测试面试题总结及问题解析

计算机网络（tcp/ip 详解卷 1、谢希仁的计算机网络）、**数据库**（sql 必知必会、数据库案例应用）、**操作系统**（现代操作系统、清华的公开课）、**数据结构和算法**（剑指 offer、大话数据结构、leetcode 只做了一点、还有牛客上的算法课）、**java 语言**（ java 程序员面试宝典、java 程序员的基本修养、大话设计模式只看了重要的 ）、**测试相关**（软件测试的艺术、从菜鸟到测试架构师、软件测试技术大全、selenium，qtp，junit 的一些相关资料和书）、**linux** 看了一点（鸟哥的私房菜基础篇和网络篇）。总之很多东西是看完就忘，我是边看会在 onenote 记笔记，忘了就回去翻笔记和书再看看。

——摘自牛客某位大神的总结

因为没有对知识点进行分类，所以我在每个问题后面加上了标签，表示属于哪本书，如果这个看的还不明白就翻对应的书或者我给的博客链接看看，后面的星星表示知识点的重要程度，我大概是根据面试被问及的次数标记的，仅供参考。问题是我整理别人的面经还有我自己的一些面经，答案是我从书上或者一些博客上找的，所以我只是一个搬运工哈哈哈

1. 计算机网络里面 socket 通信常用的的函数叫啥?（计算机网络）

Socket:网络上的两个程序通过一个双向的通信连接实现数据的交换,根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：**服务器监听**，**客户端请求**，**连接确认**。

（1）服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

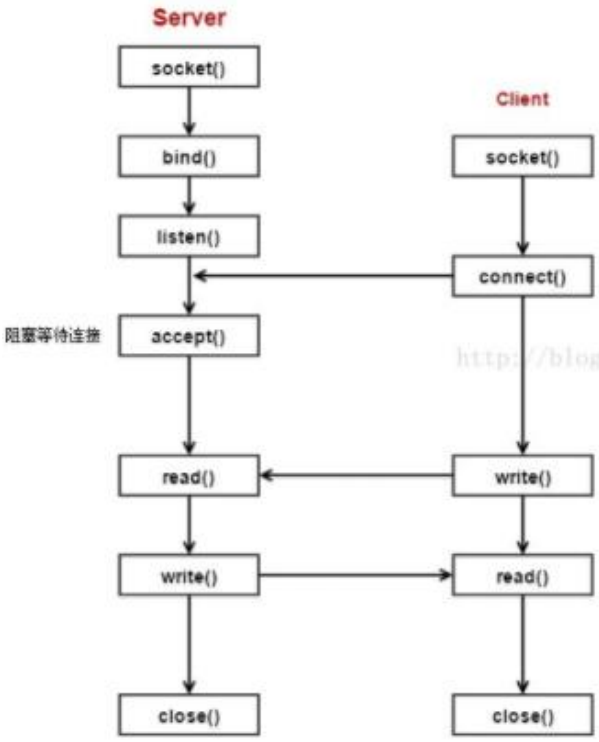
（2）客户端请求：由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

（3）连接确认：是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

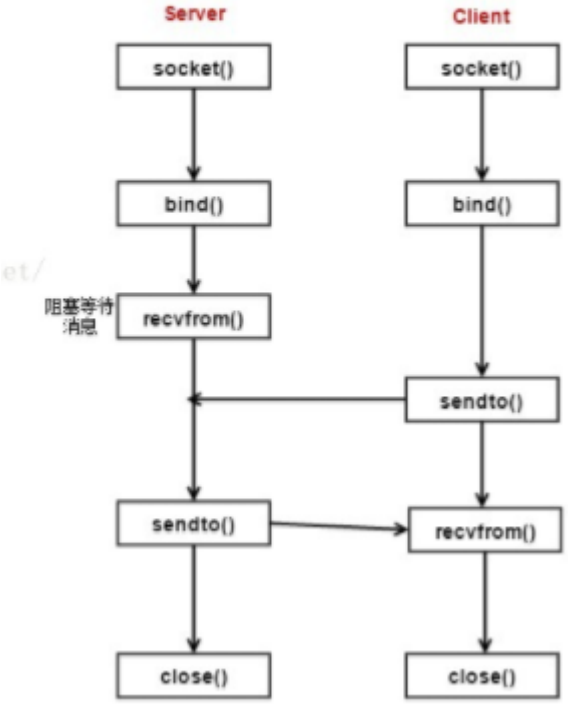
常用的函数：

1. 创建：socket()函数，这个操作类似于打开文件操作，返回 socket 的 socket 描述符。
2. 绑定： bind()函数，把一个地址族的特定地址指定给 socket，而不是由系统随机分配.
3. listen()、connect()函数，客户端通过调用 connect 函数来建立与 TCP 服务器的连接。服务器端调用 listen()，socket 开始等待客户的链接请求
4. accept()函数，服务器收到请求后，用 accept 接受请求，然后链接就建立了，可以开始读写操作。
5. read(),write()读写操作
6. close()函数,读写完毕后要关闭相应的 socket 描述字

TCP方式


Server
socket()
bind()
listen()
阻塞等待连接
accept()
read()
write()
close()
Client
socket()
connect()
write()
read()
close()

UDP方式


Server
socket()
bind()
阻塞等待消息
recvfrom()
sendto()
close()
Client
socket()
bind()
sendto()
recvfrom()
close()

2 数据结构里面的排序算法（排序算法，表格很重要）

各种排序的稳定性，时间复杂度和空间复杂度总结：

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数。

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况		
插入排序	插入排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	Shell 排序	O(N^1.3)	O(N)	O(N^2)	O(1)	不稳定
选择排序	选择排序	O(N^2)	O(N^2)	O(N^2)	O(1)	不稳定
	堆排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(1)	不稳定
交换排序	冒泡排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	快速排序	O(N*lgN)	O(N*lgN)	O(N^2)	O(lgN)	不稳定
归并排序	归并排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(N)	稳定

当 **n 较大**，则应采用时间复杂度为 O(Nlog2N)的排序方法：**快速排序、堆排序或归并排序**。 **快速排序**是目前基于比较的内部排序中被认为是**最好**的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短。黑色表中快速排序的空间复杂度不正确其他正确快速排序的空间复杂度为 O(logN).

1. 从平均时间来看，快速排序是效率最高的，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后者相比较的结果是，在 n 较大时归并排序使用时间较少，但使用辅助空间较多。

2. 上面说的简单排序包括除希尔排序之外的所有冒泡排序、插入排序、简单选择排序。其中直接插入排序最简单，当**序列基本有序或者 n 较小时，直接插入排序**是好的方法，因此常将它和其他的排序方法，如快速排序、归并排序等结合在一起使用。

3. 基数排序的时间复杂度也可以写成 O(d*n)。因此它最使用于 n 值很大而关键字较小的的序列。若关键字也很大，而序列中大多数记录的最高关键字均不同，则亦可先按最高关键字不同，将序列分成若干小的子序列，而后进行直接插入排序。

4. 从方法的稳定性来比较，基数排序是稳定的内排方法，所有时间复杂度为 O(n^2)的简单排序也是稳定的。但是快速排序、堆排序、希尔排序等时间性能较好的排序方法都是不稳定的。稳定性需要根据具体需求选择

2.1 直接插入排序基本思想:

将一个记录插入到已排序好的有序表中，从而得到一个新记录数增 1 的有序表。即：先将序列的第 1 个记录看成是一个有序的子序列，然后从第 2 个记录逐个进行插入，直至整个序列有序为止。时间复杂度：**O（n^2）**

如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，**所以插入排序是稳定的。**

```
void insert(int a[],int n)
{
    int i, j;
    for(i=1;i<n;i++)
    {
        if(a[i]<a[i-1])
        {
            int temp = a[i];
            for(j=i-1;a[j]>temp;j--)
            {
                a[j+1] = a[j];
            }
            a[j+1] = temp;
        }
    }
}
```

2.2 希尔排序（插入排序）基本思想:

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

操作方法：

选择一个增量序列 t1，t2，…，tk，其中 ti>tj(i<j)，tk=1；

按增量序列个数 k，对序列进行 k 趟排序；

每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

我们简单处理增量序列：增量序列 d = {n/2 ,n/4, n/8 .....1} n 为要排序数的个数

即：先将要排序的一组记录按某个增量 d（n/2,n 为要排序数的个数）分成若干组子序列，每组中记录的下标相差 d.对每组中全部元素进行直接插入排序，然后再用一个较小的增量（d/2）对它进行分组，在每组中再进行直接插入排序。继续不断缩小增量直至为 1，最后使用直接插入排序完成排序。

```
1.  void ShellInsertSort(int a[], int n, int dk)
2.  {
3.    for(int i= dk; i<n; ++i){
4.      if(a[i] < a[i-dk]){        //若第 i 个元素大于 i-1 元素，直接插入。小于的话，移动有序表后插入
5.        int j = i-dk;
6.        int x = a[i];          //复制为哨兵，即存储待排序元素
7.        a[i] = a[i-dk];        //首先后移一个元素
8.        while(x< a[j]){     //查找在有序表的插入位置
9.          a[j+dk] = a[j];
10.         j -= dk;           //元素后移
11.       }
12.       a[j+dk] = x;           //插入到正确位置
13.     }
14.     print(a, n,i );
15.   }
16. }
17. /**
18.  * 先按增量 d（n/2,n 为要排序数的个数进行希尔排序
19.  *
20.  */
21. void shellSort(int a[], int n){
22.
23.   int dk = n/2;
24.   while( dk >= 1  ){
25.     ShellInsertSort(a, n, dk);
26.     dk = dk/2;
27.   }
28. }
```

希尔排序时效分析很难，关键码的比较次数与记录移动次数依赖于增量因子序列 d 的选取，特定情况下可以准确估算出关键码的比较次数和记录的移动次数。目前还没有人给出选取最好的增量因子序列的方法。增量因子序列可以有各种取法，有取奇数的，也有取质数的，但需要注

意：增量因子中除 1 外没有公因子，且最后一个增量因子必须为 1。希尔排序方法是一个不稳定的排序方法。

2.3 简单选择排序基本思想:

在要排序的一组数中，选出最小（或者最大）的一个数与第 1 个位置的数交换；然后在剩下的数当中再找最小（或者最大）的与第 2 个位置的数交换，依次类推，直到第 n-1 个元素（倒数第二个数）和第 n 个元素（最后一个数）比较为止。

操作方法：

第一趟，从 n 个记录中找出关键码最小的记录与第一个记录交换；

第二趟，从第二个记录开始的 n-1 个记录中再选出关键码最小的记录与第二个记录交换；

以此类推.....

