

《InterviewGuide》第三版

它还有一个名字叫《逆袭进大厂》

更新记录

1、看这里，十分重要！！！

2、面试准备

 2.1、简历篇

 2.1.1、求求你了，不要乱写简历了

 2.1.2、阿秀个人26版秋招简历迭代修改过程

 2.1.3、多用一些简历网站

 2.1.4、投递简历的三种方式

 2.2、了解校招

 2.2.1、校招重要时间点

 2.2.2、确定工作方向

 2.3、C++能投哪些岗位

3、知识储备

 3.1、C++

 1、在main执行之前和之后执行的代码可能是什么？

 2、结构体内存对齐问题？

 3、指针和引用的区别

 4、堆和栈的区别

 5、区别以下指针类型？

 6、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

 7、new / delete 与 malloc / free的异同

 8、new和delete是如何实现的？

 9、malloc和new的区别？

 9.1、delete和delete[]区别？(补充)

 10、宏定义和函数有何区别？

 11、宏定义和typedef区别？

 12、变量声明和定义区别？

 13、哪几种情况必须用到初始化成员列表？

 14、strlen和sizeof区别？

 15、常量指针和指针常量区别？

 16、a和&a有什么区别？

 17、数组名和指针（这里为指向数组首元素的指针）区别？

 18、野指针和悬空指针

 19、迭代器失效的情况

 20、C和C++的区别

 21、C++与Java的区别

 22、C++中struct和class的区别

 23、define宏定义和const的区别

 24、C++中const和static的作用

 25、C++的顶层const和底层const

 26、类的对象存储空间？

 27、final和override关键字

 28、拷贝初始化和直接初始化

 29、初始化和赋值的区别

 30、extern"C"的用法

 31、模板函数和模板类的特例化

 32、C和C++的类型安全

 33、为什么析构函数一般写成虚函数

 34、构造函数能否声明为虚函数或者纯虚函数，析构函数呢？

 35、C++中的重载、重写（覆盖）和隐藏的区别

- 36、C++的多态如何实现
- 37、C++有哪几种的构造函数
- 38、浅拷贝和深拷贝的区别
- 39、内联函数和宏定义的区别
- 40、构造函数、析构函数、虚函数可否声明为内联函数
- 41、auto、decltype和decltype(auto)的用法
- 42、public, protected和private访问和继承权限/public/protected/private的区别?
- 43、如何用代码判断大小端存储
- 44、volatile、mutable和explicit关键字的用法
- 45、什么情况下会调用拷贝构造函数
- 46、C++中有几种类型的new
- 47、C++中NULL和nullptr区别
- 48、简要说明C++的内存分区
- 49、C++的异常处理的方法
- 50、static的用法和作用?
- 51、静态变量什么时候初始化
- 52、const关键字?
- 53、指针和const的用法
- 54、形参与实参的区别?
- 55、值传递、指针传递、引用传递的区别和效率
- 56、什么是类的继承?
- 57、什么是内存池，如何实现
- 58、从汇编层去解释一下引用
- 59、深拷贝与浅拷贝是怎么回事?
- 60、C++模板是什么，你知道底层怎么实现的?
- 61、new和malloc的区别?
- 62、delete p、delete [] p、allocator都有什么作用?
- 63、new和delete的实现原理，delete是如何知道释放内存的大小的额?
- 64、malloc申请的存储空间能用delete释放吗
- 65、malloc与free的实现原理?
- 66、malloc、realloc、calloc的区别
- 67、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?
- 68、成员列表初始化?
- 69、什么是内存泄露，如何检测与避免
- 70、对象复用的了解，零拷贝的了解
- 71、解释一下什么是trivial destructor
- 72、介绍面向对象的三大特性，并且举例说明
- 73、C++中类的数据成员和成员函数内存分布情况
- 74、成员初始化列表的概念，为什么用它会快一些?
- 75、(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?
- 76、析构函数的作用，如何起作用?
- 77、构造函数和析构函数可以调用虚函数吗，为什么
- 78、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?
- 79、虚析构函数的作用，父类的析构函数是否要设置为虚函数?
- 80、构造函数析构函数可否抛出异常
- 81、构造函数一般不定义为虚函数的原因
- 82、类什么时候会析构?
- 83、构造函数或者析构函数中可以调用虚函数吗
- 84、智能指针的原理、常用的智能指针及实现
- 85、构造函数的几种关键字
- 86、C++的四种强制转换reinterpret_cast/const_cast/static_cast /dynamic_cast
- 87、C++函数调用的压栈过程
- 88、说说移动构造函数
- 89、C++中将临时变量作为返回值时的处理过程
- 90、关于this指针你知道什么? 全说出来
- 91、几个this指针的易混问题
 - A. this指针是什么时候创建的?
 - B. this指针存放在何处? 堆、栈、全局变量，还是其他?

- C. 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”？
- D. this指针是如何访问类中的变量的？
- E. 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？
- F. 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？
- 92、构造函数、拷贝构造函数和赋值操作符的区别
- 93、拷贝构造函数和赋值运算符重载的区别？
- 94、智能指针的作用；
- 95、说说你了解的auto_ptr作用
- 96、智能指针的循环引用
- 97、什么是虚拟继承
- 98、如何获得结构成员相对于结构开头的字节偏移量
- 99、静态类型和动态类型，静态绑定和动态绑定的介绍
- 100、C++ 11有哪些新特性？
- 101、引用是否能实现动态绑定，为什么可以实现？
- 102、全局变量和局部变量有什么区别？
- 103、指针加减计算要注意什么？
- 104、怎样判断两个浮点数是否相等？
- 105、方法调用的原理（栈，汇编）
- 106、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？
- 107、类如何实现只能静态分配和只能动态分配
- 108、如果想将某个类用作基类，为什么该类必须定义而非声明？
- 109、什么情况会自动生成默认构造函数？
- 110、抽象基类为什么不能创建对象？
- 111、继承机制中对象之间如何转换？指针和引用之间如何转换？
- 112、知道C++中的组合吗？它与继承相比有什么优缺点吗？
- 113、函数指针？
- 114、内存泄漏的后果？如何监测？解决方法？
- 115、使用智能指针管理内存资源，RAII是怎么回事？
- 116、手写实现智能指针类
- 117、说一说你理解的内存对齐以及原因
- 118、结构体变量比较是否相等
- 119、函数调用过程栈的变化，返回值和参数变量哪个先入栈？
- 120、define、const、typedef、inline的使用方法？他们之间有什么区别？
- 121、你知道printf函数的实现原理是什么吗？
- 122、说一说你了解的关于lambda函数的全部知识
- 123、将字符串“hello world”从开始到打印到屏幕上的全过程？
- 124、模板类和模板函数的区别是什么？
- 125、为什么模板类一般都是放在一个h文件中
- 126、C++中类成员的访问权限和继承权限问题
- 127、cout和printf有什么区别？
- 128、你知道重载运算符吗？
- 129、当程序中有函数重载时，函数的匹配原则和顺序是什么？
- 130、定义和声明的区别
- 131、全局变量和static变量的区别
- 132、静态成员与普通成员的区别是什么？
- 133、说一下你理解的 ifdef endif 代表着什么？
- 134、隐式转换，如何消除隐式转换？
- 135、虚函数的内存结构，那菱形继承的虚函数内存结构呢
- 136、多继承的优缺点，作为一个开发者怎么看待多继承
- 137、迭代器：++it、it++哪个好，为什么
- 138、C++如何处理多个异常的？
- 139、模板和实现可不可以不写在一个文件里面？为什么？
- 140、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？
- 141、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
- 142、你知道strcpy和memcpy的区别是什么吗？
- 143、程序在执行int main(int argc, char *argv[])时的内存结构，你了解吗？

- 144、`volatile`关键字的作用?
- 145、如果有一个空类，它会默认添加哪些函数?
- 146、C++中标准库是什么?
- 147、你知道`const char*`与`string`之间的关系是什么吗?
- 148、为什么拷贝构造函数必须传引用不能传值?
- 149、你知道空类的大小是多少吗?
- 150、你什么情况用指针当参数，什么时候用引用，为什么?
- 151、静态函数能定义为虚函数吗? 常函数呢? 说说你的理解
- 152、`this`指针调用成员变量时，堆栈会发生什么变化?
- 153、你知道静态绑定和动态绑定吗? 讲讲?
- 154、如何设计一个类计算子类的个数?
- 155、怎么快速定位错误出现的地方
- 156、虚函数的代价?
- 157、类对象的大小受哪些因素影响?
- 158、移动构造函数听说过吗? 说说
- 159、什么时候合成构造函数? 都说一说，你知道的都说一下
- 160、那什么时候需要合成拷贝构造函数呢?
- 161、成员初始化列表会在什么时候用到? 它的调用过程是什么?
- 162、构造函数的执行顺序是什么?
- 163、一个类中的全部构造函数的扩展过程是什么?
- 164、哪些函数不能是虚函数? 把你知道的都说一说
- 165、说一说`strcpy`、`sprintf`与`memcpy`这三个函数的不同之处
- 166、将引用作为函数参数有哪些好处?
- 167、你知道数组和指针的区别吗?
- 168、如何阻止一个类被实例化? 有哪些方法?
- 169、如何禁止程序自动生成拷贝构造函数?
- 170、你知道`Debug`和`Release`的区别是什么吗?
- 171、`main`函数的返回值有什么值得考究之处吗?
- 172、模板会写吗? 写一个比较大小的模板函数
- 173、智能指针出现循环引用怎么解决?
- 174、`strcpy`函数和`strncpy`函数的区别? 哪个函数更安全?
- 175、`static_cast`比C语言中的转换强在哪里?
- 176、成员函数里`memset(this, 0, sizeof(*this))`会发生什么
- 177、你知道回调函数吗? 它的作用?
- 178、什么是一致性哈希?
- 179、什么是纯虚函数，与虚函数的区别
- 180、C++从代码到可执行程序经历了什么?
 - (1) 预编译
 - (2) 编译
 - (3) 汇编
 - (4) 链接

静态链接
动态链接
- 181、为什么友元函数必须在类内部声明?
- 182、用C语言实现C++的继承
- 183、动态编译与静态编译
- 184、`hello.c`程序的编译过程
 - 静态链接
 - 目标文件
 - 动态链接
- 185、介绍一下几种典型的锁
- 186、说一下C++左值引用和右值引用
- 187、STL中`hashtable`的实现?
- 188、简单说一下STL中的traits技法
- 189、STL的两级空间配置器
 - 一级配置器
 - 二级配置器
 - 一级分配器

二级分配器

- 190、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
- 191、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？
- 192、容器内部删除一个元素
- 193、STL迭代器如何实现
- 194、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？
- 195、如何在共享内存上使用stl标准库？
- 196、map插入方式有几种？
- 197、STL中unordered_map(hash_map)和map的区别，hash_map如何解决冲突以及扩容
- 198、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？
- 199、map中[]与find的区别？
- 200、STL中list与queue之间的区别
- 201、STL中的allocator,deallocator
- 202、STL中hash_map扩容发生什么？
- 203、常见容器性质总结？
- 204、vector的增加删除都是怎么做的？为什么是1.5或者是2倍？
- 205、说一下STL每种容器对应的迭代器
- 206、STL中vector的实现
- 207、STL中list的实现
- 208、STL中queue的实现
- 209、STL中的deque的实现
- 210、STL中stack和queue的实现
- 211、STL中的heap的实现
- 212、STL中的priority_queue的实现
- 213、STL中set的实现？
- 214、STL中map的实现
- 215、set和map的区别，multimap和multiset的区别
- 216、STL中unordered_map和map的区别和应用场景
- 217、hashtable中解决冲突有哪些方法？

3.2、数据结构与算法

- 1、合并有序链表
- 2、反转链表
- 3、单例模式
- 4、简单工厂模式
- 5、快排排序
- 6、归并排序
- 7、设计LRU缓存
- 8、重排链表
- 9、奇偶链表

3.3、操作系统

- 1、进程、线程和协程的区别和联系
- 2、线程与进程的比较
- 3、一个进程可以创建多少线程，和什么有关？
- 4、外中断和异常有什么区别？
- 5、进程线程模型你知道多少？
 - 多线程
 - 多进程
- 6、进程调度算法你了解多少？
- 7、Linux下进程间通信方式？
- 8、Linux下同步机制？
- 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？
- 10、内存交换和覆盖有什么区别？
- 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
 - 2、最佳适应算法
 - 3、最坏适应算法
 - 4、邻近适应算法
 - 5、总结

- 12、虚拟技术你了解吗？
- 13、进程状态的切换你知道多少？
- 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？
- 15、通过例子讲解逻辑地址转换为物理地址的基本过程
- 16、进程同步的四种方法？
 1. 临界区
 2. 同步与互斥
 3. 信号量
 4. 管程
- 17、操作系统在对内存进行管理的时候需要做些什么？
- 18、进程通信方法（Linux和windows下），线程通信方法（Linux和windows下）
- 19、程间通信有哪几种方式？把你知道的都说出来
管道
消息队列
共享内存
信号量
辅助命令
套接字
- 20、虚拟内存的目的是什么？
- 21、说一下你理解中的内存？他有什么作用呢？
- 22、操作系统经典问题之哲学家进餐问题
- 23、操作系统经典问题之读者-写者问题
- 24、介绍一下几种典型的锁
读写锁
互斥锁
条件变量
自旋锁
- 25、逻辑地址VS物理地址
- 26、怎么回收线程？有哪几种方法？
- 27、内存的覆盖是什么？有什么特点？
- 28、内存交换是什么？有什么特点？
- 29、什么时候会进行内存的交换？
- 30、终端退出，终端运行的进程会怎样
- 31、如何让进程后台运行
- 32、什么是快表，你知道多少关于快表的知识？
- 33、地址变换中，有快表和没快表，有什么区别？
- 35、守护进程、僵尸进程和孤儿进程
守护进程
孤儿进程
僵尸进程
- 36、如何避免僵尸进程？
- 37、局部性原理你知道吗？主要有哪两大局部性原理？各自是什么？
- 38、父进程、子进程、进程组、作业和会话
父进程
子进程
进程组
作业
会话
- 39、进程终止的几种方式
- 40、Linux中异常和中断的区别
- 41、Windows和Linux环境下内存分布情况
- 42、一个由C/C++编译的程序占用的内存分为哪几个部分？
- 43、一般情况下在Linux/windows平台下栈空间的大小
- 44、程序从堆中动态分配内存时，虚拟内存上怎么操作的
- 45、常见的几种磁盘调度算法
 1. 先来先服务
 2. 最短寻道时间优先
 3. 电梯扫描算法

- 46、交换空间与虚拟内存的关系
47、抖动你知道是什么吗？它也叫颠簸现象
48、从堆和栈上建立对象哪个快？（考察堆和栈的分配效率比较）
49、常见内存分配方式有哪些？
50、常见内存分配内存错误
51、内存交换中，被换出的进程保存在哪里？
52、在发生内存交换时，有些进程是被优先考虑的？你可以说一说吗？
53、ASCII、Unicode和UTF-8编码的区别？
54、原子操作的是如何实现的
55、内存交换你知道有哪些需要注意的关键点吗？
56、系统并发和并行，分得清吗？
57、可能是最全的页面置换算法总结了
 1、最佳置换法(OPT)
 2、先进先出置换算法(FIFO)
 3、最近最久未使用置换算法(LRU)
 4、时钟置换算法(CLOCK)
 5、改进型的时钟置换算法
 6、总结
58、共享是什么？
59、死锁相关问题大总结，超全！
 1、死锁产生原因
 2、死锁演示
 3、死锁的解决方案
 4、死锁必要条件
 5、处理方法
 6、死锁恢复
 7、死锁预防
 8、死锁避免
60、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？
61、内部碎片与外部碎片
62、如何消除碎片文件
- ### 3.4、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
 总结
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
 第二种回答
3、你知道DNS是什么？
4、DNS的工作原理？
5、为什么域名解析用UDP协议？
6、为什么区域传递用TCP协议？
7、HTTP长连接和短连接的区别
8、什么是TCP粘包/拆包？发生的原因？
 原因
 解决方案
9、为什么服务器会缓存这一项功能？如何实现的？
10、HTTP请求方法你知道多少？
11、GET 和 POST 的区别，你知道哪些？
12、一个TCP连接可以对应几个HTTP请求？
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）？
14、浏览器对同一 Host 建立 TCP 连接到数量有没有限制？
15、在浏览器中输入url地址后显示主页的过程？
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
 第二种回答

- 17、谈谈DNS解析过程，具体一点
- 18、DNS负载均衡是什么策略？
- 19、HTTPS和HTTP的区别
- 20、什么是SSL/TLS？
- 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）
- 22、如何保证公钥不被篡改？
- 23、HTTP请求和响应报文有哪些主要字段？
 - 请求报文
 - 响应报文
- 24、Cookie是什么？
- 25、Cookie有什么用途？用途
- 26、Session知识大总结
- 27、Session的工作原理是什么？
- 28、Cookie与Session的对比
- 29、SQL注入攻击了解吗？
- 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
- 32、端口有效范围是多少到多少？
- 33、为何需要把TCP/IP协议栈分成5层（或7层）？开放式回答。
- 34、DNS查询方式有哪些？
 - 递归解析
 - 迭代解析
- 35、HTTP中缓存的私有和共有字段？知道吗？
- 36、GET方法参数写法是固定的吗？
- 37、GET方法的长度限制是怎么回事？
- 38、POST方法比GET方法安全？
- 39、POST方法会产生两个TCP数据包？你了解吗？
- 40、Session是什么？
- 41、使用Session的过程是怎样的？
- 42、Session和cookie应该如何去选择（适用场景）？
- 43、Cookies和Session区别是什么？
- 44、DDos攻击了解吗？
- 45、MTU和MSS分别是什么？
- 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）
- 47、TCP头部中有哪些信息？
- 48、常见TCP的连接状态有哪些？
- 49、网络的七层/五层模型主要的协议有哪些？
- 50、TCP是什么？
- 51、TCP头部报文字段介绍几个？各自的功能？
- 52、OSI的七层模型的主要功能？
- 53、应用层常见协议知道多少？了解几个？
- 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？
- 55、三次握手相关内容
 第一种回答
 第二种回答
- 56、为什么需要三次握手，两次不行吗？
- 57、什么是半连接队列？
- 58、ISN(Initial Sequence Number)是固定的吗？
- 59、三次握手过程中可以携带数据吗？
- 60、SYN攻击是什么？
- 61、四次挥手相关内容
 第一种回答
 第二种回答
- 62、挥手为什么需要四次？
 第一种回答
 第二种回答
- 63、2MSL等待状态？

- 64、四次挥手释放连接时，等待2MSL的意义?
两个理由
- 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
第二种回答
- 66、TCP粘包问题是什么？你会如何去解决它？
- 67、OSI七层模型中表示层和会话层功能是什么？
- 68、三次握手四次挥手的变迁图
- 69、对称密钥加密的优点缺点？
- 70、非对称密钥加密你了解吗？优缺点？
- 71、HTTPS是什么
- 72、HTTP的缺点有哪些？
- 73、HTTPS采用的加密方式有哪些？是对称还是非对称？
- 74、为什么有的时候刷新页面不需要重新建立SSL连接？
- 75、SSL中的认证中的证书是什么？了解过吗？
- 76、HTTP如何禁用缓存？如何确认缓存？
- 77、GET与POST传递数据的最大长度能够达到多少呢？
- 78、网络层常见协议？可以说一下吗？
- 79、TCP四大拥塞控制算法总结？（极其重要）
慢热启动算法 - Slow Start
拥塞避免算法 - Congestion Avoidance
拥塞发生状态时的算法
快速恢复算法 - Fast Recovery
- 80、为何快速重传是选择3次ACK？
- 81、对于FIN_WAIT_2, CLOSE_WAIT状态和TIME_WAIT状态？你知道多少？
- 82、你了解流量控制原理吗？
- 83、建立TCP服务器的各个系统调用过程是怎样的？
- 84、TCP协议如何保证可靠传输?
第一种回答
第二种回答
第三种回答
- 85、UDP是什么
- 86、TCP和UDP的区别
- 87、UDP的特点有哪些（附赠TCP的特点）？
- 88、TCP对应的应用层协议
- 89、UDP对应的应用层协议
- 90、数据链路层常见协议？可以说一下吗？
- 91、Ping命令基于哪一层协议的原理是什么？
- 92、在进行UDP编程的时候，一次发送多少bytes好？
- 93、TCP利用滑动窗口实现流量控制的机制？
- 94、可以解释一下RTO, RTT和超时重传分别是什么吗？
- 95、XSS攻击是什么？（低频）
- 96、CSRF攻击？你知道吗？
- 97、如何防范CSRF攻击
- 98、文件上传漏洞是如何发生的？你有经历过吗？
- 99、如何防范文件上传漏洞
- 100、拥塞控制原理听说过吗？
- 101、如何区分流量控制和拥塞控制？
- 102、常见的HTTP状态码有哪些？
1xx 信息
2xx 成功
3xx 重定向
4xx 客户端错误
5xx 服务器错误
- 3.5、MySQL
- 3.6、Redis
- 3.7、常见智力题、情景题
- 3.8、常见非技术性问题

4、优质面经

- 4.1、阿秀个人秋招总结文章：双非渣硕的秋招之路总结（已拿抖音研发岗SP）
- 4.2、朋友先后折戟腾讯、字节、快手、网易、滴滴、深信服后，终于成功上岸了

背景介绍

一面

二面

- 4.3、虚度大——年又如何，双非本科大三学弟连斩腾讯字节

Offer情况

背景介绍

腾讯面经

字节面经

结语

5、内推信息

- 5.1、字节跳动
- 5.2、携程
- 5.3、其余公司

更新记录

V1.1 - 2021.02.27

- 增加C/C++部分 217 道面试题

V1.2 - 2021.03.23

- 增加操作系统部分 62 道面试题
- 增加计算机网络部分 33 道面试题
- 粉丝建议增加页码，已采纳
- 再次增加计算机网络部分 33 道面试题，计网部分目前共计 66 道题
- 增加个人秋招时期 26 版简历迭代修改过程，位于 1.1.1 小节

V1.3 - 2021.04.05

- 累计 14 位小伙伴指出部分问题解答存在错误，现已改正，感谢这 14 位小伙伴。
- 增加计算机网络部分 36 道面试题，计网部分目前共计 102 道题
- 增加常考高频算法 9 道
- 开放编辑权限，目前PDF可以复制、编辑、注释等（前期主要考虑盗版太严重，但很多粉丝都来私信说希望开放编辑权限，也好做标记，思前想后决定安排上！）

1、看这里，十分重要！！！

本系列持续更新中，目前已收录 C/C++、操作系统、数据结构、计算机网络、MySQL、Redis、以及面试常见问答等面试资料，未来打算继续收录Java、Python、Go等面试常见问题。

本笔记也是帮助阿秀在 2020 年校招中体会到一把offer收割机的快乐(个人秋招总结文章见 4.1 小节),阿秀凭借本笔记拿到字节跳动、华为、百度、农业银行等公司的offer，最终签约字节跳动旗下~

现已将该面试总结开源到github仓库上：<https://github.com/forthespada/InterviewGuide>，坚持将此开源仓库维护下去，立志打造最全面面试宝典，欢迎各位star，第一时间了解最新面试题型。

如果你发现你手中的**PDF内容不全**，请扫描下方**右侧公众号『拓跋阿秀』**回复关键字“**PDF**”即可领到**最新最全的《逆袭进大厂》PDF电子版本**。

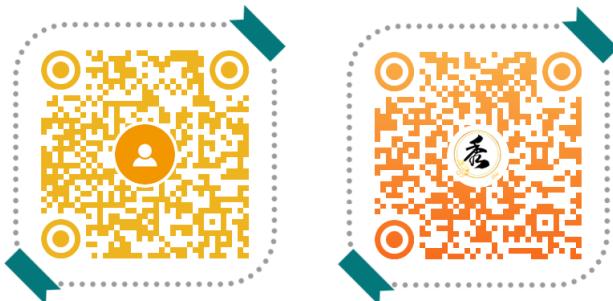
画黑板，关键字是“**PDF**”。

是的，永远是 **github仓库与个人公众号上的面试题是最新的**...传到你手里的可能是不知道多少版之后的了。

并且感谢各位小伙伴对本面试宝典提出的一些改进意见，对于其中的笔误阿秀也在不断更新。

比如这个 **PDF** 是 **2021.04.05** 就整理好了的，可能三五天之后就会更新下一期了。

左侧个人微信，右侧公众号，专注计算机小白、新手，以求帮助更多像我一样的普通学生



长按来找阿秀唠嗑 觉得不错就长按关注阿秀

写于**2021.04.05**，祝大家顺利拿到offer

对了，我还有一个**计算机书籍籍仓库**来着，仓库地址：<https://github.com/forthespada/CS-Books>，欢迎白嫖~

超过1000本的**计算机经典书籍**、个人笔记资料以及本人在各平台发表文章中所涉及的资源等。书籍资源包括C/C++、Java、Python、Go语言、数据结构与算法、操作系统、后端架构、计算机系统知识、数据库、计算机网络、设计模式、前端、汇编以及校招社招各种面经~

2. 面试准备

2.1、简历篇

2.1.1、求求你了，不要乱写简历了

很多人貌似根本不知道间的重要性，简历真的很重要，比你们想象中的要重要得多！

简历是你跟公司的第一次接触，简历不合格直接pass！纵然你有**张翼德一般万军从中取上将首级**的本领，压根不能上场又能有什么用呢？

你的简历根本到不了面试官手中，就连简历关你都过不了还说什么呢？你说对吧！

2.1.2、阿秀个人26版秋招简历迭代修改过程

现先把自己以前修改简历的文章贴一下，简历的写法千千万，我只是分享一下自己的简历修改过程，不喜勿喷！

此文首发于个人公众号：拓跋阿秀

原文链接：<https://mp.weixin.qq.com/s/VrTP58lOui3TQzXoRPQPlg>

如需要阿秀个人简历模板与其余优秀简历模板，扫描下面阿秀个人公众号下回复关键字“**阿秀简历**”即可领取



阿秀在秋招中直接简历挂的只有苏州微软以及拼多多了

其他的互联网公司基本都给面试机会了，靠的是什么？靠我**普通双非学校**的学历吗？

不是，靠的就是我迭代了足足 **26** 版的个人简历。

现在阿秀把它分享出来，跟大家分享如何打造自己的简历，然后如何去迭代它，最终直到完美。

英文简历.docx	2020/6/28
中文简历_v1.docx	2020/6/28
中文简历_v2_详细一点.docx	2020/6/30
中文简历_v3.docx	2020/7/6 8
中文简历_v4_C++工程师.docx	2020/7/7 1
中文简历_v5_修改C++工程师.docx	2020/7/7 1
中文简历_v6_加上信息版C++工程师.docx	2020/7/7 1
中文简历_v7.docx	2020/7/8 1
中文简历_v8.docx	2020/7/10
中文简历_v9_包装版本.docx	2020/7/10
中文简历_v10_再次包装.docx	2020/7/22
中文简历_v11_加github.docx	2020/7/22
中文简历_v12.docx	2020/8/9 1
中文简历_v13_加时间.docx	2020/8/10
中文简历_v14.docx	2020/8/13
中文简历_v15.docx	2020/8/13
中文简历_v16_改时间.docx	2020/8/13
中文简历_v17-改项目描述.docx	2020/8/13
中文简历_v18.docx	2020/8/13
中文简历_v19_润色.docx	2020/8/15
中文简历_v20_改校园经历.docx	2020/8/16
中文简历_v21_改.docx	2020/8/16
中文简历_v22.docx	2020/8/17
中文简历_v23.docx	2020/8/17
中文简历_v24_最终版v1.docx	2020/8/17
中文简历_v25_最终版v2.docx	2020/8/18
中文简历_v26_最最最终版.docx	2020/8/21

引言

首先需要明确一点的就是简历上只是你展示自己的一种途径，最终决定面试成功与否的**关键在于简历内容**，而不是简历本身，获得offer的决定权在你自身实力，如果你本来就很水，简历做得再好也没啥用。

我拿出我第**1**版、**15**版、**26**版的简历为大家展示其中的变化，以下分为**个人信息、专业技能、实习经历、项目经验、奖项荣誉**等这几个部分分别叙述。

个人信息

不扯虚的，直接看效果

韩 立

应聘职位：C++研发工程师

性别：男 电话：XXXXXXXXXXXX 邮箱：XXXXXXXXXXXX@foxmail.com

教育背景

2018.09 - 2021.06	家里蹲大学	计算机技术	工学硕士(99/100)
2013.09 - 2017.07	窝里睡大学	计算机科学与技术	工学学士
2014.09 - 2017.07	窝里睡大学	英语(第二学位)	文学学士

韩 立

应聘职位: C++工程师

性别: 男 电话: XXXXXXXXXXXX 邮箱: XXXXXXXXXXXXXXX@foxmail.com

教育背景

2018.09 - 2021.06	家里蹲大学	计算机技术	工学硕士(99/100)
2013.09 - 2017.07	窝里睡大学	计算机科学与技术	工学学士
2014.09 - 2017.07	窝里睡大学	英语(第二学位)	文学学士

韩 立

应聘职位: C++开发工程师

性别: 男 电话: XXXXXXXXXXXX 邮箱: XXXXXXXXXXXXXXX@foxmail.com

教育背景

2018.09 - 2021.06	家里蹲大学	计算机技术	工学硕士(Top 5%)
2013.09 - 2017.07	窝里睡大学	计算机科学与技术	工学学士
2014.09 - 2017.07	窝里睡大学	英语(第二学位)	文学学士

要点

- 1、关键信息该有的要有，比如姓名、性别、电话、邮箱，其余的年龄/出生日期、籍贯等可以自己选择写不写，其中**关键部分**该大写大写、该加粗加粗，比如姓名、应聘职位，醒目一点比较好。
- 2、这里小技巧，在写排名的时候，如果你年级前 **10**，那你直接写年级排名/总人数，比如 **9/100**。如果你年纪排名不高，那你就换成 **Top 20%、Top 30%** 这个样子，后者给人的第一感受是比 **20/100、30/100** 这种要好上一些。要知道秋招时期，简历实在是太多了，**HR**在一份简历上用时可能就**10秒-30秒**之间，所以你的简历给人的第一印象很重要。
- 3、关键时间点写清楚，教育经历的时间一般都是从后往前的，把最高学历写在第一个，这个是改不了的，千万不要试图学历造假。
- 4、要把应聘岗位写清楚。你来应聘公司基本的诚意要有吧，别人招“服务端研发工程师”，你写个“**C++开发工程师**”，明显没什么诚意啊，一看就是海投选手，简历被刷的可能就多一分了，我每投一个简历肯定把应聘岗位这一栏修改为对应的岗位名称，不要一份简历投天下。

 韩立 - 家里蹲大学 - 后端开发工程师.pdf	2020/8/18 17:41
 韩立 - 家里蹲大学 - 编译器开发工程师.pdf	2020/8/18 19:29
 韩立 - 家里蹲大学 - 游戏后端开发工程师.pdf	2020/8/20 20:35
 韩立 - 家里蹲大学 - 游戏开发工程师.pdf	2020/8/25 15:57
 韩立 - 家里蹲大学 - C++工程师.pdf	2020/8/25 22:35
 韩立 - 家里蹲大学 - 后台开发工程师.pdf	2020/8/26 0:56
 韩立 - 家里蹲大学 - 编译器研发工程师.pdf	2020/8/27 8:42
 韩立 - 家里蹲大学 - 云平台开发.pdf	2020/8/27 14:44
 韩立 - 家里蹲大学 - 后端研发工程师.pdf	2020/8/28 20:26
 韩立 - 家里蹲大学 - 游戏开发工程师.pdf	2020/8/29 16:42
 韩立 - 家里蹲大学 - C++开发工程师.pdf	2020/8/31 8:44
 韩立 - 家里蹲大学 - 后端开发工程师.pdf	2020/8/31 11:14
 韩立 - 家里蹲大学 - 后端开发工程师-电商业务.pdf	2020/8/31 11:26
 韩立 - 家里蹲大学 - C++类研发工程师.pdf	2020/8/31 12:14
 韩立 - 家里蹲大学 - GoLang研发工程师.pdf	2020/9/7 22:37
 韩立 - 家里蹲大学 - 软件开发工程师.pdf	2020/9/20 21:25

- 5、放不放照片？个人建议不放，如果你是运营、产品岗可以考虑放或者对自己相貌有自信的同学直接放上去就完事了；技术岗真的没必要。

专业技能

来吧，直接看三版区别。

专业技能

- 熟练使用 C++ 进行编程，基本功扎实，熟悉 Python、SQL 等；
- 了解 socket 编程及 TCP/IP 协议栈；
- 了解 Linux 操作系统及基本命令；
- 了解常见数据库 MySQL/Redis；
- 熟练 Visual Studio 等专业软件的使用和调试；

专业技能

- 熟练使用 C++ 进行编程，熟悉 Python、SQL 等；
- 熟悉网络通信，能够利用 Socket 进行网络编程；
- 具有分布式环境下的系统编码经验；
- 熟悉 Linux 操作系统，能够使用 Linux 环境从事开发；
- 熟悉常见数据库 MySQL/Redis，了解常见 MySQL 下的性能优化；
- 熟练 Visual Studio、GDB 等专业软件的使用和调试；

专业技能

- 熟练使用 C++ 编程语言，熟悉 STL 下常见容器底层数据结构，了解 Python、SQL 等；
- 熟悉常见数据结构及算法，如十大排序（快速排序、归并排序、堆排序等）；
- 熟悉 OSI 七层模型，掌握 HTTP、TCP/UDP、IP 等常见协议；
- 对 Linux 下 I/O 复用技术有深刻理解，能够利用 Socket 套接字进行网络编程；
- 熟悉 Linux 环境下常用命令及相关工具的使用（gcc、gdb、vim、git 等）；
- 具有分布式环境下的系统编码经验，能够实现多机间的网络通信与数据交互；
- 了解常见数据库 MySQL/Redis，了解 MySQL 下的性能优化以及 Redis 底层通信模型；

要点

1、技能栈可能是简历中最重要的两项之一（另一个是实习/项目），如果不匹配的话面试官也没兴趣问下去。比如你是 **C++** 岗位，但是写了很多熟悉**CNN**深度学习、**TensorFlow**框架啥的，挺厉害的，但是抱歉，我们不招你这样的。

2、用好动词：了解、熟悉、掌握、精通四个等级，一般不建议写精通，因为你可能会被怼到怀疑人生。

3、把技能栈描述清楚，并且适当展开。比如不要光秃秃地写一句“熟悉数据结构与算法”，适当展开一下，比如我熟悉十大排序中的快排、归并、堆排，但是其实还有计数排序和桶排序我没有写出来，留了一手。因为面试官很多时候看你会什么反而不问你什么，他会默认你会这些东西，他问一些拓展的点，如果你拓展的能答出来，那写在简历上的还能不会吗？

比如你可以写“冒泡、快排”，不写归并，这样面试官问的时候可能会问“除了冒泡快排，你还知道什么排序？”这个时候你再把你藏起来的归并说出来，并且手撕一下。这一回合下来肯定很加你的面试分啊，面试官以为考查到了你不会的点，其实这都是在你的计划之内的。正所谓**虚虚实实，虚则实之，实则虚之**~满满的都是套路啊。

4、注意细节部分。该大写大写该小写小写，比如 **C++** 而不是 **c++**、**Java** 而不是 **java**、**Python** 而不是 **python**、**MySQL** 而不是 **mysql**、**Redis** 而不是 **redis**，往往细节决定成败，一点一滴慢慢做起。

实习经历

不知道你们发现没有，我在实习经历这一栏字是越来越多的，描述也是越来越具体的。

实习经历

2018.12 - 2019.06 狮驼岭快递有限公司

数据分析师

- XXXXXXXXXXXX 信息查询网爬虫系统研发，在系统中负责双重验证码的破解与解除账号次数限制。
- 对公司提供的 XXX 号以及 XXXX 信息，进行基本 XXX 信息抓取，该系统难点在于登录时的验证码验证为中文点选验证码识别，主要通过将验证码图片进行保存到本地然后接入打码平台返回相应坐标进行解决。

实习经历

2019.03 - 2019.06 狮驼岭快递有限公司

数据分析师

- XXXXXXXXXX 信息查询网信息抓取系统研发，在系统中负责双重验证码的破解与解除账号次数限制。
- 对公司提供的 XX 号以及 XXX 信息，进行基本 XXX 信息抓取。该难点在于登录时的验证码验证为中文点选验证码识别，主要通过将验证码图片进行保存到本地然后接入打码平台返回相应坐标进行解决，通过伪造 Cookie 来解决账号次数限制，增加账号重复利用率。

实习经历

2019.03 - 2019.06

狮驼岭快递有限公司

程序开发

- 工作描述：**1、【技术研发】负责人：主要负责项目的 XXXX 重构工作与后续点选验证码的破解；
2、【团队支持】主要参与者：负责项目环境搭建以及验证码平台的洽谈接入；
3、【项目落地】主要参与者：参与 XXXXXX 系统在公司的落地。

成果描述：成功为公司提供 7*24 小时的无间断 XXXX 服务，每秒可更新 32 条以上的 XXXX 信息，每周可成功更新数据 20W 条以上。

个人收获：对于一个完整的项目重构过程有了更深刻的认识和理解，掌握了更多线程间的通信和数据交换细则，使自己明白了快速学习新知识并将其运用到业务开发的重要性。

要点

- 1、对于互联网公司来说，实习是加分项，特别是**大厂实习**，非常加分！所以小伙伴们能去实习一定要去实习。
- 2、实习过程中要注意找实习中的亮点、难点所在，不要老老实实的把整天写SQL这种杂活一板一眼地写出来了，适当包装自己的工作并不过分，毕竟“**面试造火箭，工作拧螺丝**”，面试可能是最难的一关了。
- 3、好好描述实习经历。这里并没有什么固定的模板，一般来说可分为：实习背景、所用技术、实习难点、实习成果、个人收获。

实习背景：尽量精简一点，或者像下面我所写的项目背景那样写

所用技术：尽可能贴合当前应聘的岗位，把用到的技术说出来。或者你也可以像我这样，我当时实习是一个爬虫岗位，用的是 **Python**，跟 **C++** 其实并不相关，所以我尽量避免 **Python** 字眼的出现。学着灵活一点~

实习难点：可以按照“针对XXX问题，采用XXX技术，成功实现了XXXX，最终XXXXX”这样的格式来写，描述清楚。

实习成果：最好要有具体的指标说明，比如我简历中的“**7*24**小时**”、“**32条**”、“**20W***”都是具体的量词，不要说“极大地提高了XX效果”，这种虚的话。如果你不知道具体指标，自己瞎编一个，不要太离谱就行。。。

个人收获：站在面试官的角度来说，他是希望看到你在一段实习经历中学到了什么的，所以最好也要有个人收获说明。

项目经验

因为我实习经历本来就不是很好，并不匹配我要应聘的岗位，所以我把重点放在了项目上，我准备的项目真的是花了大心思了，看这三版变化你就能够看出来了。

(1) 主流电商平台数据爬取（实验室课题项目）

项目描述: 以京东、苏宁、国美三大电商平台为目标，抓取全站商品基本信息以及评论数据。

主要工作: 将电商网站商品数据按照类别分别进行商品信息的爬取，后期将国美和苏宁的部分利用 Scrapy 框架进行封装，提升了整体抓取速度。

项目开源: <https://github.com/forthespada/E-commerce>

(2) CPU+GPU 模式下的大规模 XXX 知识管理技术研究（实验室课题项目）

项目描述: 以大规模 XXX 知识管理为目标，系统地研究面向 CPU+GPU 计算模式下高性能数据管理技术以及面向 XXX 的数据查询技术。

主要工作: 将属性相似或相近的商品实体划分到相同的存储结点，然后将 CPU 下的查询任务转移到 GPU 中，借助 GPU 强劲计算能力提高大规模数据下 XXX 及相关信息的检索效率。

(1) CPU+GPU 模式下的 XXX 知识管理技术研究（实验室课题项目）

项目描述: 以大规模 XXXX 知识管理为目标，系统地研究在分布式环境下面向 CPU+GPU 计算模式下高性能数据管理技术以及面向 XXX 的数据查询技术。

主要工作: 将属性相似或相近的 XXXX 实体划分到相同的存储机器节点，然后将 CPU 下的查询任务进行划分转移到不同机器中分别执行，在子任务机器中借助 GPU 的强劲计算能力来进行 XXXX 及相关信息的检索，最后将 XXXXXX 进行汇总合并以得到最终结果。

(2) 主流电商平台数据爬取（实验室课题项目）

项目描述: 以京东、苏宁、国美三大电商平台为目标，抓取全站商品基本信息以及评论数据。

主要工作: 将电商网站商品数据按照类别分别进行商品信息的爬取，后期将国美和苏宁的部分利用 Scrapy 框架进行封装，提升了整体抓取速度。

项目开源: <https://github.com/forthespadan/E-commerce>

(1) 2019.12-2020.06**基于 GPU 的分布式环境下的 XX 知识管理技术研究**

应用技术: Linux、C/C++、Socket、UDP、MySQL、CUDA。

项目描述: 借助分布式+GPU，解决因 XXXX 而带来的 XXXX 及其属性特征检索效率低下的问题，辅助 XXX 快速进行 XXXX 以及 XXXXX。个人主要负责 PC 机之间的底层 Socket 通信、CPU 环境下的 XXXXX、GPU 环境下的 XXXXX 与后续系统优化等部分。

主要工作: 1、主机节点采用 XXXXXX 与 4 个子节点完成网络通信和数据传输，后改为 XXXXX 模式与 XXXX 方式，减少 XXXXXX 时间的同时保证了 XXXX 兼容性；
2、将 XXXX 进行划分，然后分配到 4 个节点中，选取 XXXXXXXXXXXXX 开始进行连接，可有效减少中间结果的产生；
3、在 XXXX 环境下，使用 XXXX 技术将 XXXXXXXXXXXXX 转化为 XXXXX 来 XXXXX，有效增加 XXXXX22.8% 以上。

项目成果: 系统已 XXXXXXXXXXXXX 使用，实行一月后的反馈报告显示同比可有效增加 XXXXX 的 30.6% 以上。

个人收获: 对于不同系统下的 XXXX 技术有了更深的理解，也明白了技术不是越新的越好，而是最适合当前项目的才是最应该选取的技术。

(2) 2020.02-2020.04**基于 Linux 的轻量级多线程 HTTP 服务器**

项目开源: <https://github.com/forthespada/MyPoorWebServer>

应用技术: Linux、C++、Socket、TCP。

项目描述: 此项目是基于 Linux 的轻量级多线程 Web 服务器，应用层实现了一个简单的 HTTP 服务器，支持静态资源访问与动态消息回显。

主要工作: 1、实现快速地址再分配，避免紧急情况下服务器因宕机而引起的服务失效；
2、实现 get/post 两种请求解析，采用 cgi 脚本进行 post 请求响应；
3、利用多线程机制提供服务，增加并行服务数量；
4、利用双管道进行不同进程间通信与数据交换，及时关闭无用管道。

个人收获: 个人对于 HTTP 的服务过程有了更清晰的认识，对于 TCP 和网络编程也有了一定的理解。

项目开源: <https://github.com/forthespada/E-commerce>

应用技术: Python、Scrapy、Json、Redis、MongoDB、布隆过滤器

项目描述: 此项目是采用 Scrapy 框架, 对京东、苏宁、国美三大网站的商品以及评论信息进行抓取的一个系统。

个人主要负责的是网站解码、反扒解决、数据去重、撰写报告等部分。

主要工作: 1、对于不同网站的防爬策略, 做了不同有效的解决工作;

2、负责网站界面的 Xpath 解析编码, 获取商品以及评论相关信息;

3、前期采用布隆过滤器进行数据去重, 后期改进相关策略, 有效减少去重时间;

4、采用 Redis 队列实现快速爬取, 缩短数据搜集时间;

5、对抓取数据进行初步分析以及数据清洗后, 采用 Json 格式进行存储;

6、负责最终团队工作报告的编写与汇报。

项目成果: 最终成功抓取商品数据 980W, 评论数据 6800W, 为上述实验室机器学习相关课题准备充分数据集, 已发表 XXXXXX 论文 2 篇。

个人收获: 对于数据在网络上的传输过程有了更清晰的认识和了解, 对于数据在传输过程中的保密措施也有了一定的了解。

要点

1、项目数量不放太多, 2-3 个足够了, 太多了反而不好, 把最想介绍的、把握最大的放在第一个。

2、自己准备的项目千万要注意贴合岗位要求, 或者你像我这样直接把实验室的项目拿出来, 包装一下。

3、项目描述一定要清晰明了, 好好描述。可以参考上面的实习经历, 大致分为: 项目背景、所用技术、项目工作 (自己承担的工作) 、项目成果、个人收获, 重点描述个人工作部分, 这是灵魂所在。简要说来就是“针对XXX问题, 采用XXX技术, 成功实现了XXXX, 最终XXXXXX”, 或者像我那样写也是可以的。

4、至于怎样在面试中介绍自己的项目, 可以参考这篇文章: [如何优雅的介绍自己的项目经历](#)。

5、建议你每次面试的时候都用手机录音再进行面试复盘, 听一听自己在面试过程中哪里答的不好, 针对性的对简历进行修改, 这样你的简历才能逐渐完美。

奖项荣誉

因为简历篇幅有一页多, 但两页又不太够的样子, 所以在第 26 版简历中我把两个比较有含金量一点的奖项单独拎出来介绍了一下, 把简历凑到了两页左右。

奖项荣誉

- 2020 年 华为软件精英挑战赛-粤港澳赛区 64 强
- 2019 年 第一届广西大学生人工智能设计大赛一等奖
- 2019 年 研究生学业奖学金二等奖
- 2018 年 第十届“外研社”全国英语阅读大赛一等奖、翻译大赛优秀奖
- 2018 年 研究生学业奖学金二等奖
- 2014 年 学校二等奖学金

等级证书

英语: 六级 (CET-6) 、六级口语 (CET-SET-6), 具备良好听说读写能力, 能够快速浏览英语专业书籍。

其他: 普通话二级乙等、驾照。

奖项荣誉

- 2020 年 华为软件精英挑战赛-粤港澳赛区 64 强
- 2019 年 第一届广西大学生人工智能设计大赛一等奖
- 2019 年 研究生学业奖学金二等奖
- 2018 年 第十届“外研社”全国英语阅读大赛一等奖、翻译大赛优秀奖
- 2018 年 研究生学业奖学金二等奖
- 2014 年 学校二等奖学金

等级证书

英语: 六级 (CET-6) 、六级口语 (CET-SET) , 具备良好听说读写能力, 能够快速浏览英语专业书籍。

其他: 普通话二级乙等、驾照。

校园经历

2020.04 华为软件精英挑战赛-粤港澳赛区 64 强

2019.11 第一届广西大学生人工智能设计大赛(一等奖)

作品描述: 基于 XXXX 的智慧课堂系统，该作品主要是建立模型对课堂录像中截取的图片进行处理分析，通过人脸识别与对齐，进而得到不同学生在课堂上的情绪表现，进而分析出学生在课堂上的用心和专注程度。

主要工作: 1、赛前参与作品整体规划，XXXX 关键技术的选取；

2、负责构建所要获取的数据标准，并亲自进行数据的采集工作；

3、负责人脸检测与对齐的功能，实现情感分析功能；

4、负责后期系统 UI 的编写，为参赛作品提供良好展示平台与媒介；

5、负责监控整体代码质量，保证程序健壮性。

奖项荣誉

➤ 2020 年 华为软件精英挑战赛-粤港澳赛区 64 强

➤ 2019 年 第一届广西大学时人工智能设计大赛一等奖

➤ 2019 年 研究生学业奖学金二等奖

➤ 2018 年 第十届“外研社”全国英语阅读大赛一等奖、翻译大赛优秀奖

➤ 2018 年 研究生学业奖学金二等奖

➤ 2014 年 学业奖学金二等奖

等级证书

英语：六级（CET-6）、六级口语（CET-SET），具备良好听说读写能力。

其他：普通话二级乙等、驾照。此外，热心参加于 **Github** 开源社区，拥抱技术共享时代。

要点

1、该加粗加粗。比如第 15 版中我就把两个“**一等奖**”和“**六级**”字眼加粗了，像本科的可以在这里写上一些社团经历或者参加的比赛之类的，毕竟你在学校里呆三四年还是要拿一点事情出来证明你这四年是干了点事的，如果没有的话，那就把你室友的经历借来使使，注意不要说漏嘴了就行。

2、如果要写上**个人评价**也是可以的，但麻烦有点新意和个人思考在里面，不要千篇一律的写自己“爱看书、爱学习”之类的，太假了...那你可以写自己爱看书，比如在“每年当当网消费XXX元，每年固定看 5 本技术书；热心于网络课程，在 B 站学习 Java/C++/前端知识，我在 B 站学知识”等还是可以写一写的。

忠告

对于我们普通人而言，要找到一份不错的互联网工作并没有那么容易。希望大家早做准备，平时多积累，不要等到每年的七八月份快找工作的时候，再去准备写简历，那个时候你很可能发现自己啥也没有，过往经历和技术栈就跟一张白纸一样，两眼一抹黑，啥也写不出来。

有意识的去积累可以写在简历上的内容，**稳扎稳打、步步为营、一步一个脚印**到最后你才能收获满满的果实，相信我到后期你的收获可能会远远超出自己的预期，那些你以前想也不敢想的公司居然会变得触手可得。

2.1.3、多用一些简历网站

说真的，千万不要过分相信自己的排版能力，很可能你的简历连对齐都没有对齐。

建议使用一些在线的简历网站，别人把排版给你做好了，直接往里面填内容，它不香吗？

完事了，自动生成简历，排版大气又能节省时间，直接导出PDF格式即可，不省事吗？

一些比较不错的简历网站，比如超级简历、500丁之类都不错。

这里也推荐一个小众的简历网站：**职徒简历**，还不错。是我在找工作期间华为 **HR** 告诉我的，这个网站貌似刚建立没多久所以还能白嫖七天会员，白嫖不要钱的事可不能少了我⑩。

