C++面试题目汇总

C++面试题目汇总

目录

1:进程与线程的区别？

2：进程间的通信方式？

3：线程间的通信方式？

一、为什么引入进程？

进程是为了提高 CPU 的执行效率，减少因为程序等待带来的 CPU 空转以及其他计算机软硬件资源的浪费而提出来的。

二、为什么引入线程？

为了减少进程切换和创建的开销，提高执行效率和节省资源 。

三、线程和进程的区别？

调度：线程是独立调度的基本单位，进程是拥有资源的基本单位。在同一进程中，线程的切换不会引起进程的切换；在不同的进程中，进行线程切换，则会引起进程的切换。

拥有资源：进程是拥有资源的基本单位，线程不拥有资源，但线程可以共享器隶属进程的系统资源。

并发性：进程可以并发执行，而且同一进程内的多个线程也可以并发执行，大大提高了系统的吞吐量。

系统开销：创建和撤销进程时，系统都要为之分配或回收资源，在进程切换时，涉及当前执行进程 CPU 环境的保存以及新调度的进程 CPU 环境的设置；而线程切换时只需保存和设置少量寄存器内容，因此开销很小，另外，由于同一进程内的多个线程共享进程的地址空间，因此这些线程之间的同步与通信比较容易实现，甚至无须操作系统的干预。

通信方面：进程间通信需要借助操作系统，而线程间可以直接读/写进程数据段来进行通信。

四、进程间通信方式

管道(pipe)

有名管道 (named pipe)

信号量(semophore)

消息队列(message queue)

信号 (signal)

套接字(socket)

五、线程间通信方式

事件(Event);

信号量(semaphore);

互斥量(mutex);

临界区(Critical section)

六、什么时候用进程？什么时候用线程？

进程与线程的选择取决以下几点：

需要频繁创建销毁的优先使用线程；因为对进程来说创建和销毁一个进程代价是很大的；

线程的切换速度快，所以在需要大量计算，切换频繁时用线程，还有耗时的操作使用线程可提高应用程序的响应；

因为对 CPU 系统的效率使用上线程更占优，所以可能要发展到多机分布的用进程，多核分布用线程；

并行操作时使用线程，如 C/S 架构的服务器端并发线程响应用户的请求；

需要更稳定安全时，适合选择进程；需要速度时，选择线程更好；

I/O 密集型和 CPU 密集型适合多线程。

4：栈和堆的区别？

1、在申请方式上

栈（stack）：现在很多人都称之为堆栈，这个时候实际上还是指的栈。它由编译器自动管理，无需我们手工控制。 例如，声明函数中的一个局部变量 int b 系统自动在栈中为 b 开辟空间；在调用一个函数时，系统自动的给函数的形参变量在栈中开辟空间。

堆（heap）：申请和释放由程序员控制，并指明大小。容易产生 memory leak。

在 C 中使用 malloc 函数。

如：char *p1 = (char *)malloc(10);

在 C++中用 new 运算符。

如：char *p2 = new char(20);

但是注意 p1 本身在全局区，而 p2 本身是在栈中的，只是它们指向的空间是在堆中。

2、申请后系统的响应上

栈（stack）:只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆（heap）: 首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后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。另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 delete 或 free 语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

3、申请大小的限制

栈（stack）:在 Windows 下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（也有的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。例如，在 VC6 下面，默认的栈空间大小是 1M（好像是，记不清楚了）。当然，我们可以修改：打开工程，依次操作菜单如下：Project->Setting->Link，在 Category 中选中 Output，然后在 Reserve 中设定堆栈的最大值和 commit。

注意：reserve 最小值为 4Byte；commit 是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

堆（heap）: 堆是向高地址扩展的数据结构，是不连续的内存区域（空闲部分用链表串联起来）。正是由于系统是用链表来存储空闲内存，自然是不连续的，而链表的遍历方向是由低地址向高地址。一般来讲在 32 位系统下，堆内存可以达到 4G 的空间，从这个角度来看堆内存几乎是没有什么限制的。由此可见，堆获得的空间比较灵活，也比较大。

4、分配空间的效率上

栈（stack）:栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。但程序员无法对其进行控制。

堆（heap）:是 C/C++函数库提供的，由 new 或 malloc 分配的内存，一般速度比较慢，而且容易产生内存碎片。它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。这样可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。显然，堆的效率比栈要低得多。

5、堆和栈中的存储内容

栈（stack）:在函数调用时，第一个进栈的是主函数中子函数调用后的下一条指令（子函数调用语句的下一条可执行语句）的地址，然后是子函数的各个形参。在大多数的 C 编译器中，参数是由右往左入栈的，然后是子函数中的局部变量。

注意：静态变量是不入栈的。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中子函数调用完成的下一条指令，程序由该点继续运行。

堆（heap）:一般是在堆的头部用一个字节存放堆的大小，堆中的具体内容有程序员安排。

6、存取效率的比较

这个应该是显而易见的。拿栈上的数组和堆上的数组来说：

```
void main()
{
    int arr[5] = { 1,2,3,4,5 };
    int *arr1 = NULL;
    arr1 = new int[5];
    for (int j = 0;j <= 4;j++)
    {
        arr1[j] = j + 6;
    }
    int a = arr[1];
    int b = arr1[1];
}
```

上面代码中，arr1（局部变量）是在栈中，但是指向的空间确在堆上，两者的存取效率，当然是 arr

高。因为 arr[1]可以直接访问，但是访问 arr1[1]，首先要访问数组的起始地址 arr1，然后才能访问到 arr1[1]。

总而言之，言而总之：

堆和栈的区别可以用如下的比喻来看出： 使用栈就象我们去饭馆里吃饭，只管点菜（声明变量）、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。

使用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。

5：C++和 C 的区别？

C 是一个结构化语言，它的重点在于算法和数据结构。对于语言本身而言，C 是 C++的子集。C 程序的设计首要考虑的是如何通过一个过程，对输入进行运算处理，得到输出。对于 C++，首要考虑的是如何构造一个对象模型，让这个模型能够配合对应的问题，这样就可以通过获取对象的状态信息得到输出或实现过程控制。

因此，C 与 C++的最大区别在于，它们用于解决问题的思想方法不一样。

C 实现了 C++中过程化控制及其他相关功能。而在 C++中的 C，相对于原来的 C 还有所加强，引入了重载、内联函数、异常处理等。C++更是拓展了面向对象设计的内容，如类、继承、虚函数、模板和包容器类等。

在 C++中，不仅需要考虑数据封装，还需要考虑对象的粒度的选择、对象接口的设计和继承、组合与继承的使用等问题。

相对于 C，C++包含了更丰富的设计概念。

6：红黑树和 B 树的区别？

背景:这几天在看《高性能 Mysql》,在看到创建高性能的索引,书上说 mysql 的存储引擎 InnoDB 采用的索引类型是B+Tree,那么,大家有没有产生这样一个疑问,对于数据索引,为什么要使用B+Tree这种数据结构,和其它树相比,它能体现的优点在哪里? 看完这篇文章你就会了解到这些数据结构的原理以及它们各自的应用场景.

二叉查找树(BST)

简介

二叉查找树也称为有序二叉查找树,满足二叉查找树的一般性质,是指一棵空树具有如下性质:

任意节点左子树不为空,则左子树的值均小于根节点的值.

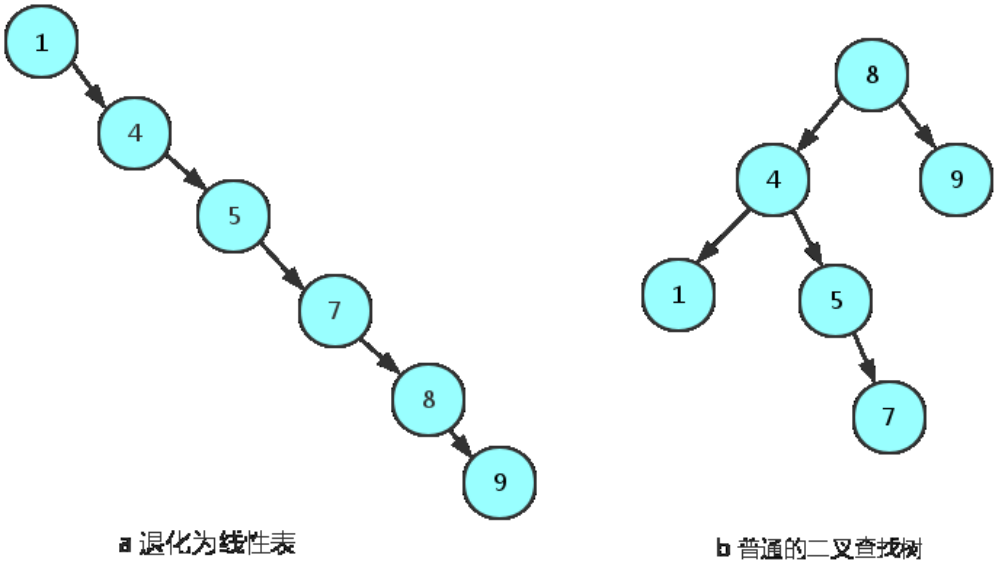
任意节点右子树不为空,则右子树的值均大于于根节点的值.

任意节点的左右子树也分别是二叉查找树.

没有键值相等的节点.

局限性及应用

一个二叉查找树是由 n 个节点**随机构成**,所以，对于某些情况,二叉查找树会退化成一个有 n 个节点的线性链.如下图:



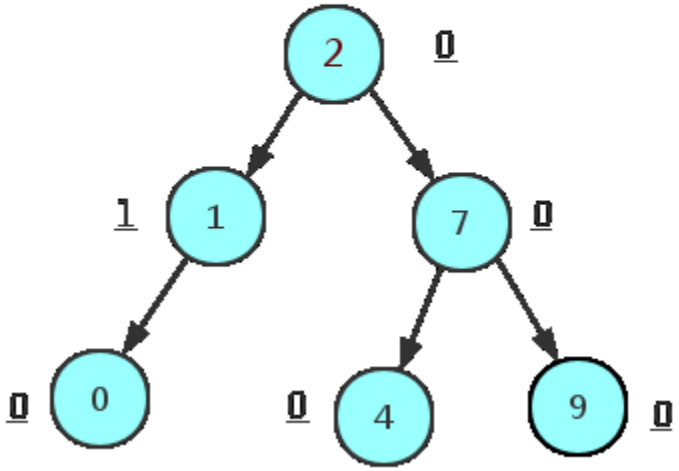
a 退化为线性表　　b 普通的二叉查找树

b 图为一个普通的二叉查找树,大家看 a 图,如果我们的根节点选择是最小或者最大的数,那么二叉查找树就完全退化成了线性结构,因此,在二叉查找树的基础上,又出现了 AVL 树,红黑树,它们两个都是基于二叉查找树,只是在二叉查找树的基础上又对其做了限制.

AVL 树

简介

AVL 树是带有平衡条件的二叉查找树,一般是用平衡因子差值判断是否平衡并通过旋转来实现平衡,左右子树树高不超过 1,和红黑树相比,它是严格的平衡二叉树,平衡条件必须满足(所有节点的左右子树高度差不超过 1).不管我们是执行插入还是删除操作,只要不满足上面的条件,就要通过旋转来保持平衡,而旋转是非常耗时的,由此我们可以知道 AVL 树适合用于插入删除次数比较少，但查找多的情况。



从上面这张图我们可以看出,任意节点的左右子树的平衡因子差值都不会大于 1.

局限性

由于维护这种高度平衡所付出的代价比从中获得的效率收益还大,故而实际的应用不多，更多的地方是用追求局部而不是非常严格整体平衡的红黑树.当然,如果应用场景中对插入删除不频繁,只是对查找要求较高,那么 AVL 还是较优于红黑树.

应用

Windows NT 内核中广泛存在.

红黑树

简介

一种二叉查找树,但在每个节点增加一个存储位表示节点的颜色,可以是 red 或 black. 通过对任何一条从根到叶子的路径上各个节点着色的方式的限制,红黑树确保没有一条路径会比其它路径长出两倍.它是一种弱平衡二叉树(**由于是若平衡,可以推出,相同的节点情况下,AVL 树的高度低于红黑树**),相对于要求严格的 AVL 树来说,它的旋转次数变少,所以对于搜索,插入,删除操作多的情况下,我们就用红黑树.

性质

每个节点非红即黑.

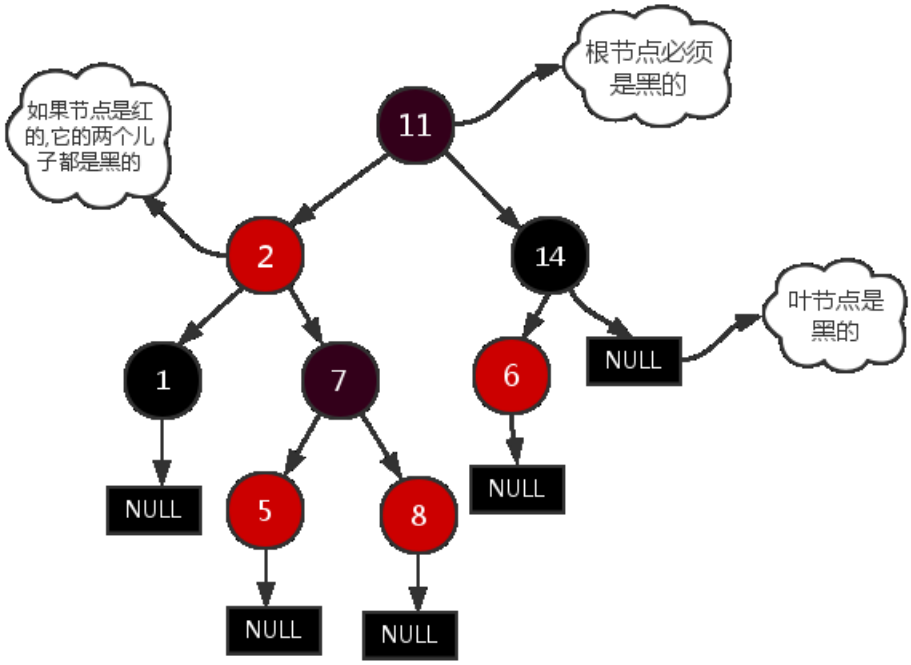
根节点是黑的。

每个叶节点(**叶节点即树尾端 NUL 指针或 NULL 节点**)都是黑的.

如果一个节点是红的,那么它的两儿子都是黑的.

对于任意节点而言,其到叶子点树 NIL 指针的每条路径都包含相同数目的黑节点.

查找、插入和删除的时间复杂度都是 O(logn)



每条路径都包含相同的黑节点.

应用

广泛用于 C++的 STL 中,map 和 set 都是用红黑树实现的.

著名的 linux 进程调度 Completely Fair Scheduler,用红黑树管理进程控制块,进程的虚拟内存区域都存储在一颗红黑树上,每个虚拟地址区域都对应红黑树的一个节点,左指针指向相邻的地址虚拟存储区域,右指针指向相邻的高地址虚拟地址空间.

IO 多路复用 epoll 的实现采用红黑树组织管理 sockfd，以支持快速的增删改查.

ngnix 中,用红黑树管理 timer,因为红黑树是有序的,可以很快的得到距离当前最小的定时器.

java 中 TreeMap 的实现.

B/B+树

注意 B-树就是 B 树,-只是一个符号.

简介

B/B+树是为了磁盘或其它存储设备而设计的一种平衡多路查找树(相对于二叉,B 树每个内节点有多个分支),与红黑树相比,在相同的的节点的情况下,一颗 B/B+树的高度远远小于红黑树的高度(**在下面 B/B+树的性能分析中会提到**).B/B+树上操作的时间通常由存取磁盘的时间和 CPU 计算时间这两部分构成,而 CPU 的速度非常快,所以 B 树的操作效率取决于访问磁盘的次数,关键字总数相同的情况下 B 树的高度越小，磁盘 I/O 所花的时间越少.

B 树中所有结点的孩子结点数的最大值称为 B 树的阶，通常用 m 表示。

一棵 m 叉树的性质如下：

树中每个结点至多有 m 棵子树（即至多含有 m-1 个关键字）

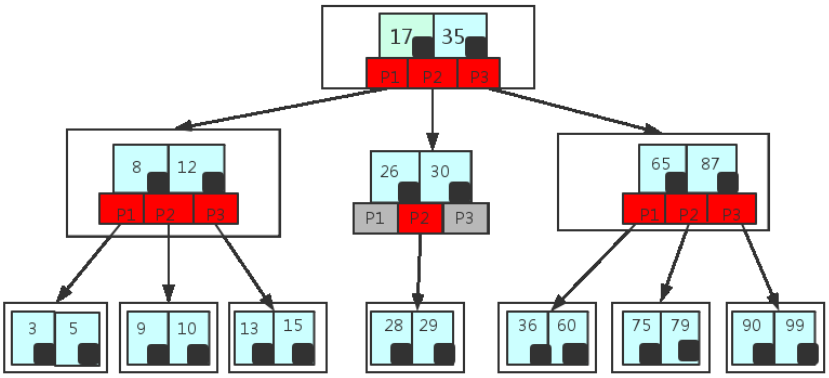
若根结点不是终端结点，则至少有两棵子树

除根结点以外的所有非叶子结点至少有[m/2]（向上取整）棵子树（即至少含有[m/2]-1 个关键字）

所有非叶子结点的关键字：K[1], K[2], …, K[m-1]；且 K[i] < K[i+1]；

非叶子结点的指针：P[1], P[2], …, P[m]；其中 P[1]指向关键字小于 K[1]的子树，P[m]指向关键字大于 K[m-1]的子树，其它 P[i]指向关键字属于(K[i-1], K[i])的子树；

所有叶子结点位于同一层；



B树

这里只是一个简单的 B 树,在实际中 B 树节点中关键字很多的.上面的图中比如 35 节点,35 代表一个 key(索引)，而小黑块代表的是这个 key 所指向的内容在内存中实际的存储位置.**是一个指针.**

B+树

B+树是应文件系统所需而产生的一种 B 树的变形树(**文件的目录一级一级索引,只有最底层的叶子节点(文件)保存数据**.),非叶子节点只保存**索引**,不保存实际的数据,数据都保存在叶子节点中.这不就是文件系统文件的查找吗?我们就举个文件查找的例子:有 3 个文件夹,a,b,c， a 包含 b,b 包含 c,一个文件 yang.c,a,b,c 就是索引(存储在非叶子节点)，a,b,c 只是要找到的 yang.c 的 key,而实际的数据 yang.c 存储在叶子节点上.

所有的非叶子节点都可以看成索引部分

一棵 m 阶的 B+树的性质(下面提到的都是和 B 树不相同的性质)

每个分支结点最多有 m 棵子树

非叶根结点至少有两棵子树，其他每个分支结点至少有[m/2]（向上取整）棵子树

结点的子树个数与关键字个数相等

所有叶结点包含全部关键字及指向相应记录的指针，而且叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来

所有分支结点中仅包含它的各个子结点中关键字的最大值及指向子结点的指针

m 阶的 B+树与 m 阶的 B 树的主要差异在于：

在 B+树中，具有 n 个关键字的结点只含有 n 棵子树，即每个关键字对应一棵子树；而在 B 树中，具有 n 个关键字的结点含有（n+1）棵子树

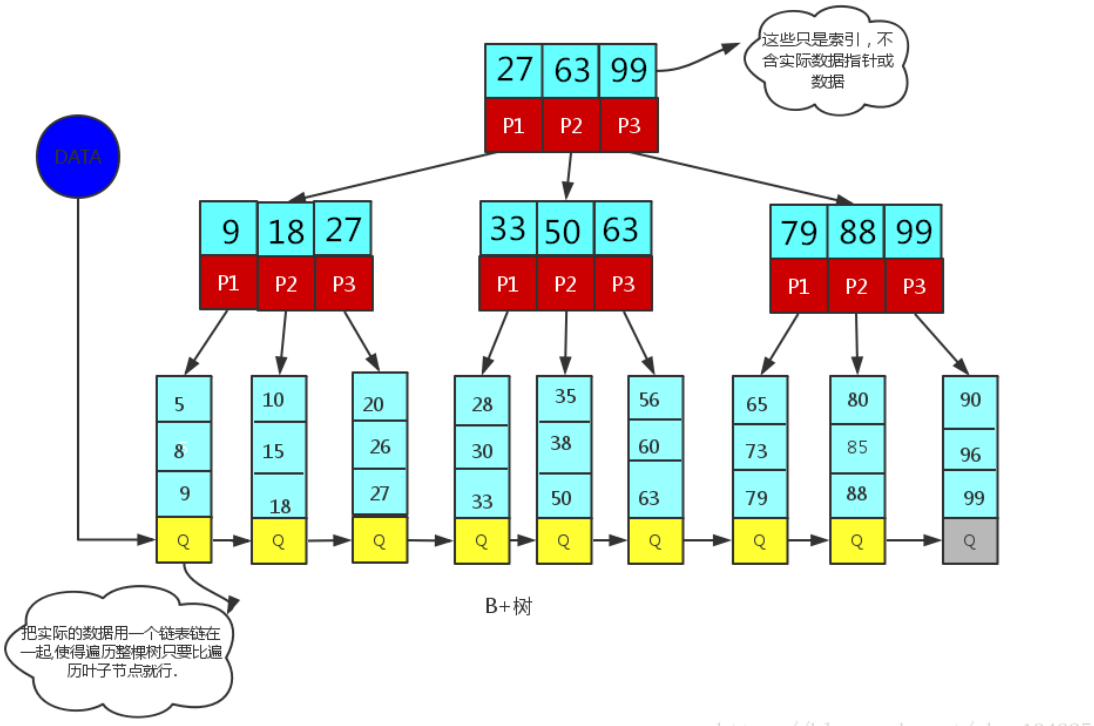
在 B+树中，每个结点（非根结点）关键字个数 n 的范围是[m/2] <=n<=m（根结点：1<=n<=m)，在 B 树

中，每个结点（非根结点）关键字个数 n 的范围是[m/2]-1<=n<=m-1（根结点：1<=n<=m-1）（都是向上取整）

在 B+中，叶结点包含信息，所有非叶结点仅起到索引作用，非叶结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址

在 B+中，叶结点包含了全部关键字，即在非叶结点中出现的关键字也会出现在叶结点中；而在 B 树中，叶结点包含的关键字和其他结点包含的关键字是不重复的

看下图:



非叶子节点(比如 5,28,65)只是一个 key(索引),实际的数据存在叶子节点上(5,8,9)才是真正的数据或指向真实数据的指针.

