAppliction对象安装事件过滤器.

一旦给qApp(每个程序中唯一的QApplication对象)装上过滤器,所有的事件在发往任何其他的过滤器时,都要先经过当前这个 eventFilter(). 在debug的时候,这个办法就非常有用, 也常常被用来处理失效了的widget的鼠标事件,通常这些事件会被QApplication::notify()丢掉. ( 在QApplication::notify() 中, 是先调用qApp的过滤器, 再对事件分析, 以决定是否合并或丢弃)

5） 继承QApplication类,并重载notify()函数.

Qt 是用QApplication::notify()函数来分发事件的.想要在任何事件过滤器查看任何事件之前先得到这些事件,重载这个函数是唯一的办法. 通常来说事件过滤器更好用一些, 因为不需要去继承QApplication类. 而且可以给QApplication对象安装任意个数的事件。

### 9 你使用的QT版本是？有没有使用过QT4？QT5的信号槽与QT4相比有什么改进？

\*编译期：检查信号与槽是否存在，参数类型检查，Q\_OBJECT是否存在

\*信号可以和普通的函数、类的普通成员函数、lambda函数连接（而不再局限于信号函数和槽函数）

\*参数可以是 typedef 的或使用不同的namespace specifier

\*可以允许一些自动的类型转换（即信号和槽参数类型不必完全匹配）

### 10 信号槽机制：能说下你的理解吗？能用什么方法替代？槽函数可以是虚函数吗？

回调函数。可以。

### 11 信号槽是同步的还是异步的？分别如何实现？

通常使用的connect，实际上最后一个参数使用的是Qt::AutoConnection类型：Qt支持6种连接方式，其中3中最主要:

1.Qt::DirectConnection（直连方式）（信号与槽函数关系类似于函数调用，同步执行）

当信号发出后，相应的槽函数将立即被调用。emit语句后的代码将在所有槽函数执行完毕后被执行。

2.Qt::QueuedConnection（排队方式）（此时信号被塞到信号队列里了，信号与槽函数关系类似于消息通信，异步执行）

当信号发出后，排队到信号队列中，需等到接收对象所属线程的事件循环取得控制权时才取得该信号，调用相应的槽函数。emit语句后的代码将在发出信号后立即被执行，无需等待槽函数执行完毕。

3.Qt::AutoConnection（自动方式）

Qt的默认连接方式，如果信号的发出和接收这个信号的对象同属一个线程，那个工作方式与直连方式相同；否则工作方式与排队方式相同。

4.Qt::BlockingQueuedConnection(信号和槽必须在不同的线程中，否则就产生死锁)

这个是完全同步队列只有槽线程执行完成才会返回，否则发送线程也会一直等待，相当于是不同的线程可以同步起来执行。

5.Qt::UniqueConnection

与默认工作方式相同，只是不能重复连接相同的信号和槽，因为如果重复连接就会导致一个信号发出，对应槽函数就会执行多次。

6.Qt::AutoCompatConnection

是为了连接Qt4与Qt3的信号槽机制兼容方式，工作方式与Qt::AutoConnection一样。

如果这个参数不设置的话，默认表示的是那种方式呢？

没加的话与直连方式相同：当信号发出后，相应的槽函数将立即被调用。emit语句后的代码将在所有槽函数执行完毕后被执行。在这个线程内是顺序执行、同步的，但是与其它线程之间肯定是异步的了。如果使用多线程，仍然需要手动同步。

### 12 Qwidget、Qobejct实现了哪些功能？继承关系是什么？

QObject

1、信号和槽的非常强大的机制,使用connect()把信号和槽连接起来并且可以用disconnect()来破坏这种连接。为了避免从不结束的通知循环，你可以调用blockSignals()临时地阻塞信号。保护函数connectNotify()和disconnectNotify()使跟踪连接成为可能。

2、QObject可以通过event()接收事件并且过滤其它对象的事件。详细情况请参考installEventFilter()和eventFilter()。一个方便的处理者，childEvent()，能够被重新实现来捕获子对象事件。

3、最后但不是最不重要的一点，QObject提供了Qt中最基本的定时器，关于定时器的高级支持请参考QTimer。

4、注意Q\_OBJECT宏对于任何实现信号、槽和属性的对象都是强制的。

5、所有的Qt窗口部件继承了QObject。方便的函数isWidgetType()返回这个对象实际上是不是一个窗口部件。它比inherits(“QWidget” )快得多。

QWidget



1、QWidget类是所有用户界面对象的基类。

2、Widget是用户界面的基本单元：它从窗口系统接收鼠标，键盘和其他事件，并在屏幕上绘制自己。每个Widget都是矩形的，它们按照Z-order进行排序。

### 13 .Qt的事件处理机制是怎样的？

Qt的事件处理机制是一种基于事件环的事件驱动架构，在程序启动时会对主窗口运行一个事件循环，当事件队列中有事件时会将该事件推送到事件循环中进行处理，并将处理结果返回给相应的Widget控件，实现各控件之间的通信。

**14.Qt主要的布局类型有哪些？**

Qt中主要的布局类型有以下几类: QVBoxLayout、QHBoxLayout、QGridLayout、QFormLayout、QStackedLayout、QSplitter和QDockWidget。

### 15.Q\_PROPERTY有哪些特性？

Q\_PROPERTY是一种声明属性的宏，通过该宏可以将类的成员函数转变为属性。Q\_PROPERTY有以下特性： - 它可以是只读或可写的； - 它可以是一个普通的数据类型，也可以是一个qt类型； - 它可以是一个最小值和最大值范围的属性； - 它可以在属性值改变时发出信号； - 它可以跟踪对象值的变化，并提供一个系统通知。

### 16 你如何动态生成控件（Widget）？

可以使用QWidget的构造函数或Qt中特有的控件类构造函数来动态生成控件，例如： QWidget \*myWindow = new QWidget(); QPushButton \*myButton = new QPushButton("Button");

### 17 什么是QWidgets和QAbstractButton？

QWidgets是Qt框架中的一个基于QWidget的窗口类，继承了QWidget的所有特性，它支持各种常见的用户界面控件，并具有高效、可维护和易于扩展的特性。QAbstractButton是一个为所有Button提供抽象接口的类，它是QPushButton和QRadioButton等其他Button的父类。

18 Qt在数据库方面做了哪些工作？

Qt提供了强大的数据库支持，可以方便地完成数据库的连接、操作和维护。目前Qt支持的数据库有SQLite、MySQL、PostgreSQL和OBDC等，同时Qt还具有良好的跨平台能力和支持多语言的特性。

19. QT里面，子线程如果需要更新界面，怎么做？  
  
Qt不允许子线程直接访问UI线程的控件，因为控件属于UI线程，不能在其他线程中使用。为了解决这个问题，可以使用Qt提供的信号和槽机制，将子线程中的数据通过信号发送到UI线程中，然后在UI线程中的槽函数中更新UI控件的状态。  
  
也可以使用Qt的线程安全队列QConcurrent::BlockingQueue来传递数据。子线程将数据放入队列，UI线程在主循环中从队列中读取数据并更新UI控件的状态。  
  
除此之外，还可以使用Qt提供的QtConcurrent::run()函数在子线程中执行任务，将任务执行结果通过信号发送给UI线程，让UI线程主动更新UI控件的状态。

### 20 什么是QObject及其它派生自QObject的类？

QObject是Qt中的一个基类，几乎所有的Qt类都集成了QObject，包括QWidget、QThread、QTimer等等。它作为Qt中类的基础达到了优雅地处理对象生命周期的目的。此外，派生自QObject的其他类，例如QApplication、QMainWindow等也具有对象模型中的事件、信号和槽等属性。

### 21 什么是Qt的事件循环？

Qt中的事件循环指的是Qt应用程序中运行的一个无限循环，在该循环中的主要作用是等待用户的操作或程序内部的交互事件，并完成对应事件的响应处理。

### 22 C++多重继承与Qt中多重继承的区别是什么？

C++的多重继承通常会带来一些派生类中有同名的函数或方法，这样就需要显示指定使用哪个父类的方法或提供适当的别名。而在Qt中，因为它采用的是基于标识符（即字符串）的信号槽机制，允许在派生类中多次继承基类中的信号和槽，并且可以显式地指定是哪个基类对象进行信号和槽通信，避免了C++中可能出现的命名冲突问题。

### 23.Qt中的QDataStream有什么作用？

QDataStream是Qt中一个用于序列化和反序列化类和数据的类，它可用于数据持久化、网络数据传输和RPC通讯等。QDataStream的主要特点是可二进制序列化和反序列化，字节在序列化和反序列化时具有跨平台的特性。

### 24.什么是Qt的模板类？它有什么应用场景？

Qt中提供了一些模板类，例如QPair、QMap和QHash等，它们都支持任意类型的数据，可用于处理键值对、从已有数据形成数据集，以及更复杂的问题。这些模板类广泛应用于很多框架、库和工具中，例如STL（C++标准模板库）和Boost。

### 25.什么是Qt Designer？它有什么作用？

Qt Designer是Qt的一个可视化设计工具，用于快速、方便、灵活地创建Qt应用程序的用户界面，并提供了一个可视化的操作界面，可以轻松创建GUI的景象。它可以用于创建对话框视图、控件布局、一些按钮处理和命令操作等。

### 26.Qt的QTimer是什么？它有什么作用？

QTimer是Qt中的一个计时器类，它可以作为一个计时器，周期性地发出指定的信号，以实现定时操作、循环处理等功能。QTimer是通过事件循环来驱动的，根据设定时间进行消息处理，因此，当一个新的时间到达时，QTimer会启动并周期性地发出它的信号。

### 27 什么是Qt事件过滤器？

Qt事件过滤器是一种针对UI部件的系统程序，它可以捕获在特定部件上发生的事件，并对其进行过滤和处理。通常情况下，Qt事件过滤器是作为QObject的实例创建的，并且在QObject的installEventFilter()方法中进行注册。

### 28.什么是Qt QThread？它有什么作用？

Qt QThread是在Qt框架中用于多线程编程的一个类，可以利用它来在多线程之间进行数据、信号和事件的交互处理。

QThread也被用作一个抽象基类，为派生实例提供了高级的多线程编程能力，例如，

使用Qt QThread可以对数据库进行读写操作，对UI窗口进行渐变字体、滑动框、动态图片等等的效果。

### 29 什么是Qt的定时器？

Qt的定时器是用于周期性地执行特定操作的一种机制，例如动态调整UI内容、执行延迟操作、定时发送信号等等。

Qt中的QTimer类可以创建定时器，它使用了事件循环机制和响应机制，可以方便地编写定时器应用程序。

### 30.什么是Qt插件？它有什么作用？

Qt插件是一种可再使用的库拓展，可以用于扩展Qt应用程序的功能，它们可以被动态加载，就像应用程序的一个外壳，在运行时可以加载、添加和删除。Qt插件在应用程序的工作过程中能够动态地改变程序的行为，常常用于提供定制功能或延迟加载模块等操作。

### 31.什么是Qt Quick控件？

Qt Quick是一种基于QML（Qt Meta-Object Language）的快捷轻量级的用户界面控件集，用于创建具有动态交互和现代感觉的应用程序。Qt Quick控件部分提供了开箱即用的控件，其中包括简单的QML控件，例如矩形、圆形和文本框等。这些控件几乎可以与桌面、嵌入式、移动和Web平台上的任何其他环境一起使用。

### 32.什么是Qt的动画框架？

Qt的动画框架是一种可扩展、可重用、并且用于在Qt应用程序中添加特效的工具集。Qt的动画框架有一个抽象接口，可以通过它来动态创建并控制各种类型的动画效果。这些动画包括缓慢淡出、曲线、旋转、缩放等效果。

### 33.什么是Qt风格样式表？

Qt风格样式表是一种基于CSS（Cascading Style Sheets）样式和规则的方式，可以通过它来呈现Qt应用程序的操作和UI控件。使用-css可以轻松地修改应用程序的外观和布局，包括颜色、字体、位置、大小、阴影、边框和背景。这一功能在子类化QWidget时相当有用。

### 34. 什么是Qt的“内存管理”？

Qt的“内存管理”通常是指通过QObject子类的父/子关系进行对象的安全释放，处理程序中的泄漏，防止空指针。Qt的内存管理依赖QObject的对象模型，能够管理对象的生存期，以此避免因为控件、窗口等对象没有充分释放而导致的内存泄漏情况。

### 35. Qt线程和POSIX线程的区别是什么？ Qt线程和POSIX线程的主要区别在于它们的API和实现方式。Qt线程库提供了一些更为高级的功能，例如信号和槽、事件循环和Qt对象模型，而且易于使用。 与POSIX线程不同的是，Qt中的线程使用信号和槽机制作为线程间通信的标准方法，并且所有线程都有一个事件循环。这意味着线程可以轻松地响应输入事件和其他异步操作，同时还可以使用Qt对象模型来管理线程中的数据

### 36. QML是什么？它和Qt有什么关系？

QQ：1069619619

# 3 数据库

## 1 数据库事务有哪些？

原子性： 所有操作要么全部成功，要么全部失败

一致性： 例如转账，一个事务执行前和执行后必须一致

隔离性： 防止脏读， 重复读问题

持久性： 永久性提交数据库

## 2、什么是 Redis？简述它的优缺点？

Redis是一种内存数据结构存储系统，具有高效的键值存储能力和对多种数据结构的支持。它可以用于缓存、消息中间件、计数器、队列等多种应用场景。  
  
Redis的优点包括：  
  
1. 快速：Redis将数据存储在内存中，因此查找和读取数据非常快速。  
2. 多种数据结构支持：Redis支持多种数据结构，包括字符串、哈希表、列表、集合和有序集合。  
3. 高可用性：Redis具有高可用性，支持主从复制和哨兵模式，可以确保数据的高可靠性和持久性。  
4. 可扩展性：Redis的性能可以水平扩展，可以根据需要添加新的节点。  
5. 简单易用：Redis API简单易用，上手容易，提供了大量的命令和操作。  
Redis的缺点包括：  
1. 数据持久化：Redis默认不支持持久化，需要手动配置。  
2. 数据量限制：Redis在内存中存储数据，因此受到物理内存大小的限制。  
3. 安全性：Redis缺少内置的权限和认证机制，需要自行配置。  
4. 单线程：Redis是单线程的，不能利用多核CPU的优势。  
5. 操作复杂度：Redis在处理一些高级操作时，可能复杂度较高，需要考虑操作的性能和可行性。

## 3、Redis 支持哪几种数据类型？

**1.** **string** 字符串

字符串类型是 Redis 最基础的数据结构，⾸先键是字符串类型，⽽且其他⼏种结构都是在字符串类型基础上构建 的。字符串类型实际上可以是字符串：简单的字符串、 XML、JSON；数字：整数、浮点数；⼆进制：图⽚、⾳ 频、视频。

使⽤场景：缓存、计数器、共享 Session 、限速。

 **2.** **Hash** (哈希)

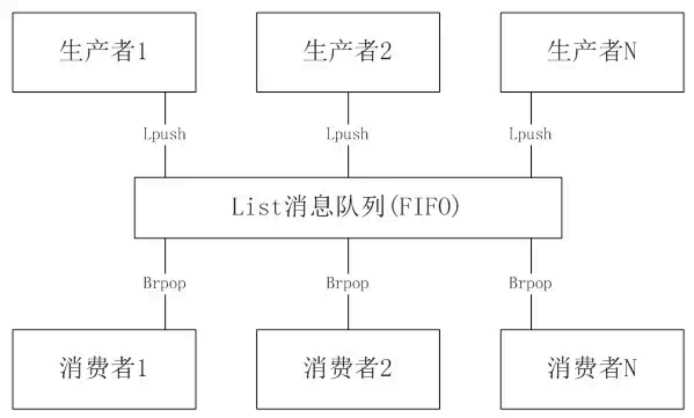
在 Redis中哈希类型是指键本身是⼀种键值对结构，如 value={{field1,value1},......{fieldN,valueN}}

使⽤场景：哈希结构相对于字符串序列化缓存信息更加直观，并且在更新操作上更加便捷。所以常常⽤于⽤户信息 等管理，但是哈希类型和关系型数据库有所不同，哈希类型是稀疏的，⽽关系型数据库是完全结构化的，关系型数 据库可以做复杂的关系查询，⽽ Redis 去模拟关系型复杂查询开发困难且维护成本⾼。

 **3.** **List** (列表)

列表类型是⽤来储存多个有序的字符串，列表中的每个字符串成为元素，⼀个列表最多可以储存 2 ^ 32 - 1 个元 素，在 Redis 中，可以队列表两端插⼊和弹出，还可以获取指定范围的元素列表、获取指定索引下的元素等，列表 是⼀种⽐较灵活的数据结构，它可以充当栈和队列的⻆⾊。

使⽤场景： Redis 的 lpush + brpop 命令组合即可实现阻塞队列，⽣产者客户端是⽤ lpush 从列表左侧插⼊元素， 多个消费者客户端使⽤ brpop 命令阻塞式的“抢”列表尾部的元素，多个客户端保证了消费的负载均衡和⾼可⽤性。



 **4.** **Set** (集合)

集合类型也是⽤来保存多个字符串的元素，但和列表不同的是集合中不允许有重复的元素，并且集合中的元素是⽆ 序的，不能通过索引下标获取元素， Redis 除了⽀持集合内的增删改查，同时还⽀持多个集合取交集、并集、差 集。合理的使⽤好集合类型，能在实际开发中解决很多实际问题。

使⽤场景：如：⼀个⽤户对娱乐、体育⽐较感兴趣，另⼀个可能对新闻感兴趣，这些兴趣就是标签，有了这些数据 就可以得到同⼀标签的⼈，以及⽤户的共同爱好的标签，这些数据对于⽤户体验以及曾强⽤户粘度⽐较重要。

 **5.** **zset** (**sorted** **set**：**\***\*有序集合) \*\*

有序集合和集合有着必然的联系，它保留了集合不能有重复成员的特性，但不同得是，有序集合中的元素是可以排 序的，但是它和列表的使⽤索引下标作为排序依据不同的是：它给每个元素设置⼀个分数，作为排序的依据。

使⽤场景：排⾏榜是有序集合经典的使⽤场景。例如：视频⽹站需要对⽤户上传的⽂件做排⾏榜，榜单维护可能是 多⽅⾯：按照时间、按照播放量、按照获得的赞数等。

## 4、Redis ⼀般都有哪些使⽤场景？

**Redis** 适合的场景

1. 缓存：减轻 MySQL 的查询压⼒，提升系统性能；

2. 排⾏榜：利⽤ Redis 的 SortSet (有序集合)实现；

3. 计算器/限速器：利⽤ Redis 中原⼦性的⾃增操作，我们可以统计类似⽤户点赞数、⽤户访问数等。这类操作 如果⽤ MySQL，频繁的读写会带来相当⼤的压⼒；限速器⽐较典型的使⽤场景是限制某个⽤户访问某个 API 的频率，常⽤的有抢购时，防⽌⽤户疯狂点击带来不必要的压⼒；

4. 好友关系：利⽤集合的⼀些命令，⽐如求交集、并集、差集等。可以⽅便解决⼀些共同好友、共同爱好之类的 功能；

5. 消息队列：除了 Redis ⾃身的发布/订阅模式，我们也可以利⽤ List 来实现⼀个队列机制，⽐如：到货通知、 邮件发送之类的需求，不需要⾼可靠，但是会带来⾮常⼤的 DB 压⼒，完全可以⽤ List 来完成异步解耦；

6. Session 共享： Session 是保存在服务器的⽂件中，如果是集群服务，同⼀个⽤户过来可能落在不同机器上， 这就会导致⽤户频繁登陆；采⽤ Redis 保存 Session 后，⽆论⽤户落在那台机器上都能够获取到对应的 Session 信息。

**Redis** 不适合的场景

数据量太⼤、数据访问频率⾮常低的业务都不适合使⽤ Redis，数据太⼤会增加成本，访问频率太低，保存在内存 中纯属浪费资源。

## 5 、Redis 有哪些常⻅的功能？

1. 数据缓存功能

2. 分布式锁的功能

3. ⽀持数据持久化

4. ⽀持事务

5. ⽀持消息队列

## 6、谈谈你对索引的理解？

索引的出现是为了提⾼数据的查询效率，就像书的⽬录⼀样。⼀本500⻚的书，如果你想快速找到其中的某⼀个知 识点，在不借助⽬录的情况下，那我估计你可得找⼀会⼉。同样，对于数据库的表⽽⾔，索引其实就是它的“⽬ 录”。

同样索引也会带来很多负⾯影响：创建索引和维护索引需要耗费时间，这个时间随着数据量的增加⽽增加；索引需 要占⽤物理空间，不光是表需要占⽤数据空间，每个索引也需要占⽤物理空间；当对表进⾏增、删、改、的时候索 引也要动态维护，这样就降低了数据的维护速度。

建⽴索引的原则：

1. 在最频繁使⽤的、⽤以缩⼩查询范围的字段上建⽴索引；

2. 在频繁使⽤的、需要排序的字段上建⽴索引。

不适合建⽴索引的情况：

1. 对于查询中很少涉及的列或者重复值⽐较多的列，不宜建⽴索引；

2. 对于⼀些特殊的数据类型，不宜建⽴索引，⽐如：⽂本字段(text)等。

## 7、索引的底层使⽤的是什么数据结构？

索引的数据结构和具体存储引擎的实现有关, ，在MySQL中使⽤较多的索引有 Hash 索引、 B+树索引等。⽽我们经 常使⽤的 InnoDB 存储引擎的默认索引实现为 B+ 树索引。

## 8、谈谈你对 B+ 树的理解？

1. B+ 树是基于 B 树和叶⼦节点顺序访问指针进⾏实现，它具有 B 树的平衡性，并且通过顺序访问指针来提⾼区 间查询的性能。

2. 在 B+ 树中，⼀个节点中的 key 从左到右⾮递减排列，如果某个指针的左右相邻 key 分别是 key i 和 key i+1，且不为 null，则该指针指向节点的所有 key ⼤于等于 key i 且⼩于等于 key i+1。

3. 进⾏查找操作时，⾸先在根节点进⾏⼆分查找，找到⼀个 key 所在的指针，然后递归地在指针所指向的节点 进⾏查找。直到查找到叶⼦节点，然后在叶⼦节点上进⾏⼆分查找，找出 key 所对应的data。

4. 插⼊、删除操作会破坏平衡树的平衡性，因此在插⼊删除操作之后，需要对树进⾏⼀个分裂、合并、旋转等操 作来维护平衡性。

## 9、为什么 InnoDB 存储引擎选⽤ B+ 树⽽不是 B 树呢？

⽤ B+ 树不⽤ B 树考虑的是 IO 对性能的影响， B 树的每个节点都存储数据，⽽ B+ 树只有叶⼦节点才存储数据，所 以查找相同数据量的情况下， B 树的⾼度更⾼， IO 更频繁。数据库索引是存储在磁盘上的，当数据量⼤时，就不能 把整个索引全部加载到内存了，只能逐⼀加载每⼀个磁盘⻚(对应索引树的节点)。

## 10、谈谈你对聚簇索引的理解？

聚簇索引是对磁盘上实际数据重新组织以按指定的⼀个或多个列的值排序的算法。特点是存储数据的顺序和索引顺 序⼀致。⼀般情况下主键会默认创建聚簇索引，且⼀张表只允许存在⼀个聚簇索引。

聚簇索引和⾮聚簇索引的区别：

聚簇索引的叶⼦节点就是数据节点，⽽⾮聚簇索引的叶⼦节点仍然是索引节点，只不过有指向对应数据块的指针。

## 11、谈谈你对哈希索引的理解？

哈希索引能以 O(1) 时间进⾏查找，但是失去了有序性。⽆法⽤于排序与分组、只⽀持精确查找，⽆法⽤于部分查 找和范围查找。

InnoDB 存储引擎有⼀个特殊的功能叫“⾃适应哈希索引”，当某个索引值被使⽤的⾮常频繁时，会在 B+ 树索引之上 再创建⼀个哈希索引，这样就让 B+Tree 索引具有哈希索引的⼀些优点，⽐如：快速的哈希查找。

## 12、谈谈你对覆盖索引的认识？

如果⼀个索引包含了满⾜查询语句中字段与条件的数据就叫做覆盖索引。具有以下优点：

1. 索引通常远⼩于数据⾏的⼤⼩，只读取索引能⼤⼤减少数据访问量。

2. ⼀些存储引擎(例如： MyISAM)在内存中只缓存索引，⽽数据依赖于操作系统来缓存。因此，只访问索引可 以不使⽤系统调⽤(通常⽐较费时)。

3. 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则⽆需访问主索引。

## 13、索引的分类？

 从数据结构⻆度

1. 树索引 (O(log(n)))

2. Hash 索引

从物理存储⻆度

1. 聚集索引(clustered index)

2. ⾮聚集索引(non-clustered index)

从逻辑⻆度

1. 普通索引

2. 唯⼀索引

3. 主键索引

4. 联合索引

5. 全⽂索引

## 13、谈谈你对最左前缀原则的理解？

MySQL 使⽤联合索引时，需要满⾜最左前缀原则。下⾯举例对其进⾏说明：

1. ⼀个 2 列的索引 (name, age)，对 (name)、 (name, age) 上建⽴了索引；

2. ⼀个 3 列的索引 (name, age, sex)，对 (name)、 (name, age)、 (name, age, sex) 上建⽴了索引。

1 、 B+ 树的数据项是复合的数据结构，⽐如： (name, age, sex) 的时候， B+ 树是按照从左到右的顺序来建⽴搜索 树的，⽐如：当(⼩明 , 22, 男)这样的数据来检索的时候， B+ 树会优先⽐较 name 来确定下⼀步的所搜⽅向，如果 name 相同再依次⽐较age 和 sex，最后得到检索的数据。但当 (22, 男) 这样没有 name 的数据来的时候， B+ 树 就不知道第⼀步该查哪个节点，因为建⽴搜索树的时候 name 就是第⼀个⽐较因⼦，必须要先根据name 来搜索 才能知道下⼀步去哪⾥查询。

2 、 当 (⼩明 , 男) 这样的数据来检索时， B+ 树可以⽤ name 来指定搜索⽅向，但下⼀个字段age 的缺失，所以只能 把名字等于⼩明的数据都找到，然后再匹配性别是男的数据了， 这个是⾮常重要的性质，即索引的最左匹配特性。

关于最左前缀的补充：

1. 最左前缀匹配原则会⼀直向右匹配直到遇到范围查询(> 、< 、between 、like)就停⽌匹配，⽐如： a = 1 and b = 2 and c > 3 and d = 4 如果建⽴ (a, b, c, d) 顺序的索引， d 是⽤不到索引的。如果建⽴ (a, b, d, c) 的 索引则都可以⽤到， a 、b 、d 的顺序可以任意调整。

2. = 和 in 可以乱序，⽐如： a = 1 and b = 2 and c = 3 建⽴ (a, b ,c) 索引可以任意顺序， MySQL 的优化器会优化

成索引可以识别的形式。

## 14、怎么知道创建的索引有没有被使⽤到？或者说怎么才可以知道这条语

句运⾏很慢的原因**?**

使⽤ Explain 命令来查看语句的执⾏计划， MySQL 在执⾏某个语句之前，会将该语句过⼀遍查询优化器，之后会 拿到对语句的分析，也就是执⾏计划，其中包含了许多信息。可以通过其中和索引有关的信息来分析是否命中了索 引，例如： possilbe\_key 、key 、key\_len 等字段，分别说明了此语句可能会使⽤的索引、实际使⽤的索引以及使⽤ 的索引⻓度。

## 15、什么情况下索引会失效？即查询不⾛索引？

下⾯列举⼏种不⾛索引的 SQL 语句：

1 、索引列参与表达式计算：

SELECT 'sname' FROM 'stu' WHERE 'age' + 10 = 30;

2 、 函数运算：

SELECT 'sname' FROM 'stu' WHERE LEFT( 'date',4) < 1990;

3 、%词语%--模糊查询：

SELECT \* FROM 'manong' WHERE `uname` LIKE '%码农%' -- ⾛索引

SELECT \* FROM 'manong' WHERE `uname` LIKE "%码农%" -- 不⾛索引

4 、 字符串与数字⽐较不⾛索引：

CREATE TABLE 'a' ( 'a' char(10));

EXPLAIN SELECT \* FROM 'a' WHERE 'a'="1" -- ⾛索引

EXPLAIN SELECT \* FROM 'a'WHERE 'a'=1 -- 不⾛索引，同样也是使⽤了函数运算

5 、 查询条件中有 or ，即使其中有条件带索引也不会使⽤。换⾔之，就是要求使⽤的所有字段，都必须建⽴索 引：

select \* from dept where dname= 'xxx' or loc= 'xx' or deptno = 45;