扫描下方二维码就能白嫖七天会员了



2.1.4、投递简历的三种方式

对于校招应届生来说，投递简历的途径主要有三种，1、校园宣讲会 2、网申 3、内推。

1、校园宣讲会

校招宣讲会一般都会有一些笔试，笔试过了即可参加正式的面试，去的时候带好笔试的东西即可。

现在也有一些线上的宣讲会，经常都是会有一些互动环节的，观众提出自己比较关心的问题，然后主持人或者嘉宾现场回答，对于那些被抽中问题的同学一般会有一些小礼品发放比如U盘，鼠标垫之类的。

除此之外还会有一些直通面试卡的发放，也就是不需要笔试，简历筛选过关即可直接面试。

2、网申

网申基本算是投递人数最多的一种方式，通过招聘网站或者官网进行信息的填写，直接投递即可。需要注意的是，不同公司用的简历系统都是独立的，所以很有可能你每投递一家公司就要填写一次，真的真的很累人滴！最开始填写的时候可能还是比较有耐心，慢慢的你就会觉得有点烦，阿秀建议午饭后坐在座位上没事了就填一个，哈哈。

3、内推

这也是近几年越来越流行的一种方式了，直接找意向公司内部的员工帮忙推荐即可，比如已经入职成功的学长学姐，阿秀建议能找内推一定要找内推，否则简历很容易石沉大海，连求职进度都不知道。

此外牛客网上也经常有一些人发布招聘信息，基本都会附赠内推码或者内推邮箱或者搜寻自己想要投递的岗位获取内推码，然后发邮件过去就好，还有部分公众号也是可以找到内推信息的，注意关注即可。

内推对于内推人来说也是有好处的，基本上每个大厂内推成功都是有奖金的，他内推你，如果你能够顺利入职的话，内推员也是会拿到奖金的，内推基本是双赢的。

虽然很多人说现在人均内推，基本没啥用，但是我还是觉得能内推就内推，人均内推+你不内推，那你不就吃亏了吗，说不定能起作用呢。

这里有两个需要注意的点：

1、注意及时添加内推人的联系方式比如微信QQ等，及时了解内推信息，很多同学随便找了一个内推码就写上去了，到最后也不知道自己的内推情况到底如何了。

2、注意发送简历的格式，最好是 PDF 的，这样简历格式才不会改变。word在不同电脑上格式可能会有所变化，还有就是别人无法修改你的简历，内推人也能更快更方便的打开。

发送给招聘邮箱的简历也要注意命名和格式：一般只发中文版本，除非你投递的岗位明确要求需要英文简历，简历命名格式为：姓名+职位+意向工作地+电话，邮件标题也命名为：内推+姓名+岗位+意向工作地+电话，因为这样也方便别人更快地把你的简历推送到合适的岗位中去。

予人方便也是于己方便，相比于标题为“工作”的邮件，内推人或者HR也更愿意打开一份标题为“内推+姓名+岗位+意向工作地+电话”的邮件，你说是不？

2.2、了解校招

2.2.1、校招重要时间点

校招只是一个比较统一的时间，又可以具体细分为：暑期实习招聘、秋招、秋招补录、来年春招、甚至是春招补录。

暑期实习招聘：暑期实习的招聘对象主要是大三下和研二下的学生，也就是每年的3-5月份左右，招聘规模比正式的校招要小一些。对于一些确定要工作而不是读研和不是要读博的同学（特别是专硕）来说，一定要尽自己的最大力量去参加实习。这是因为：

1、一段甚至若干段实习经历是可以写在简历上的，还有就是实习是有几率转正的。通过实习上岸可比通过正式秋招上岸要容易一些的，这相当于你在正式秋招前就已经有了一个保底的offer了，在秋招过程中压力也没有那么大。

阿秀的所见所知告诉我：**不要把鸡蛋放在一个篮子里。**

也就是说不是很建议因为已经有了实习offer就放弃正式秋招，阿秀身边有很多人都后悔去了实习的公司了，有一些是转正失败，还有一些则是转正成功后给的offer待遇比较低。

还有一个需要注意的点就是有些公司可能说是有转正名额的，但是转正率很低，10个里可能就只有1个能成功转正，建议实习的小伙伴一定要及时跟自己的小组长请教转正事宜，以及私下里向组内前辈们问转正的情况，争取成功上岸。

2、实习过程是可以写在简历上，并且为自己的简历增色不少的，如果你有互联网大中厂的实习经历那就更好了，二面三面的面试官在看到你有实习的时候一般都会深挖你的实习过程的，这样在面试的时候也有的聊。

秋招：秋招又分为**提前批**和**正式批**两个阶段，甚至还有**秋招补录**这一环节。

其中提前批一般集中在6月-7月，正式批一般是7月-10月，有个词叫做“金九银十”，就是在形容秋招时期九月和十月是竞争最激烈的时候，因为这个时候基本上互联网公司都开始秋招了，求职者也是最多，竞争也是最大的时候。

- **提前批：**提前批主要是一些公司为了能够更快的抢到一些更好的人才、更优质的人才所设置的，所以提前批一般都是神仙打架（竞争极其激烈，各种本硕985人才），但是**提前批是非常重要的**。因为已经有越来越多的公司看中提前批，有不少公司都是给予二次投递机会的，也就是说如果提前批你挂了，正式批还是能够再次投递这个公司的。

比如说，我想投递字节跳动的抖音旗下的算法工程师岗位，那么我可以在6月份的时候投递一份简历，如果能通过面试更好。即使不能顺利通过面试，那么在7-10月份秋招正式批的时候我还可以再投递一次算法工程师这个岗位，也就是说提前批给了你一次复活机会。能够有一个“复活甲”，岂不美滋滋儿。所以同学们千万要参加提前批，千万不要因为觉得自己学历不好或者水平不够就放弃了提前批，千万不要，切记切记！

换一个角度来说，即使你是普通学校的学生没有什么很厉害的背景或者经历而被直接刷下来的话，你的简历也不会被直接弃之不用，而是会放在正式批里对你发起面试。说人话就是，你相当于是最早投递正式批的一批人了，比别人能更早被发起面试邀约。

- **正式批：**正式批是紧跟在提前批后的，一般提前批结束后马上就会进入正式批环节了。这是整个招聘环节HC最多的时候，建议尽量早一点投递，因为岗位空缺就是那么多，招够一定人数后就不再招了，千万不要想着等自己完全准备好了，万事俱备那种再去投递简历，最好是在提前批就进行简历的投递，因为很有可能出现即使顺利通过面试但坑位不足的情况，导致offer无法顺利审批下来。因此一定要尽早投递，早早地保住一个萝卜坑。

还有一点需要注意的是就是尽量多拿几家公司的意向书（意向书是相当于正式offer前的一种保障），因为只有你多拿几个意向书，在十月底的HR谈薪资期间你才有更多的底气和筹码拿到更高的薪资和更好的待遇。

我出来打工，不图钱？我图什么啊？要钱，不丢人！



除了这个原因之外还有就是薪资水平并不是一定的，前几届某家公司给的高并不代表今年会继续给的高，前几届某家公司给的低并不代表今年会继续给的低，比如 2020 年秋招中的快手、美团、虾皮涨薪幅度很大，美团直接涨薪1/3，以前 18K 的月薪直接暴涨到 27Kemmm，真香！所以建议不要死盯着一个公司，多尝试几家公司，到时候选择也更多一些。

秋招补录：秋招补录一般是在每年的10月末 - 11月份。主要因为在某些大佬或者因为薪资或者地域等原因拒掉手中的offer后，某些岗位出现了没招满的情况下，这些岗位又重新开始招聘了。

一般来说补录的名额是相对较少一些的，能够在提前批、秋招时期上岸就尽量上岸，不要把宝压在秋招补录甚至是来年的春招时期。在补录环节，多多关注一下招聘网站上的补录信息以及意向公司的官方公众号，最新的补录信息都会在一些招聘网站比如牛客、实习僧和相关公众号放出来的。

来年春招：来年的春招一般是在第二年的3-5月份，相较于秋招规模，春招的招聘规模要小很多，主要是因为公司企业在整个秋招过程中没有招到足够的人数进行的一次补招，这也是应届毕业生最后一次找到工作的校招机会了。春招时期的竞争压力比秋招时要更大一些，主要原因如下：

- (一) **水平更强**：对于一些没有在秋招环节拿到意向offer的同学来说经历了一整个秋招的历练，整体水平上升了一个台阶，也就是说跟你同期竞争的人更厉害了，整体水平要更高一些的。
- (二) **人数更多**：一些考研失利的大四学生或者考博失利的研三学生也投入了找工作的大军中，人数更多了，竞争激烈程度更大了。
- (三) **名额更少**：春招时期的岗位比秋招时期要少上不少，某些岗位招够人了就不会在春招时期开放该岗位的投递链接了，还有就是岗位HC也更少一些，如果说秋招时期是3-5个人竞争一个岗位，春招时期可能就是8-10个人竞争一个岗位了。

贴心建议：虽然很多公司说提前批挂了也不影响正式批的投递，但不可能说是完全没有影响的。对于一些大型的招聘公司来说都是有自己的一套招聘系统的，你每一次的面试都会记录在案，也就是说如果你提前批、正式批都投递了这家公司的这个岗位，那么在正式批的时候面试官是能够看到你前一次提前批面试的面试结果的，包括前一个面试官给你打的面试评价和你的面试情况。

即使很多公司都说第二次面试不会受第一次面试结果的印象，emmm，你要信你就信，反正我不信哈哈

如果你前一次面试评价极差，你觉得会没有影响吗？

因此并不建议盲目的去投递提前批，如果你提前批面试结果很差，在正式批的时候很可能会影响你的面试。

不要以一个纯小白的身份去投递这些大公司，因为大厂面试机会来之不易，不要直接拿大厂练手，最好将这些大厂放在个人秋招安排进程的中期，这个时候你已经有了一定的面试经验了，成功率也更高一些。

2.2.2、确定工作方向

一般来说，越大的公司分工也就越明确，更加注重技术的深度；一些小一点的公司要求你是个多面手，前端后端测试一把抓，对于技术深度要求低一些，更加注重技术的广度。

因此也更加建议同学们第一份工作找一家大一点的互联网公司，精进一下个人的技术，也能够镀金。尽早的确定个人的方向能够节省很多时间，避免像无头苍蝇一样到处乱撞。

目前互联网主要有**算法、前端、后端、客户端、测试**等几个大方向，内存程度**算法>后端>前端>=客户端>测试**。

这个排名是个人2021年秋招时期所见所感，不喜勿喷~

算法岗位对于学历和论文要求很大，如果你没有很好的学历加成以及论文和相关比赛加持一般不建议去投递算法岗位。

前后端岗位前景较好，需求大，相应的竞争也比较大一些，前后端在大公司小公司都有相应的岗位，后端语言上主要是Java、C++，以及最近几年比较火的 Python、Go语言都可以；前端使用 JS 较多，包括VUE、React等常用框架。

客户端又分为安卓客户端和IOS客户端，客户端主要是一些大中厂需求比较多，就业面相对于前后端是要窄一些的，但是客户端需求较多，对比大家一窝蜂的涌向前后端方向来说，客户端算是竞争小一些的岗位了，不少大公司甚至达到了自掏腰包求客户端简历的程度了。

测试入门算是最简单的一个，对于代码能力要求也没有其他岗位那么强，对于一些代码能力较弱的同学可以尝试测试岗位

2.3、C++能投哪些岗位

嵌入式研发岗位

嵌入式方向可能比较偏向于硬件一些，比如国内的华为、中兴、小米、紫光展锐这些公司都是招嵌入式开发的。

嵌入式开发由于涉及硬件比较多，所以对于通信、电信、自动化这些偏硬件的专业会友好一些，嵌入式开发一般会涉及到一些网络编程、Socket通信之类的，还有一些会涉及到并发编程等。

主要的业务方向是物联网以及芯片等方向，国家也是大力发展战略芯片方向，所以嵌入式也是一个不错的职业方向。

后端/服务器研发

这是C++方向的最大缺口之一了，同样也是竞争最为激烈的岗位之一，后端研发要求掌握了解的知识技能也是非常多的。除了要有比较扎实的C++语言基础，还要求你具有多线程编程、跨平台编码等知识，也需要你有一定的算法能力和了解常见的数据结构等。还要一些常用数据库，比如关系型数据库MySQL和内存型数据库Redis、Memcached你要了解一些的。

对于社招选手来说，还会要求他们具有一些中间件的使用，包括微服务等，但是对于校招更看中扎实的基础和潜力，微服务和中间件，有则加分，没有也无伤大雅。

还有就是计算机四大基础：操作系统，计算机网络，计算机组成原理以及编译原理了，这些基础才能保障你在前进的路上走的更远更稳，在实际的后端开发过程中涉及到的东西很多，从网络通信到性能优化再到系统总体架构，都需要你具有很扎实的计算机基础。

游戏研发岗位

不少游戏的引擎和服务器都是基于C++研发的，要是论业务范围广度，C++远远比不上Java，可你要是比性能高、速度快，Java就得给C++让道了。

C++游戏研发比较吃经验，但是国内游戏大厂就那么几家：网易游戏、腾讯游戏、米哈游、巨人网络等，所以游戏研发岗位竞争也比较激烈，坑位没有后端那么多，你在跳槽时的可选择面就要窄一些。

最重要的是游戏研发是需要有一定兴趣的，因为就业面比较窄想转行就不是那么容易了。从事游戏研发除了要求你具有C++基础之外，像一些基本的图形学理论也是必须了解的，Unity3D等引擎也是必问的。

多媒体研发岗位

近年来短视频和直播行业的崛起有一部分原因要归功于多媒体研发比如音视频等，一些播放器直播平台和一些特效的实现都离不开C++开发。

但多媒体研发所需的不仅仅只是语言这一门，你可能还需要了解一些图像视频的采集、音视频的加工、编码等，常见的可能需要你了解OpenCV、ffmpeg、x264等协议，还有就是一些基于音视频传输协议等，具体还要看岗位的详细要求。

客户端研发岗位

这里主要说的是ios客户端，ios研发主要使用Objective-C/Swift开发，但是现在鉴于客户端研发岗位比较缺人，已经把门槛放宽到C/C++方向。毕竟现在是超级APP的时代，比如淘宝、支付宝、抖音等，微信小程序把很多客户端的市场份额抢占了。

客户端相较于后端，就业选择要窄一些，特别是对于一些二三线的互联网公司来说，可能根本就没有客户端开发的岗位，这是一个很大的弊端，但是好处就是竞争要比后端小上不少，而且需求也不小，建议大家自己好好思考，结合个人兴趣以及未来规划等加以把握。

3、知识储备

双非学历、字节全栈、互联网一线卑微打工仔，欢迎关注个人公众号。
关注那些曾经像我一样的小白新手程序员们，我踩过的坑不希望你再踩，我走过的路希望你能照着走下来。



3.1、C++

1、在main执行之前和之后执行的代码可能是什么？

main函数执行之前，主要就是初始化系统相关资源：

- 设置栈指针
- 初始化静态 static 变量和 global 全局变量，即 .data 段的内容
- 将未初始化部分的全局变量赋初值：数值型 short, int, long 等为 0, bool 为 FALSE, 指针为 NULL 等等，即 .bss 段的内容
- 全局对象初始化，在 main 之前调用构造函数，这是可能会执行前的一些代码
- 将 main 函数的参数 argc, argv 等传递给 main 函数，然后才真正运行 main 函数
- `__attribute__((constructor))`

main函数执行之后：

- 全局对象的析构函数会在 main 函数之后执行；
- 可以用 `atexit` 注册一个函数，它会在 main 之后执行；
- `__attribute__((destructor))`

update1:<https://github.com/forthespada/InterviewGuide/issues/2>, 由 stanleyguo0207 提出 - 2021.03.22

2、结构体内存对齐问题？

- 结构体内成员按照声明顺序存储，第一个成员地址和整个结构体地址相同。
- 未特殊说明时，按结构体中 size 最大的成员对齐（若有 double 成员，按 8 字节对齐。）

c++11 以后引入两个关键字 `alignas` 与 `alignof`。其中 `alignof` 可以计算出类型的对齐方式，`alignas` 可以指定结构体的对齐方式。

但是 `alignas` 在某些情况下是不能使用的，具体见下面的例子：

```
1 // alignas 生效的情况
2
3 struct Info {
4     uint8_t a;
5     uint16_t b;
6     uint8_t c;
7 };
8
9 std::cout << sizeof(Info) << std::endl;    // 6  2 + 2 + 2
10 std::cout << alignof(Info) << std::endl;   // 2
11
12 struct alignas(4) Info2 {
13     uint8_t a;
```

```

14     uint16_t b;
15     uint8_t c;
16 };
17
18 std::cout << sizeof(Info2) << std::endl; // 8 4 + 4
19 std::cout << alignof(Info2) << std::endl; // 4

```

`alignas` 将内存对齐调整为4个字节。所以 `sizeof(Info2)` 的值变为了8。

```

1 // alignas 失效的情况
2
3 struct Info {
4     uint8_t a;
5     uint32_t b;
6     uint8_t c;
7 };
8
9 std::cout << sizeof(Info) << std::endl; // 12 4 + 4 + 4
10 std::cout << alignof(Info) << std::endl; // 4
11
12 struct alignas(2) Info2 {
13     uint8_t a;
14     uint32_t b;
15     uint8_t c;
16 };
17
18 std::cout << sizeof(Info2) << std::endl; // 12 4 + 4 + 4
19 std::cout << alignof(Info2) << std::endl; // 4

```

若 `alignas` 小于自然对齐的最小单位，则被忽略。

- 如果想使用单字节对齐的方式，使用 `alignas` 是无效的。应该使用 `#pragma pack(push,1)` 或者使用 `__attribute__((packed))`。

```

1 #if defined(__GNUC__) || defined(__GNUG__)
2     #define ONEBYTE_ALIGN __attribute__((packed))
3 #elif defined(_MSC_VER)
4     #define ONEBYTE_ALIGN
5     #pragma pack(push,1)
6 #endif
7
8 struct Info {
9     uint8_t a;
10    uint32_t b;
11    uint8_t c;
12 } ONEBYTE_ALIGN;
13
14 #if defined(__GNUC__) || defined(__GNUG__)
15     #undef ONEBYTE_ALIGN
16 #elif defined(_MSC_VER)
17     #pragma pack(pop)
18     #undef ONEBYTE_ALIGN
19 #endif
20
21 std::cout << sizeof(Info) << std::endl; // 6 1 + 4 + 1
22 std::cout << alignof(Info) << std::endl; // 6

```

- 确定结构体中每个元素大小可以通过下面这种方法:

```

1 #if defined(__GNUC__) || defined(__GNUG__)
2     #define ONEBYTE_ALIGN __attribute__((packed))
3 #elif defined(_MSC_VER)
4     #define ONEBYTE_ALIGN
5     #pragma pack(push,1)
6 #endif
7
8 /**
9 *
10 * 0 1   3       6   8 9           15
11 * +---+-----+---+---+
12 * | |   |       |   | |
13 * |a| b | c | d |e|     pad   |
14 * | |   |       |   | |
15 * +---+-----+---+---+
16 *
17 */
18 struct Info {
19     uint16_t a : 1;
20     uint16_t b : 2;
21     uint16_t c : 3;
22     uint16_t d : 2;
23     uint16_t e : 1;
24     uint16_t pad : 7;
25 } ONEBYTE_ALIGN;
26
27 #if defined(__GNUC__) || defined(__GNUG__)
28     #undef ONEBYTE_ALIGN
29 #elif defined(_MSC_VER)
30     #pragma pack(pop)
31     #undef ONEBYTE_ALIGN
32 #endif
33
34 std::cout << sizeof(Info) << std::endl; // 2
35 std::cout << alignof(Info) << std::endl; // 1

```

这种处理方式是 `alignas` 处理不了的。

update1:<https://github.com/forthespada/InterviewGuide/issues/2>,由 stanleyguo0207 提出 -
2021.03.22

3、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）。

- 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
- 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
- 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

参考代码：

```

1 void test(int *p)
2 {
3     int a=1;
4     p=&a;
5     cout<<p<<" "<<*p<<endl;
6 }
7
8 int main(void)
9 {
10     int *p=NULL;
11     test(p);
12     if(p==NULL)
13         cout<<"指针p为NULL"<<endl;
14     return 0;
15 }
16 //运行结果为:
17 //0x22ff44 1
18 //指针p为NULL
19
20
21
22 void testPTR(int* p) {
23     int a = 12;
24     p = &a;
25 }
26
27
28 void testREFF(int& p) {
29     int a = 12;
30     p = a;
31 }
32
33 void main()
34 {
35     int a = 10;
36     int* b = &a;
37     testPTR(b);//改变指针指向，但是没改变指针的所指的内容
38     cout << a << endl;// 10
39     cout << *b << endl;// 10
40
41     a = 10;
42     testREFF(a);
43     cout << a << endl;//12
44 }
```

在编译器看来, int a = 10; int &b = a; 等价于 int * const b = &a; 而 b = 20; 等价于 *b = 20; 自动转换为指针和自动解引用.

4、堆和栈的区别

- 申请方式不同。
 - 栈由系统自动分配。
- 堆是自己申请和释放的。
- 申请大小限制不同。
 - 栈顶和栈底是之前预设好的，栈是向栈底扩展，大小固定，可以通过ulimit -a查看，由ulimit -s修改。
 - 堆向高地址扩展，是不连续的内存区域，大小可以灵活调整。
- 申请效率不同。
 - 栈由系统分配，速度快，不会有碎片。
 - 堆由程序员分配，速度慢，且会有碎片。

栈空间默认是4M, 堆区一般是 1G - 4G

	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。

形象的比喻

栈就像我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

《C++中堆 (heap) 和栈(stack)的区别》：https://blog.csdn.net/qq_34175893/article/details/83502412

5、区别以下指针类型？

```
1 int *p[10]
2 int (*p)[10]
3 int *p(int)
4 int (*p)(int)
```

- int *p[10]表示指针数组，强调数组概念，是一个数组变量，数组大小为10，数组内每个元素都是指向int类型的指针变量。
- int (*p)[10]表示数组指针，强调是指针，只有一个变量，是指针类型，不过指向的是一个int类型的数组，这个数组大小是10。
- int *p(int)是函数声明，函数名是p，参数是int类型的，返回值是int *类型的。
- int (*p)(int)是函数指针，强调是指针，该指针指向的函数具有int类型参数，并且返回值是int类型的。

6、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。

《虚函数表存放在哪里》：<https://blog.csdn.net/u013270326/article/details/82830656>

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。

7、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：

```
1 | int *p = new float[2]; //编译错误
2 | 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

8、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，分配足够大的原始未类型化的内存，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

9、malloc和new的区别？

- malloc和free是标准库函数，支持覆盖；new和delete是运算符，并且支持重载。
- malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险；new和delete除了分配回收功能外，还会调用构造函数和析构函数。
- malloc和free返回的是void类型指针（必须进行类型转换），new和delete返回的是具体类型指针。

9.1、delete和delete[]区别？(补充)

- delete只会调用一次析构函数。
- delete[]会调用数组中每个元素的析构函数。

10、宏定义和函数有何区别？

- 宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
- 宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。
- 宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
- 宏定义不要在最后加分号。

11、宏定义和typedef区别？

- 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
- 宏替换发生在编译阶段之前，属于文本插入替换；typedef是编译的一部分。
- 宏不检查类型；typedef会检查数据类型。
- 宏不是语句，不在最后加分号；typedef是语句，要加分号标识结束。
- 注意对指针的操作，`typedef char * p_char`和`#define p_char char *`区别巨大。

12、变量声明和定义区别？

- 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间；定义要在定义的地方为其分配存储空间。
- 相同变量可以在多处声明（外部变量extern），但只能在一处定义。

13、哪几种情况必须用到初始化成员列表？

- 初始化一个const成员。
- 初始化一个reference成员。
- 调用一个基类的构造函数，而该函数有一组参数。
- 调用一个数据成员对象的构造函数，而该函数有一组参数。

14、strlen和sizeof区别？

- sizeof是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen是字符处理的库函数。
- sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化）；strlen的参数只能是字符指针且结尾是'\0'的字符串。
- 因为sizeof值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小。

```
1 int main(int argc, char const *argv[]){
2
3     const char* str = "name";
4
5     sizeof(str); // 取的是指针str的长度，是8
6     strlen(str); // 取的是这个字符串的长度，不包含结尾的 \0。大小是4
7
8     return 0;
}
```

15、常量指针和指针常量区别？

- 指针常量是一个指针，读成常量的指针，指向一个只读变量。如int const *p或const int *p。
- 常量指针是一个不能被改变指向的指针。指针是个常量，不能中途改变指向，如int *const p。

update1:<https://www.nowcoder.com/discuss/597948>，网友“牛客191489444号”指出笔误，感谢！

16、a和&a有什么区别？

```
1 假设数组int a[10];
2 int (*p)[10] = &a;
3
```

- 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），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。

17、数组名和指针（这里为指向数组首元素的指针）区别？

- 二者均可通过增减偏移量来访问数组中的元素。
- 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
- 当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。

18、野指针和悬空指针

都是是指向无效内存区域(这里的无效指的是"不安全不可控")的指针，访问行为将会导致未定义行为。

- 野指针

野指针，指的是没有被初始化过的指针

```

1 int main(void) {
2
3     int* p;      // 未初始化
4     std::cout<< *p << std::endl; // 未初始化就被使用
5
6     return 0;
7 }
8

```

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。

```

1 int main(void) {
2     int * p = nullptr;
3
4     int* p2 = new int;
5
6     p = p2;
7
8     delete p2;
9 }
10

```

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，编译器会直接报错。

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。

19、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。

20、C和C++的区别

- C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。
- 标准C++中的字符串类取代了标准C函数库头文件中的字符数组处理函数（C中没有字符串类型）。
- C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。
- C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。
- 在C++中，允许有相同的函数名，不过它们的参数类型不能完全相同，这样这些函数就可以相互区别开来。而这在C语言中是不允许的。也就是C++可以重载，C语言不允许。
- C++语言中，允许变量定义语句在程序中的任何地方，只要是在使用它之前就可以；而C语言中，必须要在函数开头部分。而且C++允许重复定义变量，C语言也是做不到这一点的
- 在C++中，除了值和指针之外，新增了引用。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。
- C++相对与C增加了一些关键字，如：bool、using、dynamic_cast、namespace等等

《[C语言与C++有什么区别？](https://www.cnblogs.com/lTziyuan/p/9487760.html)》 <https://www.cnblogs.com/lTziyuan/p/9487760.html>

21、C++与Java的区别

语言特性

- Java语言给开发人员提供了更为简洁的语法；完全面向对象，由于JVM可以安装到任何的操作系统上，所以说它的可移植性强
- Java语言中没有指针的概念，引入了真正的数组。不同于C++中利用指针实现的“伪数组”，Java引入了真正的数组，同时将容易造成麻烦的指针从语言中去掉，这将有利于防止在C++程序中常见的因为数组操作越界等指针操作而对系统数据进行非法读写带来的不安全问题

- C++也可以在其他系统运行，但是需要不同的编码（这一点不如Java，只编写一次代码，到处运行），例如对一个数字，在Windows下是大端存储，在Unix中则为小端存储。Java程序一般都是生成字节码，在JVM里面运行得到结果
- Java用接口(Interface)技术取代C++程序中的多继承性。接口与多继承有同样的功能，但是省却了多继承在实现和维护上的复杂性

垃圾回收

- C++用析构函数回收垃圾，写C和C++程序时一定要注意内存的申请和释放
- Java语言不使用指针，内存的分配和回收都是自动进行的，程序员无须考虑内存碎片的问题

应用场景

- Java在桌面程序上不如C++实用，C++可以直接编译成exe文件，指针是C++的优势，可以直接对内存的操作，但同时具有危险性。（操作内存的确是一项非常危险的事情，一旦指针指向的位置发生错误，或者误删除了内存中某个地址单元存放的重要数据，后果是可想而知的）
- Java在Web应用上具有C++无可比拟的优势，具有丰富多样的框架
- 对于底层程序的编程以及控制方面的编程，C++很灵活，因为有句柄的存在

《C++和Java的区别和联系》：<https://www.cnblogs.com/tanrong/p/8503202.html>

22、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

《struct结构在C和C++中的区别》：https://blog.csdn.net/mm_hh/article/details/70456240

23、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，在内存中会产生多份相同的备份。`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。

24、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

```

1 // i 可以是 int 型或者 const int 型
2 void fun(const int& i){
3     //...
4 }
5

```

- 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`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

25、C++的顶层`const`和底层`const`

概念区分

- **顶层const**: 指的是const修饰的变量**本身**是一个常量，无法修改，指的是指针，就是`*`号的右边
- **底层const**: 指的是const修饰的变量**所指向的对象**是一个常量，指的是所指变量，就是`*`号的左边

举个例子

```
1 int a = 10;
2 int* const b1 = &a;           //顶层const, b1本身是一个常量
3 const int* b2 = &a;           //底层const, b2本身可变, 所指的对象是常量
4 const int b3 = 20;            //顶层const, b3是常量不可变
5 const int* const b4 = &a;      //前一个const为底层, 后一个为顶层, b4不可变
6 const int& b5 = a;           //用于声明引用变量, 都是底层const
7
8
9
```

区分作用

- 执行对象拷贝时有限制，常量的底层const不能赋值给非常量的底层const
- 使用命名的强制类型转换函数`const_cast`时，只能改变运算对象的底层const

《C++ 顶层const与底层const总结》：<https://www.jianshu.com/p/fbbcf11100f6>

《C++的顶层const和底层const浅析》：https://blog.csdn.net/qq_37059483/article/details/78811231

```
1 const int a;
2 int const a;
3 const int *a;
4 int *const a;
5
```

- `int const a`和`const int a`均表示定义常量类型a。
- `const int *a`, 其中a为指向int型变量的指针, `const`在`*`左侧, 表示a指向不可变常量。(看成`const (*a)`, 对引用加`const`)
- `int *const a`, 依旧是是指针类型, 表示a为指向整型数据的常指针。(看成`const(a)`, 对指针`const`)

26、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.

27、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 {
7     void foo(); //OK
8     virtual void foo(); // OK
9     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 {
7     virtual void f00(); //OK, 这个函数是B新增的, 不是继承的
8     virtual void foo() override; //Error, 加了override之后, 这个函数一定是继承自
9     A的, A找不到就报错
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。例子如下：

```

1 class Base
2 {
3     virtual void foo();
4 };
5
6 class A : public Base
7 {
8     void foo() final; // foo 被override并且是最后一个override, 在其子类中不可以重
9     写
10}
11
12 class B final : A // 指明B是不可以被继承的
13 {
14     void foo() override; // Error: 在A中已经被final了
15 }
16
17 class C : B // Error: B is final
18 {

```

```
18 };  
19  
20  
21
```

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

28、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  
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  
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  
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/qg936836/article/details/83450218>

29、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：

```
1 class A{  
2 public:  
3     int num1;  
4     int num2;  
5 public:  
6     A(int a=0, int b=0):num1(a),num2(b){};  
7     A(const A& a){};  
8     //重载 = 号操作符函数  
9     A& operator=(const A& a){  
10         num1 = a.num1 + 1;  
11         num2 = a.num2 + 1;  
12         return *this;  
13     };  
14 };  
15 int main(){  
16     A a(1,1);
```

```
17     A a1 = a; //拷贝初始化操作，调用拷贝构造函数
18     A b;
19     b = a; //赋值操作，对象a中，num1 = 1, num2 = 1; 对象b中，num1 = 2, num2 = 2
20     return 0;
21 }
22 }
```

30、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++；

哪些情况下使用extern "C"：

- (1) C++代码中调用C语言代码；
- (2) 在C++中的头文件中使用；
- (3) 在多人协同开发时，可能有人擅长C语言，而有人擅长C++；

举个例子，C++中调用C代码：

```
1 #ifndef __MY_HANDLE_H__
2 #define __MY_HANDLE_H__
3
4 extern "C"{
5     typedef unsigned int result_t;
6     typedef void* my_handle_t;
7
8     my_handle_t create_handle(const char* name);
9     result_t operate_on_handle(my_handle_t handle);
10    void close_handle(my_handle_t handle);
11 }
12
13
14
```

- 参考的blog中有一篇google code上的文章，专门写extern "C"的，有兴趣的读者不妨去看看
《extern "C"的功能和用法研究》：https://blog.csdn.net/sss_369/article/details/84060561

综上，总结出使用方法，在C语言的头文件中，对其外部函数只能指定为extern类型，C语言中不支持extern "C"声明，在.c文件中包含了extern "C"时会出现编译语法错误。所以使用extern "C"全部都放在于cpp程序相关文件或其头文件中。

总结出如下形式：

- (1) C++调用C函数：

```
1 //xx.h
2 extern int add(...)

3
4 //xx.c
5 int add(){
6
7 }
```

```
9 //xx.cpp
10 extern "C" {
11     #include "xx.h"
12 }
13
14
15
```

(2) C调用C++函数

```
1 //xx.h
2 extern "C"{
3     int add();
4 }
5 //xx.cpp
6 int add(){
7
8 }
9 //xx.c
10 extern int add();
11
12
13
```

31、模板函数和模板类的特例化

引入原因

编写单一的模板，它能适应多种类型的需求，使每种类型都具有相同的功能，但对于某种特定类型，如果要实现其特有的功能，单一模板就无法做到，这时就需要模板特例化

定义

对单一模板提供的一个特殊实例，它将一个或多个模板参数绑定到特定的类型或值上

(1) 模板函数特例化

必须为原函数模板的每个模板参数都提供实参，且使用关键字template后跟一个空尖括号对<>，表明将原模板的所有模板参数提供实参，举例如下：

```
1 template<typename T> //模板函数
2 int compare(const T &v1,const T &v2)
3 {
4     if(v1 > v2) return -1;
5     if(v2 > v1) return 1;
6     return 0;
7 }
8 //模板特例化,满足针对字符串特定的比较,要提供所有实参,这里只有一个T
9 template<>
10 int compare(const char* const &v1,const char* const &v2)
11 {
12     return strcmp(p1,p2);
13 }
14
15
16
```

本质

特例化的本质是实例化一个模板，而非重载它。特例化不影响参数匹配。参数匹配都以最佳匹配为原则。例如，此处如果是compare(3,5)，则调用普通的模板，若为compare("hi","haha")则调用**特例化版本**（因为这个const char*相对于T，更匹配实参类型），注意二者函数体的语句不一样了，实现不同功能。

注意

模板及其特例化版本应该声明在同一个头文件中，且所有同名模板的声明应该放在前面，后面放特例化版本。

(2) 类模板特例化

原理类似函数模板，**不过在类中，我们可以对模板进行特例化，也可以对类进行部分特例化**。对类进行特例化时，仍然用template<>表示是一个特例化版本，例如：

```
1 template<>
2 class hash<sales_data>
3 {
4     size_t operator()(sales_data& s);
5     //里面所有T都换成特例化类型版本sales_data
6     //按照最佳匹配原则，若T != sales_data，就用普通类模板，否则，就使用含有特定功能的特
7     //例化版本。
8 };
9
10
```

类模板的部分特例化

不必为所有模板参数提供实参，可以**指定一部分而非所有模板参数**，一个类模板的部分特例化本身仍是一个模板，使用它时还必须为其特例化版本中未指定的模板参数提供实参(特例化时类名一定要和原来的模板相同，只是参数类型不同，按最佳匹配原则，哪个最匹配，就用相应的模板)

特例化类中的部分成员

可以特例化类中的部分成员函数而不是整个类，举个例子：

```
1 template<typename T>
2 class Foo
3 {
4     void Bar();
5     void Barst(T a)();
6 };
7
8 template<>
9 void Foo<int>::Bar()
10 {
11     //进行int类型的特例化处理
12     cout << "我是int型特例化" << endl;
13 }
14
15 Foo<string> fs;
16 Foo<int> fi;//使用特例化
17 fs.Bar(); //使用的是普通模板，即Foo<string>::Bar()
18 fi.Bar(); //特例化版本，执行Foo<int>::Bar()
```

```
19 //Foo<string>::Bar()和Foo<int>::Bar()功能不同  
20  
21  
22
```

《类和函数模板特例化》：<https://blog.csdn.net/wang664626482/article/details/52372789>

32、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

```
1 #include <stdio.h>  
2  
3 int main()  
4 {  
5     printf("整型输出: %d\n", 10);  
6     printf("浮点型输出: %f\n", 10);  
7     return 0;  
8 }
```

整型输出：10
浮点型输出：0.000000

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。

(2) 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

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int i=5;
7     void* pInt=&i;
8     double d=*(double*)pInt;
9     cout<<"转换后输出: "<< d << endl;
10 }
11 }
```

转换后输出: 1.78416e-307

例2：不同类型指针之间转换

```

1 #include<iostream>
2 using namespace std;
3
4 class Parent{};
5 class Child1 : public Parent
6 {
7 public:
8     int i;
9     Child1(int e):i(e){}
10 };
11 class Child2 : public Parent
12 {
13 public:
14     double d;
15     Child2(double e):d(e){}
16 };
17 int main()
18 {
19     Child1 c1(5);
20     Child2 c2(4.1);
21     Parent* pp;
22     Child1* pc1;
23
24     pp=&c1;
25     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
26     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 }
```

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。

33、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
4 class Parent{
5 public:
6     Parent(){
7         cout << "Parent construct function" << endl;
8     };
9     ~Parent(){
10        cout << "Parent destructor function" << endl;
11    }
12};
13
14 class Son : public Parent{
15 public:
16     Son(){
17         cout << "Son construct function" << endl;
18     };
19     ~Son(){
20         cout << "Son destructor function" << endl;
21     }
22};
23
24 int main()
25 {
26     Parent* p = new Son();
27     delete p;
28     p = NULL;
29     return 0;
30 }
31 //运行结果:
32 //Parent construct function
33 //Son construct function
34 //Parent destructor function
35
36
37
```

```

1 #include <iostream>
2 using namespace std;
3
4 class Parent{
5 public:
6     Parent(){
7         cout << "Parent construct function" << endl;
8     };
9     virtual ~Parent(){
10        cout << "Parent destructor function" << endl;
11    }
12};
13
14 class Son : public Parent{
15 public:
16     Son(){
17         cout << "Son construct function" << endl;
18     };
19     ~Son(){
20         cout << "Son destructor function" << endl;
21     }
22};
23
24 int main()
25 {
26     Parent* p = new Son();
27     delete p;
28     p = NULL;
29     return 0;
30 }
31 //运行结果:
32 //Parent construct function
33 //Son construct function
34 //Son destructor function
35 //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。

update1:<https://github.com/forthespada/InterviewGuide/issues/2> ,由 stanleyguo0207 提出 -
2021.03.22

34、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。

update1:<https://github.com/forthespada/InterviewGuide/issues/2>,由 stanleyguo0207 提出 -
2021.03.22

35、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     ...  
3     virtual int fun();  
4     void fun(int);  
5     void fun(double, double);  
6     static int fun(char);  
7     ...  
8 }  
9  
10  
11
```

(2) 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 }  
6 //子类  
7 class B : public A{  
8 public:  
9     //重写,一般加override可以确保是重写父类的函数  
10    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

(3) 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 };
8 //子类
9 class B : public A{
10 public:
11     //隐藏父类的fun函数
12     void fun(int a){
13         cout << "B中的fun函数" << endl;
14     }
15 };
16 int main(){
17     B b;
18     b.fun(2); //调用的是B中的fun函数
19     b.A::fun(2); //调用A中fun函数
20     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：

```
1 //父类
2 class A{
3 public:
4     virtual void fun(int a){
5         cout << "A中的fun函数" << endl;
6     }
7 };
8 //子类
9 class B : public A{
10 public:
11     //隐藏父类的fun函数
12     virtual void fun(char* a){
13         cout << "B中的fun函数" << endl;
14     }
15 };
```

```
16 int main(){
17     B b;
18     b.fun(2); //报错，调用的是B中的fun函数，参数类型不对
19     b.A::fun(2); //调用A中fun函数
20     return 0;
21 }
22
23
24
```

36、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。

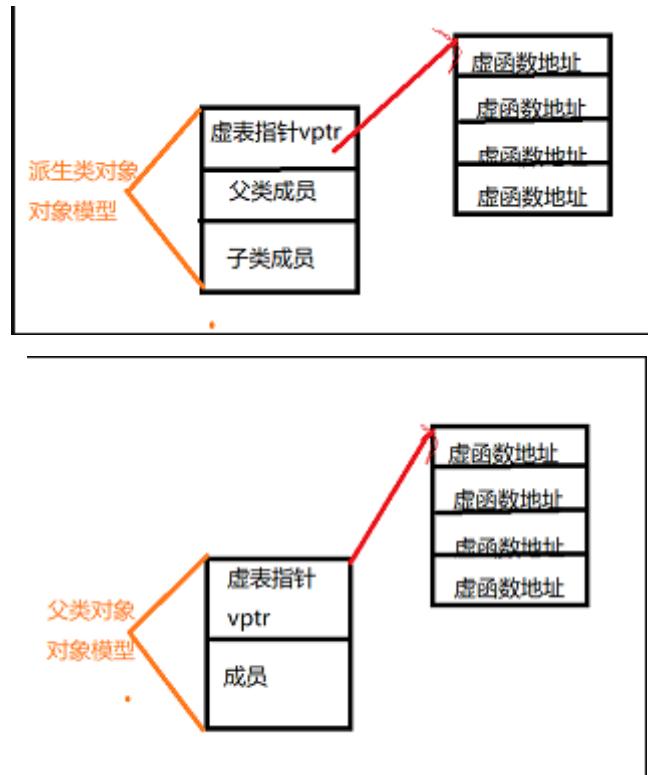
举个例子：

```
1 #include <iostream>
2 using namespace std;
3
4 class Base{
5 public:
6     virtual void fun(){
7         cout << " Base::fun()" << endl;
8     }
9 };
10
11 class Son1 : public Base{
12 public:
13     virtual void fun() override{
14         cout << " Son1::fun()" << endl;
15     }
16 };
17
18 class Son2 : public Base{
19 };
20
21
22 int main()
23 {
24     Base* base = new Son1;
25     base->fun();
26     base = new Son2;
27     base->fun();
28     delete base;
29     base = NULL;
30     return 0;
31 }
32 // 运行结果
33 // Son1::fun()
34 // Base::fun()
```

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。

《C++实现多态的原理》：https://blog.csdn.net/qq_37954088/article/details/79947898

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
4 class Student{
5 public:
6     Student()://默认构造函数，没有参数
7         this->age = 20;
8         this->num = 1000;
9     };
10    Student(int a, int n):age(a), num(n){};//初始化构造函数，有参数和参数列表
11    Student(const Student& s){//拷贝构造函数，这里与编译器生成的一致
12        this->age = s.age;
13        this->num = s.num;
14    };
15    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
16        this->age = r;
17        this->num = 1002;
18    };
19    ~Student(){}
20 public:
21     int age;
22     int num;
23 };
24
25 int main(){
26     Student s1;
27     Student s2(18,1001);
28     int a = 10;
29     Student s3(a);
30     Student s4(s3);
31
32     printf("s1 age:%d, num:%d\n", s1.age, s1.num);
33     printf("s2 age:%d, num:%d\n", s2.age, s2.num);
34     printf("s3 age:%d, num:%d\n", s3.age, s3.num);
35     printf("s4 age:%d, num:%d\n", s4.age, s4.num);
36     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 //s4 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>

38、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。

```

1 #include <iostream>
2 #include <string.h>
3 using namespace std;
4
5 class Student
6 {
7 private:
8     int num;
9     char *name;
10 public:
11     Student(){
12         name = new char(20);
13         cout << "Student" << endl;
14     };
15     ~Student(){
16         cout << "~Student" << &name << endl;
17         delete name;
18         name = NULL;
19     };
20     Student(const Student &s){//拷贝构造函数
21         //浅拷贝，当对象的name和传入对象的name指向相同的地址
22         name = s.name;
23         //深拷贝
24         //name = new char(20);
25         //memcpy(name, s.name, strlen(s.name));
26         cout << "copy Student" << endl;
27     };
28 };
29
30 int main()
31 {
32     {//花括号让s1和s2变成局部对象，方便测试
33     Student s1;
34     Student s2(s1);// 复制对象
35     }
36     system("pause");
37     return 0;

```

```

38 }
39 //浅拷贝执行结果:
40 //Student
41 //copy Student
42 //~Student 0x7ffffed0c3ec0
43 //~Student 0x7ffffed0c3ed0
44 //*** Error in `/tmp/815453382/a.out': double free or corruption (fasttop):
45 //0x0000000001c82c20 ***
46
47 //深拷贝执行结果:
48 //Student
49 //copy Student
50 //~Student 0x7ffffebca9fb0
51 //~Student 0x7ffffebca9fc0
52
53

```

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。

《C++面试题之浅拷贝和深拷贝的区别》：<https://blog.csdn.net/caoshangpa/article/details/79226270>

39、内联函数和宏定义的区别

内联(inline)函数和普通函数相比可以加快程序运行的速度，因为不需要中断调用，在编译的时候内联函数可以直接嵌入到目标代码中。

内联函数适用场景

- 使用宏定义的地方都可以使用inline函数
- 作为类成员接口函数来读写类的私有成员或者保护成员，会提高效率

为什么不能把所有的函数写成内联函数

内联函数以代码复杂为代价，它以省去函数调用的开销来提高执行效率。所以一方面如果内联函数体内代码执行时间相比函数调用开销较大，则没有太大的意义；另一方面每一处内联函数的调用都要复制代码，消耗更多的内存空间，因此以下情况不宜使用内联函数：

- 函数体内的代码比较长，将导致内存消耗代价
- 函数体内有循环，函数执行时间要比函数调用开销大

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；

《inline函数和宏定义区别 整理》：<https://blog.csdn.net/wangliang888888/article/details/77990650>

- 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。

40、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

举个例子：

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5 public:
6     inline A() {
7         cout << "inline construct()" << endl;
8     }
9     inline ~A() {
10        cout << "inline destruct()" << endl;
11    }
12    inline virtual void virtualFun() {
13        cout << "inline virtual function" << endl;
14    }
15 };
16
17 int main()
18 {
19     A a;
20     a.virtualFun();
21     return 0;
22 }
23 //输出结果
24 //inline construct()
25 //inline virtual function
26 //inline destruct()
27
28
29

```

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。

将虚函数声明为inline，要分情况讨论

有的人认为虚函数被声明为inline，但是编译器并没有对其内联，他们给出的理由是inline是编译期决定的，而虚函数是运行期决定的，即在不知道将要调用哪个函数的情况下，如何将函数内联呢？

上述观点看似正确，其实不然，如果虚函数在编译器就能够决定将要调用哪个函数时，就能够内联，那么什么情况下编译器可以确定要调用哪个函数呢，答案是当用对象调用虚函数（此时不具有多态性）时，就内联展开

综上，当是指向派生类的指针（多态性）调用声明为inline的虚函数时，不会内联展开；当是对象本身调用虚函数时，会内联展开，当然前提依然是函数并不复杂的情况下

《构造函数、析构函数、虚函数可否内联，有何意义》：<https://www.cnblogs.com/helloweworld/archive/2013/06/14/3136705.html>

41、auto、decltype和decltype(auto)的用法

(1) auto

C++11新标准引入了auto类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应某种特定的类型说明符(例如 int)不同，

auto 让编译器通过初始值来进行类型推演。从而获得定义变量的类型，所以说 auto 定义的变量必须有初始值。举个例子：

```
1 //普通：类型
2 int a = 1, b = 3;
3 auto c = a + b; // c为int型
4
5 //const类型
6 const int i = 5;
7 auto j = i; // 变量i是顶层const，会被忽略，所以j的类型是int
8 auto k = &i; // 变量i是一个常量，对常量取地址是一种底层const，所以b的类型是const
9 int*
10 const auto l = i; //如果希望推断出的类型是顶层const的，那么就需要在auto前面加上cosnt
11
12 //引用和指针类型
13 int x = 2;
14 int& y = x;
15 auto z = y; //z是int型不是int& 型
16 auto& p1 = y; //p1是int&型
17 auto p2 = &x; //p2是指针类型int*
```

(2) decltype

有的时候我们还会遇到这种情况，**我们希望从表达式中推断出要定义变量的类型，但却不想用表达式的值去初始化变量。**还有可能是函数的返回类型为某表达式的值类型。在这些时候auto显得就无力了，所以C++11又引入了第二种类型说明符decltype，它的作用是选择并返回操作数的数据类型。在此过程中，编译器只是分析表达式并得到它的类型，却不进行实际的计算表达式的值。

```
1 int func() {return 0;};
2
3 //普通类型
4 decltype(func()) sum = 5; // sum的类型是函数func()的返回值的类型int，但是这时不会
  实际调用函数func()
```

```

5 int a = 0;
6 decltype(a) b = 4; // a的类型是int, 所以b的类型也是int
7
8 //不论是顶层const还是底层const, decltype都会保留
9 const int c = 3;
10 decltype(c) d = c; // d的类型和c是一样的, 都是顶层const
11 int e = 4;
12 const int* f = &e; // f是底层const
13 decltype(f) g = f; // g也是底层const
14
15 //引用与指针类型
16 //1. 如果表达式是引用类型, 那么decltype的类型也是引用
17 const int i = 3, &j = i;
18 decltype(j) k = 5; // k的类型是 const int&
19
20 //2. 如果表达式是引用类型, 但是想要得到这个引用所指向的类型, 需要修改表达式:
21 int i = 3, &r = i;
22 decltype(r + 0) t = 5; // 此时是int类型
23
24 //3. 对指针的解引用操作返回的是引用类型
25 int i = 3, j = 6, *p = &i;
26 decltype(*p) c = j; // c是int&类型, c和j绑定在一起
27
28 //4. 如果一个表达式的类型不是引用, 但是我们需要推断出引用, 那么可以加上一对括号, 就变成了
29 //引用类型了
30 int i = 3;
31 decltype((i)) j = i; // 此时j的类型是int&类型, j和i绑定在了一起
32
33