应用

B 和 B+树主要用在文件系统以及数据库做索引.比如 Mysql;

B/B+树性能分析

n 个节点的平衡二叉树的高度为 H(即 logn),而 n 个节点的 B/B+树的高度为 logt((n+1)/2)+1;

若要作为内存中的查找表,B 树却不一定比平衡二叉树好,尤其当 m 较大时更是如此.因为查找操作 CPU 的时间在 B-树上是 O(mlogtn)=O(lgn(m/lgt)),而 m/lgt>1;所以 m 较大时 O(mlogtn)比平衡二叉树的操作时间大得多. 因此在内存中使用 B 树必须取较小的 m.（通常取最小值 m=3，此时 B-树中每个内部结点可以有 2 或 3 个孩子，这种 3 阶的 B-树称为 2-3 树）。

为什么说 B+tree 比 B 树更适合实际应用中操作系统的文件索引和数据索引.

B+-tree 的内部节点并没有指向关键字具体信息的指针,因此其内部节点相对 B 树更小,如果把所有同一内部节点的关键字存放在同一盘块中,那么盘块所能容纳的关键字数量也越多,一次性读入内存的需要查找的关键字也就越多,相对 IO 读写次数就降低了.

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

ps:我在知乎上看到有人是这样说的,我感觉说的也挺有道理的:

他们认为数据库索引采用 B+树的主要原因是:B 树在提高了 IO 性能的同时并没有解决元素遍历的我效率低下的问题,正是为了解决这个问题,B+树应用而生.B+树只需要去遍历叶子节点就可以实现整棵树的遍历.而且在数据库中基于范围的查询是非常频繁的，而 B 树不支持这样的操作（或者说效率太低）.

7：产生死锁的必要条件？已经如何预防死锁？

一、计算机系统中的死锁
竞争不可抢占性资源引起死锁
竞争可消耗资源引起死锁
进程推进顺序不当引起死锁
二、产生死锁的必要条件
互斥条件（资源独占）
请求和保持条件
不可抢占条件（不可剥夺）
循环等待条件
三、处理死锁的方法
预防死锁
避免死锁
检测死锁
解除死锁
四、预防死锁
破坏‘请求和保持’条件
破坏‘不可抢占条件’条件
破坏‘循环等待’条件（主要是破坏产生死锁的后三个条件）
五、解决死锁
最简单的办法是终止各锁住进程，或按一定的顺序中止进程序列，直到已释放到有足够的资源来完成剩下的进程时为止。
也可以从被锁住进程强迫剥夺资源以解除死锁

8：TCP 和 UDP 的区别？

TCP 和 UDP 的区别：
传输控制协议 TCP 的特点：
(1) TCP 是面向连接的运输层协议
(2) 每一条 TCP 连接只能有两个端口，即：点对点
(3) TCP 提供可靠交付的服务
(4) TCP 提供全双工通信
(5) 面向字节流，TCP 中的“流”指流入到进程或从进程流出的字节序列
用户数据报协议 UDP 的特点：
(1) UDP 是无连接的
(2) UDP 使用尽最大努力交付
(3) UDP 是面向报文
(4) UDP 没有拥塞控制
(5) UDP 支持一对一、一对多、多对一和多对的的交互通信
(6) 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

9：TCP 状态中 time_wait 的作用？

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间。这么做有两个理由：
确保最后一个确认报文段能够到达。如果 B 没收到 A 发送来的确认报文段，那么就会重新发送连接释放请求报文段，A 等待一段时间就是为了处理这种情况的发生。
可能存在“已失效的连接请求报文段”，为了防止这种报文段出现在本次连接之外，需要等待一段时间。

10：HTTP 2.0 与 HTTP 1.0 的区别 ？

1、什么是 HTTP 2.0
HTTP/2（超文本传输协议第 2 版，最初命名为 HTTP 2.0），是 HTTP 协议的的第二个主要版本，使用于万维网。HTTP/2 是 HTTP 协议自 1999 年 HTTP 1.1 发布后的首个更新，主要基于 SPDY 协议（是 Google 开发的基于 TCP 的应用层协议，用以最小化网络延迟，提升网络速度，

优化用户的网络使用体验)。

2、与 HTTP 1.1 相比，主要区别包括

HTTP/2 采用二进制格式而非文本格式

HTTP/2 是完全多路复用的，而非有序并阻塞的——只需一个连接即可实现并行

使用报头压缩，HTTP/2 降低了开销

HTTP/2 让服务器可以将响应主动“推送”到客户端缓存中

3、HTTP/2 为什么是二进制？

比起像 HTTP/1.x 这样的文本协议，二进制协议解析起来更高效、“线上”更紧凑，更重要的是错误更少。

4、为什么 HTTP/2 需要多路传输?

HTTP/1.x 有个问题叫线端阻塞(head-of-line blocking)，它是指一个连接(connection)一次只提交一个请求的效率比较高，多了就会变慢。 HTTP/1.1 试过用流水线(pipelining)来解决这个问题，但是效果并不理想(数据量较大或者速度较慢的响应，会阻碍排在他后面的请求)．此外，由于网络媒介(intermediary)和服务器不能很好的支持流水线，导致部署起来困难重重。而多路传输(Multiplexing)能很好的解决这些问题，因为它能同时处理多个消息的请求和响应；甚至可以在传输过程中将一个消息跟另外一个掺杂在一起。所以客户端只需要一个连接就能加载一个页面。

5、消息头为什么需要压缩?

假定一个页面有 80 个资源需要加载（这个数量对于今天的 Web 而言还是挺保守的），而每一次请求都有 1400 字节的消息头（着同样也并不少见，因为 Cookie 和引用等东西的存在），至少要 7 到 8 个来回去“在线”获得这些消息头。这还不包括响应时间——那只是从客户端那里获取到它们所花的时间而已。这全都由于 TCP 的慢启动机制，它会基于对已知有多少个包，来确定还要来回去获取哪些包 - 这很明显的限制了最初的几个来回可以发送的数据包的数量。相比之下，即使是头部轻微的压缩也可以是让那些请求只需一个来回就能搞定——有时候甚至一个包就可以了。这种开销是可以被节省下来的，特别是当你考虑移动客户端应用的时候，即使是良好条件下，一般也会看到几百毫秒的来回延迟。

6、服务器推送的好处是什么？

当浏览器请求一个网页时，服务器将会发回 HTML，在服务器开始发送 JavaScript、图片和 CSS 前，服务器需要等待浏览器解析 HTML 和发送所有内嵌资源的请求。服务器推送服务通过“推送”那些它认为客户端将会需要的内容到客户端的缓存中，以此来避免往返的延迟。

11：HTTP 与 HTTPS 的区别？

超文本传输协议 HTTP 协议被用于在 Web 浏览器和网站服务器之间传递信息，HTTP 协议以明文方式发送内容，不提供任何方式的数据加密，如果攻击者截取了 Web 浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息，因此，HTTP 协议不适合传输一些敏感信息，比如：信用卡号、密码等支付信息。

为了解决 HTTP 协议的这一缺陷，需要使用另一种协议：安全套接字层超文本传输协议 HTTPS，为了数据传输的安全，HTTPS 在 HTTP 的基础上加入了 SSL 协议，SSL 依靠证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。

一、HTTP 和 HTTPS 的基本概念

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从 WWW 服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版，即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。

HTTPS 协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

二、HTTP 与 HTTPS 有什么区别？

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，为了保证这些隐私数据能加密传输，于是网景公司设计了 SSL（Secure Sockets Layer）协议用于对 HTTP 协议传输的数据进行加密，从而就诞生了 HTTPS。简单来说，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。

HTTPS 和 HTTP 的区别主要如下：

1、https 协议需要到 ca(证书授权机构:Certificate Authority)申请证书，一般免费证书较少，因而需要一定费用。

2、http 是超文本传输协议，信息明文传输，https 则是具有安全性的 ssl 加密传输协议。

3、http 和 https 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。

4、http 的连接很简单，是无状态的；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，比 http 协议安全。

三、HTTPS 的工作原理

我们都知道 HTTPS 能够加密信息，以免敏感信息被第三方获取，所以很多银行网站或电子邮箱等等安全级别较高的服务都会采用 HTTPS 协议。

客户端在使用 HTTPS 方式与 Web 服务器通信时有以下几个步骤，如图所示。

（1）客户使用 https 的 URL 访问 Web 服务器，要求与 Web 服务器建立 SSL 连接。

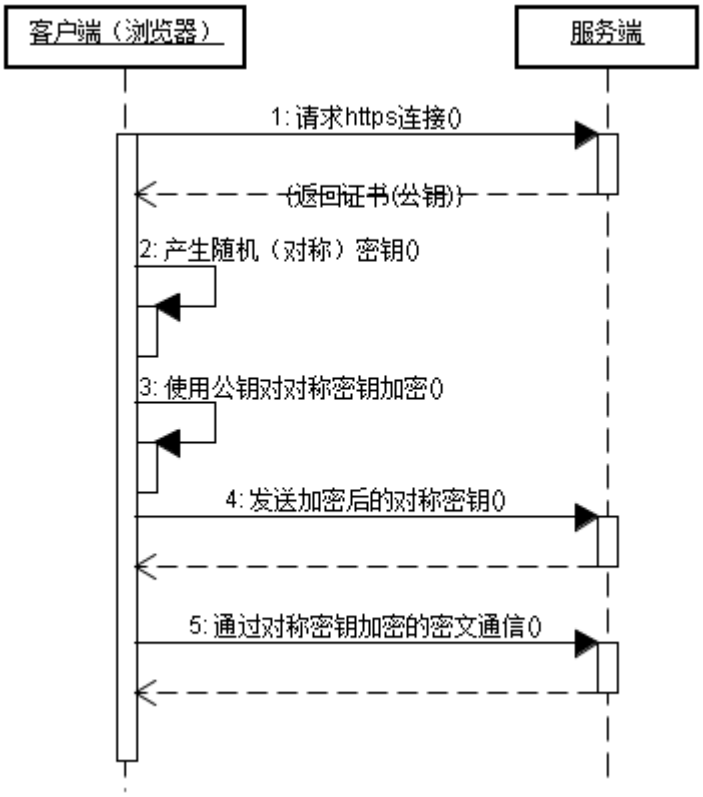
（2）Web 服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

（3）客户端的浏览器与 Web 服务器开始协商 SSL 连接的安全等级，也就是信息加密的等级。

（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

（5）Web 服务器利用自己的私钥解密出会话密钥。

（6）Web 服务器利用会话密钥加密与客户端之间的通信。



四、HTTPS 的优点

尽管 HTTPS 并非绝对安全，掌握根证书的机构、掌握加密算法的组织同样可以进行中间人形式的攻击，但 HTTPS 仍是现行架构下最安全的解决方案，主要有以下几个好处：

（1）使用 HTTPS 协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；

（2）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，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。

（3）HTTPS 是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

（4）谷歌曾在 2014 年 8 月份调整搜索引擎算法，并称“比起同等 HTTP 网站，采用 HTTPS 加密的网站在搜索结果中的排名将会更高”。

五、HTTPS 的缺点

虽然说 HTTPS 有很大的优势，但其相对来说，还是存在不足之处的：

（1）HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10%到 20%的耗电；

（2）HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；

（3）SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。

（4）SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗。

（5）HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

六、http 切换到 HTTPS

如果需要将网站从 http 切换到 https 到底该如何实现呢？

这里需要将页面中所有的链接，例如 js，css，图片等等链接都由 http 改为 https。例如：http://www.baidu.com 改为 https://www.baidu.com

BTW,这里虽然将 http 切换为了 https,还是建议保留 http。所以我们在切换的时候可以做 http 和 https 的兼容，具体实现方式是，去掉页面链接中的 http 头部，这样可以自动匹配 http 头和 https 头。例如：将 http://www.baidu.com 改为//www.baidu.com。然后当用户从 http 的入口进入访问页面时,页面就是 http,如果用户是从 https 的入口进入访问页面，页面即使 https 的。

12：TCP 的三次握手和四次挥手的过程？

TCP 建立连接的过程叫做握手，握手需要在客户和服务器之间交换三个 TCP 报文段。图 5-28 画出了三报文握手[①]建立 TCP 连接的过程。

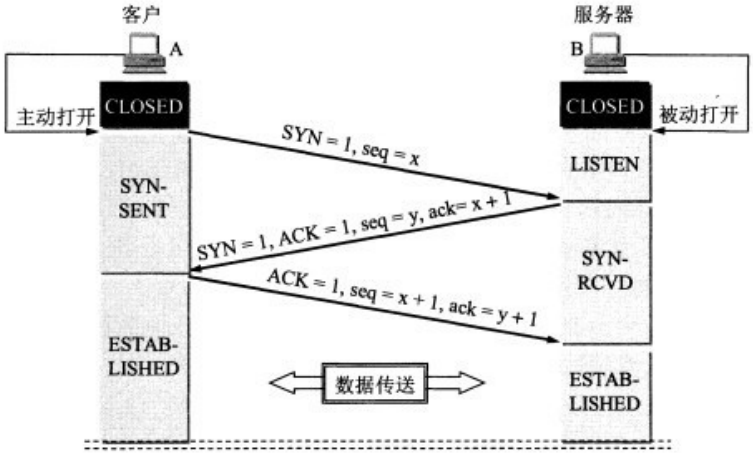


图 5-28 用三报文握手建立 TCP 连接

假定主机 A 运行的是 TCP 客户程序，而 B 运行 TCP 服务器程序。最初两端的 TCP 进程都处于 CLOSED（关闭）状态。图中在主机下面的方框分别是 TCP 进程所处的状态。请注意，在本例中，A **主动打开连接**，而 B **被动打开连接**。

一开始，B 的 TCP 服务器进程先创建**传输控制块** TCB[①]，准备接受客户进程的连接请求。然后服务器进程就处于 LISTEN（收听）状态，等待客户的连接请求。如有，即作出响应。

A 的 TCP 客户进程也是首先创建**传输控制模块** TCB。然后，在打算建立 TCP 连接时，向 B 发出连接请求报文段，这时首部中的同步位 SYN = 1，同时选择一个初始序号 seq = x。TCP 规定，SYN 报文段（即 SYN = 1 的报文段）不能携带数据，但要**消耗掉一个序号**。这时，TCP 客户进程进入 SYN-SENT（同步已发送）状态。

B 收到连接请求报文段后，如同意建立连接，则向 A 发送确认。在确认报文段中应把 SYN 位和 ACK 位都置 1，确认号是 ack = x + 1，同时也为自己选择一个初始序号 seq = y。请注意，这个报文段也不能携带数据，但同样**要消耗掉一个序号**。这时 TCP 服务器进程进入 SYN-RCVD（同步收到）状态。

TCP 客户进程收到 B 的确认后，还要向 B 给出确认。确认报文段的 ACK 置 1，确认号 ack = y + 1，而自己的序号 seq = x + 1。TCP 的标准规定，ACK 报文段可以携带数据。但**如果不携带数据则不消耗序号**，在这种情况下，下一个数据报文段的序号仍是 seq = x + 1。这时，TCP 连接已经建立，A 进入 ESTABLISHED（已建立连接）状态。

当 B 收到 A 的确认后，也进入 ESTABLISHED 状态。

上面给出的连接建立过程叫做**三报文握手**。请注意，在图 5-28 中 B 发送给 A 的报文段，也可拆成两个报文段。可以先发送一个确认报文段（ACK = 1, ack = x + 1），然后再发送一个同步报文段（SYN = 1, seq = y）。这样的过程就变成了**四报文握手**，但效果是一样的。

为什么 A 最后还要发送一次确认呢？这主要是为了防止已失效的连接请求报文段突然又传送到了 B，因而产生错误。

TCP 连接释放过程比较复杂，我们仍结合双方状态的改变来阐明连接释放的过程。

数据传输结束后，通信的双方都可释放连接。现在 A 和 B 都处于 ESTABLISHED 状态（图 5-29）。A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接。A 把连接释放报文段首部的终止控制位 FIN 置 1，其序号 seq = u，它等于前面已传送过的数据的最后一个字节的序号加 1。这时 A 进入 FIN-WAIT-1（终止等待 1）状态，等待 B 的确认。请注意，TCP 规定，FIN 报文段即使不携带数据，它也消耗掉一个序号。

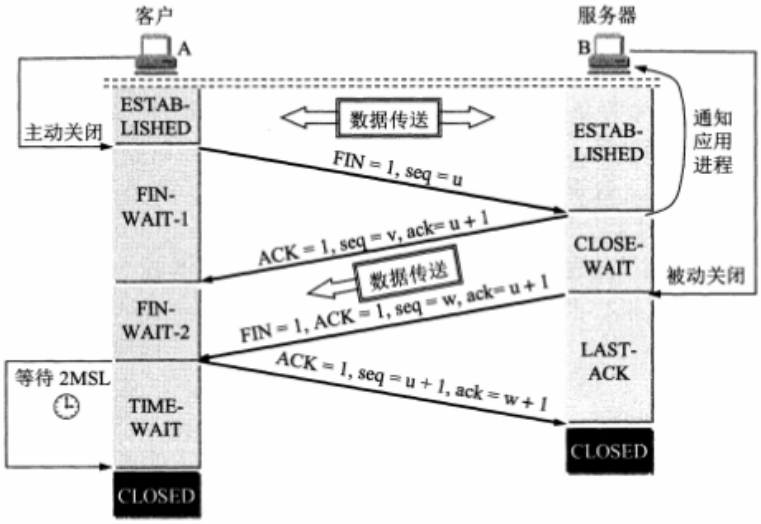


图 5-29　TCP 连接释放的过程

B 收到连接释放报文段后即发出确认，确认号是 ack = u + 1，而这个报文段自己的序号是 v，等于 B 前面已传送过的数据的最后一个字节的序号加 1。然后 B 就进入 CLOSE-WAIT（关闭等待）状态。TCP 服务器进程这时应通知高层应用进程，因而从 A 到 B 这个方向的连接就释放了，这时的 TCP 连接处于**半关闭**(half-close)状态，即 A 已经没有数据要发送了，但 B 若发送数据，A 仍要接收。也就是说，从 B 到 A 这个方向的连接并未关闭，这个状态可能会持续一段时间。

A 收到来自 B 的确认后，就进入 FIN-WAIT-2（终止等待 2）状态，等待 B 发出的连接释放报文段。

若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接。这时 B 发出的连接释放报文段必须使 FIN = 1。现假定 B 的序号为 w（在半关闭状态 B 可能又发送了一些数据）。B 还必须重复上次已发送过的确认号 ack = u + 1。这时 B 就进入 LAST-ACK（最后确认）状态，等待 A 的确认。

A 在收到 B 的连接释放报文段后，必须对此发出确认。在确认报文段中把 ACK 置 1，确认号 ack = w + 1，而自己的序号是 seq = u + 1（根据 TCP 标准，前面发送过的 FIN 报文段要消耗一个序号）。然后进入到 TIME-WAIT（时间等待）状态。请注意，现在 TCP 连接还没有释放掉。必须经过**时间等待计时器**(TIME-WAIT timer)设置的时间 2MSL 后，A 才进入到 CLOSED 状态。时间 MSL 叫做**最长报文段寿命**(Maximum Segment Lifetime)，RFC 793 建议设为 2 分钟。但这完全是从工程上来考虑的，对于现在的网络，MSL = 2 分钟可能太长了一些。因此 TCP 允许不同的实现可根据具体情况使用更小的 MSL 值。因此，从 A 进入到 TIME-WAIT 状态后，要经过 4 分钟才能进入到 CLOSED 状态，才能开始建立下一个新的连接。当 A 撤销相应的传输控制块 TCB 后，就结束了这次的 TCP 连接。

13：事务具有四个特性？

- 原子性(Atomicity)

- 一致性(Consistency)
- 隔离性(Isolation)
- 持续性(Durability)

14：树的先序、中序和后序的非递归实现？

一、先序遍历

1、递归算法

```
struct Tree
{
    int date;
    Tree* lchild;
    Tree* rchild;
    Tree(int x) :date(x), lchild(nullptr), rchild(nullptr) {}
};

void PreOrder(Tree* root)
{
    if (root != nullptr)
    {
        cout << root->date << " ";
        PreOrder(root->lchild);
        PreOrder(root->rchild);
    }
}
```

2、非递归算法

```
void PreOrder(Tree* root)
{
    stack<Tree* > Stack;
    if (root == nullptr)
        return;
    while (root != nullptr || !Stack.empty())
    {
        while (root != nullptr)
        {
            Stack.push(root);
            cout << root->date << " ";
            root = root->lchild;
        }
        root = Stack.top();
        Stack.pop();
        root = root->rchild;
    }
}
```

二、中序遍历

1、递归算法

```
void InOrder(Tree* root)
{
    if (root != nullptr)
    {
        InOrder(root->lchild);
        cout << root->date << " ";
```

```
        InOrder(root->rchild);
    }
}
```

2、非递归算法

```
void InOrder(Tree* root)
{
    stack<Tree*> s;
    while (root != nullptr || !s.empty())
    {
        if (root != nullptr)
        {
            s.push(root);
            root = root->lchild;
        }
        else
        {
            root = s.top(); s.pop();
            cout << root->date << " ";
            root = root->rchild;
        }
    }
}
```

三、后序遍历

1、递归算法

```
void PostOrder(Tree* root)
{
    if (root != nullptr)
    {
        PostOrder(root->lchild);
        PostOrder(root->rchild);
        cout << root->date << " ";
    }
}
```

2、非递归算法

```
void PostOrder(Tree* root)
{
    stack<Tree*> s;
    Tree* r = nullptr;//使用辅助指针，指向最近访问过的节点
    while (root != nullptr || !s.empty())
    {
        if (root != nullptr)
        {
            s.push(root);
            root = root->lchild;
        }
        else
        {
            root = s.top();
            if (root->rchild != nullptr && root->rchild != r)
```

```
                root = root->rchild;
            else
            {
                s.pop();
                cout << root->date << " ";
                r = root;
                root = nullptr;
            }
        }
    }
}
```

15：树的层次遍历？

1、方法1

```
void LevelOrder(Tree* root)
{
    if (root == nullptr)
        return;
    queue<Tree*> que;
    que.push(root);
    while (!que.empty())
    {
        root = que.front();
        cout << root->date << " ";
        que.pop();
        if (root->lchild != nullptr)
            que.push(root->lchild);
        if (root->rchild != nullptr)
            que.push(root->rchild);
    }
}
```

2、方法2

```
void LevelOrder(Tree * root)
{
    if (root == nullptr)
        return;
    queue<Tree*> que;
    que.push(root);
    while (!que.empty())
    {
        int size = que.size();
        while (size)
        {
            root = que.front();
            cout << root->date << " ";
            que.pop();
            if (root->lchild != nullptr)
                que.push(root->lchild);
            if (root->rchild != nullptr)
                que.push(root->rchild);
```

```
            --size;
        }
    }
}
```