第 i 趟，则从第 i 个记录开始的 n-i+1 个记录中选出关键码最小的记录与第 i 个记录交换，

直到整个序列按关键码有序。

```
void Select(int a[],int n)
{
    int min = 0;
    for(int i=0;i<n-1;i++)
    {
        min = i;
        for(int j=i+1;j<n;j++)
        {
            if(a[min]>a[j])
                min = j;
        }
        if(min!=i)
            swap(a[i],a[min]);
    }
}
```

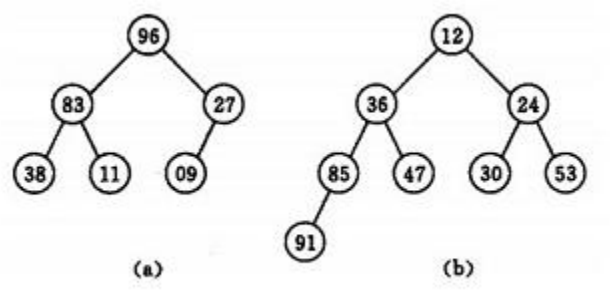
2.4 堆排序（选择排序）基本思想:

堆排序是一种树形选择排序，是对直接选择排序的有效改进。堆的定义如下：具有 n 个元素的序列（k1,k2,...,kn),当且仅当满足

时称之为堆。由堆的定义可以看出，**堆顶元素**（即第一个元素）必为最小项（小顶堆）。若以一维数组存储一个堆，则堆对应一棵完全二叉树，且所有非叶结点的值均不大于(或不小于)其子女的值，根结点（堆顶元素）的值是最小(或最大)的。如：

（a）大顶堆序列：（96, 83,27,38,11,09)

(b) 小顶堆序列：（12，36，24，85，47，30，53，91）



初始时把要排序的 n 个数的序列看作是一棵**顺序存储的二叉树（一维数组存储二叉树）**，调整它们的存储顺序，使之成为一个堆，将堆顶元素输出，得到 n 个元素中最小(或最大)的元素，这时堆的根节点的数最小（或者最大）。然后对前面(n-1)个元素重新调整使之成为堆，输出堆顶元素，得到 n 个元素中次小(或次大)的元素。依此类推，直到只有两个节点的堆，并对它们作交换，最后得到有 n 个节点的有序序列。称这个过程为**堆排序**。

因此，实现堆排序需解决两个问题：
1. 如何将 n 个待排序的数建成堆；（建堆）
2. 输出堆顶元素后，怎样调整剩余 n-1 个元素，使其成为一个新堆。（调整堆）

首先讨论第二个问题：输出堆顶元素后，对剩余 n-1 元素重新建成堆的调整过程。
调整小顶堆的方法：

1）设有 m 个元素的堆，输出堆顶元素后，剩下 m-1 个元素。将堆底元素送入堆顶（（最后一个元素与堆顶进行交换），堆被破坏，其原因仅是根结点不满足堆的性质。

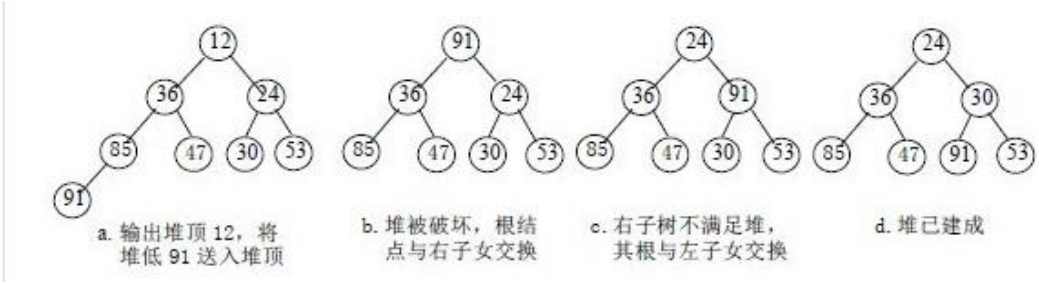
2）将根结点与左、右子树中较小元素的进行交换。

3）若与左子树交换：如果左子树堆被破坏，即左子树的根结点不满足堆的性质，则重复方法 （2）.

4）若与右子树交换，如果右子树堆被破坏，即右子树的根结点不满足堆的性质。则重复方法 （2）.

5）继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建成。

称这个自根结点到叶子结点的调整过程为筛选。如图：



再讨论对 n 个元素初始建堆的过程。

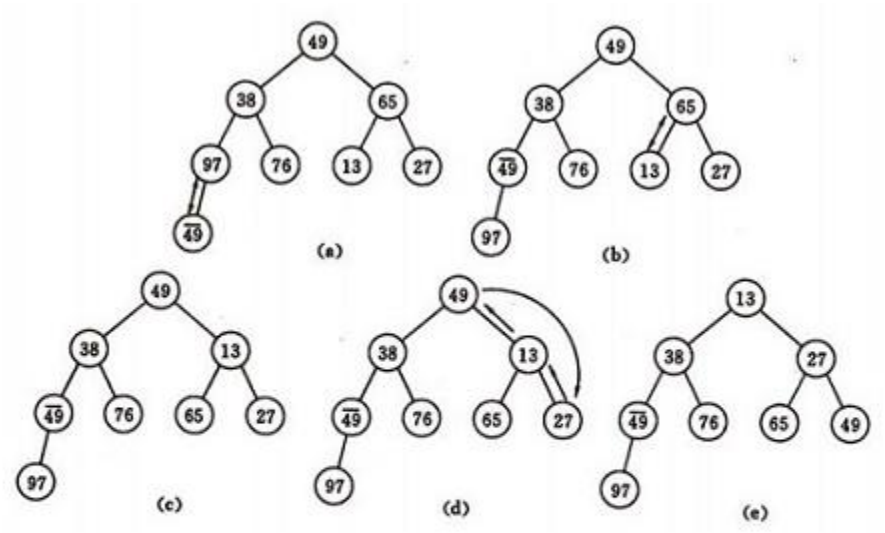
建堆方法：对初始序列建堆的过程，就是一个反复进行筛选的过程。

1）n 个结点的完全二叉树，则最后一个结点是第 round（n/2）个结点的子树。

2）筛选从第 round（n/2）个结点为根的子树开始，该子树成为堆。

3）之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。

如图建堆初始过程：无序序列：（49，38，65，97，76，13，27，49）



（a）无序序列；（b）97 被筛选之后的状态；（c）65 被筛选之后的状态；
（d）38 被筛选之后的状态；（e）49 被筛选之后建成的堆

从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。堆排序最坏情况下，**时间复杂度也为：O(nlogn)**。

```
1.  void HeapAdjust(int H[],int s, int length)
2.  {
3.    int tmp  = H[s];
4.    int child = 2*s+1; //左孩子结点的位置。(i+1 为当前调整结点的右孩子结点的位置)
5.    while (child < length) {
6.      if(child+1 <length && H[child]<H[child+1]) { // 如果右孩子大于左孩子(找到比当前待调整结点大的孩子结点)
7.        ++child ;
8.      }
9.      if(H[s]<H[child]) {  // 如果较大的子结点大于父结点
10.       H[s] = H[child]; // 那么把较大的子结点往上移动，替换它的父结点
11.       s = child;      // 重新设置 s ,即待调整的下一个结点的位置
12.       child = 2*s+1;
13.     } else {        // 如果当前待调整结点大于它的左右孩子，则不需要调整，直接退出
14.       break;
15.     }
16.     H[s] = tmp;      // 当前待调整的结点放到比其大的孩子结点位置上
17.   }
18.   print(H,length);
19. }
20.
```

```
21.
22.         /**
23.          * 初始堆进行调整
24.          * 将 H[0..length-1]建成堆
25.          * 调整完之后第一个元素是序列的最小的元素
26.          */
27.         void BuildingHeap(int H[], int length)
28.         {
29.           //最后一个有孩子的节点的位置 i= (length -1) / 2
30.           for (int i = (length -1) / 2 ; i >= 0; --i)
31.             HeapAdjust(H,i,length);
32.         }
33.         /**
34.          * 堆排序算法
35.          */
36.         void HeapSort(int H[],int length)
37.         {
38.           //初始堆
39.           BuildingHeap(H, length);
40.           //从最后一个元素开始对序列进行调整
41.           for (int i = length - 1; i > 0; --i)
42.           {
43.             //交换堆顶元素 H[0]和堆中最后一个元素
44.             int temp = H[i]; H[i] = H[0]; H[0] = temp;
45.             //每次交换堆顶元素和堆中最后一个元素之后，都要对堆进行调整
46.             HeapAdjust(H,0,i);
47.          }
48.         }
```

2.5 冒泡排序（选择排序）基本思想:

在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。

```
//1.冒泡排序
void bubbleSort(int  a[],int  n)
{
Bool flag  = true;
      for(int  i=0;i<n-1;i++)
      {
            flag  = false;
            for(int  j=n-1;j>=i;j--)
            {
```

```
            if(a[j]<a[j-1])
            {
                swap(a[j],a[j-1]);
                Flag = true;
            }
        }
        If(!flag)
            Break;
    }
}
```

如果上面代码中，里面一层循环在某次扫描中没有执行交换，则说明此时数组已经全部有序列，无需再扫描了。因此，增加一个标记，每次发生交换，就标记，如果某次循环完没有标记，则说明已经完成排序。

2.6 快速排序（交换排序）基本思想:

1）选择一个基准元素,通常选择第一个元素或者最后一个元素,

2）通过一趟排序将待排序的记录分割成独立的两部分，其中一部分记录的元素值均比基准元素值小。另一部分记录的元素值比基准值大。

3）此时基准元素在其排好序后的正确位置。

4）然后分别对这两部分记录用同样的方法继续进行排序，直到整个序列有序。

快速排序是通常被认为在同数量级（O(nlog2n)）的排序方法中平均性能最好的。但若初始序列按**关键码有序或基本有序时，快排序反而退化为冒泡排序**。为改进之，通常以“三者取中法”来选取基准记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个**不稳定**的排序方法。

```
//4.快速排序
int partion(int a[],int low,int high)
{
    int key = a[low];
    while(low<high)
    {
        while(low<high&&a[high]>=key)
            high--;
        swap(a[high],a[low]);
        while(low<high&&a[low]<=key)
            low++;
        swap(a[high],a[low]);
    }
    return low;
```

```
}
void Qsort(int a[],int low,int high)
{
    if(low<high)
    {
        int pivot;
        pivot = partion(a , low , high);
        Qsort(a,0,pivot-1);
        Qsort(a,pivot+1,high);
    }
}
void fast(int a[],int n)
{
    Qsort(a,0,n-1);
}
```

2.7 归并排序基本思想:

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

合并方法：

设 r[i…n]由两个有序子表 r[i…m]和 r[m+1…n]组成，两个子表长度分别为 n-i +1、n-m。

1. j=m+1；k=i；i=i; //置两个子表的起始下标及辅助数组的起始下标

2. 若 i>m 或 j>n，转⑷ //其中一个子表已合并完，比较选取结束

3. //选取 r[i]和 r[j]较小的存入辅助数组 rf
如果 r[i]<r[j]，rf[k]=r[i]； i++； k++； 转⑵
否则，rf[k]=r[j]； j++； k++； 转⑵