```

(3) decltype(auto)

decltype(auto)是C++14新增的类型指示符, 可以用来声明变量以及指示函数返回类型。在使用时, 会将“=”号左边的表达式替换掉auto, 再根据decltype的语法规则来确定类型。举个例子:

```

1 int e = 4;
2 const int* f = &e; // f是底层const
3 decltype(auto) j = f; //j的类型是const int* 并且指向的是e
4
5
6

```

《auto和decltype的用法总结》: <https://www.cnblogs.com/XiangfeiAi/p/4451904.html>

《C++11新特性中auto 和 decltype 区别和联系》: <https://www.jb51.net/article/103666.htm>

42、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。

基类成员	private	protected	public	private	protected	public	private	protected	public
派生方式	private			protected			public		
派生类中	不可见	private	private	不可见	protected	protected	不可见	protected	public
外部	不可见	不可见	不可见	不可见	不可见	不可见	不可见	不可见	可见

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，而基类的私有成员仍然是私有的，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

基类成员		private成员	protected成员	public成员
访问方式	内部访问	不可访问	可访问	可访问
	外部访问	不可访问	不可访问	不可访问

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

基类成员		private成员	protected成员	public成员
访问方式	内部访问	不可访问	可访问	可访问
	外部访问	不可访问	不可访问	不可访问

43、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：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12
内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

大端模式中的存储方式为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78
内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

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;
6     //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
7     char c = (char)(a);
8     if (c == 0x12)
9         cout << "big endian" << endl;
10    else if(c == 0x34)
11        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 {
6     int a;
7     char ch;
8 };
9 int main()
10 {
11     endian value;
12     value.a = 0x1234;
13     //a和ch共用4字节的内存空间
14     if (value.ch == 0x12)
15         cout << "big endian" << endl;
16     else if (value.ch == 0x34)
17         cout << "little endian" << endl;
18 }
19
20
21

```

《写程序判断系统是大端序还是小端序》：<https://www.cnblogs.com/zhoudayang/p/5985563.html>

44、volatile、mutable和explicit关键字的用法

(1) volatile

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。

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。

volatile 定义变量的值是易变的，每次用到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。多线程中被几个任务共享的变量需要定义为 volatile 类型。

volatile 指针

volatile 指针和 const 修饰词类似，const 有常量指针和指针常量的说法，volatile 也有相应的概念

修饰由指针指向的对象、数据是 const 或 volatile 的：

```
1 | const char* cpch;
2 | volatile char* vpch;
3 |
```

指针自身的值——一个代表地址的整数变量，是 const 或 volatile 的：

```
1 | char* const pchc;
2 | char* volatile pchv;
3 |
```

注意：

- 可以把一个非volatile int 赋给 volatile int，但是不能把非volatile 对象赋给一个 volatile 对象。
- 除了基本类型外，对用户定义类型也可以用 volatile 类型进行修饰。
- C++ 中一个有 volatile 标识符的类只能访问它接口的子集，一个由类的实现者控制的子集。用户只能用 const_cast 来获得对类型接口的完全访问。此外，volatile 向 const 一样会从类传递到它的成员。

多线程下的 volatile

有些变量是用 volatile 关键字声明的。当两个线程都要用到某一个变量且该变量的值会被改变时，应该用 volatile 声明，该关键字的作用是防止优化编译器把变量从内存装入 CPU 寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。volatile 的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值。

(2) mutable

mutable 的中文意思是“可变的，易变的”，跟 constant（既 C++ 中的 const）是反义词。在 C++ 中，mutable 也是为了突破 const 的限制而设置的。被 mutable 修饰的变量，将永远处于可变的状态，即使在一个 const 函数中。我们知道，如果类的成员函数不会改变对象的状态，那么这个成员函数一般会声明成 const 的。但是，有些时候，我们需要在 const 函数里面修改一些跟类状态无关的数据成员，那么这个函数就应该被 mutable 来修饰，并且放在函数后后面关键字位置。

(3) explicit

explicit关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只能以**显示的方式进行类型转换**，注意以下几点：

- explicit 关键字只能用于类内部的构造函数声明上
- explicit 关键字作用于单个参数的构造函数
- 被explicit修饰的构造函数的类，不能发生相应的隐式类型转换

45、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：

```
1 class A
2 {
3 public:
4     A() {};
5     A(const A& a)
6     {
7         cout << "copy constructor is called" << endl;
8     };
9     ~A() {};
10 };
11
12 void useClassA(A a) {}
13
14 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
15 {
16     A a;
17     return a;
18 }
19
20
21 //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
```

```

22 //{
23 // A a;
24 // return a;
25 //}
26
27
28 int main()
29 {
30     A a1, a2,a3,a4;
31     A a2 = a1; //调用拷贝构造函数,对应情况1
32     useClassA(a1); //调用拷贝构造函数, 对应情况2
33     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
34     a4 = getClassA2(a1); //发生NRV优化, 且引用返回自身, 不会调用
35     return 0;
36 }
37
38
39
40

```

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

《C++拷贝构造函数详解》：<https://www.cnblogs.com/alantu2018/p/8459250.html>

update1:<https://github.com/forthespada/InterviewGuide/issues/2> 提出，感谢！ - 2021.03.22

46、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：

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 int main()
5 {
6     try

```

```

7  {
8      char *p = new char[10e11];
9      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

```

(2) noexcept new

noexcept 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 {
7     char *p = new(nothrow) char[10e11];
8     if (p == NULL)
9     {
10         cout << "alloc failed" << endl;
11     }
12     delete p;
13     return 0;
14 }
15 //运行结果: alloc failed
16
17
18
19

```

(3) 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：

- placement 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 {  
19     char *p = new(nothrow) char[sizeof ADT + 1];  
20     if (p == NULL) {  
21         cout << "alloc failed" << endl;  
22     }  
23     ADT *q = new(p) ADT; //placement new:不必担心失败，只要p所指对象的空间足够  
ADT创建即可  
24     //delete q;//错误！不能在此处调用delete q;  
25     q->ADT::~ADT(); //显示调用析构函数  
26     delete[] p;  
27     return 0;  
28 }  
29 //输出结果:  
30 //ADT construct i=10j=100  
31 //ADT destruct  
32  
33  
34  
35
```

47、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
4 void fun(char* p) {
5     cout << "char*" << endl;
6 }
7
8 void fun(int p) {
9     cout << "int" << endl;
10 }
11
12 int main()
13 {
14     fun(NULL);
15     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 {
19     fun((char*)nullptr); //语句1
20     fun(nullptr); //语句2
21     fun(NULL); //语句3
22     return 0;
23 }
24 //运行结果:
25 //语句1: char* p
26 //语句2: 报错, 有多个匹配
27 //3: int p
28
29
30

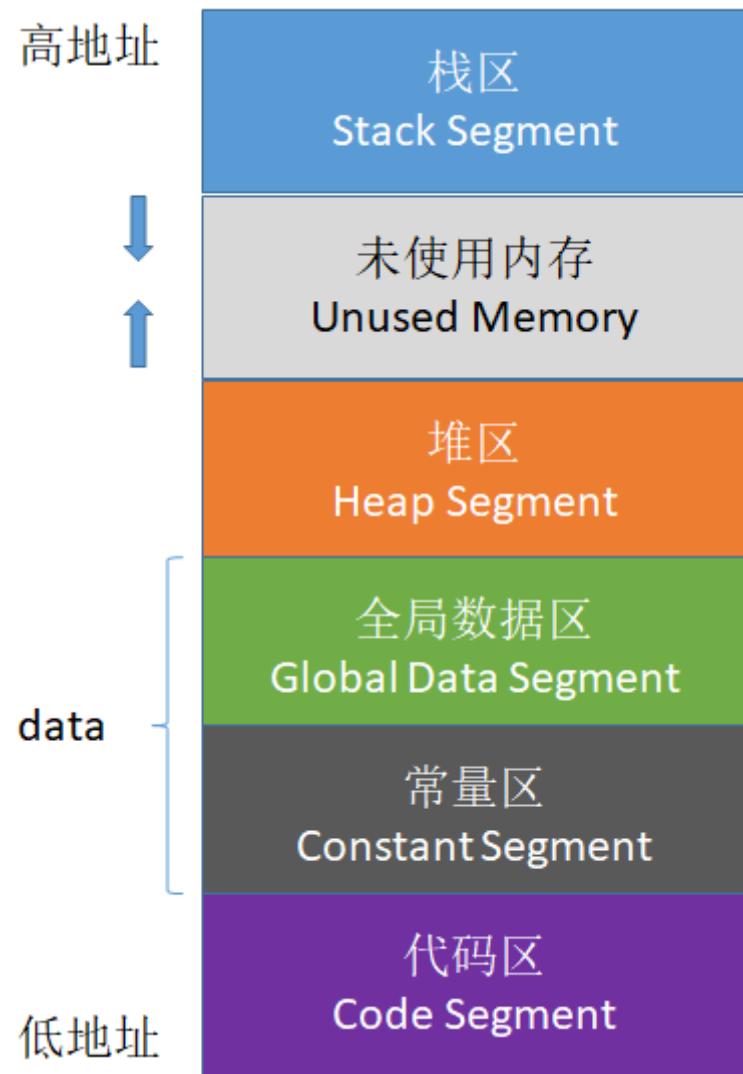
```

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

48、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/>

49、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 {
5     double m = 1, n = 0;
6     try {
7         cout << "before dividing." << endl;
8         if (n == 0)
9             throw - 1; //抛出int型异常
10        else if (m == 0)
11            throw - 1.0; //抛出 double 型异常
12        else
13            cout << m / n << endl;
14        cout << "after dividing." << endl;
15    }
16    catch (double d) {
17        cout << "catch (double)" << d << endl;
18    }
19    catch (...) {
20        cout << "catch (...)" << endl;
21    }
22    cout << "finished" << endl;
23    return 0;
24 }
25 //运行结果
26 //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。

(2) 函数的异常声明列表

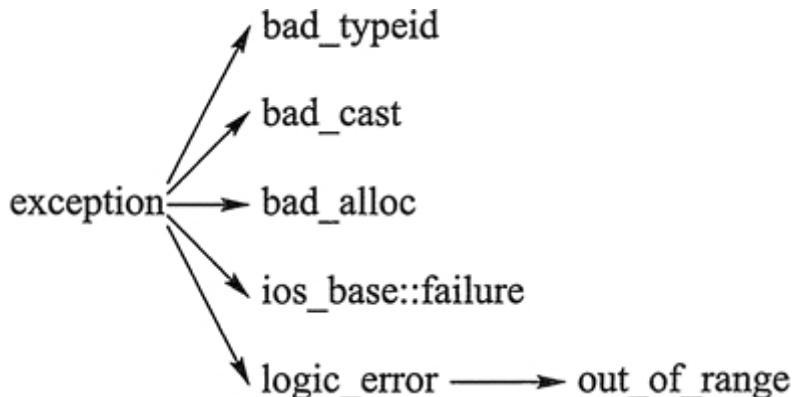
有时候，程序员在定义函数的时候知道函数可能发生的异常，可以在函数声明和定义时，指出所能抛出异常的列表，写法如下：

```
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

(3) 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，则会抛出此异常，例如：

```
1 #include <iostream>  
2 #include <typeinfo>  
3 using namespace std;  
4  
5 class A{  
6 public:  
7     virtual ~A();  
8 };  
9  
10 using namespace std;  
11 int main() {  
12     A* a = NULL;  
13     try {  
14         cout << typeid(*a).name() << endl; // Error condition  
15     }  
16     catch (bad_typeid){  
17         cout << "Object is NULL" << endl;  
18     }  
19     return 0;  
20 }  
21 //运行结果: Object 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，则会抛出此异常
- 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，则会抛出此异常

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

50、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。

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->cpointer->virtual function`

51、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。

52、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；
- 3) 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；
- 4) 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；
- 5) 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”。
- 6) 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；
- 7) 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；
- 8) 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。
- 9) 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；
- 10) 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；
- 11) 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。

53、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。

54、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。

55、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。

56、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

57、什么是内存池，如何实现

<https://www.bilibili.com/video/BV1Kb411B7N8?p=25> C++内存管理：P23-26

<https://www.bilibili.com/video/BV1db411q7B8?p=12> C++STL P11

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

- 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 $(20 \times 32\text{byte})$
- 客户端之后又有内存需求，想申请 $(20 \times 64\text{bytes})$ 的空间，这时内存池只有 $(20 \times 32\text{bytes})$ ，就先将 $(10 \times 64\text{bytes})$ 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
- 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。
- 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

《STL源码解析》侯捷 P68

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，所以老师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()

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 $16 \times 8 = 128\text{bytes}$ ，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

58、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。

59、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。

2) 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。

60、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。

61、new和malloc的区别？

- 1、 new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持；
- 2、 使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而malloc则需要显式地指出所需内存的尺寸。
- 3、 new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void *，需要通过强制类型转换将void*指针转换成我们需要的类型。
- 4、 new内存分配失败时，会抛出bad_alloc异常。malloc分配内存失败时返回NULL。
- 5、 new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

62、delete p、delete [] p、allocator都有什么作用？

- 1、 动态数组管理new一个数组时，[]中必须是一个整数，但是不一定是常量整数，普通数组必须是一个常量整数；
- 2、 new动态数组返回的并不是数组类型，而是一个元素类型的指针；
- 3、 delete[]时，数组中的元素按逆序的顺序进行销毁；
- 4、 new在内存分配上面有一些局限性，new的机制是将内存分配和对象构造组合在一起，同样的，delete也是将对象析构和内存释放组合在一起的。allocator将这两部分分开进行，allocator申请一部分内存，不进行初始化对象，只有当需要的时候才进行初始化操作。

63、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。

64、malloc申请的存储空间能用delete释放吗

不能，malloc /free主要为了兼容C，new和delete完全可以取代malloc /free的。

malloc /free的操作对象都是必须明确大小的，而且不能用在动态类上。

new 和delete会自动进行类型检查和大小，malloc/free不能执行构造函数与析构函数，所以动态对象它是不行的。

当然从理论上说使用malloc申请的内存是可以通过delete释放的。不过一般不这样写的。而且也不能保证每个C++的运行时都能正常。

65、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。

66、malloc、realloc、calloc的区别

1) malloc函数

```
1 void* malloc(unsigned int num_size);
2
3 int *p = malloc(20*sizeof(int)); //申请20个int类型的空间;
4
```

2) calloc函数

```
1 void* calloc(size_t n, size_t size);
2
3 int *p = calloc(20, sizeof(int));
4
```

省去了人为空间计算；malloc申请的空间的值是随机初始化的，calloc申请的空间的值是初始化为0的；

3) realloc函数

```
1 | void realloc(void *p, size_t new_size);  
2 |
```

给动态分配的空间分配额外的空间，用于扩充容量。

67、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。

2) 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。

3) 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。

68、成员列表初始化？

1) 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；

2) 成员初始化列表做了什么

- ① 编译器会——操作初始化列表，以适当的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之前；
- ② list中的项目顺序是由类中的成员声明顺序决定的，不是由初始化列表的顺序决定的；

69、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**

70、对象复用的了解，零拷贝的了解

对象复用

对象复用其本质是一种设计模式：Flyweight享元模式。

通过将对象存储到“对象池”中实现对象的重复利用，这样可以避免多次创建重复对象的开销，节约系统资源。

零拷贝

零拷贝就是一种避免 CPU 将数据从一块存储拷贝到另外一块存储的技术。

零拷贝技术可以减少数据拷贝和共享总线操作的次数。

在C++中，vector的一个成员函数**emplace_back()**很好地体现了零拷贝技术，它跟push_back()函数一样可以将一个元素插入容器尾部，区别在于：**使用push_back()函数需要调用拷贝构造函数和转移构造函数，而使用emplace_back()插入的元素原地构造，不需要触发拷贝构造和转移构造**，效率更高。举个例子：

```
1 #include <vector>
2 #include <string>
3 #include <iostream>
4 using namespace std;
5
6 struct Person
7 {
8     string name;
9     int age;
10    //初始构造函数
11    Person(string p_name, int p_age): name(std::move(p_name)), age(p_age)
12    {
13        cout << "I have been constructed" << endl;
14    }
15    //拷贝构造函数
16    Person(const Person& other): name(std::move(other.name)),
17    age(other.age)
18    {
19        cout << "I have been copy constructed" << endl;
20    }
21}
```

```

19     }
20     //转移构造函数
21     Person(Person&& other): name(std::move(other.name)), age(other.age)
22     {
23         cout << "I have been moved" << endl;
24     }
25 };
26
27 int main()
28 {
29     vector<Person> e;
30     cout << "emplace_back:" << endl;
31     e.emplace_back("Jane", 23); //不用构造类对象
32
33     vector<Person> p;
34     cout << "push_back:" << endl;
35     p.push_back(Person("Mike", 36));
36     return 0;
37 }
38 //输出结果:
39 //emplace_back:
40 //I have been constructed
41 //push_back:
42 //I have been constructed
43 //I am being moved.
44
45
46

```

71、解释一下什么是trivial destructor

“trivial destructor”一般是指用户没有自定义析构函数，而由系统生成的，这种析构函数在《STL源码解析》中成为“无关痛痒”的析构函数。

反之，用户自定义了析构函数，则称之为“non-trivial destructor”，这种析构函数如果申请了新的空间一定要显式的释放，否则会造成内存泄露

对于trivial destructor，如果每次都进行调用，显然对效率是一种伤害，如何进行判断呢？《STL源码解析》中给出的说明是：

首先利用value_type()获取所指对象的型别，再利用__type_traits判断该型别的析构函数是否 trivial，若是(__true_type)，则什么也不做，若为(__false_type)，则去调用destory()函数

也就是说，在实际的应用当中，STL库提供了相关的判断方法__type_traits，感兴趣的读者可以自行查阅使用方式。除了trivial destructor，还有trivial construct、trivial copy construct等，如果能够对是否trivial进行区分，可以采用内存处理函数memcpy()、malloc()等更加高效的完成相关操作，提升效率。

《C++中的 trivial destructor》：<https://blog.csdn.net/wudishine/article/details/12307611>

72、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

(2) 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。

(3) 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。

《C++封装继承多态总结》：https://blog.csdn.net/IOT_SHUN/article/details/79674293

73、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：

```
1 #include <iostream>
2 using namespace std;
3
4 class Person
5 {
6 public:
7     Person()
8     {
9         this->age = 23;
```

```

10     }
11     void printAge()
12     {
13         cout << this->age << endl;
14     }
15     ~Person(){}
16 public:
17     int age;
18 };
19
20 int main()
21 {
22     Person p;
23     cout << "对象地址: "<< &p << endl;
24     cout << "age地址: "<< &(p.age) << endl;
25     cout << "对象大小: "<< sizeof(p) << endl;
26     cout << "age大小: "<< sizeof(p.age) << endl;
27     return 0;
28 }
29 //输出结果
30 //对象地址: 0x7fffec0f15a8
31 //age地址: 0x7fffec0f15a8
32 //对象大小: 4
33 //age大小: 4
34
35
36
37

```

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。**

《C++类对象成员变量和函数内存分配的问题》：<https://blog.csdn.net/z2664836046/article/details/78967313>

74、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    {
12        value = a;
13        cout << "A(int "<<value<<")" << endl;
14    }
15    A(const A& a)
16    {
17        value = a.value;
18        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     }
30     A a;
31     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，等于是第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/details/82590368

75、(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访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，就不能正确认识对象类型从而不能正确调用析构函数。

4. 从实现上看，vtable 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就是初始化它的VPTTR。

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VPTTR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，VPTTR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VPTTR指向它的VTABLE，直到最后的构造函数结束。

VPTTR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VPTTR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。

76. 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。

2) 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。

77、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。

78、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。

79、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展，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。

80、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。

81、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>

82、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，而其基类虚构造函数是虚函数时；

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。

83、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：

```
1 #include<iostream>
2 using namespace std;
3
```

```
4 class Base
5 {
6     public:
7         Base()
8         {
9             Function();
10        }
11
12     virtual void Function()
13     {
14         cout << "Base::Function" << 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 {
42     Base* a = new Base;
43     delete a;
44     cout << "-----" << endl;
45     Base* b = new A;//语句1
46     delete b;
47 }
48 //输出结果
49 //Base::Function
50 //Base::Function
51 //-----
52 //Base::Function
53 //A::Function
54 //Base::Function
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/TaJ1JONX>

84、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）

(2) 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。

(3) 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有效的，在使用之前使用函数lock()检查weak_ptr是否为空指针。

(4) 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 {
4 public:
5     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
12    SharedPtr<T>& operator=(const SharedPtr& s){
13        if (this != &s)
14        {
15            if (--(*(this->_pcount)) == 0)
16            {
17                delete this->_ptr;
18                delete this->_pcount;
19            }
20            _ptr = s._ptr;
21            _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        {
39            delete _ptr;
40            _ptr = NULL;
41            delete _pcount;
42            _pcount = NULL;
43        }
44    }
45 private:
46     T* _ptr;
47     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>

85、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 {
6 public:
7     CString() = default; //语句1
8     //构造函数
9     CString(const char* pstr) : _str(pstr){}
10    void* operator new() = delete;//这样不允许使用new关键字
11    //析构函数
12    ~CString(){}
13 public:
14     string _str;
15 };
16
17
18 int main()
19 {
20     auto a = new CString(); //语句2
21     cout << "Hello world" << endl;
22     return 0;
23 }
24 //运行结果
25 //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

```
1 #include <iostream>
2 using namespace std;
3
4 class CString
5 {
6 public:
7     void* operator new() = delete;//这样不允许使用new关键字
8     //析构函数
9     ~CString(){}
10 }
11
12
13 int main()
14 {
```

```
15     auto a = new CString(); //语句1
16     cout << "Hello world" << endl;
17     return 0;
18 }
19
20
21
22
```

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/71101737>

86、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（也就是说，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：

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 class Base
5 {
6 public:
7     Base() :b(1) {}
8     virtual void fun() {};
9     int b;
10};
11
12 class Son : public Base
13 {
14 public:
15     Son() :d(2) {}
16     int d;
17 };
18
19 int main()
20 {
21     int n = 97;
22
23     //reinterpret_cast
24     int *p = &n;
25     //以下两者效果相同
26     char *c = reinterpret_cast<char*>(p);
27     char *c2 = (char*)(p);
28     cout << "reinterpret_cast输出: " << *c2 << endl;
29     //const_cast
30     const int *p2 = &n;
31     int *p3 = const_cast<int*>(p2);
32     *p3 = 100;
33     cout << "const_cast输出: " << *p3 << endl;
34 }
```

```

35     Base* b1 = new Son;
36     Base* b2 = new Base;
37
38     //static_cast
39     Son* s1 = static_cast<Son*>(b1); //同类型转换
40     Son* s2 = static_cast<Son*>(b2); //下行转换, 不安全
41     cout << "static_cast输出: " << endl;
42     cout << s1->d << endl;
43     cout << s2->d << endl; //下行转换, 原先父对象没有d成员, 输出垃圾值
44
45     //dynamic_cast
46     Son* s3 = dynamic_cast<Son*>(b1); //同类型转换
47     Son* s4 = dynamic_cast<Son*>(b2); //下行转换, 安全
48     cout << "dynamic_cast输出: " << endl;
49     cout << s3->d << endl;
50     if(s4 == nullptr)
51         cout << "s4指针为nullptr" << endl;
52     else
53         cout << s4->d << endl;
54
55
56     return 0;
57 }
58 //输出结果
59 //reinterpret_cast输出: a
60 //const_cast输出: 100
61 //static_cast输出:
62 //2
63 //-33686019
64 //dynamic_cast输出:
65 //2
66 //s4指针为nullptr
67
68
69
70

```

从输出结果可以看出，在进行下行转换时，dynamic_cast是安全的，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。

87、C++函数调用的压栈过程

从代码入手，解释这个过程：

```

1 #include <iostream>
2 using namespace std;
3
4 int f(int n)
5 {
6     cout << n << endl;
7     return n;
8 }
9
10 void func(int param1, int param2)

```

```

11 {
12     int var1 = param1;
13     int var2 = param2;
14     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
15 }
16
17 int main(int argc, char* argv[])
18 {
19     func(1, 2);
20     return 0;
21 }
22 //输出结果
23 //2
24 //1
25 //var1=1,var2=2
26
27
28
29

```

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/2012/07/20/2601204.html>

《C/C++函数调用的压栈模型》：https://blog.csdn.net/m0_37717595/article/details/80368411

88、说说移动构造函数

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；

3) 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。

89、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任何意义的值了

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)

90、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。

91、几个**this**指针的易混问题

A. **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

B. **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。

C. **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

D. **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。

E. 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。

F.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>

92、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    {
17        cout << "我是赋值操作符" << endl;
18        return *this;
19    }
20    ~A() {};
21 };
22
23 int main()
24 {
25     A a1; //调用构造函数
26     A a2 = a1; //调用拷贝构造函数
27     a2 = a1; //调用赋值操作符
28     return 0;
29 }
```

```
30 //输出结果  
31 //我是构造函数  
32 //我是拷贝构造函数  
33 //我是赋值操作符  
34  
35  
36  
37
```

93、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：

```
1 Student s;  
2 Student s1 = s;      // 调用拷贝构造函数  
3 Student s2;  
4 s2 = s;      // 赋值运算符操作
```

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

94、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

- 4) `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。

5) 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）。

6) 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计数的增加或减少。

95、说说你了解的auto_ptr作用

- 1) auto_ptr的出现，主要是为了解决“有异常抛出时发生内存泄漏”的问题；抛出异常，将导致指针p所指向的空间得不到释放而导致内存泄漏；
- 2) auto_ptr构造时取得某个对象的控制权，在析构时释放该对象。我们实际上是创建一个auto_ptr类型的局部对象，该局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏；
- 3) auto_ptr的构造函数是explicit，阻止了一般指针隐式转换为 auto_ptr的构造，所以不能直接将一般类型的指针赋值给auto_ptr类型的对象，必须用auto_ptr的构造函数创建对象；
- 4) 由于auto_ptr对象析构时会删除它所拥有的指针，所以使用时避免多个auto_ptr对象管理同一个指针；
- 5) Auto_ptr内部实现，析构函数中删除对象用的是delete而不是delete[]，所以auto_ptr不能管理数组；
- 6) auto_ptr支持所拥有的指针类型之间的隐式类型转换。
- 7) 可以通过*和->运算符对auto_ptr所有用的指针进行提领操作；
- 8) T* get(),获得auto_ptr所拥有的指针；T* release(), 释放auto_ptr的所有权，并将所有用的指针返回。

96、智能指针的循环引用

循环引用是指使用多个智能指针share_ptr时，出现了指针之间相互指向，从而形成环的情况，有点类似于死锁的情况，这种情况下，智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏。举个例子：

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 class Node
6 {
7 public:
8     Node(const T& value)
9         :_pPre(NULL)
10        , _pNext(NULL)
11        , _value(value)
12    {
13        cout << "Node()" << endl;
14    }
}
```

```

15     ~Node()
16     {
17         cout << "~Node()" << endl;
18         cout << "this:" << this << endl;
19     }
20
21     shared_ptr<Node<T>> _pPre;
22     shared_ptr<Node<T>> _pNext;
23     T _value;
24 };
25
26 void Funtest()
27 {
28     shared_ptr<Node<int>> sp1(new Node<int>(1));
29     shared_ptr<Node<int>> sp2(new Node<int>(2));
30
31     cout << "sp1.use_count:" << sp1.use_count() << endl;
32     cout << "sp2.use_count:" << sp2.use_count() << endl;
33
34     sp1->_pNext = sp2; //sp1的引用+1
35     sp2->_pPre = sp1; //sp2的引用+1
36
37     cout << "sp1.use_count:" << sp1.use_count() << endl;
38     cout << "sp2.use_count:" << sp2.use_count() << endl;
39 }
40 int main()
41 {
42     Funtest();
43     system("pause");
44     return 0;
45 }
//输出结果
//Node()
//Node()
//sp1.use_count:1
//sp2.use_count:1
//sp1.use_count:2
//sp2.use_count:2
53
54
55
56

```

从上面shared_ptr的实现中我们知道了只有当引用计数减减之后等于0，析构时才会释放对象，而上述情况造成了一个僵局，那就是析构对象时先析构sp2，可是由于sp2的空间sp1还在使用中，所以sp2.use_count减减之后为1，不释放，sp1也是相同道理，由于sp1的空间sp2还在使用中，所以sp1.use_count减减之后为1，也不释放。sp1等着sp2先释放，sp2等着sp1先释放，二者互不相让，导致最终都没能释放，内存泄漏。

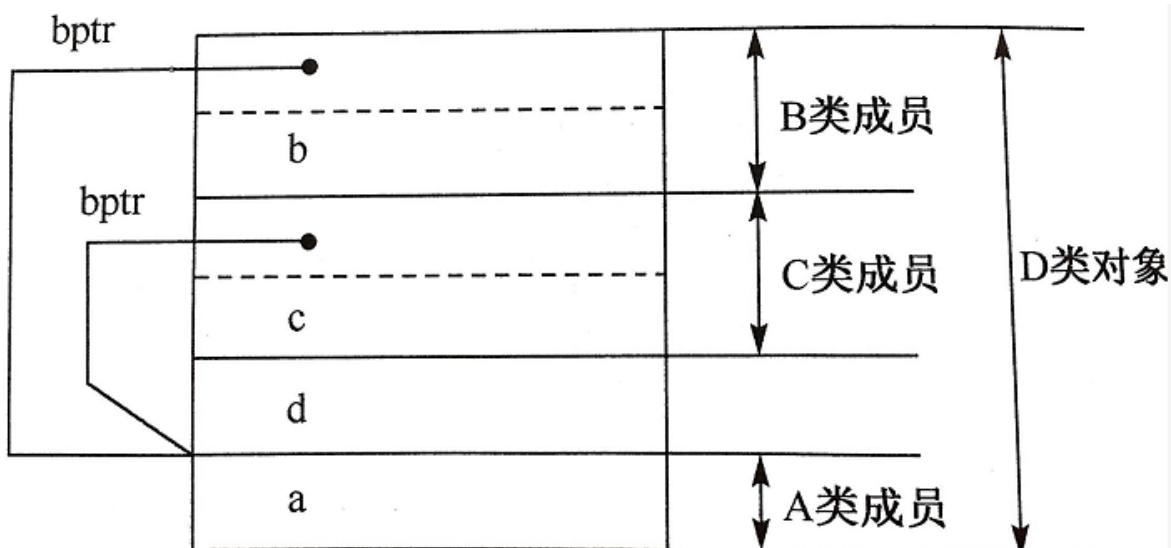
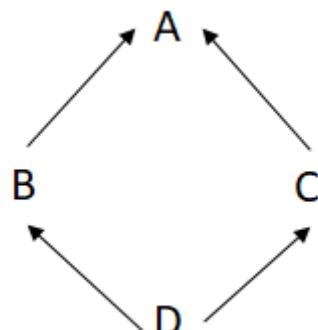
在实际编程过程中，应该尽量避免出现智能指针之前相互指向的情况，如果不可避免，可以使用使用弱指针——weak_ptr，它不增加引用计数，只要出了作用域就会自动析构。

《C++ 智能指针（及循环引用问题）》：https://blog.csdn.net/m0_37968340/article/details/76737395

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
4 class A{}
5 class B : virtual public A{};
6 class C : virtual public A{};
7 class D : public B, public C{};
8
9 int main()
10 {
11     cout << "sizeof(A): " << sizeof A << endl; // 1, 空对象, 只有一个占位
12     cout << "sizeof(B): " << sizeof B << endl; // 4, 一个bptr指针, 省去占位, 不需
要对齐
13     cout << "sizeof(C): " << sizeof C << endl; // 4, 一个bptr指针, 省去占位, 不需
要对齐
14     cout << "sizeof(D): " << sizeof D << endl; // 8, 两个bptr, 省去占位, 不需要对
齐
15 }
```

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，或者指向虚基类对象，或者指向一个相关的表格；表格中存放的不是虚基类对象的地址，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

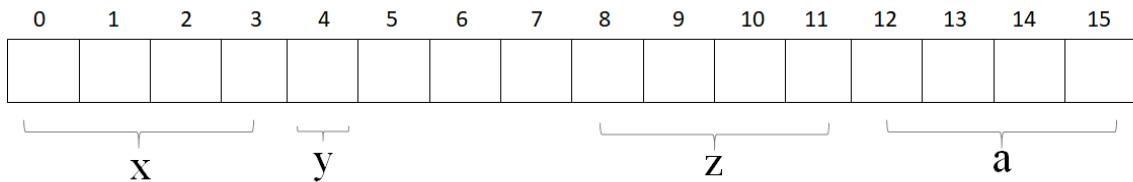
98、如何获得结构成员相对于结构开头的字节偏移量

使用<stddef.h>头文件中的，offsetof宏。

举个例子：

```
1 #include <iostream>
2 #include <stddef.h>
3 using namespace std;
4
5 struct S
6 {
7     int x;
8     char y;
9     int z;
10    double a;
11 };
12 int main()
13 {
14     cout << offsetof(S, x) << endl; // 0
15     cout << offsetof(S, y) << endl; // 4
16     cout << offsetof(S, z) << endl; // 8
17     cout << offsetof(S, a) << endl; // 12
18     return 0;
19 }
20
21 在vs2019 + win下 并不是这样的
22
23     cout << offsetof(S, x) << endl; // 0
24     cout << offsetof(S, y) << endl; // 4
25     cout << offsetof(S, z) << endl; // 8
26     cout << offsetof(S, a) << endl; // 16 这里是 16的位置，因为 double是8字节，需要找一个8的倍数对齐
27 当然了，如果加上 #pragma pack(4)指定 4字节对齐就可以了
28 #pragma pack(4)
29 struct S
30 {
31     int x;
32     char y;
33     int z;
34     double a;
35 };
36 void test02()
37 {
38
39     cout << offsetof(S, x) << endl; // 0
40     cout << offsetof(S, y) << endl; // 4
41     cout << offsetof(S, z) << endl; // 8
42     cout << offsetof(S, a) << endl; // 12
43 }
44
```

S结构体中各个数据成员的内存空间划分如下所示，需要注意内存对齐



99、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：

```

1 #include <iostream>
2 using namespace std;
3
4 class A
5 {
6 public:
7     /*virtual*/ void func() { std::cout << "A::func()\n"; }
8 };
9 class B : public A
10 {
11 public:
12     void func() { std::cout << "B::func()\n"; }
13 };
14 class C : public A
15 {
16 public:
17     void func() { std::cout << "C::func()\n"; }
18 };
19 int main()
20 {
21     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*；
22     B* pb = new B(); //pb的静态类型和动态类型也都是B*；
23     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
24     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
25     //NULL;
26
27     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
28     //是直接调用A::func();
29     pc->func();      //C::func() pc的动、静态类型都是C*，因此调用C::func();

```

```

29     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；
30     return 0;
31 }
32
33
34

```

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；
2 pc->func(); //C::func() pc的动、静态类型都是C*，因此也是先在C中查找；
3 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：

```

1 #include <iostream>
2 using namespace std;
3
4 class E
5 {
6 public:
7     virtual void func(int i = 0)
8     {
9         std::cout << "E::func()\t" << i << "\n";
10    }
11 };
12 class F : public E

```

```

13 {
14     public:
15         virtual void func(int i = 1)
16         {
17             std::cout << "F::func()\t" << i << "\n";
18         }
19     };
20
21 void test2()
22 {
23     F* pf = new F();
24     E* pe = pf;
25     pf->func(); //F::func() 1 正常，就该如此;
26     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！
27 }
28 int main()
29 {
30     test2();
31     return 0;
32 }
33
34
35

```

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

100、C++ 11有哪些新特性？

- nullptr替代NULL
- 引入了auto 和 decltype 这两个关键字实现了类型推导
- 基于范围的for循环for(auto& i : res){}
- 类和结构体的中初始化列表
- Lambda 表达式（匿名函数）
- std::forward_list（单向链表）
- 右值引用和move语义
- ...

101、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。

举个例子：

```

1 #include <iostream>
2 using namespace std;
3
4 class Base
5 {
6 public:

```

```

7     virtual void fun()
8     {
9         cout << "base :: fun()" << endl;
10    }
11 };
12
13 class Son : public Base
14 {
15 public:
16     virtual void fun()
17     {
18         cout << "son :: fun()" << endl;
19     }
20     void func()
21     {
22         cout << "son :: not virtual function" << endl;
23     }
24 };
25
26 int main()
27 {
28     Son s;
29     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
30     s.fun(); //son::fun()
31     b.fun(); //son :: fun()
32     return 0;
33 }
34
35
36

```

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。

102、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

103、指针加减计算要注意什么？

指针加减本质是对其所指地址的移动，移动的步长跟指针的类型是有关的，因此在涉及到指针加减运算需要十分小心，加多或者减多都会导致指针指向一块未知的内存地址，如果再进行操作就会很危险。

举个例子：

```

2  using namespace std;
3
4  int main()
{
5      int *a, *b, c;
6      a = (int*)0x500;
7      b = (int*)0x520;
8      c = b - a;
9      printf("%d\n", c); // 8
10     a += 0x020;
11     c = b - a;
12     printf("%d\n", c); // -24
13     return 0;
14 }
15
16
17
18

```

首先变量a和b都是以16进制的形式初始化，将它们转成10进制分别是1280 ($5*16^2=1280$) 和1312 ($5*16^2+2*16=1312$)，那么它们的差值为32，也就是说a和b所指向的地址之间间隔32个位，但是考虑到是int类型占4位，所以c的值为 $32/4=8$

a自增16进制0x20之后，其实际地址变为 $1280 + 2*16*4 = 1408$ ，(因为一个int占4位，所以要乘4)，这样它们的差值就变成了 $1312 - 1280 = -96$ ，所以c的值就变成了 $-96/4 = -24$

遇到指针的计算，需要明确的是指针每移动一位，它实际跨越的内存间隔是指针类型的长度，建议都转成10进制计算，计算结果除以类型长度取得结果

104、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。

105、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。

4) 过程调用和返回指令

① call指令

② leave指令

③ ret指令

106、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）。

2) 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。

3) 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。

4) 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。

107、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；

3) 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。

108、如果想将某个类用作基类，为什么该类必须定义而非声明？

派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类必须知道他们是什么。

109、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。

110、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。

(2) 抽象类的作用：

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

(3) 使用抽象类时注意：

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

抽象类是不能定义对象的。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。

111、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。

112、知道C++中的组合吗？它与继承相比有什么优缺点吗？

一：继承

继承是Is a 的关系，比如说Student继承Person,则说明Student is a Person。继承的优点是子类可以重写父类的方法来方便地实现对父类的扩展。

继承的缺点有以下几点：

- ①：父类的内部细节对子类是可见的。
- ②：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。
- ③：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

二：组合

组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量。

组合的优点：

- ①：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的。
- ②：当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码。
- ③：当前对象可以在运行时动态的绑定所包含的对象。可以通过set方法给所包含对象赋值。

组合的缺点：①：容易产生过多的对象。②：为了能组合多个对象，必须仔细对接口进行定义。

113、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)。

2) 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。

3) 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。

- 4) 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；

5) 两种方法赋值：

指针名 = 函数名； 指针名 = &函数名

114、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；

2) 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；

3) 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。

4) 解决方法

智能指针。

5) 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>

115、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。

2) 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。

116、手写实现智能指针类

1) 智能指针是一个数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录SmartPointer<T*>对象的引用计数，一旦T类型对象的引用计数为0，就释放该对象。

除了指针对象外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为1，需要一个构造函数。新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存。

通过覆写赋值运算符，才能将一个旧的智能指针赋值给另一个指针，同时旧的引用计数减1，新的引用计数加1

2) 一个构造函数、拷贝构造函数、复制构造函数、析构函数、移走函数；

117、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

118、结构体变量比较是否相等

1) 重载了“==”操作符

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

2) 元素的话，一个个比；

3) 指针直接比较，如果保存的是同一个实例地址，则($p1==p2$)为真；

119、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;

120、**define**、**const**、**typedef**、**inline**的使用方法？他们之间有什么区别？

一、**const**与**#define**的区别：

- 1) const定义的常量是变量带类型，而#define定义的只是个常数不带类型；
- 2) define只在预处理阶段起作用，简单的文本替换，而const在编译、链接过程中起作用；
- 3) define只是简单的字符串替换没有类型检查。而const是有数据类型的，是要进行判断的，可以避免一些低级错误；
- 4) define预处理后，占用代码段空间，const占用数据段空间；
- 5) const不能重定义，而define可以通过#undef取消某个符号的定义，进行重定义；
- 6) define独特功能，比如可以用来防止文件重复引用。

二、**#define**和别名**typedef**的区别

- 1) 执行时间不同，typedef在编译阶段有效，typedef有类型检查的功能；#define是宏定义，发生在预处理阶段，不进行类型检查；
- 2) 功能差异，typedef用来定义类型的别名，定义与平台无关的数据类型，与struct的结合使用等。#define不只是可以为类型取别名，还可以定义常量、变量、编译开关等。
- 3) 作用域不同，#define没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。而typedef有自己的作用域。

三、**define**与**inline**的区别

- 1) #define是关键字，inline是函数；
- 2) 宏定义在预处理阶段进行文本替换，inline函数在编译阶段进行替换；
- 3) inline函数有类型检查，相比宏定义比较安全；

121、你知道**printf**函数的实现原理是什么吗？

在C/C++中，对函数参数的扫描是从后向前的。

C/C++的函数参数是通过压入堆栈的方式来给函数传参数的（堆栈是一种先进后出的数据结构），最先压入的参数最后出来，在计算机的内存中，数据有2块，一块是堆，一块是栈（函数参数及局部变量在这里），而栈是从内存的高地址向低地址生长的，控制生长的就是堆栈指针了，最先压入的参数是在最上面，就是说在所有参数的最后面，最后压入的参数在最下面，结构上看起来是第一个，所以最后压入的参数总是能够被函数找到，因为它就在堆栈指针的上方。

printf的第一个被找到的参数就是那个字符指针，就是被双引号括起来的那一部分，函数通过判断字符串里控制参数的个数来判断参数个数及数据类型，通过这些就可算出数据需要的堆栈指针的偏移量了，下面给出printf("%d,%d",a,b); (其中a、b都是int型的) 的汇编代码.

122、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};
```

- 4) 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；

123、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. helloworld程序执行puts函数（系统调用），在显示器上写一字符串
8. 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
9. 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
10. 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
11. 显示器解释信号，激发液晶屏
12. OK，我们在屏幕上看到了HelloWorld

124、模板类和模板函数的区别是什么？

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。在使用时类模板必须加，而函数模板不必。

125、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。

2) 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。

126、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。

2) 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。

127、cout和printf有什么区别？

`cout<<`是一个函数，`cout<<`后可以跟不同的类型是因为`cout<<`已存在针对各种类型数据的重载，所以会自动识别数据的类型。输出过程会首先将输出字符放入缓冲区，然后输出到屏幕。

`cout`是有缓冲输出：

```
1 cout << "abc " << endl;
2 或cout << "abc\n ";cout << flush; 这两个才是一样的.
3
4
```

flush立即强迫缓冲输出。

printf是无缓冲输出。有输出时立即输出

128、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；

129、当程序中有函数重载时，函数的匹配原则和顺序是什么？

- 1) 名字查找
- 2) 确定候选函数
- 3) 寻找最佳匹配

130、定义和声明的区别

如果是指变量的声明和定义

从编译原理上来说，声明是仅仅告诉编译器，有个某类型的变量会被使用，但是编译器并不会为它分配任何内存。而定义就是分配了内存。

如果是指函数的声明和定义

声明：一般在头文件里，对编译器说：这里我有一个函数叫function() 让编译器知道这个函数的存在。

定义：一般在源文件里，具体就是函数的实现过程 写明函数体。

131、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）

132、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；

2) 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；

3) 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；

4) 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；

5) 默认实参

可以使用静态成员变量作为默认实参，

133、说一下你理解的 ifdef endif 代表着什么？

1) 一般情况下，源程序中所有的行都参加编译。但是有时希望对其中一部分内容只在满足一定条件才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。有时，希望当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。

2) 条件编译命令最常见的形式为：

```
1 #ifdef 标识符  
2   程序段1  
3 #else  
4   程序段2  
5 #endif  
6
```

它的作用是：当标识符已经被定义过(一般是用#define命令定义)，则对程序段1进行编译，否则编译程序段2。

其中#else部分也可以没有，即：

```
1  \#ifdef  
2  程序段1  
3  \#endif  
4
```

3) 在一个大的软件工程里面，可能会有多个文件同时包含一个头文件，当这些文件编译链接成一个可执行文件上时，就会出现大量“重定义”错误。

在头文件中使用#define、#ifndef、#ifdef、#endif能避免头文件重定义。

134、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。

135、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。

136、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。

137、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
```

- 2) 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
```

138、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。

- 2) 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 }
```

139、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。

140、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吗，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“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。

141、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 |
10| x = x^y; // 只能对int,char..
11| y = x^y;
12| x = x^y;
13| x ^= y ^= x;
14|
15|
```

142、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`

143、程序在执行`int main(int argc, char *argv[])`时的内存结构，你了解吗？

参数的含义是程序在命令行下运行的时候，需要输入`argc`个参数，每个参数是以`char`类型输入的，依次存在数组里面，数组是`argv[]`，所有的参数在指针

`char *`指向的内存中，数组的中元素的个数为`argc`个，第一个参数为程序的名称。

144、`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；

145、如果有一个空类，它会默认添加哪些函数？

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

146、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。

2) 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

3) 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

147、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：

```
1) a) string转const char*
2)
3) string s = "abc";
4)
5) const char* c_s = s.c_str();
6)
7) b) const char* 转string, 直接赋值即可
8)
9) const char* c_s = "abc";
10) string s(c_s);
11)
12) c) string 转char*
13) string s = "abc";
14) char* c;
15) const int len = s.length();
16) c = new char[len+1];
17) strcpy(c,s.c_str());
18)
19) d) char* 转string
20) char* c = "abc";
21) string s(c);
```

```

22
23 e) const char* 转char*
24     const char* cpc = "abc";
25     char* pc = new char[strlen(cpc)+1];
26     strcpy(pc,cpc);
27
28 f) char* 转const char*, 直接赋值即可
29     char* pc = "abc";
30     const char* cpc = pc;
31
32

```

148、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)!

i) 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

ii) 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。

149、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。

150、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

2) 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）；

3) 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

151、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virtual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vptra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vptra。

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

152、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。

153、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。

154、如何设计一个类计算子类的个数？

- 1、为类设计一个static静态变量count作为计数器；
- 2、类定义结束后初始化count；
- 3、在构造函数中对count进行+1；
- 4、设计拷贝构造函数，在进行拷贝构造函数中进行count +1，操作；
- 5、设计复制构造函数，在进行复制函数中对count+1操作；
- 6、在析构函数中对count进行-1；

155、怎么快速定位错误出现的地方

- 1、如果是简单的错误，可以直接双击错误列表里的错误项或者生成输出的错误信息中带行号的地方就可以让编辑窗口定位到错误的位置上。
- 2、对于复杂的模板错误，最好使用生成输出窗口。

多数情况下出发错误的位置是最靠后的引用位置。如果这样确定不了错误，就需要先把自己写的代码里的引用位置找出来，然后逐个分析了。

156、虚函数的代价？

- 1) 带有虚函数的类，每一个类会产生一个虚函数表，用来存储指向虚成员函数的指针，增大类；
- 2) 带有虚函数的类的每一个对象，都会有有一个指向虚表的指针，会增加对象的空间大小；
- 3) 不能再是内敛的函数，因为内敛函数在编译阶段进行替代，而虚函数表示等待，在运行阶段才能确定到底是采用哪种函数，虚函数不能是内敛函数。

157、类对象的大小受哪些因素影响？

- 1) 类的非静态成员变量大小，静态成员不占据类的空间，成员函数也不占据类的空间大小；
- 2) 内存对齐另外分配的空间大小，类内的数据也是需要进行内存对齐操作的；
- 3) 虚函数的话，会在类对象插入vptr指针，加上指针大小；
- 4) 当该类是某类的派生类，那么派生类继承的基类部分的数据成员也会存在在派生类中的空间中，也会对派生类进行扩展。

158、移动构造函数听说过吗？说说

- 1) 有时候我们会遇到这样一种情况，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；

3) C++引入了移动构造函数，专门处理这种，用a初始化b后，就将a析构的情况；

4) 与拷贝类似，移动也使用一个对象的值设置另一个对象的值。但是，又与拷贝不同的是，移动实现的是对象值真实的转移（源对象到目的对象）：源对象将丢失其内容，其内容将被目的对象占有。移动操作的发生的时候，是当移动值的对象是未命名的对象的时候。这里未命名的对象就是那些临时变量，甚至都不会有名称。典型的未命名对象就是函数的返回值或者类型转换的对象。使用临时对象的值初始化另一个对象值，不会要求对对象的复制：因为临时对象不会有其它使用，因而，它的值可以被移动到目的对象。做到这些，就要使用移动构造函数和移动赋值：当使用一个临时变量对象进行构造初始化的时候，调用移动构造函数。类似的，使用未命名的变量的值赋给一个对象时，调用移动赋值操作；

5)

```
1 Example6 (Example6&& x) : ptr(x.ptr)
2
3 {
4
5     x.ptr = nullptr;
6
7 }
8
9 // move assignment
10
11 Example6& operator= (Example6&& x)
12
13 {
14
15     delete ptr;
16
17     ptr = x.ptr;
18
19     x.ptr=nullptr;
20
21     return *this;
22
23 }
24
25 }
```

159、什么时候合成构造函数？都说一说，你知道的都说一下

1) 如果一个类没有任何构造函数，但他含有一个成员对象，该成员对象含有默认构造函数，那么编译器就为该类合成一个默认构造函数，因为不合成一个默认构造函数那么该成员对象的构造函数不能调用；

2) 没有任何构造函数的类派生自一个带有默认构造函数的基类，那么需要为该派生类合成一个构造函数，只有这样基类的构造函数才能被调用；

3) 带有虚函数的类，虚函数的引入需要进入虚表，指向虚表的指针，该指针是在构造函数中初始化的，所以没有构造函数的话该指针无法被初始化；

4) 带有一个虚基类的类

还有一点需要注意的是：

- 1) 并不是任何没有构造函数的类都会合成一个构造函数
- 2) 编译器合成出来的构造函数并不会显示设定类内的每一个成员变量

160、什么时候需要合成拷贝构造函数呢？

有三种情况会以一个对象的内容作为另一个对象的初值：

- 1) 对一个对象做显示的初始化操作，`X xx = x;`
- 2) 当对象被当做参数交给某个函数时；
- 3) 当函数传回一个类对象时；

- 1) 如果一个类没有拷贝构造函数，但是含有一个类类型的成员变量，该类型含有拷贝构造函数，此时编译器会为该类合成一个拷贝构造函数；
- 2) 如果一个类没有拷贝构造函数，但是该类继承自含有拷贝构造函数的基类，此时编译器会为该类合成一个拷贝构造函数；
- 3) 如果一个类没有拷贝构造函数，但是该类声明或继承了虚函数，此时编译器会为该类合成一个拷贝构造函数；
- 4) 如果一个类没有拷贝构造函数，但是该类含有虚基类，此时编译器会为该类合成一个拷贝构造函数；

161、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——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。

162、构造函数的执行顺序是什么？

- 1) 在派生类构造函数中，所有的虚基类及上一层基类的构造函数调用；
- 2) 对象的`vptr`被初始化；
- 3) 如果有成员初始化列表，将在构造函数体内扩展开来，这必须在`vptr`被设定之后才做；
- 4) 执行程序员所提供的代码；

163、一个类中的全部构造函数的扩展过程是什么？

- 1) 记录在成员初始化列表中的数据成员初始化操作会被放在构造函数的函数体内，并与成员的声明顺序为顺序；
- 2) 如果一个成员并没有出现在成员初始化列表中，但它有一个默认构造函数，那么默认构造函数必须被调用；

- 3) 如果class有虚表，那么它必须被设定初值；
- 4) 所有上一层的基类构造函数必须被调用；
- 5) 所有虚基类的构造函数必须被调用。

164、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。

165、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。
- 2) 执行效率不同
memcpy最高，strcpy次之，sprintf的效率最低。
- 3) 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。

166、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。
- 2) 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。

3) 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。

167、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`所指向的内存容量。
- 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。

168、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

169、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。

170、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。

171、main函数的返回值有什么值得考究之处吗？

程序运行过程入口点main函数，main（）函数返回值类型必须是int，这样返回值才能传递给程序激活者（如操作系统）表示程序正常退出。

main (int args, char **argv) 参数的传递。参数的处理，一般会调用getopt () 函数处理，但实践中，这仅仅是一部分，不会经常用到的技能点。

172、模板会写吗？写一个比较大小的模板函数

```
1 #include<iostream>
2
3 using namespace std;
4 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     cout<<"Max = "<<Max(5.5,'a')<<endl;
18
19 }
20
21
22 }
```

173、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。

174、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
```

2) `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。

3) 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;

175、`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

176、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, 但是一句一句的写太麻烦, 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。

177、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。

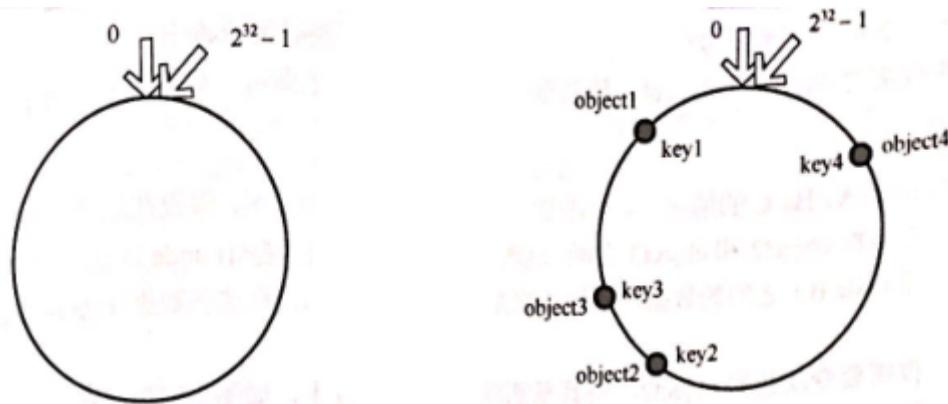
178、什么是一致性哈希?