16：static 关键字的作用？

（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。

17：const 关键字的作用？

（1）欲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；

（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”。

18：指针和引用的区别？

1.引用不可以为空，但指针可以为空。定义一个引用的时候，必须初始化；

2.引用一旦初始化后不可以再改变指向(但可以改变所指向对象的内容)，而指针可以改变指向。

3.引用的大小是所指向的变量的大小，因为引用只是一个别名而已；指针是指针（地址）本身的大小，32 位系统下，一般为 4 个字节。

4.引用比指针更安全。由于不存在空引用，并且引用一旦被初始化为指向一个对象，它就不能被改变为另一个对象的引用，因此引用很安全。对于指针来说，它可以随时指向别的对象，并且可以不被初始化，或为 NULL，所以不安全。const 指针虽然不能改变指向，但仍然存在空指针，并且有可能产生野指针（即多个指针指向一块内存，free 掉一个指针之后，别的指针就成了野指针）。

指针传递，值传递和引用传递

1.指针传递:传递的是一个地址值

2.值传递：被调函数的形式参数作为被调函数的局部变量处理，即在栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为了实参的一个副本。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，不会影响主调函数的实参变量的值。

3.引用传递：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。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

4.引用传递和指针传递是不同的，虽然它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，它将影响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， 那就得使用指向指针的指针，或者指针引用。

19：哈希表处理冲突的方法？

一、基本概念

哈希表，也叫散列表，是根据关键字而直接进行访问的数据结构。也就是说，它将关键字通过某种规则映射到数组中某个位置，以加快查找的速度。这个映射规则称为哈希函数（散列函数），存放记录的数组称为哈希表。哈希表建立了关键字和存储地址之间的一种直接映射关系。

二、处理冲突的方法

(1) 链地址法

链地址法是指把所有的冲突关键字存储在一个线性链表中，这个链表由其散列地址唯一标识。

(2) 开放定址法

开放定址法是指可存放新表项的空闲地址，既向它的同义词表项开放，又向它的非同义词表项开放。其数学递推公式为（Hi 表示冲突发生后第 i 次探测的散列地址）

Hi = (H(key) + di) % m

式中，i = 1，2，…，k，m 为散列表表长，di 为增量序列。di 通常有以下几种取法：

当 di = 1，2，…，m - 1 时，称为线性探测法。其特点是，冲突发生时顺序查看表中下一个单元，直到找出一个空单元或查遍全表。

当 di = 12，-12，22，-22，…，k2，-k2 时，又称为二次探测法。

当 di = 伪随机数序列时，称为伪随机探测法。

(3) 再散列法

当发生冲突时，利用另一个哈希函数再次计算一个地址。直到冲突不再发生。

(4) 建立一个公共溢出区

一旦由哈希函数得到的地址冲突，就都填入溢出表。

20：C++ 面向对象的三大特性和五个原则？

三大特性

封装：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。一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

继承：指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：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。 通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”(Inheritance)和“组合”(Composition)来实现。继承概念的实现方式有二类：实现继承与接口继承。

实现继承：是指直接使用基类的属性和方法而无需额外编码的能力；

接口继承：是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。

多态：是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

C++多态基于虚函数和虚继承实现。总之，C++多态的核心，就是用一个更通用的基类指针指向不同的子类实例，为了能调用正确的方法，我们需要用到虚函数和虚继承。在内存中，通过虚函数表来实现子类方法的正确调用；通过虚基类指针，仅保留一份基类的内存结构，避免冲突。

所谓虚，就是把“直接”的东西变“间接”。成员函数原先是由静态的成员函数指针来定义的，而虚函数则是由一个虚函数表来指向真正的函数指针，从而达到在运行时，间接地确定想要的函数实现。继承原先是直接将基类的内存空间拷贝一份来实现的，而虚继承则用一个虚基类指针来指向虚基类，避免基类的重复。

五大原则

单一职责原则 SRP(Single Responsibility Principle)：是指一个类的功能要单一，不能包罗万象。如同一个人一样，分配的工作不能太多，否则一天到晚虽然忙忙碌碌的，但效率却高不起来。

开放封闭原则 OCP(Open－Close Principle)：一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。比如：一个网络模块，原来只服务端功能，而现在要加入客户端功能，那么应当在不用修改服务端功能代码的前提下，就能够增加客户端功能的实现代码，这要求在设计之初，就应当将服务端和客户端分开，公共部分抽象出来。

里式替换原则 LSP(the Liskov Substitution Principle LSP)：子类应当可以替换父类并出现在父类能够出现的任何地方。比如：公司搞年度晚会，所有员工可以参加抽奖，那么不管是老员工还是新员工，也不管是总部员工还是外派员工，都应当可以参加抽奖，否则这公司就不和谐了。

依赖倒置原则 DIP(the Dependency Inversion Principle DIP)：具体依赖抽象，上层依赖下层。假设 B 是较 A 低的模块，但 B 需要使用到 A 的功能，这个时候，B 不应当直接使用 A 中的具体类： 而应当由 B 定义一抽象接口，并由 A 来实现这个抽象接口，B 只使用这个抽象接口：这样就达到了依赖倒置的目的，B 也解除了对 A 的依赖，反过来是 A 依赖于 B 定义的抽象接口。通过上层模块难以避免依赖下层模块，假如 B 也直接依赖 A 的实现，那么就可能 造成循环依赖。一个常见的问题就是编译 A 模块时需要直接包含到 B 模块的 cpp 文件，而编译 B 时同样要直接包含到 A 的 cpp 文件。

接口分离原则 ISP(the Interface Segregation Principle ISP)：模块间要通过抽象接口隔离开，而不是通过具体的类强耦合起来

21：多态的实现？

（1）编译时的多态性。编译时的多态性是通过重载来实现的。对于非虚的成员来说。系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。

（2）运行时的多态性。运行时的多态性就是直到系统运行时，才根据实际情况决定实现何种操作。C++中，运行时的多态性通过虚函数实现。

22：深拷贝和浅拷贝的区别？

浅拷贝

对一个已知对象进行拷贝，编译系统会自动调用一种构造函数——拷贝构造函数，如果用户未定义拷贝构造函数，则会调用默认拷贝构造函数，调用一次构造函数，调用两次析构函数，两个对象的指针成员所指内存相同，但是程序结束时该内存却被释放了两次，会造成内存泄漏问题。

深拷贝

在对含有指针成员的对象进行拷贝时，必须要自己定义拷贝构造函数，使拷贝后的对象指针成员有自己的内存空间，即进行深拷贝，这样就避免了内存泄漏发生，调用一次构造函数，一次自定义拷贝构造函数，两次析构函数。两个对象的指针成员所指内存不同。

总结：浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间，深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

23：vector 的实现原理

vector 的数据安排以及操作方式，与 array 非常相似。两者的唯一区别在于空间的运用的灵活性。array 是静态空间，一旦配置了就不能改变；要换个大（或小）一点的房子，可以，一切琐细都得由客户端自己来：首先配置一块新空间，然后将元素从旧址一一搬往新址，再把原来的空间释还给系统。vector 是动态空间，随着元素的加入，它的内部机制会自行扩充空间以容纳新元素。因此，vector 的运用对于内存的合理利用与运用的灵活性有很大的帮助，我们再也不必因为害怕空间不足而一开始要求一个大块头的 array 了，我们可以安心使用 array，吃多少用多少。

vector 的实现技术，关键在于其对大小的控制以及重新配置时的数据移动效率。一旦 vector 的旧有空间满载，如果客户端每新增一个元素，vector 的内部只是扩充一个元素的空间，实为不智。因为所谓扩充空间（不论多大），一如稍早所说，是” 配置新空间/数据移动/释还旧空间 “的大工程，时间成本很高，应该加入某种未雨绸缪的考虑。稍后我们便可看到 SGI vector 的空间配置策略了。

另外，由于 vector 维护的是一个连续线性空间，所以 vector 支持随机存取 。

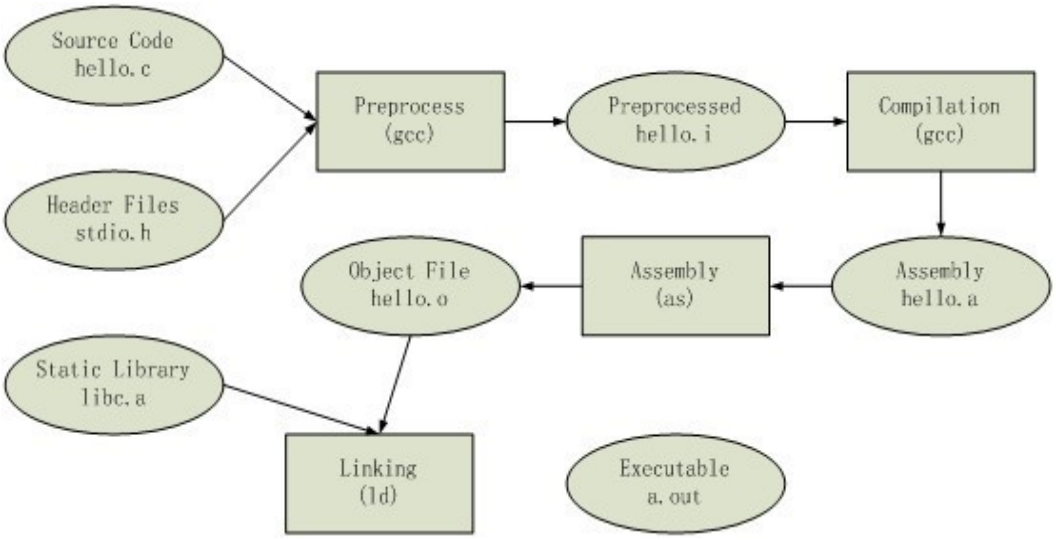
注意：vector 动态增加大小时，并不是在原空间之后持续新空间（因为无法保证原空间之后尚有可供配置的空间），而是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。因此， 对 vector 的任何操作，一旦引起空间重新配置，指向原 vector 的所有迭代器就都失效了 。这是程序员易犯的一个错误，务需小心。

24：C++ 源代码到可执行代码的详细过程 ？

编译，编译程序读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码，再由汇编程序转换为机器语言，并且按照操作系统对可执行文件格式的要求链接生成可执行程序。

源代码－－>预处理－－>编译－－>优化－－>汇编－－>链接-->可执行文件

Source--（编译）--> Assembly--（汇编）-->Obj--（链接）-->PE/ELF



1.编译预处理(Preprocessing)

读取源程序，对其中的伪指令（以#开头的指令）和特殊符号进行处理

通常使用以下命令来进行预处理：

gcc -E hello.c -o hello.i

参数-E 表示只进行预处理 或者也可以使用以下指令完成预处理过程

cpp hello.c > hello.i /* cpp - The C Preprocessor */

直接 cat hello.i 你就可以看到预处理后的代码

[析] 伪指令主要包括以下四个方面:

（1）宏定义指令，如#define Name TokenString,#undef 等。对于前一个伪指令，预编译所要做的是将程序中的所有 Name 用 TokenString 替换，但作为字符串常量的 Name 则不被替换。对于后者，则将取消对某个宏的定义，使以后该串的出现不再被替换。

（2）条件编译指令，如#ifdef,#ifndef,#else,#elif,#endif,等等。这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉

（3）头文件包含指令，如#include "FileName"或者#include <FileName>等。在头文件中一般用伪指令#define 定义了大量的宏（最常见的是字符常量），同时包含有各种外部符号的声明。采用头文件的目的主要是为了使某些定义可以供多个不同的 C 源程序使用。因为在需要用到这些定义的 C 源程序中，只需加上一条#include 语句即可，而不必再在此文件中将这些定义重复一遍。预编译程序将把头文件中的定义统统都加入到它所产生的输出文件中，以供编译程序对之进行处理。

包含到 c 源程序中的头文件可以是系统提供的，这些头文件一般被放在/usr/include 目录下。在程序中#include 它们要使用尖括号（<>）。另外开发人员也可以定义自己的头文件，这些文件一般与 c 源程序放在同一目录下，此时在#include 中要用双引号（""）。

（4）特殊符号，预编译程序可以识别一些特殊的符号。例如在源程序中出现的 LINE 标识将被解释为当前行号（十进制数），FILE 则被解释为当前被编译的 C 源程序的名称。预编译程序对于在源程序中出现的这些串将用合适的值进行替换。

预处理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。这个文件的含义同没有经过预处理的源文件是相同的，但内容有所不同。下一步，此输出文件将作为编译程序的输出而被翻译成为机器指令。

2.编译阶段(Compilation)

编译过程就是把预处理完的文件进行一系列的词法分析，语法分析，语义分析及优化后生成相应的汇编代码。

$gcc -S hello.i -o hello.s

或者

$ /usr/lib/gcc/i486-linux-gnu/4.4/cc1 hello.c

注：现在版本的 GCC 把预处理和编译两个步骤合成一个步骤，用 cc1 工具来完成。gcc 其实是后台程序的一些包装，根据不同参数去调用其他的实际处理程序，比如：预编译编译程序 cc1、汇编器 as、连接器 l

编译器在编译时是以 c/c++文件为单位进行的， 如果项目中没有 c/c++文件，那么你的项目将无法编译。经过预编译得到的输出文件中，将只有常量。如数字、字符串、变量的定义，以及 C 语言的关键字，如 main,if,else,for,while,{,},+,-,*,\, 等等。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。

3.优化阶段

优化处理是编译系统中一项比较艰深的技术。它涉及到的问题不仅同编译技术本身有关，而且同机器的硬件环境也有很大的关系。优化一部分是对中间代码的优化。这种优化不依赖于具体的计算机。另一种优化则主要针对目标代码的生成而进行的。上图中，我们将优化阶段放在编译程序的后面，这是一种比较笼统的表示。

对于前一种优化，主要的工作是删除公共表达式、循环优化（代码外提、强度削弱、变换循环控制条件、已知量的合并等）、复写传播，以及无用赋值的删除，等等。

后一种类型的优化同机器的硬件结构密切相关，最主要的是考虑是如何充分利用机器的各个硬件寄存器存放的有关变量的值，以减少对于内存的访问次数。另外，如何根据机器硬件执行指令的特点（如流水线、RISC、CISC、VLIW 等）而对指令进行一些调整使目标代码比较短，执行的效率比较高，也是一个重要的研究课题.

经过优化得到的汇编代码必须经过汇编程序的汇编转换成相应的机器指令，方可能被机器执行。

4.汇编过程(Assembly)

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。

$ gcc -c hello.c -o hello.o

或者

$ as hello.s -o hello.co 目标文件由段组成。通常一个目标文件中至少有两个段：

代码段 该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。

数据段 主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

UNIX 环境下主要有三种类型的目标文件：

（1）可重定位文件 其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。

（2）共享的目标文件 这种文件存放了适合于在两种上下文里链接的代码和数据。第一种事链接程序可把它与其它可重定位文件及共享的目标文件一起处理来创建另一个目标文件；第二种是动态链接程序将它与另一个可执行文件及其它的共享目标文件结合到一起，创建一个进程映象。

（3）可执行文件 它包含了一个可以被操作系统创建一个进程来执行之的文件。

汇编程序生成的实际上是第一种类型的目标文件。对于后两种还需要其他的一些处理方能得到，这个就是链接程序的工作了。

5.链接程序(Linking)

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

通过调用链接器 ld 来链接程序运行需要的一大堆目标文件，以及所依赖的其它库文件，最后生成可执行文件。

ld -static crt1.o crti.o crtbeginT.o hello.o -start-group -lgcc -lgcc_eh -lc-end-group crtend.o crtn.o (省略了文件的路径名)。

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诶操作系统装入执行的统一整体。

根据开发人员指定的同库函数的链接方式的不同，链接处理可分为两种：

（1）静态链接 在这种链接方式下，函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目

标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。(个人备注：静态链接将链接库的代码复制到可执行程序中，使得可执行程序体积变大)

（2）动态链接　　在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链指链接阶段仅仅只加入一些描述信息，而程序执行时再从系统中把相应动态库加载到内存中去。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

（个人备注：动态链接指的是需要链接的代码放到一个共享对象中，共享对象映射到进程虚地址空间，链接程序记录可执行程序将来需要用的代码信息，根据这些信息迅速定位相应的代码片段。）

对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小，并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链接要优越。在某些情况下动态链接可能带来一些性能上损害。

经过上述五个过程，C++源程序就最终被转换成可执行文件了。缺省情况下这个可执行文件的名字被命名为a.out。

25：memcpy 和 strcpy 的区别 ？

strcpy 和 memcpy 主要有以下 3 方面的区别。

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。

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，如果空间不够，就会引起内存溢出。memcpy 则是根据其第 3 个参数决定复制的长度。

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，由于字符串是以“\0”结尾的，所以对于在数据中包含“\0”的数据只能用 memcpy。

从 s1 复制字符串到 s2

strncpy 和 memcpy 很相似，只不过它在一个终止的空字符处停止。当 n>strlen(s1)时，给 s2 不够数的空间里填充“\0”（n 为 s2 的空间大小）；当 n<=strlen(s1)时，s2 是没有结束符“\0”的，所以使用 strncpy 时，确保 s2 的最后一个字符是“\0”。

26：vector 删除数据时有什么需要注意的吗 ？

先用迭代器指向 vector 的起始位置，然后用 while 循环，找到符合的元素，删除迭代器所指向的元素，返回一个指向被删元素之后元素的迭代器，这时不能在自增了，因为迭代器指针已经指向下一个元素了，如果在自增，就将被删除的元素的后面一个元素就跳过去了，如果在被删除的元素在末尾，那么迭代器指针就会变成野指针。

27：虚函数和纯虚函数的区别？

虚函数

引入原因：为了方便使用多态特性，我们常常需要在基类中定义虚函数。

纯虚函数在基类中是没有定义的，必须在子类中加以实现。

纯虚函数

引入原因：在很多情况下，基类本身生成对象是不合情理的。

纯虚函数就是基类只定义了函数体，没有实现过程，定义方法如下；

virtual void Eat() = 0; 直接=0 不要 在 cpp 中定义就可以了。

纯虚函数相当于接口，不能直接实例话，需要派生类来实现函数定义。

虚函数在子类里面也可以不重载的；但纯虚必须在子类去实现，

普通成员函数是静态编译的，没有运行时多态，只会根据指针或引用的“字面值”类对象，调用自己的普通函数；虚函数为了重载和多态的需要，在基类中定义的，即便定义为空；纯虚函数是在基类中声明的虚函数，它可以再基类中有定义，且派生类必须定义自己的实现方法。

一旦父类的成员函数声明 virtual，其子类的函数不管有没有声明为 virtual，都是虚函数。

普通函数如果不被使用，可以只有声明没有定义，虚函数必须要有定义，即使是一个空实现,因为编译器无法确定会使用哪个函数。

28：C++中 overload,override,overwrite 的区别？

Overload(重载)： 在 C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，

但参数或返回值不同（包括类型、顺序不同），即函数重载。
（1）相同的范围（在同一个类中）；
（2）函数名字相同；
（3）参数不同；
（4）virtual 关键字可有可无。

Override(覆盖)：是指派生类函数覆盖基类函数，特征是：
（1）不同的范围（分别位于派生类与基类）；
（2）函数名字相同；
（3）参数相同；
（4）基类函数必须有 virtual 关键字。

Overwrite(重写)：是指派生类的函数屏蔽了与其同名的基类函数，规则如下：
（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。
（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

29：C++中 4 种强制类型转换 ？

事情要从头说起,这个头就是 C 语言.我们已经习惯了使用 C-like 类型转换,因为它强大而且简单.
主要有一下两种形式:
(new-type) expression
new-type (expression)
C 风格的转换格式很简单，但是有不少缺点：
1.转换太过随意，可以在任意类型之间转换。你可以把一个指向 const 对象的指针转换成指向非 const 对象的指针，把一个指向基类对象的指针转换成一个派生类对象的指针，这些转换之间的差距是非常巨大的，但是传统的 C 语言风格的类型转换没有区分这些。
2.C 风格的转换没有统一的关键字和标示符。对于大型系统，做代码排查时容易遗漏和忽略。

C++中的类型转换:

C++风格完美的解决了上面两个问题。1.对类型转换做了细分，提供了四种不同类型转换，以支持不同需求的转换；2.类型转换有了统一的标示符，利于代码排查和检视。下面分别来介绍这四种转换:static_cast、dynamic_cast、const_cast 和 reinterpreter_cast.

static_cast，命名上理解是静态类型转换。如 int 转换成 char。
dynamic_cast，命名上理解是动态类型转换。如子类和父类之间的多态类型转换。
const_cast，字面上理解就是去 const 属性。
reinterpret_cast，仅仅重新解释类型，但没有进行二进制的转换。

一、static_cast 转换

1.**基本用法**：static_cast expression
2.**使用场景**：
a、用于类层次结构中基类和派生类之间指针或引用的转换
上行转换（派生类—->基类）是安全的；
下行转换（基类—->派生类）由于没有动态类型检查，所以是不安全的。
b、用于基本数据类型之间的转换，如把 int 转换为 char，这种带来安全性问题由程序员来保证
c、把空指针转换成目标类型的空指针
d、把任何类型的表达式转为 void 类型

3.**使用特点**
a、主要执行非多态的转换操作，用于代替 C 中通常的转换操作
b、隐式转换都建议使用 static_cast 进行标明和替换

```
int n = 6;
double d = static_cast<double>(n); // 基本类型转换
int *pn = &n;
```

```
double *d = static_cast<double *>(&n) //无关类型指针转换，编译错误
void *p = static_cast<void *>(pn); //任意类型转换成 void 类型
```

二、dynamic_cast 转换

1.**基本用法**：dynamic_cast expression

2.**使用场景**:只有在派生类之间转换时才使用dynamic_cast,type-id必须是类指针,类引用或者void*。

3.**使用特点**：

a、基类必须要有虚函数，因为 dynamic_cast 是运行时类型检查，需要运行时类型信息，而这个信息是存储在类的虚函数表中，只有一个类定义了虚函数，才会有虚函数表（如果一个类没有虚函数，那么一般意义上，这个类的设计者也不想它成为一个基类）。

b、对于下行转换，dynamic_cast 是安全的（当类型不一致时，转换过来的是空指针），而 static_cast 是不安全的（当类型不一致时，转换过来的是错误意义的指针，可能造成踩内存，非法访问等各种问题）

c、dynamic_cast 还可以进行交叉转换

```
class BaseClass
{
public:
    int m_iNum;
    virtual void foo(){};//基类必须有虚函数。保持多台特性才能使用 dynamic_cast
};
class DerivedClass: public BaseClass
{
public:
    char *m_szName[100];
    void bar(){};
};
BaseClass* pb = new DerivedClass();
DerivedClass *pd1 = static_cast<DerivedClass *>(pb);//子类->父类，静态类型转换，正确但不推荐
DerivedClass *pd2 = dynamic_cast<DerivedClass *>(pb);//子类->父类，动态类型转换，正确
BaseClass* pb2 = new BaseClass();
//父类->子类，静态类型转换，危险！访问子类 m_szName 成员越界
DerivedClass *pd21 = static_cast<DerivedClass *>(pb2);
//父类->子类，动态类型转换，安全的。结果是 NULL
DerivedClass *pd22 = dynamic_cast<DerivedClass *>(pb2);
```