4. //将尚未处理完的子表中元素存入 rf
如果 i<=m，将 r[i…m]存入 rf[k…n] //前一子表非空
如果 j<=n， 将 r[j…n] 存入 rf[k…n] //后一子表非空

5. 合并结束。

```
1.  //将 r[i…m]和 r[m +1 …n]归并到辅助数组 rf[i…n]
2.  void Merge(ElemType *r,ElemType *rf, int i, int m, int n)
3.  {
```

```
4.              int j,k;
5.              for(j=m+1,k=i; i<=m && j <=n ; ++k){
6.                 if(r[j] < r[i]) rf[k] = r[j++];
7.                 else rf[k] = r[i++];
8.              }
9.              while(i <= m)  rf[k++] = r[i++];
10.             while(j <= n)  rf[k++] = r[j++];
11.          }
```

2.8 桶排序/基数排序基本思想:

基本思想：是将阵列分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递回方式继续使用桶排序进行排序）。桶排序是鸽巢排序的一种归纳结果。当要被排序的阵列内的数值是均匀分配的时候，桶排序使用线性时间（Θ（n））。但桶排序并不是 比较排序，他不受到 O(n log n) 下限的影响。

简单来说，就是把数据分组，放在一个个的桶中，然后对每个桶里面的在进行排序。

例如要对大小为[1..1000]范围内的 n 个整数 A[1..n]排序

首先，可以把桶设为大小为 10 的范围，具体而言，设集合 B[1]存储[1..10]的整数，集合 B[2]存储 (10..20]的整数，……集合 B[i]存储((i-1)*10, i*10]的整数，i = 1,2,..100。总共有 100 个桶。 然后，对 A[1..n]从头到尾扫描一遍，把每个 A[i]放入对应的桶 B[j]中。 再对这 100 个桶中每个桶里的数字排序，这时可用冒泡，选择，乃至快排，一般来说任 何排序法都可以。最后，依次输出每个桶里面的数字，且每个桶中的数字从小到大输出，这样就得到所有数字排好序的一个序列了。

假设有 n 个数字，有 m 个桶，如果数字是平均分布的，则每个桶里面平均有 n/m 个数字。如果对每个桶中的数字采用快速排序，那么整个算法的复杂度是

O(n + m * n/m*log(n/m)) = O(n + nlogn - nlogm)

从上式看出，当 m 接近 n 的时候，桶排序复杂度接近 O(n)。当然，以上复杂度的计算是基于输入的 n 个数字是平均分布这个假设的。这个假设是很强的 ，实际应用中效果并没有这么好。如果所有的数字都落在同一个桶中，那就退化成一般的排序了。

前面说的几大排序算法 ，大部分时间复杂度都是 O（n^2），也有部分排序算法时间复杂度是 O(nlogn)。而桶式排序却能实现 O（n）的时间复杂度。但桶排序的缺点是：

1）首先是空间复杂度比较高，需要的额外开销大。排序有两个数组的空间开销，一个存放待排序数组，一个就是所谓的桶，比如待排序值是从 0 到 m-1，那就需要 m 个桶，这个桶数组就要至少 m 个空间。

2）其次待排序的元素都要在一定的范围内等等。

桶式排序是一种分配排序。分配排序的特点是不需要进行关键码的比较，但前提是要知道待排序列的一些具体情况。

分配排序的基本思想： **说白了就是进行多次的桶式排序。**基数排序过程无须比较关键字，而是通过“分配”和“收集”过程来实现排序。它们的时间复杂度可达到线性阶：O(n)。

实例：

扑克牌中 52 张牌，可按花色和面值分成两个字段，其大小关系为：

花色： **梅花< 方块< 红心< 黑心**

面值： **2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A**

若对扑克牌按花色、面值进行升序排序，得到如下序列：
即两张牌，若花色不同，不论面值怎样，花色低的那张牌小于花色高的，只有在同花色情况下，大小关系才由面值的大小确定。这就是多关键码排序。

为得到排序结果，我们讨论两种排序方法。
方法 1：先对花色排序，将其分为 4 个组，即梅花组、方块组、红心组、黑心组。再对每个组分别按面值进行排序，最后，将 4 个组连接起来即可。
方法 2：先按 13 个面值给出 13 个编号组(2 号，3 号，...，A 号)，将牌按面值依次放入对应的编号组，分成 13 堆。再按花色给出 4 个编号组(梅花、方块、红心、黑心)，将 2 号组中牌取出分别放入对应花色组，再将 3 号组中牌取出分别放入对应花色组，……，这样，4 个花色组中均按面值有序，然后，将 4 个花色组依次连接起来即可。

设 n 个元素的待排序列包含 d 个关键码{k1，k2，…，kd}，则称序列对关键码{k1，k2，…，kd}有序是指：对于序列中任两个记录 r[i]和 r[j](1≤i≤j≤n)都满足下列有序关系：

其中 k1 称为最主位关键码，kd 称为最次位关键码 。

两种多关键码排序方法：

多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序逐次排序，分两种方法：

最高位优先(Most Significant Digit first)法，简称 MSD 法：

1）先按 k1 排序分组，将序列分成若干子序列，同一组序列的记录中，关键码 k1 相等。

2）再对各组按 k2 排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码 kd 对各子组排序后。

3）再将各组连接起来，便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法一即是 MSD 法。

最低位优先(Least Significant Digit first)法，简称 LSD 法：

1) 先从 kd 开始排序，再对 kd-1 进行排序，依次重复，直到按 k1 排序分组分成最小的子序列后。2) 最后将各个子序列连接起来，便可得到一个有序的序列，扑克牌按花色、面值排序中介绍的方法二即是 LSD 法。

基于 LSD 方法的链式基数排序的基本思想

“多关键字排序”的思想实现“单关键字排序”。对数字型或字符型的单关键字，可以看作由多个数位或多个字符构成的多关键字，此时可以采用“分配-收集”的方法进行排序，这一过程称作基数排序法，其中每个数字或字符可能的取值个数称为基数。比如，扑克牌的花色基数为 4，面值基数为 13。在整理扑克牌时，既可以先按花色整理，也可以先按面值整理。按花色整理时，先按红、黑、方、花的顺序分成 4 摞（分配），再按此顺序再叠放在一起（收集），然后按面值的顺序分成 13 摞（分配），再按此顺序叠放在一起（收集），如此进行二次分配和收集即可将扑克牌排列有序。

基数排序:

是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。**基数排序**基于分别排序，分别收集，所以是**稳定**的。

3 操作系统里面的加锁机制（操作系统）

在多线程编程中，为了保证数据操作的一致性，操作系统引入了锁机制，用于保证临界区代码的安全。通过锁机制，能够保证在多核多线程环境中，在某一个时间点上，只能有一

个线程进入临界区代码，从而保证临界区中操作数据的一致性。
所谓的锁，说白了就是内存中的一个整型数，拥有两种状态：空闲状态和上锁状态。加锁时，判断锁是否空闲，如果空闲，修改为上锁状态，返回成功；如果已经上锁，则返回失败。解锁时，则把锁状态修改为空闲状态。单核情况下禁止中断，多核时硬件提供了锁内存总线的机制，我们在锁内存总线的状态下执行 test and set 操作，就能保证同时只有一个核来 test and set，从而避免了多核下发生的问题。

4 数据库中的多表查询（使用 SQL）和海量数据时候怎么处理（数据库）

连接查询包括合并、内连接、外连接和交叉连接，如果涉及多表查询，了解这些连接的特点很重要。只有真正了解它们之间的区别，才能正确使用。

1、**Union**

UNION 操作符用于合并两个或多个 SELECT 语句的结果集。
UNION 运算符通过组合其他两个结果表（例如 TABLE1 和 TABLE2）并消去表中任何重复行而派生出一个结果表。
当 ALL 随 UNION 一起使用时（即 UNION ALL），不消除重复行。两种情况下，派生表的每一行不是来自 TABLE1 就是来自 TABLE2。
注意：使用 UNION 时，两张表查询的结果有相同数量的列，列类型相似。

2、**INNER JOIN（内连接）**

INNER JOIN（内连接），也成为自然连接
作用：根据两个或多个表中的列之间的关系，从这些表中查询数据。
注意: 内连接是从结果中删除其他被连接表中没有匹配行的所有行，所以内连接可能会丢失信息。
重点：内连接，只查匹配行。

3、**外连接**

与内连接相比，即使没有匹配行，也会返回一个表的全集。
外连接分为三种：左外连接，右外连接，全外连接。对应 SQL：LEFT/RIGHT/FULL OUTER JOIN。通常我们省略 outer 这个关键字。写成：LEFT/RIGHT/FULL JOIN。
重点：至少有一方保留全集，没有匹配行用 NULL 代替。
1）LEFT OUTER JOIN，简称 LEFT JOIN，左外连接（左连接）
结果集保留左表的所有行，但只包含第二个表与第一表匹配的行。第二个表相应的空行被放入 NULL 值。

4、**CROSS JOIN（交叉连接）**

交叉连接。交叉连接返回左表中的所有行，左表中的每一行与右表中的所有行组合。交叉连接也称作笛卡尔积。
简单查询两张表组合，这是求笛卡儿积，效率最低。

2)海量数据处理 SQL 语句方面：

尽量避免使用一些需要全表扫描的语句。
a. 应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。
b. 应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如： select id from t where num=10 or num=20 可以这样查询： select id from t where num=10 union all select id from t where num=20
c. in 和 not in 也要慎用，否则会导致全表扫描，如： select id from t where num in(1,2,3) 对于连续的数值，能用 between 就不要用 in 了： select id from t where num between 1 and 3

d. 下面的查询也将导致全表扫描： select id from t where name like ‘%abc%’
e. 如果在 where 子句中使用参数，也会导致全表扫描。因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描： select id from t where num=@num 可以改为强制查询使用索引：select id from t with(index(索引名)) where num=@num
f. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如： select id from t where num/2=100 应改为: select id from t where num=100*2
g. 应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如： select id from t where substring(name,1,3)=’abc’–name 以 abc 开头的 id select id from t where datediff(day,createdate,’2005-11-30′)=0–‘2005-11-30’生成的 id 应改为: select id from t where name like ‘abc%’ select id from t where createdate>=’2005-11-30′ and createdate<’2005-12-1′
h. 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。
i. 不要写一些没有意义的查询，如需要生成一个空表结构： select col1,col2 into #t from t where 1=0 这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：create table #t(…)
j. 很多时候用 exists 代替 in 是一个好的选择： select num from a where num in(select num from b) 用下面的语句替换： select num from a where exists(select 1 from b where num=a.num)
k. 任何地方都不要使用 select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。
l. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过 1 万行，那么就应该考虑改写。
m. 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。
n. 尽量避免大事务操作，提高系统并发能力