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 $0 \sim 2^{32}-1$ ，整个哈希空间环如下左图所示

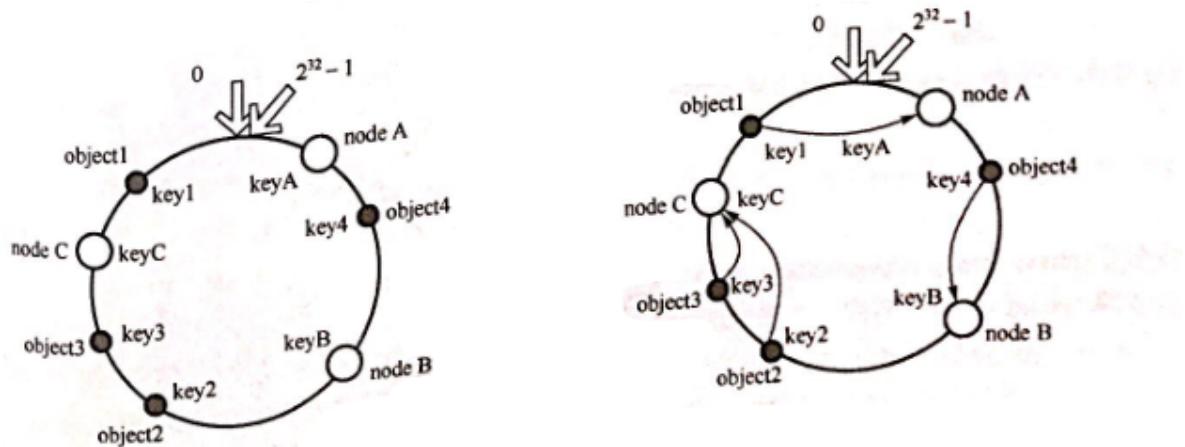


一致性hash的基本思想就是使用相同的hash算法将数据和结点都映射到图中的环形哈希空间中，上右图显示了4个数据object1-object4在环上的分布图

结点和数据映射

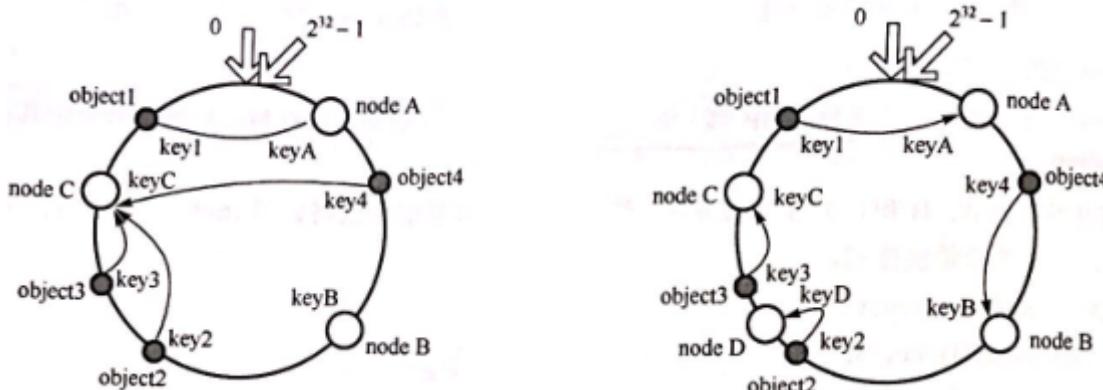
假如有一批服务器，可以根据IP或者主机名作为关键字进行哈希，根据结果映射到哈希环中，3台服务器分别是nodeA-nodeC

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

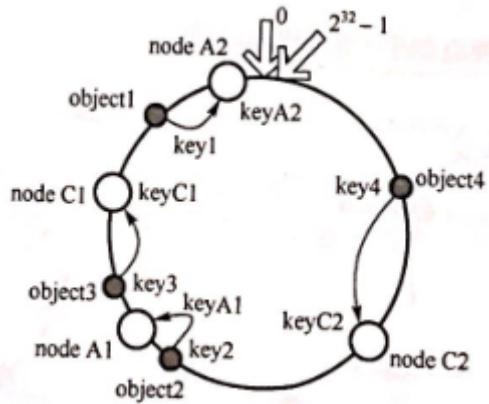


添加结点

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

179、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：

```
1 #include <iostream>
2 using namespace std;
3
4 class Base
5 {
6 public:
7     virtual void fun1()
8     {
9         cout << "普通虚函数" << endl;
10    }
```

```

10     }
11     virtual void fun2() = 0;
12     virtual ~Base() {}
13 };
14
15 class Son : public Base
16 {
17 public:
18     virtual void fun2()
19     {
20         cout << "子类实现的纯虚函数" << endl;
21     }
22 };
23
24 int main()
25 {
26     Base* b = new Son;
27     b->fun1(); //普通虚函数
28     b->fun2(); //子类实现的纯虚函数
29     return 0;
30 }
31
32
33

```

180、C++从代码到可执行程序经历了什么？

(1) 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时能够显示行号。

(2) 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。

(3) 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——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)。

(4) 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/93/675fd4af3ab34b2db0ae650855aa52d5>

181、为什么友元函数必须在类内部声明？

因为编译器必须能够读取这个结构的声明以理解这个数据类型的大、行为等方面的所有规则。

有一条规则在任何关系中都很重要，那就是谁可以访问我的私有部分。

182、用C语言实现C++的继承

```
1 #include <iostream>
2
3 using namespace std;
4
5 //C++中的继承与多态
6 struct A
7 {
8
9     virtual void fun() //C++中的多态：通过虚函数实现
10
11     {
12         cout<<"A:fun()"<<endl;
13     }
14
15     int a;
16
17 };
18
19
20 struct B:public A //C++中的继承：B类公有继承A类
21
22 {
23     virtual void fun() //C++中的多态：通过虚函数实现（子类的关键字virtual可加可不
24     加）
25
26     {
27
28         cout<<"B:fun()"<<endl;
29
30     }
31     int b;
32
33 };
34
35 //C语言模拟C++的继承与多态
36
37 typedef void (*FUN)(); //定义一个函数指针来实现对成员函数的继承
38
39 struct _A //父类
40 {
41
42     FUN _fun; //由于C语言中结构体不能包含函数，故只能用函数指针在外面实现
43     int _a;
44
45 };
46
47 struct _B //子类
48 {
49
50     _A _a_; //在子类中定义一个基类的对象即可实现对父类的继承
51 }
```

```

53     int _b;
54
55 }
56
57 void _fA() //父类的同名函数
58 {
59     printf("_A:_fun()\n");
60 }
61
62 void _fB() //子类的同名函数
63 {
64     printf("_B:_fun()\n");
65 }
66
67 void Test()
68 {
69
70 //测试C++中的继承与多态
71
72 A a; //定义一个父类对象a
73
74 B b; //定义一个子类对象b
75
76
77
78 //C语言模拟继承与多态的测试
79
80
81
82
83
84
85
86 A* p1 = &a; //定义一个父类指针指向父类的对象
87
88 p1->fun(); //调用父类的同名函数
89
90 p1 = &b; //让父类指针指向子类的对象
91
92 p1->fun(); //调用子类的同名函数
93
94
95
96
97
98 _A _a; //定义一个父类对象_a
99
100 _B _b; //定义一个子类对象_b
101
102 _a._fun = _fA; //父类的对象调用父类的同名函数
103
104 _b._a._fun = _fB; //子类的对象调用子类的同名函数
105
106
107
108 _A* p2 = &_a; //定义一个父类指针指向父类的对象
109
110 p2->_fun(); //调用父类的同名函数

```

```

111
112     p2 = (_A*)&_b; //让父类指针指向子类的对象,由于类型不匹配所以要进行强转
113
114     p2->_fun(); //调用子类的同名函数
115
116 }
117
118

```

183、动态编译与静态编译

- 1) 静态编译，编译器在编译可执行文件时，把需要用到的对应动态链接库中的部分提取出来，连接到可执行文件中去，使可执行文件在运行时不需要依赖于动态链接库；
- 2) 动态编译的可执行文件需要附带一个动态链接库，在执行时，需要调用其对应动态链接库的命令。所以其优点一方面是缩小了执行文件本身的体积，另一方面是加快了编译速度，节省了系统资源。缺点是哪怕是很简单的程序，只用到了链接库的一两条命令，也需要附带一个相对庞大的链接库；二是如果其他计算机上没有安装对应的运行库，则用动态编译的可执行文件就不能运行。

184、hello.c 程序的编译过程

以下是一个 hello.c 程序：

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
8
9
10

```

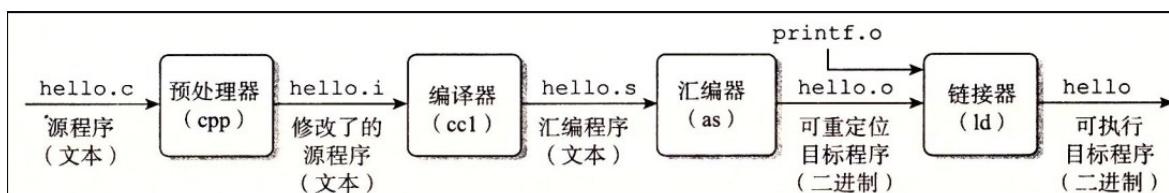
在 Unix 系统上，由编译器把源文件转换为目标文件。

```

1 gcc -o hello hello.c
2

```

这个过程大致如下：

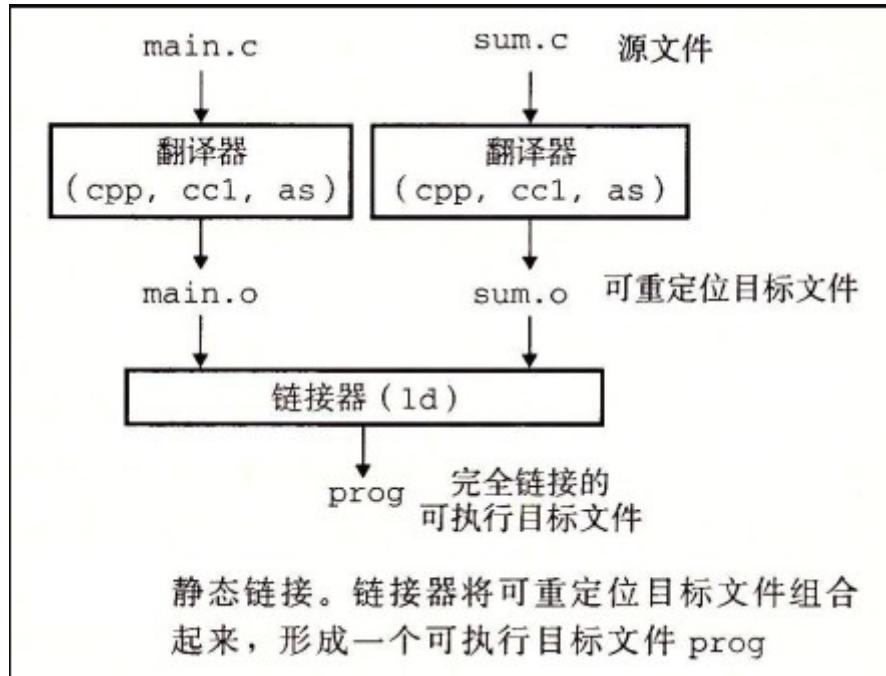


- 预处理阶段：处理以 # 开头的预处理命令；
- 编译阶段：翻译成汇编文件；
- 汇编阶段：将汇编文件翻译成可重定位目标文件；
- 链接阶段：将可重定位目标文件和 printf.o 等单独预编译好的目标文件进行合并，得到最终的可执行目标文件。

静态链接

静态链接器以一组可重定位目标文件为输入，生成一个完全链接的可执行目标文件作为输出。链接器主要完成以下两个任务：

- 符号解析：每个符号对应于一个函数、一个全局变量或一个静态变量，符号解析的目的是将每个符号引用与一个符号定义关联起来。
- 重定位：链接器通过把每个符号定义与一个内存位置关联起来，然后修改所有对这些符号的引用，使得它们指向这个内存位置。



目标文件

- 可执行目标文件：可以直接在内存中执行；
- 可重定位目标文件：可与其它可重定位目标文件在链接阶段合并，创建一个可执行目标文件；
- 共享目标文件：这是一种特殊的可重定位目标文件，可以在运行时被动态加载进内存并链接；

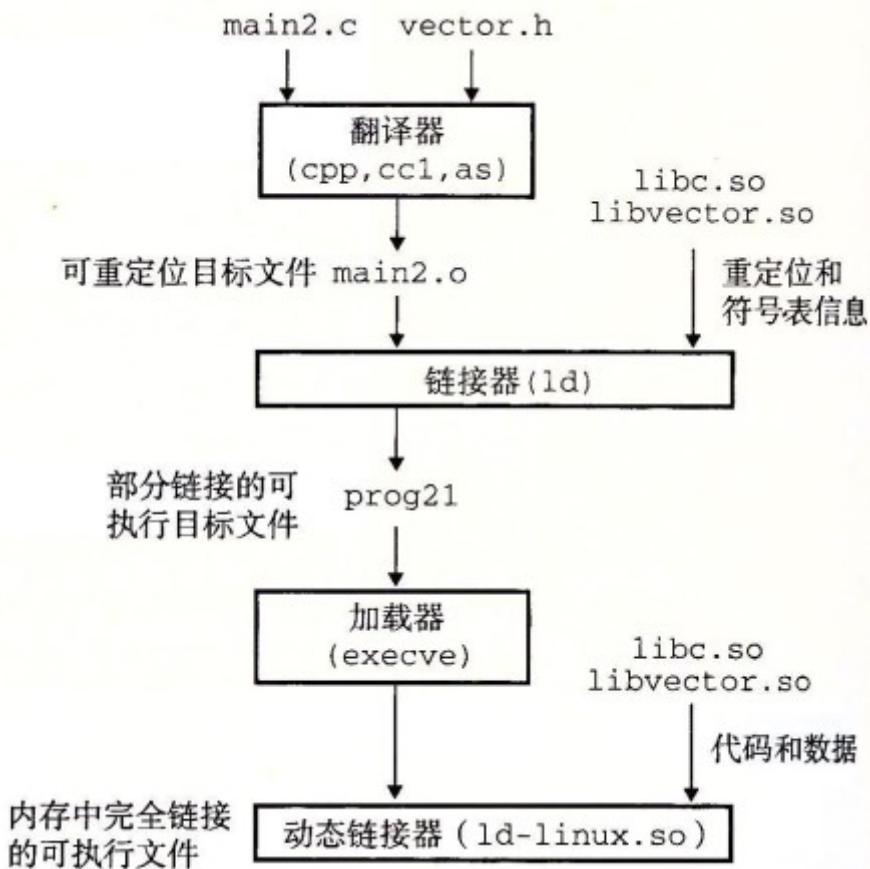
动态链接

静态库有以下两个问题：

- 当静态库更新时那么整个程序都要重新进行链接；
- 对于 printf 这种标准函数库，如果每个程序都要有代码，这会极大浪费资源。

共享库是为了解决静态库的这两个问题而设计的，在 Linux 系统中通常用 .so 后缀来表示，Windows 系统上它们被称为 DLL。它具有以下特点：

- 在给定的文件系统中一个库只有一个文件，所有引用该库的可执行目标文件都共享这个文件，它不会被复制到引用它的可执行文件中；
- 在内存中，一个共享库的 .text 节（已编译程序的机器代码）的一个副本可以被不同的正在运行的进程共享。



源代码 - -> 预处理 - -> 编译 - -> 优化 - -> 汇编 - -> 链接--> 可执行文件

1) 预处理

读取C源程序，对其中的伪指令（以#开头的指令）和特殊符号进行处理。包括宏定义替换、条件编译指令、头文件包含指令、特殊符号。预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。.i预处理后的c文件，.ii预处理后的C++文件。

2) 编译阶段

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。.s文件

3) 汇编过程

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个C语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。.o目标文件

4) 链接阶段

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。

185、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/article/details/80169978>

186、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。

举个例子：

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 template<typename T>
5 void fun(T&& t)
6 {
7     cout << t << endl;
8 }
9
10 int getInt()
11 {
12     return 5;
13 }
14
15 int main() {
16
17     int a = 10;
18     int& b = a; //b是左值引用
19     int& c = 10; //错误，c是左值不能使用右值初始化
20     int&& d = 10; //正确，右值引用用右值初始化
21     int&& e = a; //错误，e是右值引用不能使用左值初始化
22     const int& f = a; //正确，左值常引用相当于万能型，可以用左值或者右值初始化
```

```

23     const int& g = 10; //正确，左值常引用相当于万能型，可以用左值或者右值初始化
24     const int&& h = 10; //正确，右值常引用
25     const int& aa = h; //正确
26     int& i = getInt(); //错误，i是左值引用不能使用临时变量（右值）初始化
27     int&& j = getInt(); //正确，函数返回值是右值
28     fun(10); //此时fun函数的参数t是右值
29     fun(a); //此时fun函数的参数t是左值
30     return 0;
31 }
32
33
34

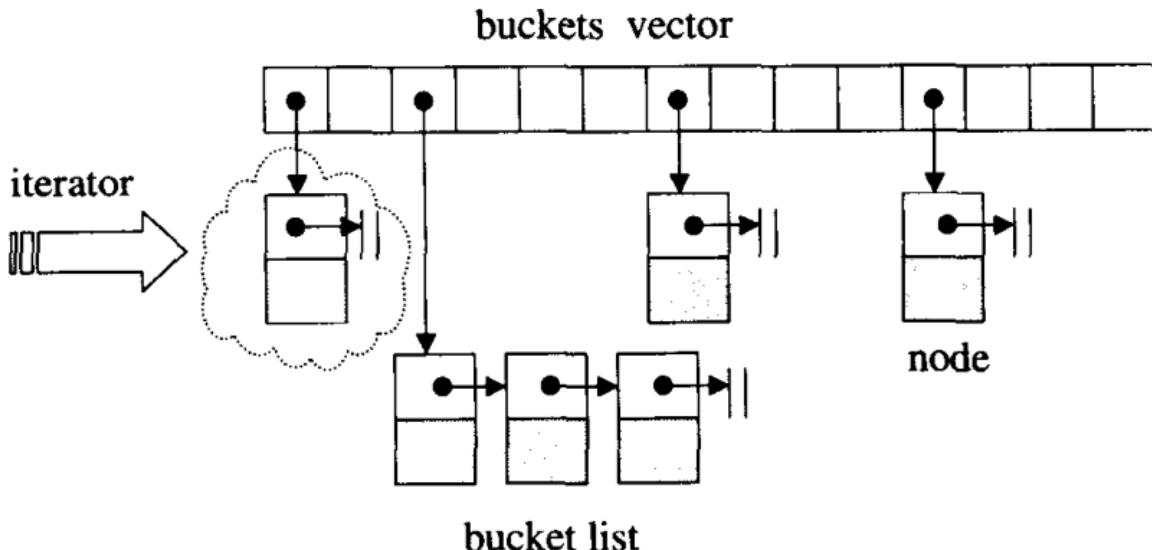
```

《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>

187、STL中hashtable的实现？

STL中的hashtable使用的是**开链法**解决hash冲突问题，如下图所示。



hashtable中的bucket所维护的list既不是list也不是slist，而是其自己定义的由hashtable_node数据结构组成的linked-list，而bucket聚合体本身使用vector进行存储。hashtable的迭代器只提供前进操作，不提供后退操作。

在hashtable设计bucket的数量上，其内置了28个质数[53, 97, 193,...,429496729]，在创建hashtable时，会根据存入的元素个数选择大于等于元素个数的质数作为hashtable的容量（vector的长度），其中每个bucket所维护的linked-list长度也等于hashtable的容量。如果插入hashtable的元素个数超过了bucket的容量，就要进行重建table操作，即找出下一个质数，创建新的buckets vector，重新计算元素在新hashtable的位置。

《STL源码解析》侯捷

188、简单说一下STL中的traits技法

traits技法利用“内嵌型别”的编程技巧与**编译器的template参数推导功能**，增强C++未能提供的关于型别认证方面的能力。常用的有iterator_traits和type_traits。

被称为**特性萃取机**，能够方便的让外界获取以下5中型别：

- value_type：迭代器所指对象的型别
- difference_type：两个迭代器之间的距离
- pointer：迭代器所指向的型别
- reference：迭代器所引用的型别
- iterator_category：三两句话说不清楚，建议看书

type_traits

关注的是型别的**特性**，例如这个型别是否具备non-trivial default ctor（默认构造函数）、non-trivial copy ctor（拷贝构造函数）、non-trivial assignment operator（赋值运算符）和non-trivial dtor（析构函数），如果答案是否定的，可以采取直接操作内存的方式提高效率，一般来说，type_traits支持以下5中类型的判断：

```
1 __type_traits<T>::has_trivial_default_constructor
2 __type_traits<T>::has_trivial_copy_constructor
3 __type_traits<T>::has_trivial_assignment_operator
4 __type_traits<T>::has_trivial_destructor
5 __type_traits<T>::is_POD_type
6
7
8
9
```

由于编译器只针对class object形式的参数进行参数推导，因此上式的返回结果不应该是bool值，实际上使用的是一个空的结构体：

```
1 struct __true_type{};
2 struct __false_type{};
3
4
5
6
```

这两个结构体没有任何成员，不会带来其他的负担，又能满足需求，可谓一举两得

当然，如果我们自行定义了一个Shape类型，也可以针对这个Shape设计type_traits的特化版本

```
1 template<> struct __type_traits<Shape>{
2     typedef __true_type has_trivial_default_constructor;
3     typedef __false_type has_trivial_copy_constructor;
4     typedef __false_type has_trivial_assignment_operator;
5     typedef __false_type has_trivial_destructor;
6     typedef __false_type is_POD_type;
7 };
8
9
10
```

《STL源码解析》侯捷 P103-P110

189、STL的两级空间配置器

1、首先明白为什么需要二级空间配置器？

我们知道动态开辟内存时，要在堆上申请，但若是我们需要

频繁的在堆开辟释放内存，则就会在堆上造成很多外部碎片，浪费了内存空间；

每次都要进行调用**malloc**、**free**函数等操作，使空间就会增加一些附加信息，降低了空间利用率；

随着外部碎片增多，内存分配器在找不到合适内存情况下需要合并空闲块，浪费了时间，大大降低了效率。

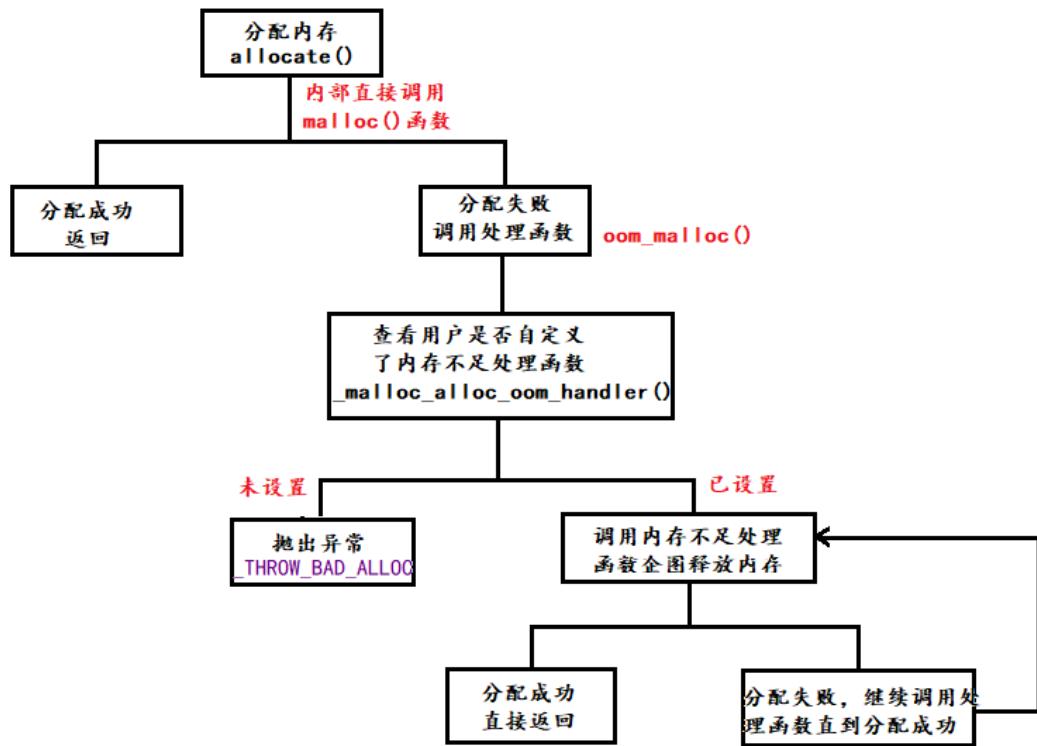
于是就设置了二级空间配置器，当开辟内存<=128bytes时，即视为开辟小块内存，则调用二级空间配置器。

关于STL中一级空间配置器和二级空间配置器的选择上，一般默认选择的为二级空间配置器。如果大于128字节再转去一级配置器器。

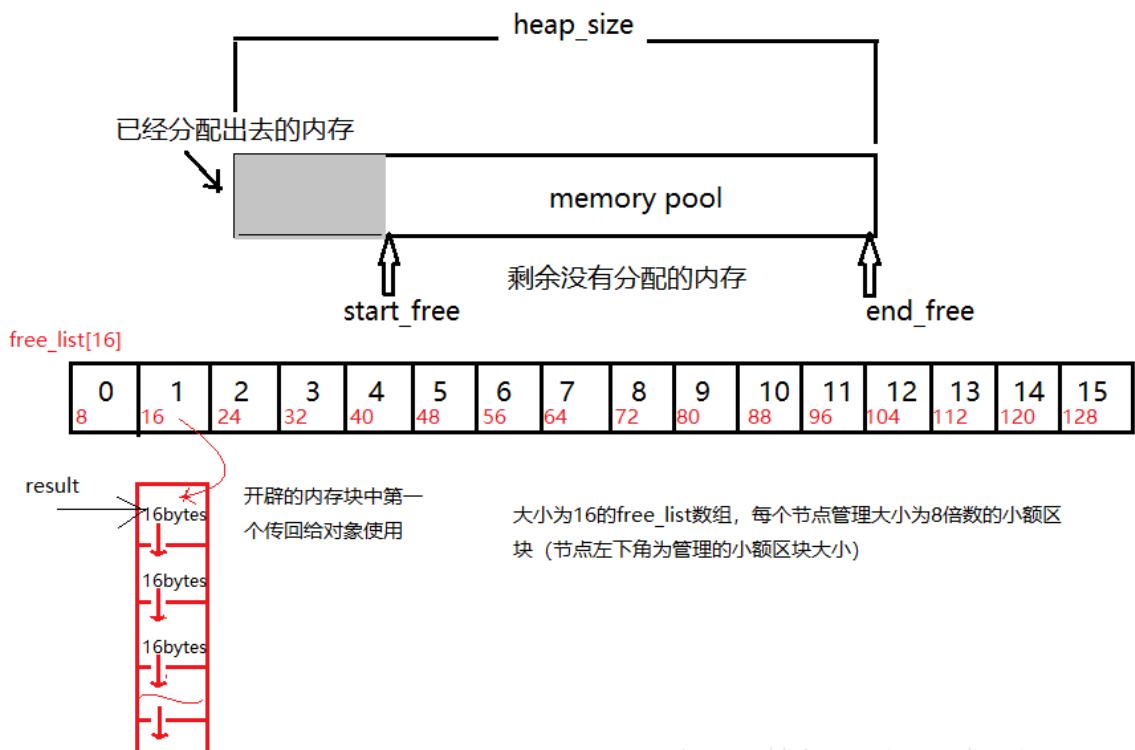
一级配置器

一级空间配置器中重要的函数就是**allocate**、**deallocate**、**reallocate**。一级空间配置器是以**malloc()**, **free()**, **realloc()**等C函数执行实际的内存配置。大致过程是：

- 1、直接**allocate**分配内存，其实就是**malloc**来分配内存，成功则直接返回，失败就调用处理函数
- 2、如果用户自定义了内存分配失败的处理函数就调用，没有的话就返回异常
- 3、如果自定义了处理函数就进行处理，完事再继续分配试试



二级配置器



1、维护16条链表，分别是0-15号链表，最小8字节，以8字节逐渐递增，最大128字节，你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（如需要13bytes空间，我们会给它分配16bytes大小），在找到第你个链表后查看链表是否为空，如果不为空直接从对应的free_list中拔出，将已经拔出的指针向后移动一位。

2、对应的free_list为空，先看其内存池是不是空时，如果内存池不为空：

(1) 先检验它剩余空间是否够20个节点大小（即所需内存大小(提升后) * 20），若足够则直接从内存池中拿出20个节点大小空间，将其中一个分配给用户使用，另外19个当作自由链表中的区块挂在相应的free_list下，这样下次再有相同大小的内存需求时，可直接拔出。

(2) 如果不够20个节点大小，则看它是否能满足1个节点大小，如果够的话则直接拿出一个分配给用户，然后从剩余的空间中分配尽可能多的节点挂在相应的free_list中。

(3) 如果连一个节点内存都不能满足的话，则将内存池中剩余的空间挂在相应的free_list中（找到相应的free_list），然后再给内存池申请内存，转到3。

3、内存池为空，申请内存

此时二级空间配置器会使用malloc()从heap上申请内存，（一次所申请的内存大小为 $2 * \text{所需节点内存大小 (提升后)} * 20 + \text{一段额外空间}$ ），申请40块，一半拿来用，一半放内存池中。

4、malloc没有成功

在第三种情况下，如果malloc()失败了，说明heap上没有足够空间分配给我们了，这时，二级空间配置器会从比所需节点空间大的free_list中一一搜索，从比它所需节点空间大的free_list中拔除一个节点来使用。如果这也没找到，说明比其大的free_list中都没有自由区块了，那就要调用一级适配器了。

释放时调用deallocate()函数，若释放的n>128，则调用一级空间配置器，否则就直接将内存块挂上自由链表的合适位置。

STL二级空间配置器虽然解决了外部碎片与提高了效率，但它同时增加了一些缺点：

1.因为自由链表的管理问题，它会把我们需求的内存块自动提升为8的倍数，这时若你需要1个字节，它会给你8个字节，即浪费了7个字节，所以它又引入了内部碎片的问题，若相似情况出现很多次，就会造成很多内部碎片；

2.二级空间配置器是在堆上申请大块的狭义内存池，然后用自由链表管理，供现在使用，在程序执行过程中，它将申请的内存一块一块都挂在自由链表上，即不会还给操作系统，并且它的实现中所有成员全是静态的，所以它申请的所有内存只有在进程结束才会释放内存，还给操作系统，由此带来的问题有：1.即我不断的开辟小块内存，最后整个堆上的空间都被挂在自由链表上，若我想开辟大块内存就会失败；2.若自由链表上挂很多内存块没有被使用，当前进程又占着内存不释放，这时别的进程在堆上申请不到空间，也不可以使用当前进程的空闲内存，由此就会引发多种问题。

一级分配器

GC4.9之后就没有第一级了，只有第二级

二级分配器

——default_alloc_template 剖析

有个自动调整的函数：你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（0-15号链表，最小8字节 最大128字节）

allocate函数：如果要分配的内存大于128字节，就转用第一级分配器，否则也就是小于128字节。那么首先判断落在第几号链表，定位到了，先判断链表是不是空，如果是空就需要充值，（调节到8的倍数，默认一次申请20个区块，当然了也要判断20个是不是能够申请到，如果只申请到一个那就直接返回好了，不止一个的话，把第2到第n个挨个挂到当前链表上，第一个返回回去给容器用，n是不大于20的，当然了如果不在1-20之间，那就是内存碎片了，那就先把碎片挂到某一条链表上，然后再重新malloc了，malloc 2*20个块）去内存池去拿或者重新分配。不为空的话

190、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行扩容。

2) 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，指向下一个元素的节点（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：

191、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
```

192、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);
```

2) 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++)
```

193、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者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 catagory;

194、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 $O(\log n)$ 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。

195、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。

2) 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。

196、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
```

197、STL中unordered_map(hash_map)和map的区别，hash_map如何解决冲突以及扩容

1) unordered_map和map类似，都是存储的key-value的值，可以通过key快速索引到value。不同的是unordered_map不会根据key的大小进行排序，

2) 存储时是根据key的hash值判断元素是否相同，即unordered_map内部元素是无序的，而map中的元素是按照二叉搜索树存储，进行中序遍历会得到有序遍历。

3) 所以使用时map的key需要定义operator<。而unordered_map需要定义hash_value函数并且重载operator==。但是很多系统内置的数据类型都自带这些，

4) 那么如果是自定义类型，那么就需要自己重载operator<或者hash_value()了。

5) 如果需要内部元素自动排序，使用map，不需要排序使用unordered_map

6) unordered_map的底层实现是hash_table；

7) hash_map底层使用的是hash_table，而hash_table使用的开链法进行冲突避免，所有hash_map采用开链法进行冲突解决。

8) **什么时候扩容：**当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即当前数组的长度乘以加载因子的值的时候，就要自动扩容啦。

9) 扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。

198、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。

2) 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。

3) 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。

199、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。

200、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。

201、STL中的allocator,deallocator

- 1) 第一级配置器直接使用malloc()、free()和realloc(), 第二级配置器视情况采用不同的策略：当配置区块超过128bytes时，视之为足够大，便调用第一级配置器；当配置器区块小于128bytes时，为了降低额外负担，使用复杂的内存池整理方式，而不再用一级配置器；
- 2) 第二级配置器主动将任何小额区块的内存需求量上调至8的倍数，并维护16个free-list，各自管理大小为8~128bytes的小额区块；
- 3) 空间配置函数allocate(), 首先判断区块大小，大于128就直接调用第一级配置器，小于128时就检查对应的free-list。如果free-list之内有可用区块，就直接拿来用，如果没有可用区块，就将区块大小调整至8的倍数，然后调用refill(), 为free-list重新分配空间；
- 4) 空间释放函数deallocate(), 该函数首先判断区块大小，大于128bytes时，直接调用一级配置器，小于128bytes就找到对应的free-list然后释放内存。

202、STL中hash_map扩容发生什么？

- 1) hash table表格内的元素称为桶 (bucket), 而由桶所链接的元素称为节点 (node), 其中存入桶元素的容器为stl本身很重要的一种序列式容器——vector容器。之所以选择vector为存放桶元素的基础容器，主要是因为vector容器本身具有动态扩容能力，无需人工干预。
- 2) 向前操作：首先尝试从目前所指的节点出发，前进一个位置（节点），由于节点被安置于list内，所以利用节点的next指针即可轻易完成前进操作，如果目前正巧是list的尾端，就跳至下一个bucket身上，那正是指向下一个list的头部节点。

203、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.
- 4.stack 底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时
- 5.queue 底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时（stack和queue其实是适配器,而不叫容器，因为是对容器的再封装）
- 6.priority_queue 的底层数据结构一般为vector为底层容器，堆heap为处理规则来管理底层容器实现
- 7.set 底层数据结构为红黑树，有序，不重复
- 8.multiset 底层数据结构为红黑树，有序，可重复
- 9.map 底层数据结构为红黑树，有序，不重复
- 10.multimap 底层数据结构为红黑树，有序，可重复
- 11.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

13.unordered_map 底层数据结构为hash表，无序，不重复

14.unordered_multimap 底层数据结构为hash表，无序，可重复

204、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。

205、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

206、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 {  
7     //在Linux + GCC下  
8     vector<int> res(2,0);  
9     cout << res.capacity() << endl; //2  
10    res.push_back(1);  
11    cout << res.capacity() << endl; //4  
12    res.push_back(2);  
13    res.push_back(3);  
14    cout << res.capacity() << endl; //8  
15    return 0;  
16  
17  
18     //在win 10 + vs2019下  
19     vector<int> res(2,0);  
20     cout << res.capacity() << endl; //2  
21     res.push_back(1);  
22     cout << res.capacity() << endl; //3  
23     res.push_back(2);  
24     res.push_back(3);  
25     cout << res.capacity() << endl; //6  
26  
27  
28 }
```

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

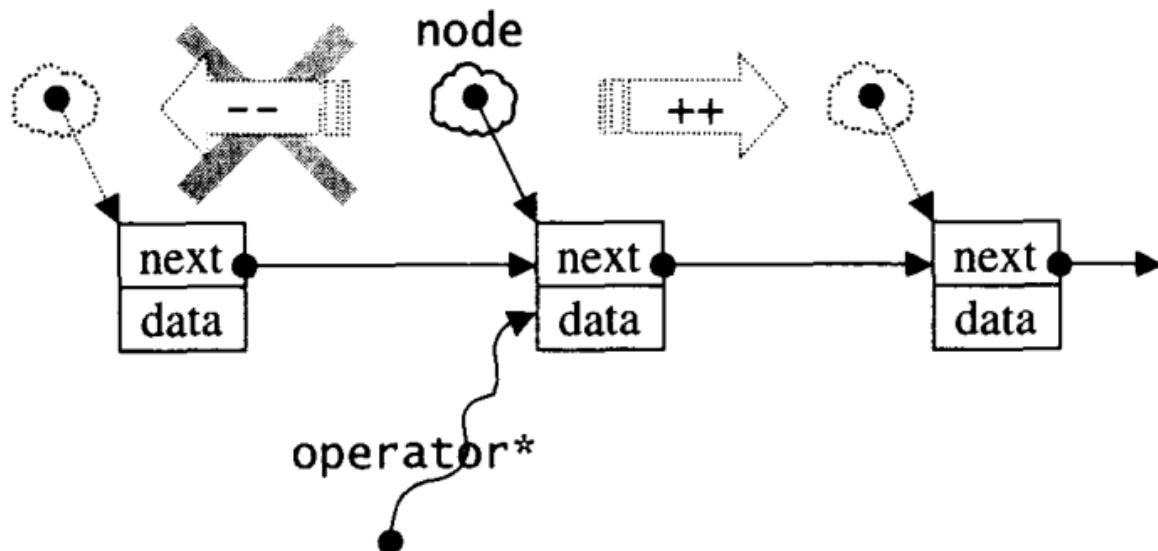
需要注意的是，频繁对vector调用push_back()对性能是有影响的，这是因为每插入一个元素，如果空间够用的话还能直接插入，若空间不够用，则需要重新配置空间，移动数据，释放原空间等操作，对程序性能会造成一定的影响

《STL源码剖析》侯捷 P115-128

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 Alloc = alloc>
2 class slist
3 {
4     ...
5 private:
6     ...
7     static list_node* create_node(const value_type& x){} //配置空间、构造元素
8     static void destroy_node(list_node* node){} //析构函数、释放空间
9 private:
10    list_node_base head; //头部
11 public:
12    iterator begin(){}
13    iterator end(){}
14    size_type size(){}
15    bool empty(){}
16    void swap(slist& L){} //交换两个slist，只需要换head即可
17    reference front(){}
18    void push_front(const value& x){} //头部插入元素
19    void pop_front(){}
20    ...
21 }
```

举个例子：

```

1 #include <forward_list>
2 #include <algorithm>
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     forward_list<int> f1;
9     f1.push_front(1);
10    f1.push_front(3);
11    f1.push_front(2);
12    f1.push_front(6);
13    f1.push_front(5);
14
15    forward_list<int>::iterator ite1 = f1.begin();
16    forward_list<int>::iterator ite2 = f1.end();
17    for(;ite1 != ite2; ++ite1)
18    {
19        cout << *ite1 << " "; // 5 6 2 3 1
20    }
21    cout << endl;
22
23    ite1 = find(f1.begin(), f1.end(), 2); //寻找2的位置
24
25    if (ite1 != ite2)
26        f1.insert_after(ite1, 99);
27    for (auto it : f1)
28    {
29        cout << it << " "; //5 6 2 99 3 1
30    }
31    cout << endl;
32
33    ite1 = find(f1.begin(), f1.end(), 6); //寻找6的位置
34    if (ite1 != ite2)
35        f1.erase_after(ite1);
36    for (auto it : f1)
37    {
38        cout << it << " "; //5 6 99 3 1
39    }
40    cout << endl;
41    return 0;
42 }
43
44
45
46

```

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

《STL源码剖析》侯捷

208、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

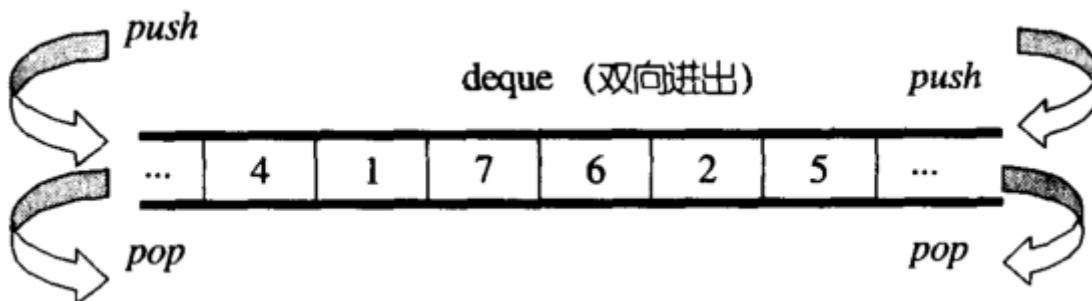
list的空间管理默认采用alloc作为空间配置器，为了方便的以节点大小为配置单位，还定义一个list_node_allocator函数可一次性配置多个节点空间

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

209、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

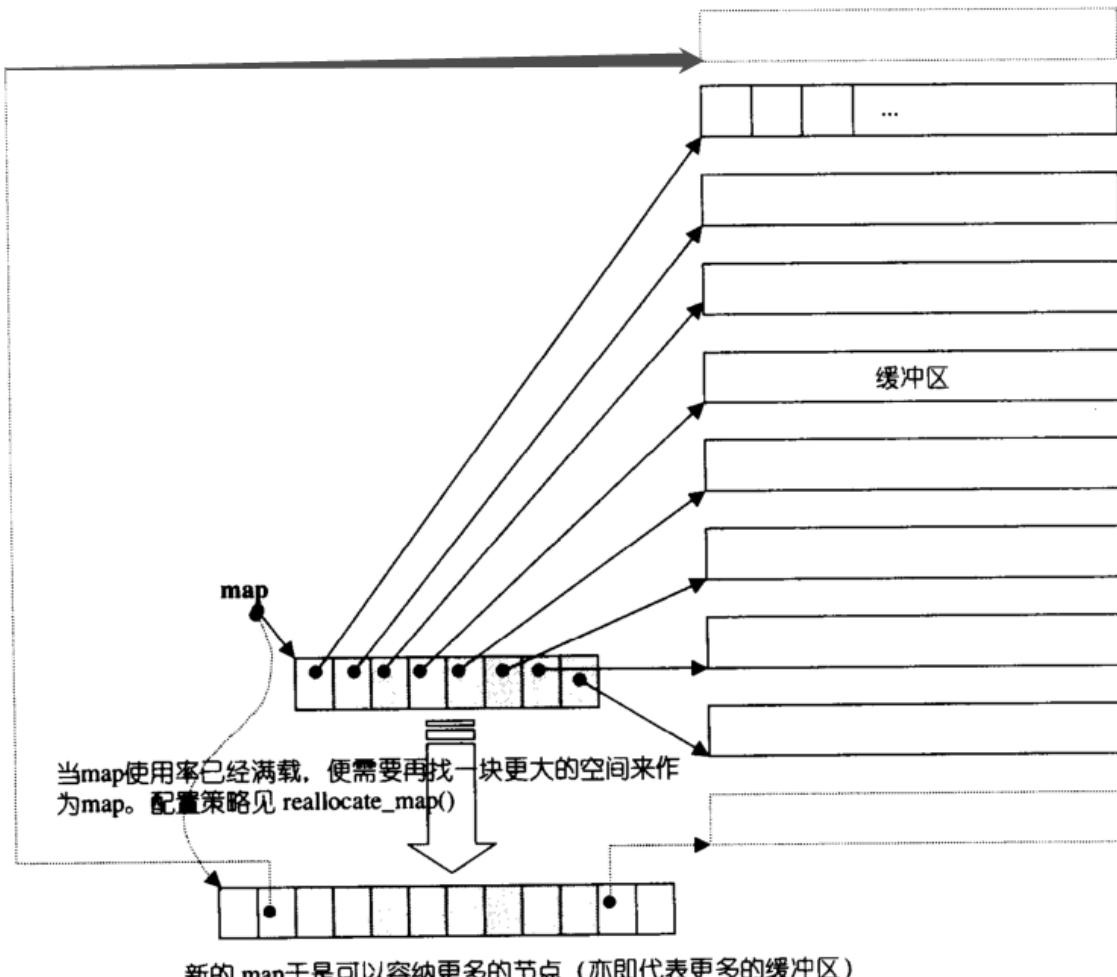
deque由一段一段的定量连续空间组成，一旦需要增加新的空间，只要配置一段定量连续空间拼接在头部或尾部即可，因此deque的最大任务是如何维护这个整体的连续性

deque的数据结构如下：

```

1 class deque
2 {
3     ...
4 protected:
5     typedef pointer* map_pointer;//指向map指针的指针
6     map_pointer map;//指向map
7     size_type map_size;//map的大小
8 public:
9     ...
10    iterator begin();
11    iterator 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。

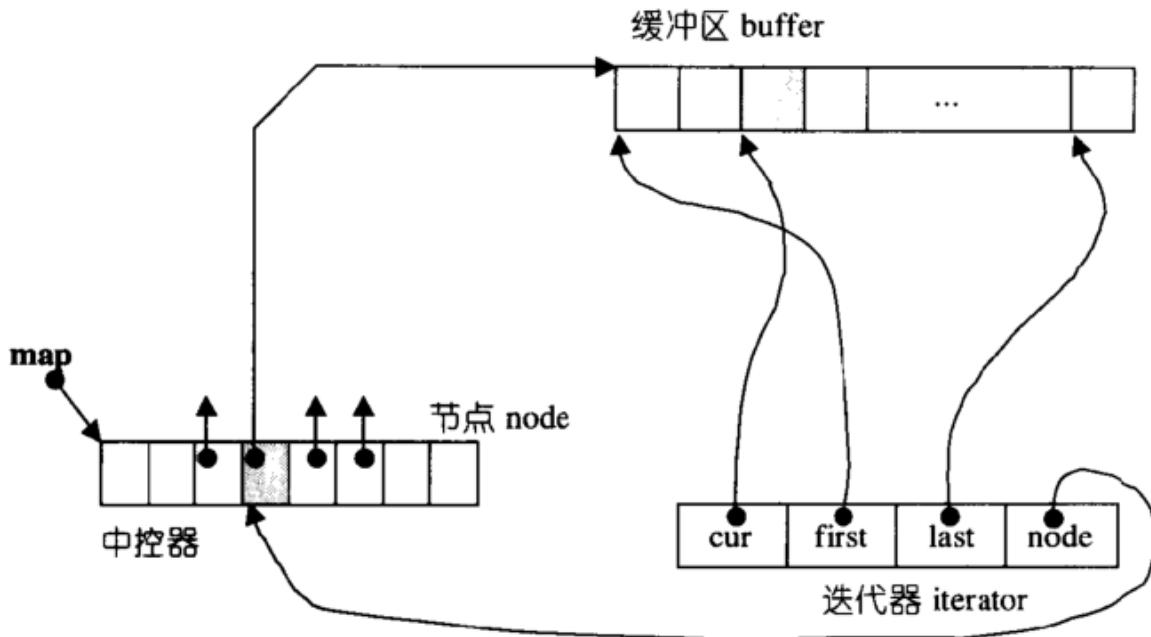
deque的迭代器数据结构如下：