6 、正则表达式不使⽤索引。

7 、 MySQL 内部优化器会对 SQL 语句进⾏优化，如果优化器估计使⽤全表扫描要⽐使⽤索引快，则不使⽤索引。

## 16、查询性能的优化⽅法？

减少请求的数据量

1. 只返回必要的列：最好不要使⽤ SELECT \* 语句。

2. 只返回必要的⾏：使⽤ LIMIT 语句来限制返回的数据。

3. 缓存重复查询的数据：使⽤缓存可以避免在数据库中进⾏查询，特别在要查询的数据经常被重复查询时，缓存 带来的查询性能提升将会是⾮常明显的。

减少服务器端扫描的⾏数

1. 最有效的⽅式是使⽤索引来覆盖查询。

## 17 、InnoDB 和 MyISAM 的⽐较？

1. 事务： MyISAM不⽀持事务， InnoDB⽀持事务；

2. 全⽂索引： MyISAM ⽀持全⽂索引， InnoDB 5.6 之前不⽀持全⽂索引；

3. 关于 count(*)：MyISAM会直接存储总⾏数，* *InnoDB* *则不会，需要按⾏扫描。意思就是对于select* *count(*) from table; 如果数据量⼤， MyISAM 会瞬间返回，⽽ InnoDB 则会⼀⾏⾏扫描；

4. 外键： MyISAM 不⽀持外键， InnoDB ⽀持外键；

5. 锁： MyISAM 只⽀持表锁， InnoDB 可以⽀持⾏锁。

## 18、谈谈你对⽔平切分和垂直切分的理解？

 ⽔平切分

⽔平切分是将同⼀个表中的记录拆分到多个结构相同的表中。当⼀个表的数据不断增多时，⽔平切分是必然的选 择，它可以将数据分布到集群的不同节点上，从⽽缓存单个数据库的压⼒。

 垂直切分

垂直切分是将⼀张表按列切分成多个表，通常是按照列的关系密集程度进⾏切分，也可以利⽤垂直切分将经常被使 ⽤的列和不经常被使⽤的列切分到不同的表中。例如：将原来的电商数据库垂直切分成商品数据库、⽤户数据库 等。

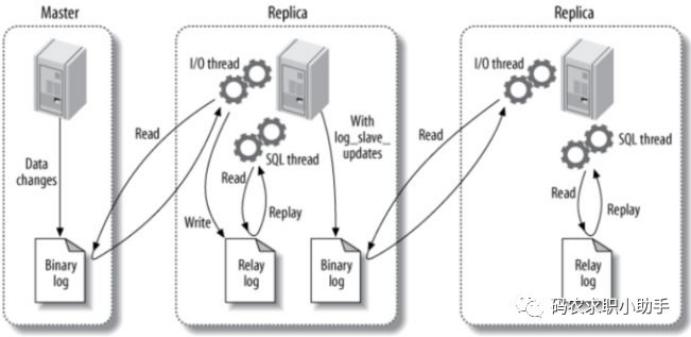
## 19、主从复制中涉及到哪三个线程？

主要涉及三个线程： binlog 线程、 I/O 线程和 SQL 线程。

1. binlog 线程 ：负责将主服务器上的数据更改写⼊⼆进制⽇志(Binary log)中。

2. I/O 线程 ：负责从主服务器上读取⼆进制⽇志，并写⼊从服务器的重放⽇志(Relay log)中。

3. SQL 线程 ：负责读取重放⽇志并重放其中的 SQL 语句。



## 20、主从同步的延迟原因及解决办法？

主从同步的延迟的原因：

假如⼀个服务器开放 N 个连接给客户端，这样有会有⼤并发的更新操作, 但是从服务器的⾥⾯读取 binlog 的线程 仅有⼀个， 当某个 SQL 在从服务器上执⾏的时间稍⻓或者由于某个 SQL 要进⾏锁表就会导致主服务器的 SQL ⼤ 量积压，未被同步到从服务器⾥。这就导致了主从不⼀致， 也就是主从延迟。

主从同步延迟的解决办法：

实际上主从同步延迟根本没有什么⼀招制敌的办法， 因为所有的 SQL 必须都要在从服务器⾥⾯执⾏⼀遍，但是主 服务器如果不断的有更新操作源源不断的写⼊，那么⼀旦有延迟产⽣，那么延迟加重的可能性就会原来越⼤。当然 我们可以做⼀些缓解的措施。

1. 我们知道因为主服务器要负责更新操作， 它对安全性的要求⽐从服务器⾼，所有有些设置可以修改，⽐如 sync\_binlog=1， innodb\_flush\_log\_at\_trx\_commit = 1 之类的设置，⽽ slave 则不需要这么⾼的数据安全， 完全可以将 sync\_binlog 设置为 0 或者关闭 binlog 、innodb\_flushlog 、innodb\_flush\_log\_at\_trx\_commit 也 可以设置为 0 来提⾼ SQL 的执⾏效率。

2. 增加从服务器，这个⽬的还是分散读的压⼒， 从⽽降低服务器负载。

## 21、谈谈你对数据库读写分离的理解？

读写分离常⽤代理⽅式来实现，代理服务器接收应⽤层传来的读写请求，然后决定转发到哪个服务器。主服务器处 理写操作以及实时性要求⽐较⾼的读操作，⽽从服务器处理读操作。

读写分离能提⾼性能的原因在于：

1. 主从服务器负责各⾃的读和写，极⼤程度缓解了锁的争⽤；

2. 从服务器可以使⽤ MyISAM，提升查询性能以及节约系统开销；

3. 增加冗余，提⾼可⽤性。

## 22、请你描述下事务的特性？

1. 原⼦性：事务是最⼩的执⾏单位，不允许分割。事务的原⼦性确保动作要么全部完成，要么完全不起作⽤；

2. ⼀致性：执⾏事务前后，数据库从⼀个⼀致性状态转换到另⼀个⼀致性状态。

3. 隔离性：并发访问数据库时，⼀个⽤户的事物不被其他事务所⼲扰，各并发事务之间数据库是独⽴的；

4. 持久性：⼀个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库 发⽣故障也不应该对其有任 何影响。

## 23、谈谈你对事务隔离级别的理解？

1. READ\_UNCOMMITTED (未提交读) : 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻 读或不可重复读；

2. READ\_COMMITTED (提交读) : 允许读取并发事务已经提交的数据，可以阻⽌脏读，但是幻读或不可重复读 仍有可能发⽣；

3. REPEATABLE\_READ (可重复读) : 对同⼀字段的多次读取结果都是⼀致的，除⾮数据是被本身事务⾃⼰所修 改，可以阻⽌脏读和不可重复读，但幻读仍有可能发⽣；

4. SERIALIZABLE (串⾏化) : 最⾼的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执⾏，这样事 务之间就完全不可能产⽣⼲扰，也就是说，该级别可以防⽌脏读、不可重复读以及幻读。但是这将严重影响程 序的性能。通常情况下也不会⽤到该级别。

## 24、解释下什么叫脏读、不可重复读和幻读？

 脏读：

表示⼀个事务能够读取另⼀个事务中还未提交的数据。⽐如：某个事务尝试插⼊记录 A，此时该事务还未提交，然 后另⼀个事务尝试读取到了记录 A。

 不可重复读 ：

是指在⼀个事务内，多次读同⼀数据。

 幻读：

指同⼀个事务内多次查询返回的结果集不⼀样。⽐如同⼀个事务 A 第⼀次查询时候有 n 条记录，但是第⼆次同等条 件下查询却有 n+1 条记录，这就好像产⽣了幻觉。发⽣幻读的原因也是另外⼀个事务新增或者删除或者修改了第⼀ 个事务结果集⾥⾯的数据，同⼀个记录的数据内容被修改了，所有数据⾏的记录就变多或者变少了。

## 25 、MySQL 默认的隔离级别是什么？

MySQL默认采⽤的 REPEATABLE\_READ隔离级别。

Oracle 默认采⽤的 READ\_COMMITTED 隔离级别。

## 26、谈谈你对MVCC 的了解？

数据库并发场景：

1. 读-读：不存在任何问题，也不需要并发控制；

2. 读-写：有线程安全问题，可能会造成事务隔离性问题，可能遇到脏读，幻读，不可重复读；

3. 写-写：有线程安全问题，可能会存在更新丢失问题。



多版本并发控制(MVCC)是⼀种⽤来解决读-写冲突的⽆锁并发控制，也就是为事务分配单向增⻓的时间戳，为每 个修改保存⼀个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。

**MVCC** 可以为数据库解决以下问题：

1. 在并发读写数据库时，可以做到在读操作时不⽤阻塞写操作，写操作也不⽤阻塞读操作，提⾼了数据库并发读 写的性能；

2. 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题。

## 27、说⼀下 MySQL 的⾏锁和表锁？

MyISAM 只⽀持表锁， InnoDB ⽀持表锁和⾏锁，默认为⾏锁。

表级锁：开销⼩，加锁快，不会出现死锁。锁定粒度⼤，发⽣锁冲突的概率最⾼，并发量最低。 ⾏级锁：开销⼤，加锁慢，会出现死锁。锁⼒度⼩，发⽣锁冲突的概率⼩，并发度最⾼。

## 28 、InnoDB 存储引擎的锁的算法有哪些？

1. Record lock：单个⾏记录上的锁；

2. Gap lock：间隙锁，锁定⼀个范围，不包括记录本身；

3. Next-key lock： record+gap 锁定⼀个范围，包含记录本身。

## 29 、MySQL 问题排查都有哪些⼿段？

1. 使⽤ show processlist 命令查看当前所有连接信息；

2. 使⽤ Explain 命令查询 SQL 语句执⾏计划；

3. 开启慢查询⽇志，查看慢查询的 SQL。

## 30 、MySQL 数据库 CPU 飙升到 500% 的话他怎么处理？

1. 列出所有进程 show processlist，观察所有进程，多秒没有状态变化的(⼲掉)；

2. 查看超时⽇志或者错误⽇志 (⼀般会是查询以及⼤批量的插⼊会导致 CPU与 I/O 上涨，当然不排除⽹络状态突 然断了，导致⼀个请求服务器只接受到⼀半。

# 4 分布式

# 5 音视频

## 1 聊聊你对H.264标准的理解

H.264是在MPEG-4技术的基础之上建立起来的，其编解码流程主要包括5个部分：帧间和帧内预测、变换和反变换、量化和反量化、环路滤波、熵编码。H.264标准的主要目标是：与其它现有的视频编码标准相比，在相同的带宽下提供更加优秀的图象质量。

## 2 什么是音视频？

音视频包括音频和视频。

音频：是声音信号的表示，它可以是人类说话的声音、音乐、环境声等。音频通过声波的振动来传递信息，通常以模拟或数字的形式进行记录和处理。音频的基本参数包括频率、振幅、时长等。频率决定了声音的高低，振幅决定了声音的强弱，时长则决定了声音的持续时间。

图表

描述已自动生成

视频：是一系列连续的图像，按照一定的速率播放，形成了动态的视觉效果。视频可以包含颜色、对比度、亮度等信息，以及物体的运动、形状和大小等。视频的基本参数包括分辨率、帧率、色彩深度等。分辨率决定了图像的清晰度和细节，帧率决定了视频的流畅度，色彩深度则影响了颜色的表现能力。

图片包含 不同, 束, 各种, 游戏机

描述已自动生成

## 3 什么是音频采样，采样率，声道数，位深？

采样：采样是将连续的音频信号在实践中轴上进行离散化的过程，通过对音频信号进行采样，可以将其转换为一系列样本。

采样率：采样率表示每秒钟采样的次数。常见的采样率包括 44.1kHz（CD 音质）、48kHz（专业音频常用）等。较高的采样率可以更准确地还原音频信号，但也会增加数据量。

声道数：声道数指音频信号的通道数量。单声道只有一个通道，双声道（立体声）有左右两个通道，多声道则可以有更多的通道，如 5.1 声道（左、中、右、左环绕、右环绕和低音）。

位深：位深表示每个采样点的二进制位数。常见的位深包括 16 位（CD 音质）和 24 位。较高的位深可以提供更丰富的动态范围和更精确的音频表示。

## 4 什么是视频帧率，码率，分辨率，位深？

帧率（Frame Rate）：帧率指每秒显示的画面数量，常见的帧率有24fps、30fps、60fps等，较高的帧率可以使视频更流畅，但也会增加数据量。

码率（Bit Rate）：码率是每秒传输的数据量，它反映了视频的质量和压缩程度。码率越高，视频质量通常越好，但文件也会越大。

分辨率（Resolution）：分辨率表示视频的像素数量，通常以水平像素数×垂直像素数的形式表示，如 1920×1080（全高清）、3840×2160（4K）等。较高的分辨率可以提供更清晰的图像。

位深（Bit Depth）：位深用于表示每个像素的颜色信息量。常见的位深有 8 位、10 位或 12 位。较高的位深可以呈现更丰富的颜色层次和细节。

## 5 为什么要对音视频编码

**存储**：

音视频编码可以将大容量的音视频数据进行压缩，从而减小文件大小，节省存储空间。有损压缩编码（如MP3、H.264）通过牺牲一定程度的音视频质量来实现高效的压缩，适用于对存储空间要求较高的场景。

无损压缩编码（如FLAC、Apple Lossless）可以保持音视频的原始质量，但通常文件大小较大，适用于对音视频质量要求较高且有足够存储空间的场景。

**带宽**：

在网络传输和流媒体播放过程中，音视频编码的选择会直接影响到所需的带宽。高效的音视频编码（如H.264、HEVC）可以在保持良好的视听质量的同时，降低传输过程中所需的带宽，有助于提升网络传输的效率。

特别是对于移动网络环境，带宽可能受限，采用适当的音视频编码可以减少数据传输量，节省用户的流量费用。

**传输**：

在实时传输和流媒体应用中，音视频编码的选择直接影响到数据的传输速度和稳定性。低延迟、高效率的编码（如VP9、AV1）可以提供更好的实时传输体验，减少延迟和卡顿现象，适用于视频会议、直播等场景。

同时，传输过程中所需的带宽也受到编码方式的影响，因此合适的编码选择对于保证传输质量至关重要。

以一个分辨率1920×1280，帧率30的视频为例：

共：1920×1280=2,073,600（Pixels 像素），每个像素点是24bit（前面算过的哦）；  
也就是：每幅图片2073600×24=49766400 bit，8 bit（位）=1 byte（字节）；  
所以：49766400bit=6220800byte≈6.22MB。  
这是一幅1920×1280图片的原始大小，再乘以帧率30。  
也就是说：每秒视频的大小是186.6MB，每分钟大约是11GB，一部90分钟的电影，约是1000GB。。。

## 6 什么是软编码、硬编码、软解码、硬解码？

软编码 (Software Encoding)：

软编码是指使用软件实现的音视频编码过程。在软编码中，音视频数据的压缩和编码是通过软件算法在CPU上完成的，不需要特殊的硬件支持。软编码通常具有灵活性强、易于实现和调整的特点，但可能会消耗较多的CPU资源，并且效率不如硬件编码高。

硬编码 (Hardware Encoding)：

硬编码是指使用专用硬件实现的音视频编码过程。在硬编码中，音视频数据的压缩和编码是由专门的编码芯片或硬件模块完成的，不需要CPU的直接参与。硬编码通常具有高效率、低功耗的特点，能够实现高质量的实时编码，适用于对性能要求较高的应用场景。

软解码 (Software Decoding)：

软解码是指使用软件实现的音视频解码过程。在软解码中，音视频数据的解码和解压缩是通过软件算法在CPU上完成的，不需要特殊的硬件支持。软解码通常具有灵活性强、易于实现和调整的特点，但可能会消耗较多的CPU资源，并且效率不如硬件解码高。

硬解码 (Hardware Decoding)：

硬解码是指使用专用硬件实现的音视频解码过程。在硬解码中，音视频数据的解码和解压缩是由专门的解码芯片或硬件模块完成的，不需要CPU的直接参与。硬解码通常具有高效率、低功耗的特点，能够实现高质量的实时解码，适用于对性能要求较高的应用场景。

## 7 有哪些音视频封装格式？

AVI（Audio Video Interleave）：AVI是由微软开发的一种经典的音视频封装格式，支持多种音视频编码格式，但不支持最新的压缩技术。

MP4（MPEG-4 Part 14）：MP4是一种常见的多媒体容器格式，支持多种音视频编码格式，如H.264、AAC等，广泛应用于网络视频、移动设备等。

MKV（Matroska Multimedia Container）：MKV是一种开放的多媒体容器格式，支持几乎所有的音视频编码格式，具有高级的功能，如多轨道支持、字幕、章节等。

MOV（QuickTime File Format）：MOV是由苹果公司开发的多媒体容器格式，常用于存储和播放音视频文件，支持多种音视频编码格式。

FLV（Flash Video）：FLV是一种使用Adobe Flash技术的流媒体容器格式，常用于网络视频播放，支持音频和视频的实时流传输。

TS（MPEG Transport Stream）：TS是一种用于传输数字音视频的容器格式，常用于数字电视、卫星电视和广播等领域。

WMV（Windows Media Video）：WMV是微软开发的一种音视频封装格式，通常用于在线视频和流媒体传输，支持Windows Media编解码器。

**8 什么是DTS与PTS？**

DTS（Decoding Time Stamp）：即解码时间戳，这个时间戳的意义在于告诉播放器该在什么时候解码这一帧的数据。

PTS（Presentation Time Stamp）：即显示时间戳，这个时间戳用来告诉播放器该在什么时候显示这一帧的数据。

由于在H264编码中存在IPB帧，导致了解码规则和显示规则的不一致，当视频流中没有 B 帧时，通常 DTS 和 PTS 的顺序是一致的。但如果有 B 帧时，就回到了我们前面说的问题：解码顺序和播放顺序不一致了。

比如一个视频中，帧的显示顺序是：I B B P，现在我们需要在解码 B 帧时知道 P 帧中信息，因此这几帧在视频流中的顺序可能是：I P B B，这时候就体现出每帧都有 DTS 和 PTS 的作用了。DTS 告诉我们该按什么顺序解码这几帧图像，PTS 告诉我们该按什么顺序显示这几帧图像。顺序大概如下：

PTS: 1 4 2 3

DTS: 1 2 3 4

Stream: I P B B

## 9 音视频同步?

一：什么是音视频同步？

音视频同步是指在播放音频和视频时，确保它们之间的时间同步，即音频和视频的内容按照正确的时间顺序播放，使其达到嘴型和声音能对的上。

二：音视频同步方案有哪些并仔细讲讲

视频同步音频：即以音频为主时间轴作为同步源

音频同步视频：即以视频为主时间轴作为同步源

音视频同步系统时钟：即以系统时钟为主时间轴作为同步源

市面上大多播放器都采用视频同步到音频，因为音频的采样率是固定的，若音频稍有卡顿，都会很明显的听出来，反则视频则不如此，虽然表面上说的是25P（每秒25帧），不一定每一帧的间隔就必须精确到40ms（所以每帧间隔大约40ms，事实上，也很难做到精确的40ms），即便偶尔视频间隔延时大了点或小了点，人眼也是察觉不出来的，所以视频的帧率可以是动态的，并不是严格标准的！

视频同步到音频，即以音频作为主时间轴， 尽量不去干扰音频的播放，音频采用独立的线程独自解码播放(音频播放的速度在参数设置完毕后是固定的，因此我们也很容易计算音频播放的时间),在整个过程中，根据视频与音频时间差，来决策如何改变视频的播放速度，来确保视频与音频时间差控制在一定范围内， 当偏移在-90ms（音频滞后于视频）到+20ms（音频超前视频）之间时，人感觉不到视听质量的变化，这个区域可以认为是同步区域；当偏移在-185到+90之外时，音频和视频会出现严重的不同步现象，此区域认为是不同步区域。这里我们认为偏移diff在‘±一个视频帧间隔’范围内即认为是同步的，如下图所示：

图示, 箱线图

描述已自动生成

## 10 弱网环境流媒体怎么优化?

传输协议选择：在弱网环境下，可以考虑使用UDP协议替代TCP协议，因为UDP协议不需要进行丢包重传，能够降低延迟。但需要注意的是，在对流媒体质量较高的情况下可以使用基于UDP的应用层协议例如KCP、QUIC，使用前向纠错或者重传机制来处理丢包问题，减少视频花屏或者模糊现象的发生。可以通过引入冗余数据或者快速重传等技术来提高数据传输的可靠性。

流量控制和带宽适应：实现流量控制和带宽适应功能，根据网络情况动态调整视频码率和分辨率，以确保在弱网环境下也能够流畅播放。

缓冲策略优化：在播放端实现合适的缓冲策略，例如增加缓冲区大小、优化缓冲控制算法等，以应对网络波动和抖动带来的影响。

网络质量监测和切换策略：实时监测网络质量，当网络环境变差时可以考虑切换到适合的低码率流或者其他CDN节点，以保证播放的稳定性和流畅性。

## 12 H.264有哪些关键技术？

1 帧内预测编码

帧内编码用来缩减图像的空间冗余。为了提高H.264帧内编码的效率，在给定帧中充分利用相邻宏块的空间相关性，相邻的宏块通常含有相似的属性。可以根据周围的宏块预测（典型的是根据左上角的宏块，因为此宏块已经被编码处理），然后对预测值与实际值的差值进行编码，这样，相对于直接对该帧编码而言，可以大大减小码率。

2帧间预测编码

帧间预测编码利用连续帧中的时间冗余来进行运动估计和补偿。H.264的运动补偿支持以往的视频编码标准中的大部分关键特性，而且灵活地添加了更多的功能，除了支持P帧、B帧外，H.264还支持一种新的流间传送帧——SP帧，码流中包含SP帧后，能在有类似内容但有不同码率的码流之间快速切换，同时支持随机接入和快速回放模式。

3整数变换

在变换方面，H.264使用了基于4×4像素块的类似于DCT（整数变换）的变换，但使用的是以整数为基础的空间变换，不存在反变换，因为取舍而存在误差的问题。此外，整数DCT变换还具有减少运算量和复杂度，有利于向定点DSP移植的优点。

4量化

H.264中可选32种不同的量化步长，这与H.263中有31个量化步长很相似，但是在H.264中，步长是以12.5%的复合率递进的，而不是一个固定常数。

5熵编码  
视频编码处理的最后一步就是熵编码，在H.264中采用了两种不同的熵编码方法：通用可变长编码（UVLC）和基于文本的自适应二进制算术编码（CABAC）。

## 13 怎么理解YUV，和RGB比较会有什么优势？

YUV和RGB都是用于表示彩色图像的编码格式；

RGB是一种直接表示红色（R）、绿色（G）和蓝色（B）三种颜色通道的编码格式。每个像素由这三种颜色的强度组成，通常使用24位或32位的表示（每个颜色通道8位）。RGB编码适用于大多数显示设备和图形处理器，因为它直接对应于人眼对颜色的感知。

YUV是一种通过亮度（Y）和色度（U和V）两种通道来表示颜色的编码格式。在YUV编码中，Y通道表示亮度信息，而U和V通道表示色度信息。YUV编码通常用于视频压缩和传输，因为人眼对亮度信息更为敏感，而对色度信息的变化不太敏感。因此，通过对色度信息进行降采样（降低分辨率），可以有效地减小数据量而保持图像质量。

在比较YUV和RGB时，它们各自有以下优势：

- YUV编码在视频压缩和传输中通常能够获得更好的压缩效率，因为它能够更好地利用人眼对颜色的感知特性。

- RGB编码在图形处理和显示中更为直观和易于理解，因为它直接对应于红、绿、蓝三种颜色通道的强度。

在实际应用中，通常会根据具体的需求和场景选择合适的编码格式。例如，在视频处理中，通常会使用YUV编码进行压缩和传输，而在图形处理和显示中，通常会使用RGB编码进行处理和显示。

## 14 什么是 IDR 帧？它和 I 帧有什么区别

IDR 帧全称叫做 Instantaneous Decoder Refresh，是 I 帧的一种。IDR 帧的作用是立刻刷新，重新算一个新的序列开始编码，使错误不致传播。IDR 帧有如下特性：IDR 帧一定是 I 帧，严格来说 I 帧不一定是 IDR 帧（但一般 I 帧就是 IDR 帧）；对于 IDR 帧来说，在 IDR 帧之后的所有帧都不能引用任何 IDR 帧之前的帧的内容。与此相反，对于普通的 I 帧来说，位于其之后的 B 和 P 帧可以引用位于普通 I 帧之前的 I 帧（普通 I 帧有被跨帧参考的可能）；播放器永远可以从一个 IDR 帧播放，因为在它之后没有任何帧引用之前的帧。因此，视频开头的 I 帧一定是 IDR 帧；一个封闭类 GOP 的开头的 I 帧也一定是 IDR 帧。所以，在直播场景通常每个 I 帧都是 IDR 帧，这样服务端下发流数据的时候总是从一个 I 帧开始，播放器就可以立即开始播放。

## 15 聊聊你对视频编码里面 I帧 B帧 P帧的理解

视频压缩中，每帧代表一幅静止的图像。而在**实际压缩时，会采取各种算法减少数据的容量**，其中IPB就是最常见的。

简单地说，I帧是关键帧，属于帧内压缩。就是和AVI的压缩是一样的。 **P是向前搜索的意思。B是双向搜索**。他们都是基于I帧来压缩数据。

I帧表示**关键帧**，你可以理解为这一帧画面的完整保留；解码时只需要本帧数据就可以完成（因为包含完整画面）

P帧表示的是**这一帧跟之前的一个关键帧（或P帧）的差别**，解码时需要用之前缓存的画面叠加上本帧定义的差别，生成最终画面。（也就是差别帧，P帧没有完整画面数据，只有与前一帧的画面差别的数据）

B帧是**双向差别帧**，也就是B帧记录的是本帧与前后帧的差别，所以，要解码B帧，不仅要取得之前的缓存画面，还要解码之后的画面，通过前后画面的与本帧数据的叠加取得最终的画面。B帧压缩率高，但是解码时CPU会比较累~。

**I和P的解码算法比较简单**，资源占用也比较少，I只要自己完成就行了，P呢，也只需要解码器把前一个画面缓存一下，遇到P时就使用之前缓存的画面就好了，如果视频流只有I和P，解码器可以不管后面的数据，边读边解码，线性前进，大家很舒服。

但网络上的电影很多都采用了B帧，因为B帧记录的是前后帧的差别，比P帧能节约更多的空间，但这样一来，文件小了，解码器就麻烦了，因为在解码时，不仅要用之前缓存的画面，还要知道下一个I或者P的画面（也就是说要预读预解码），而且，B帧不能简单地丢掉，因为B帧其实也包含了画面信息，如果简单丢掉，并用之前的画面简单重复，就会造成画面卡（其实就是丢帧了），并且由于网络上的电影为了节约空间，往往使用相当多的B帧，**B帧用的多，对不支持B帧的播放器就造成更大的困扰，画面也就越卡**。

一般平均来说，I的压缩率是7（跟JPG差不多），P是20，B可以达到50，可见使用B帧能节省大量空间，节省出来的空间可以用来保存多一些I帧，这样在相同码率下，可以提供更好的画质。

参考链接：https://blog.csdn.net/abcjennifer/article/details/6577934

## 16. 如何评估音视频传输质量，在弱网环境下如何优化？

评估音视频传输质量可以考虑以下指标：

1. 丢包率（Packet Loss Rate）：丢包率是衡量数据包在传输过程中丢失的比例。较低的丢包率通常表示较好的传输质量。

2. 延迟（Latency）： 延迟是指从发送端发出数据到接收端接收到数据的时间。低延迟有助于实时性要求较高的音视频应用。

3. 带宽利用率（Bandwidth Utilization）： 带宽利用率表示网络带宽被有效利用的程度。优化带宽利用率有助于提高整体传输效率。

4. 抖动（Jitter）：抖动描述数据包到达时间的不稳定性。较小的抖动有助于维持音视频流畅性。

5. 卡顿感知（Buffering and Freezing）：卡顿感知是用户体验的关键指标，涉及到音视频播放时是否出现卡顿或冻结现象。

在弱网环境下，可以采取以下优化策略：

1. 前向纠错（Forward Error Correction）：在传输过程中引入冗余数据，以便在出现丢包时进行纠错，提高数据恢复能力。

2. 重传机制（Automatic Repeat reQuest，ARQ）：当检测到丢包时，通过重传丢失的数据包来确保完整的数据传输。

3. 自适应码率调整（Adaptive Bitrate Streaming）：根据网络状况动态调整传输的码率，以适应不同带宽情况。

4. 动态帧率调整： 根据网络条件调整视频的帧率，平衡传输效果和资源占用。

5. 丢包恢复策略： 实现丢包恢复机制，例如通过插值算法填充丢失的音视频数据。

6. 网络层优化： 采用优化的网络协议、连接池管理，以减少网络传输延迟和提高稳定性。

7. 优化编解码器参数：针对弱网环境，调整编解码器的参数以降低对网络带宽的依赖。

通过综合考虑这些指标和采取相应的优化策略，可以在弱网环境下提高音视频传输的质量和稳定性。

## 17 H264/H265在编码上有哪些差别？

H.264和H.265（也称为HEVC）是两种视频编码标准

1. 压缩效率：H.265可以在相同的画质下以更低的比特率进行视频编码，或者在相同的比特率下提供更高质量的视频。2. 编码算法： H.265采用了更先进的编码算法，包括更强大的预测方法、更高效的变换方法和更有效的熵编码方法。使H.265能够更好地利用视频帧之间的相关性和冗余性，从而实现更高的压缩效率。