7 输入一个 URL 到看到结果，从输入网址到显示网页的过程，涉及了哪些协议。（这个我把牵扯到的知识点都写进去了，回答的时候只说大的点就好了）（计算机网络）

DNS 解析-->建立连接
发送数据包 -->服务器响应请求
返回给浏览器-->浏览器渲染程序页面。

1.DNS 解析

当我搜索这个问题的时候，首先在浏览器输入了一个 URL 地址，但 URL 中服务器地址是一个域名而不是一个指定的 IP 地址，路由器并不知道你想要查找的地址，那么 DNS 域名解析系统会将该域名解析成 ip，而 IP 地址是唯一的， 每一个 ip 地址对应网络上的一台计算机。

2.建立网络连接，发送数据包

由于 1 的努力，已经能够根据 ip 和端口号与网络上对应的服务器建立连接，浏览器这边会向服务器发送一个数据包，里面包含了大量的信息，但这个数据包有一定的格式。就像我给你邮个快递，也得遵循邮递公司的一些规则吧！我得写上我的身份信息、寄的物品、标明邮递地址....道理是一样的，到了网络中这些规则就是“Http 协议(网络协议)”。

HTTP 是一个属于应用层的面向对象的协议，HTTP 协议一共有五大特点：1、支持客户/服务器模式（C/S 模式）；2、简单快速；3、灵活；4、无连接；5、无状态。

无连接

无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

早期这么做的原因是 HTTP 协议产生于互联网，因此服务器需要处理同时面向全世界数十万、上百万客户端的网页访问，但每个客户端（即浏览器）与服务器之间交换数据的间歇性较大（即传输具有突发性、瞬时性），并且网页浏览的联想性、发散性导致两次传送的数据关联性很低，大部分通道实际上会很空闲、无端占用资源。因此 HTTP 的设计者有意利用这种特点将协议设计为请求时建连接、请求完释放连接，以尽快将资源释放出来服务其他客户端。随着时间的推移，网页变得越来越复杂，里面可能嵌入了很多图片，这时候每次访问图片都需要建立一次 TCP 连接就显得很低效。后来，Keep-Alive 被提出用来解决这效率低的问题。

Keep-Alive 功能使客户端到服务器端的连接持续有效（**长连接**），当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接。市场上的大部分 Web 服务器，包括 iPlanet、IIS 和 Apache，都支持 HTTP Keep-Alive。对于提供静态内容的网站来说，这个功能通常很有用。但是，对于负担较重的网站来说，这里存在另外一个问题：虽然为客户保留打开的连接有一定的好处，但它同样影响了性能，因为在处理暂停期间，本来可以释放的资源仍旧被占用。当 Web 服务器和应用服务器在同一台机器上运行时，Keep-Alive 功能对资源利用的影响尤其突出。 这样一来，客户端和服务器之间的 HTTP 连接就会被保持，不会断开（超过 Keep-Alive 规定的时间，意外断电等情况除外），当客户端发送另外一个请求时，就使用这条已经建立的连接。

无状态

无状态是指协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。即我们给服务器发送 HTTP 请求之后，服务器根据请求，会给我们发送数据过来，但是，发送完，不会记录任何信息。HTTP 是一个无状态协议，这意味着每个请求都是独立的，Keep-Alive 没能改变这个结果。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。HTTP 协议这种特性有优点也有缺点，优点在于解放了服务器，每一次请求“点到为止”不会造成不必要连接占用，缺点在于每次请求会传输大量重复的内容信息。

客户端与服务器进行动态交互的 Web 应用程序出现之后，HTTP 无状态的特性严重阻碍了这些应用程序的实现，毕竟交互是需要承前启后的，简单的购物车程序也要知道用户到底在之前选择了什么商品。于是，两种**用于保持 HTTP 连接状态**的技术就应运而生了，一个是 **Cookie**，而另一个则是 **Session**。

Cookie 可以保持登录信息到用户下次与服务器的会话，换句话说，下次访问同一网站时，用户会发现不必输入用户名和密码就已经登录了（当然，不排除用户手工删除 Cookie）。而还有一些 Cookie 在用户退出会话的时候就被删除了，这样可以有效保护个人隐私。

Cookies 最典型的应用是判定注册用户是否已经登录网站，用户可能会得到提示，是否在下一次进入此网站时保留用户信息以便简化登录手续，这些都是 Cookies 的功用。另一个重要应用场合是“购物车”之类处理。用户可能会在一段时间内在同一家网站的不同页面中选择不同的商品，这些信息都会写入 Cookies，以便在最后付款时提取信息。

与 Cookie 相对的一个解决方案是 **Session**，它是**通过服务器来保持状态**的。

当客户端访问服务器时，服务器根据需求设置 Session，将会话信息保存在服务器上，同时将标示 Session 的 SessionId 传递给客户端浏览器，浏览器将这个 SessionId 保存在内存中，我们称之为无过期时间的 Cookie。浏览器关闭后，这个 Cookie 就会被清掉，它不会存在于用户的 Cookie 临时文件。以后浏览器每次请求都会额外加上这个参数值，服务器会

根据这个 SessionId，就能取得客户端的数据信息。
如果客户端浏览器意外关闭，服务器保存的 Session 数据不是立即释放，此时数据还会存在，只要我们知道那个 SessionId，就可以继续通过请求获得此 Session 的信息，因为此时后台的 Session 还存在，当然我们可以设置一个 Session 超时时间，一旦超过规定时间没有客户端请求时，服务器就会清除对应 SessionId 的 Session 信息。

什么是 http 协议？它的**主要内容**。

http 协议（超文本传输协议）是客户端和服务器端两者通信共同遵循的一些规则。主要内容是定义了客户端如何向服务器请求资源，服务器如何响应客户端请求。

请求中的 POST 与 GET 方法有什么区别？

1.根据 HTTP 规范，**GET 用于信息获取**，而且应该是安全的,这里的安全是指非修改的信息，就像在数据库执行查询一样，不会修改数据，也不会增加删减数据，不会影响资源的状态，而 **post 可能会改变数据**的原始状态。

2.GET 提交的数据最多只有 1024 字节，理论上 POST 是没有限制的。

3.从请求的 URL 中可以找到一个区别：GET 请求的数据会附在 URL 之后，在浏览器 URL 栏就能看的。似乎 POST 比 GET 更可靠一些，因为它请求把提交的数据放在包体中，地址栏上不可见。（也有的解释说两者都没有长度限制，根本的区别就是一个是获取数据，一个修改数据！）

HTTP 定义了与服务器交互的不同方法，最基本的方法是 GET 和 POST.

HTTP-GET 和 HTTP-POST 是使用 HTTP 的标准协议动词，用于编码和传送变量名/变量值对参数，并且使用相关的请求语义。每个 HTTP-GET 和 HTTP-POST 都由一系列 HTTP 请求头组成，这些请求头定义了客户端从服务器请求了什么，而响应则是由一系列 HTTP 应答头和应答数据组成，如果请求成功则返回应答。

HTTP-GET 以使用 MIME 类型 application/x-www-form-urlencoded 的 urlencoded 文本的格式传递参数。Urlencoding 是一种字符编码，保证被传送的参数由遵循规范的文本组成，例如一个空格的编码是"%20"。附加参数还能被认为是一个查询字符串。

与 HTTP-GET 类似，HTTP-POST 参数也是被 URL 编码的。然而，变量名/变量值不作为 URL 的一部分被传送，而是放在实际的 HTTP 请求消息内部被传送。

（1）get 是从服务器上获取数据，post 是向服务器传送数据。

（1） 在客户端，Get 方式在通过 URL 提交数据，数据在 URL 中可以看到；POST 方式，数据放置在 HTML HEADER 内提交。

（2） 对于 get 方式，服务器端用 Request.QueryString 获取变量的值，对于 post 方式，服务器端用 Request.Form 获取提交的数据。

（2） GET 方式提交的数据最多只能有 1024 字节，而 POST 则没有此限制。

（3） 安全性问题。正如在（1）中提到，使用 Get 的时候，参数会显示在地址栏上，而 Post 不会。所以，如果这些数据是中文数据而且是非敏感数据，那么使用 get；如果用户输入的数据不是中文字符而且包含敏感数据，那么还是使用 post 为好。

注:所谓的安全意味着该操作用于获取信息而非修改信息。幂等的意味着对同一 URL 的多个请求应该返回同样的结果。完整的定义并不像看起来那样严格。换句话说，GET 请求一般不应产生副作用。从根本上讲，其目标是当用户打开一个链接时，她可以确信从自身的角度来看没有改变资源。比如，新闻站点的头版不断更新。虽然第二次请求会返回不同的一批新闻，该操作仍然被认为是安全的和幂等的，因为它总是返回当前的新闻。反之亦然。POST 请求就不那么轻松了。POST 表示可能改变服务器上的资源的请求。仍然以新闻站点为例,读者对文章的注解应该通过 POST 请求实现,因为在注解提交之后站点已经不同了(比方说文章下面出现一条注解）。

3.服务器响应请求，返回给浏览器

服务器会分解你的数据包，例如你查找的是一个文档，那么服务器可能会返回一个 doc 文档或者 zip 压缩资源给你；如果你访问的是一个链接页面，那么服务器相应的返回一个包含 HTML/CSS 标记文档，这些请求和响应都有一个通用的写法，这些规则也就是前面提到的"http 协议"。

客户端向服务器请求资源时，除了告诉服务器要请求的资源，同时还会附带一些其他的信息，这部分信息放在"header"部分（服务器响应请求也一样！），主要有请求头(略)和响应头，这里以响应头部信息为例：

看到 **http 响应状态码**我突然想到了 404,==！顺便带几个常见的：
状态码的职责是当客户端向服务器端发送请求时，描述返回的请求结果。借助于状态码，浏览器（或者说用户）可以知道服务器是正常的处理了请求，还是出现了错误。
数字的第一位指定了响应类型，后两位无分类。响应类别一共有 5 种：
1XX Informational(信息性状态码)
2XX Success(成功状态码)
3XX Redirection(重定向状态码)
4XX Client Error(客户端错误状态码)
5XX Server Error(服务器错误状态码)
HTTP 响应状态码有很多，但是实际经常使用的大概只有 14 个。

2XX Success

200 OK 表示从客户端发来的请求在服务器端被正常处理了。
204 No Content 该状态码表示服务器接收的请求已成功处理，但在返回的响应报文中不含实体的主体部分。比如，当从浏览器发出请求处理后，返回 204 响应，那么浏览器显示的页面不发生更新。
206 Partial Content 该状态码表示客户端进行了范围请求，而服务器成功执行了这部分的 GET 请求。

3XX Redirection

301 Moved Permanently 永久性重定向。该状态码表示请求的资源已经被分配了新的 URI，以后应使用资源现在所指的 URI。 像下方给出的请求 URI，当指定的资源路径的最后忘记添加斜杠"/"，就会产生 301 状态码
http://example.com/sample
302 Found 临时性重定向。该状态码表示请求的资源已被分配了新的 URI，希望用户(本次)能使用新的 URI 访问。
303 See Other 该状态码表示由于请求对应的资源存在另外一个 URI，应使用 GET 方法定向获取请求的资源。 303 状态码和 302 状态码有着相同的功能，但 303 状态码明确表明客户端应当采用 GET 方法获取资源。 当 301，302，303 响应状态码返回时，几乎所有的浏览器都会把 POST 改成 GET，并删除请求报文的主体，之后请求会自动再次发送。 301，302 标准是禁止将 POST 方法改变成 GET 方法的，但实际上使用时大家都会这么做。
304 Not Modified 该状态码表示客户端发送附带条件的请求时，服务器端允许请求访问资源，但未满足条件的情况。304 状态码返回时，不包含任何响应的主体部分。304 虽然被划分在 3XX 类别中，但是和重定向没有关系。
307 Temporary Redirect 临时重定向。该状态码与 302 Found 有着相同的含义。307 会遵照浏