```

1 struct __deque_iterator
2 {
3     ...
4     T* cur; //迭代器所指缓冲区当前的元素
5     T* first; //迭代器所指缓冲区第一个元素
6     T* last; //迭代器所指缓冲区最后一个元素
7     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



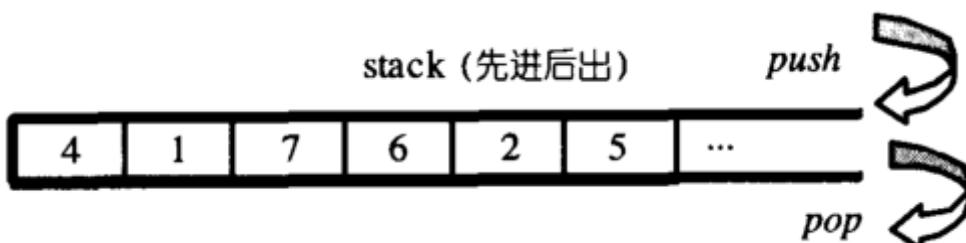
deque迭代器的“++”、“--”操作是远比vector迭代器繁琐，其主要工作在于缓冲区边界，如何从当前缓冲区跳到另一个缓冲区，当然deque内部在插入元素时，如果map中node数量全部使用完，且node指向的缓冲区也没有多余的空间，这时会配置新的map（2倍于当前+2的数量）来容纳更多的node，也就是可以指向更多的缓冲区。在deque删除元素时，也提供了元素的析构和空闲缓冲区空间的释放等机制。

《STL源码剖析》侯捷 P143-164

210、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：

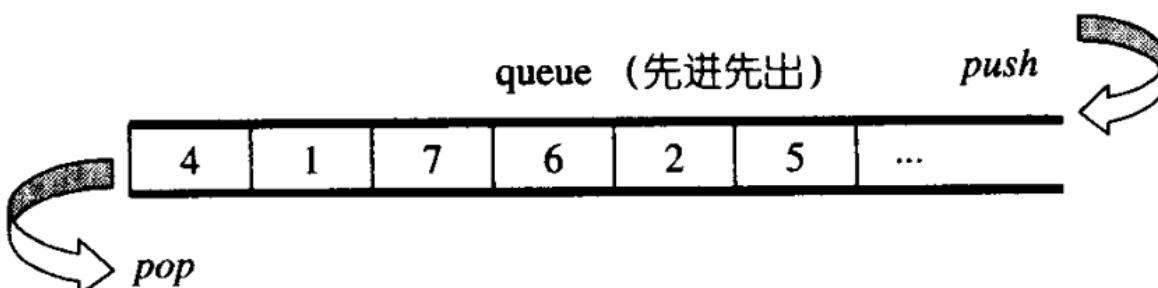
```
1 template <class T, class Sequence = deque<T> >
2 class stack
3 {
4     ...
5 protected:
6     Sequence c;
7 public:
8     bool empty(){return c.empty();}
9     size_type size() const{return c.size();}
10    reference top() const {return c.back();}
11    const_reference top() const{return c.back();}
12    void push(const value_type& x){c.push_back(x);}
13    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：

```
1 template <class T, class Sequence = deque<T> >
2 class queue
3 {
4     ...
5 protected:
6     Sequence c;
7 public:
8     bool empty(){return c.empty();}
9     size_type size() const{return c.size();}
10    reference front() const {return c.front();}
11    const_reference front() const{return c.front();}
```

```

12     void push(const value_type& x){c.push_back(x);}
13     void pop(){c.pop_front();}
14 };
15
16
17

```

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

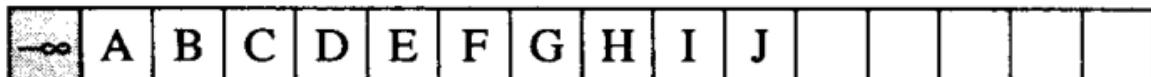
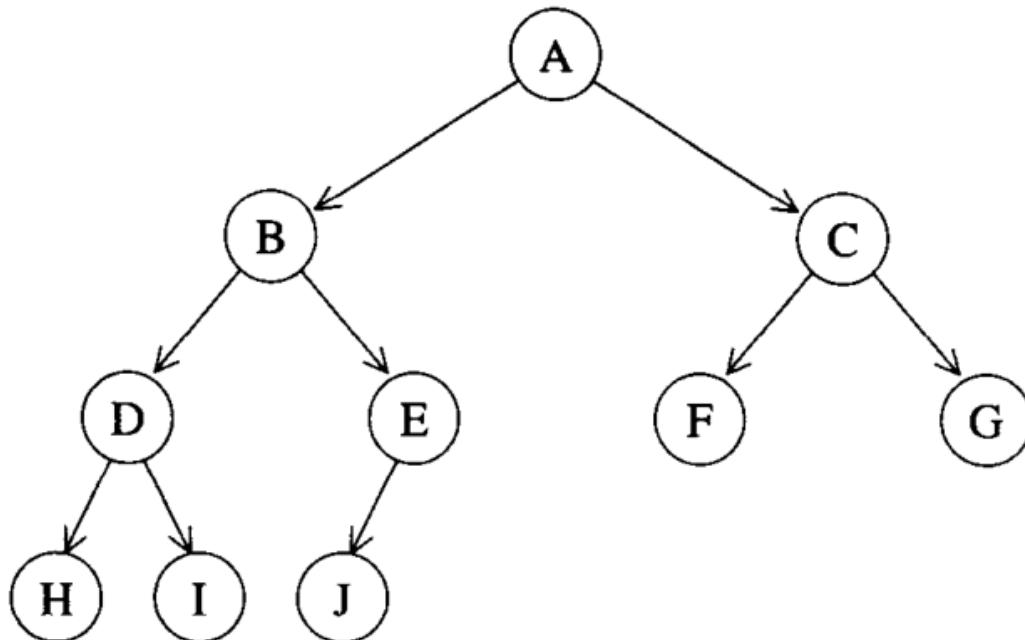
同样，queue也可以使用list作为底层容器，不具有遍历功能，没有迭代器。

《STL源码剖析》侯捷

211、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



某种实现技巧

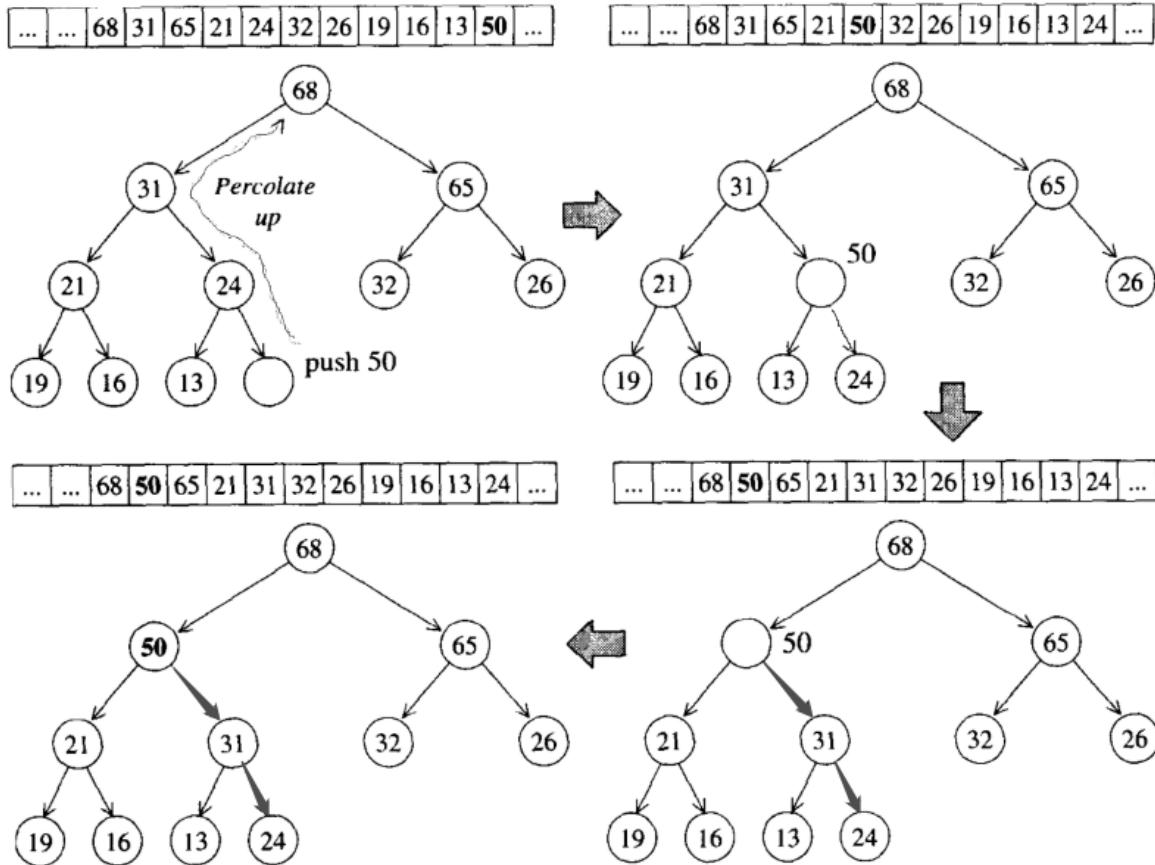
完全二叉树内没有任何节点漏洞，是非常紧凑的，这样的一个好处是可以使用array来存储所有的节点，因为当其中某个节点位于 i 处，其左节点必定位于 $2i$ 处，右节点位于 $2i+1$ 处，父节点位于 $i/2$ （向下取整）处。这种以array表示tree的方式称为隐式表述法。

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

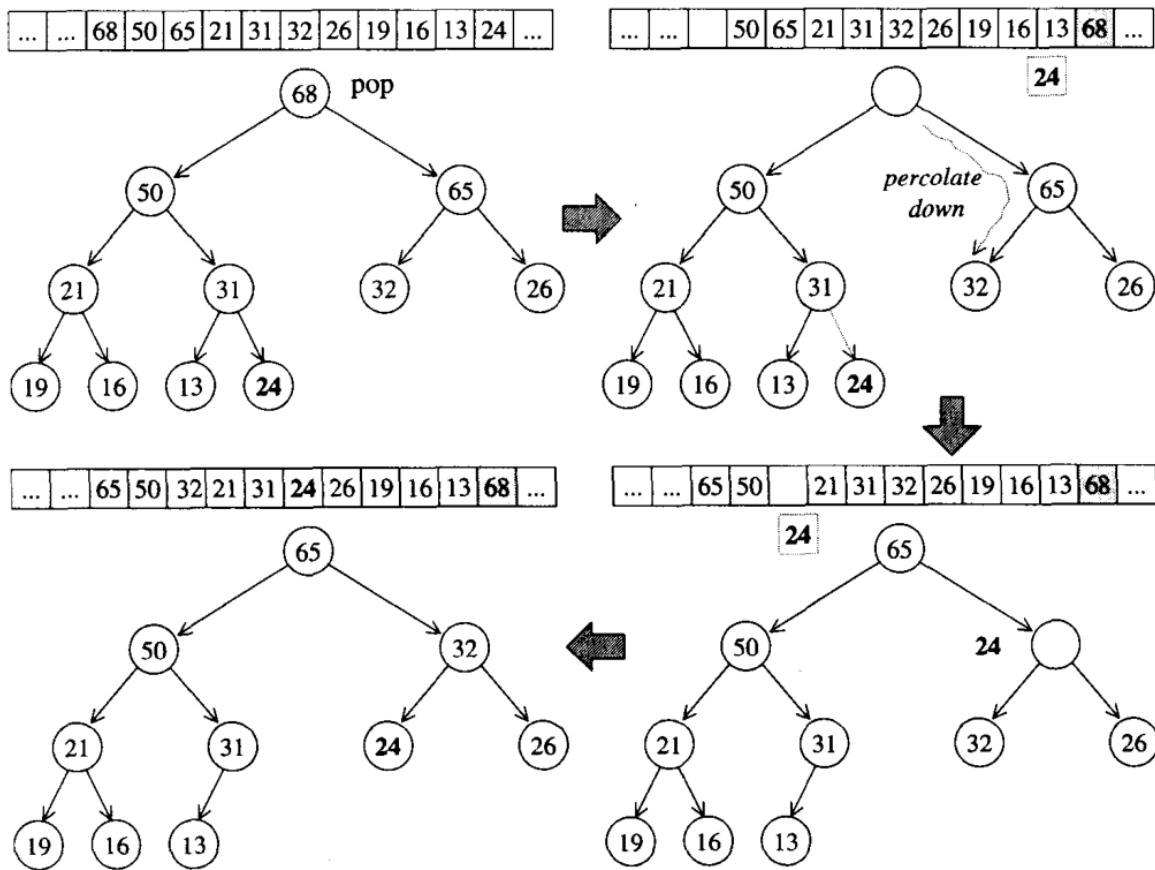
由于完全二叉树的性质，新插入的元素一定是位于树的最底层作为叶子节点，并填补由左至右的第一个空格。事实上，在刚执行插入操作时，新元素位于底层vector的end()处，之后是一个称为percolate up（上溯）的过程，举个例子如下图：



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：

```

1 #include <iostream>
2 #include <algorithm>
3 #include <vector>
4 using namespace std;
5
6 int main()
7 {
8     vector<int> v = { 0,1,2,3,4,5,6 };
9     make_heap(v.begin(), v.end()); //以vector为底层容器
10    for (auto i : v)
11    {
12        cout << i << " "; // 6 4 5 3 1 0 2
13    }
14    cout << endl;
15    v.push_back(7);
16    push_heap(v.begin(), v.end());
17    for (auto i : v)
18    {

```

```

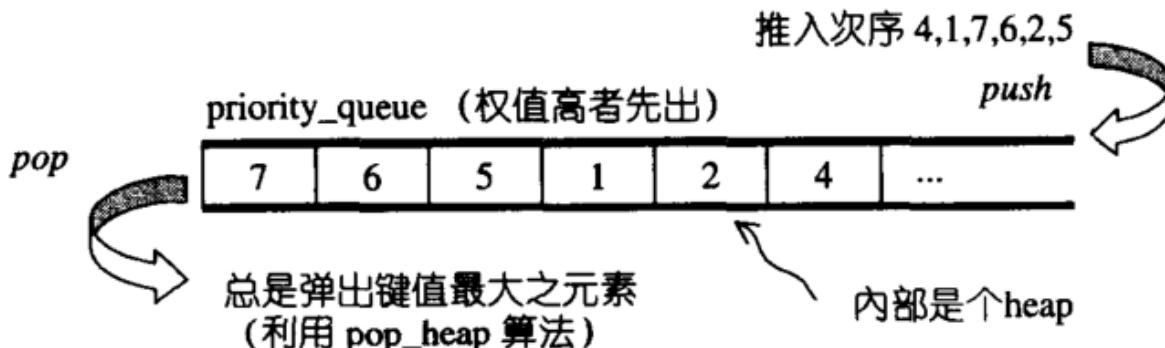
19         cout << i << " "; // 7 6 5 4 1 0 2 3
20     }
21     cout << endl;
22     pop_heap(v.begin(), v.end());
23     cout << v.back() << endl; // 7
24     v.pop_back();
25     for (auto i : v)
26     {
27         cout << i << " "; // 6 4 5 3 1 0 2
28     }
29     cout << endl;
30     sort_heap(v.begin(), v.end());
31     for (auto i : v)
32     {
33         cout << i << " "; // 0 1 2 3 4 5 6
34     }
35     return 0;
36 }
37
38
39

```

《STL源码剖析》侯捷

212、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：

```

1 template <class T, class Sequence = vector<T>,
2 class Compare = less<typename Sequence::value_type> >
3 class priority_queue{
4     ...
5 protected:
6     Sequence c; // 底层容器
7     Compare comp; // 元素大小比较标准
8 public:
9     bool empty() const {return c.empty();}
10    size_type size() const {return c.size();}
11    const_reference top() const {return c.front();}


```

```

12     void push(const value_type& x)
13     {
14         c.push_heap(x);
15         push_heap(c.begin(), c.end(), comp);
16     }
17     void pop()
18     {
19         pop_heap(c.begin(), c.end(), comp);
20         c.pop_back();
21     }
22 };
23
24
25
26

```

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};
8     priority_queue<int> pq(ia, ia + 9);
9     cout << pq.size() << endl; // 9
10    for(int i = 0; i < pq.size(); i++)
11    {
12        cout << pq.top() << " "; // 8 8 8 8 8 8 8 8
13    }
14    cout << endl;
15    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

213、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 };
10    set<int> s(ia, ia + 5);
11    cout << s.size() << endl; // 5
12    cout << s.count(3) << endl; // 1
13    cout << s.count(10) << endl; // 0
14
15    s.insert(3); //再插入一个3
16    cout << s.size() << endl; // 5
17    cout << s.count(3) << endl; // 1
18
19    s.erase(1);
20    cout << s.size() << endl; // 4
21
22    set<int>::iterator b = s.begin();
23    set<int>::iterator e = s.end();
24    for (; b != e; ++b)
25        cout << *b << " "; // 2 3 4 5
26    cout << endl;
27
28    b = find(s.begin(), s.end(), 5);
29    if (b != s.end())
30        cout << "5 found" << endl; // 5 found
31
32    b = s.find(2);
33    if (b != s.end())
34        cout << "2 found" << endl; // 2 found
35
36    b = s.find(1);
37    if (b == s.end())
38        cout << "1 not found" << endl; // 1 not found
39    return 0;
40 }
```

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。

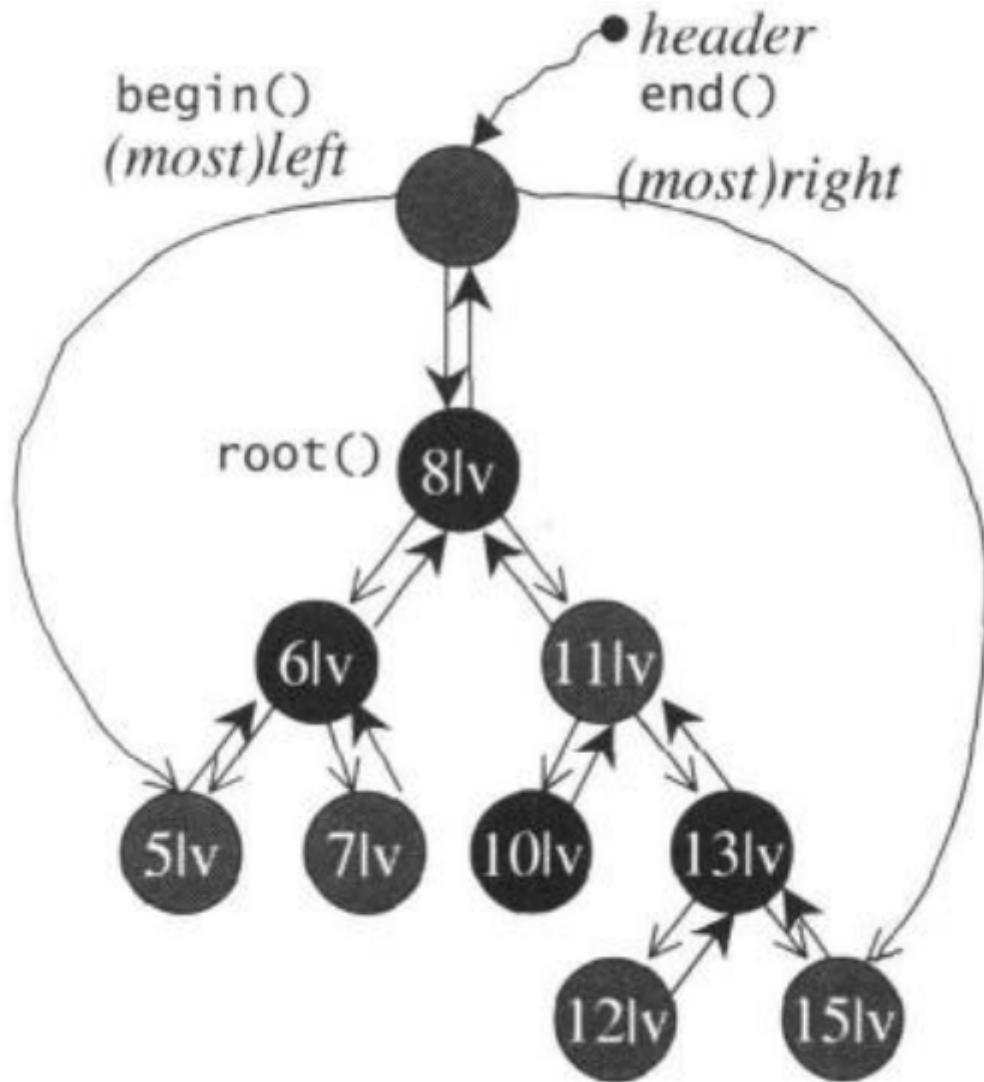
《STL源码剖析》侯捷

214、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_equal() (multimap使用)

举个例子：

```
1 #include <map>
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
```

```

6 int main()
7 {
8     map<string, int> maps;
9     //插入若干元素
10    maps["jack"] = 1;
11    maps["jane"] = 2;
12    maps["july"] = 3;
13    //以pair形式插入
14    pair<string, int> p("david", 4);
15    maps.insert(p);
16    //迭代输出元素
17    map<string, int>::iterator iter = maps.begin();
18    for (; iter != maps.end(); ++iter)
19    {
20        cout << iter->first << " ";
21        cout << iter->second << "--"; //david 4--jack 1--jane 2--july 3--
22    }
23    cout << endl;
24    //使用subscript操作取实值
25    int num = maps["july"];
26    cout << num << endl; // 3
27    //查找某key
28    iter = maps.find("jane");
29    if(iter != maps.end())
30        cout << iter->second << endl; // 2
31    //修改实值
32    iter->second = 100;
33    int num2 = maps["jane"]; // 100
34    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 = maps["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

215、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 $O(\log n)$ 。

216、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 $O(\log n)$ ，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

217、hashtable中解决冲突有哪些方法？

记住前三个：

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

3.2、数据结构与算法

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。

我仅在这章节中为大家盘点一下互联网大厂面试考察频率比较高的几道手撕算法题，希望我的整理对大家有一点点用处，那我就很高兴了！。

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/>

```
1 #include <iostream>
2 using namespace std;
3
4 struct myList {
5     int val;
6     myList* next;
7     myList(int _val) :val(_val), next(nullptr) {}
8 };
9
10 myList* merge(myList* l1, myList* l2) {
11
12     if (l1 == nullptr) return l2;
13     if (l2 == nullptr) return l1;
14     myList head(0);
15     myList* node = &head;
16     while (l1 != nullptr && l2 != nullptr) {
17         if (l1->val < l2->val) {
18             node->next = l1;
19             l1 = l1->next;
20
21         }
22         else {
23             node->next = l2;
24             l2 = l2->next;
25         }
26         node = node->next;
27     }
28
29     if (l1 == nullptr)
30         node->next = l2;
31     if (l2 == nullptr)
32         node->next = l1;
33
34     return head.next;
35
36 };
37
38 int main(void) {
39
40     myList* node0 = new myList(0);
41     myList* node1 = new myList(1);
42     myList* node2 = new myList(2);
43     myList* node3 = new myList(3);
44
45     myList* node4 = new myList(1);
46     myList* node5 = new myList(4);
47     node0->next = node1;
```

```

48     node1->next = node2;
49     node2->next = node3;
50     node3->next = nullptr;
51     node4->next = node5;
52     node5->next = nullptr;
53
54     auto node = merge(node0, node4);
55     while (node != nullptr) {
56         cout << node->val << endl;
57         node = node->next;
58     }
59
60     return 0;
61 }
```

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

```

1 #include<algorithm>
2 #include<unordered_map>
3 #include <iostream>
4 #include<vector>
5
6 using namespace std;
7
8 struct node {
9     int data;
10    struct node* next;
11    node(int _data) :data(_data), next(nullptr) {
12    }
13};
14
15 struct node* init() {
16     node* head = new node(1);
17     node* node1 = new node(2);
18     node* node2 = new node(3);
19     node* node3 = new node(4);
20     node* node4 = new node(5);
21
22     head->next = node1;
23     node1->next = node2;
24     node2->next = node3;
25     node3->next = node4;
26     node4->next = nullptr;
27
28     return head;
29 }
30
```

```

31 struct node* reverse(node* head) {
32     struct node* pre = new node(-1);
33     struct node* temp = new node(-1);
34     pre = head;
35     temp = head->next;
36     pre->next = nullptr;
37     struct node* cur = new node(-1);
38     cur = temp;
39     while (cur != nullptr) {
40         temp = cur;
41         cur = cur->next;
42         temp->next = pre;
43         pre = temp;
44     }
45
46     return pre;
47 }
48
49 int main(){
50     auto head = init();
51     head = reverse(head);
52     while (head != nullptr) {
53         cout << head->data << endl;
54         head = head->next;
55     }
56
57     return 0;
58 }
```

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
4     // 头插法
5     if (pHead == nullptr || pHead->next == nullptr) return pHead;
6     ListNode dummyNode = ListNode(0);
7     ListNode* pre = &(dummyNode);
8     pre->next = pHead;
9     ListNode* cur = pHead->next;
10    pHead->next = nullptr;
11    //pre = cur;
12    ListNode* temp = nullptr;
13    while (cur != nullptr) {
14        temp = cur;
15        cur = cur->next;
16        temp->next = pre->next;
17        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* instance();
7
8         class CG {
9             public:
10            ~CG() {
11                if (singlePattern::p != nullptr) {
12                    delete singlePattern::p;
13                    singlePattern::p = nullptr;
14                }
15            }
16        };
17    };
18
19 singlePattern* singlePattern::p = new singlePattern();
20 singlePattern* singlePattern::instance() {
21     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() {
10                     if (singlePattern::p != nullptr) {
11                         delete singlePattern::p;
12                         singlePattern::p = nullptr;
13                     }
14                 }
15             };
16         };
17         singlePattern* singlePattern::p = nullptr;
18         singlePattern* singlePattern::instance() {
19             if (p == nullptr) {
20                 return new singlePattern();
21             }
22             return p;
23 }
```

4、简单工厂模式

```
2     TypeA,
3     TypeB,
4     TypeC
5 } productTypeTag;
6
7 class Product {
8
9 public:
10    virtual void show() = 0;
11    virtual ~Product() = 0;
12};
13
14 class ProductA :public Product {
15 public:
16    void show() {
17        cout << "ProductA" << endl;
18    }
19    ~ProductA() {
20        cout << "~ProductA" << endl;
21    }
22};
23
24 class ProductB :public Product {
25 public:
26    void show() {
27        cout << "ProductB" << endl;
28    }
29    ~ProductB() {
30        cout << "~ProductB" << endl;
31    }
32};
33
34 class ProductC :public Product {
35 public:
36    void show() {
37        cout << "ProductC" << endl;
38    }
39    ~ProductC() {
40        cout << "~ProductC" << endl;
41    }
42};
43
44 class Factory {
45
46 public:
47    Product* createProduct(productType type) {
48        switch (type) {
49        case TypeA:
50            return new ProductA();
51        case TypeB:
52            return new ProductB();
53        case TypeC:
54            return new ProductC();
55        default:
56            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     //"; });
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
12        while (begin<end && data[begin]<= key) {
13            begin++;
14        }
15        if (begin < end) data[end--] = data[begin];
16
17    }
18
19
20    data[begin] = key;
21    quickSort(data, low, begin - 1);
22    quickSort(data, begin + 1,high);
23 }
```

6、归并排序

```
1 void mergeSort(vector<int>& data, vector<int>& copy, int begin, int end) {
2     if (begin >= end) return;
3     int mid = begin + (end - begin) / 2;
4     int low1 = begin, high1 = mid, low2 = mid + 1, high2 = end, index =
begin;
5     mergesort(copy, data, low1, high1);
6     mergesort(copy, data, low2, high2);
7     while (low1 <= high1 && low2 <= high2) {
8
9         copy[index++] = data[low1] < data[low2] ? data[low1++] :
data[low2++];
10    }
11    while (low1 <= high1) {
12        copy[index++] = data[low1++];
13    }
14
15    while (low2 <= high2) {
16        copy[index++] = data[low2++];
17    }
18
19
20 void mergeTest() {
21     vector<int> nums = { -5, -10, 6, 5, 12, 96, 1, 2, 3 };
```

```

22     vector<int> copy(nums);
23     mergeSort(nums, copy, 0, nums.size() - 1);
24     nums.assign(copy.begin(), copy.end());
25     for_each(nums.begin(), nums.end(), [](const auto& a) {cout << a << " ";
26 });

```

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
3 cache.put(1, 1);
4 cache.put(2, 2);
5 cache.get(1);      // 返回 1
6 cache.put(3, 3);    // 该操作会使得密钥 2 作废
7 cache.get(2);      // 返回 -1 (未找到)
8 cache.put(4, 4);    // 该操作会使得密钥 1 作废
9 cache.get(1);      // 返回 -1 (未找到)
10 cache.get(3);     // 返回 3
11 cache.get(4);     // 返回 4

```

链接：<https://leetcode-cn.com/problems/lru-cache-lcci>

```

1 struct DoubleList {
2     int key, val;
3     DoubleList* pre, * next;
4     DoubleList(int _key, int
5     _val):key(_key),val(_val),pre(nullptr),next(nullptr){ }
6 };
7
8 class LRU {
9 private:
10     int capacity;
11     DoubleList* head, * tail;
12     unordered_map<int, DoubleList*> memory;
13 public:
14     LRU(int _capacity) {
15         this->capacity = _capacity;
16         head = new DoubleList(-1, -1);
17         tail = new DoubleList(-1, -1);
18         head->next = tail;
19         tail->pre = head;
20     }
21     ~LRU(){
22         if (head != nullptr) {

```

```

22         delete head;
23         head = nullptr;
24     }
25     if (tail != nullptr) {
26         delete tail;
27         tail = nullptr;
28     }
29     for (auto& a : memory) {
30         if (a.second != nullptr) {
31             delete a.second;
32             a.second = nullptr;
33         }
34     }
35 }
36 void set(int _key, int _val) {
37     if (memory.find(_key) != memory.end()) {
38         DoubleList* node = memory[_key];
39         removeNode(node);
40         node->val = _val;
41         pushNode(node);
42         return ;
43     }
44     if (memory.size() == this->capacity) { // 这里很重要，也很爱错，千万记得更新memory
45         int topKey = head->next->key; // 取得key值，方便在后面删除
46         removeNode(head->next); // 移除头部的下一个
47         memory.erase(topKey); // 在memory中删除当前头部的值
48     }
49     DoubleList* node = new DoubleList(_key, _val); // 新增node
50     pushNode(node); // 放在尾部
51     memory[_key] = node; // 记得在memory中添加进去
52 }
53 int get(int _key) {
54     if (memory.find(_key) != memory.end()) {
55         DoubleList* node = memory[_key];
56         removeNode(node);
57         pushNode(node);
58         return node->val;
59     }
60     return -1;
61 }
62
63 void removeNode(DoubleList* node) {
64     node->pre->next = node->next;
65     node->next->pre = node->pre;
66 }
67 void pushNode(DoubleList* node) {
68     tail->pre->next = node;
69     node->pre = tail->pre;
70     node->next = tail;
71     tail->pre = node;
72 }
73 };

```

8、重排链表

给定一个单链表 $L: L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L^{**n-1} \rightarrow L_n$ ，
将其重新排列后变为： $L_0 \rightarrow L^{**n} \rightarrow L_1 \rightarrow L^{**n-1} \rightarrow L_2 \rightarrow L^{**n-2} \rightarrow \dots$

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/>

```
1 | #include<iostream>  
2 | #include<string>  
3 | #include<vector>  
4 | #include<algorithm>  
5 | #include<unordered_map>  
6 |  
7 | using namespace std;  
8 |  
9 | struct ListNode {  
10 |     int val;  
11 |     ListNode * next;  
12 |     ListNode(int _val):val(_val),next(nullptr){}  
13 | };  
14 |  
15 | ListNode* myReverseList(ListNode*head) {  
16 |  
17 |     if (head == nullptr || head->next == nullptr) return head;  
18 |     ListNode dumyhead(0);  
19 |     ListNode* pre = &dumyhead;  
20 |     pre->next = head;  
21 |     ListNode* cur = head->next;  
22 |     head->next = nullptr;  
23 |     ListNode* node = new ListNode(-1);  
24 |     while (cur != nullptr) {  
25 |         node = cur;  
26 |         cur = cur->next;  
27 |         node->next = pre->next;  
28 |         pre->next = node;  
29 |     }  
30 |  
31 |     return dumyhead.next;  
32 | }  
33 |  
34 | ListNode* myMerge(ListNode* p1, ListNode* p2) {  
35 |     if (p1 == nullptr) return p2;  
36 |     if (p2 == nullptr) return p1;  
37 |  
38 |     ListNode dumyhead(0);  
39 |     ListNode* pre = &dumyhead;  
40 |     while (p1 != nullptr && p2 != nullptr) {  
41 |         pre->next = p1;  
42 |         p1 = p1->next;  
43 |         pre = pre->next;  
44 |         pre->next = p2;  
45 |         p2 = p2->next;
```

```

46         pre = pre->next;
47     }
48     if (p1 != nullptr) pre->next = p1;
49     return dumyhead.next;
50 }
51
52
53 ListNode* myReverOrderList(ListNode *head) {
54     if (head == nullptr || head->next == nullptr) return head;
55     ListNode* slow = head, * fast = head->next;
56     while (fast != nullptr && fast->next != nullptr) {
57         slow = slow->next;
58         fast = fast->next->next;
59     }
60
61     ListNode* second = slow->next;
62     slow->next = nullptr;
63     second = myReverseList(second);
64     head = myMerge(head, second);
65     return head;
66 }
67
68 int main() {
69     ListNode* head = new ListNode(1);
70     ListNode* node1 = new ListNode(2);
71     ListNode* node2 = new ListNode(3);
72     ListNode* node3 = new ListNode(4);
73     ListNode* node4 = new ListNode(5);
74     ListNode* node5 = new ListNode(6);
75
76     head->next = node1;
77     node1->next = node2;
78     node2->next = node3;
79     node3->next = node4;
80     node4->next = node5;
81     node5->next = nullptr;
82
83
84     head = myReverOrderList(head);
85     while (head != nullptr) {
86         cout << head->val << endl;
87         head = head->next;
88     }
89
90     return 0;
91 }
```

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 $O(\text{nodes})$ ， 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 |  
5 | 示例 2:  
6 | 输入: 2->1->3->5->6->4->7->NULL  
7 | 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 |     }  
6 |     ListNode* first=head;//奇链表头结点  
7 |     ListNode* second=head->next;//偶链表头结点  
8 |     ListNode* cur=second;//保存偶链表头结点  
9 |     while(second != nullptr && second->next != nullptr)  
10 |    {  
11 |        first->next=second->next;  
12 |        second->next=first->next->next;  
13 |        first=first->next;  
14 |        second=second->next;  
15 |    }  
16 |    first->next=cur;  
17 |    return head;  
18 |}  
19 |}
```

第二种解法

```
1 |  
2 | ListNode* oddEvenList(ListNode* head)  
3 |{  
4 |     if(head == NULL)  
5 |     {  
6 |         return head;  
7 |     }  
8 |  
9 |     ListNode* p = head;  
10 |     ListNode* q = head->next;  
11 |     ListNode* evenhead = q;  
12 |  
13 |     while(q != NULL && q->next != NULL)  
14 |    {  
15 |        p->next = p->next->next;  
16 |        p = p->next;  
17 |        q->next = q->next->next;  
18 |        q = q->next;  
19 |    }
```

```
20  
21     p->next = evenhead;  
22  
23     return head;  
24 }
```

双非学历、字节全栈、互联网一线卑微打工仔，欢迎关注个人公众号。
关注那些曾经像我一样的小白新手程序员们，我踩过的坑不希望你再踩，我走过的路希望你能照着走下来。



3.3、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，而独自占有栈

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。

而异常是由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 *arg);

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。

2. 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：

```

1  typedef struct{
2      int etachstate;      // 线程分离的状态
3      int schedpolicy;    // 线程调度策略
4      struct sched_param schedparam; // 线程的调度参数
5      int inheritsched;   // 线程的继承性
6      int scope;         // 线程的作用域
7      // 以下为线程栈的设置
8      size_t guardsize;   // 线程栈末尾警戒缓冲大小
9      int stackaddr_set; // 线程的栈设置
10     void * stackaddr; // 线程栈的位置
11     size_t stacksize;  // 线程栈大小
12 }pthread_attr_t;
13

```

- 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。

多进程

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`$?`，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。

2. 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 ~ 0xC0000000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, 进入内核模式, 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-served (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。

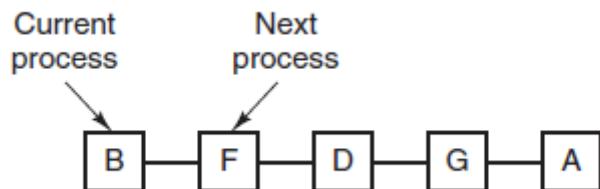
4、时间片轮转

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。

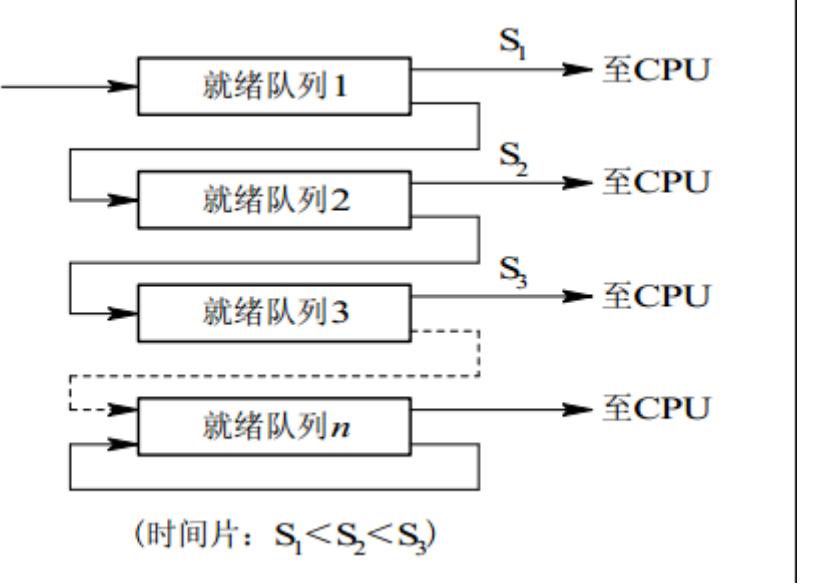
6、多级反馈队列

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，则必须按照一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，一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。访问一次快表耗时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？

$$(1+100) * 0.9 + (1+100+100) * 0.1 = 111 \text{ us}$$

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 $(1+100) * 0.9 + (100+100) * 0.1 = 110.9 \text{ us}$

若未采用快表机制，则访问一个逻辑地址需要 $100+100 = 200\text{us}$

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？

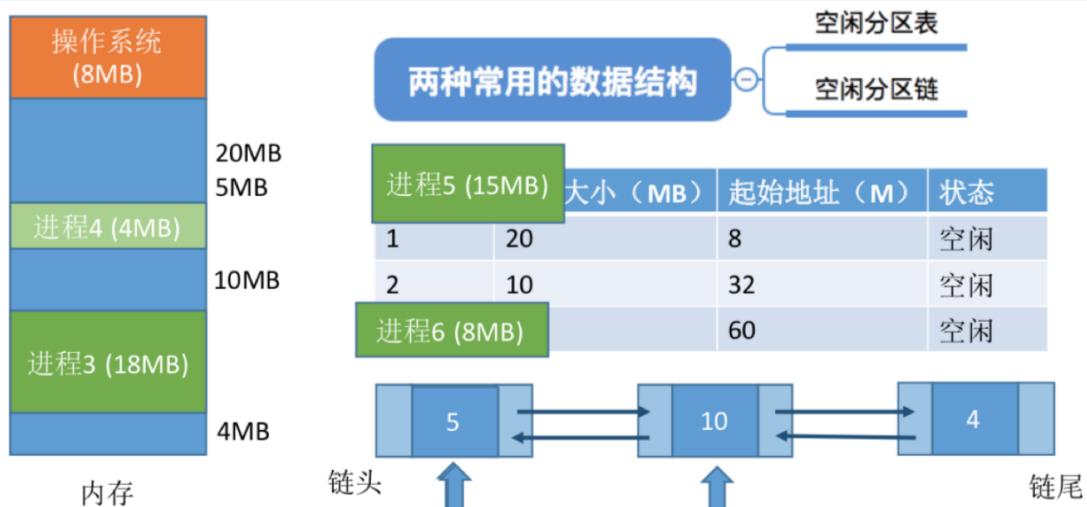
交换技术主要是在不同进程（或作业）之间进行，而覆盖则用于同一程序或进程中。

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，找到第一个能满足大小的空间分区。

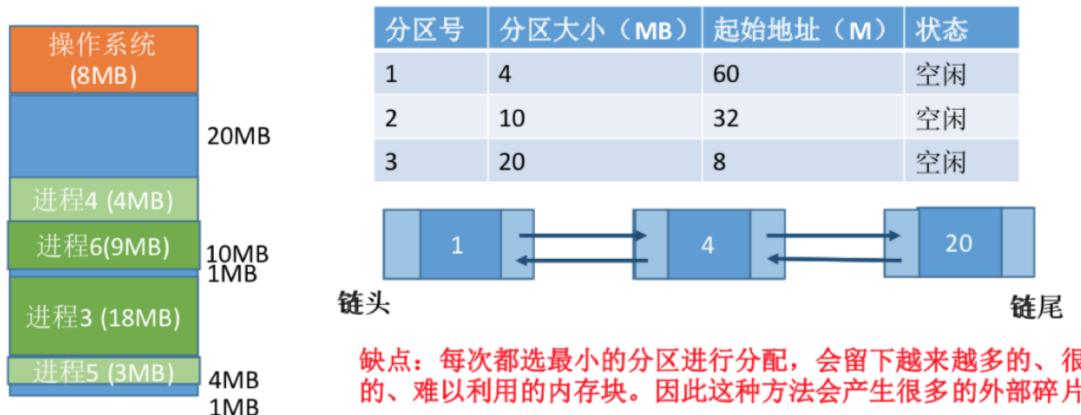
如何实现：空闲分区以地址递增的次序排列。每次分配内存时顺序查找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。



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间，即，优先使用更小的空间区。

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。

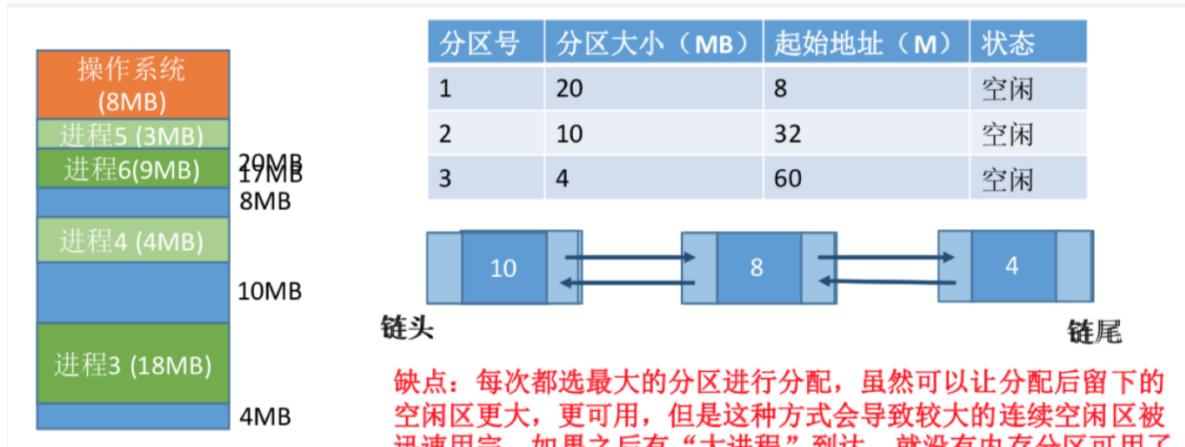


3. 最坏适应算法

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。

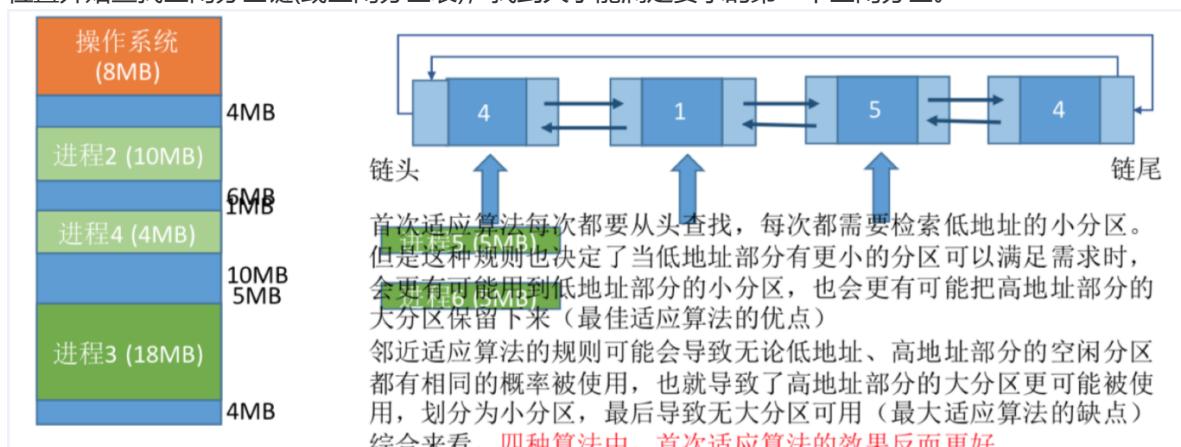
如何实现:空闲分区按容量递减次序链接。每次分配内存时顺序查找空闲分区链(或空闲分区表),找到大小能满足要求的第一个空闲分区。



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。

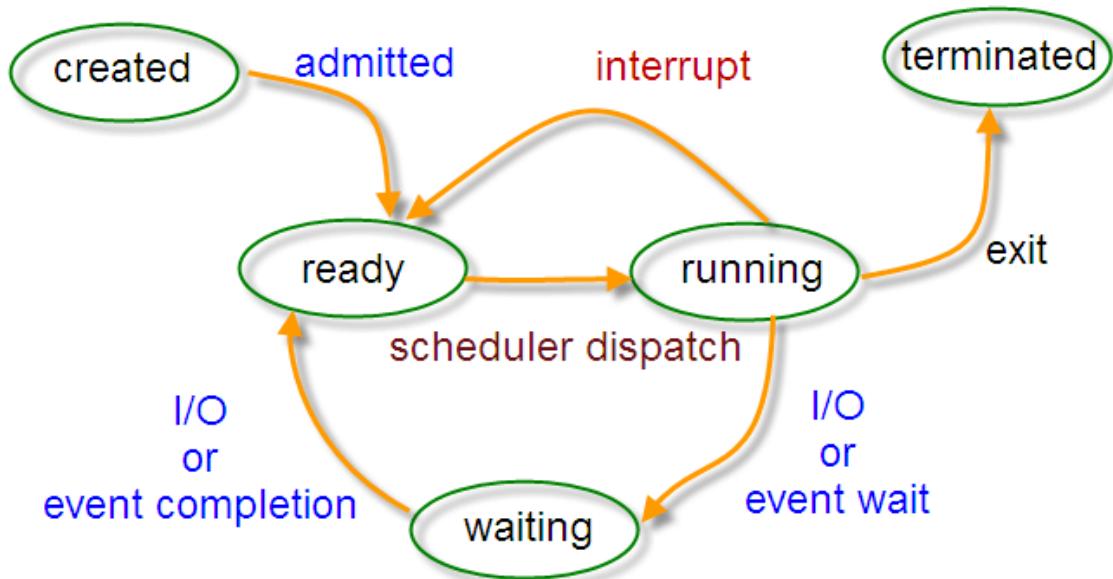
主要有两种虚拟技术：时（时间）分复用技术和空（空间）分复用技术。

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，每次只执行一小段时间片并快速切换。

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？

Process State



- 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：

(1) 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时能够显示行号。

(2) 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。

(3) 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——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)。