3. 复杂度： H.265采用了更复杂的编码算法，相对于H.264来说，它的编码和解码复杂度更高。这意味着在相同的硬件条件下，H.265的编码和解码速度可能会慢一些。

4. 延迟：H.265编码和解码的复杂性，它可能会导致更高的编码和解码延迟，尤其是在实时应用中，如视频会议或直播等场景下，这可能是一个重要的考虑因素。

## 18 解释RTP、RTCP和RTMP的作用及其区别？

**RTP（Real-time Transport Protocol）**是一种用于在互联网上传输实时数据的协议，通常用于音频和视频传输。它适用于各种实时应用场景，如IP电话、视频会议和流媒体等。

RTP协议的主要特点包括：

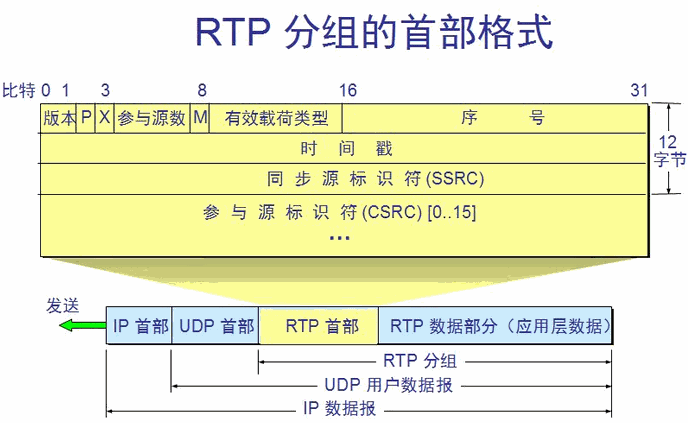
1. 时间戳和序列号： RTP包含时间戳和序列号，用于在接收端重组数据包并恢复时序关系，确保接收端能够按正确的顺序播放音频或视频数据。

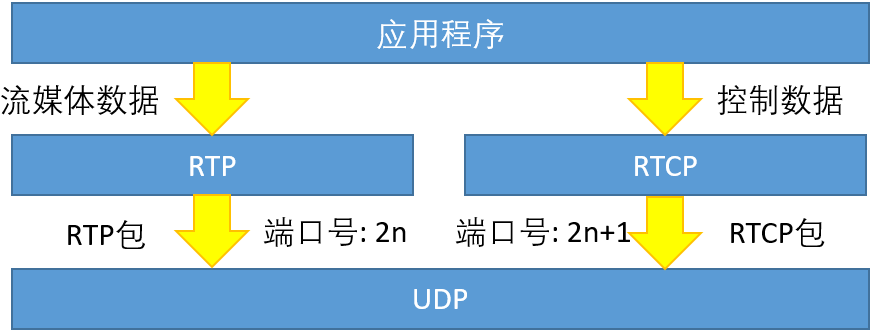
2. 负载类型标识：RTP头部包含负载类型标识，指示了RTP数据包中实际数据的类型，如音频、视频或其他自定义数据。

3. 扩展头部： RTP协议支持可选的扩展头部，用于传输额外的控制信息或应用特定的数据。

4. NACK和FEC支持： RTP本身并不提供丢包恢复功能，但可以与NACK（Negative Acknowledgement）和FEC（Forward Error Correction）等机制结合使用，以提高传输的可靠性。

在实时音视频传输中，RTP通常与RTCP（RTP Control Protocol）配合使用。RTCP用于传输控制信息，如丢包率、网络延迟等，以便发送端动态调整传输参数。RTP和RTCP通常一起使用，构成了RTP/RTCP协议组。





总的来说，RTP协议提供了一种标准的、可靠的实时数据传输机制，为实时音视频应用提供了必要的支持，并且具有足够的灵活性和可扩展性，以适应不同的应用场景和需求。

## 19 H264/H265在编码上有哪些差别？

4. \*\*动态帧率调整：\*\* 根据网络条件调整视频的帧率，平衡传输效果和资源占用。

5. \*\*丢包恢复策略：\*\* 实现丢包恢复机制，例如通过插值算法填充丢失的音视频数据。

6. \*\*网络层优化：\*\* 采用优化的网络协议、连接池管理，以减少网络传输延迟和提高稳定性。

7. \*\*优化编解码器参数：\*\* 针对弱网环境，调整编解码器的参数以降低对网络带宽的依赖。

通过综合考虑这些指标和采取相应的优化策略，可以在弱网环境下提高音视频传输的质量和稳定性。

## 20 你的视频播放器调整视频进度是如何实现的？需要考虑哪些因素？

视频播放器调整视频进度的实现涉及多个方面的考虑和步骤，以下是主要的因素：

1. 用户交互：视频播放器通常提供用户界面来让用户控制视频进度。用户可以通过拖动进度条或点击按钮来调整视频的播放位置。因此，首先需要实现与用户交互的功能，包括界面设计和交互逻辑的实现。

2. 视频解码和渲染：视频播放器需要能够解码视频文件，并将解码后的视频帧渲染到屏幕上。当用户调整视频进度时，播放器可能需要重新定位到新的时间点，并重新开始解码和渲染视频数据。

3. 时间定位：播放器需要准确地定位到用户所选择的时间点，并从该时间点开始播放视频。这涉及到从视频文件中定位到对应的帧或时间戳，并确保视频可以从所选时间点正确地开始播放。

4. 缓冲管理：在调整视频进度时，播放器可能需要重新加载视频数据或重新缓冲部分视频内容。这需要管理视频的缓冲区，确保在调整进度时能够快速地加载新的视频数据，并且能够实时地播放视频。

5. 音频同步：如果视频包含音频，播放器还需要确保视频和音频的同步。当调整视频进度时，需要考虑到音频和视频的时间同步关系，以避免出现音视频不同步的情况。

6. 性能优化：调整视频进度可能会对播放器的性能产生影响，特别是在重新加载视频数据或重新定位到新的时间点时。因此，需要对播放器进行性能优化，以确保在调整进度时仍然能够保持流畅的播放体验。

## 21 在使用FFmpeg库时，如何处理音视频同步的问题？

采用以下方法来确保音视频同步：

1时间戳管理： 使用FFmpeg获取音视频帧的时间戳，并根据时间戳将音视频帧同步播放。

2缓冲策略： 使用适当的缓冲策略，确保音视频帧的预加载和合适的缓冲区大小，以避免播放中断和不同步。

3时钟同步： 使用一个时钟作为整体的时间基准，确保音视频帧按照相同的时钟进行同步播放。

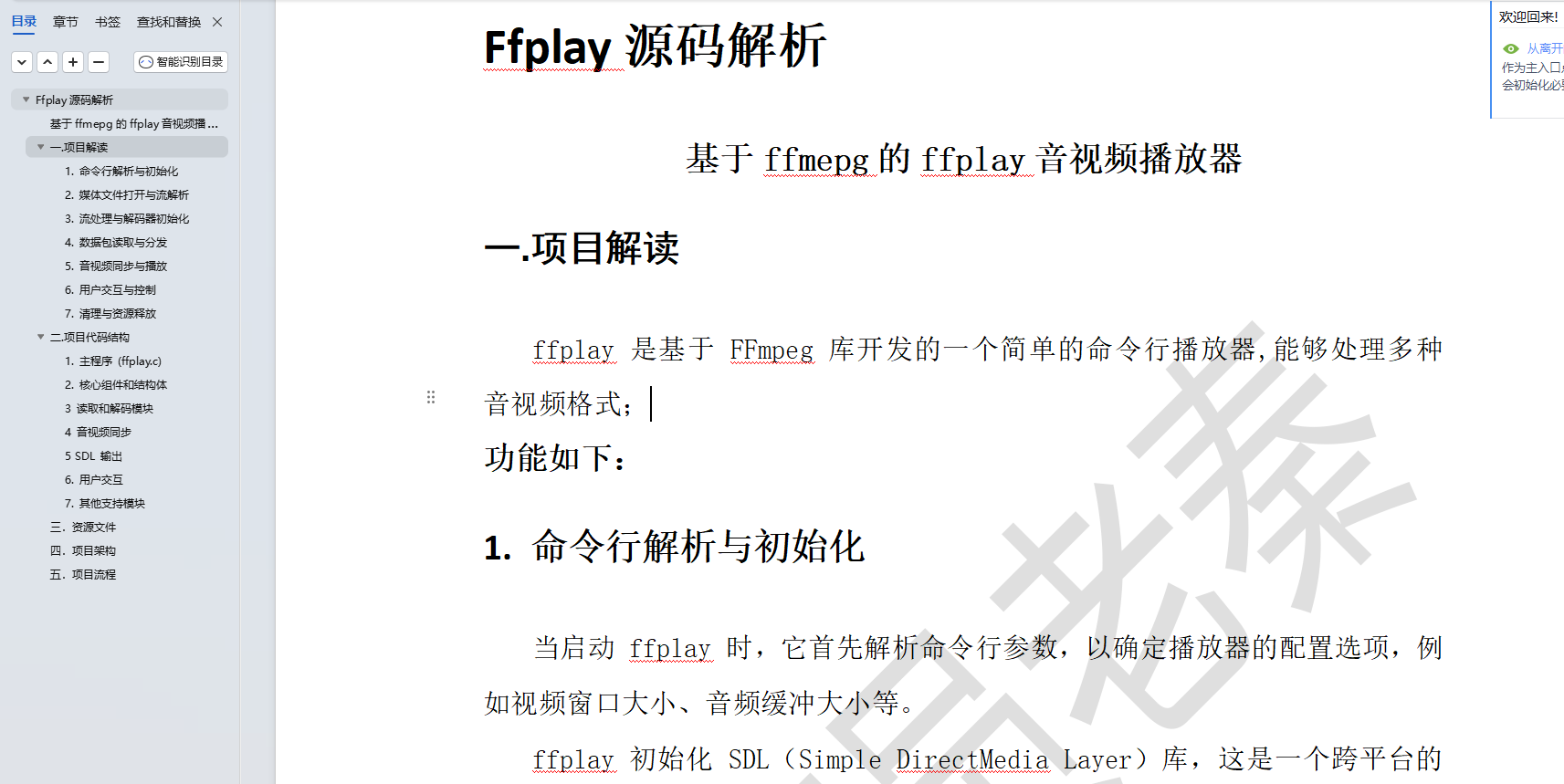
4调整播放速度： 可以通过调整音频或视频的播放速度来实现同步，确保它们在时间上保持一致。

H聊聊你对ffplay和ffprobe的理解，从源码层面分析

## 22 聊聊你对ffplay和ffprobe的理解

**ffplay源码解析**： 链接：https://pan.baidu.com/s/1bk0GG9014pIZO5nOBnpUcA 提取码：zth0

VS2019上运行



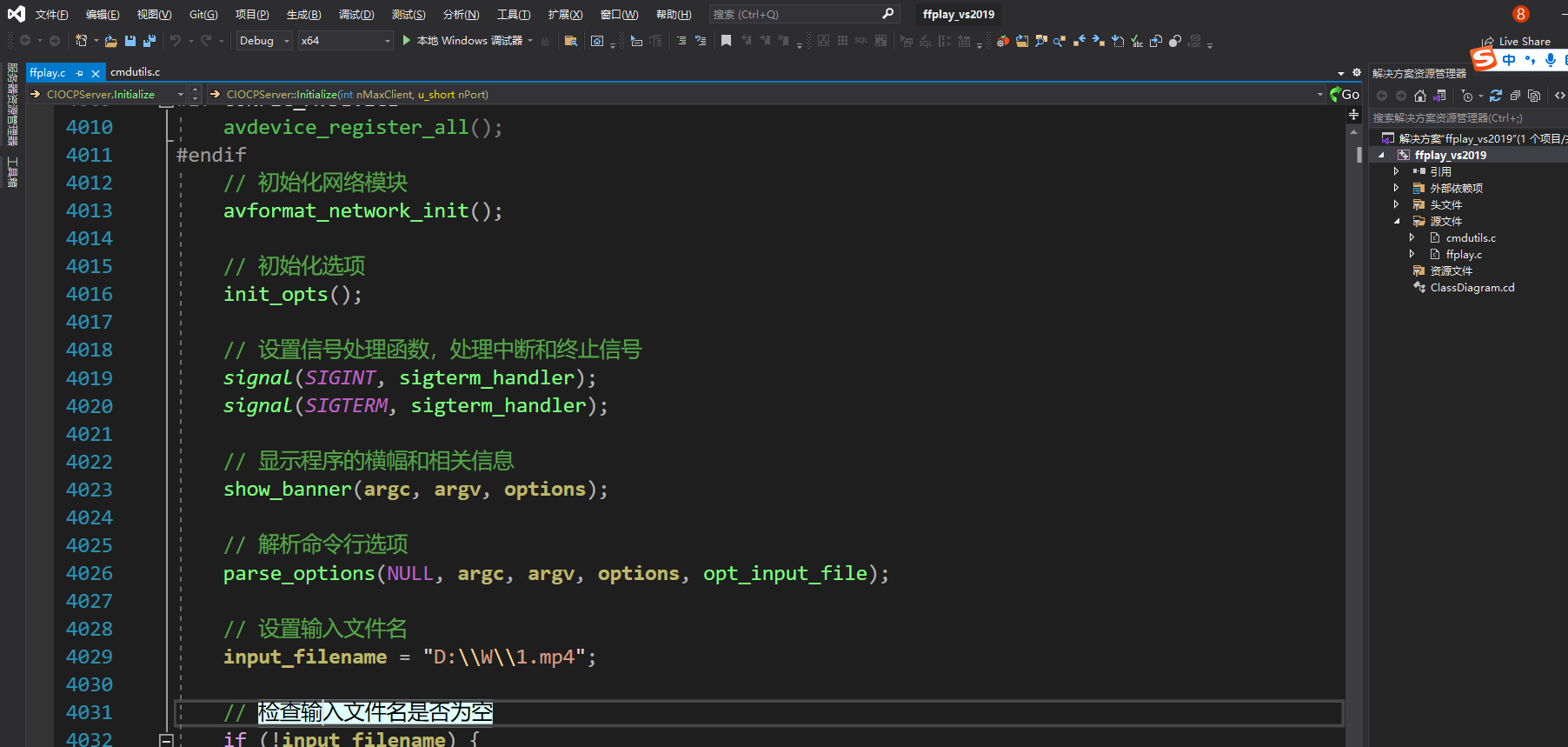
**ffplay**是一个简单的媒体播放器，使用FFmpeg的libavcodec和libavformat库来解码和播放多媒体文件。它支持大多数视频和音频格式，能够处理各种编解码器，并且支持多种播放控制功能。`ffplay`主要用于调试或者作为FFmpeg功能的快速演示。

从源码层面看，`ffplay`主要是通过调用FFmpeg库的功能来实现的。它初始化了一个SDL（Simple DirectMedia Layer）窗口来渲染视频帧，并使用FFmpeg的解码接口来解码视频和音频流。其主要流程包括打开媒体文件、读取媒体数据、解码视频和音频帧、将解码后的帧送到对应的视频或音频播放器中。

**ffprobe**是一个用于探测和分析多媒体内容的工具。它可以显示出媒体文件中包含的流信息，包括编解码器类型、视频的分辨率、帧率、比特率，以及音频的采样率、通道数等信息。`ffprobe`非常适合于脚本处理和自动化任务，因为它能够以多种格式（如JSON、XML）输出媒体文件的元数据，方便其他程序处理。

从源码角度看，ffprobe主要利用了FFmpeg的libavformat库来分析媒体文件。它通过读取和解析媒体文件的头部信息，而不实际解码媒体内容，来获取关于文件的详细信息。ffprobe的工作流程主要包括打开媒体文件、读取文件信息、根据用户的选项输出相关信息。

在源码层面，ffplay和ffprobe都是利用FFmpeg的核心库来实现其功能的。



libavformat：负责格式封装和解封装，即读取和写入不同媒体容器格式的文件。

libavcodec：负责编解码工作，提供了一套丰富的编解码器，支持多种音频和视频编解码。

libavutil：提供了一系列的工具函数，包括日志、错误处理等。

libswscale：用于视频帧的格式转换和缩放。

libswresample：用于音频重采样、格式转换和混音。

libavfilter：提供了复杂的音视频处理功能，如滤镜。

在使用这些库时，`ffplay`更侧重于**媒体的解码和播放功能，涉及到解码、视频渲染、音频播放**等方面。而`ffprobe`主要使用了libavformat来**获取媒体文件的结构和元数据信息，不涉及解码过程**。这两个工具都展示了FFmpeg库强大功能的不同方面，一个侧重于播放，一个侧重于分析。

## 23 聊聊音视频容器格式和协议MP4 FLV MPEG-TS HLS的理解？

**MP4（MPEG-4 Part 14）**：

MP4是一种常见的多媒体容器格式，被广泛应用于存储和传输在线视频、电影、音乐等多媒体内容。

它支持多种视频和音频编码格式，如H.264视频编码和AAC音频编码，具有良好的跨平台兼容性。

MP4文件通常以.mp4为扩展名，可以在各种设备上播放，如电脑、智能手机、平板电脑等。

**FLV（Flash Video）**：

FLV是一种流行的流媒体格式，支持多种视频和音频编码格式，如H.264和MP3。

它被广泛应用于网络视频平台、视频分享网站等在线视频播放场景，如YouTube、优酷等。

FLV文件通常以.flv为扩展名，能够实现在线流媒体播放和直播。

**MPEG-TS**（MPEG Transport Stream：

MPEG-TS是一种面向传输的流媒体容器格式，常用于数字电视广播、有线电视、卫星电视等领域。

它支持高清视频、多声道音频以及其他附加数据，适用于广播行业中的实时传输需求。

MPEG-TS文件通常以.ts为扩展名，被广泛应用于数字电视标准DVB等各种数字广播标准。

**HLS（HTTP Live Streaming）**：

HLS是一种基于HTTP协议的流媒体传输协议，将整个流媒体文件切分成多个小文件进行传输。

它支持自适应码率和跨平台播放，常用于网络直播、在线视频点播等场景。

HLS能够适应不同网络环境和设备类型，被广泛应用于各种移动端和桌面端的流媒体播放。

这些音视频容器格式在不同的应用场景中发挥着重要作用，满足了用户对多媒体内容播放和传输的各种需求。每种格式都有其特定的优势和适用范围，在选择使用时需要根据具体场景和需求进行综合考虑。

## 24 GOP画面组（Group of Pictures）

GOP说白了就是两个I帧之间的间隔。视频的编码(压缩)是按照“组”来进行的，每一个组叫作GOP（Group of Picture，图像组），表示一组连续的画面。通常意义上的GOP由I帧开始，到下一个I帧之前的帧结束（严格意义上讲，GOP由IDR帧开始，到下一个IDR帧之前的帧结束，这种情形下一个GOP中会有多个I帧。不过，一般而言，I帧基本上都是IDR帧。）

**I帧：**表示关键帧，又称帧内编码帧，一帧画面的完整保留，解码时只需要本帧数据就可以进行独立解码。

**P帧：**又称帧间预测（向前预测）编码帧，需要参考前面一帧才能进行编码。表示的是当前帧画面与前一帧（前一帧可能是I帧也可能是P帧）的差别。解码时需要用之前缓存的画面叠加上本帧定义的差别，生成最终画面。与I帧相比，P帧通常占用更少的数据位，但不足是，由于P帧对前面的P和I参考帧有着复杂的依耐性，因此对传输错误非常敏感。

**B帧：**又称双向预测编码帧，也就是B帧记录的是本帧与前后帧的差别。也就是说要解码B帧，不仅要取得之前的缓存画面，还要解码之后的画面，通过前后画面的与本帧数据的叠加取得最终的画面。B帧压缩率高，但是对解码性能要求较高，对于不支持B帧解码的播放器容易卡顿。

## 25 直播与点播的区别

直播（Live Streaming）：

直播是指实时的音视频内容传输和观看过程，内容在传输时即时产生并且实时传输给观众。观众在直播过程中不能控制内容的播放进度，只能实时观看当前直播内容。。

特点：

实时性强：内容在传输时即时产生并实时传输给观众。

无法控制播放进度：观众不能控制内容的播放进度，只能实时观看。

一次性消费：直播内容通常只能在直播过程中观看，无法进行暂停、快进等操作。

点播（Video on Demand）：

点播是指事先录制好的音视频内容存储在服务器上，用户可以根据自己的需求随时选择并播放内容，可以控制播放进度。并且可以在播放过程中进行暂停、快进、倒带等操作。

特点：

随时观看：用户可以根据自己的需求随时选择想要观看的内容。

可控制播放进度：观众可以在播放过程中进行暂停、快进、倒带等操作。

可重复观看：点播内容通常可以反复观看，不受时间和地点的限制。

## 26 直播技术的流程

采集（Capture）：

采集阶段涉及到从摄像头、麦克风等音视频设备中获取原始音视频数据的过程。这些设备捕获的数据可以是实时的现场内容，也可以是事先录制好的内容。

编码（Encoding）：

采集到的原始音视频数据通常非常庞大，需要进行压缩编码以减小数据量，提高传输效率。音视频编码器会将原始音视频数据编码为流媒体格式，如H.264、H.265等。编码后的数据称为码流。

协议封装（Protocol Encapsulation）：

编码后的音视频数据需要封装成特定的网络传输协议格式，以便在网络中传输。常见的流媒体传输协议有RTMP、RTSP、HLS等。协议封装的过程将编码后的音视频数据打包成特定的网络数据包，添加必要的控制信息和标识，以便接收端能够正确解析和处理。

推流（Push Streaming）：

推流是指将经过编码和封装的音视频数据通过网络传输到流媒体服务器的过程。推流可以使用RTMP、RTSP等协议将数据推送到流媒体服务器上，也可以使用HTTP-based协议将数据推送到CDN节点。

流服务转发（Media Server Forwarding）：

流媒体服务器接收到推流后，可以进行流的处理和转发。流媒体服务器负责管理连接、存储流数据、处理转码等操作。流媒体服务器可以将接收到的流数据转发给CDN节点，也可以进行其他处理后再转发。

CDN加速（CDN Acceleration）：

CDN（内容分发网络）是一种分布式网络架构，通过将内容缓存到离用户更近的节点上，提供更快的访问速度和更稳定的服务质量。CDN节点可以缓存流媒体内容，并根据用户的位置和网络条件选择最合适的节点进行内容分发，以提供更快的拉流速度和更稳定的播放体验。

拉流（Pull Streaming）：

拉流是指用户通过网络请求从流媒体服务器或CDN节点获取音视频数据的过程。用户可以使用支持流媒体协议的播放器或应用程序发送拉流请求，并在接收到数据后进行解码和播放。常见的流媒体协议有RTMP、RTSP、HLS等。

图示

描述已自动生成

## 27 直播怎么做到首屏秒开?

一：推流端

压缩和编码优化：对直播内容进行有效的压缩和编码，以减小数据量，并且优化编码算法，以降低编码延迟

固定码率编码 CBR 可以一定程度上消除网络抖动影响，如果能够使用可变码率编码 VBR 可以节省一些不必要的网络带宽，降低一定的延迟。因此建议尽量使用 VBR 进行编码。

二：服务端

GOP缓存策略：流媒体服务使用GOP缓存技术，在推流端往流媒体服务推流的时候开始缓存GOP组，在播放端请求媒体数据的时候将缓存的GOP统一发送。

CDN加速：利用内容分发网络（CDN）提供直播流加速服务，将直播内容分发到全球各地的服务器节点，使用户能够从最近的服务器获取数据，从而提高加载速度。

三：播放端

播放的缓冲策略做一些调整，如果第一帧还没有渲染出来的情况下， 不要做任何缓冲，直接送入解码器解码播放,其次拉流一开始不要进行音视频同步，同步会导致视频播放延迟。

# 6 嵌入式

### 什么是嵌入式系统，相对于桌面系统有哪些特点？

嵌入式系统指的是嵌入到其他设备中的计算机系统，主要用于特定的应用领域。通常用于控制、通信、显示、数据采集等领域。相对于桌面系统，嵌入式系统有以下几个特点：  
  
1. 硬件资源受限：嵌入式系统通常面临资源受限的问题，如内存、存储、处理能力等。  
2. 实时性要求高：嵌入式系统通常需要满足实时性要求，保证系统能够及时响应外部事件的发生。  
3. 稳定性要求高：嵌入式系统往往需要长时间运行，因此稳定性和可靠性要求非常高，需要对外部环境进行适当的适应。  
4. 电源管理：嵌入式系统通常需要较长时间的运行，并且需要使用尽量少的功率，因此需要考虑功耗控制和电源管理问题。  
5. 系统开放度低：嵌入式系统往往使用专用的硬件和操作系统，并且软件开发往往需要进行定制化开发，因此系统的开放度比桌面系统低。  
总之，嵌入式系统相对于桌面系统来说，更加专用、灵活，而且通常需要有更高的实时性、稳定性和可靠性。同时，嵌入式系统的资源受限和功耗要求也更高，需要通过特定的设计和优化来满足要求。

### 2. （场景题）嵌入式程序跑飞了，一般怎么定位和排查？

1 使用调试工具：使用嵌入式开发工具链中提供的调试工具，如 GDB（GNU调试器）或 JTAG 调试器，来跟踪程序的执行过程。通过设置断点、查看变量的值、追踪函数调用堆栈等方式，可以帮助定位异常的发生位置。

2 添加日志输出：在关键的代码段或事件发生处添加日志输出，记录程序的执行状态和变量的值。通过分析日志输出，可以定位到可能导致程序跑飞的问题。

3 代码回溯：当程序跑飞时，可以通过回溯（backtrace）技术来获取程序执行的堆栈信息。这可以提供有关导致程序异常的函数调用路径和上下文信息。

4 硬件监控：使用硬件监控设备（如逻辑分析仪或示波器）来监视系统的时序和信号，以确定是否存在硬件故障或异常。

5 剖析工具：使用剖析工具（如OProfile、perf等）来分析程序的性能瓶颈和调用路径，以确定是否存在耗时过长的操作或不正常的调用序列。

6 代码审查：仔细审查程序的源代码，查找可能引发问题的代码段。检查是否有未初始化的变量、内存越界访问、死循环、资源竞争等常见问题。

7 硬件调试：在一些特定情况下，可能需要检查硬件电路和外设的连接是否正确，以及时序和时钟是否正常。

8 利用日志和错误代码：查看系统日志、错误代码和异常处理机制，以了解程序执行过程中是否发生了异常或错误。

9 逐步排除法：通过逐步注释掉或禁用一部分代码来确定问题的来源，缩小定位范围。

10 时序和时钟：检查时序和时钟信号，确保硬件的时序和时钟配置正确。时序问题可能导致系统的不稳定和跑飞

### 3. 嵌入式系统中常见的 CPU 架构有哪些？

常见的嵌入式CPU架构有ARM、MIPS、PowerPC、x86等；

### 4 嵌入式系统中常见的外设接口有哪些？

常见的嵌入式系统外设接口有SPI、I2C、UART、USB、CAN协议等;以下是比较：

| **协议** | **原理** | **应用场景** | **设备支持** | **速度** |
| --- | --- | --- | --- | --- |
| SPI | 同步串行通信 | 连接微控制器和外围设备，如存储器、传感器和显示器 | 广泛支持 | 高速率（最高几百Mbps） |
| I2C | 同步串行通信 | 连接微控制器和外围设备，如传感器、EEPROM和实时时钟 | 广泛支持 | 低速率（最高几百Kbps） |
| UART | 异步串行通信 | 连接微控制器和外围设备，如调试器、无线模块和GPS接收器 | 广泛支持 | 低速率（最高几十Mbps） |
| USB | 通用串行总线 | 连接计算机和外围设备，如键盘、鼠标、打印机和存储设备 | 广泛支持 | 高速率（最高几Gbps） |
| CAN | 控制器局域网 | 汽车领域中的控制和通信 | 主要用于汽车领域，支持较少 | 中等速率（最高几百Kbps） |