览器标准，不会从 POST 变成 GET。

4XX Client Error

400 Bad Request 该状态码表示请求报文中存在语法错误。当错误发生时，需要修改请求的内容后再次放松请求。

401 Unauthorized 该状态码表示发送的请求需要有通过 HTTP 认证的认证信息，另外若之前已进行过 1 此请求，则表示用户认证失败。

403 Forbidden 该状态码表明对请求资源的访问被服务器拒绝了。

404 Not Found 该状态码表明服务器上无法找到请求的资源。除此之外，也可以在服务器端拒绝请求且不想说明理由时使用。

5XX Server Error

500 Internal Server Error 该状态码表明服务器端在执行请求时发生了错误。

503 Service Unavailable 该状态码表明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

常见状态码

100 Continue 继续，一般在发送 post 请求时，已发送了 http、header 之后服务端将返回此信息，表示确认，之后发送具体参数信息

200 OK 正常返回信息

201 Created 请求成功并且服务器创建了新的资源

301 Moved Permanently 请求的网页已永久移动到新位置。

400 Bad Request 服务器无法理解请求的格式，客户端不应当尝试再次使用相同的内容发起请求。

404 Not Found 找不到如何与 URI 相匹配的资源。

500 Internal Server Error 最常见的服务器端错误。

4.浏览器渲染呈现

浏览器拿到响应的页面代码，将其解析呈现在用户面前，至于为什么会是看到的这个样子，有时又是另外的一些页面效果，这里就涉及到 web 标准了，也就是我们经常提到的 w3c 标准。根据资源的类型，在网页上呈现给用户，这个过程叫网页渲染。解析和呈现的过程主要由浏览器的渲染引擎实现，浏览器的渲染引擎质量就决定了浏览器的好坏（引擎这一块已经超出了我的理解范围了）。

但实际上输入 URL 到页面呈现这背后涉及的内容远远不止这些，例如后台 web 服务器、双向的网络数据传输、http 缓存策略等

涉及协议：

（1）应用层 HTTP(www 访问协议)，DNS（域名解析服务）

（2）传输层 TCP（为 HTTP 提供可靠的数据传输），UDP（DNS 使用 UDP 传输）

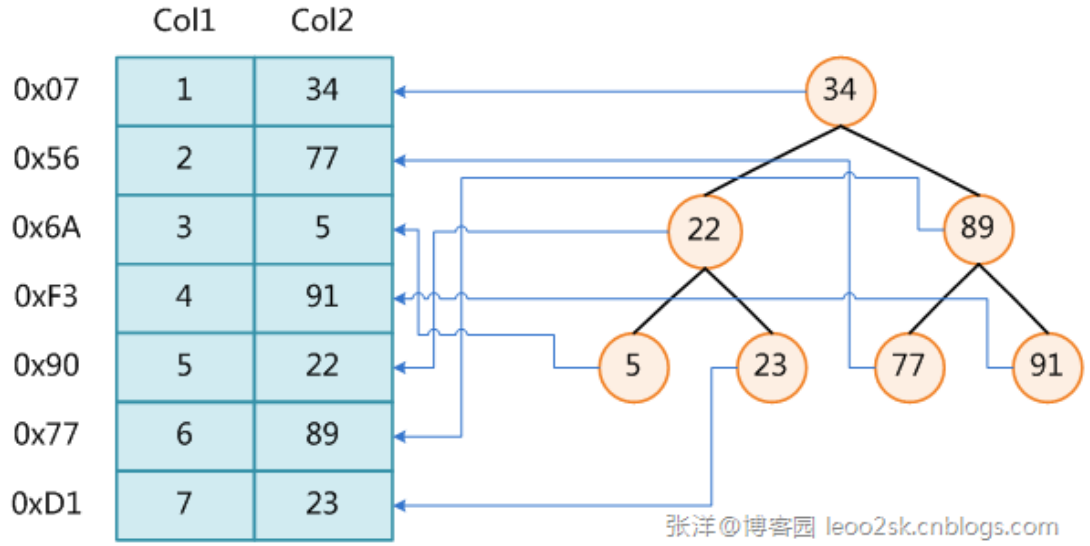
（3）网络层：IP（IP 数据包传输和路由选择），ICMP（提供网络传输过程中的差错检测），ARP（将本机的默认网关 IP 地址映射成物理 MAC 地址）

8 数据库也问了索引（数据库）

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变种 B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



上图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快 Col2 的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在 O(log2n)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。（优点）

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

增加索引也有许多不利的方面。（缺点）

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，**应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在 WHERE 子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。一般来说，**不应该创建索引的的这些列具有下列特点：**

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降

低了系统的维护速度和增大了空间需求。
第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。
第三，对于那些定义为 text, image 和 bit 数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。
第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

索引的实现方式（数据库）

1 **B+树**　　我们经常听到 B+树就是这个概念，用这个树的目的和红黑树差不多，也是为了尽量保持树的平衡，当然红黑树是二叉树，但 B+树就不是二叉树了，节点下面可以有多个子节点，数据库开发商会设置子节点数的一个最大值，这个值不会太小，所以 B+树一般来说比较矮胖，而红黑树就比较瘦高了。关于 B+树的插入，删除，会涉及到一些算法以保持树的平衡，这里就不详述了。ORACLE 的默认索引就是这种结构的。如果经常需要同时对两个字段进行 AND 查询,那么使用两个单独索引不如建立一个复合索引，因为两个单独索引通常数据库只能使用其中一个，而使用复合索引因为索引本身就对应到两个字段上的，效率会有很大提高。

2 **散列索引**　　第二种索引叫做散列索引，就是通过散列函数来定位的一种索引，不过很少有单独使用散列索引的，反而是散列文件组织用的比较多。散列文件组织就是根据一个键通过散列计算把对应的记录都放到同一个槽中，这样的话相同的键值对应的记录就一定是放在同一个文件里了，也就减少了文件读取的次数，提高了效率。散列索引呢就是根据对应键的散列码来找到最终的索引项的技术，其实和 B 树就差不多了，也就是一种索引之上的二级辅助索引，我理解散列索引都是二级或更高级的稀疏索引，否则桶就太多了，效率也不会很高。

3 **位图索引**　　位图索引是一种针对多个字段的简单查询设计一种特殊的索引，适用范围比较小，只适用于字段值固定并且值的种类很少的情况，比如性别，只能有男和女，或者级别，状态等等，并且只有在同时对多个这样的字段查询时才能体现出位图的优势。位图的基本思想就是对每一个条件都用 0 或者 1 来表示，如有 5 条记录，性别分别是男，女，男，男，女，那么如果使用位图索引就会建立两个位图，对应男的 10110 和对应女的 01001,这样做有什么好处呢，就是如果同时对多个这种类型的字段进行 and 或 or 查询时，可以使用按位与和按位或来直接得到结果了。

B+树最常用，性能也不差，用于范围查询和单值查询都可以。特别是范围查询，非得用 B+树这种顺序的才可以了。HASH 的如果只是对单值查询的话速度会比 B+树快一点，但是 ORACLE 好像不支持 HASH 索引，只支持 HASH 表空间。位图的使用情况很局限，只有很少的情况才能用，一定要确定真正适合使用这种索引才用（值的类型很少并且需要复合查询），否则建立一大堆位图就一点意义都没有了。

聚集索引和非聚集索引的区别？（数据库）

聚集索引和非聚集索引的根本区别是表中记录的排列顺序与索引的排列顺序是否一致。

聚集索引（CLUSTER INDEX）是指索引项的顺序与表中记录的物理顺序一致的索引组织。优点是查询速度快，因为一旦具有第一个索引值的纪录被找到，具有连续索引值的记录也一定物理的紧跟其后。用户可以在最经常查询的列上建立聚簇索引以提高查询效率。在一个基本表上最多只能建立一个聚簇索引。聚集索引的缺点是对表进行修改速度较慢，这是为了保持表中的记录的物理顺序与索引的顺序一致，而把记录插入到数据页

的相应位置，必须在数据页中进行数据重排，降低了执行速度。建议使用聚集索引的场合为：

a.此列包含有限数目的不同值；

b.查询的结果返回一个区间的值；

c.查询的结果返回某值相同的大量结果集。

非聚集索引指定了表中记录的逻辑顺序，但记录的物理顺序和索引的顺序不一致，聚集索引和非聚集索引都采用了 B+树的结构，但非聚集索引的叶子层并不与实际的数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针的方式。非聚集索引比聚集索引层次多，添加记录不会引起数据顺序的重组。建议使用非聚集索引的场合为：

a.此列包含了大量数目不同的值；

b.查询的结束返回的是少量的结果集；

c.order by 子句中使用了该列。

9 优先队列（堆）的时间复杂度（数据结构）

优先队列(priority queue, 以下简称 PQ)，PQ 又叫"堆"(heap), 但是可能优先队列这个名字更容易记忆它的用途: PQ 是一种队列, 不过不是先进先出(FIFO), 而是每次出队的元素永远是优先级最高的.PQ 是一种树(tree), 准确的说, 是一种二叉树(binary tree), 说得再准确一点, 它是一种完全二叉树(complete binary tree): 没错, PQ 是一种满足某些条件的完全二叉树.

所谓的"**完全二叉树**", 要满足:

1. 除了最后一层, 所有层都排满(没有非空节点)
2. 最后一层的所有非空节点都排在左边

一个完全二叉树能被成为 PQ 的话, 要满足的条件就是:对于任何一个节点, 它的优先级都大于左右子节点的优先级.

对于一个完全二叉树, 没有必要用常规的树结构(使用指针)来表示, 因为如果从上到下走过每层(每层内从左到右)给所有节点编号(根节点的编号为 1)的话, 完全二叉树有以下性质:

father(i) = i/2 其中 father(i)表示编号为 i 的节点的父节点的下标

leftchild(i) = i*2, rightchild(i) = i*2+1

既然用数组表示的时候, 任何节点的父亲节点和左右子节点都可以轻松得到, 就没有必要使用指针了. 所以只需要一个数组即可表示 PQ ！比如一个 int 的 PQ 在内部只要表示为: int pq[].另外注意, 上面的公式成立的前提是数组下标从 1 开始, 实现的时候我们把数组的第 0 个元素空出来即可.