(4) 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/93/675fd4af3ab34b2db0ae650855aa52d5>

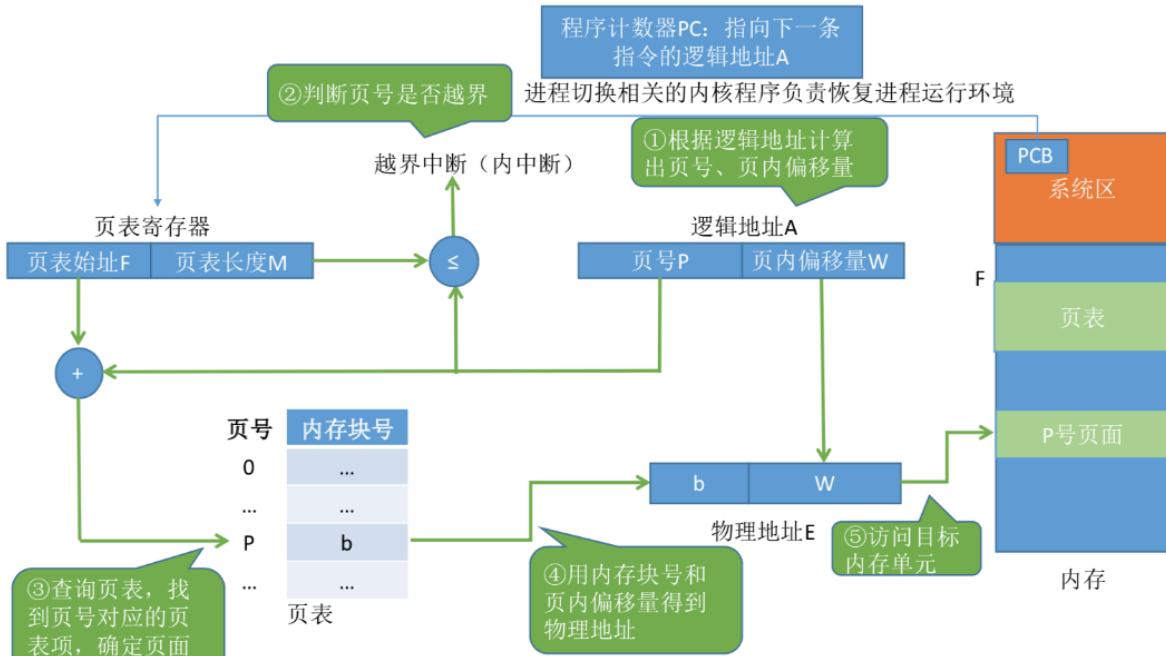
15、通过例子讲解逻辑地址转换为物理地址的基本过程

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：



- ①计算页号 P 和页内偏移量 W （如果用十进制数手算，则 $P=A/L$, $W=A \% L$; 但是在计算机实际运行时，逻辑地址结构是固定不变的，因此计算机硬件可以更直接地得到二进制表示的页号和页内偏移量）
 手动验证：假设页面大小 $L=1KB$, 最终要访问的内存块号 $b=2$, 页内偏移量 $W=1023$ 。
 ①尝试用 $E=b * L + W$ 计算目标物理地址。
 ②尝试把内存块号、页内偏移量用二进制表示，并把它们拼接起来得到物理地址。
 对比①②的结果是否一致
- ②比较页号 P 和页表长度 M , 若 $P \geq M$, 则产生越界中断（内中断）。
 ③页表中页号 P 对应的页表项地址 = 页表起始地址 + 页号 P * 页表项长度。即为内存块号。（注意区分页表项长度、页表长度。页表中总共有几个页表项，即总共有几个页；**页表项长度**指的是每个页表项的字节数；**页面大小**指的是一个页面占多大的存储空间）
 ④计算 $E = b * L + W$, 用得到的物理地址 E 去访存。（如果内存块号、页面偏移量是用二进制表示的，那么把二者拼接起来就是最终的物理地址了）

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 $2^{10}B = 1KB$ ），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
 $P = A / L = 2500 / 1024 = 2$; 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 $E = b * L + W = 8 * 1024 + 452 = 8644$

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，页式管理中地址是唯一的。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？

1. 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

```

2. 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。

3. 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() {
10     down(&mutex);
11     // 临界区
12     up(&mutex);
13 }
14 }
```

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。

```
1  #define N 100
2  typedef int semaphore;
3  semaphore mutex = 1;
4  semaphore empty = N;
5  semaphore full = 0;
```

```

7 void producer() {
8     while(TRUE) {
9         int item = produce_item();
10        down(&empty);
11        down(&mutex);
12        insert_item(item);
13        up(&mutex);
14        up(&full);
15    }
16 }
17
18 void consumer() {
19     while(TRUE) {
20         down(&full);
21         down(&mutex);
22         int item = remove_item();
23         consume_item(item);
24         up(&mutex);
25         up(&empty);
26     }
27 }
28

```

4. 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
5     procedure insert();
6     begin
7         // ...
8     end;
9
10    procedure remove();
11    begin
12        // ...
13    end;
14 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。

使用管程实现生产者-消费者问题

```

1 // 管程
2 monitor ProducerConsumer

```

```

3     condition full, empty;
4     integer count := 0;
5     condition c;
6
7     procedure insert(item: integer);
8     begin
9         if count = N then wait(full);
10        insert_item(item);
11        count := count + 1;
12        if count = 1 then signal(empty);
13    end;
14
15    function remove: integer;
16    begin
17        if count = 0 then wait(empty);
18        remove = remove_item;
19        count := count - 1;
20        if count = N - 1 then signal(full);
21    end;
22 end monitor;
23
24 // 生产者客户端
25 procedure producer
26 begin
27     while true do
28     begin
29         item = produce_item;
30         ProducerConsumer.insert(item);
31     end
32 end;
33
34 // 消费者客户端
35 procedure consumer
36 begin
37     while true do
38     begin
39         item = ProducerConsumer.remove;
40         consume_item(item);
41     end
42 end;

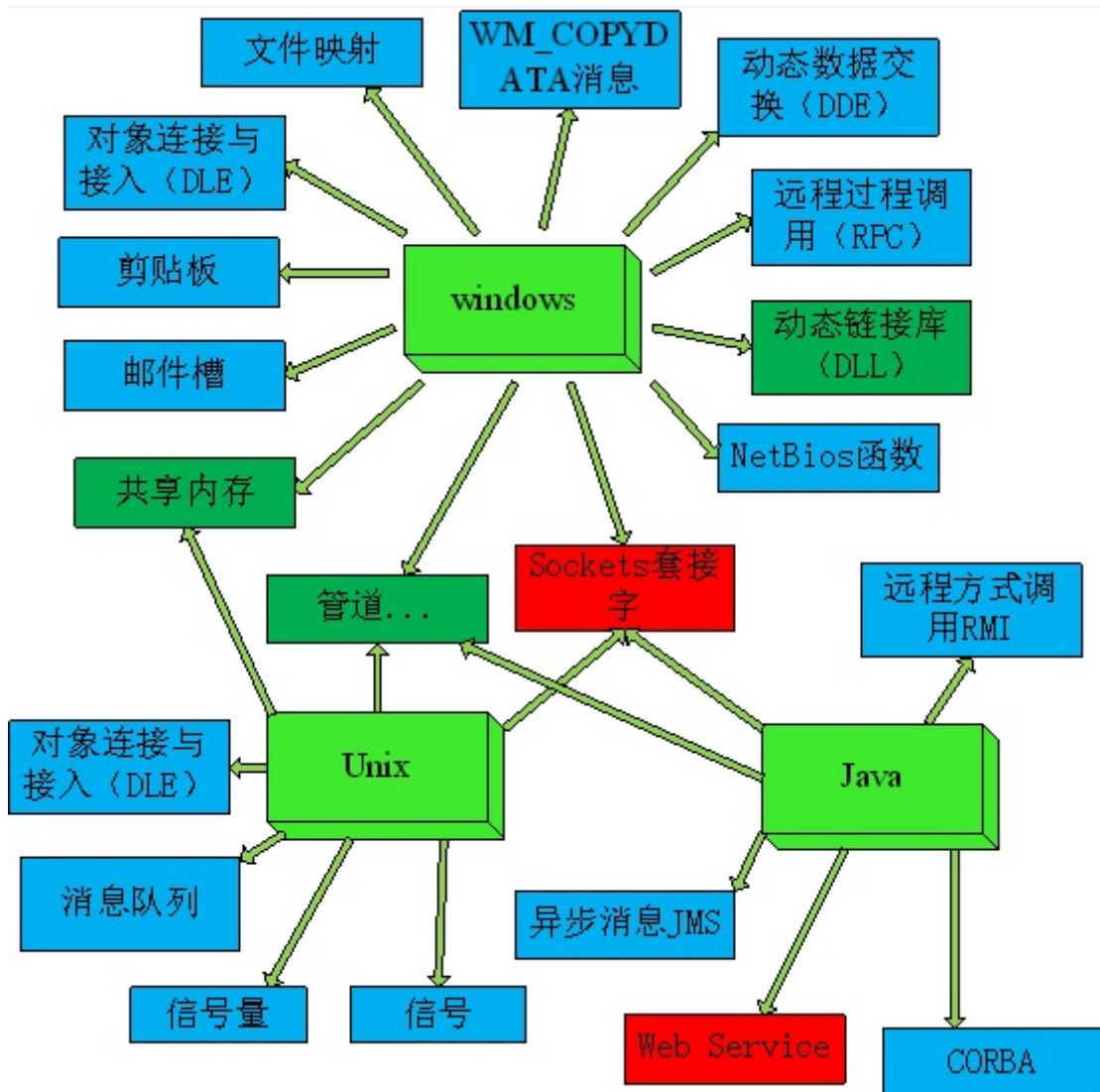
```

17、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

18、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。有足够的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；

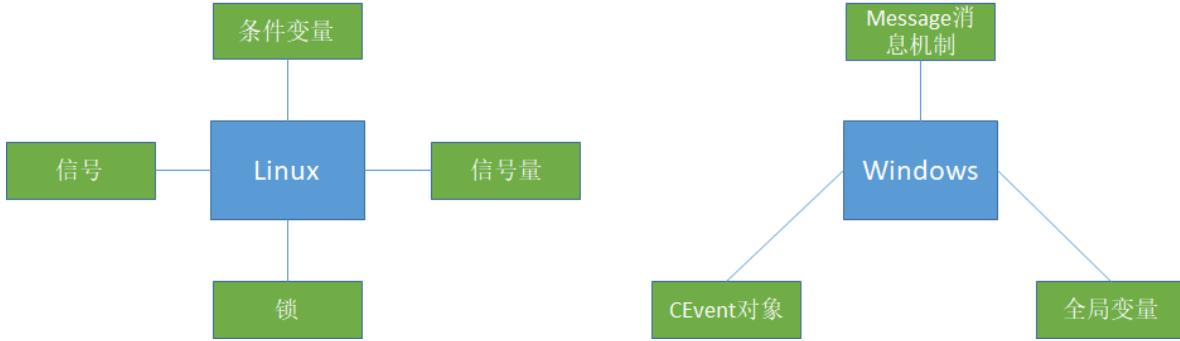
信号量(semaphore): 信号量主要作为进程间以及同进程不同线程之间的同步手段；

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



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，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。

19、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，失败返回-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时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。

20、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。

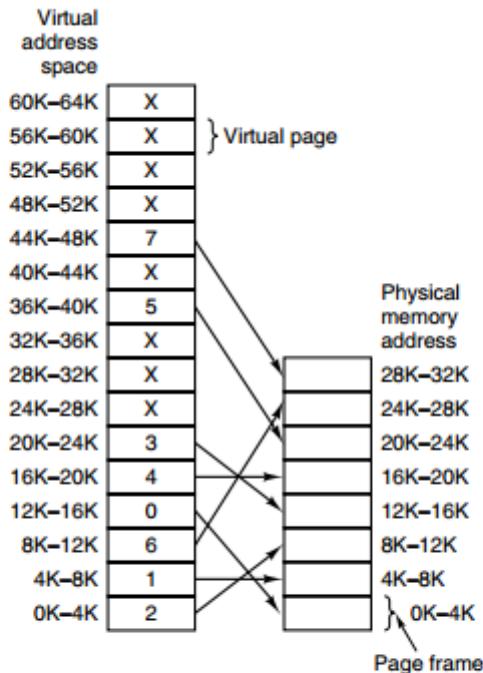


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K-8K really means 4096-8191 and 8K to 12K means 8192-12287.

21、说一下你理解中的内存？他有什么作用呢？

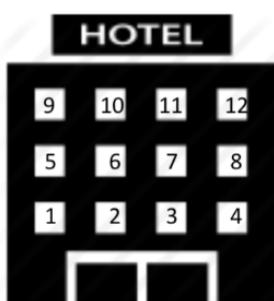
什么是内存？有何作用？

内存是用于存放数据的硬件。程序执行前需要先放到内存中才能被CPU处理。



思考：在多道程序环境下，系统中会有多个程序并发执行，也就是说会有多个程序的数据需要同时放到内存中。那么，如何区分各个程序的数据是放在什么地方的呢？

方案：给内存的存储单元编地址



内存地址从0开始，每个地址对应一个存储单元

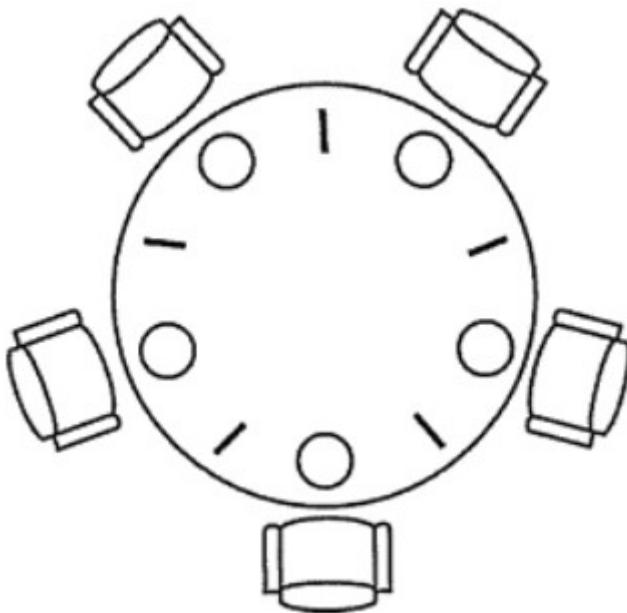
地址	内存
0	“小房间”
1	“小房间”
2
3	
4	
5	
6	

内存中也有一个一个的“小房间”，每个小房间就是一个“存储单元”

如果计算机“按字节编址”，则每个存储单元大小为1字节，即1B，即8个二进制位

如果字长为16位的计算机“按字编址”，则每个存储单元大小为1个字；每个字的大小为16个二进制位

22、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
3 void philosopher(int i) {
4     while(TRUE) {
5         think();
6         take(i);           // 拿起左边的筷子
7         take((i+1)%N); // 拿起右边的筷子
8         eat();
9         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。

```
1 #define N 5
2 #define LEFT (i + N - 1) % N // 左邻居
3 #define RIGHT (i + 1) % N    // 右邻居
4 #define THINKING 0
5 #define HUNGRY    1
6 #define EATING    2
7 typedef int semaphore;
8 int state[N];           // 跟踪每个哲学家的状态
9 semaphore mutex = 1;    // 临界区的互斥，临界区是 state 数组，对其修改需要互斥
10 semaphore s[N];        // 每个哲学家一个信号量
11
12 void philosopher(int i) {
13     while(TRUE) {
14         think(i);
15         take_two(i);
```

```

16     eat(i);
17     put_two(i);
18 }
19 }
20
21 void take_two(int i) {
22     down(&mutex);
23     state[i] = HUNGRY;
24     check(i);
25     up(&mutex);
26     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
27 }
28
29 void put_two(i) {
30     down(&mutex);
31     state[i] = THINKING;
32     check(LEFT); // 尝试通知左右邻居，自己吃完了，你们可以开始吃了
33     check(RIGHT);
34     up(&mutex);
35 }
36
37 void eat(int i) {
38     down(&mutex);
39     state[i] = EATING;
40     up(&mutex);
41 }
42
43 // 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
44 void check(i) {
45     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT]
46     != EATING) {
47         state[i] = EATING;
48         up(&s[i]);
49     }
50 }
51

```

23、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
6 void reader() {
7     while(TRUE) {
8         down(&count_mutex);
9         count++;

```

```

10         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
11         up(&count_mutex);
12         read();
13         down(&count_mutex);
14         count--;
15         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
16         up(&count_mutex);
17     }
18 }
19
20 void writer() {
21     while(TRUE) {
22         down(&data_mutex);
23         write();
24         up(&data_mutex);
25     }
26 }
27
28

```

24、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/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上进行忙等待并不断的进行锁请求，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。

25、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。

26、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。

27、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。

28、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。

29、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。

30、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

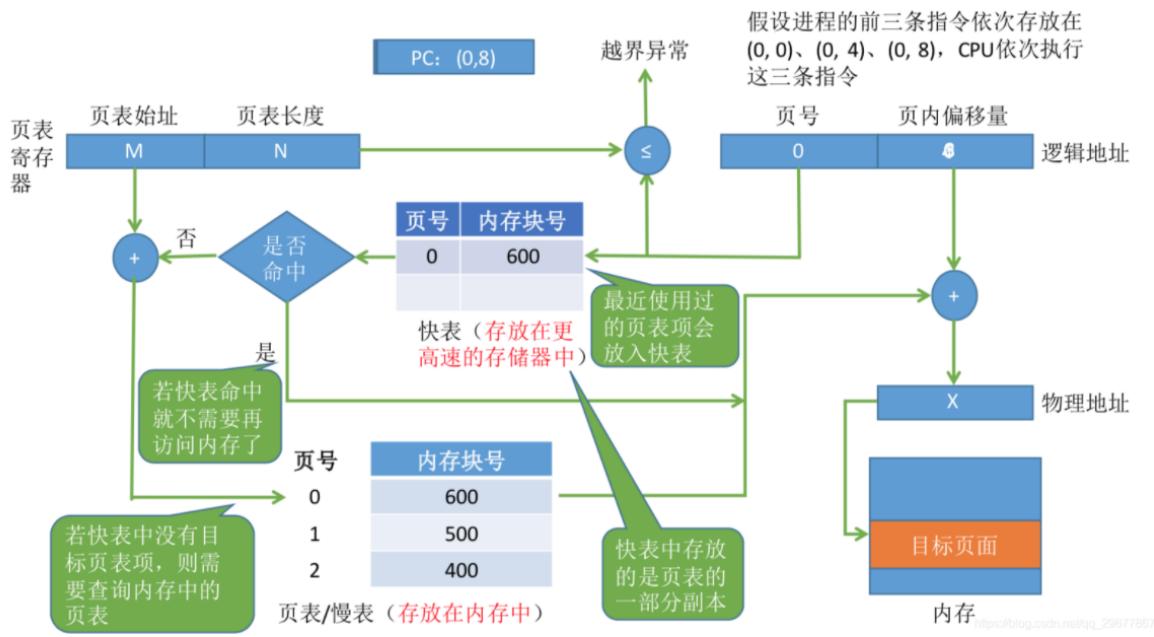
《linux终端关闭时为什么会导致在其上启动的进程退出？》：<https://blog.csdn.net/QFire/article/details/80112701>

31、如何让进程后台运行

- (1) 命令后面加上&即可，实际上，这样是将命令放入到一个作业队列中了
- (2) ctrl + z 挂起进程，使用jobs查看序号，在使用bg %序号后台运行进程
- (3) nohup + &，将标准输出和标准错误缺省会被重定向到 nohup.out 文件中，忽略所有挂断(SIGHUP) 信号
- (4) 运行指令前面 + setsid，使其父进程编程init进程，不受HUP信号的影响
- (5) 将命令+ &放在()括号中，也可以是进程不受HUP信号的影响

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

35、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。
- (7) 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）。

36、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/details/8128889

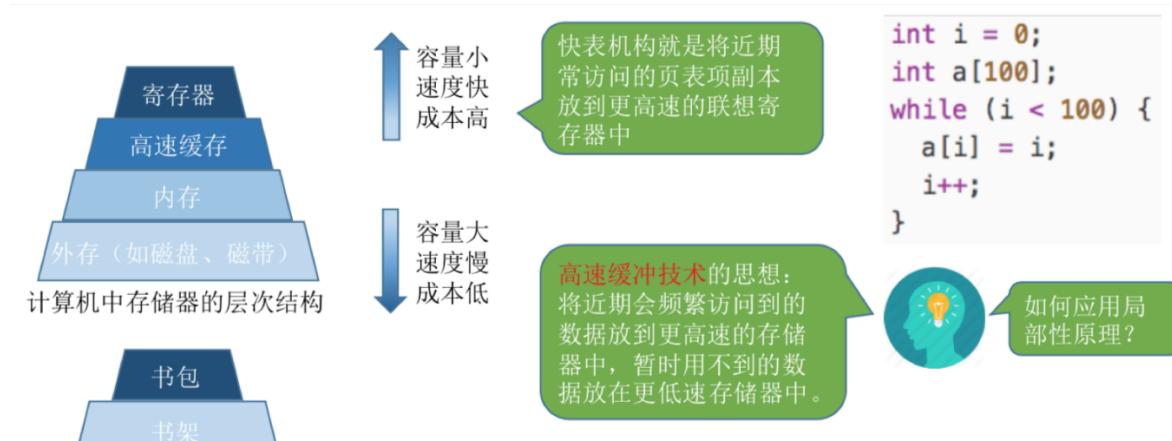
《01_fork()的使用》：<https://blog.csdn.net/WUZHU2017/article/details/81636851>

37、局部性原理你知道吗？主要有哪两大局部性原理？各自是什么？

主要分为**时间局部性**和**空间局部性**。

时间局部性:如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行;如果某个数据被访问过，不久之后该数据很可能再次被访问。(因为程序中存在大量的循环)

空间局部性:一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能被访问。(因为很多数据在内存中都是连续存放的，并且程序的指令也是顺序地在内存中存放的)



38、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

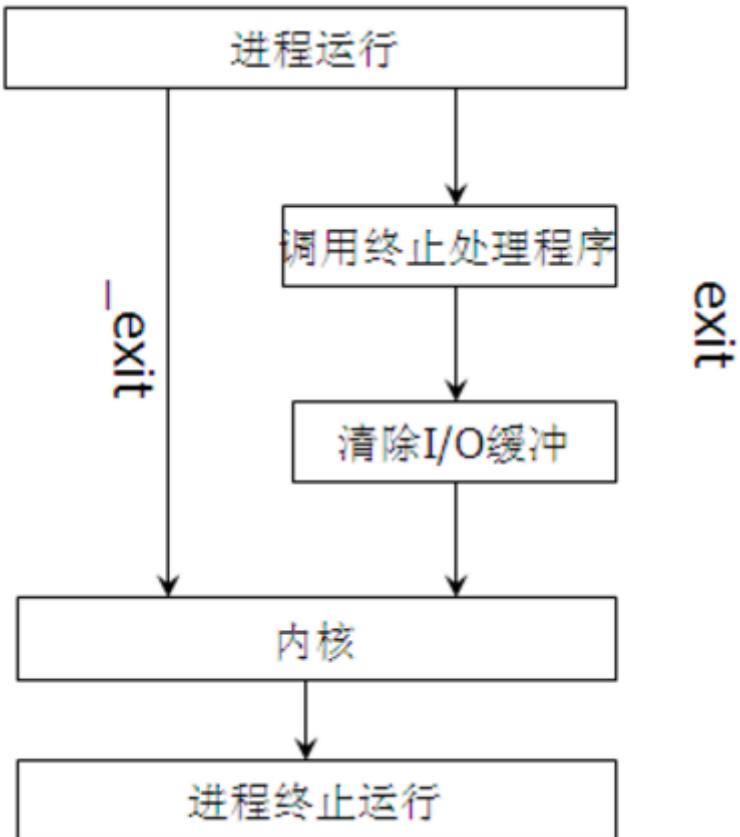
会话

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。

39、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

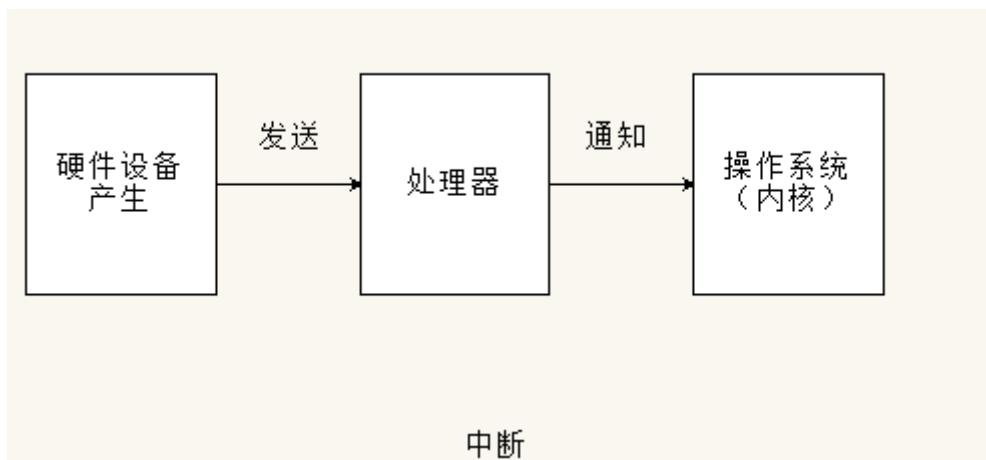


《学习笔记]进程终止的5种方式》：<https://www.cnblogs.com/shichuan/p/4432503.html>

40、Linux中异常和中断的区别

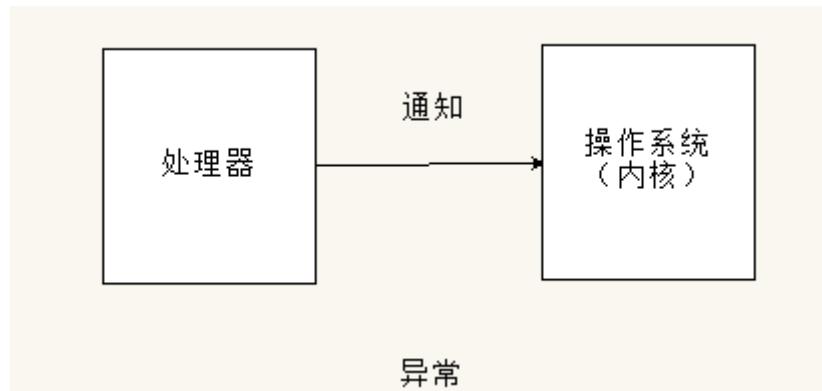
中断

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

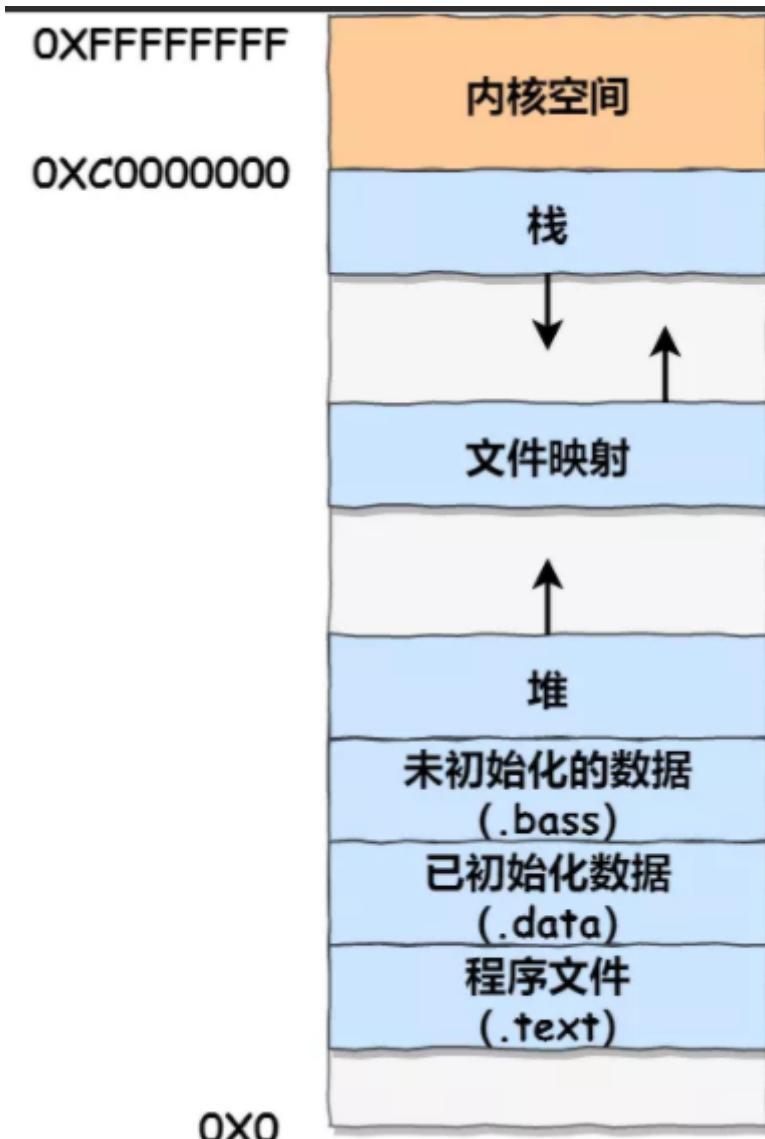
- 最后都是由CPU发送给内核，由内核去处理
- 处理程序的流程设计上是相似的

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/10284741>

41、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；

42、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。

43、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>

44、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。

45、常见的几种磁盘调度算法

读写一个磁盘块的时间的影响因素有：

- 旋转时间（主轴转动盘面，使得磁头移动到适当的扇区上）
- 寻道时间（制动手臂移动，使得磁头移动到适当的磁道上）
- 实际的数据传输时间

其中，寻道时间最长，因此磁盘调度的主要目标是使磁盘的平均寻道时间最短。

1. 先来先服务

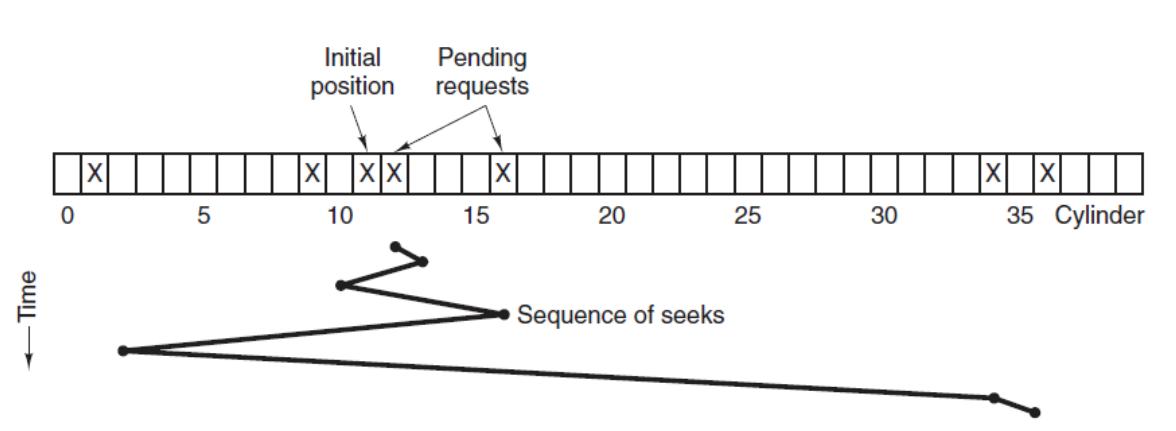
按照磁盘请求的顺序进行调度。

优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。

2. 最短寻道时间优先

优先调度与当前磁头所在磁道距离最近的磁道。

虽然平均寻道时间比较低，但是不够公平。如果新到达的磁道请求总是比一个在等待的磁道请求近，那么在等待的磁道请求会一直等待下去，也就是出现饥饿现象。具体来说，两端的磁道请求更容易出现饥饿现象。

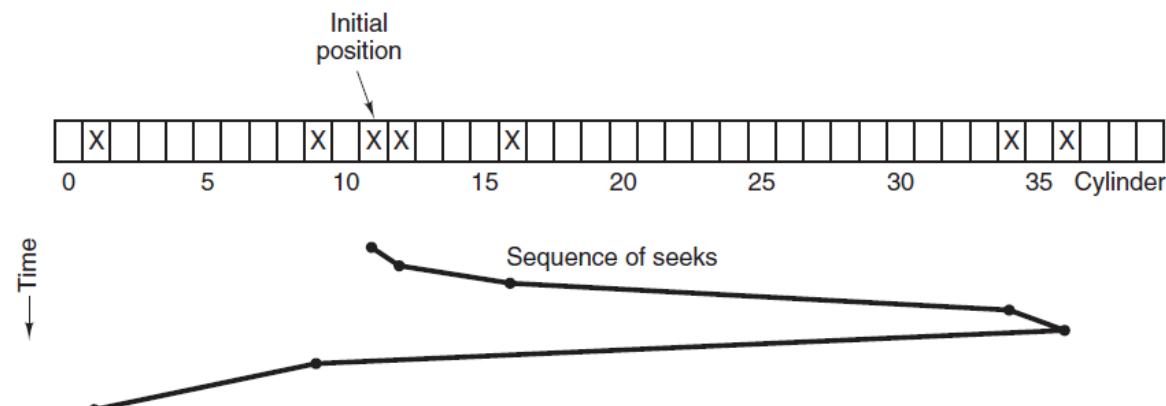


3. 电梯扫描算法

电梯总是保持一个方向运行，直到该方向没有请求为止，然后改变运行方向。

电梯算法（扫描算法）和电梯的运行过程类似，总是按一个方向来进行磁盘调度，直到该方向上没有未完成的磁盘请求，然后改变方向。

因为考虑了移动方向，因此所有的磁盘请求都会被满足，解决了 SSTF 的饥饿问题。



46、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>

47、抖动你知道是什么吗？它也叫颠簸现象

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数(分配给进程的物理块不够)

为进程分配的物理块太少，会使进程发生抖动现象。为进程分配的物理块太多，又会降低系统整体的并发度，降低某些资源的利用率

为了研究为应该为每个进程分配多少个物理块，Denning 提出了进程工作集”的概念

48、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。

49、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。

50、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。

(2) 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。

(3) 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。

(4) 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）。

(5) 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/2010/12/03/1895045.html>

应该在外存(磁盘)的什么位置保存被换出的进程？

51、内存交换中，被换出的进程保存在哪里？

保存在磁盘中，也就是外存中。具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式；对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式（学过文件管理章节后即可理解）。总之，对换区的I/O速度比文件区的更快。

52、在发生内存交换时，有些进程是被优先考虑的？你可以说一说吗？

可优先换出阻塞进程；可换出优先级低的进程；为了防止优先级低的进程在被调入内存后很快又被换出，有的系统还会考虑进程在内存的驻留时间...

(注意：PCB 会常驻内存，不会被换出外存)

53、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。

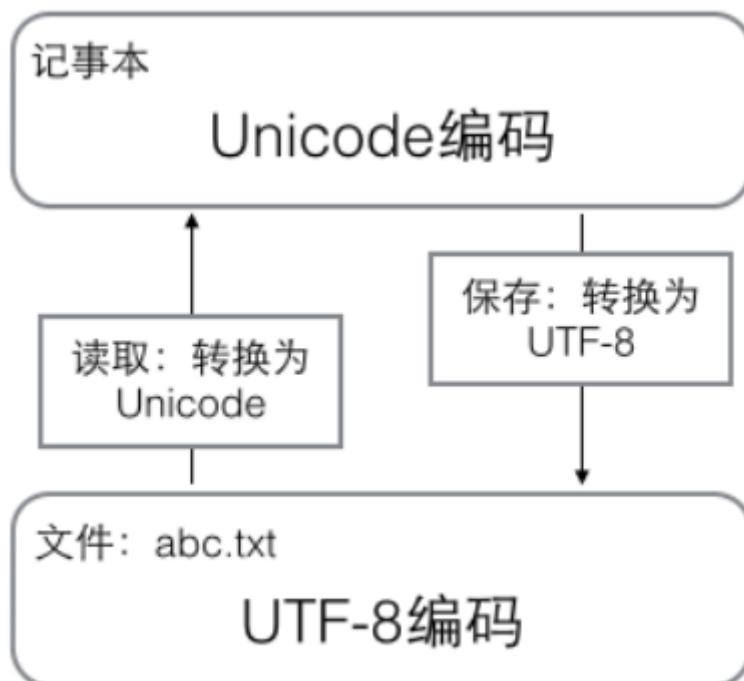
UTF-8

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：

服务器

Unicode编码

输出UTF-8网页

浏览器

《字符编码中ASCII、Unicode和UTF-8的区别》：<https://www.cnblogs.com/moumoon/p/1098234.html>

54、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。

	CPU1	CPU2
1	$i=1$	$i=1$
2	$i+1$	$i+1$
3	$i=2$	$i=2$
4		
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。

(2) 使用缓存锁定保证原子性

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。

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 4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>

55、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因素是转移时间，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紧张时开始启动，而系统负载降低就暂停。
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。

56、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。

57、可能是最全的页面置换算法总结了

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。

例：假设系统为某进程分配了三个内存块，并考虑到有以下页面号引用串（会依次访问这些页面）：

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2	2		2		2							7		
内存块2		0	0	0		0	4			0		0						0		
内存块3			1	1		3		3		3		1					1			
是否缺页	v	v	v	v		v	v		v		v		v		v		v		v	

选择从 0, 1, 7 中淘汰一页。按最佳置换的规则，往后寻找，最后一个出现的页号就是要淘汰的页面

整个过程缺页中断发生了9次，页面置换发生了6次。

注意：缺页时未必发生页面置换。若还有可用的空闲内存块，就不用进行页面置换。

$$\text{缺页率} = 9/20 = 45\%$$

https://ilog.cdn.netlog_29677867

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。

例：假设系统为某进程分配了三个内存块，并考虑到有以下页面号引用串：

3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4

访问页面	3	2	1	0	3	2	4	3	2	1	0	4
内存块1	3	3	3	0	0	0	4			4	4	
内存块2		2	2	2	3	3	3			1	1	
内存块3			1	1	1	2	2			2	0	
是否缺页	v	v	v	v	v	v	v	v		v	v	



分配三个内存块时，缺页次数：9次

例：假设系统为某进程分配了四个内存块，并考虑到有以下页面号引用串：

3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4

访问页面	3	2	1	0	3	2	4	3	2	1	0	4
内存块1	3	3	3	3		4	4	4	4	0	0	
内存块2		2	2	2		2	3	3	3	3	4	
内存块3			1	1		1	1	2	2	2	2	
内存块4				0		0	0	0	1	1	1	
是否缺页	v	v	v	v		v	v	v	v	v	v	

分配四个内存块时，缺页次数：10次
分配三个内存块时，缺页次数：9次

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，如果对一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。

页号	内存块号	状态位	访问字段	修改位	外存地址	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										
访问页面	1	8	1	7	8	2	7	2	1	8	3	8	2	1	3	1
内存块1	1	1	1	1							1					1
内存块2		8		8	8						8					7
内存块3				7		7					3					3
内存块4					2		2				2					2
缺页否	✓	✓	✓	✓	✓						✓					✓

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。

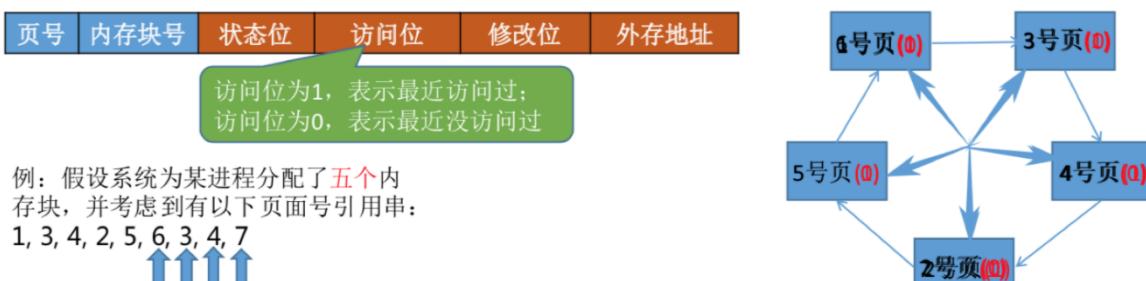
4. 时钟置换算法(CLOCK)

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师尝试了很多算法，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算法选择一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一个循环队列

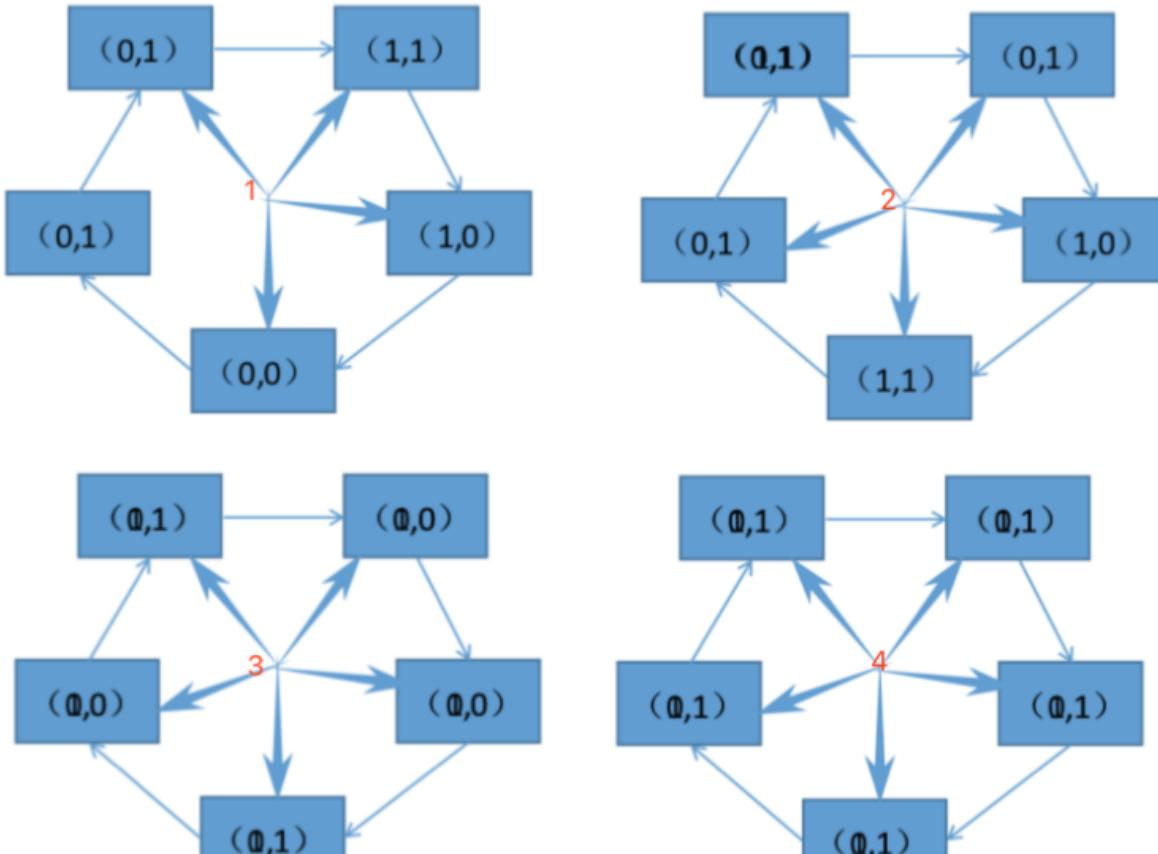
第一轮：从当前位置开始扫描到第一个(A = 0, M = 0)的帧用于替换。表示该页面最近既未被访问，又未被修改，是最佳淘汰页

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置换算法选择一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。若第一轮没选中, 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(0,1), 并将扫描过的页面访问位都置为0 第三轮:淘汰(0, 0) 第四轮:淘汰(0, 1)	算法开销较小, 性能也不错

58、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。

59、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。

```
1 #include <iostream>
2 #include <vector>
3 #include <list>
4 #include <thread>
5 #include <mutex> //引入互斥量头文件
6 using namespace std;
7
8 class A {
9 public:
10     //插入消息，模拟消息不断产生
11     void insertMsg() {
12         for (int i = 0; i < 100; i++) {
13             cout << "插入一条消息:" << i << endl;
14             my_mutex1.lock(); //语句1
15             my_mutex2.lock(); //语句2
16             Msg.push_back(i);
17             my_mutex2.unlock();
18             my_mutex1.unlock();
19         }
20     }
21     //读取消息
22     void readMsg() {
23         int MsgCom;
24         for (int i = 0; i < 100; i++) {
25             MsgCom = MsgLULProc(i);
26             if (MsgLULProc(MsgCom)) {
27                 //读出消息了
28                 cout << "消息已读出" << MsgCom << endl;
29             }
30             else {
31                 //消息暂时为空
32                 cout << "消息为空" << endl;
33             }
34         }
35     }
36     //加解锁代码
37     bool MsgLULProc(int &command) {
38         int curMsg;
39         my_mutex2.lock(); //语句3
40         my_mutex1.lock(); //语句4
41         if (!Msg.empty()) {
42             //读取消息，读完删除
43             command = Msg.front();
44             Msg.pop_front();
45         }
46     }
47 }
```