**5 用GPIO模拟I2C？**

使用GPIO模拟I2C总线的思路如下：

1. 设置GPIO引脚的输入/输出模式：根据你的硬件平台和引脚分配，将两个GPIO引脚分别设置为输入和输出模式。

2. 发送起始信号：将SDA引脚置为高电平，然后将SCL引脚置为高电平，延时一段时间后，将SDA引脚置为低电平，再次延时一段时间后，将SCL引脚置为低电平。

3. 发送地址和命令：使用GPIO引脚模拟发送从设备的7位地址和读/写命令。根据I2C协议的规定，每个位的发送包括将SDA引脚置为相应的电平，然后将SCL引脚置为高电平，再次延时一段时间后，将SCL引脚置为低电平。

4. 接收应答信号：读取SDA引脚的电平状态，以确定从设备是否正确接收到地址和命令。如果接收到应答，SDA引脚的电平将保持低电平，否则将保持高电平。

5. 发送/接收数据：使用GPIO引脚模拟发送和接收数据字节。对于发送数据，将每个数据位的值分别写入SDA引脚，然后将SCL引脚置为高电平，再次延时一段时间后，将SCL引脚置为低电平。对于接收数据，读取SDA引脚的电平状态，并将每个数据位的值存储起来。

6. 发送停止信号：将SDA引脚置为低电平，然后将SCL引脚置为低电平，延时一段时间后，将SCL引脚置为高电平，再次延时一段时间后，将SDA引脚置为高电平。

以上是使用GPIO模拟I2C总线的基本步骤，你可以根据具体的硬件平台和引脚分配，编写相应的代码实现。请注意根据实际情况调整延时时间，以确保时序满足I2C协议的要求。以下是一个简单的示例代码，演示如何使用两个GPIO引脚模拟I2C总线的读写操作。

|  |
| --- |
| #include <stdio.h>  #include <stdint.h>  #include <wiringPi.h>  // GPIO引脚定义  #define SDA\_PIN 2  #define SCL\_PIN 3  // I2C总线延时函数  void i2c\_delay()  {  delayMicroseconds(5);  }  // 初始化I2C总线  void i2c\_init()  {  pinMode(SDA\_PIN, OUTPUT);  pinMode(SCL\_PIN, OUTPUT);  digitalWrite(SDA\_PIN, HIGH);  digitalWrite(SCL\_PIN, HIGH);  }  // I2C总线起始信号  void i2c\_start()  {  digitalWrite(SDA\_PIN, HIGH);  digitalWrite(SCL\_PIN, HIGH);  i2c\_delay();  digitalWrite(SDA\_PIN, LOW);  i2c\_delay();  digitalWrite(SCL\_PIN, LOW);  i2c\_delay();  }  // I2C总线停止信号  void i2c\_stop()  {  digitalWrite(SDA\_PIN, LOW);  digitalWrite(SCL\_PIN, LOW);  i2c\_delay();  digitalWrite(SCL\_PIN, HIGH);  i2c\_delay();  digitalWrite(SDA\_PIN, HIGH);  i2c\_delay();  }  // I2C总线写入一个字节  void i2c\_write(uint8\_t data)  {  for (int i = 0; i < 8; i++) {  if (data & 0x80)  digitalWrite(SDA\_PIN, HIGH);  else  digitalWrite(SDA\_PIN, LOW);  digitalWrite(SCL\_PIN, HIGH);  i2c\_delay();  digitalWrite(SCL\_PIN, LOW);  i2c\_delay();  data <<= 1;  }  digitalWrite(SDA\_PIN, HIGH);  digitalWrite(SCL\_PIN, HIGH);  i2c\_delay();  digitalWrite(SCL\_PIN, LOW);  i2c\_delay();  }  // I2C总线读取一个字节  uint8\_t i2c\_read()  {  uint8\_t data = 0;  for (int i = 0; i < 8; i++) {  data <<= 1;  digitalWrite(SCL\_PIN, HIGH);  i2c\_delay();  if (digitalRead(SDA\_PIN))  data |= 0x01;  digitalWrite(SCL\_PIN, LOW);  i2c\_delay();  }  return data;  }  int main()  {  // 初始化wiringPi库  wiringPiSetup();  // 初始化I2C总线  i2c\_init();  // 发送起始信号  i2c\_start();  // 向从设备写入数据  i2c\_write(0xA0); // 从设备地址  // 读取从设备返回的数据  uint8\_t data = i2c\_read();  // 发送停止信号  i2c\_stop();  // 打印读取的数据  printf("Data: 0x%02X\n", data);  return 0;  } |

本段代码演示了通过GPIO模拟I2C总线，向从设备写入数据并读取返回的数据。你可以根据需要扩展该代码，实现更复杂的I2C通信操作。

**6  I2C的时序图？**

以下是I2C总线的基本时序图：

Start S Addr Wr [A] Data [A] ... Data [A] Stop

Master: | | | | |

|<---->|<-------->|<---------->| |

Slave: | | | | |

在I2C总线的通信过程中，时序图中包含以下关键步骤：

1. Start（起始信号）：主设备发送一个低电平的起始信号来启动通信。

2. S（起始位）：主设备发送一个高电平的S信号，表示起始位。

3. Addr（地址）：主设备发送从设备的地址，并指定是读操作还是写操作。

4. Wr（写命令）：主设备发送一个写命令，指示从设备接收数据。

5. [A]（应答）：从设备发送应答信号，表示它已经接收到地址。

6. Data（数据）：主设备发送数据到从设备，可以有多个数据字节。

7. [A]（应答）：从设备发送应答信号，表示它已经接收到数据。

8. ...（重复数据和应答）：重复发送数据和应答，直到所有数据都被发送。

9. Stop（停止信号）：主设备发送一个高电平的停止信号来结束通信。

当然，I2C协议还有其他的特性和操作，如读取操作、重复启动、仲裁等，但以上时序图是I2C通信的基本流程。

### **7. 嵌入式系统中输入和输出设备的控制方法有哪些？**

嵌入式系统中输入和输出设备的控制方法有：中断IO、DMA、轮询方式等；

### 8 同步串口和异步串口的区别？

| **比较** | **异步串口** | **同步串口** |
| --- | --- | --- |
| 传输方式 | 按字符进行传输 | 按比特进行传输 |
| 时钟同步 | 无共享时钟信号 | 需要共享时钟信号 |
| 数据传输单位 | 字符（起始位、数据位、校验位、停止位） | 比特 |
| 数据传输速率 | 可调整，通常使用固定波特率（如9600 bps、115200 bps等） | 较高，可达到高速数据传输速率 |
| 应用场景 | 短距离通信，如计算机与外围设备、传感器、无线模块等 | 长距离通信，高速数据传输，如计算机与外部存储设备、图像传感器等 |
| 适用性 | 适用于低速、短距离、逐个字符传输的应用 | 适用于高速、长距离、按比特同步传输的应用 |
| 成本 | 相对较低，简单实现 | 相对较高，复杂实现 |

1. IO口工作方式有哪些，分别的作用和特点是什么？

| **工作方式** | **描述** | **特点** |
| --- | --- | --- |
| 上拉输入（Pull-up Input） | 当外部信号未连接时，IO口通过内部上拉电阻将输入电平拉高。当外部信号连接并拉低时，IO口输入为低电平。 | 1需要外部信号拉低才能触发输入状态 2适用于外部信号被拉高，需要检测外部信号是否被拉低的场景 3有内部上拉电阻，不需要额外的电路元件 |
| 下拉输入（Pull-down Input） | 当外部信号未连接时，IO口通过内部下拉电阻将输入电平拉低。当外部信号连接并拉高时，IO口输入为高电平。 | 1需要外部信号拉高才能触发输入状态2适用于外部信号被拉低，需要检测外部信号是否被拉高的场景3有内部下拉电阻，不需要额外的电路元件 |
| 推挽输出（Push-Pull Output） | IO口通过驱动器主动输出高或低电平，可以给外部设备提供高电平或低电平信号。 | 1可以主动驱动外部设备2能够提供较高的输出电流3适用于需要驱动高电平或低电平信号的场景 |
| 开漏输出（Open-Drain Output） | IO口只能主动输出低电平，而在高电平状态下处于高阻态，需要外部上拉电阻将电平拉高。 | 1只能主动输出低电平，高电平需要外部上拉电阻2适用于需要输出低电平信号，而在高电平时需要多个IO口共享的场景可以实现多个IO口的逻辑“与”或“非”功能 |

## **（场**景题）使用示波器来定位串口数据是否丢失？

1. 连接示波器：首先，将示波器的探头连接到串口通信线路的发送线和接收线上。确保示波器的地线（GND）连接到正确的地线上，以确保信号的参考电平一致。

2. 设置示波器参数：在示波器上设置以下参数：

- 时间基准（Timebase）：选择适当的时间刻度，使得可以观察到完整的串口数据传输周期。

- 纵轴（Vertical Axis）：设置合适的电压范围，以确保能够观察到串口数据的高低电平。

3. 观察串口数据传输：启动串口通信并观察示波器上的波形。以下是观察时需要关注的几个方面：

- 发送线（Tx）：观察发送线上的波形，确保有正确的电平变化。如果出现连续的低电平，可能表示数据丢失。

- 接收线（Rx）：观察接收线上的波形，确保有正确的电平变化。如果接收线上没有电平变化，可能表示数据丢失。

- 数据位：观察每个数据位的高低电平变化是否符合预期。如果有数据位缺失或错误的电平变化，可能表示数据丢失。

- 停止位：观察停止位的电平变化是否正确。如果停止位没有出现或电平不正确，可能表示数据丢失。

通过观察示波器上的波形，可以判断串口数据是否丢失或出现问题。如果有数据丢失，可能是由于信号干扰、电缆连接问题、波特率设置错误等引起的。

### RTOS是实时多任务系统吗？他的抢占式是如何实现的？？

是的，RTOS（实时操作系统）是**实时多任务系统**。它被设计用于处理实时任务，具有快速响应和可预测性的特点。

RTOS可以是抢占式的，也可以是非抢占式的。

抢占式RTOS的实现需要满足以下要求：

1. 任务调度器：实现任务的调度和切换，根据任务的优先级确定下一个要运行的任务。

2. 上下文切换：保存当前任务的上下文状态，并恢复下一个任务的上下文状态。

3. 中断处理：处理外部中断和定时器中断，触发任务的切换。

4. 任务堆栈：为每个任务分配独立的堆栈空间，用于保存任务的局部变量和上下文信息。

抢占式RTOS的实现通常依赖于底层硬件的支持，如时钟中断、中断控制器等。RTOS的设计和实现需要考虑任务的调度策略、中断处理、任务同步和通信等方面，以满足实时性和可靠性的需求。

### 12 BootLoader是什么？描述一下他的启动流程？

Bootloader（引导加载程序）是一种特殊的软件程序，位于计算机或嵌入式系统的存储器中，用于在系统上电或复位后引导加载操作系统或其他应用程序。

Bootloader的启动流程通常包括以下几个步骤：

1. 上电或复位：当计算机或嵌入式系统上电或复位时，处理器开始执行Bootloader所在的存储器位置处的指令。

2. 初始化：Bootloader首先进行一些初始化操作，例如初始化存储器、外设等，并设置一些必要的系统参数。

3. 硬件检测和初始化：Bootloader可能会进行硬件检测，以确定系统中存在的硬件设备和配置。它可能会初始化和配置这些硬件设备，例如设置时钟、外设等。

4. 加载操作系统或应用程序：Bootloader的主要任务是加载操作系统或其他应用程序。它会从指定的存储介质（如硬盘、闪存等）中读取操作系统或应用程序的映像文件，并将其加载到系统的内存中。

5. 启动操作系统或应用程序：一旦操作系统或应用程序被成功加载到内存中，Bootloader会传递控制权给操作系统或应用程序的入口点，从而启动它们的执行。

Bootloader还可能提供其他功能，例如配置引导选项、更新Bootloader本身、进行系统自检和修复等。Bootloader的具体实现和启动流程会因不同的硬件平台和操作系统而有所差异。

### **13** SPI四种模式比较

| **模式** | **CPOL**  **（时钟极性）** | **CPHA（时钟相位）** | **数据采样时间** | **数据传输方式** |
| --- | --- | --- | --- | --- |
| 模式 0 | 0 | 0 | 时钟的下降沿 | 数据在时钟的上升沿进行采样，下降沿进行传输 |
| 模式 1 | 0 | 1 | 时钟的上升沿 | 数据在时钟的下降沿进行采样，上升沿进行传输 |
| 模式 2 | 1 | 0 | 时钟的上升沿 | 数据在时钟的下降沿进行采样，上升沿进行传输 |
| 模式 3 | 1 | 1 | 时钟的下降沿 | 数据在时钟的上升沿进行采样，下降沿进行传输 |

### 14. SPI协议的原理解析？

SPI（Serial Peripheral Interface）是一种串行外设接口协议，用于在嵌入式系统中实现设备之间的通信。SPI协议的原理如下：

1. 硬件架构：SPI协议通常由一个主设备（通常是主控器）和一个或多个从设备（通常是外设）组成。主设备通过共享的时钟信号（SCLK）和两个单向的数据线（MOSI和MISO）与从设备进行通信。此外，每个从设备通常还有一个片选线（SS）用于选择特定的从设备。

2. 时钟信号：SPI通信中的时钟信号由主设备提供，并在主设备和从设备之间同步数据传输。时钟信号的频率由系统的时钟源决定，可以根据需要进行配置。

3. 数据传输：SPI协议是全双工的，可以同时进行数据的发送和接收。数据的发送由主设备通过MOSI线发送，从设备通过MISO线接收。数据在每个时钟周期上升沿或下降沿时进行传输，具体取决于SPI的模式设置。

4. 片选信号：为了选择特定的从设备进行通信，每个从设备通常都有一个片选信号（SS），由主设备控制。当特定的从设备被选中时，其对应的片选信号变为低电平，表示该从设备准备好接收或发送数据。

5. 传输格式：SPI协议没有规定特定的数据格式，数据的格式和意义由主设备和从设备之间的协议定义。可以是简单的原始数据传输，也可以是特定的通信协议。

6. 通信流程：SPI通信的流程通常为：选择特定的从设备（使其片选信号为低电平） -> 主设备通过SCLK时钟信号进行数据传输（发送和接收） -> 通信结束后，取消选择从设备（使片选信号为高电平）。

SPI协议的优点是简单、高速、灵活，适用于短距离、高速的设备间通信。但它不提供错误检测和纠错机制，通信距离受限，且需要多个引脚。因此，在选择使用SPI协议时，需要根据具体应用的需求来权衡其优缺点。

### 15. 蓝牙协议、433、zigbee协议的比较？

| **特点** | **蓝牙协议** | **433MHz无线协议** | **Zigbee协议** |
| --- | --- | --- | --- |
| 应用场景 | 个人设备之间的数据传输 | 低成本、低功耗的远程控制 | 大规模传感器网络，智能家居 |
| 传输距离 | 10米至100米范围内 | 几百米 | 10米至100米范围内 |
| 功耗 | 低功耗 | 低功耗 | 低功耗 |
| 数据速率 | 较高的传输速率 | 传输速率较低 | 速率中等 |
| 网络拓扑结构 | 点对点和多对多连接方式 | 单向广播或点对点通信 | 星型、网状或混合型网络 |

### 16. 你了解整个SPI的通信过程吗？SPI有几根线，分别是什么？你使用SPI的时候速率配置的是多少？

SPI（Serial Peripheral Interface）是一种用于外围设备间通信的同步串行通信协议。SPI通信通常由一个主设备和一个或多个从设备组成。

SPI通信的基本过程如下：

1. 主设备选择所需的从设备，并将其从逻辑状态拉低，以启动通信。

2. 主设备以全双工方式通过MOSI（Master Out Slave In）线发送数据到从设备，并从MISO（Master In Slave Out）线接收从设备返回的数据。

3. 通信过程中，时钟信号由主设备产生并控制，每个时钟周期传输一位数据。

4. 通信完成后，主设备将从设备的片选信号拉高，结束通信。

SPI通信通常使用四根线：

1. MOSI（Master Out Slave In）：主设备到从设备的数据线，用于主设备向从设备发送数据。

2. MISO（Master In Slave Out）：从设备到主设备的数据线，用于从设备向主设备发送数据。

3. SCK（Serial Clock）：时钟线，由主设备产生时钟信号，控制数据的传输速率。

4. SS（Slave Select）：片选线，由主设备控制，用于选择与之通信的从设备。

SPI的速率配置取决于具体的硬件和应用需求，通常可以配置为不同的速率。在使用SPI时，您可以根据具体的硬件和通信要求，通过配置寄存器或软件接口来设置通信速率。速率配置通常通过控制时钟信号的频率来实现。

### 17. 嵌入式系统中常见的数字信号处理器有哪些？

### **19. 单片机如何实现低功耗模式？** 单片机实现低功耗模式通常通过以下几种方式：

1. 降低时钟频率：将单片机的时钟频率降低可以减少功耗。可以通过设置时钟分频器或者更改时钟源来实现。

2. 关闭不需要的模块：将不需要的外设或模块关闭可以降低功耗。例如，关闭未使用的串口、定时器、ADC等模块。

3. 进入睡眠模式：单片机通常提供多种睡眠模式，例如待机模式、休眠模式、停止模式等。在睡眠模式下，单片机主频关闭，只保留必要的模块供电，以降低功耗。

4. 优化代码：通过优化代码，减少不必要的循环、延时等操作，可以降低功耗。例如使用中断代替轮询、使用低功耗的算法等。

5. 优化电源管理：合理设计供电电路，降低待机电流。可以通过选择合适的电源电路、使用低功耗电源芯片等方式来实现。

6. 选择合适的工作模式：根据具体应用需求，选择合适的工作模式。例如，使用低功耗模式的外设或者使用省电模式的传感器等。

不同的单片机厂商和型号可能具有不同的低功耗模式和实现方式，具体的实现方法请参考单片机的数据手册和相关技术文档。

### 20. 怎样实现数字和模拟信号的转换？

### 21. 中断的作用是什么？如何实现中断处理？

### 22. 嵌入式系统中常见的外设控制模式有哪些？

### 23. 串口的工作原理和实现方法是什么？

### 24. 如何实现定时和计数器功能？

### 25. 如何实现 PWM 功能？ 26. 如何实现 A/D 转换？

### 27. 嵌入式系统中如何实现 D/A 转换？

### 28.  SPI 总线的工作原理和实现方法是什么？

### 29. I2C 总线的工作原理和实现方法是什么？

### 30. UART 和 USART 的差异和实现方法是什么？

### 31.  PWM 和 DAC 的差异和应用场景是什么？

### 32. USB 接口的控制方法是什么？

### 33 ARM嵌入式系统中时序控制的作用是什么？如何实现？

### 34. 嵌入式系统中如何实现定点运算和浮点运算？

# 7 项目面试



## 0 公共类高频面试题

### 1 C++项目中你遇到的最难的问题是什么？你是如何解决的？

回答1：

在我的 C++ 项目中，我遇到最困难的问题是**多线程数据访问冲突导致的数据竞争**问题。在我的项目中，程序需要采集大量的传感器数据，并对数据进行分析和处理，最终输出结果。由于采集的传感器数据量很大，为了提高处理效率，我们使用了多线程技术。  
  
然而，在多线程场景下，由于多个线程同时访问共享的资源（如存储数据的数组等），会出现数据竞争问题，导致程序错误。在我的项目中，由于数据竞争问题导致的程序崩溃和逻辑错误，导致我们的项目进度受到很大的影响。为了解决这个问题，我采取了以下几个步骤：

1. **使用锁机制**：在多线程场景下，为了保证线程安全，需要对访问共享资源的代码块使用锁机制，以保证同一时间只有一个线程访问共享资源。我使用了互斥锁（mutex）来实现锁机制，并通过 RAII 技术来管理锁的生命周期，避免出现死锁等问题。  
   2. **优化设计**：为了减少数据冲突的发生，我对程序进行了优化设计，避免不必要的数据访问。例如，我将数据分段处理，每个线程只处理特定范围内的数据，避免不必要的数据访问和竞争。  
   3. **调试和测试**：为了确保程序的正确性，我采取了一系列的调试和测试措施。例如，我使用了调试工具来跟踪线程间的数据交互以及锁的使用情况，避免数据访问冲突。我还编写了一套测试用例，并使用随机数据对程序进行多次测试，以确保程序的正确性和稳定性。  
   通过上述措施，我成功地解决了C++项目中的数据竞争问题，使程序的准确性、可靠性和效率都有了显著提高。

回答2：

在我的后台开发经历中，遇到了一些具有挑战性的问题，其中最难的一个问题涉及到多线程和内存管理。  
  
我曾经参与的一个项目需要处理海量的数据，并且需要在单个程序实例中同时运行多个任务。这意味着我们需要设计一个高效的多线程方案，以便快速地处理大量的数据。由于是C++开发，我们自己管理内存而不是使用垃圾回收机制，所以需要谨慎处理内存分配和释放。  
  
我们最初的设计是使用多个线程和大量的共享内存来处理数据。这种方法看起来很不错，因为它能够快速地解决我们所面临的问题。然而，在实践中，我们发现，由于内存分配和释放的问题，程序往往会出现死锁和内存泄漏的情况，并且无法保证数据的正确性。  
  
为了解决这个问题，我们彻底修改了设备架构，使用了一种全新的多线程架构，这种架构将任务分解成小的、独立的代码块，并使用消息队列来处理这些代码块。每个线程都具有自己的内存池和堆栈，这些内存池和堆栈被专门用于分配和释放线程本地的内存资源。如果需要在不同的线程之间共享数据，我们将数据副本传递到下一个任务中，而不是共享内存。  
  
通过这种新的方法，我们成功地提高了程序的响应速度，并且减少了程序发生死锁和内存泄漏的情况。 此外，我们还使用一些静态代码分析工具来排除任何潜在的内存管理问题。  
  
总之，解决这个问题需要全面理解多线程编程以及内存分配和释放的最佳实践。通过重新设计设备架构并使用新的方法，我们成功地克服了这个困难，实现了高效的数据处理。

### 2 面试问题：QT开发中你遇到的最难的问题是什么？你是如何解决的？

在我的 QT 开发中，我遇到的最困难的问题是跨平台问题。由于 QT 可以在多种平台上运行，但各个平台的操作系统和编译器不同，因此在编写 QT 应用程序时，很容易遇到跨平台兼容性问题。下面是我遇到的一些问题以及我是如何解决的：  
  
1. 界面布局问题：在不同平台上，很容易遇到界面布局不同的情况，有些部件无法正确对齐或调整大小。为了解决这个问题，我使用了 QT 提供的布局管理器（如 QBoxLayout、QGridLayout 等）和相对布局技术，以及指定相应的平台样式文件。  
  
2. 操作系统差异问题：在不同的操作系统之间，系统调用和系统环境可能有所不同，导致程序在不同平台上运行表现不同。我解决这个问题的方法是编写平台相关代码，并使用条件编译技术定义不同平台的预编译选项。  
  
3. 编码问题：在不同的平台上，字符编码格式可能不同，导致程序在不同平台上输出结果不同。我使用 QT 提供的字符编码转换库（例如 QTextCodec），将字符编码统一为 UTF-8，在不同平台之间实现字符的正确转换。  
  
4. 外部库兼容性问题：在使用外部库时，由于不同平台的库版本和编译器不同，可能会遇到外部库兼容性问题。为了解决这个问题，我使用了 Git 模块和 CMake 构建工具来管理和构建依赖库，确保依赖库与操作系统和编译器兼容。  
  
通过掌握上述技巧和经验，我成功地解决了 QT 开发中的跨平台问题，并保证了程序在不同平台上的可靠性和兼容性。

关于工作经历的介绍：

### 4 关于Git的使用

#### 1. 什么是Git？有哪些优势？

Git是一个分布式版本控制系统，用于管理软件源代码的版本和历史记录。相较于其他版本控制系统，Git具有以下优势：

- 分布式系统：每个开发人员都拥有完整的代码库，而不是依赖于中央服务器。

- 高效性：与其他版本控制系统相比，Git能更快地执行较大规模的版本控制工作。

- 分支管理：Git的分支管理非常方便，能够支持各种完美的分支模型。

- 可扩展性：Git通过插件模型支持多种扩展，如工作流程处理、日志等。

#### 2. Git的基本工作流程是怎样的？

Git的基本工作流程如下：

- 在本地创建代码库——就是本地仓库。

- 在工作区对文件进行修改。

- 将修改后的文件暂存到缓存区。

- 将缓存区提交到本地仓库中。

- 将本地仓库中的代码推送到远程仓库，以完成版本更新。

#### 3. 什么是分支？如何在Git中创建和切换分支？

分支是代码库中的一个分支，用于支持多人协作开发。在Git中，可以创建和切换分支，使得多人可以在不同的分支上进行开发。创建和切换分支的命令如下：

- 创建分支：`$ git branch branch-name`

- 切换分支：`$ git checkout branch-name`

#### 4. 什么是提交？如何进行提交？

提交是在本地仓库中保存代码变更的操作。在Git中，需要先将修改后的文件放到缓存区中，然后再将缓存区中的修改提交到本地仓库中。提交的命令如下：

- 将修改后的文件放到缓存区：`$ git add file-name`

- 将缓存区中的修改提交到本地仓库中：`$ git commit -m "commit message"`

#### 5. 如何在本地仓库中撤销修改？

可以使用以下命令撤销在本地仓库中的修改：

- 撤销工作区中的修改：`$ git checkout -- file-name`

- 撤销已经添加到缓存区的修改：`$ git reset HEAD file-name`

#### 6. 如何进行远程仓库相关操作？

远程仓库相关操作主要包括以下几个步骤：

- 将本地仓库推送到远程仓库：`$ git push origin branch-name`

- 从远程仓库中获取代码：`$ git pull origin branch-name`

- 查看远程仓库信息：`$ git remote -v`

#### 7. git是怎么解决合并冲突？

当多个人在同一个文件的同一个位置同时进行修改时，会引发合并冲突。需要使用以下命令解决合并冲突：

- 查看合并冲突的文件：`$ git status`

- 手动解决合并冲突：在文件中使用编辑器手动修改代码。

- 将修改后的文件再次提交到本地仓库中：`$ git add file-name && git commit -m "resolved conflict"`

#### 8. Git中查看历史提交记录？

可以使用以下命令在Git中查看历史提交记录：

- 查看提交历史：`$ git log`

- 查看指定文件的提交历史：`$ git log file-name`

- 查看某个分支的提交历史：`$ git log branch-name`

#### 9. 如何在回退到某个历史版本？

可以使用以下命令在Git中回退到某个历史版本：

- 查看当前历史版本的信息：`$ git log`

- 回退到指定版本：`$ git reset commit-hash`

- 强制推送本地代码到远程仓库：`$ git push origin branch-name -f`

#### 10. 在Git中删除文件？

可以使用以下命令在Git中删除文件：

- 删除工作区中的文件：`$ rm file-name`

- 将删除操作提交到缓存区：`$ git rm file-name`

- 将删除操作提交到本地仓库中：`$ git commit -m "remove file"`

#### 11. 在Git中重命名文件？

可以使用以下命令在Git中重命名文件：

- 重命名工作区中的文件：`$ mv old-file-name new-file-name`

- 将重命名操作提交到缓存区：`$ git add new-file-name && git rm old-file-name`

- 将重命名操作提交到本地仓库中：`$ git commit -m "rename file"`

#### 12. 在Git中查看不同版本之间的差异？

可以使用以下命令在Git中查看不同版本之间的差异：

- 查看工作区和缓存区之间的差异：`$ git diff`

- 查看缓存区和本地仓库之间的差异：`$ git diff --cached`

- 查看本地仓库和某个历史版本之间的差异：`$ git diff commit-hash`

#### 13. 在Git中合并分支？

可以使用以下命令在Git中合并分支：

- 切换到目标分支：`$ git checkout target-branch`

- 合并源分支至目标分支：`$ git merge source-branch`

#### 14. 在Git中管理标签？

可以使用以下命令在Git中管理标签：

- 查看所有标签：`$ git tag`

- 查看指定标签的信息：`$ git show tag-name`

- 创建标签：`$ git tag -a tag-name -m "tag message"`

- 推送标签到远程仓库：`$ git push origin tag-name`

#### 15. 在Git中忽略某些文件不被追踪？

可以使用以下命令在Git中忽略某些文件不被追踪：

- 创建一个名为`.gitignore`的文件。

- 在文件中添加需要忽略的文件或目录，每行一个。

- 将`.gitignore`文件提交到本地仓库中：`$ git add .gitignore && git commit -m "add git ignore"`