不难看出一个 **PQ 有以下性质**:

1. 高度为 lgN
2. 第 k 层有 2^k 个节点 (root 是第 0 层)
3. 最后一层的节点对应的下标>=N/2

创建堆的算法。把 n 个元素建立一个堆，首先我可以将这 n 个结点以自顶向下、从左到右的方式从 1 到 n 编码。这样就可以把这 n 个结点转换成为一棵完全二叉树。紧接着从最后一个非叶结点（结点编号为 n/2）开始到根结点（结点编号为 1），逐个扫描所有的结点，根据需要将当前结点向下调整，直到以当前结点为根结点的子树符合堆的特性。虽然讲起来起来很复杂，但是实现起来却很简单，用这种方法来建立一个堆的时间**复杂度是 O(N)**

用途 1：堆还有一个作用就是**堆排序**，与快速排序一样**堆排序**的时间**复杂度也是 O(NlogN)**。堆排序的实现很简单，比如我们现在要进行从小到大排序，可以先建立最小堆，然后每次删除顶部元素并将顶部元素输出或者放入一个新的数组中，直到堆为空为止。最终输出的或者存放在新数组中数就已经是排序好的了。

用途 2：堆还经常被用来求一个数列中**第 K 大的数**。只需要建立一个大小为 K 的最小堆，

堆顶就是第 K 大的数。如果求一个数列中第 K 小的数，只最需要建立一个大小为 K 的最大堆，堆顶就是第 K 小的数，这种方法的时间**复杂度是 O(NlogK)**。当然你也可以用堆来求前 K 大的数和前 K 小的数。堆调整的时间复杂度是 O(log n)

MAX-HEAPIFY（维护堆）：用于维护最大堆的性质，时间复杂度为 O(logn)。

BUILD-MAX-HEAP（建最大堆）：用于把无序的输入数据数组构造为一个最大堆，时间复杂度为 O(n)。

HEAPSORT（堆排序）：对一个数组进行原址排序，时间复杂度为 O(nlogn)。

MAX-HEAP-INSERT 、 HEAP-EXTRACT-MAX 、 HEAP-INCREASE-KEY 和 HEAP-MAXIMUM：利用这些操作可以实现一个优先队列，时间复杂度为 O(nlogn)。

10. 关系型数据库 & 非关系型数据库（数据库）

关系型数据库，是指采用了关系模型来组织数据的数据库。 简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。

关系模型中常用的概念：

关系：可以理解为一张二维表，每个关系都具有一个关系名，就是通常说的表名

元组：可以理解为二维表中的一行，在数据库中经常被称为记录

属性：可以理解为二维表中的一列，在数据库中经常被称为字段

域：属性的取值范围，也就是数据库中某一列的取值限制

关键字：一组可以唯一标识元组的属性，数据库中常称为主键，由一个或多个列组成

关系模式：指对关系的描述。其格式为：关系名(属性 1，属性 2， ... ... ，属性 N)，在数据库中成为表结构

关系型数据库的优点：

容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解

使用方便：通用的 SQL 语言使得操作关系型数据库非常方便

易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率

关系型数据库瓶颈

1). 高并发读写需求

网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘 I/O 是一个很大的瓶颈

2). 海量数据的高效率读写

网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询，效率是非常低的

3). 高扩展性和可用性

在基于 web 的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像 web server 和 app server 那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供 24 小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移。

非关系型数据库提出另一种理念，例如，以键值对存储，且结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销。使用这种方式，用户可以根据需要去添加自己需要的字段，这样，为了获取用户的不同信息，不需要像关系型数据库中，要对多表进行关联查询。仅需要根据 id 取出相应的 value 就可以完成查询。但非关系型数据库由于很少的约

束，他也不能够提供像 SQL 所提供的 where 这种对于字段属性值情况的查询。并且难以体现设计的完整性。他只适合存储一些较为简单的数据，对于需要进行较复杂查询的数据，SQL 数据库显的更为合适。

依据结构化方法以及应用场合的不同，主要分为以下几类：

1).面向高性能并发读写的 key-value 数据库：key-value 数据库的主要特点就是具有极高的并发读写性能，Redis,Tokyo Cabinet,Flare 就是这类的代表

2).面向海量数据访问的面向文档数据库：这类数据库的特点是，可以在海量的数据中快速的查询数据，典型代表为 MongoDB 以及 CouchDB

3).面向可扩展性的分布式数据库：这类数据库想解决的问题就是传统数据库存在可扩展性上的缺陷，这类数据库可以适应数据量的增加以及数据结构的变化

关系型数据库 V.S. 非关系型数据库

关系型数据库的最大特点就是事务的一致性：传统的关系型数据库读写操作都是事务的，具有 ACID 的特点，这个特性使得关系型数据库可以用于几乎所有对一致性有要求的系统中，如典型的银行系统。

但是，在网页应用中，尤其是 SNS 应用中，一致性却不是显得那么重要，用户 A 看到的内容和用户 B 看到同一用户 C 内容更新不一致是可以容忍的，或者说，两个人看到同一好友的数据更新的时间差那么几秒是可以容忍的，因此，关系型数据库的最大特点在这里已经无用武之地，起码不是那么重要了。

相反地，关系型数据库为了维护一致性所付出的巨大代价就是其读写性能比较差，而像微博、facebook 这类 SNS 的应用，对并发读写能力要求极高，关系型数据库已经无法应付(在读方面，传统上为了克服关系型数据库缺陷，提高性能，都是增加一级 memcache 来静态化网页，而在 SNS 中，变化太快，memchache 已经无能为力了)，因此，必须用新的一种数据结构存储来代替关系数据库。

关系数据库的另一个特点就是其具有固定的表结构，因此，其扩展性极差，而在 SNS 中，系统的升级，功能的增加，往往意味着数据结构巨大变动，这一点关系型数据库也难以应付，需要新的结构化数据存储。

于是，非关系型数据库应运而生，由于不可能用一种数据结构化存储应付所有的新的需求，因此，非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

必须强调的是，数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库这员老将。

11. 数据库事务（数据库）

所谓**事务**是用户定义的一个数据库操作序列，这些操作要么全部执行，要么全部不执行，是一个不可分割的工作单位。例如在关系数据库中，一个事务可以是一条 sql 语句、一组 sql 语句或整个程序。

事务的 ACID 特性

A(Atomicity)原子性

事务必须是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行。通常，与某个事务关联的操作具有共同的目标，并且是相互依赖的。如果系统只执行这些操作的一个子集，则可能会破坏事务的总体目标。原子性消除了系统处理操作子集的可能性。

C(Consistency)一致性

事务在完成时，必须使所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。事务结束时，所有的内部数据结构（如 B 树索引或双向链表）都必须是正确的。某些维护一致性的责任由应用程序开发人员承担，他们必须确保应用程序已强制所有已知的完整性约束。例如，当开发用于转帐的应用程序时，应避免在转帐过程中任意移动小数点。

I(Isolation)隔离性

指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看到中间状态的数据。

D(Durability)持久性

指的是只要事务成功结束，它对数据库所做的更新就必须永久保存下来。即使发生系统崩溃，重新启动数据库系统后，数据库还能恢复到事务成功结束时的状态。

事务的 ACID 特性是由关系数据库管理系统（RDBMS，数据库系统）来实现的。数据库管理系统采用日志来保证事务的原子性、一致性和持久性。日志记录了事务对数据库所做的更新，如果某个事务在执行过程中发生错误，就可以根据日志，撤销事务对数据库已做的更新，使数据库退回到执行事务前的初始状态。数据库管理系统采用锁机制来实现事务的隔离性。当多个事务同时更新数据库中相同的数据时，只允许持有锁的事务更新数据，其他事务必须等待，直到前一个事务释放了锁，其他事务才有机会更新该数据。

12. 数据库锁（数据库）

数据库和操作系统一样，是一个多用户使用的共享资源。当多个用户并发地存取数据时，在数据库中就会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数据库的一致性。加锁是实现数据库并发控制的一个非常重要的技术。在实际应用中经常会遇到的与锁相关的异常情况，当两个事务需要一组有冲突的锁，而不能将事务继续下去的话，就会出现死锁，严重影响应用的正常执行。

在数据库中有两种基本的锁类型：**排它锁**（Exclusive Locks，即 **X 锁**）和**共享锁**（Share Locks，即 **S 锁**）。当数据对象被加上排它锁时，其他的事务不能对它读取和修改。加了共享锁的数据对象可以被其他事务读取，但不能修改。数据库利用这两种基本的锁类型来对数据库的事务进行并发控制。

死锁的几种情况

死锁的第一种情况

一个用户 A 访问表 A(锁住了表 A),然后又访问表 B；另一个用户 B 访问表 B(锁住了表 B)，然后企图访问表 A；这时用户 A 由于用户 B 已经锁住表 B，它必须等待用户 B 释放表 B 才能继续，同样用户 B 要等用户 A 释放表 A 才能继续，这就死锁就产生了。

解决方法：

这种死锁比较常见，是由于程序的 BUG 产生的，除了调整的程序的逻辑没有其它的办法。仔细分析程序的逻辑，对于数据库的多表操作时，尽量按照相同的顺序进行处理，尽量避免同时锁定两个资源，如操作 A 和 B 两张表时，总是按先 A 后 B 的顺序处理，必须同时锁定两个资源时，要保证在任何时刻都应该按照相同的顺序来锁定资源。

死锁的第二种情况

用户 A 查询一条纪录，然后修改该条纪录；这时用户 B 修改该条纪录，这时用户 A 的事务

里锁的性质由查询的共享锁企图上升到独占锁，而用户 B 里的独占锁由于 A 有共享锁存在所以必须等 A 释放掉共享锁，而 A 由于 B 的独占锁而无法上升的独占锁也就不可能释放共享锁，于是出现了死锁。这种死锁比较隐蔽，但在稍大点的项目中经常发生。如在某项目中，页面上的按钮点击后，没有使按钮立刻失效，使得用户会多次快速点击同一按钮，这样同一段代码对数据库同一条记录进行多次操作，很容易就出现这种死锁的情况。

解决方法：

1、对于按钮等控件，点击后使其立刻失效，不让用户重复点击，避免对同时对同一条记录操作。

2、使用乐观锁进行控制。乐观锁大多是基于数据版本（Version）记录机制实现。即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数 据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。乐观锁机制避免了长事务中的数据库加锁开销（用户 A 和用户 B 操作过程中，都没有对数据库数据加锁），大大提升了大并发量下的系统整体性能表现。Hibernate 在其数据访问引擎中内置了乐观锁实现。需要注意的是，由于乐观锁机制是在我们的系统中实现，来自外部系统的用户更新操作不受我们系统的控制，因此可能会造成脏数据被更新到数据库中。

3、使用悲观锁进行控制。悲观锁大多数情况下依靠数据库的锁机制实现，如 Oracle 的 Select … for update 语句，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。如一个金融系统， 当某个操作员读取用户的数据，并在读出的用户数据的基础上进行修改时（如更改用户账户余额），如果采用悲观锁机制，也就意味着整个操作过程中（从操作员读 出数据、开始修改直至提交修改结果的全过程，甚至还包括操作员中途去煮咖啡的时间），数据库记录始终处于加锁状态，可以想见，如果面对成百上千个并发，这样的情况将导致灾难性的后果。所以，采用悲观锁进行控制时一定要考虑清楚。