三、const_cast 转换

1.**基本用法**：const_castexpression

2.**使用场景**：

a、常量指针转换为非常量指针，并且仍然指向原来的对象

b、常量引用被转换为非常量引用，并且仍然指向原来的对象

3.**使用特点**：

a、cosnt_cast 是四种类型转换符中唯一可以对常量进行操作的转换符

b、去除常量性是一个危险的动作，尽量避免使用。一个特定的场景是：类通过 const 提供重载时，一般都是非常量函数调用const_cast将参数转换为常量,然后调用常量函数,然后得到结果再调用const_cast去除常量性。

```
struct SA
{
    int i;
};
const SA ra;
//ra.i = 10; //直接修改 const 类型，编译错误
SA &rb = const_cast<SA&>(ra);
rb.i = 10;
```

四、reinterpret_cast 转换

1. **基本用法**：reinterpret_castexpression
2. **使用场景**：不到万不得已，不用使用这个转换符，高危操作
3. **使用特点**：

a、reinterpret_cast 是从底层对数据进行重新解释，依赖具体的平台，可移植性差

b、reinterpret_cast 可以将整型转换为指针，也可以把指针转换为数组

c、reinterpret_cast 可以在指针和引用里进行肆无忌惮的转换

```
int doSomething(){return 0;};
//FuncPtr is 一个指向函数的指针，该函数没有参数，返回值类型为 void
typedef void(*FuncPtr)();
//10 个 FuncPtrs 指针的数组 让我们假设你希望（因为某些莫名其妙的原因）把一个指向下面函数的指针存
//入 funcPtrArray 数组：
FuncPtr funcPtrArray[10];
funcPtrArray[0] = &doSomething;
// 编译错误！类型不匹配，reinterpret_cast 可以让编译器以你的方法去看待它们：funcPtrArray
funcPtrArray[0] = reinterpret_castFuncPtr>(&doSomething);
//不同函数指针类型之间进行转换
```

总结：

去 const 属性用 const_cast。

基本类型转换用 static_cast。

多态类之间的类型转换用 daynamic_cast。

不同类型的指针类型转换用 reinterpret_cast。

30：有了 malloc/free，为什么还要 new/delete？

malloc 与 free 是 C/C++的标准库函数，new/delete 是 C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象的消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。

因此，C++需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。注意：new/delete 不是库函数。请看下面例子：

```
#include <iostream>
using namespace std;
class Obj
{
public:
    Obj(void)
    {
        cout << "Initialization" << endl;
    }
    ~Obj(void)
    {
        cout << "Destroy" << endl;
    }
};
void UseMallocFree(void)
{
    cout << "in UseMallocFree()" << endl;
    Obj *a = (Obj*)malloc(sizeof(Obj));
```

```
    free(a);
}
void UseNewDelete(void)
{
    cout << "in UseNewDelete()" << endl;
    Obj *a = new Obj;
    delete a;
}

int main()
{
    UseMallocFree();
    UseNewDelete();
    return 0;
}
```

执行结果：

```
in UseMallocFree()
in UseNewDelete()
Initialization
Destroy
```

在这个示例中，类 Obj 只有构造函数和析构函数的定义，这两个成员函数分别打印一句话。函数 UseMallocFree()中调用 malloc/free 申请和释放堆内存；函数 UseNewDelete()中调用 new/delete 申请和释放堆内存。可以看出函数 UseMallocFree()执行时，类 Obj 的构造函数和析构函数都不会被调用；而函数 UseNewDelete()执行时，类 Obj 的构造函数和析构函数都会被调用。

总结

对于非内部数据类型的对象而言，对象的消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能把执行构造函数和析构函数的任务强加于 malloc/free，因此只有使用 new/delete 运算符。

31：map 可以用结构体作为健值吗，已经注意事项？

在使用 map 时，有时候我们需要自定义键值，才能符合程序的需要。

比如我们需要使用自定义的结构体来作为 map 的键值：

```
struct  Test
{
    int x;
    int y;
};
```

这样直接使用的话，在编译时会出问题：

```
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

看错误是说，键值无法比较。因为 map 的键值是自动比较后进插入的，键值是递增的。

现在我们自定义的键值，编译器无法进行比较，找不到类似的模板，所以报错。

既然是没有‘<’，那我们自己重载小于操作符应该就可以了：

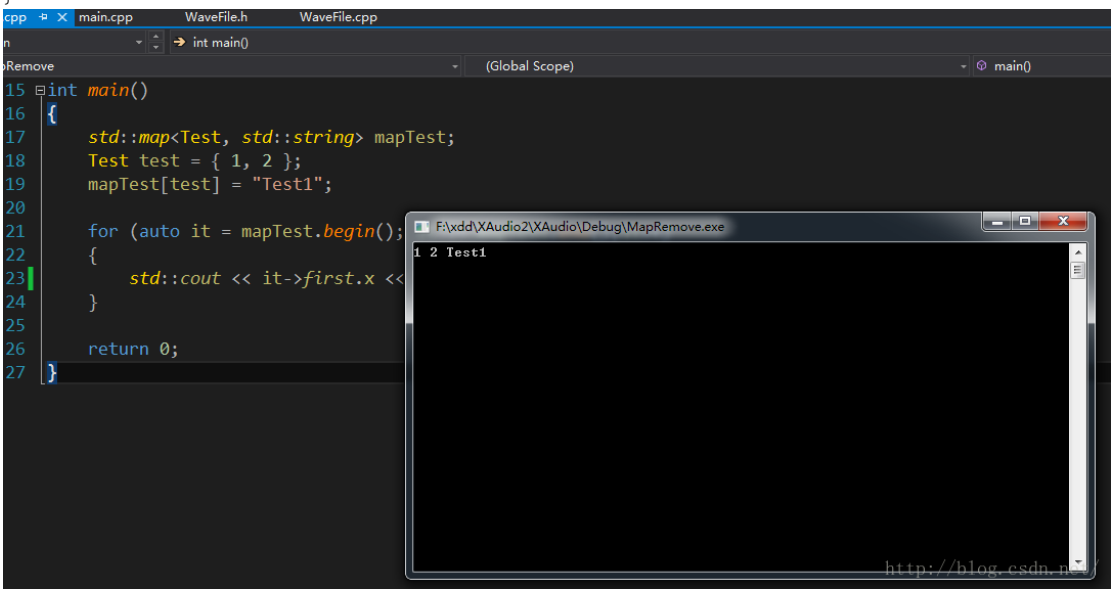
```
struct  Test
{
    int x;
    int y;
    bool operator < (const Test &o) const
    {
        return x < o.x || y < o.y;
    }
};
```

重载后，重新编译，顺利通过。测试代码如下：

```
#include <map>
#include <iostream>
struct  Test
{
    int x;
    int y;
    bool operator < (const Test &o) const
    {
        return x < o.x || y < o.y;
    }
};
 int main()
{
    std::map<Test, std::string> mapTest;
    Test test = { 1, 2 };
    mapTest[test] = "Test1";

    for (auto it = mapTest.begin(); it != mapTest.end();it++)
    {
        std::cout << it->first.x << " " << it->first.y << " " << it->second.c_str() << std::endl;
    }

    return 0;
}
```



32：Volatile 的作用？

背景

此微博，引发了朋友们的大量讨论：赞同者有之；批评者有之；当然，更多的朋友，是希望我能更详细的解读 C/C++ Volatile 关键词，来佐证我的微博观点。而这，正是我写这篇博文的初衷：本文，将详细分析 C/C++ Volatile 关键词的功能（有多种功能）、Volatile 关键词在多线程编程中存在的问题、Volatile 关键词与编译器/CPU 的关系、C/C++ Volatile 与 Java Volatile 的区别，以及 Volatile 关键词的起源，希望对大家更好的理解、使用 C/C++ Volatile，有所帮助。

Volatile，词典上的解释为：易失的；易变的；易挥发的。那么用这个关键词修饰的 C/C++变量，应该也能够体现出”易变”的特征。大部分人认识 Volatile，也是从这个特征出发，而这也是本文揭秘的 C/C++ Volatile 的第一个特征。

Volatile：易变的

在介绍 C/C++ Volatile 关键词的”易变”性前，先让我们看看以下的两个代码片段，以及他们对应的汇编指令（以下用例的汇编代码，均为 VS 2008 编译出来的 Release 版本）：

测试用例一：非 Volatile 变量

代码	汇编

void main ()
{
int a = 5;
int b = 10;
int c = 20;
int d;
 	
scanf("%d", &c);
 	
a = fn(c);
	call dword ptr [__imp__scanf (12A20A8h)]
	mov eax,dword ptr [esp+8]
	add esp,8

b = a + 1;	lea ecx,[eax+1]

d = fn(b);
 	
cout << a << b << c << d;
}	

b = a + 1;这条语句，对应的汇编指令是：lea ecx, [eax + 1]。由于变量 a，在前一条语句 a = fn(c)执行时，被缓存在了寄存器 eax 中，因此 b = a + 1；语句，可以直接使用仍旧在寄存器 eax 中的 a，来进行计算，对应的也就是汇编：[eax + 1]。

测试用例二：Volatile 变量

代码	汇编

void main ()
{
volatile inta = 5;
int b = 10;
int c = 20;
int d;
 	
scanf("%d", &c);
 	
a = fn(c);
	mov ecx,dword ptr [esp+8]
	mov dword ptr [esp+0Ch],ecx

b = a + 1;	mov eax,dword ptr [esp+0Ch] ... inc eax

d = fn(b);
 	
cout << a << b << c << d;
}	

与测试用例一唯一的不同之处，是变量 a 被设置为 volatile 属性，一个小小的变化，带来的是汇编代码上很大的变化。a = fn(c)执行后，寄存器 ecx 中的 a，被写回内存：mov dword ptr [esp+0Ch], ecx。然后，在执行 b = a + 1；语句时，变量 a 有重新被从内存中读取出来：mov eax, dword ptr [esp + 0Ch]，而不再直接使用寄存器 ecx 中的内容。

小结

从以上的两个用例，就可以看出 C/C++ Volatile 关键词的第一个特性：易变性。所谓的易变性，在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的 volatile 变量的寄存器内容，而是重新从内存中读取。volatile 的这个特性，相信也是大部分朋友所了解的特性。

Volatile：不可优化的

与前面介绍的”易变”性类似，关于 C/C++ Volatile 关键词的第二个特性：”不可优化”性，也通过两个对比的代码片段来说明：

测试用例三：非 Volatile 变量

代码	汇编

void main ()
{
int a;
int b;
int c;
 	
a = 1;
b = 2;
c = 3;
 	
printf("%d, %d, %d", a, b, c);
 	
}
	
	push 3
	push 2
	push 1
	call ...

在这个用例中，非 volatile 变量 a，b，c 全部被编译器优化掉了 (optimize out)，因为编译器通过分析，发觉 a，b，c 三个变量是无用的，可以进行常量替换。最后的汇编代码相当简介，高效率。

测试用例四：Volatile 变量

代码	汇编

void main ()
{
volatile int a;
volatile int b;
volatile int c;
 	
a = 1;
b = 2;
c = 3;
 	
printf("%d, %d, %d", a, b, c);
 	
}
	
	mov eax,dword ptr [esp]
	mov ecx,dword ptr [esp+4]
	mov edx,dword ptr [esp+8]
	
	push eax
	push ecx
	push edx
	call ...

测试用例四，与测试用例三类似，不同之处在于，a，b，c 三个变量，都是 volatile 变量。这个区别，反映到汇编语言中，就是三个变量仍旧存在，需要将三个变量从内存读入到寄存器之中，然后再调用 printf() 函数。

小结

从测试用例三、四，可以总结出 C/C++ Volatile 关键词的第二个特性：“不可优化”特性。volatile 告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。相对于前面提到的第一个特性：”易变”性，”不可优化”特性可能知晓的人会相对少一些。但是，相对于下面提到的 C/C++ Volatile 的第三个特性，无论是”易变”性，还是”不可优化”性，都是 Volatile 关键词非常流行的概念。

Volatile：顺序性

C/C++ Volatile 关键词前面提到的两个特性，让 Volatile 经常被解读为一个为多线程而生的关键词：

一个全局变量，会被多线程同时访问/修改，那么线程内部，就不能假设此变量的不变性，并且基于此假设，来做一些程序设计。当然，这样的假设，本身并没有什么问题，多线程编程，并发访问/修改的全局变量，通常都会建议加上 Volatile 关键词修饰，来防止 C/C++编译器进行不必要的优化。但是，很多时候，C/C++ Volatile 关键词，在多线程环境下，会被赋予更多的功能，从而导致问题的出现。

回到本文背景部分我的那篇微博，我的这位朋友，正好犯了一个这样的问题。其对 C/C++ Volatile 关键词的使用，可以抽象为下面的伪代码：

```
代码

int something = 0;
volatile int flag = false;

Thread1 ()                              Thread2 ()
{                                       {
      // do something;                        if (flag == true)
      // 实际情况，肯定更加复杂               {
      something = 1;                              // assert something happens;
                                                  // 实际情况，在假设something已经
      flag = true;                                // 发生的前提下，做接下来的工作
}                                                 assert (something == 1);

                                                  // do other things,depends on sth
                                                  other things;
                                            }
                                      }
```

这段伪代码，声明另一个 Volatile 的 flag 变量。一个线程(Thread1)在完成一些操作后，会修改这个变量。而另外一个线程(Thread2)，则不断读取这个 flag 变量，由于 flag 变量被声明了 volatile 属性，因此编译器在编译时，并不会每次都从寄存器中读取此变量，同时也不会通过各种激进的优化，直接将 if (flag == true)改写为 if (false == true)。只要 flag 变量在 Thread1 中被修改，Thread2 中就会读取到这个变化，进入 if 条件判断，然后进入 if 内部进行处理。在 if 条件的内部，由于 flag == true，那么假设 Thread1 中的 something 操作一定已经完成了，在基于这个假设的基础上，继续进行下面的 other things 操作。

通过将 flag 变量声明为 volatile 属性，很好的利用了本文前面提到的 C/C++ Volatile 的两个特性:”易变”性;”不可优化”性。按理说，这是一个对于 volatile 关键词的很好应用，而且看到这里的朋友，也可以去检查检查自己的代码，我相信肯定会有这样的使用存在。

但是，这个多线程下看似对于 C/C++ Volatile 关键词完美的应用，实际上却是有大问题的。问题的关键，就在于前面标红的文字：由于 flag = true，那么假设 Thread1 中的 something 操作一定已经完成了。flag == true，为什么能够推断出 Thread1 中的 something 一定完成了？其实既然我把这作为一个错误的用例，答案是一目了然的：这个推断不能成立，你不能假设看到 flag == true 后，flag = true;这条语句前面的 something 一定已经执行完成了。这就引出了 C/C++ Volatile 关键词的第三个特性：顺序性。

同样，为了说明 C/C++ Volatile 关键词的”顺序性”特征，下面给出三个简单的用例 (注：与上面的测试用例不同，下面的三个用例，基于的是 Linux 系统，使用的是”GCC: (Debian 4.3.2-1.1) 4.3.2″)：

测试用例五：非 Volatile 变量

代码	汇编
// cordering.c	gcc -O2 -S -masm=intel cordering.c cat cordering.s

int A, B;
void foo()
{
A = B + 1;
B = 0;
}
	mov eax, DWORD PTR B[rip]
	mov DWORD PTR B[rip], 0
	add eax, 1
	mov DWORD PTR A[rip], eax
	ret

一个简单的示例，全局变量 A，B 均为非 volatile 变量。通过 gcc O2 优化进行编译，你可以惊奇的发现，A，B 两个变量的赋值顺序被调换了!!! 在对应的汇编代码中，B = 0 语句先被执行，然后才是 A = B + 1 语句被执行。

在这里，我先简单的介绍一下 C/C++编译器最基本优化原理：保证一段程序的输出，在优化前后无变化。将此原理应用到上面，可以发现，虽然 gcc 优化了 A，B 变量的赋值顺序，但是 foo()函数的执行结果，优化前后没有发生任何变化，仍旧是 A = 1；B = 0。因此这么做是可行的。

测试用例六：一个 Volatile 变量

代码	汇编
// cordering.c	gcc -O2 -S -masm=intel cordering.c

int A;
volatile int B;
void foo()
{
A = B + 1;
B = 0;
}
	mov eax, DWORD PTR B[rip]
	mov DWORD PTR B[rip], 0
	add eax, 1
	mov DWORD PTR A[rip], eax
	ret

此测试，相对于测试用例五，最大的区别在于，变量 B 被声明为 volatile 变量。通过查看对应的汇编代码，B 仍旧被提前到 A 之前赋值，Volatile 变量 B，并未阻止编译器优化的发生，编译后仍旧发生了乱序现象。

如此看来，C/C++ Volatile 变量，与非 Volatile 变量之间的操作，是可能被编译器交换顺序的。

通过此用例，已经能够很好的说明，本章节前面，通过 flag == true，来假设 something 一定完成是不成立的。在多线程下，如此使用 volatile，会产生很严重的问题。但是，这不是终点，请继续看下面的测试用例七。

测试用例七：两个 Volatile 变量

代码	汇编
// cordering.c	gcc -O2 -S -masm=intel cordering.c

volatile int A;
volatile int B;
void foo()
{
A = B + 1;
B = 0;
}
	mov eax, DWORD PTR B[rip]
	add eax, 1
	mov DWORD PTR A[rip], eax
	mov DWORD PTR B[rip], 0
	ret

同时将 A，B 两个变量都声明为 volatile 变量，再来看看对应的汇编。奇迹发生了，A，B 赋值乱序的现象消失。此时的汇编代码，与用户代码顺序高度一直，先赋值变量 A，然后赋值变量 B。

如此看来，C/C++ Volatile 变量间的操作，是不会被编译器交换顺序的。

happens-before

通过测试用例六，可以总结出：C/C++ Volatile 变量与非 Volatile 变量间的操作顺序，有可能被编译器交换。因此，上面多线程操作的伪代码，在实际运行的过程中，就有可能变成下面的顺序：

代码

```
int something = 0;
volatile int flag = false;

Thread1 ()
{
    // do something;
    // 实际情况，肯定更加复杂
    flag = true;
    something = 1;
}

Thread2 ()
{
    if (flag == true)
    {
        // assert something happens;
        // 实际情况，在假设something已经
        // 发生的前提下，做接下来的工作
        assert (something == 1);
        // do other things,depends on sth
        other things;
    }
}
```

顺序交换

由于 Thread1 中的代码执行顺序发生变化，flag = true 被提前到 something 之前进行，那么整个 Thread2 的假设全部失效。由于 something 未执行，但是 Thread2 进入了 if 代码段，整个多线程代码逻辑出现问题，导致多线程完全错误。

细心的读者看到这里，可能要提问，根据测试用例七，C/C++ Volatile 变量间，编译器是能够保证不交换顺序的，那么能不能将 something 中所有的变量全部设置为 volatile 呢？这样就阻止了编译器的乱序优化，从而也就保证了这个多线程程序的正确性。

针对此问题，很不幸，仍旧不行。将所有的变量都设置为 volatile，首先能够阻止编译器的乱序优化，这一点是可以肯定的。但是，别忘了，编译器编译出来的代码，最终是要通过 CPU 来执行的。目前，市场上有各种不同体系架构的 CPU 产品，CPU 本身为了提高代码运行的效率，也会对代码的执行顺序进行调整，这就是所谓的 CPU Memory Model (CPU 内存模型)。关于 CPU 的内存模型，可以参考这些资料：Memory Ordering From Wiki；Memory Barriers Are Like Source Control Operations From Jeff Preshing；CPU Cache and Memory Ordering From 何登成。下面，是截取自 Wiki 上的一幅图，列举了不同 CPU 架构，可能存在的指令乱序。

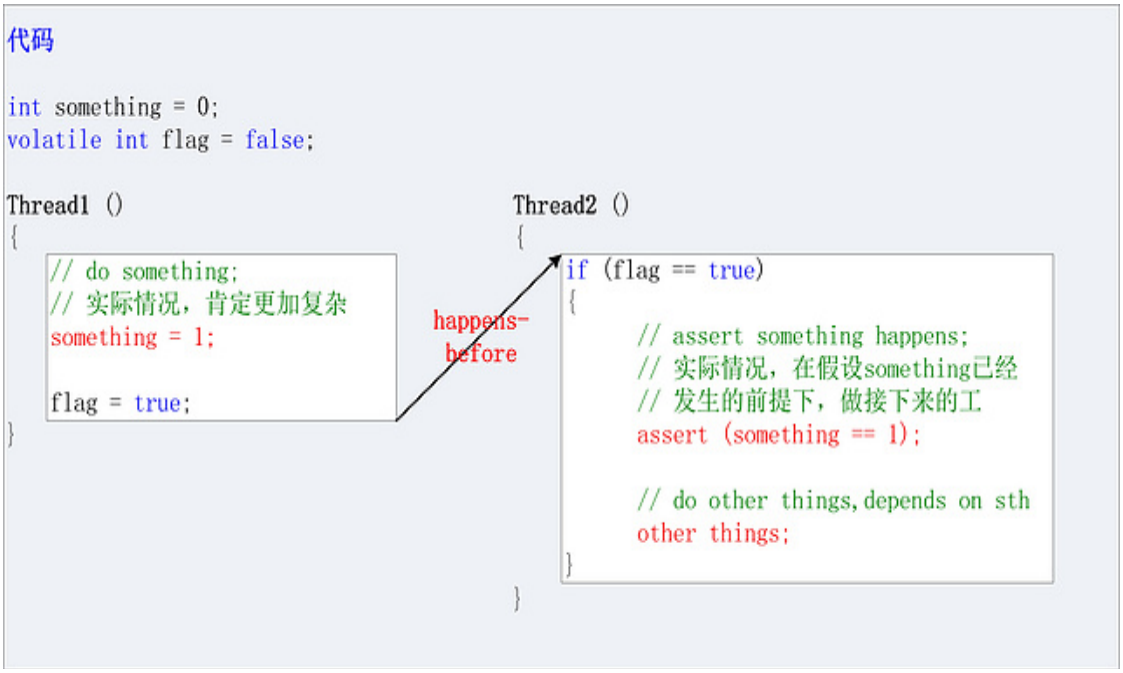
Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

从图中可以看到，X86 体系(X86，AMD64)，也就是我们目前使用最广的 CPU，也会存在指令乱序执行的行为：StoreLoad 乱序，读操作可以提前到写操作之前进行。

因此，回到上面的例子，哪怕将所有的变量全部都声明为 volatile，哪怕杜绝了编译器的乱序优化，但是针对生成的汇编代码，CPU 有可能仍旧会乱序执行指令，导致程序依赖的逻辑出错，volatile 对此无能为力。

其实，针对这个多线程的应用，真正正确的做法，是构建一个 happens-before 语义。关于 happens-before 语义的定义，可参考文章：The Happens-Before Relation。下面，用图的形式，来展示 happens-before 语义：



如图所示，所谓的 happens-before 语义，就是保证 Thread1 代码块中的所有代码，一定在 Thread2 代码块的第一条代码之前完成。当然，构建这样的语义有很多方法，我们常用的 Mutex、Spinlock、RWLock，都能保证这个语义（关于 happens-before 语义的构建，以及为什么锁能保证 happens-before 语义，以后专门写一篇文章进行讨论）。但是，C/C++ Volatile 关键词不能保证这个语义，也就意味着 C/C++ Volatile 关键词，在多线程环境下，如果使用的不够细心，就会产生如同我这里提到的错误。

小结

C/C++ Volatile 关键词的第三个特性：”顺序性”，能够保证 Volatile 变量间的顺序性，编译器不会进行乱序优化。Volatile 变量与非 Volatile 变量的顺序，编译器不保证顺序，可能会进行乱序优化。同时，C/C++ Volatile 关键词，并不能用于构建 happens-before 语义，因此在进行多线程程序设计时，要小心使用 volatile，不要掉入 volatile 变量的使用陷阱之中。

Volatile 的起源

C/C++的 Volatile 关键词，有三个特性：易变性；不可优化性；顺序性。那么，为什么 Volatile 被设计成这样呢？要回答这个问题，就需要从 Volatile 关键词的产生说起。(注：这一小节的内容，参考自 C++ and the Perils of Double-Checked Locking 论文的第 10 章节：

volatile：A Brief History。这是一篇顶顶好的论文，值得多次阅读，强烈推荐！）

Volatile 关键词，最早出现于 19 世纪 70 年代，被用于处理 memory-mapeed I/O (MMIO) 带来的问题。在引入 MMIO 之后，一块内存地址，既有可能是真正的内存，也有可能被映射到一个 I/O 端口。相对的，读写一个内存地址，既有可能操作内存，也有可能读写的是一个 I/O 设备。MMIO 为什么需要引入 Volatile 关键词？考虑如下的一个代码片段：