#### 16. 在Git中恢复被删除的文件？

可以使用以下命令在Git中恢复被删除的文件：

- 从本地仓库中恢复文件：`$ git checkout commit-hash -- file-name`

- 从远程仓库中恢复文件：`$ git checkout -f origin/branch-name -- file-name`

#### 17. 在Git中查看某个文件的修改历史？

可以使用以下命令在Git中查看某个文件的修改历史：

- 查看文件的修改历史：`$ git log file-name`

#### 18. 如何在Git中撤销上一次提交？

可以使用以下命令在Git中撤销上一次提交：

- 撤销上一次提交：`$ git revert HEAD`

#### 19. 如何在Git中修改上一次提交？

可以使用以下命令在Git中修改上一次提交：

- 修改上一次提交：`$ git commit --amend`

#### 20. 如何在Git中管理多个远程仓库？

可以使用以下命令在Git中管理多个远程仓库：

- 添加远程仓库：`$ git remote add remote-name remote-url`

- 查看远程仓库信息：`$ git remote -v`

- 提交代码到指定远程仓库：`$ git push remote-name branch-name`

## 1、远控项目（基于工控环境的跨局域网的远程桌面控制系统）

### 1、能描述一下你做整个项目的过程吗？？（项目的美化）（最重要）

这个项目是我做软件开发的师兄介绍给我的，之前都是他在做，后面因为一些原因，他没有做了，当时没有办法，就找到了我，我当时刚好也看了一些类似于这种项目的文档，也熟悉网络编程、操作系统、服务器编程、桌面应用编程，于是我就接了我师兄的这个项目；

因为是工控项目，是在工厂里面的车间环境，本来市面上就有类似于teamview、向日葵、QQ自带的这种远控软件，但是由于工控环境的定制性，没有办法，就只能重新写这么一个软件。

我拿到这个软件的时候，代码写得一塌糊涂，用的是老的VS2008，IDE太老了，我直接把开发环境换成了最新的VS2019，最新的VS2019还支持git，很完美，我觉得做项目就是要做好版本管理。而且那个项目全是面向过程的设计，没有分层，我利用我之前看的MVC设计模式，我就开始重构，从画UML开始，到一个一个类的实现，全是我手把手写的，一路下来也出现了很多bug，当时也没什么经验，只有自己写日志的经验，也只会本地的VS2019调试，后面我又查资料，知道了如何看堆栈，如何监视一些变量等等，后面我还是辛辛苦苦把这个项目做完了，甚至还实现了跨局域网，过程很艰辛，结果还是做完了，大概是这样子。

这个项目是我上家公司的项目，是个老项目，我们公司当时也接了一些业务，因为深圳有很多工厂，刚好，公司接了这个项目，然后之前一个前辈被调到了其他的开发组，这个项目，就由我负责，我当时刚好也看了一些类似于这种项目的文档，也熟悉网络编程、操作系统、服务器编程、桌面应用编程，于是我就接了这个项目；

因为是工控项目，是在工厂里面的车间环境，本来市面上就有类似于teamview、向日葵、QQ自带的这种远控软件，但是由于工控环境的定制性，没有办法，就只能重新写这么一个软件。

我拿到这个软件的时候，代码写得一塌糊涂，用的是老的VS2008，IDE太老了，我直接把开发环境换成了最新的VS2019，最新的VS2019还支持git，很完美，我觉得做项目就是要做好版本管理。而且那个项目全是面向过程的设计，没有分层，我利用我之前看的MVC设计模式，我就开始重构，从画UML开始，到一个一个类的实现，全是我手把手写的，一路下来也出现了很多bug，当时也没什么经验，只有自己写日志的经验，也只会本地的VS2019调试，后面我又查资料，知道了如何看堆栈，如何监视一些变量等等，后面我还是辛辛苦苦把这个项目做完了，甚至还实现了跨局域网，做了UDP穿透，过程很艰辛，结果还是做完了，大概是这样子。

### 1.1. 请介绍一下您在该项目中的角色和职责。

答：我的角色是客户端和服务端的主要开发人员，负责完成整个客户端和服务端业务逻辑的实现，并进行二次代码重构。我的职责包括设计和实现整个系统的架构和模块，编写业务逻辑代码，进行测试和调试，并对代码进行重构，提高代码质量和可维护性。

### 2、能不能描述一下IOCP ？你在项目里面是如何使用的？

**1 IOCP是什么？（名词解释）**

IOCP模型属于一种通讯模型，适用于Windows平台下高负载服务器的一个技术。　IOCP全称I/O Completion Port，中文译为I/O完成端口。IOCP是一个异步I/O的Windows API，它可以高效地将I/O事件通知给应用程序，类似于Linux中的Epoll。

**2 为什么要使用IOCP？（为什么）**

在处理大量用户并发请求时，如果采用一个用户一个线程的方式那将造成CPU在这成千上万的线程间进行切换，后果是不可想象的。而IOCP完成端口模型则完全不会如此处理，它的理论是并行的线程数量必须有一个上限-也就是说同时发出500个客户请求，不应该允许出现500个可运行的线程。目前来说，IOCP完成端口是Windows下性能最好的I/O模型，同时它也是最复杂的内核对象。它避免了大量用户并发时原有模型采用的方式，极大的提高了程序的并行处理能力。

此外，传统的send/recv函数会要求驱动缓冲、系统缓冲和用户内存相互进行内存复制和传递。当交互量很大的时候，即浪费内存也效率低下。IOCP避免了这个问题，内核对象直接给用户分配内存，然后交给驱动。少了很多内存转移的过程。

**3 怎么使用IOCP？（怎么用？）**

1 创建和关联完成端口。

//功能：创建完成端口和关联完成端口

*HANDLE* *WINAPI* CreateIoCompletionPort(

\**\_\_in* *HANDLE* *FileHandle*, // 已经打开的文件句柄或者空句柄，一般是客户端的句柄

\**\_\_in* *HANDLE* ExistingCompletionPort, // 已经存在的IOCP句柄

\**\_\_in* *ULONG\_PTR* *CompletionKey*, // 完成键，包含了指定I/O完成包的指定文件

\**\_\_in* *DWORD* NumberOfConcurrentThreads // 真正并发同时执行最大线程数，一般推介是CPU核心数\*2

\*);

2 与socket进行关联

typedef struct {

*SOCKET* socket;//客户端socket

*SOCKADDR\_STORAGE* ClientAddr;//客户端地址

}PER\_HANDLE\_DATA, \* LPPER\_HANDLE\_DATA;

//与socket进行关联

CreateIoCompletionPort((*HANDLE*)(PerHandleData->socket),

completionPort, (*DWORD*)PerHandleData, 0);

3 获取队列完成状态

//功能：获取队列完成状态

/\*

返回值：

调用成功，则返回非零数值，相关数据存于lpNumberOfBytes、lpCompletionKey、lpoverlapped变量中。失败则返回零值。

\*/

*BOOL* GetQueuedCompletionStatus(

*HANDLE* CompletionPort, //完成端口句柄

*LPDWORD* lpNumberOfBytes, //一次I/O操作所传送的字节数

*PULONG\_PTR* lpCompletionKey, //当文件I/O操作完成后，用于存放与之关联的CK

*LPOVERLAPPED*\* lpOverlapped, //IOCP特定的结构体

*DWORD* dwMilliseconds); //调用者的等待时间

4 用于IOCP的特定函数

例如AcceptEx、ConnectEx、WSARecv、WSASend

只有这些函数可以附加重叠IO数据，

这样IOCP才能知道关注的套接字上发生了什么

//用于IOCP的特定函数

typedef struct \_OVERLAPPEDPLUS {

*OVERLAPPED* ol; //一个固定的用于处理网络消息事件返回值的结构体变量

*SOCKET* s, sclient; int OpCode; //用来区分本次消息的操作类型（在完成端口的操作里面，　　　　　　　　　　　　　　　　　　　　　　　是以消息通知系统，读数据/写数据，都是要发这样的　　　　　　　　　　　　　　　　　　　　　　 消息结构体过去的）

*WSABUF* wbuf;　　　　 //读写缓冲区结构体变量

*DWORD* dwBytes, dwFlags; //一些在读写时用到的标志性变量

}OVERLAPPEDPLUS;

5 投递一个队列完成状态用来进行特殊控制

特殊控制包括添加、删除和关闭完成端口

//功能：投递一个队列完成状态

*BOOL* PostQueuedCompletionStatus(

*HANDLE* CompletlonPort, //指定想向其发送一个完成数据包的完成端口对象

DW0RD dwNumberOfBytesTrlansferred, //指定—个值,直接传递给GetQueuedCompletionStatus　　　　　　　　　　　　　　　　　　　　　　　　函数中对应的参数

*DWORD* dwCompletlonKey, //指定—个值,直接传递给GetQueuedCompletionStatus函数中对应的参数

*LPOVERLAPPED* lpoverlapped, ); //指定—个值,直接传递给GetQueuedCompletionStatus　　　　　　　　　　　　　　　　　　　　函数中对应的参数

### 3、你在这个项目里面有遇到什么问题？是怎么解决的。

**问题1：画面卡死的问题**

回答：原因是缓冲区太小了。桌面的截图，如果内容比较少的时候，截图数据会比较小。一般在200k左右。但是如果东西很多，或者桌面背景比较复杂，则图片可以超过500k。

一开始的时候，随便测试了几次，就按照余量设计了一个400k的缓冲区。结果后面在有壁纸的机器上，就因为图片过大，导致缓冲区塞满了，都无法解析出一个包。使得网络通信实际上卡死了。

后面追加缓冲区到1M，解决了这个问题。

当然，**另外一个思路是拆分图片包。一个图片如果太大，则分割为200k或者300k的若干个包。**

**也可以防止这种情况的出现**。

 1 描述现象 2 解释排查思路 3 解决方案 4 验证

**问题2：文件列表不稳定，经常缺数据**

回答： （从日志调试的角度来回答解决过程）一开始不知道什么原因，只能加日志。因为下断点调试的时候，发现一切正常；一旦运行就会复现bug。加上日志后，发现部分数据包丢失，但是发送端数据是完整发送的。所以确定问题出现在接收的时候。然后给recv直接加日志，发现这里的数据没有丢失。进而判断数据问题是出现在解析的时候。然后在解析的地方加日志，发现如果出现长数据包分批次到达客户端的时候，最容易出现丢包。最后发现是index出了问题，本来应该是成员变量的，结果被一个本地的同名变量给覆盖了，变成了本地变量，没有记录作用，每次缓冲区都从0开始接收和解析数据，使得数据发生了丢失。修改本地变量名称后，该问题消失。

**问题3：驱动列表缺失最后一个驱动**

回答：一开始怀疑驱动获取函数有错误，断点调试发现并不是。然后在发送端下断点，发现发送的数据正常。怀疑是接收端有错误。在接收端下断点，接收到的数据一切正常。然后顺着往后单步，结果发现发现是驱动字符串转树节点的时候出现了问题。因为解析的时候依赖逗号分隔符，最后一个驱动后面没有逗号，所以被忽略了。后面在最后追加了一个逻辑，针对最后一个驱动进行了处理。再次运行，bug消失。

### 能说一下线程池的原理吗？你在项目中是如何使用的？

由于线程创建和销毁都会消耗系统资源，为了降低这些消耗，所以创建线程池来解决问题。当有线程任务要执行的时候，将任务打包成数据对象，然后交给空闲的线程去执行。

这种打包对象一般也叫线程任务。

有了线程池，所有需要使用线程的地方，都可以打包成线程任务，然后交给线程去执行。比如等待客户端的连接，接收客户端的数据，一些比较耗时的操作（获取文件列表，截屏等等操作）

有时候为了获取线程的执行结果，也会创建一些事件。如果执行完成，线程可以设置事件，外面的程序就可以去取结果了。

### 5、项目中文件操作（打开文件、操作文件）是怎么处理的？

要处理文件，有几个问题需要解决：一是文件函数比较零散，但是又需要前后呼应；二是部分文件尺寸可能较大，无法直接加载到内存中来；三是文件读写其实是比较耗时间的。

对于第一个问题，一般是将文件封装到类当中，打开和关闭可以在构造和析构里面做。也可以防止打开忘记关闭的情况。第二种情况，则需要按照需求分批次读取。这种需要能够在指定位置读取指定大小的数据。第三种情况则往往需要配合线程，进行异步操作。（一种是直接在线程中读取；一种是使用select异步io；还有一种是使用IOCP来读取文件。）

常用的文件操作函数和类有

C语言库：fopen、fclose、fread、fwrite

C++库：ifstream、ofstream

WindowAPI：CreateFile、WriteFile、ReadFile、CloseHandle

MFC：CFile

### 6、你是怎么做传输数据的协议封装和解析的？说下为什么要这么做？

我在命令发送这块使用了自定义协议，主要是封装命令和应答数据。

封装首先定义一个包头，我这里使用的是FE FF 然后是四字节包长度、包命令和包数据三个部分。最后是包和校验。

解析的时候，先从缓冲区定位包头字节，然后解析包长度。依据包长度，准备完包数据缓存，就可以开始提取包数据。最后以包命令在内的数据开始，计算和校验，如果通过，则表明包数据正常。

网络环境非常复杂，有各种嗅探包和垃圾数据。如果不对数据进行打包，则难以区分这些数据和我们的包数据。

### 7、手写线程池（参考代码）

1 **首先要先封装线程类**。线程的创建、销毁、设置、启动、停止和主线程函数都要设置好接口。

2 **然后设计线程池类**。需要设计好接口：初始化、大小调整、添加任务、关闭、获取尺寸、启动等。线程池有一个任务列表属性，一个事件属性和一个互斥锁。外部投递进来的任务先是加入任务列表属性。然后设置事件属性。每个子线程都会等待事件。一旦事件被设置，则获取到该事件的线程就会通过互斥锁来同步任务列表，取其中一个任务出来开始执行。

3 **设计任务类**。任务类需要有一个任务函数接口。这个接口线程类会自动调用。此外需要封装一些做参数用的属性。一般任务类是基类，用户可以依据自己的需求去派生子类。子类重载任务函数即可，也可以自己追加一些属性，用于任务函数。

4 **最后整合三个模块**，一个最原始的线程池就完成了。

### 8、你是如何做到跨局域网的？（UDP穿透的原理）

其原理是基于以下几点：

1 UDP是无连接的通信，只要知道对方地址，就可以发送消息。无需像TCP那样，需要三次握手建立连接。

2 UDP服务器创建后，防火墙、路由器会开放端口，等待外面的数据。

UDP 穿透技术是一种在局域网和广域网中实现数据传输的方法，主要用于处理 NAT 网络环境下的数据传输问题。在远程桌面控制软件中，由于被控制端和控制端所处的网络环境可能不同，如果没有采用 UDP 穿透技术，使用 TCP 协议进行数据传输时可能会因为被控制端或控制端处于 NAT 网络环境中而不能正常传输数据。

通过使用 UDP 穿透技术，可以绕过 NAT 网络环境中的限制，实现远程桌面控制软件在不同网络环境中的正常数据传输。

### 9、你是如何看到对方计算机的屏幕图像？能讲一下原理吗？

基本原理就是将对方屏幕显示做成图片，然后回传到监控机器，只要回传的速度够快，我们就能观察到对方屏幕的活动。

但是考虑到屏幕上一个点是四个字节（RGBA），一般1920×1080的分辨率会有至少829440个字节的图片需求。我们会在截屏之后，将其压缩为png格式的图片，来缩小需要的内存空间。不用jpg，是因为png能在相同的清晰度情况下，有更小的体积。

### 能描述一下MVC设计模式吗？（请说明你在该项目中使用了哪些设计模式，并简要介绍它们的作用）

我在该项目中使用了 MVC 设计模式和 IOCP 模型。

MVC 设计模式是一种用于开发用户界面的模式，它将应用程序分为三个部分：模型、视图和控制器。其中，模型代表应用程序的数据和业务逻辑，视图是用户界面，控制器负责逻辑控制。MVC 设计模式有助于提高应用程序的可重用性和可维护性，对于远程桌面控制软件中的用户界面设计和业务逻辑实现非常适用。

IOCP 模型是一种基于事件驱动的异步 I/O 模型，它能够高效地处理大量连接和数据传输请求，提高系统的性能和稳定性。在远程桌面控制软件中，使用 IOCP 模型可以实现高效的网络通信，处理大量的连接和数据传输请求。

M表示model，即模型层

V表示view，即显示层、表示层、界面层

C表示controller，即控制层

界面层不关心具体逻辑，也不关心具体数据。如果界面层需要数据，则开放接口或者属性，让其他层来调用。注意，界面层需要处理用户的交互。将用户的行为反馈给控制层，但是如何处理要依据具体的情况来定。如果是界面层的处理，则自己完成；如果还涉及业务处理，则交由控制层来处理

模型层只关心数据模型，对于界面层如何展示、控制层如何使用，毫不关心。如果外部需要使用某些数据，或者进行某些数据处理，模型层会开放接口出来，供外部使用。

控制层是MVC的核心，这层的代码可复用性最差，一般都是和具体业务深度绑定的。控制层负责整个产品的逻辑和调度。从界面层来的用户请求，到模型层的数据摘选，以及业务进度的控制，都是在这一层完成的。

### 远控项目里面你是如何锁定对方屏幕？

锁屏本质上是要控制用户的所见和所行

所以这里要做几件事情

1 控制用户所能看到的东西

将我们的程序铺满屏幕

将我们的程序置顶显示

2 控制用户的行为

屏蔽用户的按键输入。

屏蔽系统按键的输入。

所谓屏蔽，就是接收这些消息，但是不做任何回应

### 为什么你的界面编写要采用mfc，不采用QT？

~~具体采用什么框架是要按照实际需求来定的。~~

~~一般QT用于多平台的情况，比如播放器。这种可以不需要怎么修改的情况下，适配各种平台，而在window上多用wtl、mfc。考虑到开发便捷性，我这里选取了mfc。Vs对mfc的开发支持比较充足，开发起来也容易一些~~

【远控分为控制端和被控端。

对于被控端要求体积小、能后台以服务方式运行、随时能响应和处理系统消息或者用户消息。这种显然使用带MFC的命令行程序要强于Qt。

首先，因为Qt本身是基于信号和槽的机制，而非消息机制。所以它在处理消息这块有很大的不便。其次，Qt适合做ui，而非做服务程序。

最后被控端还是有一些UI需求的，用带MFC的命令行程序，就是更好的选择。而且命令行程序转化成后台服务程序也更加容易。

所以被控端应该使用更容易使用的带MFC的命令行程序，这是最佳下选择。

对于控制端，当然也可以使用Qt做，但是这样会导致一套项目采用两种架构。

无论是联调、代码维护、版本控制，都有很大的不便

而使用MFC做，则可以最大限度的降低这些因素的影响。

VS不仅仅支持多个项目的启动，还支持多个项目的同时调试。

而且可以将两套项目加入同一个解决方案，这样代码维护和版本控制也比较容易进行。所以我最后选择了MFC而没有采用Qt。

】

### 13、服务器不用linux下的epoll，要用windows的IOCP？

~~首先远控项目是window下的项目，所以只能在window下运行的epoll肯定是不能使用的。~~

~~其次，原本是打算用select的，但是考虑到未来的扩展，所以使用了能够支持更大并发，效率更高的ICOP。~~

【

我们的远控是基于Windows系统的远程控制。Linux下面大多数是命令行环境，图形界面比较少。如果单纯是命令行界面，开源的OpenSSH已经足够了，没有必要再重复造轮子。

而Windows下的开源远程桌面项目，很少有成熟易用的C++项目。所以我们只能自己实现一个Windows下的远程桌面系统。

此外，作为网络服务器的是被控端。既然被控端设定是Windows系统，我们肯定是无需考虑Linux的问题。

虽然控制端和被控端都无需使用Linux，但是我最后还做了一些UDP穿透的尝试。

UDP穿透需要一台公网的服务器作为桥梁。这台服务器，我们可以使用Linux做服务器，自然也就可以使用epoll作为提高通信并发能力的强力技术。

最后我想说的是，其实无论是epoll还是IOCP，只是不同系统下，用于提高IO并发能力的技术。我会依据项目实际需求来使用，而不是无脑随意炫技式使用。毕竟这些技术还需要搭配Reactor模式来使用，会使得整个项目的开发成本，维护成本变高。

这些问题，都是我在项目第一次重构的时候，切身的体会。按需开发，才能减少开发周期，降低开发成本。

】

### 14、你的跨局域网功能是如何实现的？

我的跨局域网功能，是利用UDP穿透技术来实现的。

和TCP通信不同，UDP通信允许一个套接字既可以做客户端发送数据，也可以做服务端接收数据。

但是两台局域网内的机器之间，是无法知道对方真实IP和端口的。

所以要跨局域网，还必须要借助一个第三方服务器。

这个服务器的地址，是局域网内的机器所知道，并且能访问的。

有了这台服务，两台不同局域网内的计算机，就可以按照下面的步骤建立通信：

第一步，在第三方服务器上创建一个UDP服务器。然后等待客户端接入。

第二步，发起者创建UDP服务器，然后连接第三方服务器。连接完成之后，会向服务器发送一个数据包，并且服务器会应答一个数据包。此时，公网上的网关会保留这个通信入口给本机。

第三步，接收者创建UDP服务器，然后连接第三方服务器。连接完成之后，会向服务器发送一个数据包，并且服务器会应答一个数据包。此时，公网上的网关会保留这个通信入口给本机。

第四步，第三方服务器将发起者的地址发给接收者，将接收者的地址发给发起者。

第五步，收发双方拿到对方地址后，就可以通过地址开始发送消息。此时，因为网关保留了一个入口给两者，所以两者可以自由的进行通信，不再需要第三方服务器了。

注意，这种方式会导致本地通信暴露在公网上，使用的时候请谨慎进行。

### 你为什么要进行代码的二次重构？

代码重构是开发当中经常出现的情况，尤其是开发需求经常变化，项目经验匮乏，或者业务场景相对复杂的项目。本次项目重构主要的原因是我没有同类型的项目开发经验。导致一些设计和构想后面发现有缺陷，需要跟进开发过程中掌握的新的经验，来重新组合和调整。如果下次有类似的项目，这些弯路就完全可以避免。

比如一开始是打算用同步网络通信的，结果后面发现一些交互是比较耗时的。比如读取文件夹下的文件信息，比如文件传输功能。同步通信的过程会导致消息处理函数无法及时返回，从而导致界面卡顿。后面改成了异步网络通信，这种问题就消失了。但是在一开始，我是完全没有预想到这种情况的。

最后，重构也不是全部推倒重来。而是尽可能保留原有的大部分代码的情况，进行重构。最好像精准的手术一样，只切除有问题的部分，替换上新的部分。这也要求设计的时候，时刻遵循高内聚，低耦合的原则。出现问题的时候，不会导致问题扩散到项目的每个模块之中去。

1 算法优化 a 封包，解决tcp的粘包；b if-else转化为key-value的回调函数；key-value是线性的

2 流程优化 if-else转化为key-value的回调函数；key-value是线性的

3 业务匹配： a 文件列表 特殊的一个包结束 测试的文件比较少；一次通信就可以发过来；

上千个文件，树状存储；next标记；

B 显示图片

4 代码的可读性

### 16、这个项目，你是如何实现获取文件驱动信息的，用什么样的数据结构存的，为什么？

文件驱动器有不少函数，这里选择的是\_chdrive函数

选择这个函数，考虑的是兼容性和可靠性。

这个函数是修改当前的驱动，成功返回0，失败返回-1

遍历1~26，可以确保返回的驱动都能够访问

而其他函数不具备这个特性，比如GetLogicalDrivers函数

有时候会返回空的光驱盘符，但是该盘无法访问，进而导致其他问题

而且这个函数无论是server还是桌面系统，都支持

但是GetLogicalDrivers则支持到2003，2000等更古老的系统就不支持了

### 17、描述一下这个项目，你是如何设计UML的？

这个项目我用到了UML中的类图、时序图和状态机。

~~一开始是从类图入手的。一般是依据架构、功能来设计类~~

~~比如UI、控制层、模型层、通信层、命令处理层等等~~

~~给出名称、大致的接口、预期的参数和返回值~~

~~然后就是依据业务逻辑开始画时序图~~

~~从测试业务、驱动读取业务、文件读取业务、远程监控业务等等开始一条条的整理~~

~~在具体的业务处理过程中，必然会有程序状态转换的情况~~

~~这个时候就画一下状态转换图来梳理一下思路~~

~~这样整个项目的UML就差不多出来了~~

~~如果是其他项目，用类似的方法也可以一样实现。当然如果涉及数据库，可能还需要补充数据流图和数据词典。把数据库的逻辑也整理出来。~~

【

类图是静态图，用来辅助设计的，是详细设计阶段我们需要做的事情。

在这个阶段，技术选型已经完成，系统架构也大体拟定，我们需要更为详细的模块设计。

类图可以将模块细化为各个类，并且确定一些重要的接口和属性。

初步完成类图之后，重要的接口和属性已经定下来了，

接下来我们就需要参考业务逻辑来画时序图。

比如测试接口、驱动读取、文件上传和下载、远程监控等等业务。

此时一些细节的内部和外部接口，会随着时序图的深入而被发现和确定。

各个接口的参数，也会初步拟定。

当时序图遇到一些复杂的情况，我们可能还需要状态图来帮助我们梳理一下状态

比如远程状态有哪些，各个状态之间是如何转换的，转换的条件和接口是什么？

通过状态图，我们可以把模糊的状态明确化，已知状态的转换确定下来。

然后在使用状态图，完善时序图，明确类图当中的接口参数和属性。

这样整个项目的UML就逐步完成了。

如果项目当中涉及到数据库，则可能还需要补充数据字典、数据流图。

明确数据项的含义、范围和类型，

梳理出数据如何产生、使用和存储的。

然后在结合时序图，把类图也相应的进行完善。

最后把数据库的逻辑也纳入到UML的描述之中。

】

### 18、请描述一下MFC的消息机制

1 消息机制：（**是什么？名词解释**）

消息简单的说就是指通过输入设备向程序发出指令要执行某个操作。具体的某个操作是你的一系列代码。称为消息处理函数。

2 为什么MFC要引入消息映射机制？（**为什么**？）

一个程序是否拥有多个窗体，主窗口就算只有一个，那菜单、工具条、控件这些都是子窗口，那我们需要写多少个switch case，并且还要为每个消息分配一个消息处理函数，这样做是多么的复杂呀。因此MFC采用了一种新的机制，利用一个数组，将窗口消息和相对应的消息处理函数进行映射，可以理解成这是一个表，这张表在窗口基类CWnd定义，派生类的消息映射表如果你没有动作它是空的，也就是说如果你不手工的增加消息处理函数，则当派生窗口接受一个消息时会执行父类的消息处理函数。简单地说就是大大提升了效率。

3 步骤（**怎么做**）

步骤如下：

第一步：定义消息。Microsoft推荐用户自定义消息至少是WM\_USER+100，因为很多新控件也要使用WM\_USER消息。

#define WM\_MYMESSAGE (WM\_USER + 100)

第二步：实现消息处理函数。该函数使用WPRAM和LPARAM参数并返回LPESULT。

LPESULT CMainFrame::OnMyMessage(WPARAM wParam, LPARAM lParam)

{

// TODO: 处理用户自定义消息，填空就是要填到这里。

return 0;

}

第三步：在类头文件的AFX\_MSG块中说明消息处理函数：

// {{AFX\_MSG(CMainFrame)

afx\_msg LRESULT OnMyMessage(WPARAM wParam, LPARAM lParam);

//}}AFX\_MSG

DECLARE\_MESSAGE\_MAP()

第四步：在用户类的消息块中，使用ON\_MESSAGE宏指令将消息映射到消息处理函数中。

ON\_MESSAGE( WM\_MYMESSAGE, OnMyMessage )

### 19 windows下面ansi版本和unicode版本的函数有何区别？

一．**定义部分**：   ANSI：char    str[1024]; 可用字符串处理函数：strcpy( ),    strcat( ),   strlen( )等等。 UNICODE：wchar\_t    str[1024]；可用字符串处理函数

二．**可用函数**：  ANSI：即char，可用字符串处理函数：strcat(    ),strcpy(    ),    strlen(    )等以str打头的函数。       UNICODE：即wchar\_t 可用字符串处理函数：wcscat(),wcscpy(),wcslen()等以wcs打头的函数。

三．**系统支持**         Windows    98    ：只支持ANSI。          Windows    2k    ：既支持ANSI又支持UNICODE。          Windows    CE    ：只支持UNICODE。

四．**如何区分**：

在我们软件开发中往往需要即支持ANSI又支持UNICODE，不可能在要求类型转换的时候，重新改变字符串的类型，和使用于字符串上的操作函数。为此, 标准C运行期库和Windows 提供了宏定义的方式。