死锁的第三种情况

如果在事务中执行了一条不满足条件的 update 语句，则执行全表扫描，把行级锁上升为表级锁，多个这样的事务执行后，就很容易产生死锁和阻塞。类似的情况还有当表中的数据量非常庞大而索引建的过少或不合适的时候，使得经常发生全表扫描，最终应用系统会越来越慢，最终发生阻塞或死锁。

解决方法：

SQL 语句中不要使用太复杂的关联多表的查询；使用“执行计划”对 SQL 语句进行分析，对于有全表扫描的 SQL 语句，建立相应的索引进行优化。

总体上来说，产生内存溢出与锁表都是由于代码写的不好造成的，因此提高代码的质量是最根本的解决办法。有的人认为先把功能实现，有 BUG 时再在测试阶段进 行修正，这种想法是错误的。正如一件产品的质量是在生产制造的过程中决定的，而不是质量检测时决定的，软件的质量在设计与编码阶段就已经决定了，测试只是对软件质量的一个验证，因为测试不可能找出软件中所有的 BUG。

如何避免死锁（数据库）

1 使用事务时，尽量缩短事务的逻辑处理过程，及早提交或回滚事务；

2 设置死锁超时参数为合理范围，如：3 分钟-10 分种；超过时间，自动放弃本次操作，避免进程悬挂；

3 所有的 SP 都要有错误处理（通过@error）

4 一般不要修改 SQL SERVER 事务的默认级别。不推荐强行加锁

5 优化程序，检查并避免死锁现象出现；
1）合理安排表访问顺序
2）在事务中尽量避免用户干预，尽量使一个事务处理的任务少些。
3）采用脏读技术。脏读由于不对被访问的表加锁，而避免了锁冲突。在客户机/服务器应用环境中，有些事务往往不允许读脏数据，但在特定的条件下，我们可以用脏读。
4）数据访问时域离散法。数据访问时域离散法是指在客户机/服务器结构中，采取各种控制手段控制对数据库或数据库中的对象访问时间段。主要通过以下方式实现: 合理安排后台事务的执行时间，采用工作流对后台事务进行统一管理。工作流在管理任务时，一方面限制同一类任务的线程数（往往限制为 1 个），防止资源过多占用; 另一方面合理安排不同任务执行时序、时间，尽量避免多个后台任务同时执行，另外，避免在前台交易高峰时间运行后台任务
5）数据存储空间离散法。数据存储空间离散法是指采取各种手段，将逻辑上在一个表中的数据分散到若干离散的空间上去，以便改善对表的访问性能。主要通过以下方法实现: 第一，将大表按行或列分解为若干小表; 第二，按不同的用户群分解。
6）使用尽可能低的隔离性级别。隔离性级别是指为保证数据库数据的完整性和一致性而使多用户事务隔离的程度，SQL92 定义了 4 种隔离性级别：未提交读、 提交读、可重复读和可串行。如果选择过高的隔离性级别，如可串行，虽然系统可以因实现更好隔离性而更大程度上保证数据的完整性和一致性，但各事务间冲突而死锁的机会大大增加，大大影响了系统性能。
7）使用 Bound Connections。Bound connections 允许两个或多个事务连接共享事务和锁，而且任何一个事务连接要申请锁如同另外一个事务要申请锁一样，因此可以允许这些事务共享数据而不会有加锁的冲突。
8）考虑使用乐观锁定或使事务首先获得一个独占锁定。

冲突问题

1、脏读

某个事务读取的数据是另一个事务正在处理的数据。而另一个事务可能会回滚，造成第一个事务读取的数据是错误的。

2、不可重复读

在一个事务里两次读入数据，但另一个事务已经更改了第一个事务涉及到的数据，造成第一个事务读入旧数据。

3、幻读

幻读是指当事务不是独立执行时发生的一种现象。例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。

4、更新丢失

多个事务同时读取某一数据，一个事务成功处理好了数据，被另一个事务写回原值，造成第一个事务更新丢失。

锁模式（数据库）

1、共享锁

共享锁（S 锁）允许并发事务在封闭式并发控制下读取 (SELECT)资源。有关详细信息，请参阅并发控制的类型（悲观锁和乐观锁）。资源上存在共享锁（S 锁）时，任何其他事务都不能修改数据。读取操作一完成，就立即释放资源上的共享锁（S 锁），除非将事务隔离级别设置为可重复读或更高级别，或者在事务持续时间内用锁定提示保留共享锁（S 锁）。

2、更新锁（U 锁）

更新锁在共享锁和排他锁的结合。更新锁意味着在做一个更新时，一个共享锁在扫描完成符合条件的数据后可能会转化成排他锁。

这里面有两个步骤：

1) 扫描获取 Where 条件时。这部分是一个更新查询，此时是一个更新锁。

2) 如果将执行写入更新。此时该锁升级到排他锁。否则，该锁转变成共享锁。

更新锁可以防止常见的死锁。

3、排他锁

排他锁（X 锁）可以防止并发事务对资源进行访问。排他锁不与其他任何锁兼容。使用排他锁（X 锁）时，任何其他事务都无法修改数据；仅在使用 NOLOCK 提示或未提交读隔离级别时才会进行读取操作。

悲观锁

悲观锁是指假设并发更新冲突会发生，所以不管冲突是否真的发生，都会使用锁机制。悲观锁会完成以下功能：锁住读取的记录，防止其它事务读取和更新这些记录。其它事务会一直阻塞，直到这个事务结束.

悲观锁是在使用了数据库的事务隔离功能的基础上，独享占用的资源，以此保证读取数据一致性，避免修改丢失。

悲观锁可以使用 Repeatable Read 事务，它完全满足悲观锁的要求。

乐观锁

乐观锁不会锁住任何东西，也就是说，它不依赖数据库的事务机制，乐观锁完全是应用系统层面的东西。

如果使用乐观锁，那么数据库就必须加版本字段，否则就只能比较所有字段，但因为浮点类型不能比较，所以实际上没有版本字段是不可行的。

12. 平衡二叉树（数据结构）

平衡二叉查找树，又称 AVL 树。它除了具备二叉查找树的基本特征之外，还具有一个非常重要的特点：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值（平衡因子）不超过 1。 也就是说 AVL 树每个节点的平衡因子只可能是-1、0 和 1（左子树高度减去右子树高度）。

使二叉查找树在添加数据的同时保持平衡呢？基本思想就是：当在二叉排序树中插入一个节点时，首先检查是否因插入而破坏了平衡，若破坏，则找出其中的最小不平衡二叉树，在保持二叉排序树特性的情况下，调整最小不平衡子树中节点之间的关系，以达到新的平衡。所谓最小不平衡子树指离插入节点最近且以平衡因子的绝对值大于 1 的节点作为根的子树。

平衡二叉树的性能**优势**：平衡二叉树的优势在于不会出现普通二叉查找树的最差情况。其查找的时间复杂度为 O(logN)。

平衡二叉树的**缺陷**：（1）为了保证高度平衡，动态插入和删除的代价也随之增加。

（2）所有二叉查找树结构的查找代价都与树高是紧密相关的，能否通过减少树高来进一步降低查找代价呢。

（3）在大数据量查找环境下(比如说系统磁盘里的文件目录，数据库中的记录查询等)，所有的二叉查找树结构(BST、AVL、RBT)都不合适。

13. B 树、B+树（数据结构）

动态查找树主要有：二叉查找树（Binary Search Tree），平衡二叉查找树（Balanced Binary Search Tree），红黑树(Red-Black Tree)，B-tree/B+-tree/ B*-tree (B~Tree)。前三者

是典型的二叉查找树结构，其查找的时间复杂度 O(log2N)与树的深度相关，那么降低树的深度自然会提高查找效率。

但是咱们有面对这样一个实际问题：就是大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘 I/O 读写过于频繁，进而导致查询效率低下，那么如何减少树的深度（当然是不能减少查询的数据量），一个基本的想法就是：采用多叉树结构（由于树节点元素数量是有限的，自然该节点的子树数量也就是有限的）。

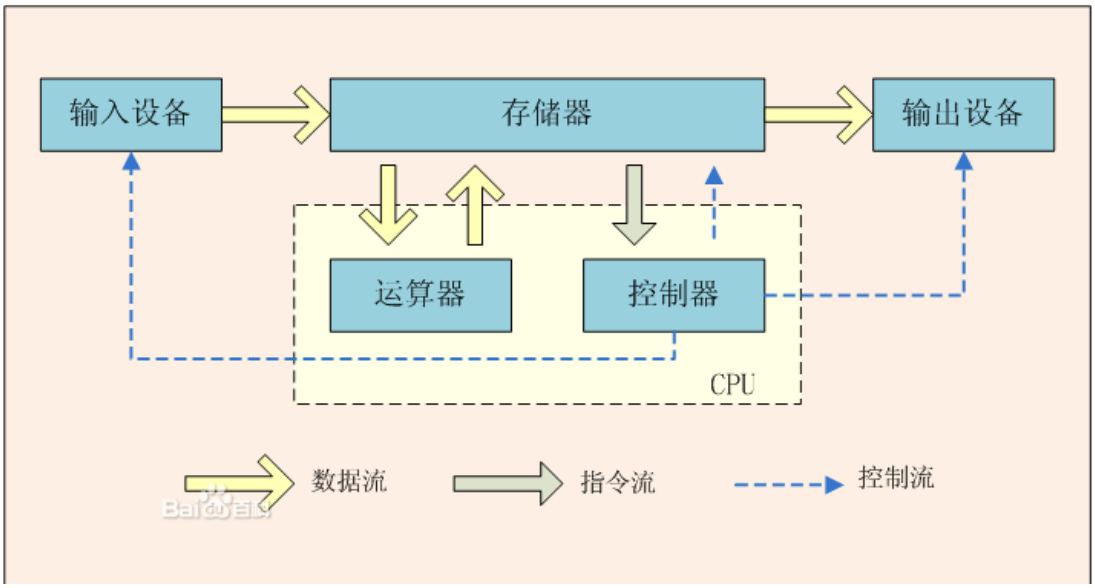
也就是说，因为磁盘的操作费时费资源，如果过于频繁的多次查找势必效率低下。那么如何提高效率，即如何避免磁盘过于频繁的多次查找呢？根据磁盘查找存取的次数往往由树的高度所决定，所以，只要我们通过某种较好的树结构改变树的结构尽量减少树的高度，那么是不是便能有效减少磁盘查找存取的次数呢？那这种有效的树结构是一种怎样的树呢？