```

46         my_mutex1.unlock();
47         my_mutex2.unlock();
48         return true;
49     }
50     my_mutex1.unlock();
51     my_mutex2.unlock();
52     return false;
53 }
54 private:
55     std::list<int> Msg; //消息变量
56     std::mutex my_mutex1; //互斥量对象1
57     std::mutex my_mutex2; //互斥量对象2
58 };
59
60 int main() {
61     A a;
62     //创建一个插入消息线程
63     std::thread insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象
64     //创建一个读取消息线程
65     std::thread readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象
66     insertTd.join();
67     readTd.join();
68     return 0;
69 }
70
71

```

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：

- 鸱鸟策略
- 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。

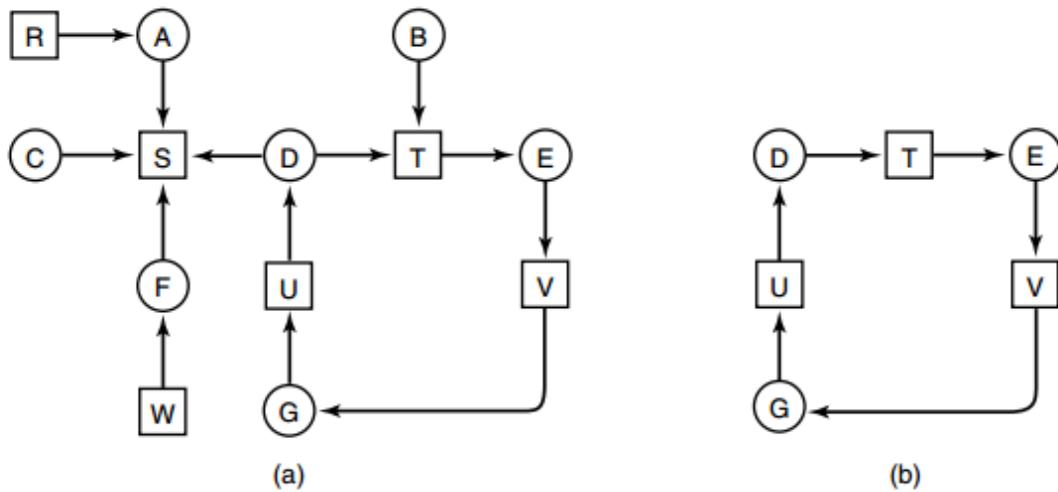
当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。

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

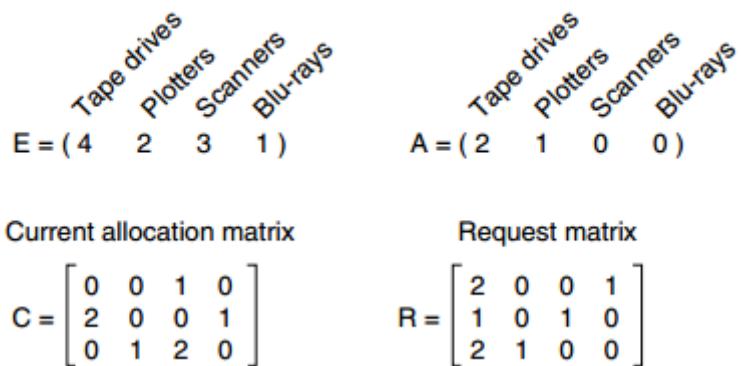


上图为资源分配图，其中方框表示资源，圆圈表示进程。资源指向进程表示该资源已经分配给该进程，进程指向资源表示进程请求获取该资源。

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 P_1 和 P_2 所请求的资源都得不到满足，只有进程 P_3 可以，让 P_3 执行，之后释放 P_3 拥有的资源，此时 $A = (2 \ 2 \ 2 \ 0)$ 。 P_2 可以执行，执行后释放 P_2 拥有的资源， $A = (4 \ 2 \ 2 \ 1)$ 。 P_1 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 P_i ，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。

2. 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。

3. 破坏不剥夺条件

允许抢占资源

4. 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。

1. 安全状态

	Has	Max		Has	Max		Has	Max		Has	Max		Has	Max	
A	3	9	(a)	A	3	9	(b)	A	3	9	(c)	A	3	9	(d)
B	2	4		B	4	4		B	0	-		B	0	-	
C	2	7		C	2	7		C	2	7		C	7	7	
	Free: 3			Free: 1			Free: 5			Free: 0			Free: 7		

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。

2. 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。

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

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。

3. 多个资源的银行家算法

	Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1	1
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Resources assigned

	Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Resources still assigned

$$\begin{aligned} E &= (6342) \\ P &= (5322) \\ A &= (1020) \end{aligned}$$

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。

60、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

61、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

62、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。

双非学历、字节全栈、互联网一线卑微打工仔，欢迎关注个人公众号。
关注那些曾经像我一样的小白新手程序员们，我踩过的坑不希望你再踩，我走过的路希望你能照着走下来。



3.4、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。

3、你知道DNS是什么？

官方解释：DNS (Domain Name System, 域名系统)，因特网上作为**域名和IP地址相互映射的一个分布式数据库**，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。

通过主机名，最终得到该主机名对应的IP地址的过程叫做**域名解析**（或**主机名解析**）。

通俗的讲，我们更习惯于记住一个网站的名字，比如www.baidu.com,而不是记住它的ip地址，比如：167.23.10.2。

4、DNS的工作原理？

将主机域名转换为ip地址，属于应用层协议，使用UDP传输。（DNS应用层协议，以前有个考官问过）



过程：

总结：浏览器缓存，系统缓存，路由器缓存，IPS服务器缓存，根域名服务器缓存，顶级域名服务器缓存，主域名服务器缓存。

一、主机向本地域名服务器的查询一般都是采用递归查询。

二、本地域名服务器向根域名服务器的查询的迭代查询。

1)当用户输入域名时，浏览器先检查自己的缓存中是否这个域名映射的ip地址，有解析结束。

- 2) 若没命中，则检查操作系统缓存（如Windows的hosts）中有没有解析过的结果，有解析结束。
- 3) 若无命中，则请求本地域名服务器解析（LDNS）。
- 4) 若LDNS没有命中就直接跳到根域名服务器请求解析。根域名服务器返回给LDNS一个主域名服务器地址。
- 5) 此时LDNS再发送请求给上一步返回的gTLD（通用顶级域），接受请求的gTLD查找并返回这个域名对应的Name Server的地址
- 6) Name Server根据映射关系表找到目标ip，返回给LDNS
- 7) LDNS缓存这个域名和对应的ip，把解析的结果返回给用户，用户根据TTL值缓存到本地系统缓存中，域名解析过程至此结束

5、为什么域名解析用UDP协议？

因为UDP快啊！ UDP的DNS协议只要一个请求、一个应答就好了。

而使用基于TCP的DNS协议要三次握手、发送数据以及应答、四次挥手，但是UDP协议传输内容不能超过512字节。

不过客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。

6、为什么区域传送用TCP协议？

因为TCP协议可靠性好啊！

你要从主DNS上复制内容啊，你用不可靠的UDP？因为TCP协议传输的内容大啊，你用最大只能传512字节的UDP协议？万一同步的数据大于512字节，你怎么办？所以用TCP协议比较好！

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。（ $MSS = TCP\text{报文段长度} - TCP\text{首部长度}$ ）
- 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。

9、为什么服务器会缓存这一项功能?如何实现的?

原因

- 缓解服务器压力；
- 降低客户端获取资源的延迟：缓存通常位于内存中，读取缓存的速度更快。并且缓存服务器在地理位置上也有可能比源服务器来得近，例如浏览器缓存。

实现方法

- 让代理服务器进行缓存；
- 让客户端浏览器进行缓存。

10、HTTP请求方法你知道多少?

客户端发送的 **请求报文** 第一行是请求行，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 (request body)
3. 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。

4. 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)。
5. GET请求会被浏览器主动缓存，而POST不会，除非手动设置。
6. 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 `gethostbyname` 库函数（操作系统不同函数也不同）进行查询。

`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](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。

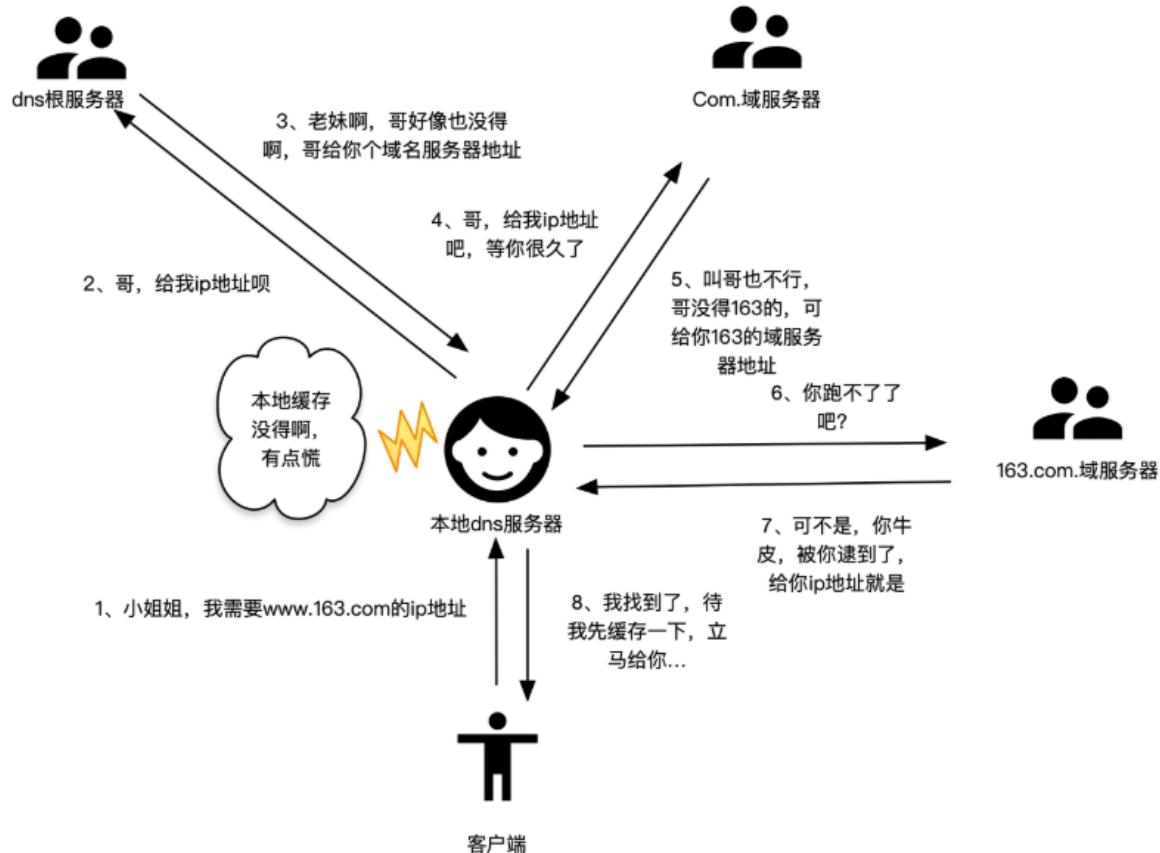
查询本地 DNS 服务器

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。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；
3. 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；
4. 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PreparedStatement是必须的，但是一般我们会从两个方面同时入手。

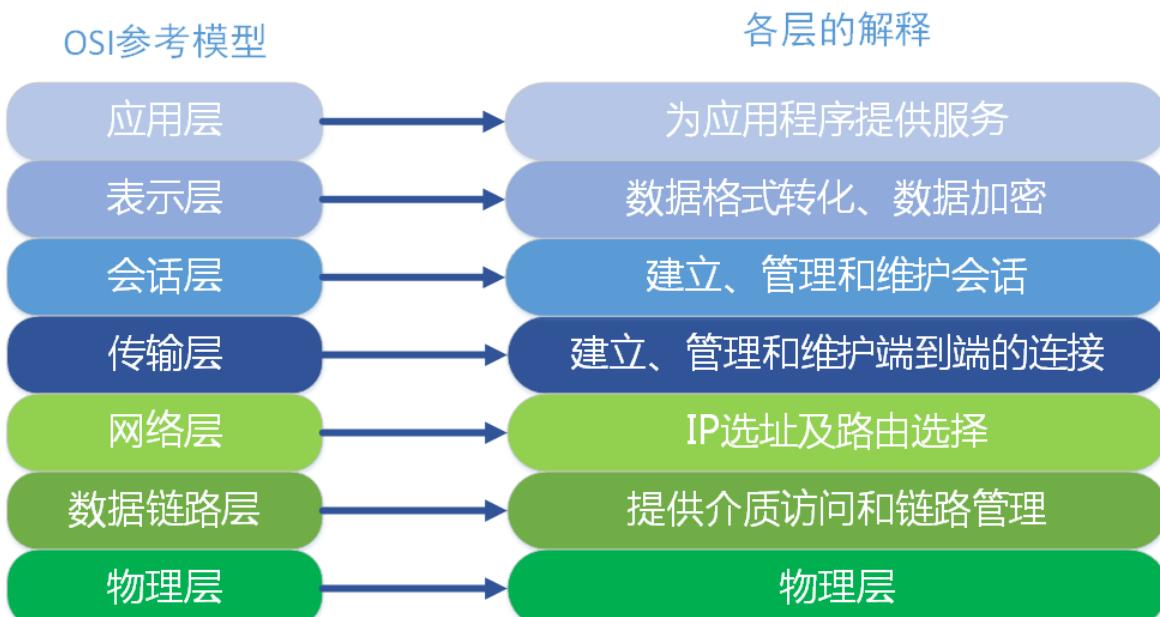
Web端

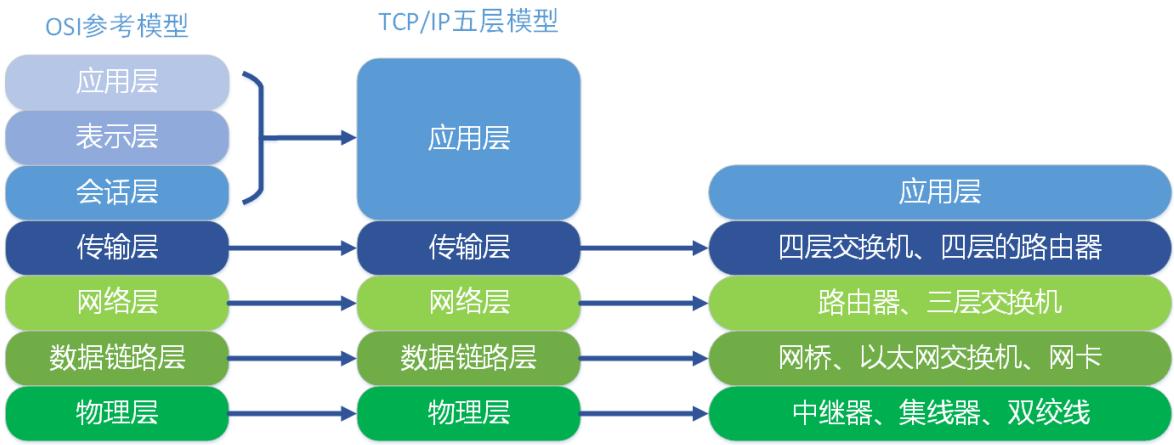
- 1) 有效性检验。
- 2) 限制字符串输入的长度。

服务端

- 1) 不用拼接SQL字符串。
- 2) 使用预编译的PreparedStatement。
- 3) 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)
- 4) 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。

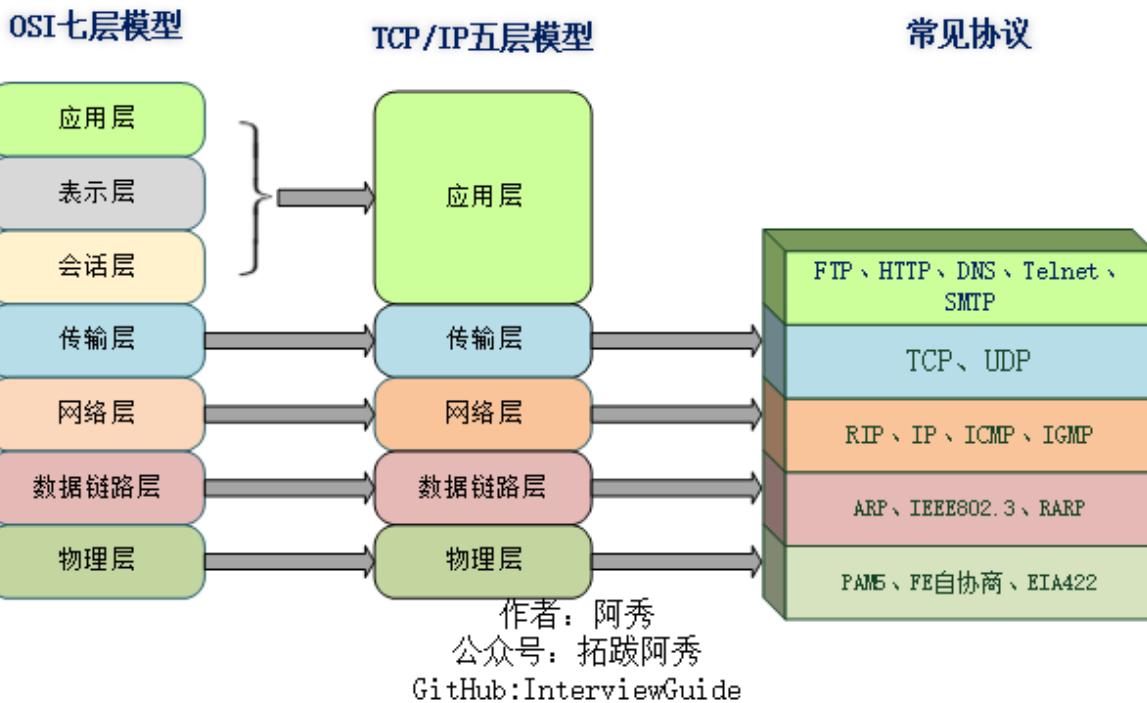
47、TCP头部中有哪些信息?

- 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。客户端接收到服务端的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，而后续的序列号由对端的 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。

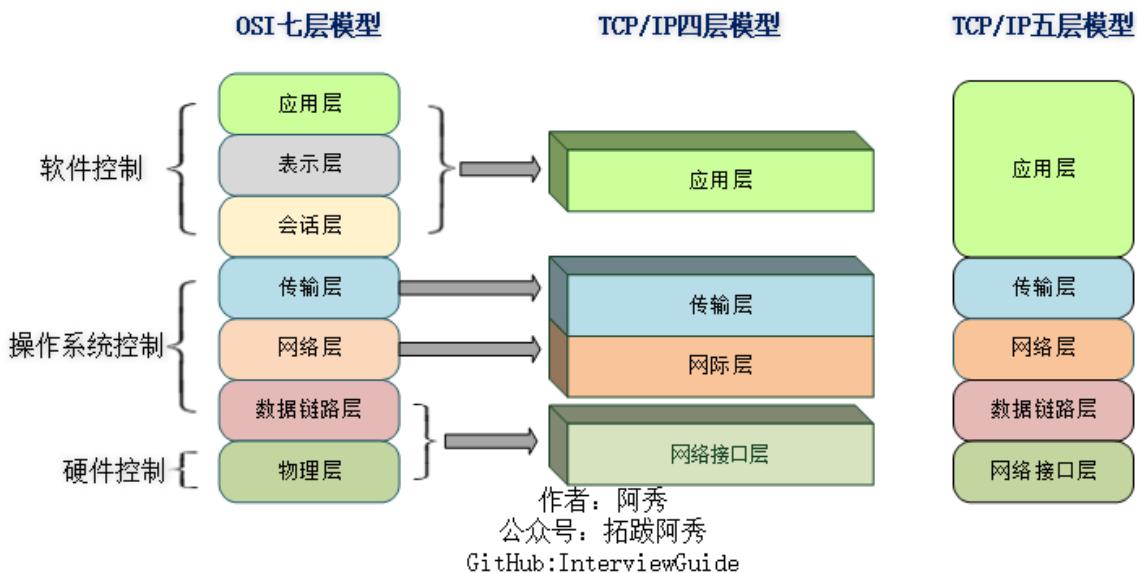
当然只介绍三个：

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

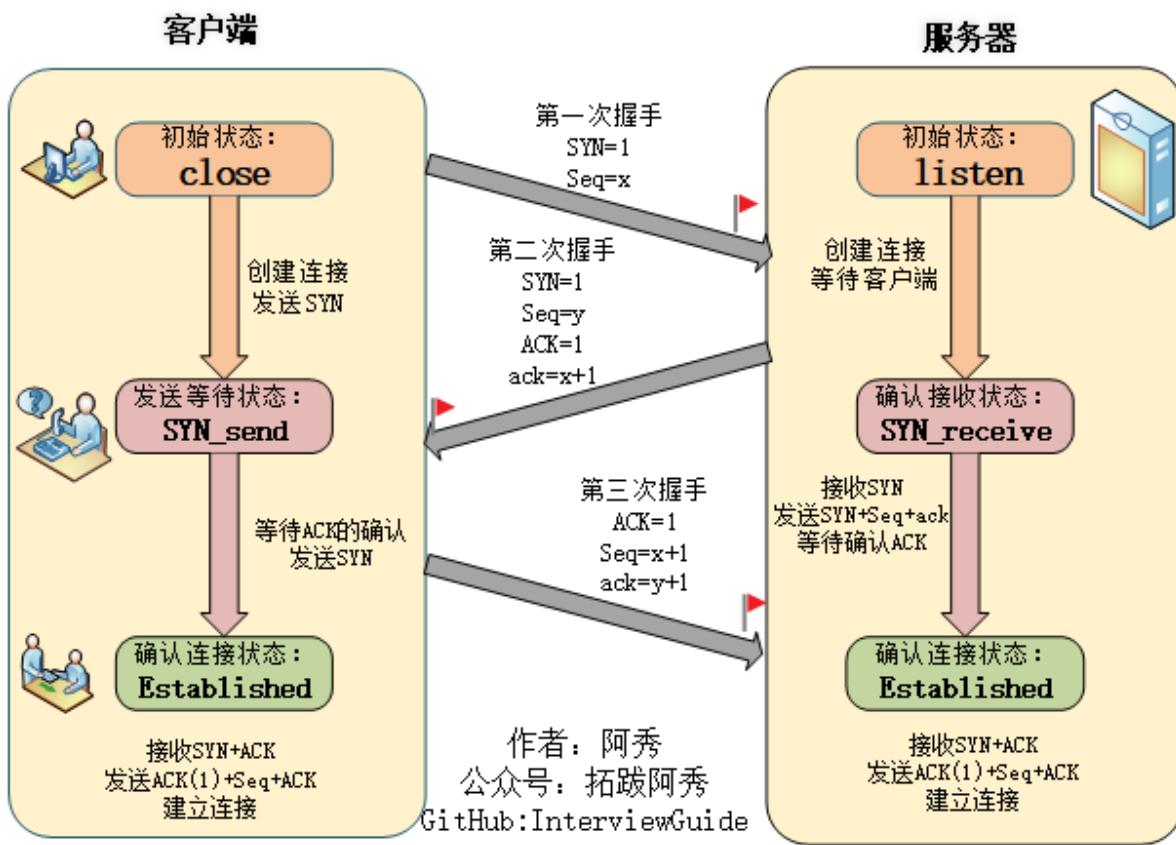
54、浏览器在与服务器建立了一个 TCP 连接后是否会在一个 HTTP 请求完成后断开？什么情况下会断开？

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 SYN_SEND 状态。
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 SYN_RCVD 的状态。

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 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立了连接。

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, 就建立了新的连接了, 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。

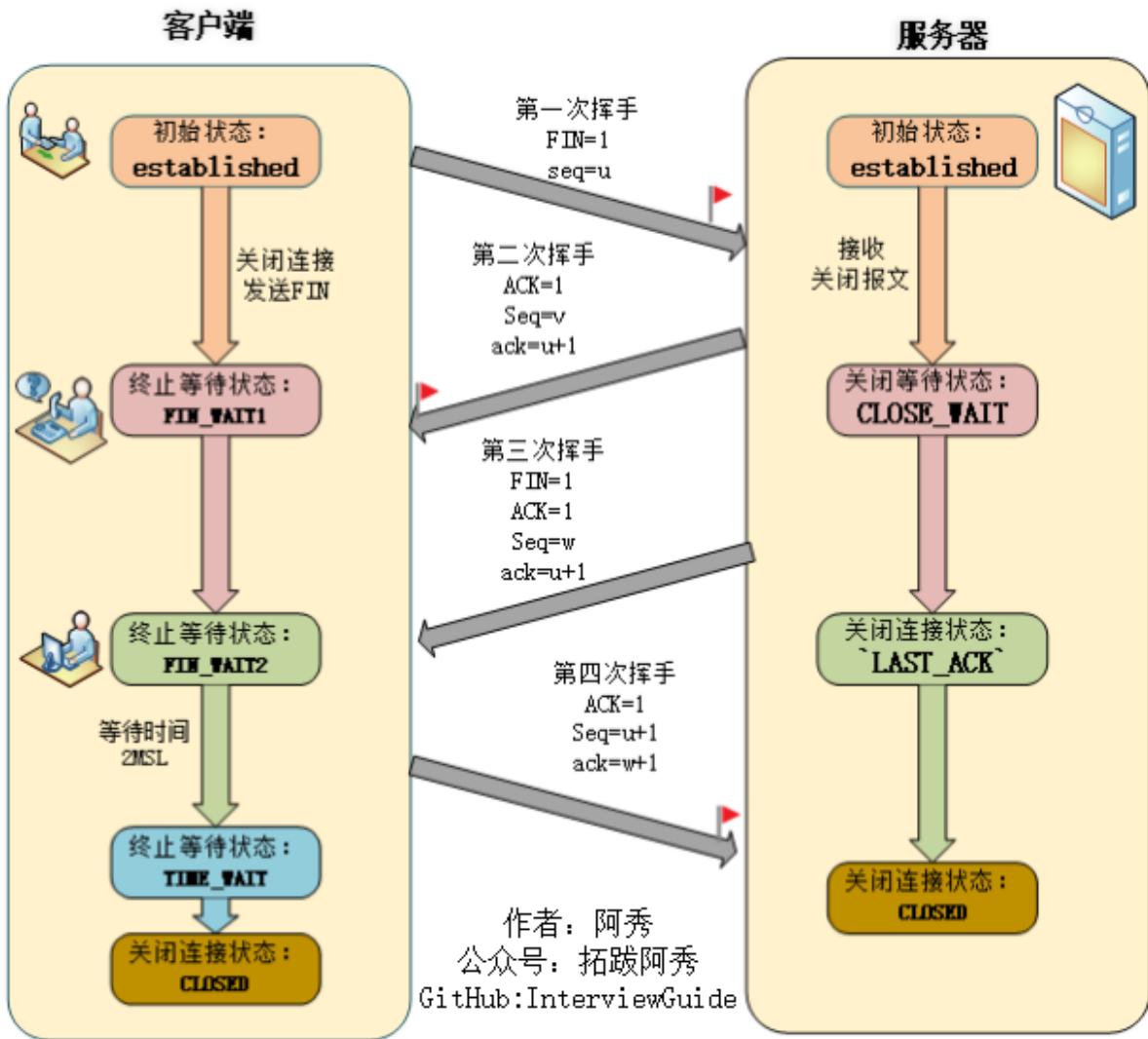
检测 SYN 攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击。在 Linux/Unix 上可以使用系统自带的 netstats 命令来检测 SYN 攻击。

```
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 CLOSE_WAIT 状态。即服务端收到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 LAST_ACK 的状态。即服务端没有要向客户端发出的数据，服务端发出**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 TIME_WAIT 状态。需要过一阵子以确保服务端

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，一旦这个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：

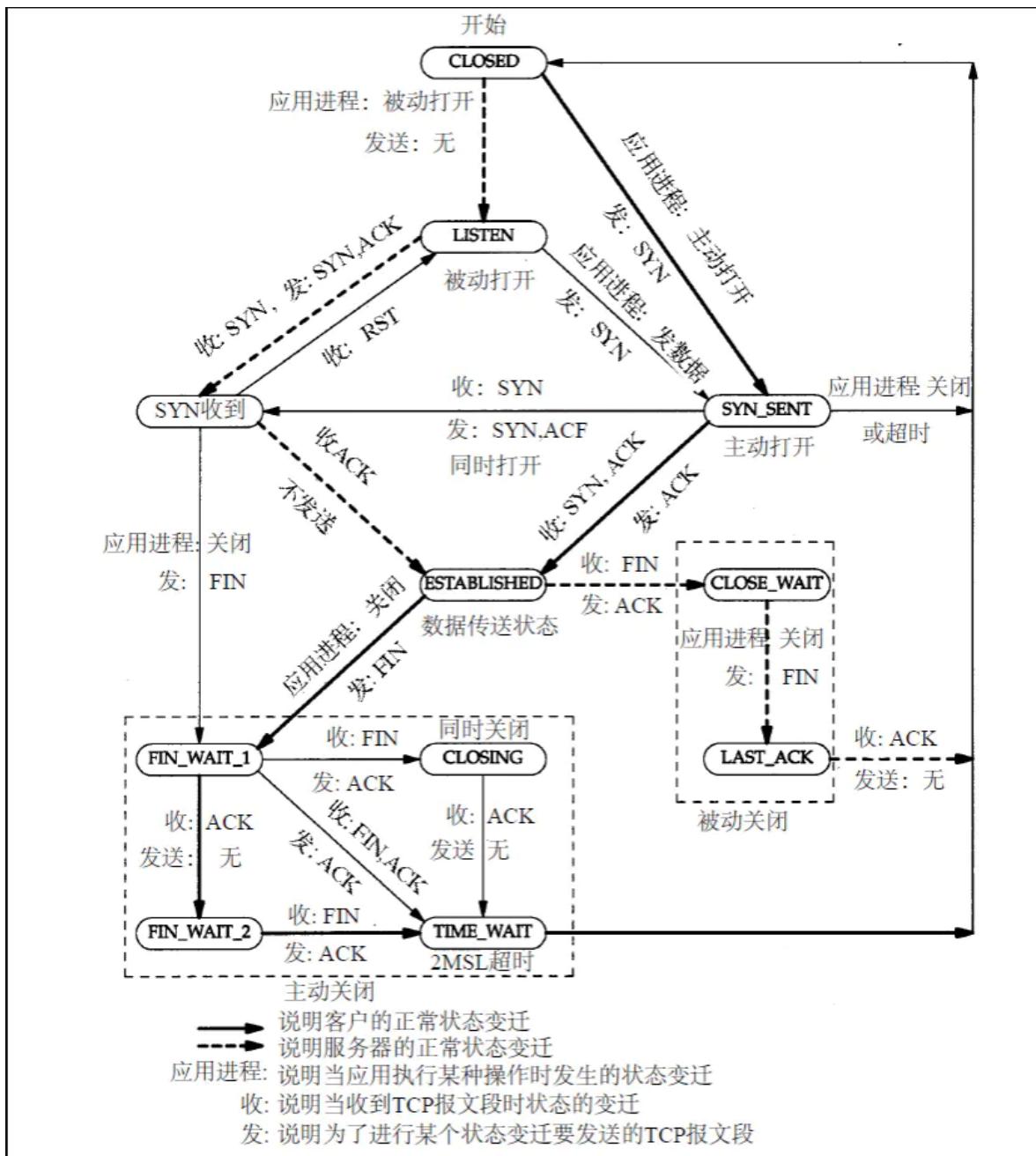
1. **Nagle算法**问题导致的，需要结合应用场景适当关闭该算法
2. 尾部标记序列。通过特殊标识符表示数据包的边界，例如\n\r, \t，或者一些隐藏字符。
3. 头部标记分步接收。在TCP报文的头部加上表示数据长度。
4. 应用层发送数据时**定长发送**。

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

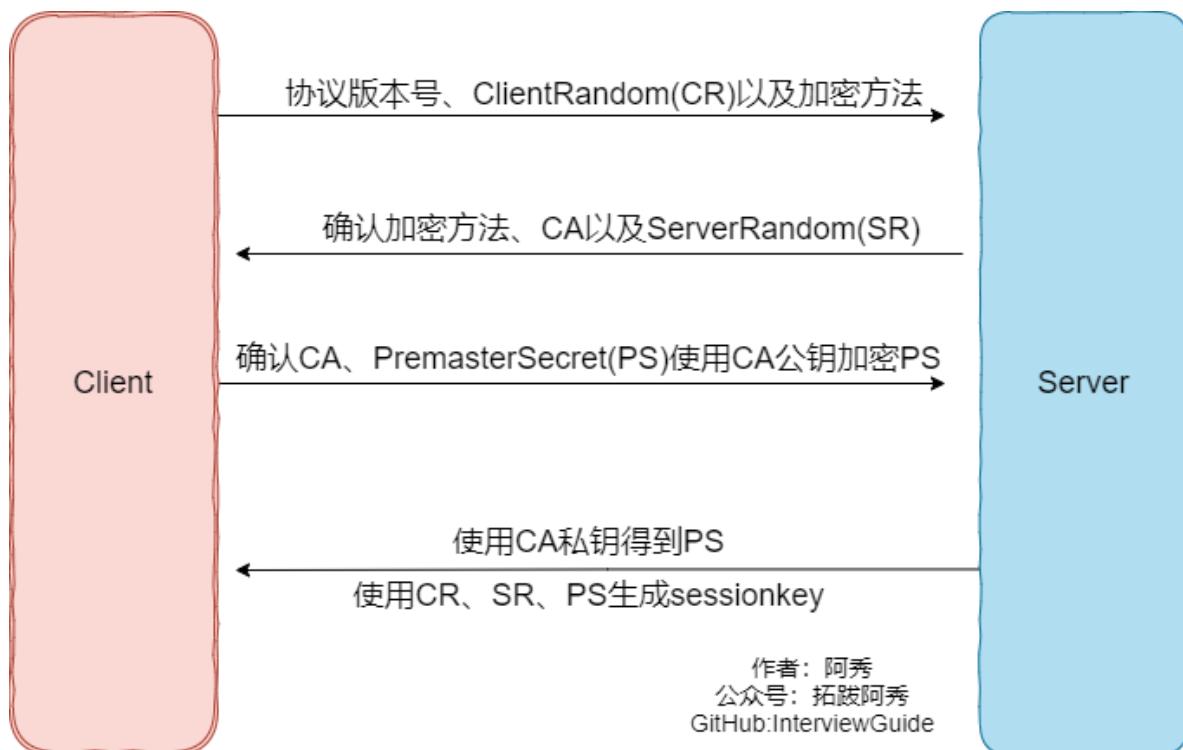
HTTPS 并不是新协议，而是让 HTTP 先和 SSL (Secure Sockets Layer) 通信，再由 SSL 和 TCP 通信，也就是说 HTTPS 使用了隧道进行通信。通过使用 SSL，HTTPS 具有了加密（防窃听）、认证（防伪装）和完整性保护（防篡改）。

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到post的数据量限制，但实际上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。

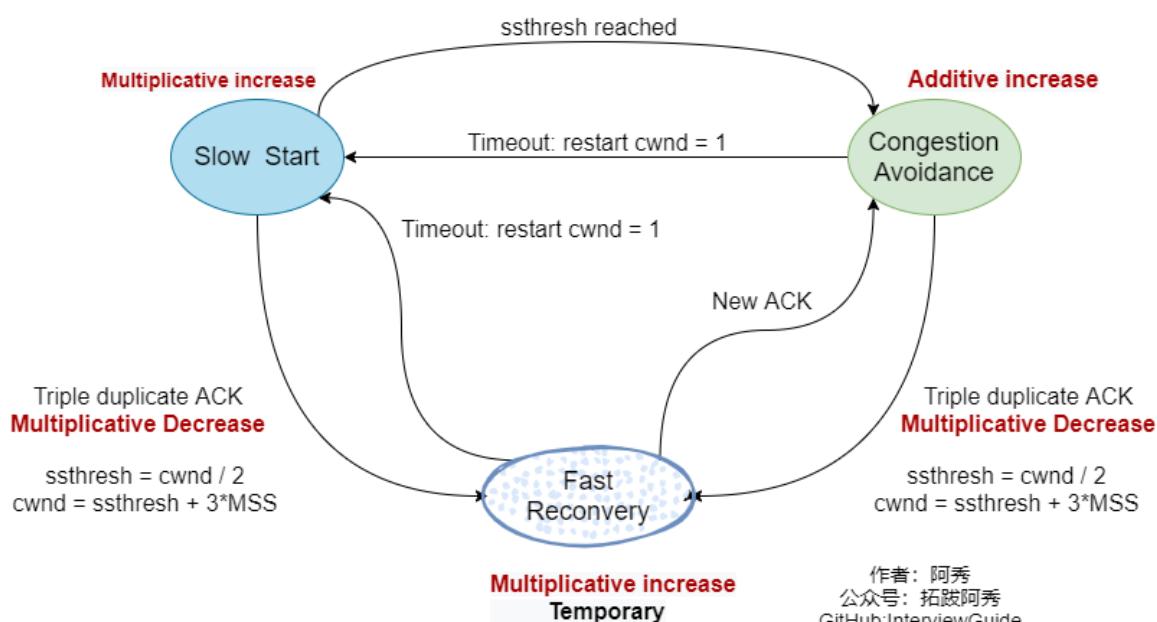
78、网络层常见协议？可以说一下吗？

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。



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 \geq 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。

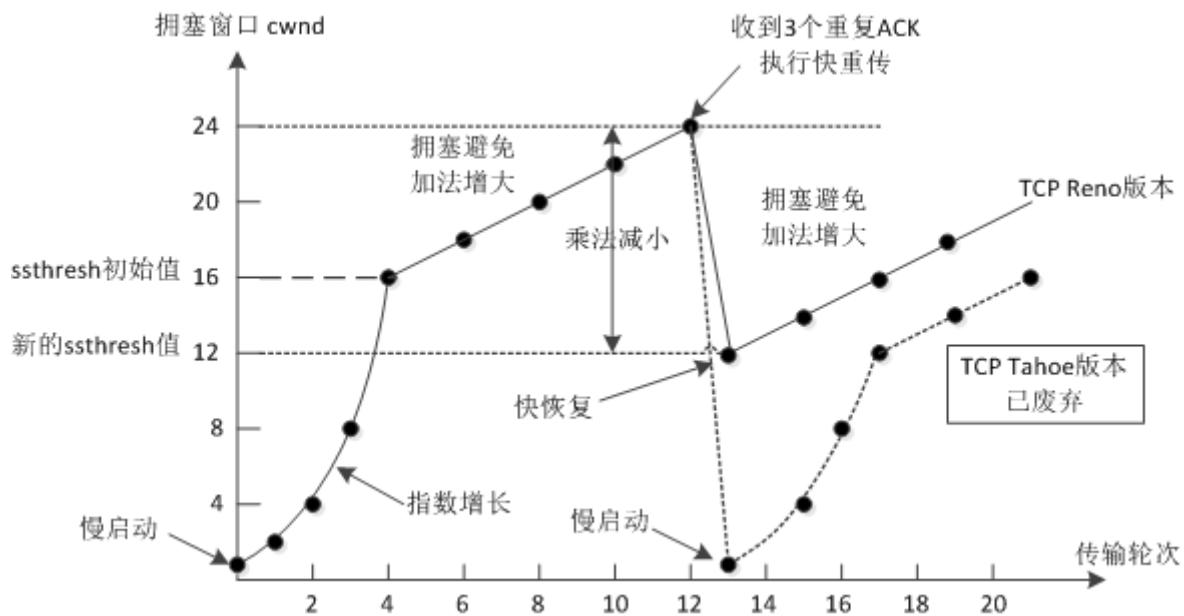
超时重传RTO[Retransmission Timeout]超时，TCP会重传数据包。TCP认为这种情况比较糟糕，反应也比较强烈：

- 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：

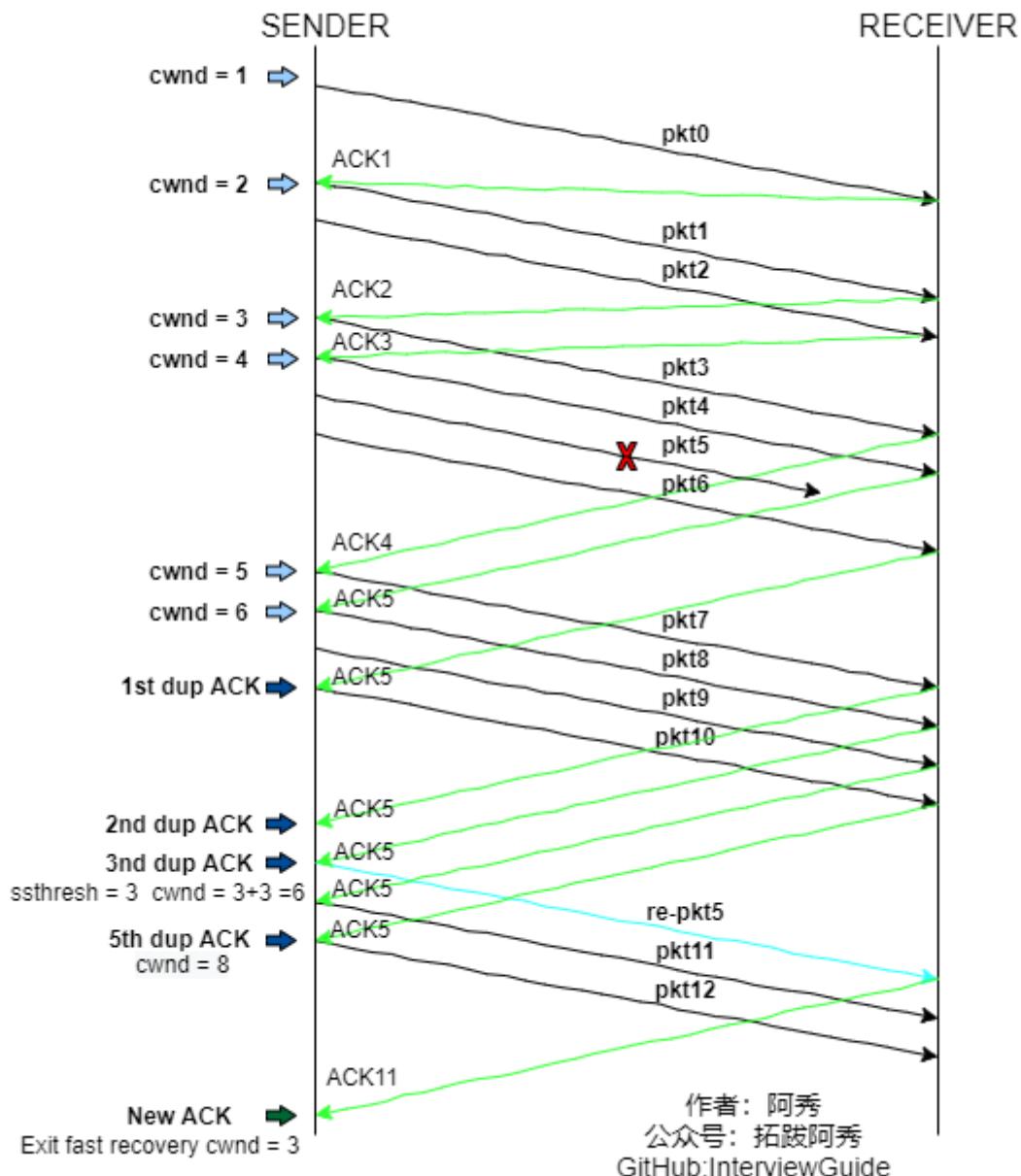
- cwnd大小缩小为当前的一半
- ssthresh设置为缩小后的cwnd大小
- 然后进入快速恢复算法Fast Recovery。



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：

- $cwnd = cwnd + 3 \text{ MSS}$, 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设置为当时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/u010202588/article/details/54563648>

81、对于FIN_WAIT_2, CLOSE_WAIT状态和TIME_WAIT状态？你知道多少？

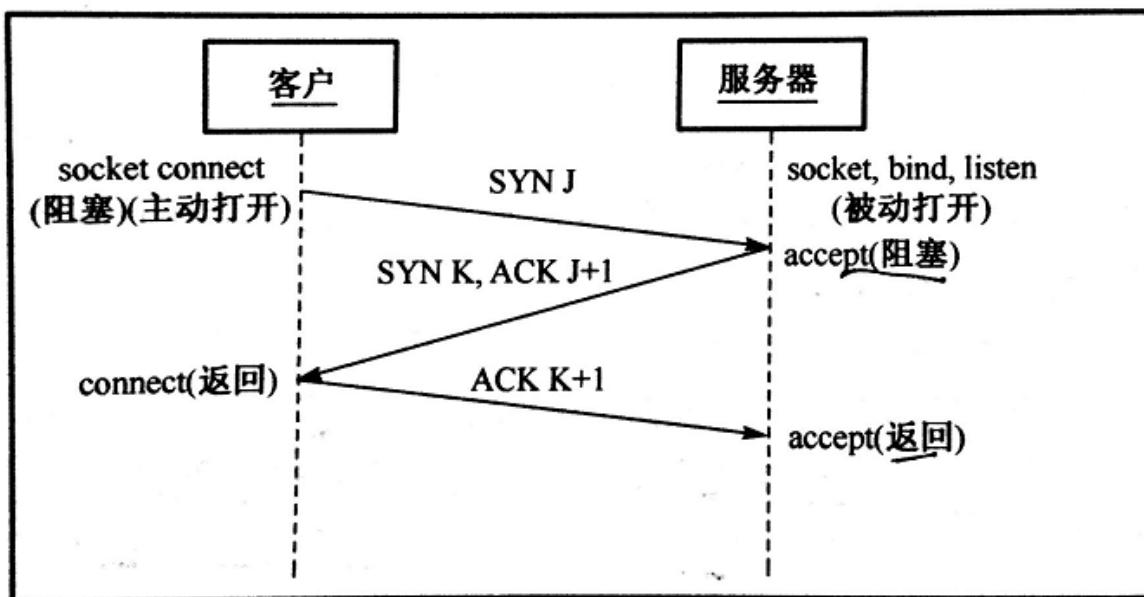
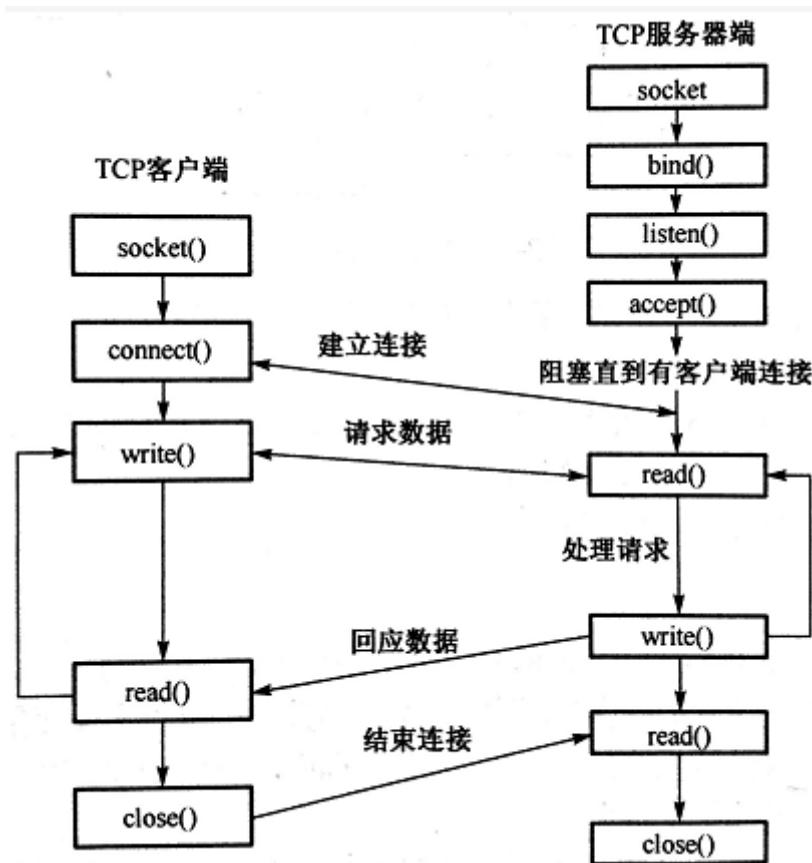
- FIN_WAIT_2:
 - 半关闭状态。
 - 发送断开请求一方还有接收数据能力，但已经没有发送数据能力。
- CLOSE_WAIT状态:
 - 被动关闭连接一方接收到FIN包会立即回应ACK包表示已接收到断开请求。
 - 被动关闭连接一方如果还有剩余数据要发送就会进入CLOSED_WAIT状态。
- TIME_WAIT状态:
 - 又叫2MSL等待状态。
 - 如果客户端直接进入CLOSED状态，如果服务端没有接收到最后一次ACK包会在超时之后重新再发FIN包，此时因为客户端已经CLOSED，所以服务端就不会收到ACK而是收到RST。所以TIME_WAIT状态目的是防止最后一次握手数据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。

```

1 // IPv4的sockaddr地址结构
2 struct sockaddr_in {
3     sa_family_t sin_family;    // 协议类型, AF_INET
4     in_port_t sin_port;        // 端口号
5     struct in_addr sin_addr;   // IP地址
6 };
7 struct in_addr {
8     uint32_t s_addr;
9 }
```

- addrlen: 地址长度。
- 监听端口号 -> int listen(int sockfd, int backlog);
 - sockfd: 要监听的socket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socket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/bjrxzyz/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/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 Protocol)：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/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、XSS攻击是什么? (低频)

跨站点脚本攻击,指攻击者通过篡改网页,嵌入恶意脚本程序,在用户浏览网页时,控制用户浏览器进行恶意操作的一种攻击方式。如何防范XSS攻击

- 1) 前端,服务端,同时需要字符串输入的长度限制。
 - 2) 前端,服务端,同时需要对HTML转义处理。将其中的“<”,“>”等特殊字符进行转义编码。
- 防 XSS 的核心是必须对输入的数据做过滤处理。

96、CSRF攻击? 你知道吗?