在C语言里面提供了 \_UNICODE宏（有下划线），在Windows里面提供了UNICODE宏（无下划线），只要定了\_UNICODE宏和UNICODE宏，系统就会自动切换到UNICODE版本，否则，系统按照ANSI的方式进行编译和运行。

### 20 假设在该项目中有一个函数处理了一个事件，但由于某种原因处理事件的时间非常长，这会导致该函数在很长时间内一直占用处理器资源，你会如何优化这个问题？

如果该函数处理事件的时间非常长，会导致该函数在很长时间内一直占用处理器资源，这会导致系统性能的下降。为了优化这个问题，我会尝试以下的解决方案：

1. 将该函数中的耗时操作放在独立的线程中进行，在主线程中只是启动和控制该线程的运行。这样可以避免主线程被阻塞，提高系统的性能。

2. 将该函数中的操作分解成多个步骤，每个步骤在处理完后调用一次系统调度函数，释放处理器资源，让其他线程有机会运行。这样可以让系统的资源得到更好的分配和利用，提高系统的处理能力。

3. 将该函数中的操作分解成多个小的操作，尽量减少每个操作的执行时间，将每个操作放到自己的处理流程中。这样可以提高系统的处理能力，并且可以及时响应用户的操作，增强系统的交互性。

### 21. 远程桌面控制软件中，为了兼容网络不稳定的环境，你会如何处理数据丢失和重传问题？

答：为了兼容网络不稳定的环境，需要对数据丢失和重传问题进行处理。在远程桌面控制软件中，我会尝试以下的解决方案：

1. 对数据包进行编号和确认机制，实现数据传输的可靠性和正确性。在发送方发送数据包时，将每个数据包赋予一个独立的编号，接收方在接收数据包后向发送方发送一个确认包，表示该数据包已被正确接收。如果发送方接收不到确认包，则说明该数据包丢失，需要重传。通过这种方式可以实现数据传输的可靠性和正确性。

2. 设置超时机制，当发送方在一定时间内没有收到接收方的确认包时，就会重新发送该数据包。在接收方接收到重复的数据包时，只处理最新的数据包，避免重复数据的处理。

3. 对于大容量的数据传输，可以采用分块传输的方法，每次只传输一个小块，当接收方确认了该小块数据时，再传输下一块数据。这样可以避免因为一个数据块传输失败而导致整个数据传输失败。

## 2、网络天气预警系统项目

### 1、能描述一下你做整个项目的过程吗？？（项目的包装）（1）

这个项目是我一个在某个研究所的同学（朋友）介绍给我的，需要特定的一个天气预警系统，而且必须规定用C++开发，而且还要有界面，所以我选择了QT，之前都是他在做，后面因为一些原因，他没有做了，当时没有办法，就找到了我，其实这个项目也挺简单的，其实就是要一个接口数据，刚好这个接口数据，直接用http就可以拉下来，而且那个研究所里面的一些硕士，基本上也不懂编程，（整天吹牛逼，实际上啥都不会），刚好我对这块也蛮熟悉的，一个http就拉下来数据，画个界面显示就行了，然后就是警示一下天气，预警一下天气会有什么问题，日出日落这玩意儿，反正我觉得技术上没什么难度，于是我就接了我师兄（朋友）的这个项目。

主要是绘图那块有点小复杂，研究所那般人，（真的比较扯淡），对美工要求比较高，我一个大直男，确确实实比较困难，但最后还是交付了，时间也比较短，大概也就一个多月做完了。

### 2、你在项目里面遇到什么问题，是怎么样解决的？

这个项目要处理的数据项比较多，包括城市、日期、风力、风向、感冒指数、通知、pm2.5、空气质量、温度、湿度、日出日落时间等等

不同的数据，不论是单位、有效值范围、数据类型、展示方式等等，都各不相同

而且今日数据和历史数据、预告数据也有些微的差异

所以我设计了两个数据类来处理这些数据

重载了=运算符，使得这些数据类可以直接从Json数据提取值

同时又可以对外提供所有需要的数据类型。

### 3、什么是Qt对象树，使用对象树模式有什么好处？

对象树是指父对管理子对象的列表

我们创建一个子对象时，如果指定了父对象，那么父对象里面有一个QList会记录这个子对象

在父对象析构的时候，子对象也会跟着析构

在UI编程的时候，很多UI控件都是临时创建的

对于这些控件会有一个比较麻烦的问题

就是何时析构这些控件

如果没有对象树模式，这会是一个比较头疼的问题

析构早了，会内存访问失败

如果没析构，内存又会泄漏

对象树则很好的解决了这个问题

### 4、这个项目你是如何实现自绘控件的，请具体描述一下？

~~Qt绘制的时候，会有一个Paint消息。~~

~~我们定义eventFilter事件过滤函数~~

~~当检测到自定义绘制控件和Paint消息时，开启自绘步骤~~

~~通过QPainter获取控件的绘制环境~~

~~然后使用QPen和QPainter来绘制图形~~

Qt是基于信号和槽在运作的体系，

所以在要绘制的时候，会有一个Paint事件信号产生。

我通过设置事件过滤器状态，

在eventFilter事件过滤函数中可以捕获到该事件

当检测到自定义绘制控件和Paint事件信号时，

通过参数定位到目标控件的绘制事件，开启自绘步骤

自绘的时候，通过QPainter获取控件的绘制环境

然后使用QPen和QPainter的绘制接口来绘制图形

完成后，绘制结果就会出现在UI界面之上。

### 在这个项目中，你是如何集成HTTP接口的

Qt有QNetworkAccessManager这个网络访问模块

创建这个manager之后，连接其finished信号

然后就可以使用manager的get接口去请求http服务器

请求完成后，finished信号会触发，进而进入我们写的槽函数

槽函数中使用QNetworkReply的readAll接口，即可获取网页内容

### 描述一下，如何解析json的数据，Qjson是如何使用的

Qt当中，我们使用QJsonDocument、QJsonObject、QJsonValue、QJsonArray来完成json的解析。

首先使用QJsonDocument的fromJson接口，将http服务器读取到的字节转换为QJsonDocument对象。

然后使用QJsonDocument的object接口拿到解析出来的根节点QJsonObject对象root

然后使用QJsonObject的value接口获取值

对于数组，则使用QJsonValue的toArray，来转换为QJsonArray对象来处理

### 8、请描述一下Qt多线程同步的几种实现方式

有以下几种方式：

1 使用互斥体QMutex

2 使用读写锁QReadWriteLock

比如对于映射表，多线程读取并不会产生什么问题

但是写入会产生严重问题。就可以设置一个写入锁。

而对于一些数据，多线程取就可能产生问题，那么就可以设置一个读取锁。

### 9、此项目的温度曲线你是如何实现的？

首先，在事件处理过滤函数中设定条件，如果温度控件和绘制消息同时满足，则开启曲线绘制

接着，创建QPainter对象，进行全局设置，例如反走样设置

然后，获取温度数据，依据数据绘制温度曲线

注意使用QPainter的save和restore接口来处理多个线条的绘制操作

绘制之前进行save，绘制完成之后，使用restore来恢复一些默认设置

### Qt中的信号能绑定多少个槽函数？槽函数能否绑定多个信号？

Qt中的信号和槽如果参数能匹配的情况下，是多对多的关系。

即一个信号可以绑定多个槽函数，一个槽函数也可以用于多个信号的绑定。

当一个信号绑定多个槽函数的时候，一旦信号触发，这些槽函数会依次触发。

而当一个槽函数绑定多个信号函数的时候，任意一个信号被触发一次，槽函数就会响应一次。

### 11、比较一下Qt和MFC的优缺点

跨平台性 Qt优于MFC

Qt支持Linux、iOS、Android和Windows等主流系统

而MFC只支持Windows

易用性 Qt优于MFC

Qt的界面采用了stylesheet来描述，对外观的呈现更加容易和轻便

而MFC要呈现出一些外观，需要通过C++代码来达到

此外Qt的信号和槽机制也比较直接和方便

而MFC的消息和消息函数则比较麻烦

开发环境 MFC优于Qt

Qt开源版本的开发环境搭建非常复杂，依赖很多第三方软件

而MFC使用VS2019，一次安装就可以一直使用

稳定性和性能 MFC优于Qt

MFC是微软提供的商用代码模块，经过了二十多年的开发和完善

有着Qt无法比拟的开发和测试环境，对Windows的兼容性有者无与伦比的天然优势

而Qt真正进入大众视野是从2005年的qt4才开始，

大规模应用则是从2012年的Qt5才开始

在用户规模，底层机制和操作系统的匹配性上，和MFC都有很大的差距

## 3、音视频项目

### 1、音视频项目的过程和场景？？（项目的包装）（最重要）

实际工作的时候，我们往往需要提供一些视频播放服务。

如果直接使用文件传输到本地播放，那么难以做到实时性。

因为文件传输往往要求不能缺失，完整性会超过传输速度要求。

而视频服务实时性可能会超过完整性要求

此外对于直播类的场景

用户往往希望内容是最新的状态，而对已经发生的事情没那么关注。

而这块集大成者就是RTSP

包含了RTP、RTCP、SDP

其他协议没有该协议全面和包容

### 你在这个项目里面有遇到什么问题？是怎么解决的

最大的麻烦就是协议联调的时候，有时候不知道是客户端还是服务器的问题

后面先拿客户端调试，保证客户端可以播放各种网上的视频之后

再拿客户端和服务器联调

这个时候出问题，就基本可以确认是服务器的问题了

此外日志模块是非常重要的模块

一些细节调试上，往往需要仔细阅读日志，来寻找错误的具体原因

甚至要分析前后的协议内容、客户端和服务器的协议数据

才能定位出错误的真正原因。

比如SDP描述媒体信息的时候，可能因为一个空格没有对齐，导致目标无法播放

或者参数错误。

这个时候只能耐下性子，一点点的看日志，找到不正常的数据

然后再去代码里面修改。

### 为什么要有视频服务器，你采用的是什么架构？

视频对网络带宽的占用是比较高的。如果所有的服务都是在一台服务器上面去做，那么没有任何一台服务器，在访问人数走高的时候，可以承受这么高的带宽消耗。所以一般视频播放有专门的视频服务器，而页面访问有专门的网页服务器。

视频服务器一般采用经典的C/S架构。服务器内部对用户的连接，采取了会话机制。因为除了一些视频信息的传递之外，还需要有视频数据的传递。所以，采用会话机制，可以很好的处理前后通信交互的数据。

此外，如果同时在线用户人数较多的时候，还需要一些分流服务器来帮忙分流。

分流服务器可以看作一个客户端和服务器的集合体。一边从源服务器获取播放内容，一边把这些内容发送给客户端。

### 4、请描述一下h264是如何分片的？

一般视频数据都是流媒体格式。这些视频文件往往都是由一个个的帧组合起来的。

所以我们对H264是依据帧进行分片。只是需要注意一下，有的帧比较大，有的帧比较小。

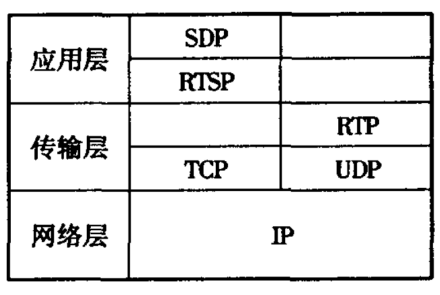
对于比较小的帧还好说，一个RTP包就可以发送出去。但是对于比较大的帧，则需要按照RTP协议进行分包发送。分包的时候，有头、尾和中间数据包的区别。在每帧头部两字节里面进行说明。

### 5、描述一下RTSP协议？

RTSP是实时流协议，包含RTP和RTCP两个子协议

RTP用于数据流传输，是基于UDP协议来的

RTCP用于传输控制，是基于TCP协议来的

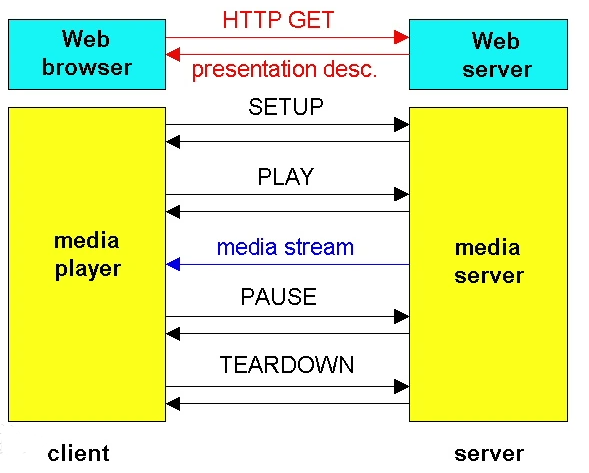


整个协议是基于文本的协议，有点类似于HTTP协议

每个请求会基于会话进行，会话常见流程如下：

1. OPTIONS 查询服务器支持的命令
2. DESCRIBE 查询服务器信息
3. SETUP 协定通信端口和信息
4. PLAY 播放
5. TEARDOWN 关闭

通信示意图如下：



【补充知识：

当然，RTSP不仅仅只有上面的命令，还包括

PAUSE 暂停播放

RECORD 记录

REDIRECT 重定向

ANNOUNCE 描述

SET/GET\_PARAMETER 设置/获取参数

】

### 6、描述一下SDP格式？

SDP是用来描述媒体文件的格式，可以描述媒体的帧率、格式、协议、播放时长等等信息

SDP由多行的key=value组成

完整的sdp包括**会话元数据**、**网络描述**、**流描述**、加密描述、Qos和分组描述

但是一般会话描述、网络描述和流描述（也叫媒体描述）是必须的，其他的可选

v=开始都是会话描述

m=开始都是媒体描述

t=开始时时间描述

剩下的有很多，完整的参数说明有好几页，一般都是使用的时候，需要用到了再去查。

### 7、你是如何集成VLC库的？

Vlc官网提供vlc的sdk

一般会有libvlc.dll、libvlccore.dll和对应的lib文件

此外要注意一点，需要将对应版本的plugins文件夹复制到运行文件夹去，否则可能导致vlc初始化失败。

然后复制对应的include文件

引用vlc.h文件

即可以集成vlc库到项目

### 你是如何实现视频流的队列的封装的？

视频流队列是用来缓存视频，提高用户体验的一种技术。

我使用了vlc库播放视频，并不需要考虑这些底层的细节。

~~客户端使用vlc库播放视频的时候，可以不用考虑这些细节。~~

Vlc会给我们处理好这些底层的逻辑。

~~我们要做的就是准备好播放环境，设置好播放参数，即可进行播放。~~

~~而服务器端，我们通过会话来保证用户和用户之间的视频数据不会冲突~~

当我设置好服务器播放地址之后，客户端会连接服务器，

并按照RTSP协议从服务器获取视频信息和参数

然后接收数据，开始播放。

由于协议本身包含会话机制，同时传输文件的时候会约定端口

所以服务器端也无需担心数据传输给错误的对象

不同的客户端，会分配不同的端口进行视频数据的传输。

### 9、客户端的调整播放进度是如何实现的？

视频播放进度是由一个播放位置的参数来确定的。

正常播放的时候，这个参数会不断发生变化。

修改这个参数，播放位置就会发生变化，从参数指定的位置开始播放。

在vlc里面，这个参数的修改是通过libvlc\_media\_player\_set\_position接口来实现的。

我将它封装到了vlc类里面，另外加了一些判定条件。

因为这个参数仅仅在播放状态下才会有效

如果视频播放停止，或者尚未开始，设置该参数将毫无作用

~~当我开始播放视频的时候，通过函数libvlc\_media\_player\_set\_position即可设置播放进度。~~

~~但是要注意，如果没有视频正在播放，那么这个设置可能导致问题。所以最好的办法是封装这个播放器函数到播放器类。然后在接口中判断一下当前媒体的状态~~

~~如果不正常，则不做任何操作，防止程序出现意外。~~

### 10、在项目中你是如何使用MVC设计模式的？

这个项目的客户端采用经典的MVC架构。

V层用来展示视频以及和用户交互操作

C层用来实现页面的业务逻辑

M层则直接通过vlc和具体的媒体打交道

这样C层可以专注于业务的实现，并且可以操控V层和M层的各种状态

M层的代码专注于vlc的封装，具有了可移植性，可以移植到其他项目中去继续使用

而V层则专注于界面的展示，无需考虑其他的事情

### 你在做视频服务器的过程中有什么指标数据，如何优化的？

视频服务器一般都是以并发人数来衡量的。

考虑到服务器大部分的消耗都是在带宽和线程上

所以一般是将服务器由普通的网络通信

改为基于异步网络的通信模式

此外，就是提升服务器带宽、硬件来提升

当然，也可以通过提供压缩率更高的媒体源来增加并发量

### 12、RTSP有哪些常用的方法，请举例说明？

RTSP有很多方法，常用的有OPTIONS、DESCRIBE、SETUP、PLAY和TEARDOWN方法

其中OPTIONS是连接的时候，第一个方法

用来告诉客户端服务器的协议版本等信息

DESCRBE则告诉客户端，提供的媒体播放信息

SETUP则是协议数据交换

PLAY是播放

TEARDOWN是关闭连接

### 13、RTP的数据格式，细节描述一下RTP协议？

RTP数据帧由RTP头和数据负载构成

头里面包含版本、负载的类型、序列号、校验、时间戳、同步信源、特约信源列表等等

数据负载格式是有两个字节的头，和视频帧数据。

这两个字节的头，第二个字节取帧数据的低五位。最高位表示开始，次高位表示结束

第一个字节一般是0x7C

主要了解协议如何组帧，如何发大帧，如何发小帧即可。

帧内容一般使用视频帧填充。

大帧会拆分发送，小帧可以合并发送。

拆分的时候，会有帧头来标记起始帧，中间帧和结束帧。

方便接收端还原帧数据。

### 14、服务器的稳定有哪些常见指标？

一般我们用连续无故障运行时间T来衡量。

这个T越大，表示服务器越稳定。

这个时间是指服务器上一次发生故障，并修复之后

到下一次发生故障之间的时间

当然，考虑到各种故障间隔可能不一样

也可以在这个基础之上，追加一个平均无故障运行时间

记录多次，取平均值

此外还有可靠性指标也可以衡量

比如3个9的可靠性，意味着服务器连续运行一年，这一年中累计的中断时间不能超过8.76小时（1-99.9%）\*365\*24 = 8.76小时

4个9的可靠性，意味着这个时间是52.6分钟 365\*24\*60/10000

5个9的可靠性，意味着这个时间是5.26分钟 365\*24\*60/100000

6个9的可靠性，意味着这个时间是31秒 365\*24\*60\*60/1000000

### 15、视频数据是如何解析的？请结合你的项目说明

我的项目播放的是H264格式的视频。服务器无需完整的解析该格式，只需要知道该格式的帧头即可成功解析。

因为服务器无需播放，只需要发送就行。

H264的头是以字节0开始的，分两种情况

一种是三字节头，第一个字节是1

一种是四字节头，第一个字节是0，第二字节是1

通过找到两个头，从而可以知道其之间的帧的长度

最后一帧的长度则是剩余文件的长度

## 4、易栈项目

### 1、聊一聊此项目的架构？

简要概括

是一套基于非阻塞的、反射式、高性能TCP通信库的服务器项目。

采取业务层、通信层和数据库分离的设计。

通信层通过回调来激活业务层

业务层依据需求来调用数据库模块完善业务。

此外消息服务器和文件服务器也进行了分离

避免了大文件传输的时候，对消息传输的干扰。

### 2、说一下一次完整的HTTP请求过程包括哪些内容？（1）

第一种回答

建立起客户机和服务器连接。

建立连接后，客户机发送一个请求给服务器。服务器收到请求给予响应信息。

客户端浏览器将返回的内容解析并呈现，断开连接。

第二种回答

域名解析 --> 发起TCP的3次握手 --> 建立TCP连接后发起http请求 --> 服务器响应http请求，浏览器得到html代码 --> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） --> 浏览器对页面进行渲染呈现给用户。