```
unsigned int *p = GetMagicAddress();
unsigned int a, b;

a = *p;                    (1)
b = *p;                    (2)

*p = a;                    (3)
*p = b;                    (4)
```

在此代码片段中，指针 p 既有可能指向一个内存地址，也有可能指向一个 I/O 设备。如果指针 p 指向的是 I/O 设备，那么(1)，(2)中的 a，b，就会接收到 I/O 设备的连续两个字节。但是，p 也有可能指向内存，此时，编译器的优化策略，就可能会判断出 a，b 同时从同一内存地址读取数据，在做完(1)之后，直接将 a 赋值给 b。对于 I/O 设备，需要防止编译器做这个优化，不能假设指针 b 指向的内容不变——易变性。

同样，代码(3)，(4)也有类似的问题，编译器发现将 a，b 同时赋值给指针 p 是无意义的，因此可能会优化代码(3)中的赋值操作，仅仅保留代码(4)。对于 I/O 设备，需要防止编译器将写操作给彻底优化消失了——”不可优化”性。

对于 I/O 设备，编译器不能随意交互指令的顺序，因为顺序一变，写入 I/O 设备的内容也就发生变化了——”顺序性”。

基于 MMIO 的这三个需求，设计出来的 C/C++ Volatile 关键词，所含有的特性，也就是本文前面分析的三个特性：易变性；不可优化性；顺序性。

33：了解哪些 c++11 特性？

最近工作中，遇到一些问题，使用 C++11 实现起来会更加方便，而线上的生产环境还不支持 C++11，于是决定新年开工后，在组内把 C++11 推广开来，整理以下文档，方便自己查阅，也方便同事快速上手。（对于异步编程十分实用的 Future/Promise 以及智能指针等，将不做整理介绍，组内使用的框架已经支持并广泛使用了，用的是自己公司参考 boost 实现的版本）

1. nullptr

nullptr 出现的目的是为了替代 NULL。

在某种意义上来说，传统 C++ 会把 NULL、0 视为同一种东西，这取决于编译器如何定义 NULL，有些编译器会将 NULL 定义为 ((void*)0)，有些则会直接将其定义为 0。

C++ 不允许直接将 void* 隐式转换到其他类型，但如果 NULL 被定义为 ((void*)0)，那么当编译 char *ch = NULL;时，NULL 只好被定义为 0。

而这依然会产生问题，将导致了 C++ 中重载特性会发生混乱，考虑：

```
void foo(char *);
void foo(int);
```

对于这两个函数来说，如果 NULL 又被定义为了 0 那么 foo(NULL); 这个语句将会去调用 foo(int)，从而导致代码违反直观。

为了解决这个问题，C++11 引入了 nullptr 关键字，专门用来区分空指针、0。

nullptr 的类型为 nullptr_t，能够隐式的转换为任何指针或成员指针的类型，也能和他们进行相等或者不等的比较。

当需要使用 NULL 时候，养成直接使用 nullptr 的习惯。

2. 类型推导

C++11 引入了 auto 和 decltype 这两个关键字实现了类型推导，让编译器来操心变量的类型。

auto

auto 在很早以前就已经进入了 C++，但是他始终作为一个存储类型的指示符存在，与 register 并存。在传统 C++ 中，如果一个变量没有声明为 register 变量，将自动被视为一个 auto 变量。而随着 register 被弃用，对 auto 的语义变更也就非常自然了。

使用 auto 进行类型推导的一个最为常见而且显著的例子就是迭代器。在以前我们需要这样来书写一个迭代器：

```
for(vector<int>::const_iterator itr = vec.cbegin(); itr != vec.cend(); ++itr)
```

而有了 auto 之后可以：

```
// 由于 cbegin() 将返回 vector<int>::const_iterator
// 所以 itr 也应该是 vector<int>::const_iterator 类型
for(auto itr = vec.cbegin(); itr != vec.cend(); ++itr);
```

一些其他的常见用法：

```
auto i = 5;              // i 被推导为 int
auto arr = new auto(10) // arr 被推导为 int *
```

注意：auto 不能用于函数传参，因此下面的做法是无法通过编译的（考虑重载的问题，我们应该使用模板）：

```
int add(auto x, auto y);
```

此外，auto 还不能用于推导数组类型：

```
#include <iostream>

int main() {
 auto i = 5;
 int arr[10] = {0};
 auto auto_arr = arr;
 auto auto_arr2[10] = arr;
 return 0;
}
```

decltype

decltype 关键字是为了解决 auto 关键字只能对变量进行类型推导的缺陷而出现的。它的用法和 sizeof 很相似：

```
decltype(表达式)
```

在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值。

有时候，我们可能需要计算某个表达式的类型，例如：

```
auto x = 1;
auto y = 2;
decltype(x+y) z;
```

拖尾返回类型、auto 与 decltype 配合

你可能会思考，auto 能不能用于推导函数的返回类型。考虑这样一个例子加法函数的例子，在传统 C++ 中我们必须这么写：

```
template<typename R, typename T, typename U>
R add(T x, U y) {
    return x+y
}
```

这样的代码其实变得很丑陋，因为程序员在使用这个模板函数的时候，必须明确指出返回类型。但事实上我们并不知道 add() 这个函数会做什么样的操作，获得一个什么样的返回类型。

在 C++11 中这个问题得到解决。虽然你可能马上回反应出来使用 decltype 推导 x+y 的类型，写出这样的代码：

```
decltype(x+y) add(T x, U y);
```

但事实上这样的写法并不能通过编译。这是因为在编译器读到 decltype(x+y) 时，x 和 y 尚未被定义。为了解决这个问题，C++11 还引入了一个叫做**拖尾返回类型（trailing return type）**，利用 auto 关键字将返回类型后置：

```
template<typename T, typename U>
auto add(T x, U y) -> decltype(x+y) {
    return x+y;
```

```
}
```

从 C++14 开始是可以直接让普通函数具备返回值推导，因此下面的写法变得合法：

```
template<typename T, typename U>
auto add(T x, U y) {
    return x+y;
}
```

3. 区间迭代

基于范围的 for 循环

C++11 引入了基于范围的迭代写法，我们拥有了能够写出像 Python 一样简洁的循环语句。

最常用的 std::vector 遍历将从原来的样子：

```
std::vector<int> arr(5, 100);
for(std::vector<int>::iterator i = arr.begin(); i != arr.end(); ++i) {
    std::cout << *i << std::endl;
}
```

变得非常的简单：

```
// & 启用了引用
for(auto &i : arr) {
    std::cout << i << std::endl;
}
```

4. 初始化列表

C++11 提供了统一的语法来初始化任意的对象，例如：

```
struct A {
    int a;
    float b;
};
struct B {

    B(int _a, float _b): a(_a), b(_b) {}
private:
    int a;
    float b;
};

A a {1, 1.1};    // 统一的初始化语法
B b {2, 2.2};
```

C++11 还把初始化列表的概念绑定到了类型上，并将其称之为 std::initializer_list，允许构造函数或其他函数像参数一样使用初始化列表，这就为类对象的初始化与普通数组和 POD 的初始化方法提供了统一的桥梁，例如：

```
#include <initializer_list>

class Magic {
public:
    Magic(std::initializer_list<int> list) {}
};

Magic magic = {1,2,3,4,5};
std::vector<int> v = {1, 2, 3, 4};
```

5. 模板增强

外部模板

传统 C++ 中，模板只有在使用时才会被编译器实例化。只要在每个编译单元（文件）中

编译的代码中遇到了被完整定义的模板，都会实例化。这就产生了重复实例化而导致的编译时间的增加。并且，我们**没有办法通知编译器不要触发模板实例化**。

C++11 引入了外部模板，扩充了原来的强制编译器在特定位置实例化模板的语法，使得**能够显式的告诉编译器何时进行模板的实例化**：

```
template class std::vector<bool>;            // 强行实例化
extern template class std::vector<double>;  // 不在该编译文件中实例化模板
```

尖括号 “>”

在传统 C++ 的编译器中，>>一律被当做右移运算符来进行处理。但实际上我们很容易就写出了嵌套模板的代码：

```
std::vector<std::vector<int>> wow;
```

这在传统 C++编译器下是不能够被编译的，而 C++11 开始，连续的右尖括号将变得合法，并且能够顺利通过编译。

类型别名模板

在传统 C++中，typedef 可以为类型定义一个新的名称，但是却没有办法为模板定义一个新的名称。因为，模板不是类型。例如：

```
template< typename T, typename U, int value>
class SuckType {
public:
    T a;
    U b;
    SuckType():a(value),b(value){}
};
template< typename U>
typedef SuckType<std::vector<int>, U, 1> NewType; // 不合法
```

C++11 使用 using 引入了下面这种形式的写法，并且同时支持对传统 typedef 相同的功效：

```
template <typename T>
using NewType = SuckType<int, T, 1>;    // 合法
```

默认模板参数

我们可能定义了一个加法函数：

```
template<typename T, typename U>
auto add(T x, U y) -> decltype(x+y) {
    return x+y
}
```

但在使用时发现，要使用 add，就必须每次都指定其模板参数的类型。

在 C++11 中提供了一种便利，可以指定模板的默认参数：

```
template<typename T = int, typename U = int>
auto add(T x, U y) -> decltype(x+y) {
    return x+y;
}
```

6. 构造函数

委托构造

C++11 引入了委托构造的概念，这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数，从而达到简化代码的目的：

```
class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
```

```
    }
    Base(int value) : Base() {  // 委托 Base() 构造函数
        value2 = 2;
    }
};
```

继承构造

在继承体系中，如果派生类想要使用基类的构造函数，需要在构造函数中显式声明。

假若基类拥有为数众多的不同版本的构造函数，这样，在派生类中得写很多对应的“透传”构造函数。如下：

```
struct A
{
  A(int i) {}
  A(double d,int i){}
  A(float f,int i,const char* c){}
  //...等等系列的构造函数版本
};
struct B:A
{
  B(int i):A(i){}
  B(double d,int i):A(d,i){}
  B(folat f,int i,const char* c):A(f,i,e){}
  //......等等好多个和基类构造函数对应的构造函数
};
```

C++11 的继承构造：

```
struct A
{
  A(int i) {}
  A(double d,int i){}
  A(float f,int i,const char* c){}
  //...等等系列的构造函数版本
};
struct B:A
{
  using A::A;
  //关于基类各构造函数的继承一句话搞定
  //......
};
```

如果一个继承构造函数不被相关的代码使用，编译器不会为之产生真正的函数代码，这样比透传基类各种构造函数更加节省目标代码空间。

7. Lambda 表达式

Lambda 表达式，实际上就是提供了一个类似**匿名函数**的特性，而匿名函数则是**在需要一个函数，但是又不想费力去命名一个函数的情况下**去使用的。

Lambda 表达式的基本语法如下：

```
[ caputrue ] ( params ) opt -> ret { body; };
```

1) capture 是捕获列表；

2) params 是参数表；(选填)

3) opt 是函数选项；可以填 mutable,exception,attribute（选填）

mutable 说明 lambda 表达式体内的代码可以修改被捕获的变量，并且可以访问被捕获的对象的 non-const 方法。

exception 说明 lambda 表达式是否抛出异常以及何种异常。

attribute 用来声明属性。
4) ret 是返回值类型（拖尾返回类型）。(选填)
5) body 是函数体。

捕获列表：lambda 表达式的捕获列表精细控制了 lambda 表达式能够访问的**外部变量**，以及如何访问这些变量。

1) []不捕获任何变量。
2) [&]捕获外部作用域中所有变量，并作为引用在函数体中使用（按引用捕获）。
3) [=]捕获外部作用域中所有变量，并作为副本在函数体中使用(按值捕获)。注意值捕获的前提是变量可以拷贝，且**被捕获的变量在 lambda 表达式被创建时拷贝，而非调用时才拷贝**。如果希望 lambda 表达式在调用时能即时访问外部变量，我们应当使用引用方式捕获。

```
int a = 0;
auto f = [=] { return a; };

a+=1;

cout << f() << endl;        //输出 0

int a = 0;
auto f = [&a] { return a; };

a+=1;

cout << f() <<endl;        //输出 1
```

4) [=,&foo]按值捕获外部作用域中所有变量，并按引用捕获 foo 变量。
5) [bar]按值捕获 bar 变量，同时不捕获其他变量。
6) [this]捕获当前类中的 this 指针，让 lambda 表达式拥有和当前类成员函数同样的访问权限。如果已经使用了&或者=，就默认添加此选项。**捕获 this 的目的是可以在 lamda 中使用当前类的成员函数和成员变量**。

```
class A
{
 public:
     int i_ = 0;

     void func(int x,int y){
         auto x1 = [] { return i_; };                    //error,没有捕获外部变量
         auto x2 = [=] { return i_ + x + y; };           //OK
         auto x3 = [&] { return i_ + x + y; };         //OK
         auto x4 = [this] { return i_; };                //OK
         auto x5 = [this] { return i_ + x + y; };        //error,没有捕获 x,y
         auto x6 = [this, x, y] { return i_ + x + y; };      //OK
         auto x7 = [this] { return i_++; };              //OK
};

int a=0 , b=1;
auto f1 = [] { return a; };                            //error,没有捕获外部变量
auto f2 = [&] { return a++ };                         //OK
auto f3 = [=] { return a; };                           //OK
auto f4 = [=] {return a++; };                         //error,a 是以复制方式捕获的，无法修改
auto f5 = [a] { return a+b; };                         //error,没有捕获变量 b
```

```
auto f6 = [a, &b] { return a + (b++); };                //OK
auto f7 = [=, &b] { return a + (b++); };                //OK
```

注意 f4，虽然按值捕获的变量值均复制一份存储在 lambda 表达式变量中，修改他们也并不会真正影响到外部，但我们却仍然无法修改它们。**如果希望去修改按值捕获的外部变量，需要显示指明 lambda 表达式为 mutable。被 mutable 修饰的 lambda 表达式就算没有参数也要写明参数列表**。

原因：lambda 表达式可以说是就地定义仿函数闭包的“语法糖”。它的捕获列表捕获住的任何外部变量，**最终会变为闭包类型的成员变量**。按照 C++标准，lambda 表达式的 **operator()默认是 const 的**，一个 const 成员函数是无法修改成员变量的值的。而 mutable 的作用，就在于取消 operator()的 const。

```
int a = 0;
auto f1 = [=] { return a++; };                  //error
auto f2 = [=] () mutable { return a++; };       //OK
```

lambda 表达式的大致原理：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**，是一个右值。所以，我们上面的 lambda 表达式的结果就是一个个闭包。**对于复制传值捕捉方式，类中会相应添加对应类型的非静态数据成员**。在运行时，会用复制的值初始化这些成员变量，从而生成闭包。对于引用捕获方式，无论是否标记 mutable，都可以在 lambda 表达式中修改捕获的值。至于闭包类中是否有对应成员，C++标准中给出的答案是：不清楚的，与具体实现有关。

lambda 表达式是不能被赋值的：

```
auto a = [] { cout << "A" << endl; };
auto b = [] { cout << "B" << endl; };

a = b;   // 非法，lambda 无法赋值
auto c = a;   // 合法，生成一个副本
```

闭包类型禁用了赋值操作符，但是没有禁用复制构造函数，所以你仍然可以用一个 lambda 表达式去初始化另外一个 lambda 表达式而产生副本。

在多种捕获方式中，**最好不要使用[=]和[&]默认捕获所有变量**。

默认引用捕获所有变量，你有很大可能会出现悬挂引用 (Dangling references)，因为引用捕获不会延长引用的变量的生命周期：

```
std::function<int(int)> add_x(int x)
{
    return [&](int a) { return x + a; };
}
```

上面函数返回了一个 lambda 表达式，参数 x 仅是一个临时变量，函数 add_x 调用后就被销毁了，但是返回的 lambda 表达式却引用了该变量，当调用这个表达式时，引用的是一个垃圾值，会产生没有意义的结果。上面这种情况，使用默认传值方式可以避免悬挂引用问题。

但是采用默认值捕获所有变量仍然有风险，看下面的例子：

```
class Filter
{
public:
    Filter(int divisorVal):
        divisor{divisorVal}
    {}

    std::function<bool(int)> getFilter()
    {
```

```
        return [=](int value) {return value % divisor == 0; };
    }

private:
    int divisor;
};
```

这个类中有一个成员方法，可以返回一个 lambda 表达式，这个表达式使用了类的数据成员 divisor。而且采用默认值方式捕捉所有变量。你可能认为这个 lambda 表达式也捕捉了 divisor 的一份副本，但是实际上并没有。**因为数据成员 divisor 对 lambda 表达式并不可见**，你可以用下面的代码验证：

```
// 类的方法，下面无法编译，因为 divisor 并不在 lambda 捕捉的范围
std::function<bool(int)> getFilter()
{
    return [divisor](int value) {return value % divisor == 0; };
}
```

原代码中，lambda 表达式实际上捕捉的是 this 指针的副本，所以原来的代码等价于：

```
std::function<bool(int)> getFilter()
{
    return [this](int value) {return value % this->divisor == 0; };
}
```

尽管还是以值方式捕获，但是捕获的是指针，其实相当于以引用的方式捕获了当前类对象，所以 lambda 表达式的闭包与一个类对象绑定在一起了，这很危险，因为你仍然有可能在类对象析构后使用这个 lambda 表达式，那么类似“悬挂引用”的问题也会产生。所以，采用默认值捕捉所有变量仍然是不安全的，主要是由于指针变量的复制，实际上还是按引用传值。

lambda 表达式可以赋值给对应类型的函数指针。但是使用函数指针并不是那么方便。所以 STL 定义在<functional>头文件提供了一个多态的函数对象封装 std::function，其类似于函数指针。它可以绑定任何类函数对象，只要参数与返回类型相同。如下面的返回一个 bool 且接收两个 int 的函数包装器：

```
std::function<bool(int, int)> wrapper = [](int x, int y) { return x < y; };
```

lambda 表达式一个更重要的应用是其可以用于函数的参数，通过这种方式可以实现回调函数。

最常用的是在 STL 算法中，比如你要统计一个数组中满足特定条件的元素数量，通过 lambda 表达式给出条件，传递给 count_if 函数：

```
int value = 3;
vector<int> v {1, 3, 5, 2, 6, 10};
int count = std::count_if(v.beigin(), v.end(), [value](int x) { return x > value; });
```

再比如你想生成斐波那契数列，然后保存在数组中，此时你可以使用 generate 函数，并辅助 lambda 表达式：

```
vector<int> v(10);
int a = 0;
int b = 1;
std::generate(v.begin(), v.end(), [&a, &b] { int value = b; b = b + a; a = value; return value; });
// 此时 v {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

当需要遍历容器并对每个元素进行操作时：

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each(v.begin(), v.end(), [&even_count](int val){
    if(!(val & 1)){
        ++ even_count;
    }
});
```

```
std::cout << "The number of even is " << even_count << std::endl;
```

大部分 STL 算法，可以非常灵活地搭配 lambda 表达式来实现想要的效果。

8. 新增容器

std::array

std::array 保存在栈内存中，相比堆内存中的 std::vector，我们能够灵活的访问这里面的元素，从而获得更高的性能。

std::array 会在编译时创建一个固定大小的数组，std::array 不能够被隐式的转换成指针，使用 std::array 只需指定其类型和大小即可：

```
std::array<int, 4> arr= {1,2,3,4};

int len = 4;
std::array<int, len> arr = {1,2,3,4}; // 非法, 数组大小参数必须是常量表达式
```

当我们开始用上了 std::array 时，难免会遇到要将其兼容 C 风格的接口，这里有三种做法：

```
void foo(int *p, int len) {
    return;
}

std::array<int 4> arr = {1,2,3,4};

// C 风格接口传参
// foo(arr, arr.size());           // 非法, 无法隐式转换
foo(&arr[0], arr.size());
foo(arr.data(), arr.size());

// 使用 `std::sort`
std::sort(arr.begin(), arr.end());
```

std::forward_list

std::forward_list 是一个列表容器，使用方法和 std::list 基本类似。

和 std::list 的双向链表的实现不同，std::forward_list 使用单向链表进行实现，提供了 O(1) 复杂度的元素插入，不支持快速随机访问（这也是链表的特点），也是标准库容器中唯一一个不提供 size() 方法的容器。当不需要双向迭代时，具有比 std::list 更高的空间利用率。

无序容器

C++11 引入了两组无序容器：

std::unordered_map/std::unordered_multimap 和 std::unordered_set/std::unordered_multiset。

无序容器中的元素是不进行排序的，内部通过 Hash 表实现，插入和搜索元素的平均复杂度为 O(constant)。

元组 std::tuple

元组的使用有三个核心的函数：

std::make_tuple: 构造元组

std::get: 获得元组某个位置的值

std::tie: 元组拆包