跨站点请求伪造，指攻击者通过跨站请求，以合法的用户的身份进行非法操作。可以这么理解CSRF攻击：攻击者盗用你的身份，以你的名义向第三方网站发送恶意请求。CSRF能做的事情包括利用你的身份发邮件，发短信，进行交易转账，甚至盗取账号信息。

97、如何防范CSRF攻击

安全框架，例如Spring Security。

token机制。在HTTP请求中进行token验证，如果请求中没有token或者token内容不正确，则认为CSRF攻击而拒绝该请求。

验证码。通常情况下，验证码能够很好的遏制CSRF攻击，但是很多情况下，出于用户体验考虑，验证码只能作为一种辅助手段，而不是最主要的解决方案。

referer识别。在HTTP Header中有一个字段Referer，它记录了HTTP请求的来源地址。如果Referer是其他网站，就有可能是CSRF攻击，则拒绝该请求。但是，服务器并非都能取到Referer。很多用户出于隐私保护的考虑，限制了Referer的发送。在某些情况下，浏览器也不会发送Referer，例如HTTPS跳转到HTTP。

- 1) 验证请求来源地址；
- 2) 关键操作添加验证码；
- 3) 在请求地址添加 token 并验证。

98、文件上传漏洞是如何发生的？你有经历过吗？

文件上传漏洞，指的是用户上传一个可执行的脚本文件，并通过此脚本文件获得了执行服务端命令的能力。

许多第三方框架、服务，都曾经被爆出文件上传漏洞，比如很早之前的Struts2，以及富文本编辑器等等，可被攻击者上传恶意代码，有可能服务端就被人黑了。

99、如何防范文件上传漏洞

文件上传的目录设置为不可执行。

- 1) 判断文件类型。在判断文件类型的时候，可以结合使用MIME Type，后缀检查等方式。因为对于上传文件，不能简单地通过后缀名称来判断文件的类型，因为攻击者可以将可执行文件的后缀名称改为图片或其他后缀类型，诱导用户执行。
- 2) 对上传的文件类型进行白名单校验，只允许上传可靠类型。
- 3) 上传的文件需要进行重新命名，使攻击者无法猜想上传文件的访问路径，将极大地增加攻击成本，同时向shell.php.rar.ara这种文件，因为重命名而无法成功实施攻击。
- 4) 限制上传文件的大小。
- 5) 单独设置文件服务器的域名。

100、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。

101、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}。

102、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

- **301 Moved Permanently**：永久性重定向
- **302 Found**：临时性重定向
- **303 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。
- **304 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

- **400 Bad Request**：请求报文中存在语法错误。
- **401 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。

3.5、MySQL

正在整理ing, 敬请期待

3.6、Redis

正在整理ing, 敬请期待

3.7、常见智力题、情景题

正在整理ing, 敬请期待

3.8、常见非技术性问题

正在整理ing, 敬请期待

4、优质面经

4.1、阿秀个人秋招总结文章：双非渣硕的秋招之路总结（已拿抖音研发岗SP）

首发于个人公众号上，原文链接：<https://mp.weixin.qq.com/s/AYe3tnuOmqR4jdDndDGW-Q>

前言

最近应邀在牛客网写 C++ 求职专栏，又把以前的秋招总结补充了很多东西，现在想想还是发出来，希望能够帮助更多的新手小伙伴们。



个人情况简介

楼主本硕均读于双非院校（普通二本学校）、本硕都是计算机相关专业，英语六级水平，本科时期辅修了一个水的不能再水的英语第二学位。

本科时期学过很多语言：**VB、C、C++、C#、Java**都有所涉猎，研究生时期则主攻**Python**和**C++**。研二上学期开始系统学习**C++**，并且不断系统看书和实践，中间崩溃过、迷茫过、放纵过，但从未放弃，始终相信自己，坚持咬牙走下去。所幸天道酬勤，最终也是拿到了一些不错的**offer**。

投递经历

笔者从 **2020.6.15** 号正式开始投递简历，到**2020.8.23**号截止一共投递过 **94** 家公司，其中既有提前批（**2020年6月-7月**），也包括正式批（**2020年7月-10月**）。

小建议：如果说求职者对自身实力不自信，可以多投投一些公司，选择面放宽一些，不要死盯着那几个大厂投。

共计笔试**59**场（最多一天做了**5**场笔试，那天差点去世），**54**家公司给了面试机会，**54**家企业中有些企业是免笔试的。

秋招结果

最终成功走到了**6**家公司的**offer**环节：**字节跳动研发岗SP、华为通用软件开发、百度C++研发岗、B站后端研发岗、深信服C++研发岗以及农业银行研发岗**，最后签了**字节跳动**，也是自己心心念念的大厂之一，十分满意了~

接下来从6个方面对秋招进行复盘和总结，希望能够帮到大家鸭，特别是大三大四的小学弟们。

1、算法

在秋招过程中，算法是**极其重要的**，再次重申一遍，真的很重要！笔试就不提了，算法不过关，笔试基本凉凉，面试基本都要手撕代码，很多面试过程中算法题具有一票否决权，如果你能够顺利解出来，面试也不一定过。即使面试过了，手撕代码没撕出来，面评估估也是一般般了。但是如果算法题做不出来或者说**bug**太多调试不通的话，面试上基本上就跪了（个人以及身边朋友经历，不一定准确），在牛客网上也看到过很多基础很好的牛友就是因为面试过程中的算法题没解出来而直接饮恨的，希望大家千万重视算法这一块，千万要重视算法。

我大概在力扣上刷了**300+**，**HOT100**都刷了，剑指**offer**刷了3遍，刷完这些基本够用了，自己也有注意总结题型，常见题型就是那些，所以算法题基本没怎么拉过我后腿。一般来说，主要考的就是动态规划、贪心、二叉树、链表、数组、字符串之类的。

推荐资料：

力扣1-300题（前300道题非常经典，建议学有余力的同学都刷一刷）

力扣HOT100（跟上面有不少是重复的，刷的时候要注意总结）

啊哈！算法、大话数据结构（这两本书都是面向新手的图书，图画很多）

剑指offer（这本书不需要多做介绍，校招必备）

挑战程序设计竞赛（这本书属于进阶一点的算法书籍了，作者是ACM-ICPC全球总冠军，可以说是世界顶级程序设计高手的经验总结了，需要慢慢消化，经典题型太多）

程序员代码面试指南（左程云大神的书，我并没有看完，只是看了其中的海量数据处理部分的题目就已经十分受用了，在某大厂三面中就考查到了其中的海量数据集处理的问题）

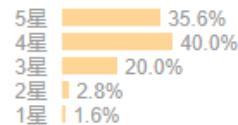
啊哈!算法



作者: 啊哈磊
出版社: 人民邮电出版社
出版年: 2014-6-1
页数: 246
定价: 45.00元
装帧: 平装
丛书: 图灵原创
ISBN: 9787115354594

豆瓣评分

7.7 ★★★★★ 505人评价



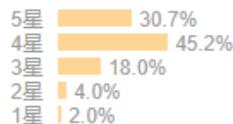
大话数据结构



作者: 程杰
出版社: 清华大学出版社
出版年: 2011-6
页数: 440
定价: 59.00元
装帧: 平装
丛书: 大话系列
ISBN: 9787302255659

豆瓣评分

7.9 ★★★★★ 1370人评价



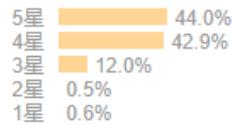
剑指Offer



作者: 何海涛
出版社: 电子工业出版社
出品方: 博文视点
副标题: 名企面试真经精讲典型编程题
出版年: 2012-1
页数: 260
定价: 45.00元
装帧: 平装
ISBN: 9787121148750

豆瓣评分

8.3 ★★★★★ 786人评价



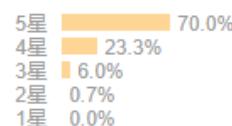
挑战程序设计竞赛



作者: [日]秋叶拓哉 / [日]岩田阳一 / [日]北川宣稔
出版社: 人民邮电出版社
原作名: プログラミングコンテストチャレンジブック [第2版] ~問題解決のアルゴリズム活用力とコーディングテクニックを鍛える~
译者: 巫泽俊 / 庄俊元 / 李津羽
出版年: 2013-7-1
页数: 414
定价: CNY 79.00
装帧: 平装
ISBN: 9787115320100

豆瓣评分

9.0 ★★★★★ 300人评价



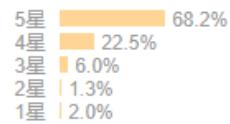
程序员代码面试指南：IT名企算法与数据结构题目最优解



作者: 左程云
出版社: 电子工业出版社
出品方: 博文视点
副标题: IT名企算法与数据结构题目最优解
出版年: 2015-9
页数: 532
定价: 79.00元
装帧: 平装
ISBN: 9787121270116

豆瓣评分

8.9 ★★★★★ 151人评价



2、操作系统

操作系统是比较重要的，面试三大要点之一（操作系统、计网、数据库），我是在B站上看过一些操作系统视频，同时自己慢慢看书、看博客学的。其中死锁、虚拟内存、堆栈、进程线程、内存管理、磁盘调度等都是重点，也是面试过程中问的比较多的一些知识点。你如果能够在面试过程中讲出来一些具体的操作系统知识，而不是泛泛而谈，肯定是很加分的，比如常见知识点进程线程区别，在提到线程切换比进程更快时，你如果能够很清楚明白的说出来进程切换做了哪些、线程切换做了哪些以及线程为什么比进程快，毫无疑问很加分的。

推荐资料：

B站哈工大操作系统：<https://www.bilibili.com/video/BV1d4411v7u7>

B站清华大学操作系统：<https://www.bilibili.com/video/BV1js411b7vg>

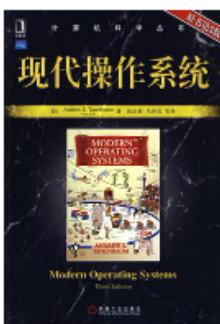
B站美国麻省理工MIT 6.828操作系统神级课程：<https://www.bilibili.com/video/BV1px411E7ST>

现代操作系统（也是讲操作系统的一本好书，讲的很细）

深入理解计算机系统（大名鼎鼎的CSAPP，被誉为“和金子一样重要的计算机基础书籍”，很厚的一本黑皮书，需要慢慢看）

现代操作系统：原理与实现(上海交通大学陈海波教授的著作，书中主要介绍操作系统的理论与具体实现细节等，感觉不如CSAPP)

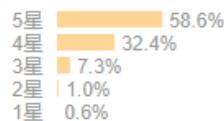
现代操作系统 (第3版)



作者: [美] Andrew S.Tanenbaum
出版社: 机械工业出版社
原作名: Modern Operating Systems
译者: 陈向群 / 马洪兵
出版年: 2009-7
页数: 582
定价: 75.00元
装帧: 平装
丛书: 计算机科学丛书
ISBN: 9787111255444

豆瓣评分

8.9 ★★★★★ 777人评价



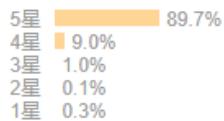
深入理解计算机系统 (原书第3版)



作者: Randal E.Bryant / David O'Hallaron
出版社: 机械工业出版社
原作名: Computer Systems: A Programmer's Perspective (3rd Edition)
译者: 龚奕利 / 贺莲
出版年: 2016-11
页数: 737
定价: 139.00元
装帧: 平装
丛书: 计算机科学丛书
ISBN: 9787111544937

豆瓣评分

9.8 ★★★★★ 1192人评价



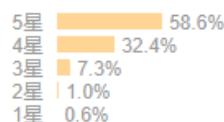
现代操作系统 (第3版)



作者: [美] Andrew S.Tanenbaum
出版社: 机械工业出版社
原作名: Modern Operating Systems
译者: 陈向群 / 马洪兵
出版年: 2009-7
页数: 582
定价: 75.00元
装帧: 平装
丛书: 计算机科学丛书
ISBN: 9787111255444

豆瓣评分

8.9 ★★★★★ 777人评价



3、计算机网络

计算机网络也是重点之一，特别是HTTP以及TCP/UDP相关知识点，算是校招必备考点了，面试必问，但是难度是逐年上升的，原因可能就在于内卷程度越来越严重了吧。比如说以前对于三次握手四次挥手只问过程，现在直接让面试者画出客户端以及服务器端的各个状态码以及解释各种意外情况，比如SYN请求丢失会怎么样？

建议计网的学习先从视频入手，然后再看经典书籍，毕竟视频中的知识都是别人总结好又给你讲解的，只有自己亲自揣摩、亲自动手实践得来的知识才是自己的，自己学来的才是真，经过实践方知分晓的~

推荐资料：

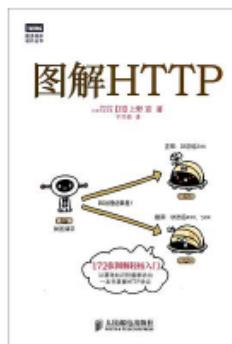
B站韩立刚老师的计算机网络（韩老师讲课诙谐易懂，让你在哈哈大笑中学到很多知识点：<http://www.bilibili.com/video/BV17p411f7ZZ>）

图解HTTP、图解TCP/IP（这两本书比较简单，日本人写的，把复杂的知识点简单化）

网络是怎样连接的（这本书紧紧围绕一个问题：输入一个URL，直到我们在网页端看到请求的内容，这中间发生了什么？抽丝剥茧将这个问题逐步细化，带你走完整个网页访问的过程）

计算机网络：自顶向下方法（也是常见经典书籍之一，重点看第三章传输层TCP/UDP）

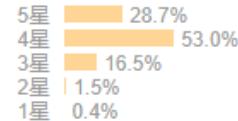
图解HTTP



作者: [日] 上野宣
出版社: 人民邮电出版社
出品方: 图灵教育
译者: 于均良
出版年: 2014-4-15
页数: 308
定价: 49.00元
装帧: 平装
丛书: 图灵程序设计丛书·图解与入门系列
ISBN: 978711531531

豆瓣评分

8.1 ★★★★★ 2617人评价



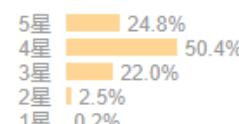
图解TCP/IP (第5版)



作者: [日]竹下隆史 / [日]村山公保 / [日]荒井透 / [日]舛田幸雄
出版社: 人民邮电出版社
出品方: 图灵教育
原作名: マスタリングTCP/IP 入門編 第5版
译者: 乌尼日其其格
出版年: 2013-7-1
页数: 312
定价: 69.00元
装帧: 平装
丛书: 图灵程序设计丛书·图解与入门系列
ISBN: 9787115318978

豆瓣评分

7.9 ★★★★★ 944人评价



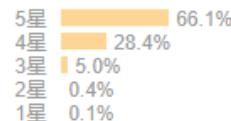
网络是怎样连接的



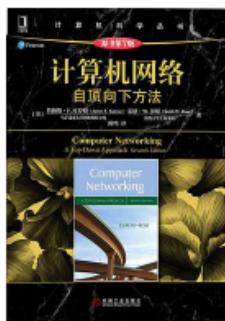
作者: [日]户根勤
出版社: 人民邮电出版社
出品方: 图灵教育
原作名: ネットワークはなぜつながるのか 第2版
译者: 周自恒
出版年: 2017-1-1
页数: 336
定价: CNY 49.00
装帧: 平装
丛书: 图灵程序设计丛书·图解与入门系列
ISBN: 9787115441249

豆瓣评分

9.2 ★★★★★ 1151人评价



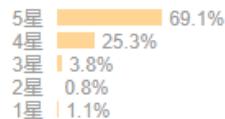
计算机网络 (原书第7版)



作者: James F. Kurose / Keith W. Ross
出版社: 机械工业出版社
副标题: 自顶向下方法
原作名: Computer Networking: A Top-Down Approach
译者: 陈鸣
出版年: 2018-6
页数: 480
定价: 89.00元
装帧: 平装
丛书: 计算机科学丛书
ISBN: 9787111599715

豆瓣评分

9.2 ★★★★★
265人评价



4. Linux

C++跟Linux基本是离不开的，特别是后端方向跟网络通信关系很大。在实际工作里，很多成熟的项目都是在Linux上进行开发的。所以有必要学一些Linux以及一些网络通信编程，网络通信涉及到的知识点很多，比如IO模型、线程池、多线程之类的。本人在秋招过程中被问过不少网络通信的问题，最频繁的就是select、poll、epoll的区别以及相关底层实现了。这里也推荐一些资料，都是我个人看过的。

推荐资料:

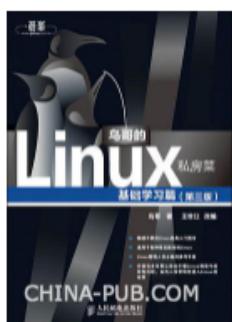
鸟哥的Linux以及Linux就该这么学这两本书 (个人感觉更适合作为一本工具书来使用，当然了，如果你有充足的时间也可以系统的看上一遍，对于Linux也会有更深的认识和了解了)

TCP/IP网络编程 (韩国人写的，书中例子很多，适合作为入门，另外github上有很多笔记，可以边看别人的笔记边看书，加深个人理解)

Linux高性能服务器编程 (游双老师的书，其中前四五章讲的是计网的东西，后面讲的很好，涉及内容很多，看完就大概明白服务端编程常见知识点和所需要掌握的技能了)

Linux多线程服务端编程：使用muduo C++网络库 (北师大陈硕大神的书，需要很多基本，建议后期再看，我也只是看了一小半)

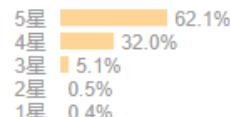
鸟哥的Linux私房菜



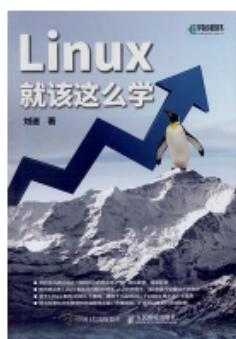
作者: 鸟哥
出版社: 人民邮电出版社
副标题: 基础学习篇
出版年: 2010-6-28
页数: 778
定价: 88.00元
装帧: 平装
丛书: 鸟哥的Linux私房菜
ISBN: 9787115226266

豆瓣评分

9.1 ★★★★★
3093人评价



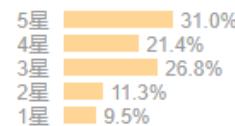
Linux就该这么学



作者: 刘遄
出版社: 人民邮电出版社
出品方: 异步图书
副标题: 必读的Linux系统与红帽认证自学书籍
出版年: 2017-11-1
页数: 450
定价: 79
装帧: 平装
ISBN: 9787115470317

豆瓣评分

5.8 ★★★★★ 168人评价



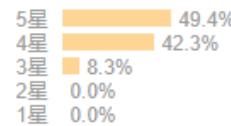
TCP/IP网络编程



作者: [韩] 尹圣雨
出版社: 人民邮电出版社
译者: 金国哲
出版年: 2014-7
页数: 410
定价: 79.00元
装帧: 平装
丛书: 图灵程序设计丛书
ISBN: 9787115358851

豆瓣评分

8.6 ★★★★★ 168人评价



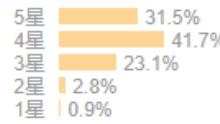
Linux高性能服务器编程



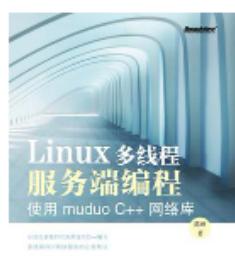
作者: 游双
出版社: 机械工业出版社
出版年: 2013-5-1
页数: 360
定价: CNY 69.00
装帧: 平装
丛书: 实战系列
ISBN: 9787111425199

豆瓣评分

7.9 ★★★★★ 216人评价



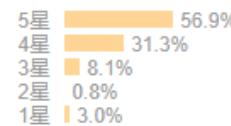
Linux多线程服务端编程



作者: 陈硕
出版社: 电子工业出版社
出品方: 博文视点
副标题: 使用muduo C++网络库
出版年: 2013-1-15
页数: 610
定价: 89.00元
装帧: 平装
ISBN: 9787121192821

豆瓣评分

8.8 ★★★★★ 508人评价



5、数据库

数据库主要问的都是MySQL以及Redis相关的一些知识，普通研发岗掌握这两个基本也够用了，数据库常问知识点包括索引相关、性能优化、B+树、Redis底层模型、跳表以及缓存击穿、雪崩、穿透等常见问题。有时候也会让你手写一些简单的SQL语句，比如给你一个学生表和课程表，让你找出成绩排名前十的学生姓名之类的。

推荐资料：

MySQL必知必会（一本很薄的小册子，不到一周就看完了，看完基本的SQL语句没什么问题了）

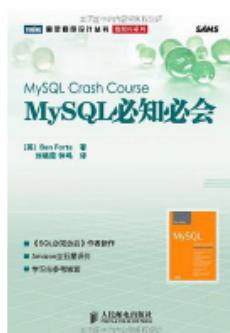
高性能MySQL（建议只看索引以及优化这两章，后续的可以慢慢再看，这本书，真的真的很厚。）

Redis设计与实现（算是Redis入门资料吧，认真看完的话就对Redis有大概了解了，话说Redis这么火爆是有原因的，其中的一些精妙设计真的看完令人大呼过瘾，不得不承认，人与人之间真是有差距的。。。）

极客时间- Redis核心技术与实战（中科院的研究员开设的Redis专栏，个人已经买了，非常不错）

另外再推荐基本数据库底层的书籍：**数据库系统实现**（华东师范大学数据学院指定数据库原理书籍）、**MySQL技术内幕 -InnoDB存储引擎**（InnoDB的详细剖析）

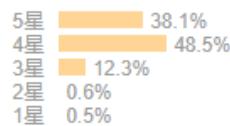
MySQL必知必会



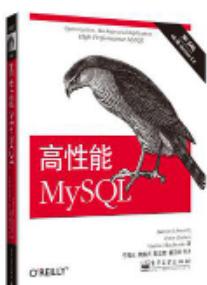
作者: [英] Ben Forta
出版社: 人民邮电出版社
原作名: MySQL Crash Course
译者: 刘晓霞 / 钟鸣
出版年: 2009-1
页数: 241
定价: 39.00元
丛书: 图灵程序设计丛书·数据库系列
ISBN: 9787115191120

豆瓣评分

8.4 1442人评价



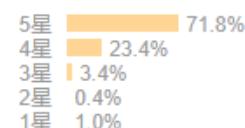
高性能MySQL(第3版)



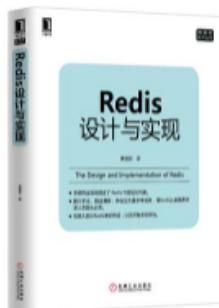
作者: 施瓦茨 (Baron Schwartz) / 扎伊采夫 (Peter Zaitsev)
/ 特卡琴科 (Vadim Tkachenko)
出版社: 电子工业出版社
副标题: 第3版
原作名: High Performance MySQL,3rd
译者: 宁海元 / 周振兴 / 彭立勋 / 翟卫祥 / 刘辉
出版年: 2013-5-1
页数: 764
定价: 128.00元
装帧: 平装
ISBN: 9787121198854

豆瓣评分

9.3 791人评价



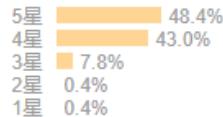
Redis设计与实现



作者: 黄健宏
出版社: 机械工业出版社
出版年: 2014-6
页数: 388
定价: 79.00
装帧: 平装
丛书: 数据库技术丛书
ISBN: 9787111464747

豆瓣评分

8.6 ★★★★★
1059人评价



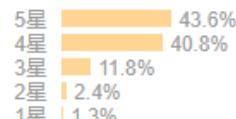
MySQL技术内幕



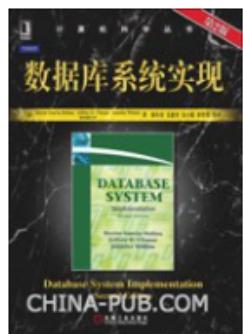
作者: 姜承尧
出版社: 机械工业出版社
副标题: InnoDB存储引擎(第2版)
出版年: 2013-5
页数: 436
定价: 79.00元
丛书: 数据库技术丛书
ISBN: 9787111422068

豆瓣评分

8.5 ★★★★★
456人评价



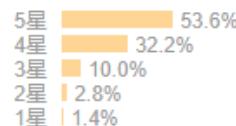
数据库系统实现



作者: 加西亚-莫利纳(Hector Garcia-Molina) / Jeffrey D.Ullman / Jennifer Widom
出版社: 机械工业出版社
原作名: Database System Implementation, Second Edition
译者: 杨冬青 / 吴愈青 / 包小源
出版年: 2010-5
页数: 385
定价: 59.00元
装帧: 平装
丛书: 计算机科学丛书
ISBN: 9787111302872

豆瓣评分

8.7 ★★★★★
211人评价



6、C++

C++的知识点比较多，也比较细，其实C++并不容易学好，如果你只是简单学习一下语法比如for循环、变量类型之类的，那么一两周你就可以上手，但是如果想要学好C++还是需要持之以恒的coding，由于个人是C++技术栈，这里也只是推荐C++相关书籍和视频，都是本人自己看过的经典书籍和资料。

推荐资料：

B站黑马C++视频 (黑马机构出版的入门级C++教学视频，很不错：<https://www.bilibili.com/video/BV1Tb411j7uM>)

STL源码剖析视频 (C++大师侯捷老师的源码视频，搭配STL源码剖析看效果更佳：<https://www.bilibili.com/video/BV1db411q7B8>)

C++ Primer 第五版 (我愿称之为C++圣经，800页左右，我看了2遍，超级棒！强推！)

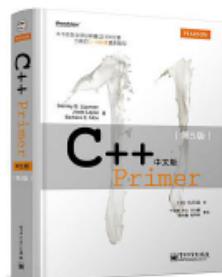
Effective C++、More Effective C++(前者2遍，后者1遍，跟C++Primer中很多内容是有重复的)

STL源码剖析 (源码方面的好书，看了2遍，现在时不时还拿出来翻翻)

深入探索C++对象模型 (重点是虚函数那一章，看完你就会对虚函数有新的认识了)

No.9 豆瓣热门编程图书TOP 10

C++ Primer 中文版 (第5版)



作者: [美] Stanley B. Lippman / [美] Josée Lajoie / [美]

Barbara E. Moo

出版社: 电子工业出版社

出品方: 博文视点

原作名: C++ Primer, 5th Edition

译者: 王刚 / 杨巨峰

出版年: 2013-9-1

页数: 838

定价: CNY 128.00

装帧: 平装

ISBN: 9787121155352

豆瓣评分

9.4



1892人评价

5星 79.0%

4星 17.6%

3星 2.4%

2星 0.4%

1星 0.6%

Effective C++



作者: [美] Scott Meyers

出版社: 电子工业出版社

出品方: 博文视点

副标题: 改善程序与设计的55个具体做法

原作名: Effective C++: 55 Specific Ways to Improve Your Programs and Designs

译者: 侯捷

出版年: 2006-7

页数: 297

定价: 58.00元

装帧: 简装本

ISBN: 9787121029097

豆瓣评分

9.5



1172人评价

5星 77.6%

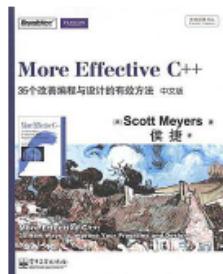
4星 20.4%

3星 1.8%

2星 0.2%

1星 0.1%

More Effective C++ (中文版)



作者: 梅耶(Scott Meyers)

出版社: 电子工业出版社

出品方: 博文视点

副标题: 35个改善编程与设计的有效方法

译者: 侯捷

出版年: 2011-1-1

页数: 317

定价: 59.00元

装帧: 平装

丛书: 传世经典书丛

ISBN: 9787121125706

豆瓣评分

9.2



326人评价

5星 68.4%

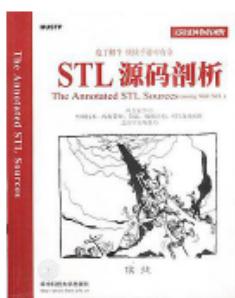
4星 27.9%

3星 3.7%

2星 0.0%

1星 0.0%

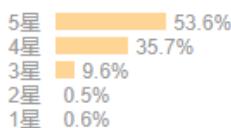
STL源码剖析



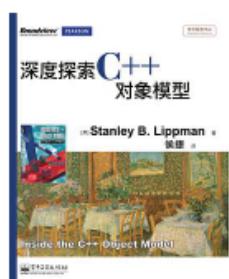
作者: 侯捷
出版社: 华中科技大学出版社
出版年: 2002-6
页数: 493
定价: 68.00元
装帧: 平装
ISBN: 9787560926995

豆瓣评分

8.7 ★★★★★
1359人评价



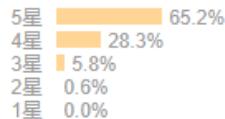
深度探索C++对象模型



作者: 斯坦利·B·李普曼 (Stanley B. Lippman)
出版社: 电子工业出版社
出品方: 博文视点
原作名: Inside the C++ Object Model
译者: 侯捷
出版年: 2012-1-1
页数: 320
定价: 69.00元
装帧: 平装
丛书: 传世经典书丛
ISBN: 9787121149528

豆瓣评分

9.0 ★★★★★
325人评价



碎碎念

可能有些人会问，这些书你都看了吗？这也太多了之类的？自己能不能看完？

说实话，看着是挺多，但是其中有很多知识点是一样的，比如你详细了解计网后，游双老师的那本Linux高性能服务端编程中的前四章你大概略过即可，就不再需要细看了，知识是有相关性和相通性的，有了前面的沉淀后期自然就好很多了。

还有就是学会善用目录。有时候，看过目录后就大概知道这章或者这小节讲的是什么了，建议在看一本书的时候先看一遍目录，挑选出自己不懂得或者感兴趣的章节来看，而将已看过的或者暂时不需要的放到后期再去看。

慢慢学、慢慢看，慢慢的就会有收获了。

如果你像我一样学校不太好，不是什么重点学校或者不是计算机专业的，那么请你笨鸟先飞，赢在起跑线上。上面的书籍资料之类的，我并不是在5个月内看完的，自从学C++以来就慢慢看、慢慢学的，我想其他语言，Java/Go之类的也应该如此。

正所谓，天道酬勤，你付出汗水和努力，剩下的交给时间就好！

最近在看汇编语言，王爽老师的那本《汇编语言》讲得真好，以前的那些寄存器、数据总线、地址总线概念忽然变得很清晰明了、活灵活现了，我自己也慢慢学会使用汇编写一些程序，懂得一些指令级程序优化的思路和方法，算是沉迷其中不可自拔吧！哈哈~

一入IT就做好终生学习的准备吧，你既然想要拿别人拿不了的高薪，怎么能不付出比别人多的汗水、时间和精力呢？天上掉馅饼是不可能的，如果你还在想着偷懒耍滑，想要不付出时间和汗水就想拿到好offer，说明你并不是很适合计算机这一行~

结语

如果你没有别人聪明，不如别人条件好，如果你下定决心学习计算机，请你多投入时间、多投入精力、多投入汗水！

4.2、朋友先后折戟腾讯、字节、快手、网易、滴滴、深信服后，终于成功上岸了

首发于个人公众号上，原文链接：<https://mp.weixin.qq.com/s/MsaAr1ofstCgxqs749W1wg>

大家好，我是阿秀

阿秀粉丝群里一位小伙伴顺利上岸百度实习岗了，把他的经历分享给大家！

本文发布相关内容，已取得粉丝本人同意~

新年就该分享一些喜庆的事情鸭！



背景介绍

本人本科就读于某计算机评级为 B 的双非学校，大学四年完全晃悠过去了。

本科时完全没有和工作有关的概念，就是该上课上课，老师的作业能混就混，后来随大流，迷迷糊糊去考研。

结果可想而知，考研结果比较差，大四春季跑了春招，被吊打的体无完肤。

还好后来，调剂去了一所比较差的一本学校，有点水往低处流了。

到了那里以后，才知道本科时候都在浪费时间，所以决心要好好找工作。

从研一开始就选择了C++，不为别的，就因为C++技术栈基本上只有大厂有，也就是想断掉自己去小厂的路，逼一逼自己。研一寒假开始刷题，我主要是刷力扣和剑指offer，其中力扣已经刷了440道了。

解决问题

440



简单

176/550

中等

229/1014

困难

35/403

由于网课的存在，刷题的时间多了起来，并在那个时候看了 redis 的源码和游双老师的 Linux 服务器开发，后面还看了 muduo (这个真没啥用，我看了还是不懂，哈哈)。

研二开始后，差不多十月份开始自己整理找工作相关的知识，就有点类似于阿秀前期发布的两期 C++ 49 问和 59 问一样，然后开始投实习，前段时间上岸百度实习岗，来分享一波，以下是面经。

一面

- 1、介绍项目
- 2、针对项目：这个日志如何实现
- 3、deque 的实现原理
- 4、vector 与 deque 的优劣
- 5、vector 的扩容实现
- 6、讲讲 C++ 内存分区
- 7、讲讲移动构造函数
- 8、讲讲指针和引用
- 9、讲讲顶层 const、底层 const，引用更接近哪个
- 10、说下虚函数机制
- 11、深浅拷贝，深拷贝除了改写拷贝构造函数以外还要做什么操作（不太会，他提示下赋值操作符呢，我就说是要重载一下赋值操作符么？）
- 12、什么情况会调用拷贝构造，什么时候会用赋值操作
- 13、析构函数设为虚函数可不可以？
- 14、public、protected、private 继承分别可以访问什么（woc，这个东西没怎么用过，也没背到八股文，猜着说了一通）
- 15、简洁说说进程、线程，不要展开太多
- 16、进程 fork 的时候，虚拟地址空间的五大分区哪些不需要写时复制过去？（猜了下 data 段？他质疑了一下，我说 bss 也不会？因为 static 不能多次初始化，他说 static 可以多次赋值啊，然后跳过了....）
- 17、算法：链表相交（先装死说了个求全部长度的思路，然后断线了，他让我把代码发邮箱，我在写代码的时候再写了一个不用求长度的思路）

二面

- 1、问了一下你看过什么书

- 2、你看过 STL，讲讲内存配置器
- 3、讲讲迭代器（通过萃取迭代器的类型，面试官非要问具体怎么实现。我说我看的时候不太明白，他说没事，其实我第一次看也不太明白2333）
- 4、说说 vector，你自己想一个 vector 的设计方法
- 5、map 和 unordered_map 的区别
- 6、list 和vector 的区别
- 7、说一下 tcp 和 udp，tcp 如何保证有序，如何保证不丢失，如何保证不重复？
- 8、http 懂什么（说了一通，他说不用展开，我明白了）
- 9、说一下如何实现一个线程池
- 10、说一下如何实现一个线程安全的队列（我还想展开无锁队列，被叫停了）
- 11、malloc需要指定内存的大小，free为什么不需要？（我不会，他说你自己想一个实现方法，我说能不能在指针的上一个地址保存所要开辟的数组个数，然后free的时候先向上查找得到元素个数，然后再进行一个释放。他说行吧，这也是一种实现方法）
- 12、算法:LRU、旋转数组的查找

4.3、虚度大一一年又如何，双非本科大三学弟连斩腾讯字节

首发于个人公众号，原文链接：https://mp.weixin.qq.com/s/lSuN7Wo8AyC_FFWXJdU7fg

大家好，我是阿秀。

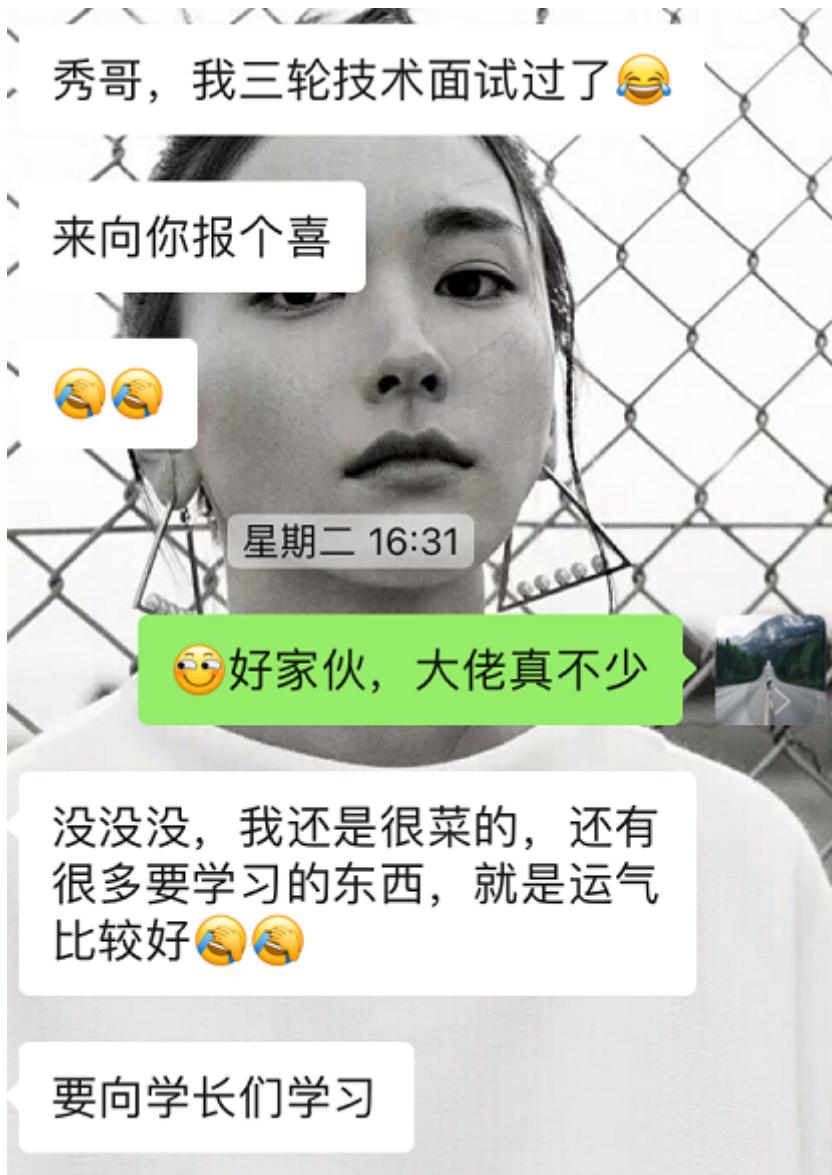
最近很开心鸭，因为不少粉丝朋友都来找我报喜，真心替他们感到高兴！

有通过自己的努力艰难爬坑社招去长沙多益做游戏开发的，也有校招实习去腾讯 WXG 部门的，还有去字节抖音的，真的太牛逼了。

不会有人不想去鹅厂吗？而且还是腾讯 WXG ...



最让我印象深刻的莫过于这位粉丝朋友了，他跟我报喜的时候我跟他聊了很多。



他跟我一样，是一位普通双非学校的学生，大一年还去搞硬件了，大二才开始学习准备软件开发的知识。

认真苦学一年半，终得正果！

恭喜这个逼！下面来看他的故事！

Offer情况

字节：深圳-安全与风控部门

腾讯：深圳- CSIG 腾讯云部门

目前状态：字节 offer 审批中，腾讯 HR 面已过

背景介绍

本人就读于某双非大学，现计科大三。大一时一直在学嵌入式做小车，从大二开始接触 Java，突然对 Java 很感兴趣，所以想学一下相关的技术，算起来，应该是疫情那个假期开始正式起步吧。

我在刚上大学的时候就下定决定毕业出去工作，没有打算考研。大一的时候听到实验室的学长年薪 15W 就傻了，当时已经刷新了我对大学生薪水的认知；大二时带我的助班去了华为工作，年薪居然有 30W 之多，这更加使我目瞪口呆。

那时，我便下定决心，我毕业之后也要年薪 15W，呜呜呜。

大二自学的时候仅仅是想着能多学一点以后工资可能就会高一点，从来没有想过进大厂

对于我这种普通学校的学生而言，毕业后能有一份不错的工作就很满意了。

这里也很感谢阿秀哥指导我HR面，让我提前知道 HR 面大概会考察哪些知识点，提前有了准备！

我也是大二开始有计划地开始刷leetcode，先刷一年简单题，是的整整一年，我刷的很慢，但每道题都是我自己亲手解出来的，并且也学了许多后端的技术框架。



大三上学期开始刷 middle 和 hard，开始看源码和一些经典书籍去深入了解知识，点亮自己的技能树。

也不知道从什么时候开始有了想去大厂的念头，也许是实验室的一次聚会上谈到每个人的规划时，学长对我说我准备了这么久可以去冲一下大厂。

这句话打动了我。

是呀，我做了这么多准备，小公司也用不上，何必不试一下大厂？试一下又不会掉块肉

然后开始疯狂地查缺补漏，去年寒假在家时只有除夕和大年初一没有学习，其它时间每天都在学习、做项目、查缺补漏，有时候做梦都能梦到自己在和面试官对话哈哈。

能通过这几轮面试，有很大的运气成分，因为我还有许多东西没有学，比如 Redis 的源码、Kafka 和 K8S 之类的。

Java 方面倒是准备挺多的，什么源码、多线程、微服务框架都有去准备，可惜字节和腾讯的面试都不怎么问我Java2333

腾讯面经

CSIG一面

- 自我介绍
- 部门主要是Python和Golang，进来转语言有没有问题？
- 上次一面为什么挂？说说上次面试没回答上来的问题
- 10000个数，小顶堆思路 找前100个和找前1000个的时间复杂度有什么区别？
- 系统中死循环如何定位？
- 如果是项目运行中呢？
- 进程、线程、协程？
- TCP三次握手、四次挥手？

- 介绍一下简历的项目
- 中途打断，为什么要用定时同步任务？
- 为什么想到用LRU设计商品推荐？
- LRU有什么缺点？
- 有了解过LFU吗？
- 点赞是如何设计的？
- 如果用户量很多，你会怎么设计点赞？
- 就比如说用别的方案，比如用rabbitMQ？（给了个不太好的方案）
- 这是最基础的方案，优化一下，如何减少对数据库的访问？（给了个好一点的方案）
- 考虑一下在MQ后面做处理？（突然领悟）
- redis了解吗？redis有哪些地方可能是你以后做项目会用到或者借鉴的？
- 有了解过设计模式吗？介绍一下工厂模式？
- 介绍一下秒杀项目？
- 讲了一下秒杀的逻辑，从前端到后端再到数据库
- 细问商品库存校验的实现（如何防止超卖）
- 为什么要加rabbitMQ？
- 为什么不用其它MQ？比如RocketMQ，Kafka？
- MySQL了解吗？说一下MySQL索引？
- 假如有一个非唯一索引，它是如何访问到数据的？
- 反问

CSIG二面

- 自我介绍
- 后端哪些方面学得比较好
- MySQL学生选课系统表的设计
- MySQL主键和唯一键的区别
- 有做过数据库性能调优吗
- 一条SQL语句执行慢是为什么
- 什么情况下会出现死锁？具体一些？
- 上一个问题我有提到间隙锁，面试官突然问我，你说什么锁？我：间隙锁....
- 间隙锁是什么锁？（我不知道面试官是真的不懂还是故意问的）
- 在编程过程中如何避免死锁（这回不能答八股文的死锁避免了）
- 场景题：你和另一个用户同时登录网站去修改一个数据，如何避免覆盖修改（或者是修改丢失）
- 哪个语言比较擅长
- 实现一个HashMap，口述思路
- hashCode和equals在什么时候需要重写
- 使用Java的时候用什么组件操作数据库
- MyBatis和其他的相比有什么优势
- MyBatis底层是如何管理Connection的
- 你觉得你的优势是什么
- 你的主动性如何
- 你的责任心如何
- 假如团队中有个别成员的进度没有达到预期，你会怎么处理

字节面经

字节一面

- 进程和线程的区别
- 进程被分配哪些资源
- 线程同步机制
- 线程共享哪些资源
- 线程独占哪些资源

- 程序计数器的作用
- 进程和线程的调度区别
- 进程常见的状态
- 阻塞态能直接到运行态吗
- 就绪态到运行态的条件（上一个进程时间片用完，本进程被系统调度）
- 解释一下 LRU
- 让你实现 LRU 你怎么实现
- LRU 各个操作的时间复杂度
- 计网七层模型、五层模型、四层模型
- 五层模型中各个层都有哪些协议，简单描述
- FTP 是什么协议
- TCP 和 UDP 的区别
- 三次握手、四次挥手
- 为什么是三次握手、四次挥手
- MySQL 的索引是什么
- 使用 B+ 树有什么优点
- B+ 树和 B 树相比较
- B+ 树的叶子结点存放数据有什么好处
- 设计题，先说思路再写代码：
- 已知一天内用户登录登出的日志（数据量较大），求这一天用户在线的最大峰值。
- 日志包含字段(userid, login_time, logout_time)，登录登出时间精确到秒。

字节二面

- 二面试官果然像传说中的那样没有一面面试官那么严肃~
- 自我介绍
- 项目中JWT的作用
- token和cookie的区别
- 进程和线程的区别与调度
- 假如有一段程序，只有main函数，也没有fork之类的操作，它跑起来的时候系统是几个进程几个线程？
- 进程之间如何通信
- 写两个代码：1、剑指offer30 的变形题 2、剑指offer13 两题很快秒了，继续问问题
- 项目中的定时任务具体实现逻辑以及功能
- 当场优化项目中的某个功能，允许去查阅资料（之前说过这个功能设计得不太好）
- Redis的几种数据结构
- Redis的容灾方案（持久化+集群）
- 定时持久化数据会有什么问题？（其实是想问RDB的缺点）
- AOF模式下，机器宕机之后如何恢复数据？
- 关系型数据库的事务要保证什么（四大特性）
- 细说事务隔离级别
- 口述思路：删除单链表倒数第k个节点（还是剑指offer！）
- 秒给思路，随后追问：思路有什么问题？
- 如果链表有环？
- 僵尸进程以及带来的问题？
- 僵尸进程中子进程未释放的具体是什么资源？
- 近期的学习规划、打算
- 反问

字节三面

没有自我介绍，上来直接开始

- 现在读大三是吗？

- 以前去实习过吗？
- 你的项目里用到了 RateLimiter 限流，你用代码实现一下？（我的内心：What.....）我说：我只是简单地用过，但是没有去研究过它的底层原理
- 说一下 RateLimiter 的工作原理？
- RateLimiter 写不了是吧，那写一个代码，输入的是一个代码段，用字符串表示，输入的代码里面会有一些注释，输出去掉所有注释之后的代码
- 写了50分钟，面试官给的最后一个输入没跑通，然后说时间关系，今天就写到这儿了，让我下去可以再看看代码
- 写代码是因为感兴趣吗？
- 你觉得你和其他同学相比写代码的能力怎么样？
- 怎么证明？（证明上一个问题的回答）
- 以前打过ACM是吗？（只参加过一次而已）
- 反问

字节HR面

- 自我介绍
- 高考之后为什么选择现在的这个学校？
- 第一志愿学校是什么
- 深圳这边夏天也很热，能否适应
- 对计算机专业的理解
- 对计算机感兴趣吗
- 怎么想到做简历上的这个项目
- 项目团队有几个人
- 团队有没有组长或负责人
- 从项目中收获了什么
- 遇到了问题怎么解决
- 有没有遇到过解决不了的问题，后来怎么办
- 开发时和同学有意见冲突怎么办
- 有没有投递别的互联网公司
- 为什么想来字节跳动
- 对字节跳动的了解，评价一下
- 来字节实习想收获什么
- 平时怎么学习
- 毕业前/后的规划
- 为什么不考研
- 评价一下自己，优缺点
- 什么时候能来，能实习多久
- 实习期间要不要兼顾学校课程

结语

有人可能会担心普通本科进不了大厂，没有 211 以上的学历加成之类的。

我想说的是，可能你没有别人学校好，但你就不去尝试了吗？这条路是很难，可再难，也有人上岸了啊。

NBA已故球星科比的名言就很好：总有人要赢得，为什么不能是我呢？

对啊，总有人要上岸的，为什么不能是你呢？

最后，**再次恭喜这个哥们！**

5、内推信息

为什么要尽可能找内推的原因已经在 2.1.4 小节中讲过了，这里再简单重述一下，其实主要有以下四个原因：

- 1、内推简历处理速度更快，不管你是简历合格被发起面试或者笔试，还是不合格被 pass 的速度都要快一些，所以能节约等待时间。
- 2、能够及时找内推人了解自己的求职进度。比如一面结束后想尽快知道面试结果，这个时候可以拜托内推人去帮忙打听一下面试结果，避免一直等下去。
- 3、内推人也可以帮你解锁简历，这里说明一下有些互联网公司是有所谓的人才池的，一个公司的不同部门都可以对人才池里的求职者发起面试，这时候就有一种情况出现：假设A部门对你发起面试，但不幸你没能通过A部门的面试，但A部分的面试官也忘记把你的简历放回人才池（俗称解锁），这就导致该公司的其他部门，比如B部门、C部门无法看到在人才池中看到你的简历信息，也就无法对你发起面试，无形中你就比别人少了一些面试机会了，也就少了一丝上岸的可能性。
- 4、互联网人人内推的时代，你不内推不就亏了，大家都内推，你不去内推...这就好像幼儿园考试，每个孩子都有一朵大红花，而你没有，嗯，你亏了。

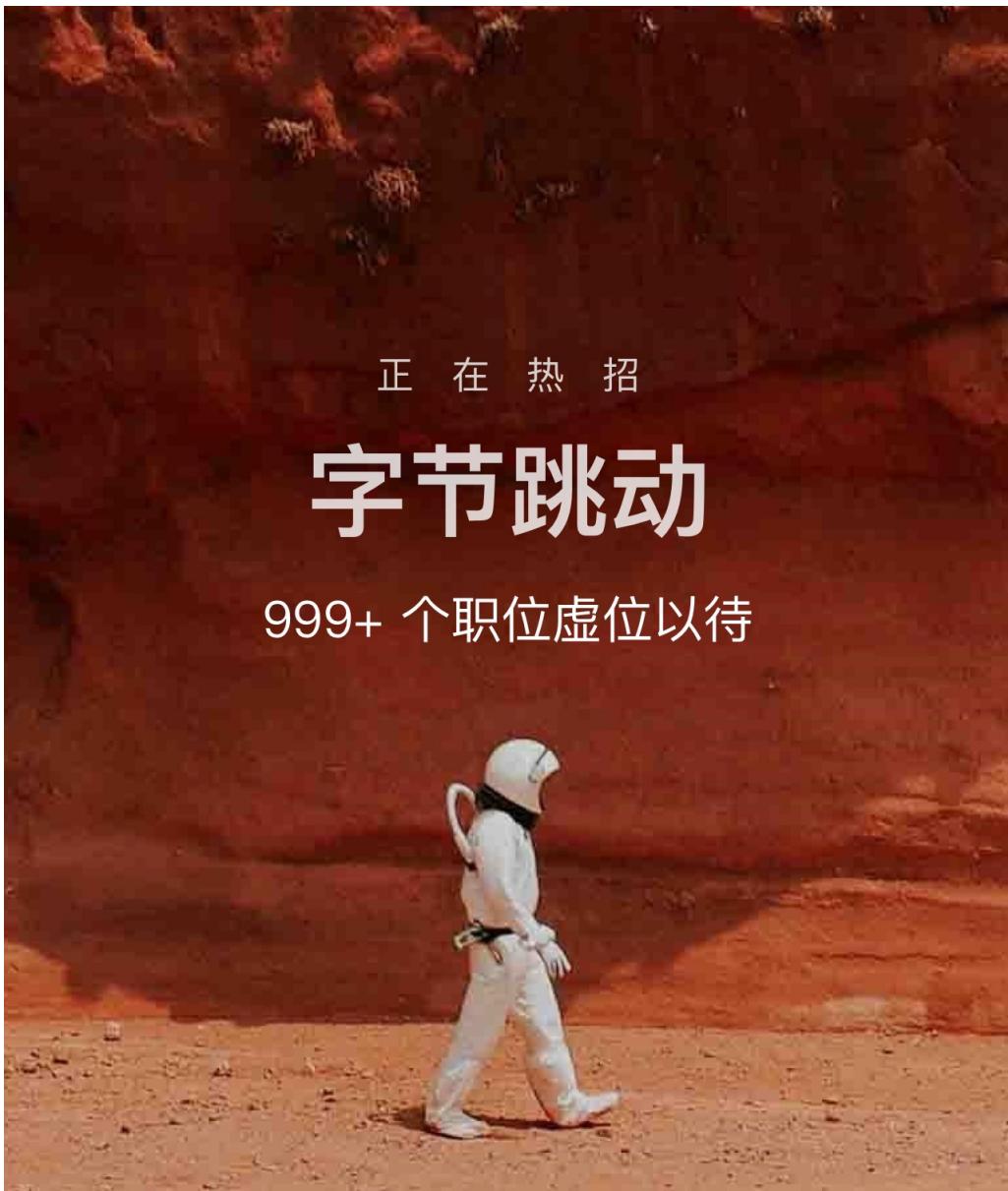
5.1、字节跳动

字节跳动大量招募新同学！本次招聘包含 **暑期实习** 和 **全职补录** 两类岗位，职位类别涵盖研发、产品、运营、设计、市场、销售、职能/支持、教研教学、游戏策划等多个方向。

大量招聘岗位，欢迎投递~

内推链接：https://job.toutiao.com/campus/m/position?external_referral_code=AMKWET6

扫码投递：



正 在 热 招

字节跳动

999+ 个职位虚位以待



长按识别二维码查看职位列表

投递进度自主查询链接: <https://job.bytedance.com/society/position/application> (电脑打开)

微信咨询: aXiu_go (加入请备注“**字节内推**”, 否则不予通过, 见谅!), 个人微信二维码:



5.2、携程

阿秀的一位好友在携程工作，大家如果有意投递携程，可以使用他的内推码~
携程招聘开始了，有想去携程工作的小伙伴欢迎使用内推码投递简历哦~
内推可以优先筛选简历，可以更快速被HR看到！个人携程内推码【 NTJi8576】，也欢迎大家联系我跟进面试进度！
携程双休不加班，每天八小时工作制，性价比极高！
投递官网地址：<http://campus.ctrip.com/#/>
微信咨询：Coder_XZ(加人请备注“携程内推”，否则不予通过，见谅！)

5.3、其余公司

其余公司内推信息正在整理ing