### 木铎库是什么库，其优缺点有哪些？

木铎库是陈硕针对Linux网络编程而开发的网络库。

Muduo库是一套小巧的、支持多核多线程的、线程安全的、非阻塞的网络库。

优点是接口简单、性能高、支持线程安全、可扩展性强。

缺点是不支持window、不支持IPv6、不支持UDP、没有考虑公网下的安全性。

### 为什么选择木铎库做这个项目

接口相对简单，性能高。

虽然不支持UDP，但是消息通信本就是可靠性要求高于传输速度的。

聊天服务器，主要是转发聊天消息为主，所以需要的功能并不算太复杂。

一个接口简单，基于tcp协议的，高性能服务器，显然是一个不错的选择。

### 能简述一下protobuf的是什么，以及怎么使用的吗？

Protobuf是一个支持多语言、多平台的协议传输设计

先要设计消息内容，完成配置文件proto

定义消息中的必备、可选字段，字段类型

也可以定义子消息（嵌套）

然后使用protoc将配置文件生成源码

然后将源码添加到项目当中，即可引用该协议类

### ORM是什么？为什么要用？如何实现的？

ORM就是对象关系映射

常用的数据库是关系数据库。里面记录的数据往往都是表结构。

实际使用的时候，这些数据比较零散，一些关联的数据可能分散在多个表之中。

查询、修改起来非常麻烦。

利用ORM可以将这些数据进行整合。

查询、修改都会变得方便。对于面向对象设计，也是非常方便。

一般会依据数据对象设计类，将紧密关联的数据聚集成类

将这些数据的增删改查过程进行集中，事务化处理，保证一致性。

### ORM有什么优点？有什么缺点？

和传统的数据集方式相比，ORM的数据可读性，可操作性，一致性要强很多。

数据集的方式数据零散，多个表的数据一致性存在隐患，修改数据的时候过程繁琐

而ORM可以封装了这种繁琐操作，使用简便，简洁直观

由于ORM需要将查询的数据封装到对象，所以在大规模数据查询的时候

对查询效率有巨大的影响

对大规模（超过一万条）数据的查询，最好不要使用ORM，否则封装过程会消耗大量资源和时间。对服务器性能有比较负面的影响。

### ORM的难点在什么地方？

最难的点在于如何设计数据类

如果设计不好，这个类会成为一次性的类

虽然有一点ORM的好处，但是大大增加了开发成本

如果类太过通用，和数据集模式相比，优点就不会那么明显了

而且不同数据库，sql语句、增删改查过程也略有不同

如果要做适配，也会比较困难

### 能描述一下epoll的工作模型/工作原理/工作流程吗？

参考epoll基础课

### 反射式设计是怎么一回事？能简单聊聊吗？

如果两个系统，相互之间有依赖，利用函数回调来实现反射式设计，可以大大提高整体性能。

比如业务处理模块和网络通信模块，属于不同的两套系统。

业务模块依赖网络模块。如果设计接口让两者相互调用，会因为相互等待而降低系统性能

比如网络模块收到包交给业务模块，业务模块需要查询数据库等待结果

这会让网络模块性能大降

但是如果采用反射式设计，网络模块收到数据，不是直接调用业务模块

而是通过回调把数据包传递给业务模块，然后就立刻返回。

对网络模块的性能就几乎没有影响了。

业务模块收到包之后，自己找工作线程去处理包，无论耗时多少，都不会影响网络模块了。

### 你在项目开发的时候遇到哪些问题？是如何解决的？

~~参考项目课程~~

我遇到的最难的一个问题是项目联调的问题。

由于核心模块采用的是多线程的Reactor模式，

所以一旦出现问题，

往往搞不清是哪个客户端连接导致的

或者进行到哪个步骤了

下断点调试，这种会导致其他线程处理出现问题

## 易播加密播放器

### 简单讲一下加密播放器的设计/实现？

易播采用MVC架构进行实现的。

V层主要是Qt的UI模块

C层主要是利用信号和槽机制来实现

M层主要是通信数据和视频数据

和MFC不同，有信号和槽机制，C层比较精简

此外VLC也进行了封装，对外提供自定义的信号，对内使用槽来响应外部信号。

### 用户登录是如何验证的/账户密码是如何传递到服务器进行验证的？

用户在客户端输入账号和密码后，通过加密算法，将账号、密码、时间戳、额外字符串进行签名。

然后将签名数据、时间戳、账号名发往服务器。

服务器则通过账号，在本地数据库找到密码，通过加密算法，将账号、密码、时间戳、额外字符串进行签名。

对比客户端的签名和服务器的签名，一致则表示账号密码正确，予以登录通过。

### 视频是如何加密的？为什么要这样加密？

视频数据是采取异或加密。密钥是一张图片（这里采用的应用图标）。

图片被看成一个超级长的字符串密钥。

解密的时候，密钥与视频数据进行异或操作，实现还原。

密钥使用完了，从头开始再次使用，仿佛一个环。

不同视频的密钥起始点不一样，避免了不同的视频密钥一样的问题。

图片大小合适的情况下，这种加密方法可以抵抗多样本分析。

此外异或操作消耗的CPU资源较少，不会影响视频的播放效率。

### 如何提高加密视频的安全性？

图片可以通过服务器进行更新，实现密钥的实时修改。

每个视频密钥开始的位置可以和md5关联，使用算法来定位。

而且可以使用非连续的数据做密钥。采用算法，跳跃的使用图片密钥数据。

或者使用非对称加密传递密钥参数，进一步增加密钥安全性。

### 如果破解人员直接拦截网络数据，修改服务器的返回结果，怎么办？

服务器的返回结果可以加入签名数据，保证每一次都不一样。

签名可以通过时间戳、播放视频的签名、客户端ip和mac值、自定义字符串来生成

这样不同的客户端，不同的时间得到的返回结果会不一样

此外应该开辟多个校验线程

应答处理模块不会直接终止或者提示错误信息。

而是校验线程函数来做这些事情

是的校验逻辑和通信接收逻辑分离。两者通过成员数据或者全局数据关联。

增加逆向者找到验证逻辑的成本。

### 加密视频播放器的工作流程是怎么样的？

首先连接服务器，进行登录

登录成功后，会拿到用户信息

然后添加视频进行播放

视频如果是非加密的，则直接播放

如果是加密的，则会访问服务器获取密钥信息

密钥信息和本地密钥结合，加上解密算法

就可以开始一边解析一边播放

### 在这个过程中如何保证安全性？

首先，密钥是数据加算法。既不是单纯的数据，也不是单纯的算法。

其次，每个视频都可以拥有独立的密钥。避免一个视频泄密，其他所有视频都会泄密。

增加破解者的破解成本——需要对所有的视频进行破解

再次，解密后不会将解密数据保存本地进行缓存。

因为采用异或运算解密，再大的视频也可以实时解密

对视频播放不会产生影响，所以无需缓存。

密钥数据是图片，更新和替换非常方便，也具备一定的隐蔽性。

可以更新多张图片，真正的密钥数据藏在其中一张图片上。

### 如何在视频播放窗口上实现半透明提示窗口的？

这个功能确实比较复杂。

Libvlc默认的播放，无法再其上面悬浮半透明窗口

要么不透明，要么全透明。

为了解决这个问题，我先做一个不透明的窗口

然后把被覆盖部分的vlc窗口内容取出来

然后使用Qt的自绘机制绘制将被覆盖的地方绘制到提示窗口

然后再在窗口叠加半透明内容

为了保持内容和播放窗口一致

还追加了一个定时器，定时更新被覆盖的内容。

### 你觉得这个项目最难做的是什么地方？你是如何解决的？

最难的是和服务器联调。

首先不仅仅是需要在业务上要实现用户登录

此外还要和服务器交换密钥信息

每一步都要经过多个环节，从网络到ui

一步错，步步错。

有些地方仅仅靠断点，是没法触发的。

所以我加入了大量的日志内容

好在Qt有QDebug模块

通过QDebug模块，我可以看到数据从服务器接收到

再进行解析、进行拆解、进行传递和进行使用的整个过程

一旦发现问题的时候，就认真读取日志

看看是在哪里出现了问题，并且对照代码进行分析

找到出问题的点。再从这个点开始分析源头，找到真正的原因。

### 自定义信号是如何实现的？

Qt中自定义信号实现非常简单，分两部即可

首先是定义信号接口

包括信号名称，信号参数

然后是在需要产生信号的地方，通过emit来触发信号。

如果信号带参数，那么emit的时候需要将参数填入。

### 你是如何应用自定义信号的？

我在封装vlc库的时候使用到了自定义库

为了方便使用vlc库，我继承了QLabel，派生了一个QMediaPlayer控件

Vlc播放，就直接播放在这个控件ui上面

方便其他UI使用vlc

为了方便其他部分使用vlc，我定义了一些信号

比如播放状态的切换、播放刻度的变化

此外我还定义了一些槽

方便外部控件来进行播放文件路径的设置，播放进度的跳转等等操作

使得vlc库完全可以当作一个Qt普通控件一样使用。

### Vlc库是如何播放加密视频的？

vlc库在播放视频的时候，可以设置回调函数

该回调函数是vlc读取流媒体内容的回调

我们需要做的就是在vlc回调函数里面，将加密的内容进行解密即可

解密完成，vlc会自己尝试读取解密内容

这样我们就完成了加密视频的播放

### 如果有人逆向拦截了vlc库的播放函数，会不会对视频的安全性产生影响？为什么？

有一定的影响，但问题应该不大

首先，真正的解密函数不是vlc库提供的，而是我们自己提供的

所以对方不可能拿到源码（vlc也许他可以拿到代码）

其次，解密后的内容和加密的内容是同一段内存片段

所以通过堆函数的拦截也无法判断监控结果

因为没有额外内存空间的申请

最后，加密的内容是碎片化的，分批的传递进来解密的

并不是一次性大批量的。所以依赖这个解密，工作量并不小。

## 易播服务器（Windows版）

### 简单描述一下易播服务器是如何实现的？

从构成上看，易播服务器是一个基于HTTP协议的加密播放器服务器。

提供用户登录、视频密钥分发等服务。

数据交互除了了HTTP协议之外，还会配合Json来发送额外的自定义数据。

从技术上看，易播服务器是一个基于多进程编程的服务器项目。

有独立的日志进程、用户接入进程、业务处理进程、数据库。

每个进程内部则会采用线程池来处理高并发的业务。

考虑到日志、用户接入和业务处理模块需要有较高的并发处理能力，

这些模块都结合了IOCP进行了优化。

整个项目也尽量不直接使用互斥锁来处理线程同步的问题。

### 自底向上开发和自顶向下开发有什么区别？什么情况下使用？

自底向上开发是从基础的功能单元开始做起。

而自顶向下开发是从整体设计开始做起。

两者做开发，入手的角度不一样，实施的条件不一样，应用的场景也不同。

自底向上开发的团队往往面对的是需求模糊或者需求变化快的项目。

这些项目需求不清晰或者开发期间有较大的可能发生变化，

如果采用自顶向下开发，则会面临需求不断修改，设计跟着修改，实现也跟着大幅修改的情况。

开发进度和开发质量完全无法保证。

因此采用自底向上开发，则可以降低这些风险和影响。

这种开发方式对团队个人开发能力要求较高，对设计的理解能力和变通能力比较强

不适合没有开发经验的团队，或者新人太多的团队去实施。

先做基础功能模块，然后做各个模块的整合，

接着再依据最新的业务需求进行业务层的调整，

最后发布完整的产品。

自顶向下开发的团队面对的是需求明确，开发周期短，开发质量要求高的产品。

由于需求非常清晰明确，而且需求变动不会太频繁或者改动太大，

因此可以整体规划，稳步推进。

相对自底向上的开发方式，这种方式时间可以预期，成本易于估算，进度也非常直观

但是要求开发团队需要有类似或者相同项目的开发经验，或者有相应项目经验的技术经理领导。

### 简单讲一下IOCP的使用流程，其中有什么难点？

IOCP即Input/Output Completion Port，输入输出完成端口，也叫完成端口映射

基本的使用方式是使用CreateIoCompletionPort先创建，

接着使用GetQueuedCompletionStatus来等待状态发生，

然后使用PostQueuedCompletionStatus来投递状态

使用完成之后，使用CloseHandle来关闭完成端口。

比较难的是关联IO对象，这个是IOCP真正的应用场景。

创建完IOCP之后，还需要再次使用CreateIoCompletionPort，来绑定IO对象的句柄

注意，这个时候IOCP还不会响应IO的事件

还需要搭配其他的接口来向IOCP注册需要响应的事件

对于网络套接字，

比较常见的是AcceptEx、ConnectEx、RecvEx、SendEx、WriteEx、ReadEx

这些函数会立刻返回，调用之后

对应的事件发生，IOCP就会进行响应

这个过程比较反直觉。

函数调用的时候，这些事件不会立刻发生，函数会立刻返回

直到事件发生的时候，GetQueuedCompletionStatus才会得到返回值

这种方式完成任务是一种Reactor模式，并非线性的过程，不太容易理解。

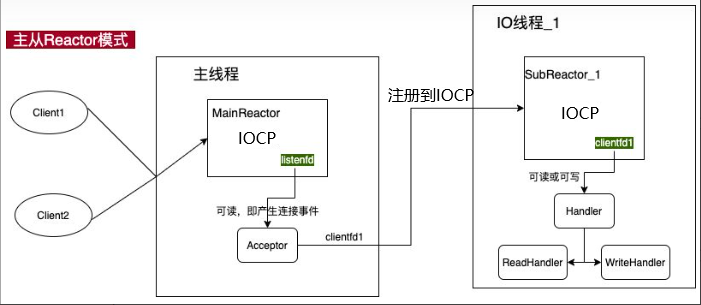
整个过程中，如何封装数据，保持前后通信的一致性也是非常复杂的事情。

不过好在微软提供了重叠IO参数，以及对应的工具宏

可以在传递重叠IO参数的时候，也可以传递关联对象。

这样我们可以将触发操作类型、相关对象、会话上下文绑定在里面。

### 什么是reactor模式？如何在项目中使用的？(参考Linux版本)



### 业务模块极度依赖底层通信，这两者你是如何解耦的？(参考Linux版本)

### 可变参数函数、可变参数宏和可变参数模板有什么区别和联系？(参考Linux版本)

### 可变参数模板如何展开模板参数？(参考Linux版本)

### 为什么你的线程类、任务类可以接入任意函数做线程函数？(参考Linux版本)

### 线程池是如何实现的？线程任务是如何分发的？(参考Linux版本)

### 你的服务器业务中是否使用到了数据库？是如何使用的？(参考Linux版本)

### 用户登录是如何实现的？你是如何保证密码安全的？(参考Linux版本)

### 你是如何设计业务模块的？为什么？(参考Linux版本)

### 服务器开发的难点在哪里？你是如何解决的？(参考Linux版本)

### 服务器调试要注意什么事项？(参考Linux版本)

## 易播服务器（Linux版）

### 简单描述一下易播服务器是如何实现的？

从构成上看，易播服务器是一个基于HTTP协议的加密播放器服务器。

提供用户登录、视频密钥分发等服务。

数据交互除了了HTTP协议之外，还会配合Json来发送额外的自定义数据。

从技术上看，易播服务器是一个基于多进程编程的服务器项目。

有独立的日志进程、用户接入进程、业务处理进程、数据库。

每个进程内部则会采用线程池来处理高并发的业务。

考虑到日志、用户接入和业务处理模块需要有较高的并发处理能力，

这些模块都结合了epoll进行了优化。

整个项目也尽量不直接使用互斥锁来处理线程同步的问题。

### 简单讲讲你是怎么使用的Epoll？

我采用了两种方式使用epoll

一是利用epoll的多线程安全特性，分发线程池任务

线程池任务分发要求尽量无锁，而且是空闲线程优先。

线程执行完一次任务后，就会进入epoll\_wait，来等待新的任务

这样，当多个任务并发产生的时候，进入wait的空闲线程就会接取任务，进行执行。

而当没有线程空闲的时候，请求会在epoll队列之中进行等待

这些都是系统自己做到的，无需我们去写一个有锁队列去排队

二是利用epoll的高性能特性，对外服务客户端

相对于传统的io方式，epoll的优势有几点：

1. 首先没有使用mmap来进行数据交互，而是直接进行内核与用户空间的交互
2. 最大并发IO数量超过其他IO接口函数
3. IO效率不会因为IO数据增加而显著下降

对于服务器来说，这些优点难以替代

### 简单讲讲Epoll和IOCP的异同

相同点：

都是用于高性能io的函数

都可以使用reactor模式来封装

都比select、poll模式性能更高

不同点：

IOCP往往配合重叠IO来实现

Epoll则是通过额外参数传递来实现

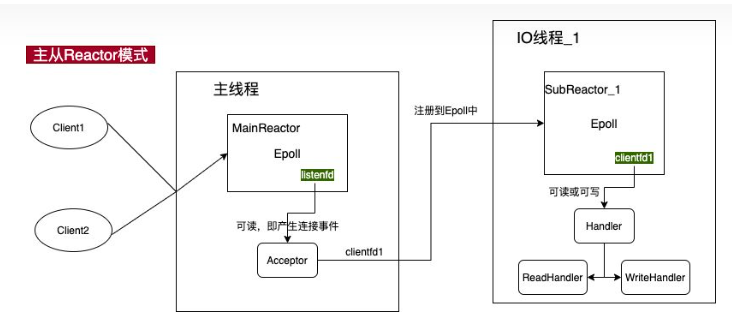
IOCP可以手动通过接口函数来触发事件

Epoll则无法做到，必须依赖绑定的io对象来触发事件

IOCP是Windows下使用

Epoll只能在Linux下使用

### 什么是reactor模式？如何在项目中使用的？



Reactor模式也叫反应器模式、应答者模式。

这种模式是一种基于事件驱动的设计模式。

一般包含请求和处理两个部分。

请求和处理的个数并不唯一，但是至少各有一个。

这种模式常用于处理并发请求，并将请求提交到一个或者多个处理程序的设计模式。

请求可以是高并发的，但是处理器的数量一般是有限的。

优点是可以在处理器数量范围内进行并发处理

缺点是如果并发请求超过处理能力，会造成阻塞。

实际工作的时候，处理器的处理一般不会阻塞或者占用大量CPU时间

这样可以保证强大的并发处理能力

但是如果处理器的处理需要耗费时间的时候，

一般会将处理剥离开来，变成异步处理模式，消解阻塞导致的并发能力的下降。

有时候会用主从Reactor模式来应对这种情况

以网络通信服务器为例

主Reactor不断接入客户端，然后将读写加入从Reactor

从Reactor处理读写这些稍微费时的处理

如果读写处理也非常费时

这个时候可以启用异步操作，将请求发往异步处理。

等处理完成后，通过回调或者查询，把后续处理拉起

这样可以大幅提高系统的吞吐能力

### 业务模块是如何与通信模块交互的？

业务模块是被通信模块驱动的

所以我们只需要在客户端连接、数据完整接收到、数据发送成功、客户端断开连接

这四步关键节点通知业务模块即可

通知的方式采取回调函数

其他的状态，业务模块无需参与通信模块。

为了方便使用，我定义了一个业务接口类（纯虚函数定义的接口）

强制业务模块实现四个接口函数，方便通信模块回调。

使用的时候，将业务对象传递给通信模块即可。

### 可变参数函数、可变参数宏和可变参数模板有什么区别和联系？

可变参数函数是参数个数可变的函数。

原理是通过栈操作来展开函数参数，

可变参数宏是参数个数可变的宏

参数会在编译的时候，原地原样进行文本展开

可变参数模板模板参数可变的模板

参数可以利用模板规则展开

都是可变参数函数，都支持参数可变的函数调用

可变参数宏往往会配合可变参数函数，或者可变参数模板一起使用，极少独立使用。

可变参数函数因为是利用栈的特性来传递和展开参数，所以会在一些细节的地方产生问题。

比如所有低于4字节的整数类型，会升级为4字节类型。

例如char、short会变成int

这在某些情况下，会导致一些严重的问题

可变参数宏只是单纯的文本展开，

如果没有可变参数函数或者可变参数模板函数对接

就只能用于有限度的可变参数条件

比如存在参数默认值的情况，或者函数重载的情况

可变参数模板是比较好的可变参数方案，缺点就是开发难度较大

首先，这种方式在进行参数展开的时候，不会出现类型升级的问题

其次，这种方式可以处理设计时未知的数据类型

可变参数宏可以处理未知的类型，但是这种处理是简单的机械的展开达成的

可变参数函数只能处理预定义的类型，完全未知的类型，会出现解析问题

最后，可变参数模板还可以通过特例化来处理一些类型之间不同的处理方法

比如字符串比较、整数比较、小数比较、自定义类型之间细微的差异

从而统一不同类型对外的接口和行为。

这是前面两者无法做到的。

### 可变参数模板如何展开模板参数？

可变参数模板的模板参数，有两种展开方式：

一是利用递归进行逐级展开

这种方式原理是

先定义一个无模板参数的接口，

同时重载一个首参数独立模板类型，其余参数可变参数模板的接口

这样通过这个接口来处理可变参数模板，

然后产生递归效应。

举例：

template<typename ... ARGS>

Void func(ARGS... args){}

template<typename HEAD,typename ... ARGS>

Void expand(HEAD head,ARGS... args){}【1】

Void expand(){}【2】

那么调用

expand(args...);

编译器就会按照【1】来展开，直到没有参数的时候，就会调用【2】

这个过程是自动推导的

第二种是利用数组初始化来自动展开

举例：

template<typename ... ARGS>

void func(ARGS... args){}

template<typename T>

void dealone(T arg) {}

需要展开的时候，这样：

Int \_[]={(dealone(args)...,0)};

假如这个时候可变参数是a,b,c,d

则，上面的代码会被展开为：

Int \_[]={(dealone(a),0),(dealone(b),0),(dealone(c),0),(dealone(d),0)};

然后dealone会依次处理各个参数

你可以通过dealone来计数，统计参数个数

也可以通过其他操作来处理这些参数

### 为什么你的线程类、任务类可以接入任意函数做线程函数、任务函数？

我们的线程类和任务类确实可以接入任意类型的函数

包括全局函数、静态成员函数、非静态成员函数。

这里我是利用了模板编程实现的。

通过模板参数来承接不同类型的函数指针和函数参数

利用std::bind模板进行保存

最后重载()运算符对外开放调用

这里要特别说明的一点是，如果是成员非静态函数，需要传一个对象指针做第一参数

因为非静态成员函数，采用的是\_\_thiscall的传参规则

该规则规定，首参数必须是this指针的值，也就是对象的值

因为是用模板传递的函数参数，所以无论\_\_stdcall（全局函数常用）还是\_\_thiscall（非静态成员函数常用）调用约定，都可以使用。

但是\_\_thiscall约定有一个隐藏的this参数需要传递，所以非静态成员函数需要额外补足这个参数。

### 线程池是如何实现的？线程任务是如何分发的？

线程池类我是这样设计的：

对外，提供设置、开启、添加任务和关闭接口。

使用的时候先设置线程池（通过构造函数），

然后开启线程池（这个时候会真正启动线程池中的线程）

然后在需要的时候，给线程池指派任务

当不再需要线程池的时候，就调用关闭接口

彻底关闭所有线程，并释放资源。

对内，我先封装了一个线程类来处理线程

然后我创建了一个基于epoll的任务分发系统

该epoll绑定了一个本地套接字

每个线程都会有一个epoll\_wait等待

当有外部的任务需要执行的时候

会将任务对象复制到堆上，然后将堆地址发送到本地套接字

此时处于epoll\_wait状态的线程就会激活

然后接收到该地址，接着处理该任务

处理完成之后，会再次进入epoll\_wait，等待下次任务

当线程池要关闭的时候，关闭套接字、关闭epoll对象

然后关闭线程，并释放线程池资源即可

### 你的服务器业务中是否使用到了数据库？是如何使用的？

我的业务使用了两套数据库。

一开始我使用的是sqlite3数据库。

这种数据库虽然不适合十万条以上记录规模的情况

但是由于接口简单、有源码可以集成，

能够快速搭建前期环境，进行项目验证

后面改多进程模式的时候，我又换成了MySQL数据库

因为sqlite3在集群模式下，要做的二次开发实在太多了

我设计了两个类去使用这些数据库

一个是数据库操作类，一个是ORM基类

数据库操作类通过一个接口类统一行为

考虑到不同的数据库初始化的参数类型和需要的参数数量都不尽相同

所以初始化接口使用了json来传递参数

有了接口类，我接下来就依据Sqlite3和MySQL各种的特性

分别实现接口，对接和封装了各种的api函数

然后还设计了一个数据库表类和数据库列类

针对两个数据库各种略有差异的SQL语法（创建表的列类型、关键字、创建条件各有不同）

派生了Sqlite3和MySQL的表类和列类

然后通过宏，将表类派生为各个实际需要的实例表类

TODO:这里需要贴一些代码，过两天我整理一下

### 用户登录是如何实现的？你是如何保证密码安全的？

用户登录过程，一直是软件安全领域当中一个最重要的话题

首先，我们要保证传输的安全性。

无论是http还是https，目前都无法彻底保证这点

所以密码是不能在传输过程中明文出现的

当然，最好是不要直接出现

其次，要有时效性。

传输的包被完整拦截到之后

不能仅仅通过重复发送这个包

服务器或者客户端就能无限验证通过

每个包要有时效性。

最后，登录的数据包最好还要能抵抗暴力破解

最好计算不要太繁琐，消耗CPU资源越少越好

所以我们采用了基于MD5的数据签名算法来实现登录的。

首先，我准备好用户账户名、时间戳、salt（噪声值）、签名混入字符串（服务器有一个一样的字符串，不要太短）和用户输入的密码这些数据

然后，我将这些数据按照特定的顺序组合起来，进行MD5计算，得到一个签名

接着，我将账户名、时间戳、salt和签名发往服务器

服务器收到后，通过账户名找到用户的密码

然后将账户名、时间戳、salt、签名混入字符串和密码在一起进行MD5计算，

得到本地的签名

将客户端发送的签名和本地的签名进行对比【1】

如果一致则表示登录成功

这时会返回一个会话id，该id和签名以及时间戳有关联

客户端拿到会话id后，【2】

后续会将会话id做参数与服务器通信。

这里注意

【1】的时候还会验证时间戳是否超时

如果时间戳和服务器时间戳之差超过若然时间（我设置的是十分钟，600秒）

则服务器拒绝服务。

这是防止有人拿用户的登录包反复登录

更严格的场景下，还应该获取用户的mac值或者ip+端口值

即下次通信的时候，这些值和登录时候的值应该一致

【2】的时候，客户端会验证会话id是否是否超时

如果超时，会判定登录失败！

这是防止有人伪装服务器，向客户端发数据，骗取用户的信息

### 你是如何设计业务模块的？为什么？

服务器一般包括底层通信模块、任务调度和处理模块、业务处理模块

其中业务模块主要是应对项目具体的业务需求

业务需求的特点是易变更、内容繁杂、执行周期不定、可移植性不强

所以业务模块的设计要尽可能简洁、易用、易修改

考虑到简洁和易用，我将客户端通信封装到了客户端类里面

这样网络操作都可以通过客户端类接口来进行

此外业务模块我们仅仅定了几个接口，用于通信模块和业务模块的交互

在客户端连接、收到数据、数据发送成功之后

通信模块可以调用这些接口来通知业务层代码

业务层实现这些接口，并在对应接口进行业务处理

业务层可以在接口直接进行简单的业务处理

也可以将业务转到任务模块处理，避免对通信模块的线程造成挤占

以上就是我对业务模块的构思与设计

### 服务器开发的难点在哪里？你是如何解决的？

我个人觉得，难点有两个方面：

一个是服务器功能繁多，模块关系复杂，很难说有通用的架构来满足所有的需求

每次新的项目，就要依据实际情况进行定制化的设计

即使有相似的需求，也需要依据细节进行调整和适配

这一块我目前觉得只能尽可能在设计上进行优化

比如通信这块，就需要尽可能将通信函数对外屏蔽

为了做到这点，我除了初始化、关闭接口，几乎屏蔽了一切对外接口

但是如果只是这样，通信模块和外部的交互也中断了

所以定义了一个业务接口类

初始化的时候，外部给一个接口对象到通信模块

通信模块通过这个对象提供的业务接口和外部通信

这样可以减少外部对通信模块的的影响

也保证通信模块收到信息后，可以驱动外部模块

另外一个点是如何保证通信的高效性

这块不仅仅是单纯使用epoll就能彻底解决的

还需要配合多线程编程，尽量避免使用互斥锁，梳理业务流程避免阻塞通信处理流程

等等一系列方法才能达到。

这个难就难在工作量大，对开发者知识面要求高，还要求开发者非常有耐心（需要一个环节一个环节进行优化）

### 服务器调试要注意什么事项？

服务器在调试的时候，会面临一些意想不到的麻烦

首先，单纯的gdb调试可能无法达成目的

因为服务器程序往往是多线程多进程并用的

一些问题只可能在子进程里面触发

一些问题只会在多线程发生竞争的时候触发

有时候这些问题并不是必然触发，而是几百上千次才会触发

其次，服务器运行的时候，往往会进入守护进程状态

这意味着没有标准输出

光靠打印语句，也无法达到定位问题和分析问题的目的。

最后，服务器如果是集群状态，那么各个模块可能都不是运行在一台机器上。

可能一些问题在单机上集群运行没有问题

但是换到多机模式运行，就会暴露

这个时候，打印、调试，都会显得比较无力

这些麻烦都需要我们有一个日志服务器来记录问题，汇总日志，协助我们进行分析

所以我在项目开发的时候，对日志模块是花费了相当大的精力的

日志不仅仅会包含源码文件，行号，还包含日志级别、生成时间（精确到毫秒）、所属进程、所属线程、所属函数

此外，对于日志还按级别进行汇总，

低级的日志会包含所有高级的日志内容

一旦服务器程序出现问题，

就可以通过日志，找到问题发生所属的模块、类以及函数

还能看到前后重要的参数值来协助分析产生的原因

所以我觉得，服务器调试，一个强大的日志模块是必不可少的东西，强烈推荐

而且这块花的成本复用性高，可以极大降低后期调试、整合以及维护成本

# 8 HR面试题（软技能）

## 请你自我介绍一下你自己

回答提示：一般人回答这个问题过于平常，只说姓名、年龄、爱好、工作经验，这些在简历上都有，其实，企业最希望知道的是求职者能否胜任工作，包括：最强的技能、最深入研究的知识领域、个性中最积极的部分、做过的最成功的事，主要的成就等，这些都可以和学习无关，也可以和学习有关，但要突出积极的个性和做事的能力，说得合情合理企业才会相信。企业很重视一个人的礼貌，求职者要尊重考官，在回答每个问题之后都说一句“谢谢”。企业喜欢有礼貌的求职者。

## 2、你觉得你个性上最大的优点是什么？

回答提示：沉着冷静、条理清楚、立场坚定、顽强向上。  
乐于助人和关心他人、适应能力和幽默感、乐观和友爱。我在经过一段时间的培训及项目实战，加上实习工作，使我适合这份工作。我相信我能成功。

## **3、说说你最大的缺点？**

回答提示：这个问题企业问的概率很大，通常不希望听到直接回答的缺点是什么等，如果求职者说自己小心眼、爱忌妒人、非常懒、脾气大、工作效率低，企业肯定不会录用你。绝对不要自作聪明地回答“我最大的缺点是过于追求完美”，有的人以为这样回答会显得自己比较出色，但事实上，他已经岌芨可危了。企业喜欢求职者从**自己的优点说起，中间加一些小缺点**，最后再把问题转回到优点上，**突出优点的部分**。企业喜欢聪明的求职者。

1．坦然承认，博得认同 如果是自己的缺点，最好的办法还是坦然地承认它。为自己的缺点辩驳也无济于事，重要的是如何使别人在感情上认同你谈及自身缺点的态度。

2．消除误会，缩小隔阂 有的“缺点”并不是缺点，而是一般意义上的误会造成的，这时你应及时澄清，缩小与面试官的心理隔阂。绝对不要明贬暗褒，谁都不是傻子，这种小聪明耍了就是害自己。说和工作无关的缺点。比如慢热。

## 4、你对加班的看法？

回答提示：实际上好多公司问这个问题，并不证明一定要加班。 只是想测试你是否愿意为公司奉献。  
回答样本：如果是工作需要我会义不容辞加班。我现在单身，没有任何家庭负担，可以全身心的投入工作。但同时，我也会提高工作效率，减少不必要的加班

## 5、你对薪资的要求？

回 答提示：如果你对薪酬的要求太低，那显然贬低自己的能力；如果你对薪酬的要求太高，那又会显得你分量过重，公司受用不起。一些雇主通常都事先对求聘的职位定下开支预算，因而他们第一次提出的价钱往往是他们所能给予的最高价钱。他们问你只不过想证实一下这笔钱是否足以引起你对该工作的兴趣。  
回答样本一：“我对工资没有硬性要求。我相信贵公司在处理我的问题上会友善合理。我注重的是找对工作机会，所以只要条件公平，我则不会计较太多  
回答样本二：我受过系统的软件编程的训练，不需要进行大量的培训。而且我本人也对编程特别感兴趣。因此，我希望公司能根据我的情况和市场标准的水平，给我合理的薪水。  
回答样本三：如果你必须自己说出具体数目，请不要说一个宽泛的范围，那样你将只能得到最低限度的数字。最好给出一个具体的数字，这样表明你已经对当今的人才市场作了调查，知道像自己这样学历的雇员有什么样的价值。

## 6、在五年的时间内，你的职业规划？

回答提示：这是每一个应聘者都不希望被问到的问题，但是几乎每个人都会被问到。比较多的答案是“管理者”。但是近几年来，许多公司都已经建立了专门的技术途径。这些工作地位往往被称作“顾问”、“参议技师”或“高级软件工程师”等等。当然，说出其他一些你感兴趣的职位也是可以的，比如产品销售部经理，生产部经理等一些与你的专业有相关背景的工作。要知道，考官总是喜欢有进取心的应聘者，此时如果说“不知道”，或许就会使你丧失一个好机会。最普通的回答应该是“我准备在技术领域有所作为”或“我希望能按照公司的管理思路发展”。

## 你朋友对你的评价？

回答提示： 想从侧面了解一下你的性格及与人相处的问题。  
回答样本：“我的朋友都说我是一个可以信赖的人。因为，我一旦答应别人的事情，就一定会做到。如果我做不到，我就不会轻易许诺。  
回答样本：”我觉的我是一个比较随和的人，与不同的人都可以友好相处。在我与人相处时，我总是能站在别人的角度考虑问题“

## 你还有什么问题要问吗？

回答提示：企业的这个问题看上去可有可无，其实很关键，企业不喜欢说“没有问题”的人，因为其很注重员工的个性和创新能力。企业不喜欢求职者问个人福利之类的问题，如果有人这样问：贵公司对新入公司的员工有没有什么培训项目，我可以参加吗？或者说贵公司的晋升机制是什么样的？企业将很欢迎，因为体现出你对学习的热情和对公司的忠诚度以及你的上进心。

## 9、如果通过这次面试我们单位录用了你，但工作一段时间却发现你根本不适合这个职位，你怎么办？

回答提示：一段时间发现工作不适合我，有两种情况：  
1、如果你确实热爱这个职业，那你就要不断学习，虚心向领导和同事学习业务知识和处事经验，了解这个职业的精神内涵和职业要求，力争减少差距；  
2、你觉得这个职业可有可无，那还是趁早换个职业，去发现适合你的，你热爱的职业，那样你的发展前途也会大点，对单位和个人都有好处。

## 10、在完成某项工作时，你认为领导要求的方式不是最好的，自己还有更好的方法，你应该怎么做？

回答提示：①.原则上我会尊重和服从领导的工作安排；同时私底下找机会以请教的口吻，婉转地表达自己的想 法，看看领导是否能改变想法；  
②如果领导没有采纳我的建议，我也同样会按领导的要求认真地去完成这项工作；  
③.还有一种情况，假如领导要求的方式违背原则，我会坚决提出反对意见；如领导仍固执己见，我会毫不犹豫地再向上级领导反映。

## 11、如果你的工作出现失误，给本公司造成经济损失，你认为该怎么办？

回答提示：  
①我本意是为公司努力工作，如果造成经济损失，我认为首要的问题是想方设法去弥补或挽回经济损失。如果我无能力负责，希望单位帮助解决；  
②是责任问题。分清责任，各负其责，如果是我的责任，我甘愿受罚；如果是一个我负责的团队中别人的失误，也不能幸灾乐祸，作为一个团队，需要互相提携共同完成工作，安慰同事并且帮助同事查找原因总结经验。  
③总结经验教训，一个人的一生不可能不犯错误，重要的是能从自己的或者是别人的错误中吸取经验教训，并在今后的工作中避免发生同类的错误。检讨自己的工作方法、分析问题的深度和力度是否不够，以致出现了本可以避免的错误。

## 12、如果你在这次考试中没有被录用，你怎么打算？

回答提示：现在的社会是一个竞争的社会,从这次面试中也可看出这一点,有竞争就必然有优劣,有成功必定就会有失败.往往成功的背后有许多的困难和挫折,如果这次失败了也仅仅是一次而已,只有经过经验经历的积累才能塑造出一个完全的成功者。我会从以下几个方面来正确看待这次失败.  
第一、要敢于面对,面对这次失败不气馁,接受已经失去了这次机会就不会回头这个现实,从心理意志和精神上体现出对这次失败的抵抗力。要有自信,相信自己经历了这次之后经过努力一定能行.能够超越自我.  
第二、善于反思,对于这次面试经验要认真总结,思考剖析,能够从自身的角度找差距。正确对待自己,实事求是地评价自己,辩证的看待自己的长短得失,做一个明白人.  
第三、走出阴影,要克服这一次失败带给自己的心理压力,时刻牢记自己弱点,防患于未然,加强学习,提高自身素质.  
第四、认真工作,回到原单位岗位上后,要实实在在、踏踏实实地工作,三十六行,行行出状元,争取在本岗位上做出一定的成绩.  
第五、再接再厉,成为软件工程师或网络工程师一直是我的梦想,以后如果有机会我仍然后再次参加竞争.