```
#include <tuple>
#include <iostream>

auto get_student(int id)
{
    // 返回类型被推断为 std::tuple<double, char, std::string>
    if (id == 0)
        return std::make_tuple(3.8, 'A', "张三");
    if (id == 1)
```

```
        return std::make_tuple(2.9, 'C', "李四");
    if (id == 2)
        return std::make_tuple(1.7, 'D', "王五");
    return std::make_tuple(0.0, 'D', "null");
    // 如果只写 0 会出现推断错误, 编译失败
}

int main()
{
    auto student = get_student(0);
    std::cout << "ID: 0, "
    << "GPA: " << std::get<0>(student) << ", "
    << "成绩: " << std::get<1>(student) << ", "
    << "姓名: " << std::get<2>(student) << '\n';

    double gpa;
    char grade;
    std::string name;

    // 元组进行拆包
    std::tie(gpa, grade, name) = get_student(1);
    std::cout << "ID: 1, "
    << "GPA: " << gpa << ", "
    << "成绩: " << grade << ", "
    << "姓名: " << name << '\n';
}
```

合并两个元组，可以通过 std::tuple_cat 来实现。

```
auto new_tuple = std::tuple_cat(get_student(1), std::move(t));
```

9. 正则表达式

正则表达式描述了一种字符串匹配的模式。一般使用正则表达式主要是实现下面三个需求：

1) 检查一个串是否包含某种形式的子串；
2) 将匹配的子串替换；
3) 从某个串中取出符合条件的子串。

C++11 提供的正则表达式库操作 std::string 对象，对模式 std::regex（本质是 std::basic_regex）进行初始化，通过 std::regex_match 进行匹配，从而产生 std::smatch（本质是 std::match_results 对象）。

我们通过一个简单的例子来简单介绍这个库的使用。考虑下面的正则表达式：

[a-z]+.txt: 在这个正则表达式中, [a-z] 表示匹配一个小写字母, + 可以使前面的表达式匹配多次，因此 [a-z]+ 能够匹配一个及以上小写字母组成的字符串。在正则表达式中一个 . 表示匹配任意字符，而 . 转义后则表示匹配字符 . ，最后的 txt 表示严格匹配 txt 这三个字母。因此这个正则表达式的所要匹配的内容就是文件名为纯小写字母的文本文件。

std::regex_match 用于匹配字符串和正则表达式，有很多不同的重载形式。最简单的一个形式就是传入 std::string 以及一个 std::regex 进行匹配，当匹配成功时，会返回 true，否则返回 false。例如：

```
#include <iostream>
#include <string>
#include <regex>

int main() {
    std::string fnames[] = {"foo.txt", "bar.txt", "test", "a0.txt", "AAA.txt"};
    // 在 C++ 中 `\` 会被作为字符串内的转义符，为使 `\.` 作为正则表达式传递进去生效，需要
```

对 `\` 进行二次转义，从而有 `\\.`

```
    std::regex txt_regex("[a-z]+\\.txt");
    for (const auto &fname: fnames)
        std::cout << fname << ": " << std::regex_match(fname, txt_regex) << std::endl;
}
```

另一种常用的形式就是依次传入 std::string/std::smatch/std::regex 三个参数，其中 std::smatch 的本质其实是 std::match_results，在标准库中， std::smatch 被定义为了 std::match_results，也就是一个子串迭代器类型的 match_results。使用 std::smatch 可以方便的对匹配的结果进行获取，例如：

```
std::regex base_regex("([a-z]+)\\.txt");
std::smatch base_match;
for(const auto &fname: fnames) {
    if (std::regex_match(fname, base_match, base_regex)) {
        // sub_match 的第一个元素匹配整个字符串
        // sub_match 的第二个元素匹配了第一个括号表达式
        if (base_match.size() == 2) {
            std::string base = base_match[1].str();
            std::cout << "sub-match[0]: " << base_match[0].str() << std::endl;
            std::cout << fname << " sub-match[1]: " << base << std::endl;
        }
    }
}
```

以上两个代码段的输出结果为：

```
foo.txt: 1
bar.txt: 1
test: 0
a0.txt: 0
AAA.txt: 0
sub-match[0]: foo.txt
foo.txt sub-match[1]: foo
sub-match[0]: bar.txt
bar.txt sub-match[1]: bar
```

10. 语言级线程支持

std::thread
std::mutex/std::unique_lock
std::future/std::packaged_task
std::condition_variable

代码编译需要使用 -pthread 选项

11. 右值引用和 move 语义

先看一个简单的例子直观感受下：

```
string a(x);                                // line 1
string b(x + y);                            // line 2
string c(some_function_returning_a_string()); // line 3
```

如果使用以下拷贝构造函数：

```
string(const string& that)
{
    size_t size = strlen(that.data) + 1;
    data = new char[size];
    memcpy(data, that.data, size);
}
```

以上 3 行中，只有第一行(line 1)的 x 深度拷贝是有必要的，因为我们可能会在后边用到 x，x 是一个左值(lvalues)。

第二行和第三行的参数则是右值,因为表达式产生的 string 对象是匿名对象,之后没有办法再使用了。

C++ 11 引入了一种新的机制叫做“**右值引用**”，以便我们通过重载直接使用右值参数。我们所要做的就是写一个以右值引用为参数的构造函数：

```
string(string&& that)   // string&& is an rvalue reference to a string
{
data = that.data;
that.data = 0;
}
```

我们**没有深度拷贝堆内存中的数据，而是仅仅复制了指针，并把源对象的指针置空**。事实上，我们“偷取”了属于源对象的内存数据。**由于源对象是一个右值，不会再被使用，因此客户并不会觉察到源对象被改变了**。在这里，我们并没有真正的复制，所以我们把这个构造函数叫做“转移构造函数”（move constructor），他的工作就是把资源从一个对象转移到另一个对象，而不是复制他们。

有了右值引用，再来看看赋值操作符：

```
string& operator=(string that)
{
std::swap(data, that.data);
return *this;
}
```

注意到我们是直接对参数 that 传值，所以 that 会像其他任何对象一样被初始化，那么确切的说，that 是怎样被初始化的呢？对于 C++ 98，答案是复制构造函数，但是**对于 C++11，编译器会依据参数是左值还是右值在复制构造函数和转移构造函数间进行选择**。

如果是 a=b，这样就会调用复制构造函数来初始化 that（因为 b 是左值），赋值操作符会与新创建的对象交换数据，深度拷贝。这就是 copy and swap 惯用法的定义：构造一个副本，与副本交换数据，并让副本在作用域内自动销毁。这里也一样。

如果是 a = x + y，这样就会调用转移构造函数来初始化 that（**因为 x+y 是右值**），所以这里没有深度拷贝，只有高效的数据转移。相对于参数，that 依然是一个独立的对象，但是他的构造函数是无用的（trivial），因此堆中的数据没有必要复制，而仅仅是转移。没有必要复制他，因为 x+y 是右值，再次，从右值指向的对象中转移是没有问题的。

总结一下：**复制构造函数执行的是深度拷贝，因为源对象本身必须不能被改变。而转移构造函数却可以复制指针，把源对象的指针置空，这种形式下，这是安全的，因为用户不可能再使用这个对象了**。

下面我们进一步讨论右值引用和 move 语义。

C++98 标准库中提供了一种唯一拥有性的智能指针 std::auto_ptr，该类型在 C++11 中已被废弃，因为其“复制”行为是危险的。

```
auto_ptr<Shape> a(new Triangle);
auto_ptr<Shape> b(a);
```

注意 b 是怎样使用 a 进行初始化的，它不复制 triangle，而是把 triangle 的所有权从 a 传递给了 b，也可以说成“a 被转移进了 b”或者“triangle 被从 a 转移到了 b”。

auto_ptr 的复制构造函数可能看起来像这样（简化）：

```
auto_ptr(auto_ptr& source)   // note the missing const
{
p = source.p;
source.p = 0;   // now the source no longer owns the object
}
```

auto_ptr 的**危险之处在于看上去应该是复制，但实际上确是转移**。调用被转移过的

auto_ptr 的成员函数将会导致不可预知的后果。所以你必须非常谨慎的使用 auto_ptr ，如果他被转移过。

```
auto_ptr<Shape> make_triangle()
{
    return auto_ptr<Shape>(new Triangle);
}

auto_ptr<Shape> c(make_triangle());      // move temporary into c
double area = make_triangle()->area();   // perfectly safe

auto_ptr<Shape> a(new Triangle);    // create triangle
auto_ptr<Shape> b(a);               // move a into b
double area = a->area();                // undefined behavior
```

显然，在持有 auto_ptr 对象的 a 表达式和持有调用函数返回的 auto_ptr 值类型的 make_triangle()表达式之间一定有一些潜在的区别，每调用一次后者就会创建一个新的 auto_ptr 对象。这里 a 其实就是一个左值（lvalue）的例子，而 make_triangle()就是右值（rvalue）的例子。

转移像 a 这样的左值是非常危险的，因为我们可能调用 a 的成员函数，这会**导致不可预知的行为**。另一方面，转移像 make_triangle()这样的右值却是非常安全的，因为复制构造函数之后，我们不能再使用这个临时对象了，因为这个转移后的临时对象会在下一行之前销毁掉。

我们现在知道**转移左值是十分危险的，但是转移右值却是很安全的**。如果 C++能从语言级别支持区分左值和右值参数，我就可以完全杜绝对左值转移，或者把转移左值在调用的时候暴露出来，以使我们不会不经意的转移左值。

C++ 11 对这个问题的答案是右值引用。**右值引用是针对右值的新的引用类型**，语法是 X&&。以前的老的引用类型 X& 现在被称作左值引用。

使用右值引用 X&&作为参数的最有用的函数之一就是转移构造函数 X::X(X&& source)，它的主要作用是把源对象的本地资源转移给当前对象。

C++ 11 中，std::auto_ptr< T >已经被 std::unique_ptr< T >所取代，后者就是利用的右值引用。

其转移构造函数：

```
unique_ptr(unique_ptr&& source)   // note the rvalue reference
{
    ptr = source.ptr;
    source.ptr = nullptr;
}
```

这个转移构造函数跟 auto_ptr 中复制构造函数做的事情一样，但是它却只能接受右值作为参数。

```
unique_ptr<Shape> a(new Triangle);
unique_ptr<Shape> b(a);                  // error
unique_ptr<Shape> c(make_triangle());       // okay
```

第二行不能编译通过，因为 a 是左值，但是参数 unique_ptr&& source 只能接受右值，这正是我们所需要的，杜绝危险的隐式转移。第三行编译没有问题，因为 make_triangle()是右值，转移构造函数会将临时对象的所有权转移给对象 c，这正是我们需要的。

34：右值引用和 move 语义？

转移左值

有时候，我们可能想转移左值，也就是说，有时候我们想让编译器把左值当作右值对待，以便能使用转移构造函数，即便这有点不安全。出于这个目的，C++ 11 在标准库的头文件<utility>中提供了一个模板函数 std::move。实际上，**std::move 仅仅是简单地将左值转换为右值，它本身并没有转移任何东西。它仅仅是让对象可以转移。**

以下是如何正确的转移左值：

```
unique_ptr<Shape> a(new Triangle);
```

```
unique_ptr<Shape> b(a);              // still an error
unique_ptr<Shape> c(std::move(a));   // okay
```

请注意，第三行之后，a 不再拥有 Triangle 对象。不过这没有关系，因为**通过明确的写出 std::move(a)，我们很清楚我们的意图**：亲爱的转移构造函数，你可以对 a 做任何想要做的事情来初始化 c；**我不再需要 a 了**，对于 a，您请自便。

当然，如果你在使用了 mova(a)之后，还继续使用 a，那无疑是搬起石头砸自己的脚，还是会导致严重的运行错误。

总之，std::move(some_lvalue)将左值转换为右值（**可以理解为一种类型转换**），使接下来的转移成为可能。

一个例子：

```
class Foo
{
    unique_ptr<Shape> member;

public:

    Foo(unique_ptr<Shape>&& parameter)
    : member(parameter)    // error
    {}
};
```

上面的 parameter，其类型是一个右值引用，只能说明 parameter 是指向右值的引用，而**parameter 本身是个左值**。（Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: if it has a name, then it is an lvalue. Otherwise, it is an rvalue.）

因此以上对 parameter 的转移是不允许的，需要使用 std::move 来显示转换成右值。

35：STL 里 resize 和 reserve 的区别？

首先，两个函数的功能是有区别的：

reserve 是容器预留空间，但并不真正创建元素对象，在创建对象之前，不能引用容器内的元素，因此当加入新的元素时，需要用 push_back()/insert()函数。

resize 是改变容器的大小，并且创建对象，因此，调用这个函数之后，就可以引用容器内的对象了，因此当加入新的元素时，用 operator[]操作符，或者用迭代器来引用元素对象。

其次，两个函数的形式是有区别的：

reserve 函数之后一个参数，即需要预留的容器的空间；

resize 函数可以有两个参数，第一个参数是容器新的大小，第二个参数是要加入容器中的新元素，如果这个参数被省略，那么就调用元素对象的默认构造函数。

初次接触这两个接口也许会混淆，其实接口的命名就是对功能的绝佳描述，resize 就是重新分配大小，reserve 就是预留一定的空间。这两个接口即存在差别，也有共同点。下面就它们的细节进行分析。

为实现 resize 的语义，resize 接口做了两个保证：

一是保证区间[0, new_size)范围内数据有效，如果下标 index 在此区间内，vector[indext]是合法的。

二是保证区间[0, new_size)范围以外数据无效，如果下标 index 在区间外，vector[indext]是非法的。

reserve 只是保证 vector 的空间大小(capacity)最少达到它的参数所指定的大小 n。在区间[0, n)范围内，如果下标是 index，vector[index]这种访问有可能是合法的，也有可能是非法的，视具体情况而定。

resize 和 reserve 接口的共同点是它们都保证了 vector 的空间大小(capacity)最少达到它的参数所指定的大小。

因两接口的源代码相当精简，以至于可以在这里贴上它们：

```
void resize(size_type new_size) { resize(new_size, T()); }
```

```
   void resize(size_type new_size, const T& x) {
       if (new_size < size())
           erase(begin() + new_size, end()); // erase 区间范围以外的数据，确保区间以外的数据无效
       else
           insert(end(), new_size - size(), x); // 填补区间范围内空缺的数据，确保区间内的数据有效
vector<int> myVec;
myVec.reserve( 100 );      // 新元素还没有构造,
                           // 此时不能用[]访问元素
for (int i = 0; i < 100; i++ )
...{
     myVec.push_back( i ); //新元素这时才构造
}
myVec.resize( 102 );       // 用元素的默认构造函数构造了两个新的元素
myVec[100] = 1;            //直接操作新元素
myVec[101] = 2;
```

36：vector 和 deque 的区别？

vector 概述

vector 的数据安排以及操作方式,与 array 非常相似。两者的唯一差别在于空间的运用的灵活性。array 是静态空间，一旦配置了就不能改变；要换个大(或小)一点的房子，可以，一切琐细得由客户端自己来：首先配置一块新空间,然后将元素从旧址一一搬往新址，再把原来的空间释还给系统。vector 是动态空间，随着元素的加入，它的内部机制会自行扩充空间以容纳新元素。因此,，vector 的运用对于内存的合理利用与运用的灵活性有很大的帮助，我们再也不必因为害怕空间不足而，一开始就要求一个大块头 array 了，我们可以安心使用 vector，吃多少用多少。

vector 的数据结构

vector 所采用的数据结构非常简单：线性线性空间。它以两个迭代器 start 和 finish 分别指向配置得来的连续空间中目前已经被使用的范围，并以迭代器 end_of_storage 指向整块连续空间的尾端：

```
template<class T, class Alloc = alloc>
class vector {
...
protected:
    iterator start;              //表示目前使用空间的头
    iterator finish;             //表示目前使用空间的尾
    iterator end_of_storage;     //表示目前可用空间的尾
};
```

为了降低空间配置时的速度成本，vector 实际配置的大小可能比客户端需求更大一些，以备将来可能的扩充。

vector 的构造与内存管理

当我们以 push_back()将新元素插入 vector 尾端时，该函数首先检查是否还有备用空间，如果有就直接在备用空间上构造元素，并调整迭代器 finish，使 vector 变大。如果没有备用空间了，就扩充空间（重新配置、移动数据、释放原空间）。

注意：所谓动态增加大小，并不是在原空间之后接续新空间，而是以原大小的两倍另外配置一块较大空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。因此，对 vector 的任何操作，一旦引起空间的重新配置，指向原 vector 的所有迭代器就都失效了。

2、deque

deque 概述

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。

deque 的中控器

deque 系由一段一段的定量连续空间构成。一旦有必要在 deque 的前端或尾端增加新空间，便配置一段定量连续空间，串接在整个 deque 的头端或尾端。deque 的最大任务，便是在这些分段的定量连续空间上，维护其整体连续的假象，并提供随机存取的接口。避开了“重新配置、复制、释放”的轮回，代价则是复杂的迭代器架构。

deque 采用一块所谓的 map(注意,不是 STL 的 map 容器)作为主控。这里所谓 map 是一小块连续空间 ，其中每个元素(此处称为一个节点，node)都是指针，指向另一段(较大的) 连续线性空间，称为缓冲区。缓冲区才是 deque 的存储空间主体。

```
template <class T, class Allod  alloc, size_t Bufsiz = 0>
class deque {
public:
    typedef T value_type;
    typedef value_type* pointer;
    ...
protected:
    typedef pointer* map pointer;

protected:
    map_pointer map;//指向 map，map 是块连续空间，其内的每个元素都是一个指针，指向一块缓冲区
    size_type map_size;//map 可容纳多少指针
...
}
```

deque 的构造与内存管理

如果申请的 map 空间不够时，也需要重新配置更大的空间，将原来 map 里的指针拷贝过来，最后释放原来的空间。

3、vector 和 deque 的区别

vector 是单向开口的连续线性空间， dequeu 是一种双向开口的连续线性定 ；

deque 允许于常数时间内对起头端进行元素的插入或移除操作 ；

deque 没有所谓容量(capacity)观念，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

4、参考资料

侯捷著《STL 源码剖析》

37：不同排序算法的比较？

一、插入排序

1、直接插入排序

```
void InsertSort(vector<int> &vec)
{
    int length = vec.size();
    if (length < 2)
        return;
    for (int i = 1; i < length; ++i)
    {
        if (vec[i - 1] > vec[i])
        {
            int temp = vec[i];
            int j = i - 1;
            for (; j > -1 && temp < vec[j]; --j)
            {
                vec[j + 1] = vec[j];
            }
            vec[j + 1] = temp;
        }
    }
}
```

2、希尔排序

```
void ShellSort(vector<int> &nums)
{
    int len = nums.size();
    if (len < 2)
        return;
    for (int dk = len / 2; dk > 0; dk /= 2)
    {
        for (int i = dk; i < len; ++i)
        {
            if (nums[i] < nums[i - dk])
            {
                int temp = nums[i];
                int j = i - dk;
                for (; j > -1 && nums[j] > temp; j -= dk)
                    nums[j + dk] = nums[j];
                nums[j + dk] = temp;
            }
        }
    }
}
```

二、交换排序

1、冒泡排序

```
void BubbleSort(vector<int> &vec)
{
    int length = vec.size();
    if (length < 2)
    {
        return;
    }
    for (int i = 0; i < length; ++i)
    {
        bool flag = false;
        for (int j = length - 1; j > i; --j)
        {
            if (vec[j - 1] > vec[j])
            {
                swap(vec[j - 1], vec[j]);
                flag = true;
            }
        }
        if (flag == false)
            return;
    }
}
```

2、快速排序

```
int Partition(vector<int> &vec, int low, int high)
{
    int pivot = vec[low];
    while (low < high)
    {
        while (low < high && vec[high] >= pivot) --high;
        vec[low] = vec[high];
        while (low < high && vec[low] <= pivot) ++low;
        vec[high] = vec[low];
    }
    vec[low] = pivot;
    return low;
}
void QuickSort(vector<int> &vec, int low, int high)
{
    if (low < high)
    {
        int pivotpos = Partition(vec, low, high);
        QuickSort(vec, low, pivotpos - 1);
        QuickSort(vec, pivotpos + 1, high);
    }
}
```

三、选择排序

1、简单选择排序

```
void SelectSort(vector<int> &vec)
{
    int length = vec.size();
    if (length < 2)
        return;
    for (int i = 0; i < length; ++i)
    {
        int min = vec[i];
        int index = i;
        for (int j = i + 1; j < length; ++j)
        {
            if (vec[j] < min)
            {
                min = vec[j];
                index = j;
            }
        }
        if (index != i)
        {
            swap(vec[i], vec[index]);
        }
    }
}
```

2、堆排序

```
void AdjustDown(vector<int> &vec, int k, int len){
    int cur = vec[k];
    for (int i = 2 * k + 1; i <= len; i = 2 * i + 1){
        if (i < len && vec[i] < vec[i + 1])
            ++i;
        if (cur >= vec[i])
            break;
        else{
            vec[k] = vec[i];
            k = i;
        }
    }
    vec[k] = cur;
}
void BuildMaxHeap(vector<int> &vec, int len){
    for (int i = len / 2; i > -1; --i){
        AdjustDown(vec, i, len);
    }
}
void HeapSort(vector<int> &vec){
    /*vec[0]存储元素*/
    int len = vec.size();
    if (len < 1)
        return;
    BuildMaxHeap(vec, len - 1);
    for (int i = len - 1; i > -1; --i){
        swap(vec[i], vec[0]);
        AdjustDown(vec, 0, i - 1);
    }
}
```

四、归并排序

1、二路归并排序

```
void Merge(vector<int> &nums, int low, int mid, int high)
{
    vector<int> vec = nums;
    int i = low;
    int j = mid + 1;
    int k = low;
    while (i <= mid && j <= high)
    {
        if (vec[i] <= vec[j])
            nums[k++] = vec[i++];
        else
            nums[k++] = vec[j++];
    }
    while (i <= mid)
        nums[k++] = vec[i++];
    while (j <= high)
        nums[k++] = vec[j++];
}
void MergeSort(vector<int> &nums, int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort(nums, low, mid);
        MergeSort(nums, mid + 1, high);
        Merge(nums, low, mid, high);
    }
}
```

五、不同排序算法比较

数组排序算法

算法	时间复杂度			空间复杂度
	最佳	平均	最差	最差
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

各种排序算法的性质					
算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	O(n)	O(n^2)	O(n^2)	O(1)	是
冒泡排序	O(n)	O(n^2)	O(n^2)	O(1)	是
简单选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	否
希尔排序	O(n)	O((nlogn)^2)	O((nlogn)^2)	O(1)	否
快速排序	O(nlogn)	O(nlogn)	O(n^2)	O(logn)	否
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	否
二路归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	是

38：大端和小端的区别，以及如何判断一台机器是大端还是小端？

```
int checkCPU()
{
    union w
    {
        int a;
        char b;
    }c;
    c.a = 1;
    return (c.b == 1);//小端返回 1，大端返回 0
}
```

采用 Little-endian 模式的 CPU 对操作数的存放方式是从低字节到高字节，而 Big-endian 模式对操作数的存放方式是从高字节到低字节。例如 32bit 的数 0x12345678 在 Little-endian 模式 CPU 内存中的存放方式(假设从地址 0x4000 开始存放)为：

```
0x4000    0x78
0x4001    0x56
0x4002    0x34
0x4003    0x12
```

而在 Big-endian 模式 CPU 内存中的存放方式为：

0x4000　　0x12
0x4001　　0x34
0x4002　　0x56
0x4003　　0x78

39：malloc 分配内存的原理？

malloc 的原理
步骤分为放置、分割和合并

在堆中，堆块由一个字的头部、有效载荷、填充以及一个字的脚部组成，空闲块是通过头部中的大小字段隐含地连接在一起形成一个隐式空闲链表，分配器可以通过遍历堆中所有的块，从而间接遍历整个空闲块的集合。

1、放置已分配的块

当一个应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大可以放置所请求块的空闲块，分配器执行这种搜索的方式是放置策略确定的。常见的策略是首次适配、下一次适配和最佳适配。

首次适配：从头开始搜索空闲链表，选择第一个适合的空闲块。
下一次适配：从上一次查询结束的地方开始搜索空闲链表，选择第一个适合的空闲块。
最佳适配：检查每个空闲块，选择适合所需请求大小的最小空闲块。

2、分割空闲块

一旦分配器找到一个匹配的空闲块，它就必须做另一个策略决定，那就是分配这个空块中多少空间。一个选择是用整个空闲块，另一个选择是将这个空闲块分割成两部分。
如果分配器不能为请求块找到合适的空闲块将发生什么呢？一个选择是通过合并那些在内存中物理上相邻的空闲块来创建一些更大的空闲块。然而，如果这样还是不能生成一个足够大的块，或者如果空闲块已经最大程度地合并了，那么分配器就会通过调用 sbrk 函数，向内核请求额外的堆内存，分配器将额外的内存转化成一个大的空闲块，将这个块插入到空闲链表中，然后将被请求的块放置在这个新的空闲块中。

3、合并空闲块

Knuth 提出了一种聪明而通用的技术，叫做边界标记，允许在常数时间内进行对前面的块合并。这种思想是，在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本，如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前面一个块的起始位置和状态，这个脚部总是在当前块开始位置一个字的距离。

参考资料：

《深入理解计算机系统》第 592 页

40：为什么构造函数不能声明为虚函数，析构函数可以，构造函数中为什么不能调用虚函数？

构造函数中为什么不能调用虚函数？

构造函数调用层次会导致一个有趣的两难选择。试想：如果我们在构造函数中并且调用了虚函数，那么会发生什么现象呢？在普通的成员函数中，我们可以想象所发生的情况——虚函数的调用是在运行时决定的。这是因为编译时这个对象并不能知道它是属于这个成员函数所在的那个类，还是属于由它派生出来的某个类。于是，我们也许会认为在构造函数中也会发生同样的事情。

然而，情况并非如此。对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。也就是说，虚机制在构造函数中不工作。

这种行为有两个理由：

第一个理由是概念上的。

在概念上，构造函数的工作是生成一个对象。在任何构造函数中，可能只是部分形成对象——我们只能知道基类已被初始化，但并不能知道哪个类是从这个基类继承来的。然而，虚函数在继承层次上是“向前”和“向外”进行调用。它可以调用在派生类中的函数。如果我们在构造函数中也这样做，那么我们所调用的函数可能操作还没有被初始化的成员，这将导致灾难发生。

第二个理由是机械上的。

当一个构造函数被调用时，它做的首要的事情之一就是初始化它的 VPTR。然而，它只能知道它属于“当前”类——即构造函数所在的类。于是它完全不知道这个对象是否是基于其它类。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（因为类不知道谁继承它）。所以它使用的 VPTR 必须是对于这个类的 VTABLE。而且，只要它是最后的构造函数调用，那

么在这个对象的生命期内，VPTR 将保持被初始化为指向这个 VTABLE。但如果接着还有一个更晚派生类的构造函数被调用，那么这个构造函数又将设置 VPTR 指向它的 VTABLE，以此类推，直到最后的构造函数结束。VRTP 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按照从基类到最晚派生类的顺序的另一个理由。

但是，当这一系列构造函数调用正发生时，每个构造函数都已经设置 VPTR 指向它自己的 VTABLE。如果函数调用使用虚机制，它将只产生通过它自己的 VTABLE 的调用，而不是最后派生的 VTABLE（所有构造函数被调用后才会有最后派生的 VTABLE）。另外，许多编译器认识到，如果在构造函数中进行虚函数调用，应该使用早绑定，因为它们知道晚绑定将只对本地函数产生调用。无论哪种情况，在构造函数中调用虚函数都不能得到预期的结果。

——来自《C++编程思想》合订本第 386 页

构造函数不能声明为虚函数，析构函数可以声明为虚函数，而且有时是必须声明为虚函数。不建议在构造函数和析构函数里面调用虚函数。

构造函数不能声明为虚函数的原因是:

1 构造一个对象的时候，必须知道对象的实际类型，而虚函数行为是在运行期间确定实际类型的。而在构造一个对象时，由于对象还未构造成功。编译器无法知道对象 的实际类型，是该类本身，还是该类的一个派生类，或是更深层次的派生类。无法确定。。。

2 虚函数的执行依赖于虚函数表。而虚函数表在构造函数中进行初始化工作，即初始化 vptr，让他指向正确的虚函数表。在构造对象期间，虚函数表还没有被初始化，将无法进行。

虚函数的意思就是开启动态绑定，程序会根据对象的动态类型来选择要调用的方法。然而在构造函数运行的时候，这个对象的动态类型还不完整，没有办法确定它到底是什么类型，故构造函数不能动态绑定。（动态绑定是根据对象的动态类型而不是函数名，在调用构造函数之前，这个对象根本就不存在，它怎么动态绑定？）

编译器在调用基类的构造函数的时候并不知道你要构造的是一个基类的对象还是一个派生类的对象。

析构函数设为虚函数的作用:

解释：在类的继承中，如果有基类指针指向派生类，那么用基类指针 delete 时，如果不定义成虚函数，派生类中派生的那部分无法析构。

例：

```
#include "stdafx.h"
#include "stdio.h"
class A
{
public:
    A();
    virtual~A();
};
A::A()
{
}
A::~A()
{
    printf("Delete class APn");
}
class B : public A
{
public:
    B();
    ~B();
};