## 13、如果你做的一项工作受到上级领导的表扬，但你主管领导却说是他做的，你该怎样？

回答提示：我首先不会找那位上级领导说明这件事，我会主动找我的主管领导来沟通，因为沟通是解决人际关系的最好办法，但结果会有两种：1.我的主管领导认识到自己的错误，我想我会视具体情况决定是否原谅他；2.他更加变本加厉的来威胁我，那我会毫不犹豫地找我的上级领导反映此事，因为他这样做会造成负面影响，对今后的工作不利。

## 14、谈谈你对跳槽的看法？

回答提示：（1）正常的"跳槽"能促进人才合理流动，应该支持；  
（2）频繁的跳槽对单位和个人双方都不利，应该反对。

## 15、工作中你难以和同事、上司相处，你该怎么办？

回答提示：  
①我会服从领导的指挥，配合同事的工作。  
②我会从自身找原因，仔细分析是不是自己工作做得不好让领导不满意，同事看不惯。还要看看是不是为人处世方面做得不好。如果是这样的话 我会努力改正。  
③如果我找不到原因，我会找机会跟他们沟通，请他们指出我的不足。有问题就及时改正。  
④作为优秀的员工，应该时刻以大局为重，即使在一段时间内，领导和同事对我不理解，我也会做好本职工作，虚心向他们学习，我相信，他们会看见我在努力，总有一天会对我微笑的！

## 16、假设你在某单位工作，成绩比较突出，得到领导的肯定。但同时你发现同事们越来越孤立你，你怎么看这个问题？你准备怎么办？

回答提示：  
①成绩比较突出，得到领导的肯定是件好事情，以后更加努力  
②检讨一下自己是不是对工作的热心度超过同事间交往的热心了，加强同事间的交往及共同的兴趣爱好。  
③工作中，切勿伤害别人的自尊心  
④不再领导前拨弄是非  
⑤乐于助人对面

## 17、你最近是否参加了培训课程？谈谈培训课程的内容。是公司资助还是自费参加？

回答提示：可以多谈谈自己学的技术。

## 18、你对于我们公司了解多少？

回答提示：在去公司面试前上网查一下该公司主营业务。如回答：贵公司有意改变策略，加强与国外大厂的OEM合作，自有品牌的部分则透过海外经销商。

## 19、请说出你选择这份工作的动机？

回答提示：这是想知道面试者对这份工作的热忱及理解度，并筛选因一时兴起而来应试的人，如果是无经验者，可以强调“就算职种不同，也希望有机会发挥之前的经验”。

## 20、你最擅长的技术方向是什么？

回答提示：说和你要应聘的职位相关的课程，表现一下自己的热诚没有什么坏处。

## 21、你能为我们公司带来什么呢？

回答提示：  
①假如你可以的话，试着告诉他们你可以减低他们的费用——“我已经接受过惠普公司一段时间的培训，立刻就可以上岗工作”。  
② 企业很想知道未来的员工能为企业做什么，求职者应再次重复自己的优势，然后说：“就我的能力，我可以做一个优秀的员工在组织中发挥能力，给组织带来高效率和更多的收益”。企业喜欢求职者就申请的职位表明自己的能力，比如申请营销之类的职位，可以说：“我可以开发大量的新客户，同时，对老客户做更全面周到的服务，开发老客户的新需求和消费。”等等。

## 22、最能概括你自己的三个词是什么？

回答提示：  
我经常用的三个词是：适应能力强，有责任心和做事有始终，结合具体例子向主考官解释，（易飞云的回答：帅气、幽默、英俊）

## 23、你的业余爱好是什么？

回答提示：找一些富于团体合作精神的，这里有一个真实的故事：有人被否决掉，因为他的爱好是深海潜水。主考官说：因为这是一项单人活动，我不敢肯定他能否适应团体工作。

## 24、作为被面试者给我打一下分

回答提示：试着列出四个优点和一个非常非常非常小的缺点，（可以抱怨一下设施，没有明确责任人的缺点是不会有人介意的）。

## 25、你怎么理解你应聘的职位？

回答提示：把岗位职责和任务及工作态度阐述一下

## 26、喜欢这份工作的哪一点？

回 答提示：相信其实大家心中一定都有答案了吧！每个人的价值观不同，自然评断的标准也会不同，但是，在回答面试官这个问题时可不能太直接就把自己心理的话说出来，尤其是薪资方面的问题，不过一些无伤大雅的回答是不错的考虑，如交通方便，工作性质及内容颇能符合自己的兴趣等等都是不错的答案，不过如果这时自己能仔细思考出这份工作的与众不同之处，相信在面试上会大大加分。

## 27、为什么要离职?

回答提示：  
①回答这个问题时一定要小心，就算在前一个工作受到再大的委屈，对公司有多少的怨言，都千万不要表现出来，尤其要避免对公司本身主管的批评，避免面试官的负面情绪及印象；建议此时最好的回答方式是将问题归咎在自己身上，例如觉得工作没有学习发展的空间，自己想在面试工作的相关产业中多加学习，或是前一份工作与自己的生涯规划不合等等，回答的答案最好是积极正面的。  
②我希望能获得一份更好的工作，如果机会来临，我会抓住；我觉得目前的工作，已经达到顶峰，即沒有升迁机会。

## 28、说说你对行业、技术发展趋势的看法？

回答提示：企业对这个问题很感兴趣，只有有备而来的求职者能够过关。求职者可以直接在网上查找对你所申请的行业部门的信息，只有深入了解才能产生独特的见解。企业认为最聪明的求职者是对所面试的公司预先了解很多，包括公司各个部门，发展情况，在面试回答问题的时候可以提到所了解的情况，企业欢迎进入企业的人是“知己”，而不是“盲人”。

## 29、对工作的期望与目标何在？

回答提示：这是面试者用来评断求职者是否对自己有一定程度的期望、对这份工作是否了解的问题。对于工作有确实学习目标的人通常学习较快，对于新工作自然较容易进入状况，这时建议你，最好针对工作的性质找出一个确实的答案，如业务员的工作可以这样回答：“我的目标是能成为一个超级业务员，将公司的产品广泛的推销出去，达到最好的业绩成效；为了达到这个目标，我一定会努力学习，而我相信以我认真负责的态度，一定可以达到这个目标。”其他类的工作也可以比照这个方式来回答，只要在目标方面稍微修改一下就可以了。

## 30、说说你的家庭。

回答提示：企业面试时询问家庭问题不是非要知道求职者家庭的情况，探究隐私，企业不喜欢探究个人隐私，而是要了解家庭背景对求职者的塑造和影响。企业希望听到的重点也在于家庭对求职者的积极影响。企业最喜欢听到的是：我很爱我的家庭！我的家庭一向很和睦，虽然我的父亲和母亲都是普通人，但是从小，我就看到我父亲起早贪黑，每天工作特别勤劳，他的行动无形中培养了我认真负责的态度和勤劳的精神。我母亲为人善良，对人热情，特别乐于助人，所以在单位人缘很好，她的一言一行也一直在教导我做人的道理。企业相信，和睦的家庭关系对一个人的成长有潜移默化的影响。

## 31、就你申请的这个职位，你认为你还欠缺什么？

回答提示：企业喜欢问求职者弱点，但精明的求职者一般不直接回答。他们希望看到这样的求职者：继续重复自己的优势，然后说：“对于这个职位和我的能力来说，我相信自己是可以胜任的，只是缺乏经验，这个问题我想我可以进入公司以后以最短的时间来解决，我的学习能力很强，我相信可以很快融入公司的企业文化，进入工作状态。”企业喜欢能够巧妙地躲过难题的求职者。

## 32、你欣赏哪种性格的人？

回答提示：诚实、不死板而且容易相处的人、有"实际行动"的人。

## 33、你通常如何处理別人的批评？

回答提示：①沈默是金。不必说什么，否则情况更糟，不过我会接受建设性的批评；②我会等大家冷靜下来再讨论。

## 34、你怎样对待自己的失敗？

回答提示：我们大家生来都不是十全十美的，我相信我有第二个机会改正我的错误。

## 35、什么会让你有成就感？

回答提示：为贵公司竭力效劳；尽我所能，完成一个项目

## 36、眼下你生活中最重要的是什么？

回答提示：对我来说，能在这个领域找到工作是最重要的；望能在贵公司任职对我说最重要。

## 37、你为什么愿意到我们公司来工作？

回答提示：对于这个问题，你要格外小心，如果你已经对该单位作了研究，你可以回答一些详细的原因，像“公司本身的高技术开发环境很吸引我。”，“我同公司出生在同样的时代，我希望能够进入一家与我共同成长的公司。”“你们公司一直都稳定发展，在近几年来在市场上很有竞争力。”或者“我认为贵公司能够给我提供一个与众不同的发展道路。”这都显示出你已经做了一些调查，也说明你对自己的未来有了较为具体的远景规划。

## 38、你和别人发生过争执吗？你是怎样解决的？

回答提示：这是面试中最险恶的问题。其实是考官布下的一个陷阱。千万不要说任何人的过错。应知成功解决矛盾是一个协作团体中成员所必备的能力。假如你工作在一个服务行业，这个问题简直成了最重要的一个环节。你是否能获得这份工作，将取决于这个问题的回答。考官希望看到你是成熟且乐于奉献的。他们通过这个问题了解你的成熟度和处世能力。在没有外界干涉的情况下，通过妥协的方式来解决才是正确答案。

## 39、问题：你做过的哪件事最令自己感到骄傲?

回答提示：这是考官给你的一个机会，让你展示自己把握命运的能力。这会体现你潜在的领导能力以及你被提升的可能性。假如你应聘于一个服务性质的单位，你很可能会被邀请去午餐。记住：你的前途取决于你的知识、你的社交能力和综合表现。

## 40、你新到一个部门,一天一个客户来找你解决问题,你努力想让他满意，可是始终达不到群众得满意,他投诉你们部门工作效率低,你这个时候怎么作?

回 答提示：(1)首先，我会保持冷静。作为一名工作人员，在工作中遇到各种各样的问题是正常的，关键是如何认识它，积极应对，妥善处理。 (2)其次，我会反思一下客户不满意的原因。一是看是否是自己在解决问题上的确有考虑的不周到的地方，二是看是否是客户不太了解相关的服务规定而提出超出规定的要求，三是看是否是客户了解相关的规定，但是提出的要求不合理。 (3)再次，根据原因采取相对的对策。如果是自己确有不周到的地方，按照服务规定作出合理的安排，并向客户作出解释；如果是客户不太了解政策规定而造成的误解，我会向他作出进一步的解释，消除他的误会；如果是客户提出的要求不符合政策规定，我会明确地向他指出。 (4)再次，我会把整个事情的处理情况向领导作出说明，希望得到他的理解和支持。(5)我不会因为客户投诉了我而丧失工作的热情和积极性，而会一如既往地牢记为客户服务的宗旨，争取早日做一名领导信任、公司放心、客户满意的职员。

## 41、对这项工作，你有哪些可预见的困难？”

回答提示：：①不宜直接说出具体的困难，否则可能令对方怀疑应聘者不行；②可以尝试迂回战术，说出应聘者对困难所持有的态度——“工作中出现一些困难是正常的，也是难免的，但是只要有坚忍不拔的毅力、良好的合作精神以及事前周密而充分的准备，任何困难都是可以克服。”  
分析：一般问这个问题，面试者的希望就比较大了，因为已经在谈工作细节。但常规思路中的回答，又被面试官“骗”了。当面试官询问这个问题的时候，有两个目的。第一，看看应聘者是不是在行，说出的困难是不是在这个职位中一般都不可避免的问题。第二，是想看一下应聘者解决困难的手法对不对，及公司能否提供这样的资源。而不是想了解应聘者对困难的态度。

## 42、如果我录用你，你将怎样开展工作？”

回答提示： ①如果应聘者对于应聘的职位缺乏足够的了解，最好不要直接说出自己开展工作的具体办法；②可以尝试采用迂回战术来回答，如“首先听取领导的指示和要求，然后就有关情况进行了解和熟悉，接下来制定一份近期的工作计划并报领导批准，最后根据计划开展工作。”  
分析：这个问题的主要目的也是了解应聘者的工作能力和计划性、条理性，而且重点想要知道细节。如果向思路中所讲的迂回战术，面试官会认为回避问题，如果引导了几次仍然是回避的话。此人绝对不会录用了。

## 43、“你希望与什么样的上级共事？”

回答提示：①通过应聘者对上级的“希望”可以判断出应聘者对自我要求的意识，这既上一个陷阱，又是一次机会；②最好回避对上级具体的希望，多谈对自己的要求；③如“做为刚步入社会的新人，我应该多要求自己尽快熟悉环境、适应环境，而不应该对环境提出什么要求，只要能发挥我的专长就可以了  
分析：这个问题比较好的回答是，希望我的上级能够在工作中对我多指导，对我工作中的错误能够立即指出。总之，从上级指导这个方面谈，不会有大的纰漏。

## 44、在完成某项工作时，你认为领导要求的方式不是最好的，自己还有更好的方法，你应该怎么做？

回答提示：  
①.原则上我会尊重和服从领导的工作安排；同时私底下找机会以请教的口吻，婉转地表达自己的想法，看看领导是否能改变想法；  
②如果领导没有采纳我的建议，我也同样会按领导的要求认真地去完成这项工作；  
③.还有一种情况，假如领导要求的方式违背原则，我会坚决提出反对意见；如领导仍固执己见，我会毫不犹豫地再向上级领导反映。

## 45、与上级意见不一是，你将怎么办？”

回答提示：①一般可以这样回答“我会给上级以必要的解释和提醒，在这种情况下，我会服从上级的意见。”②如果面试你的是总经理，而你所应聘的职位另有一位经理，且这位经理当时不在场，可以这样回答：“对于非原则性问题，我会服从上级的意见，对于涉及公司利益的重大问题，我希望能向更高层领导反映。”  
分析：这个问题的标准答案是思路1，如果用2的回答，必死无疑。你没有摸清楚改公司的内部情况，先想打小报告，这样的人没有人敢要。

## 46、“你工作经验欠缺，如何能胜任这项工作？”

常规思路：①如果招聘单位对应届毕业生的应聘者提出这个问题，说明招聘公司并不真正在乎“经验”，关键看应聘者怎样回答；②对这个问题的回答最好要体现出应聘者的诚恳、机智、果敢及敬业；③如“作为应届毕业生，在工作经验方面的确会有所欠缺，因此在读书期间我一直利用各种机会在这个行业里做兼职。我也发现，实际工作远比书本知识丰富、复杂。但我有较强的责任心、适应能力和学习能力，而且比较勤奋，所以在兼职中均能圆满完成各项工作，从中获取的经验也令我受益非浅。请贵公司放心，学校所学及兼职的工作经验使我一定能胜任这个职位。” 点评：这个问题思路中的答案尚可。突出自己的吃苦能力和适应性以及学习能力（不是学习成绩）为好。

## 47、您在前一家公司的离职原因是什么？”

回答提示：①最重要的是：应聘者要使找招聘单位相信，应聘者在过往的单位的“离职原因”在此家招聘单位里不存在；②避免把“离职原因”说得太详细、太具体；③不能掺杂主观的负面感受，如“太辛苦”、“人际关系复杂”、“管理太混乱”、“公司不重视人才”、“公司排斥我们某某的员工”等；④但也不能躲闪、回避，如“想换换环境”、“个人原因”等；⑤不能涉及自己负面的人格特征，如不诚实、懒惰、缺乏责任感、不随和等；⑥尽量使解释的理由为应聘者个人形象添彩；⑦相关例子：如“我离职是因为这家公司倒闭；我在公司工作了三年多，有较深的感情；从去年始，由于市场形势突变，公司的局面急转直下；到眼下这一步我觉得很遗憾，但还要面对显示，重新寻找能发挥我能力的舞台。”同一个面试问题并非只有一个答案，而同一个答案并不是在任何面试场合都有效，关键在应聘者掌握了规律后，对面试的具体情况进行把握，有意识地揣摩面试官提出问题的心理背景，然后投其所好。  
分析：除非是薪资太低，或者是最初的工作，否则不要用薪资作为理由。“求发展”也被考官听得太多，离职理由要根据每个人的真实离职理由来设计，但是在回答时一定要表现得真诚。实在想不出来的时候，家在外地可以说是因为家中有事，须请假几个月，公司又不可能准假，所以辞职。这个答案一般面试官还能接受。

## 48、“你工作经验欠缺，如何能胜任这项工作？”

回答提示：①如果招聘单位对应届毕业生的应聘者提出这个问题，说明招聘公司并不真正在乎“经验”，关键看应聘者怎样回答；②对这个问题的回答最好要体现出应聘者的诚恳、机智、果敢及敬业；③如“作为应届毕业生，在工作经验方面的确会有所欠缺，因此在读书期间我一直利用各种机会在这个行业里做兼职。我也发现，实际工作远比书本知识丰富、复杂。但我有较强的责任心、适应能力和学习能力，而且比较勤奋，所以在兼职中均能圆满完成各项工作，从中获取的经验也令我受益非浅。请贵公司放心，学校所学及兼职的工作经验使我一定能胜任这个职位。”  
分析：这个问题思路中的答案尚可。突出自己的吃苦能力和适应性以及学习能力（不是学习成绩）为好。

## 49、为了做好你工作份外之事，你该怎样获得他人的支持和帮助？

回答提示：每个公司都在不断变化发展的过程中；你当然希望你的员工也是这样。你希望得到那些希望并欢迎变化的人，因为这些人明白，为了公司的发展，变化是公司日常生活中重要组成部分。这样的员工往往很容易适应公司的变化，并会对变化做出积极的响应。此外，他们遇到矛盾和问题时，也能泰然处之。下面的问题能够考核应聘者这方面的能力。  
据说有人能从容避免正面冲突。请讲一下你在这方面的经验和技巧。  
有些时候，我们得和我们不喜欢的人在一起共事。说说你曾经克服了性格方面的冲突而取得预期工作效果的经历。

## 50、如果你在这次面试中没有被录用，你怎么打算？

回答提示：现在的社会是一个竞争的社会,从这次面试中也可看出这一点,有竞争就必然有优劣,有成功必定就会有失败.往往成功的背后有许多的困难和挫折,如果这次失败了也仅仅是一次而已,只有经过经验经历的积累才能塑造出一个完全的成功者。我会从以下几个方面来正确看待这次失败.  
第一、要敢于面对,面对这次失败不气馁,接受已经失去了这次机会就不会回头这个现实,从心理意志和精神上体现出对这次失败的抵抗力。要有自信,相信自己经历了这次之后经过努力一定能行.能够超越自我.  
第二、善于反思,对于这次面试经验要认真总结,思考剖析,能够从自身的角度找差距。正确对待自己,实事求是地评价自己,辩证的看待自己的长短得失,做一个明白人.  
第三、走出阴影,要克服这一次失败带给自己的心理压力,时刻牢记自己弱点,防患于未然,加强学习,提高自身素质.  
第四、认真工作,回到原单位岗位上后,要实实在在、踏踏实实地工作,三十六行,行行出状元,争取在本岗位上做出一定的成绩.  
第五、再接再厉,成为国家公务员一直是我的梦想,以后如果有机会我仍然后再次参加竞争.

## 51、假如你晚上要去送一个出国的同学去机场，可单位临时有事非你办不可，你怎么办？

回答提示：我觉得工作是第一位的，但朋友间的情谊也是不能偏废的。这个问题我觉得要按照当时具体的情况来决定。  
（1）、如果我的朋友晚上9点中的飞机，而我的 加班八点就能够完成的话，那就最理想了，干完工作去机场，皆大欢喜。  
（2）、如果说工作不是很紧急，加班仅仅是为了明天上班的时候能把报告交到办公室，那完全可以跟领导打声招呼，先去机场然后回来加班，晚点睡就是了。  
（3）、如果工作很紧急，两者不可能兼顾的情况下，我觉得可以由两种选择。1）如果不是全单位都加班的话，是不是可以要其他同事来代替以下工作，自己去机场，哪怕就是代替你离开的那一会儿。2）如果连这一点都做不到的话，  
那只好忠义不能两全了，打电话给朋友解释一下，小心他会理解，毕竟工作做完了就完了，朋友还是可以再见面的。

## 52、如果通过这次面试我们单位录用了你，但工作一段时间却发现你根本不适合这个职位，你怎么办？

回答提示：一段时间发现工作不适合我，有两种情况：  
1、如果你确实热爱这个职业，那你就要不断学习，虚心向领导和同事学习业务知识和处事经验，了解这个职业的精神内涵和职业要求，力争减少差距；  
2、你觉得这个职业可有可无，那还是趁早换个职业，去发现适合你的，你热爱的职业，那样你的发展前途也会大点，对单位和个人都有好处。

## 53、你做过的哪件事最令自己感到骄傲?

回答提示：这是考官给你的一个机会，让你展示自己把握命运的能力。这会体现你潜在的领导能力以及你被提升的可能性。假如你应聘于一个服务性质的单位，你很可能会被邀请去午餐。记住：你的前途取决于你的知识、你的社交能力和综合表现。

## 54、谈谈你过去做过的成功案例

回答提示：举一个你最有把握的例子，把来龙去脉说清楚，而不要说了很多却没有重点。切忌夸大其词，把别人的功劳到说成自己的，很多主管为了确保要用的人是最适合的，会打电话向你的前一个主管征询对你的看法及意见，所以如果说谎，是很容易穿梆的。

## 55、谈谈你过去的工作经验中，最令你挫折的事情

回答提示：曾经接触过一个客户，原本就有耳闻他们以挑剔出名，所以事前的准备功夫做得十分充分，也投入了相当多的时间与精力，最后客户虽然并没有照单全收，但是接受的程度已经出乎我们意料之外了。原以为从此可以合作愉快，却得知客户最后因为预算关系选择了另一家代理商，之前的努力因而付诸流水。尽管如此，我还是从这次的经验学到很多，如对该产业的了解，整个team的默契也更好了。  
分析：借此了解你对挫折的容忍度及调解方式。

## 56、如何安排自己的时间？会不会排斥加班？

回答提示：基本上，如果上班工作有效率，工作量合理的话，应该不太需要加班。可是我也知道有时候很难避免加班，加上现在工作都采用责任制，所以我会调配自己的时间，全力配合。  
分析：虽然不会有人心甘情愿的加班，但依旧要表现出高配合度的诚意。

## 57、为什么我们要在众多的面试者中选择你？

回答提示：根据我对贵公司的了解，以及我在这份工作上所累积的专业、经验及人脉，相信正是贵公司所找寻的人才。而我在工作态度、ＥＱ上，也有圆融、成熟的一面，和主管、同事都能合作愉快。  
分析：别过度吹嘘自己的能力，或信口开河地乱开支票，例如一定会为该公司带来多少钱的业务等，这样很容易给人一种爱说大话、不切实际的感觉。

## 58、对这个职务的期许？

回答提示：希望能借此发挥我的所学及专长，同时也吸收贵公司在这方面的经验，就公司、我个人而言，缔造“双赢”的局面。  
分析：回答前不妨先询问该公司对这项职务的责任认定及归属，因为每一家公司的状况不尽相同。以免说了一堆理想抱负却发现牛头不对马嘴。

## 59、为什么选择这个职务？

回答提示：：这一直是我的兴趣和专长，经过这几年的磨练，也累积了一定的经验及人脉，相信我一定能胜任这个职务的。  
分析：适时举出过去的“丰功伟业”，表现出你对这份职务的熟稔度，但避免过于夸张的形容或流于炫耀。

## 60、为什么选择我们这家公司？

回答提示：曾经在报章杂志看过关于贵公司的报道，与自己所追求的理念有志一同。而贵公司在业界的成绩也是有目共睹的，而且对员工的教育训练、升迁等也都很有制度。  
分析：去面试前先做功课，了解一下该公司的背景，让对方觉得你真的很有心想得到这份工作，而不只是探探路。

## 61、你认为你在学校属于好学生吗？

回答提示：企业的招聘者很精明，问这个问题可以试探出很多问题：如果求职者学习成绩好，就会说：“是的，我的成绩很好，所有的成绩都很优异。当然，判断一个学生是不是好学生有很多标准，在学校期间我认为成绩是重要的，其他方面包括思想道德、实践经验、团队精神、沟通能力也都是很重要的，我在这些方面也做得很好，应该说我是一个全面发展的学生。”如果求职者成绩不尽理想，便会说：“我认为是不是一个好学生的标准是多元化的，我的学习成绩还可以，在其他方面我的表现也很突出，比如我去很多地方实习过，我很喜欢在快节奏和压力下工作，我在学生会组织过 ××活动，锻炼了我的团队合作精神和组织能力。” 有经验的招聘者一听就会明白，企业喜欢诚实的求职者。

## 62、请谈谈如何适应办公室工作的新环境？

回答提示①办公室里每个人有各自的岗位与职责，不得擅离岗位。  
②根据领导指示和工作安排，制定工作计划，提前预备，并按计划完成。  
③多请示并及时汇报，遇到不明白的要虚心请教。  
④抓间隙时间，多学习，努力提高自己的政治素质和业务水平。

## 63、在工作中学习到了些什么？

回答提示：这是针对转职者提出的问题，建议此时可以配合面试工作的特点作为主要依据来回答，如业务工作需要与人沟通，便可举出之前工作与人沟通的例子，经历了哪些困难，学习到哪些经验，把握这些要点做陈述，就可以轻易过关了

## 64、有想过创业吗？

回答提示：这个问题可以显示你的冲劲，但如果你的回答是“有”的话，千万小心，下一个问题可能就是“那么为什么你不这样做呢？”

## 65、最能概括你自己的三个词是什么？

回答提示：我经常用的三个词是：适应能力强，有责任心和做事有始终，结合具体例子向主考官解释，使他们觉得你具有发展潜力

## 66、你认为你在学校属于好学生吗？

回答提示：企业的招聘者很精明，问这个问题可以试探出很多问题：如果求职者学习成绩好，就会说：“是的，我的成绩很好，所有的成绩都很优异。当然，判断一个学生是不是好学生有很多标准，在学校期间我认为成绩是重要的，其他方面包括思想道德、实践经验、团队精神、沟通能力也都是很重要的，我在这些方面也做得很好，应该说我是一个全面发展的学生。”如果求职者成绩不尽理想，便会说：“我认为是不是一个好学生的标准是多元化的，我的学习成绩还可以，在其他方面我的表现也很突出，比如我去很多地方实习过，我很喜欢在快节奏和压力下工作，我在学生会组织过 ××活动，锻炼了我的团队合作精神和组织能力。” 有经验的招聘者一听就会明白，企业喜欢诚实的求职者。

## 67、除了本公司外，还应聘了哪些公司？

回答提示：很奇怪，这是相当多公司会问的问题，其用意是要概略知道应徵者的求职志向，所以这并非绝对是负面答案，就算不便说出公司名称，也应回答“销售同种产品的公司”，如果应聘的其他公司是不同业界，容易让人产生无法信任的感觉。

## 68、何时可以到职？

回答提示：大多数企业会关心就职时间，最好是回答\’如果被录用的话，到职日可按公司规定上班”，但如果还未辞去上一个工作、上班时间又太近，似乎有些强人所难，因为交接至少要一个月的时间，应进一步说明原因，录取公司应该会通融的

## 69、你并非毕业于名牌院校？

回答提示：是否毕业于名牌院校不重要，重要的是有能力完成您交给我的工作，我接受了易道云教育公司软件培训，掌握的技能完全可以胜任贵公司现在工作，而且我比一些名牌院校的毕业生的动手能力还要强，我想我更适合贵公司这个职位。

## 70、你怎样看待学历和能力？

回答提示：学历我想只要是本科的学历，就表明觉得我具备了根本的学习能力。剩下的，你是学士也好，还是博士也好，对于这一点的讨论，不是看你学了多少知识，而是看你在这个领域上发挥了什么，也就是所说的能力问题。一个人工作能力的高低直接决定其职场命运，而学历的高低只是进入一个企业的敲门砖，如果贵公司把学历卡在博士上，我就无法进入贵公司，当然这不一定只是我个人的损失，如果一个本科生都能完成的工作，您又何必非要招聘一位博士生呢？