B::B()
```

```
{ }

B::~B()
{
    printf("Delete class BPn");
}
int main(int argc, char* argv[])
{
    A* b = new B;
    delete b;
    return 0;
}
```

输出结果为：Delete class B
Delete class A

如果把 A 的 virtual 去掉：那就变成了 Delete class A 也就是说不会删除派生类里的剩余部分内容，也即不调用派生类的虚函数

因此在类的继承体系中，基类的析构函数不声明为虚函数容易造成内存泄漏。所以如果你设计一定类可能是基类的话，必须要声明其为虚函数。正如 Symbian 中的 CBase 一样。

Note：

1. 如果我们定义了一个构造函数，编译器就不会再为我们生成默认构造函数了。
2. 编译器生成的析构函数是非虚的，除非是一个子类，其父类有个虚析构，此时的函数虚特性来自父类。
3. 有虚函数的类，几乎可以确定要有个虚析构函数。
4. 如果一个类不可能是基类就不要申明析构函数为虚函数，虚函数是要耗费空间的。
5. 析构函数的异常退出会导致析构不完全，从而有内存泄露。最好是提供一个管理类，在管理类中提供一个方法来析构，调用者再根据这个方法的结果决定下一步的操作。
6. 在构造函数不要调用虚函数。在基类构造的时候，虚函数是非虚，不会走到派生类中，既是采用的静态绑定。显然的是：当我们构造一个子类的对象时，先调用基类的构造函数，构造子类中基类部分，子类还没有构造，还没有初始化，如果在基类的构造中调用虚函数，如果可以的话就是调用一个还没有被初始化的对象，那是很危险的，所以 C++中是不可以在构造父类对象部分的时候调用子类的虚函数实现。但是不是说你不可以那么写程序，你这么写，编译器也不会报错。只是你如果这么写的话编译器不会给你调用子类的实现，而是还是调用基类的实现。
7. 在析构函数中也不要调用虚函数。在析构的时候会首先调用子类的析构函数，析构掉对象中的子类部分，然后在调用基类的析构函数析构基类部分，如果在基类的析构函数里面调用虚函数，会导致其调用已经析构了的子类对象里面的函数，这是非常危险的。
8. 记得在写派生类的拷贝函数时，调用基类的拷贝函数拷贝基类的部分，不能忘记了。

41：stl 中 unordered_map 和 map 的区别 ？

标题中提到的四种容器，对于概念不清的人来说，经常容易弄混淆。这里我不去把库里面复杂的原码拿出剖析，这个如果有兴趣其实完全可以查 C++Reference,网上的原码是最权威和细致的了，而且我觉得有耐心直接认真看原码的人，也不需要我这篇速记博文了，所以我这里还是讲的通俗一些，把它们区分的七七八八。

一、hash_map 与 unordered_map

这两个的内部结构都是采用哈希表来实现。区别在哪里？unordered_map 在 C++11 的时候被引入标准库了，而 hash_map 没有，所以建议还是使用 unordered_map 比较好。

哈希表的好处是什么？查询平均时间是 O(1)。顾名思义，unordered，就是无序了，数据是按散列函数插入到槽里面去的，数据之间无顺序可言，但是有些时候我只要访问而不需要顺序，因此可以选择哈希表。举个最好的例子，就是我的 LeetCode 的博文里------Longest Consecutive Sequence 这一题，我的方法二就是用的 unordered_map 来实现“空间弥补时间”这样的做法。

二、unordered_map 与 map

虽然都是 map，但是内部结构大大的不同哎，map 的内部结构是 R-B-tree 来实现的，所以保证了一个

稳定的动态操作时间，查询、插入、删除都是 O（logn），最坏和平均都是。而 unordered_map 如前所述，是哈希表。顺便提一下，哈希表的查询时间虽然是 O（1），但是并不是 unordered_map 查询时间一定比 map 短，因为实际情况中还要考虑到数据量，而且 unordered_map 的 hash 函数的构造速度也没那么快，所以不能一概而论，应该具体情况具体分析。

三、unordered_map 与 unordered_set

后者就是在哈希表插入 value，而这个 value 就是它自己的 key，而不是像之前的 unordered_map 那样有键-值对，这里单纯就是为了方便查询这些值。我在 Longest Consecutive Sequence 这一题，我的方法一就是用了 unordered_set，同样是一个将空间弥补时间的方法。再举个大家好懂的例子，给你 A,B 两组数，由整数组成，然后把 B 中在 A 中出现的数字取出来，要求用线性时间完成。很明显，这道题应该将 A 的数放到一个表格，然后线性扫描 B，发现存在的就取出。可是 A 的范围太大，你不可能做一个包含所有整数的表，因为这个域太大了，所以我们就用 unordered_set 来存放 A 的数，具体实现库函数已经帮你搞定了，如果对于具体怎么去散列的同学又兴趣可以查看《算法导论》的第 11 章或者《数据结构与算法分析》的第五章，如果要看《算法导论》的同学我给个建议，完全散列你第一遍的时候可以暂时跳过不看，确实有点复杂。

42：C/C++中 extern 的用法 ？

在 C 语言中，修饰符 extern 用在变量或者函数的声明前，用来说明“此变量/函数是在别处定义的，要在此处引用”。

1：**extern 修饰变量的声明**。举例来说，如果文件 a.c 需要引用 b.c 中变量 int v，就可以在 a.c 中声明 extern int v，然后就可以引用变量 v。这里需要注意的是，被引用的变量 v 的链接属性必须是外链接（external）的，也就是说 a.c 要引用到 v，不只是取决于在 a.c 中声明 extern int v，还取决于变量 v 本身是能够被引用到的。这涉及到 c 语言的另外一个话题——变量的作用域。能够被其他模块以 extern 修饰符引用到的变量通常是全局变量。还有很重要的一点是，extern int v 可以放在 a.c 中的任何地方，比如你可以在 a.c 中的函数 fun 定义的开头处声明 extern int v，然后就可以引用到变量 v 了，只不过这样只能在函数 fun 作用域中引用 v 罢了，这还是变量作用域的问题。对于这一点来说，很多人使用的时候都心存顾虑。好像 extern 声明只能用于文件作用域似的。

2：**extern 修饰函数声明**。从本质上来讲，变量和函数没有区别。函数名是指向函数二进制块开头处的指针。如果文件 a.c 需要引用 b.c 中的函数，比如在 b.c 中原型是 int fun(int mu)，那么就可以在 a.c 中声明 extern int fun（int mu），然后就能使用 fun 来做任何事情。就像变量的声明一样，extern int fun（int mu）可以放在 a.c 中任何地方，而不一定非要放在 a.c 的文件作用域的范围中。对其他模块中函数的引用，最常用的方法是包含这些函数声明的头文件。

使用 extern 和包含头文件来引用函数有什么区别呢？

extern 的引用方式比包含头文件要简洁得多！extern 的使用方法是直接了当的，想引用哪个函数就用 extern 声明哪个函数。这大概是 KISS 原则的一种体现吧！这样做的一个明显的好处是，会加速程序的编译（确切的说是预处理）的过程，节省时间。在大型 C 程序编译过程中，这种差异是非常明显的。

3：**此外，extern 修饰符可用于指示 C 或者 C++函数的调用规范**。比如在 C++中调用 C 库函数，就需要在 C++程序中用 extern “C”声明要引用的函数。这是给链接器用的，告诉链接器在链接的时候用 C 函数规范来链接。主要原因是 C++和 C 程序编译完成后在目标代码中命名规则不同。

43：I/O 模型

1 概念说明

在进行解释之前，首先要说明几个概念：

用户空间和内核空间

进程切换

进程的阻塞

文件描述符
缓存 IO

1.1 用户空间与内核空间

现在操作系统都是采用虚拟存储器，那么对 32 位操作系统而言，它的寻址空间（虚拟存储空间）为 4G（2 的 32 次方）。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操作系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。**针对 linux 操作系统而言**，将最高的 1G 字节（**从虚拟地址 0xC0000000 到 0xFFFFFFFF），供内核使用，称为内核空间**，而将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF），供各个进程使用，称为用户空间。

1.2 进程切换

为了控制进程的执行，内核必须有能力挂起正在 CPU 上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换。因此可以说，任何进程都是在操作系统内核的支持下运行的，是与内核紧密相关的。

从一个进程的运行转到另一个进程上运行，这个过程中经过下面这些变化：

保存处理机上下文，包括程序计数器和其他寄存器。
更新 PCB 信息。
把进程的 PCB 移入相应的队列，如就绪、在某事件阻塞等队列。
选择另一个进程执行，并更新其 PCB。
更新内存管理的数据结构。
恢复处理机上下文。

注：**总而言之就是很耗资源**，具体的可以参考这篇文章：进程切换。

1.3 进程的阻塞

正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态。可见，**进程的阻塞是进程自身的一种主动行为**，也因此只有处于运行态的进程（获得 CPU），才可能将其转为阻塞状态。当进程**进入阻塞状态，是不占用 CPU 资源的**。

1.4 文件描述符 fd

文件描述符（File descriptor）是计算机科学中的一个术语，是一个用于表述指向文件的引用的抽象化概念。

文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。但是文件描述符这一概念往往只适用于 UNIX、Linux 这样的操作系统。

1.5 缓存 IO

缓存 I/O 又被称作标准 I/O，大多数文件系统的默认 I/O 操作都是缓存 I/O。在 Linux 的缓存 I/O 机制中，操作系统会将 I/O 的数据缓存在文件系统的页缓存（ page cache ）中，也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。

缓存 I/O 的缺点：

数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的。

2 Linux I/O 模型

网络 I/O 的本质是 socket 的读取,socket 在 linux 系统中被抽象为流,**I/O 可以理解为对流的操作**.对于一次 IO 访问,数据会先被拷到操作系统内核的缓冲区中,然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间,所以说**当一个 read 操作发生时,它会经理两个阶段:**

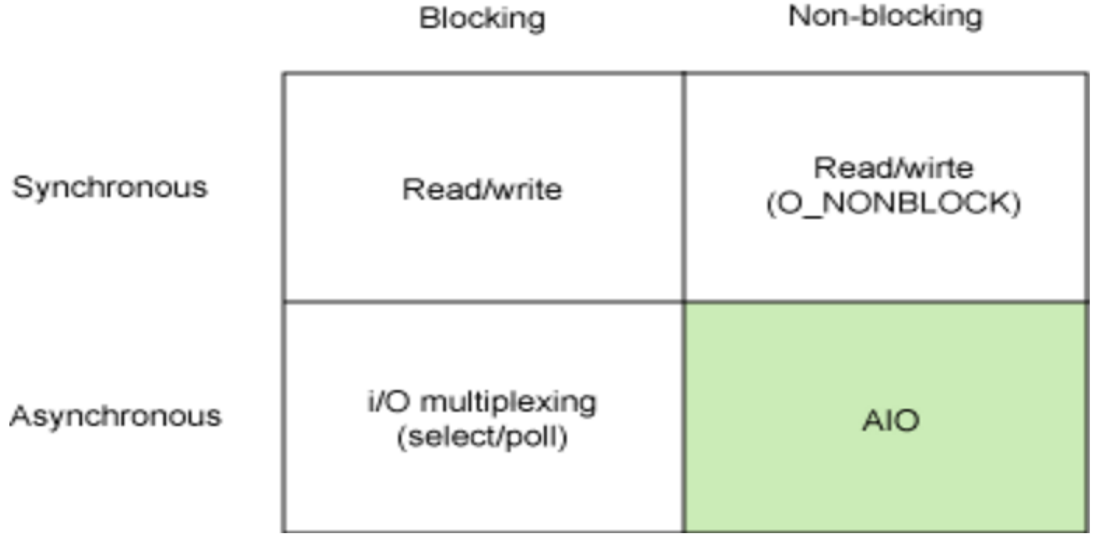
第一阶段：等待数据准备 (Waiting for the data to be ready)。
第二阶段：将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)。

对 socket 流而言:

第一步：通常涉及等待网络上的数据分组到达，然后被复制到内核的某个缓冲区。
第二步：把数据从内核缓冲区复制到应用进程缓冲区。

网络应用需要处理的无非就是两大类问题，**网络 I/O，数据计算**。相对于后者，网络 I/O 的延迟，给应用带来的性能瓶颈大于后者。网络 IO 的模型大致有如下几种：

异步 I/O（asynchronous I/O）
同步 I/O 模型（synchronous I/O）
阻塞 I/O（bloking I/O）
非阻塞 I/O（non-blocking I/O）
I/O 复用（multiplexing I/O）
信号驱动式 I/O（signal-driven I/O）
注：由于信号驱动式(signal driven IO)在实际中并不常用，所以我这只提及剩下的四种 IO Model。
在深入介绍 Linux IO 各种模型之前，让我们先来探索一下基本 Linux IO 模型的简单矩阵。如下图所示：



每个 IO 模型都有自己的使用模式，它们对于特定的应用程序都有自己的优点。本节将简要对其一一进行介绍。常见的 IO 模型有阻塞、非阻塞、IO 多路复用，异步。以一个生动形象的例子来说明这四个概念。
周末我和女友去逛街，中午饿了，我们准备去吃饭。周末人多，吃饭需要排队，我和女友有以下几种方案。

2.1 阻塞 I/O（blocking I/O）

2.1.1 场景描述

我和女友点完餐后，不知道什么时候能做好，只好坐在餐厅里面等，直到做好，然后吃完才离开。女友本想还和我一起逛街的，但是不知道饭能什么时候做好，只好和我一起在餐厅等，而不能去逛街，直到吃完饭才能去逛街，中间等待做饭的时间浪费掉了。这就是典型的阻塞。

2.1.2 网络模型

同步阻塞 I/O 模型是最常用的一个模型，也是最简单的模型。在 linux 中，默认情况下所有的 socket 都是 blocking。它符合人们最常见的思考逻辑。阻塞就是进程 “被” 休息, CPU 处理其它进程去了。
在这个 I/O 模型中，用户空间的应用程序执行一个系统调用（recvform），这会导致应用程序阻塞，什么也不干，直到数据准备好，并且将数据从内核复制到用户进程，最后进程再处理数据，在等待数据到处理数据的两个阶段，整个进程都被阻塞。不能处理别的网络 I/O。调用应用程序处于一种不再消费 CPU 而只是简单等待响应的状态，因此从处理的角度来看，这是非常有效的。在调用 recv()/recvfrom()函数时，发生在内核中等待数据和复制数据的过程，大致如下图：

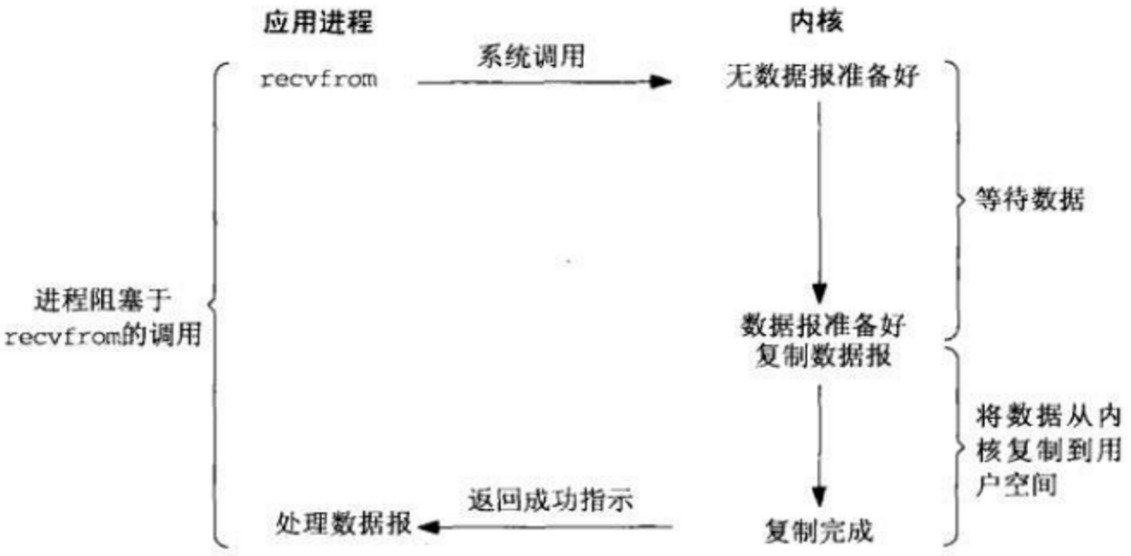


图6-1　阻塞式I/O模型

2.1.3 流程描述

当用户进程调用了 recv()/recvfrom()这个系统调用，**kernel 就开始了 IO 的第一个阶段：准备数据**（对于网络 IO 来说，很多时候数据在一开始还没有到达。比如，还没有收到一个完整的 UDP 包。这个时候 kernel 就要等待足够的数据到来）。这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区中是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。**第二个阶段：当 kernel 一直等到数据准备好了，它就会将数据从 kernel 中拷贝到用户内存**，然后 kernel 返回结果，用户进程才解除 block 的状态，重新运行起来。

所以，同步阻塞(blocking IO)的特点就是在 IO 执行的两个阶段都被 block 了。

优点：

能够及时返回数据，无延迟；

对内核开发者来说这是省事了；

缺点：

对用户来说处于等待就要付出性能的代价了；

2.2 非阻塞 I/O（nonblocking I/O）

2.2.1 场景描述

我女友不甘心白白在这等，又想去逛商场，又担心饭好了。所以我们逛一会，回来询问服务员饭好了没有，来来回回好多次，饭都还没吃都快累死了啦。这就是非阻塞。需要不断的询问，是否准备好了。

2.2.2 网络模型

同步非阻塞就是 “每隔一会儿瞄一眼进度条” 的轮询（polling）方式。在这种模型中，设备是以非阻塞的形式打开的。这意味着 I/O 操作不会立即完成，read 操作可能会返回一个错误代码，说明这个命令不能立即满足（EAGAIN 或 EWOULDBLOCK）。

在网络 I/O 时候，非阻塞 I/O 也会进行 recvform 系统调用，检查数据是否准备好，与阻塞 I/O 不一样，”非阻塞将大的整片时间的阻塞分成 N 多的小的阻塞, 所以进程不断地有机会 ‘被’ CPU 光顾”。

也就是说非阻塞的 recvform 系统调用调用之后，进程并没有被阻塞，内核马上返回给进程，如果数据还没准备好，此时会返回一个 error。进程在返回之后，可以干点别的事情，然后再发起 recvform 系统调用。重复上面的过程，循环往复的进行 recvform 系统调用。这个过程通常被称之为轮询。轮询检查内核数据，直到数据准备好，再拷贝数据到进程，进行数据处理。需要注意，拷贝数据整个过程，进程仍然是属于阻塞的状态。

在 linux 下，可以通过设置 socket 使其变为 non-blocking。当对一个 non-blocking socket 执行读操作时，流程如图所示：

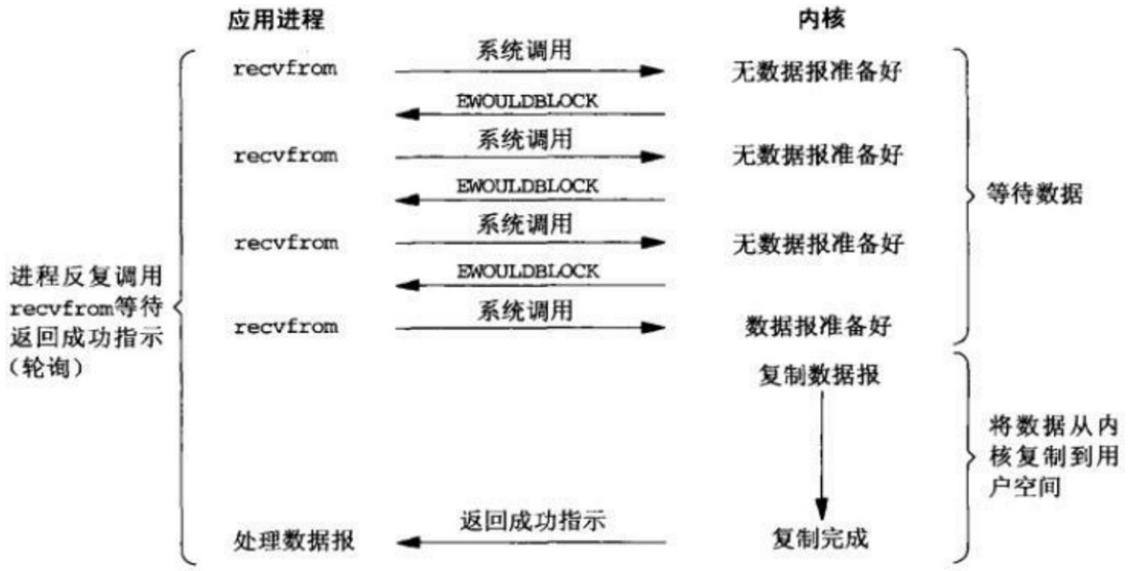


图6-2　非阻塞式I/O模型

2.2.3 流程描述

当用户进程发出 read 操作时，如果 kernel 中的数据还没有准备好，那么它并不会 block 用户进程，而是立刻返回一个 error。从用户进程角度讲，它发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦 kernel 中的数据准备好了，并且又再次收到了用户进程的 system call，那么它马上就将数据拷贝到了用户内存，然后返回。

所以，nonblocking I/O 的特点是用户进程需要不断的主动询问 kernel 数据好了没有。

非阻塞方式相比阻塞方式：

优点：能够在等待任务完成的时间里干其他活了（包括提交其他任务，也就是 “后台” 可以有多个任务在同时执行）。

缺点：任务完成的响应延迟增大了，因为每过一段时间才去轮询一次 read 操作，而任务可能在两次轮询之间的任意时间完成。这会导致整体数据吞吐量的降低。

2.3 I/O 多路复用（ I/O multiplexing）

2.3.1 场景描述

与第二个方案差不多，餐厅安装了电子屏幕用来显示点餐的状态，这样我和女友逛街一会，回来就不用去询问服务员了，直接看电子屏幕就可以了。这样每个人的餐是否好了，都直接看电子屏幕就可以了，这就是典型的 IO 多路复用。

2.3.2 网络模型

由于同步非阻塞方式**需要不断主动轮询**，轮询占据了很大一部分过程，轮**询会消耗大量的 CPU 时间**，而 “后台” 可能有多个任务在同时进行，人们就想到了循环查询多个任务的完成状态，只要有任何一个任务完成，就去处理它。如果轮询不是进程的用户态，而是有人帮忙就好了。那么这就是所谓的 “I/O 多路复用”。UNIX/Linux 下的 select、poll、epoll 就是干这个的（**epoll 比 poll、select 效率高，做的事情是一样的**）。

I/O 多路复用有两个特别的系统调用 select、poll、epoll 函数。select 调用是内核级别的，select 轮询相对非阻塞的轮询的区别在于—前者可以等待多个 socket，能实现同时对多个 I/O 端口进行监听，当其中任何一个 socket 的数据准好了，就能返回进行可读，然后进程再进行 recvform 系统调用，将数据由内核拷贝到用户进程，当然这个过程是阻塞的。**select 或 poll 调用之后，会阻塞进程，与同步阻塞(blocking I/O)不同在于，此时的 select 不是等到 socket 数据全部到达再处理，而是有了一部分数据就会调用用户进程来处理。如何知道有一部分数据到达了呢？监视的事情交给了内核，内核负责数据到达的处理。也可以理解为”非阻塞”吧。**

I/O 复用模型会用到 select、poll、epoll 函数，这几个函数也会使**进程阻塞**，但是和**阻塞 I/O** 所不同的的，这两个函数可以**同时阻塞多个 I/O 操作**。而且**可以同时对多个读操作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时（注意不是全部数据可读或可写），才真正调用 I/O 操作函数。**

对于多路复用，也就是轮询多个 socket。多路复用既然可以处理多个 I/O，也就带来了新的问题，多

个 I/O 之间的顺序变得不确定了，当然也可以针对不同的编号。具体流程，如下图所示：

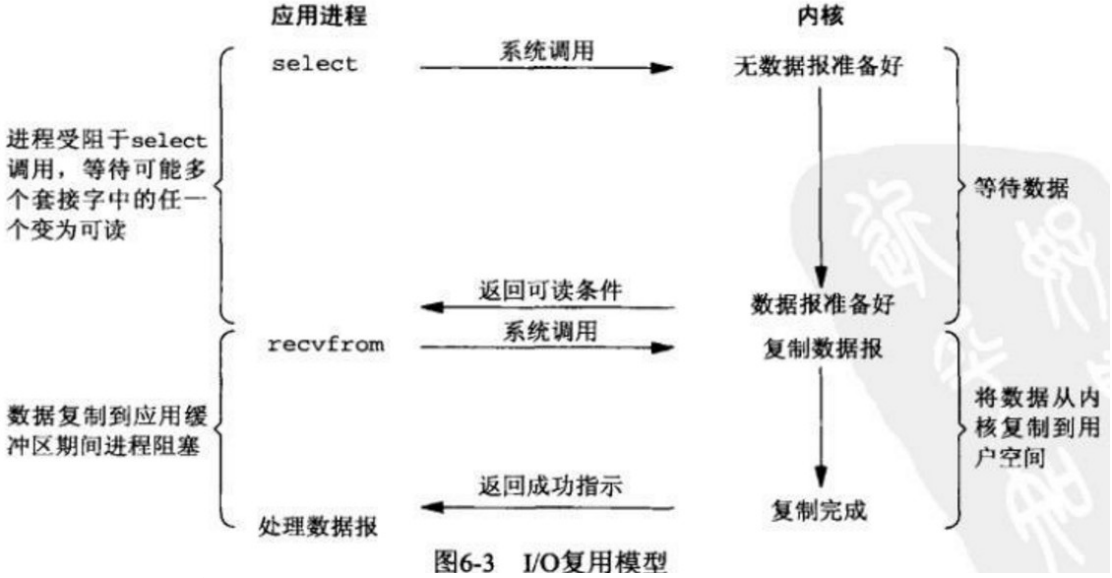


图6-3 I/O复用模型

2.3.3 流程描述

I/O multiplexing 就是我们说的 select，poll，epoll，有些地方也称这种 I/O 方式为 event driven I/O。select/epoll 的好处就在于**单个 process 就可以同时处理多个网络连接的 I/O**。它的基本原理就是 **select，poll，epoll 这个 function 会不断的轮询所负责的所有 socket，当某个 socket 有数据到达了，就通知用户进程。**

当用户进程调用了 select，那么整个进程会被 block，而同时，kernel 会“监视”所有 select 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。这个时候用户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程。

多路复用的特点是通过一种机制一个进程能同时等待 IO 文件描述符，内核监视这些文件描述符（套接字描述符），其中的任意一个进入读就绪状态，select， poll，epoll 函数就可以返回。

对于监视的方式，又可以分为 select， poll， epoll 三种方式。

上面的图和 blocking I/O 的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个 system call (select 和 recvfrom)，而 blocking I/O 只调用了一个 system call (recvfrom)。但是，用 select 的优势在于它可以同时处理多个 connection。所以，**如果处理的连接数不是很高的话，使用 select/epoll 的 web server 不一定比使用 multi-threading + blocking I/O 的 web server 性能更好，可能延迟还更大。（select/epoll 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）**

在 IO multiplexing Model 中，实际中，对于每一个 socket，一般都设置成为 non-blocking，但是，如上图所示，整个用户的 process 其实是一直被 block 的。只不过 process 是被 select 这个函数 block，而不是被 socket I/O 给 block。**所以 I/O 多路复用是阻塞在 select，epoll 这样的系统调用之上，而没有阻塞在真正的 I/O 系统调用如 recvfrom 之上。**

在 I/O 编程过程中，当**需要同时处理多个客户端**接入请求时，可以利用**多线程或者 I/O 多路复用**技术进行处理。**I/O 多路复用技术通过把多个 I/O 的阻塞复用到同一个 select 的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求**。与传统的多线程/多进程模型比，I/O 多路复用的最大优势是系统**开销小**，系统**不需要创建新的额外进程或者线程，也不需要维护**这些进程和线程的运行，降底了系统的维护工作量，节省了系统资源，I/O 多路复用的主要应用场景如下：

服务器需要同时处理多个处于监听状态或者多个连接状态的套接字。

服务器需要同时处理多种网络协议的套接字。

了解了前面三种 I/O 模式，在用户进程进行系统调用的时候，他们在等待数据到来的时候，处理的方式不一样，直接等待，轮询，select 或 poll 轮询，两个阶段过程：

第一个阶段有的阻塞，有的不阻塞，有的可以阻塞又可以不阻塞。

第二个阶段都是阻塞的。**从整个 I/O 过程来看，他们都是顺序执行的，因此可以归为同步模型**

(synchronous)。都是进程主动等待且向内核检查状态。【此句很重要!!!】

高并发的程序一般使用同步非阻塞方式而非多线程 + 同步阻塞方式。要理解这一点，首先要扯到**并发和并行的区别**。比如去某部门办事需要依次去几个窗口，**办事大厅里的人数就是并发数**，而**窗口个数就是并行**度。也就是说并发数是指同时进行的任务数（如同时服务的 HTTP 请求），而并行数是可以同时工作的物理资源数量（如 CPU 核数）。通过合理调度任务的不同阶段，并发数可以远远大于并行度，这就是区区几个 CPU 可以支持上万个用户并发请求的奥秘。在这种高并发的情况下，为每个任务（用户请求）创建一个进程或线程的开销非常大。而同步非阻塞方式可以把多个 I/O 请求丢到后台去，这就可以在一个进程里服务大量的并发 I/O 请求。

注意：I/O 多路复用是同步阻塞模型还是异步阻塞模型，在此给大家分析下：

此处仍然不太清楚的，强烈建议大家在细究《聊聊同步、异步、阻塞与非阻塞》中讲同步与异步的根本性区别，同步是需要主动等待消息通知，而异步则是被动接收消息通知，通过回调、通知、状态等方式来被动获取消息。IO 多路复用在阻塞到 select 阶段时，用户进程是主动等待并调用 select 函数获取数据就绪状态消息，并且其进程状态为阻塞。所以，把 IO 多路复用归为同步阻塞模式。

2.4 信号驱动式 I/O（signal-driven I/O）

信号驱动式 I/O：首先我们允许 Socket 进行信号驱动 I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 I/O 操作函数处理数据。过程如下图所示：

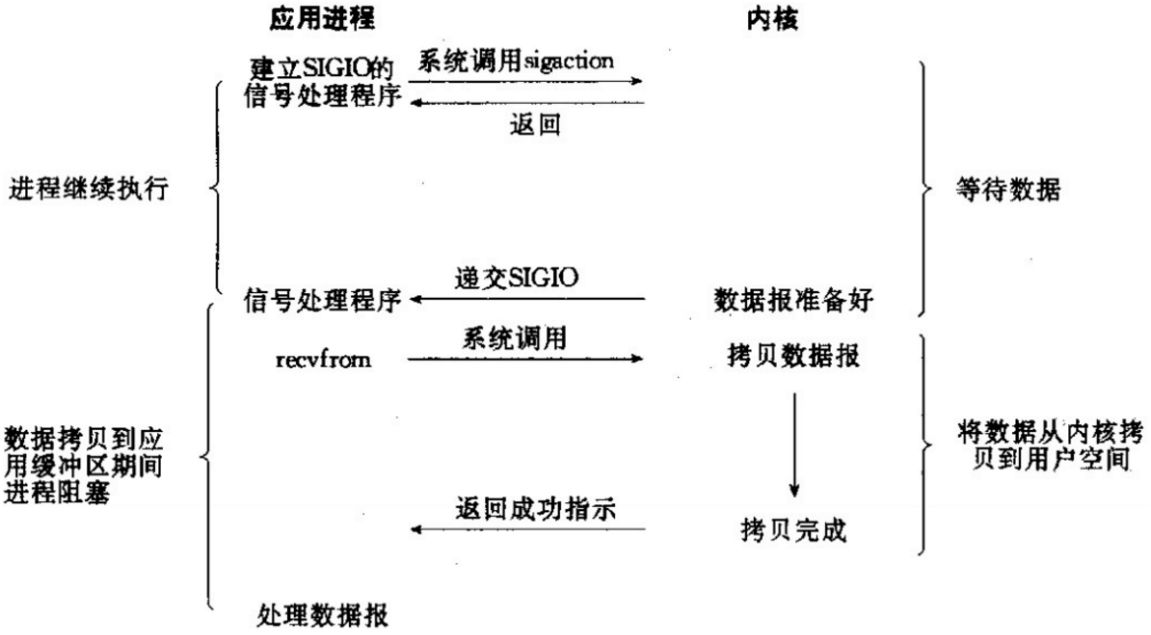


图 6.4 信号驱动 I/O 模型

2.5 异步非阻塞 I/O（asynchronous I/O）

2.5.1 场景描述

女友不想逛街，又餐厅太吵了，回家好好休息一下。于是我们叫外卖，打个电话点餐，然后我和女友可以在家好好休息一下，饭好了送货员送到家里来。这就是典型的异步，只需要打个电话说一下，然后可以做自己的事情，饭好了就送来了。

2.5.2 网络模型

相对于同步 I/O，异步/IO 不是顺序执行。用户进程进行 aio_read 系统调用之后，无论内核数据是否准备好，都会直接返回给用户进程，然后用户态进程可以去做别的事情。等到 socket 数据准备好了，内核直接复制数据给进程，然后从内核向进程发送通知。I/O 两个阶段，进程都是非阻塞的。

Linux 提供了 AIO 库函数实现异步，但是用的很少。目前有很多开源的异步 I/O 库，例如 libevent、libev、libuv。异步过程如下图所示：

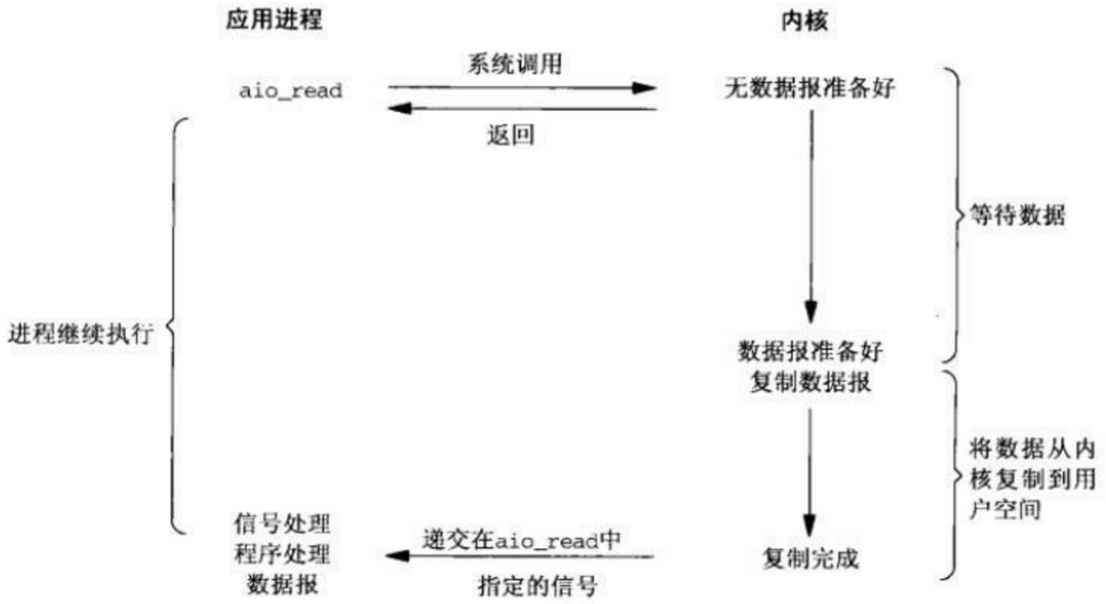


图6-5 异步I/O模型

2.5.3 流程描述

用户进程发起 aio_read 操作之后，立刻就可以开始去做其它的事。而另一方面，从 kernel 的角度，当它受到一个 asynchronous read 之后，首先它会立刻返回，所以不会对用户进程产生任何 block。然后，kernel 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel 会给用户进程发送一个 signal 或执行一个基于线程的回调函数来完成这次 I/O 处理过程，告诉它 read 操作完成了。

在 Linux 中，通知的方式是 “信号”：

如果这个进程正在用户态忙着做别的事（例如在计算两个矩阵的乘积），那就强行打断之，调用事先注册的信号处理函数，这个函数可以决定何时以及如何处理这个异步任务。由于信号处理函数是突然闯进来的，因此跟中断处理程序一样，有很多事情是不能做的，因此保险起见，一般是把事件 “登记” 一下放进队列，然后返回该进程原来在做的事。

如果这个进程正在内核态忙着做别的事，例如以同步阻塞方式读写磁盘，那就只好把这个通知挂起来了，等到内核态的事情忙完了，快要回到用户态的时候，再触发信号通知。

如果这个进程现在被挂起了，例如无事可做 sleep 了，那就把这个进程唤醒，下次有 CPU 空闲的时候，就会调度到这个进程，触发信号通知。

异步 API 说来轻巧，做来难，这主要是对 API 的实现者而言的。Linux 的异步 I/O（AIO）支持是 2.6.22 才引入的，还有很多系统调用不支持异步 I/O。Linux 的异步 I/O 最初是为数据库设计的，因此通过异步 I/O 的读写操作不会被缓存或缓冲，这就无法利用操作系统的缓存与缓冲机制。

很多人把 Linux 的 O_NONBLOCK 认为是异步方式，但事实上这是前面讲的同步非阻塞方式。需要指出的是，虽然 Linux 上的 I/O API 略显粗糙，但每种编程框架都有封装好的异步 I/O 实现。操作系统少做事，把更多的自由留给用户，正是 UNIX 的设计哲学，也是 Linux 上编程框架百花齐放的一个原因。

从前面 I/O 模型的分类中，我们可以看出 AIO 的动机：

同步阻塞模型需要在 IO 操作开始时阻塞应用程序。这意味着不可能同时重叠进行处理和 IO 操作。

同步非阻塞模型允许处理和 IO 操作重叠进行，但是这需要应用程序根据重现的规则来检查 IO 操作的状态。

这样就剩下异步非阻塞 IO 了，它允许处理和 IO 操作重叠进行，包括 IO 操作完成的通知。

I/O 多路复用除了需要阻塞之外，select 函数所提供的功能（异步阻塞 I/O）与 AIO 类似。不过，它是对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。