这样我们就提出了一个新的查找树结构——多路查找树。根据平衡二叉树的启发，自然就想到平衡多路查找树结构，也就是这篇文章所要阐述的第一个主题 B~tree，即 B 树结构(后面，我们将看到，B 树的各种操作能使 B 树保持较低的高度，从而达到有效避免磁盘过于频繁的查找存取操作，从而有效提高查找效率)。

http://blog.csdn.net/v_JULY_v/article/details/6530142 July 博客：从 B 树、B+ 树、B* 树谈到 R 树（实在看不进去）

http://blog.csdn.net/v_july_v/article/details/6105630 July 博客：红黑树

14. 冯诺依曼体系分为几部分（计算机基础）



冯诺依曼体系结构有以下**特点**：

（1）计算机处理的数据和指令一律用二进制数表示；

（2）指令和数据不加区别混合存储在同一个存储器中；

（3）顺序执行程序的每一条指令；

（4）计算机硬件由运算器、控制器、存储器、输入设备和输出设备五大部分组成。

冯诺依曼体系结构的计算机必须具有如下功能：

（1）把需要的程序和数据送至计算机中；
（2）必须具有长期记忆程序、数据、中间结果及最终运算结果的能力；
（3）能够完成各种算术、逻辑运算和数据传送等数据加工处理的能力；
（4）能够根据需要控制程序走向，并能根据指令控制机器的各部件协调操作；
（5）能够按照要求将处理结果输出给用户。

15. CPU、指令（计算机基础）

中央处理器（英语：Central Processing Unit，缩写：CPU），是计算机的主要设备之一，功能主要是解释计算机指令以及处理计算机软件中的数据。计算机的可编程性主要是指对中央处理器的编程。

“**指令**”是由指令集架构定义的单个的 CPU 操作。在更广泛的意义上，“指令”可以是任何可执行程序的元素的表述，例如字节码。

16. **进程、线程（操作系统）**

进程是进程实体的运行过程，是系统进行**资源分配和调度**的一个独立单位。**线程**是进程的一个实体，是 **CPU 调度和分派**的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。**一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行**。**多进程**，允许多个任务同时运行；**多线程**，允许单个任务分成不同的部分运行；

进程和程序的区别：进程即运行中的程序，从中即可知，进程是在运行的，程序是非运行的，当然本质区别就是动态和静态的区别。程序可以存在外存中，也可以存在内存中。

进程和线程的关系：

（1）一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。
（2）资源分配给进程，同一进程的所有线程共享该进程的所有资源。
（3）处理机分给线程，即真正在处理机上运行的是线程。
（4）线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。线程是指进程内的一个执行单元,也是进程内的可调度实体.

进程与线程的区别：

（1）**调度**：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位
（2）**并发性**：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行
（3）**拥有资源**：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源.
（4）**系统开销**：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。

http://blog.csdn.net/mxsgoden/article/details/8821936 **进程和程序的区别**

进程间通信方式

进程间通信是指在并行计算过程中,各进程之间进行**数据交互**或**消息传递**, 其通信量的大小主要取决于并行设计的粒度划分和各个执行进程之间的相对独立性。也就是在多进程环境下,使用的数据交互、事件通知等方法使各进程协同工作。

进程间通信是同一台处理机或不同处理机的多个进程间传送数据或消息的一些技术或方法，**方式**有：

1. 管道（pipe）及有名管道（named pipe）：

管道可用于具有亲缘关系进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

2. 信号（signal）：

信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一致得。

3. 消息队列（message queue）：

消息队列是消息的链接表，它克服了上两种通信方式中信号量有限的缺点，具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息。

4. 共享内存（shared memory）：

可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。

5. 信号量（semaphore）：主要作为进程之间及同一种进程的不同线程之间得同步和互斥手段。

6. 套接字（socket）：这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

进程间通信的目的

1. **数据传输：**一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。
2. **共享数据：**多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
3. **通知事件：**一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
4. **资源共享：**多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。
5. **进程控制：**有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

引入线程的好处

（1）**易于调度**。

（2）**提高并发性**。通过线程可方便有效地实现并发性。进程可创建多个线程来执行同一程序的不同部分。

（3）**开销少**。创建线程比创建进程要快，所需开销很少。。

（4）**利于充分发挥多处理器的功能**。通过创建多线程进程（即一个进程可具有两个或更多个线程），每个线程在一个处理器上运行，从而实现应用程序的并发性，使每个处理器都得到充分运行。

线程之间的同步通信：

1.信号量 二进制信号量 互斥信号量 整数型信号量 记录型信号量

2.消息　消息队列 消息邮箱

3.事件 event

互斥型信号量：必须是同一个任务申请，同一个任务释放，其他任务释放无效。同一个任务可以递归申请。（互斥信号量是二进制信号量的一个子集）

二进制信号量：一个任务申请成功后，可以由另一个任务释放。（与互斥信号量的区别）

整数型信号量：取值不局限于 0 和 1,可以一个任务申请，另一个任务释放。（包含二进制信号量，二进制信号量是整数型信号量的子集）

二进制信号量实现任务互斥：

打印机资源只有一个，a b c 三个任务共享，当 a 取得使用权后，为了防止其他任务错误地释放了信号量（二进制信号量允许其他任务释放），必须将打印机房的门关起来(进入临界段)，用完后，释放信号量，再把门打开(出临界段)，其他任务再进去打印。(而互斥型信号量由于必须由取得信号量的那个任务释放，故不会出现其他任务错误地释放了信号量的情况出现，故不需要有临界段。互斥型信号量是二进制信号量的子集。)

二进制信号量实现任务同步：

a 任务一直等待信号量，b 任务定时释放信号量，完成同步功能

记录型信号量（record semaphore)：

每个信号量 s 除一个整数值 value（计数）外，还有一个等待队列 List，其中是阻塞在该信号量的各个线程的标识。当信号量被释放一个，值被加一后，系统自动从等待队列中唤醒一个等待中的线程，让其获得信号量，同时信号量再减一。

同步和互斥的区别：

当有多个线程的时候，经常需要去同步这些线程以访问同一个数据或资源。例如，假设有一个程序，其中一个线程用于把文件读到内存，而另一个线程用于统计文件中的字符数。当然，在把整个文件调入内存之前，统计它的计数是没有意义的。但是，由于每个操作都有自己的线程，操作系统会把两个线程当作是互不相干的任务分别执行，这样就可能在没有把整个文件装入内存时统计字数。为解决此问题，你必须使两个线程同步工作。

所谓**互斥**，是**指散布在不同进程之间的若干程序片断，当某个进程运行其中一个程序片段时，其它进程就不能运行它们之中的任一程序片段，只能等到该进程运行完这个程序片段后才可以运行。**如果用对资源的访问来定义的话，互斥某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的

所谓**同步**，是**指散步在不同进程之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务**。如果用对资源的访问来定义的话，同步是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

17. c++继承 （C++）

C++ 类的继承：http://01160529.blog.51cto.com/11479142/1865206

封装(encapsulation)：封装就是将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成“类”，其中数据和函数都是类的成员。封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只要通过外部接口，和特定的访问权限来使用类的成员。通过封装使一部分成员充当类与外部的接口，而将其他的成员隐蔽起来，这样就达到了对成员访问权限的合理控制，使不同类之间的相互影响减少到最低限度，进而增强数据的安全性和简化程序的编写工作。

由于我们不是字符串类的设计者，当我们对 string 进行种种操作时，我们只能了解到它的操作结果，而对它的操作原理和操作实现过程却无法得知。

我们把**类的数据不可知性和操作实现过程不可知性称为类的封装性**（Encapsulation）。不难理解，作为使用者，我们不需要对数据和操作实现过程感兴趣。就好像买一个手机，我们只关心它是否能够正常通话，正常发短消息，却对它如何接通电话，如何把信号发送出去等等不感兴趣。类的封装性把类的设计者和类的使用者分隔开，使他们在设计程序时互不干扰，责任明确。

继承：继承是面向对象软件技术当中的一个概念。如果一个类 B 继承自另一个类 A，就把这个 B 称为 A 的子类，而把 A 称为 B 的父类。继承可以使得子类具有父类的各种属性和方法，而不需要再次编写相同的代码。在令子类继承父类的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类的原有属性和方法，使其获得与父类不同的功能。

访问控制和继承权限

访问控制：在基类中，public 和 private 标号具有普通含义：用户代码可以访问类的 public 成员而不能访问 private 成员，private 成员只能由基类的成员和友元访问。派生类对基类的 public 和 private 成员的访问权限与程序中任意其他部分一样：它可以访问 public 成员而不能访问 private 成员。有时作为基类的类具有一些成员，它希望允许派生类访问但仍禁止其他用户访问这些成员。对于这样的成员应使用受保护的访问标号。protected 成员可以被派生类对象访问但不能被该类型的普通用户访问。

继承权限：公用继承：基类成员保持自己的访问级别，基类的 public 成员为派生类的 public 成员，基类的 protected 成员为派生类的 protected 成员。

受保护继承：基类的 public 和 protected 成员在派生类中为 protected 成员。

私有继承：基类的的所有成员在派生类中为 private 成员。

多态（Polymorphisn）：不同的类的同名成员函数有不同的表现形式，称为**多态性**。也就是称呼相同，所指不同。多态**允许将子类类型的指针赋值给父类类型的指针。多态性在 C++中都是通过虚函数（Virtual Function）实现的。虚函数就是指允许被其子类重新定义的成员函数。**而**子类重新定义父类虚函数的做法，称为“覆盖”或者称为“重写”**（override）。

多态性往往只有在使用**对象指针或对象引用**时才体现出来。编译器在编译程序的时候完全不知道对象指针可能会指向哪种对象（引用也是类似的情况），只有到程序运行了之后才能明确指针访问的成员函数是属于哪个类的。我们把 C++的这种功能称为“滞后联编”。多态性是面向对象的一个标志性特点，没有这个特点，就无法称为面向对象。

18. 什么是虚函数、虚函数的作用和虚函数的使用方法（C++）

我们知道，在同一类中是不能定义两个名字相同、参数个数和类型都相同的函数的，否则就是“重复定义”。但是在类的继承层次结构中，在不同的层次中可以出现名字相同、参数个数和类型都相同而功能不同的函数。例如在例 12.1（具体代码请查看：C++多态性的一个典型例子）程序中，在 Circle 类中定义了 area 函数，在 Circle 类的派生类 Cylinder 中也定义了一个 area 函数。这两个函数不仅名字相同，而且参数个数相同(均为 0)，但功能不同，函数体是不同的。前者的作用是求圆面积，后者的作用是求圆柱体的表面积。这是合法的，因为它们不在同一个类中。 编译系统按照同名覆盖的原则决定调用的对象。在例 12.1 程序中用 cy1.area( ) 调用的是派生类 Cylinder 中的成员函数 area。如果想调用 cy1 中的直接基类 Circle 的 area 函数，应当表示为 cy1.Circle::area()。用这种方法来区分两个同名的函数。但是这样做很不方便。

人们提出这样的设想，能否用同一个调用形式，既能调用派生类又能调用基类的同名函数。**在程序中不是通过不同的对象名去调用不同派生层次中的同名函数，而是通过指针调用它们**。例如，用同一个语句“pt->display( );”可以调用不同派生层次中的 display 函数，只需在调用前给指针变量 pt 赋以不同的值(使之指向不同的类对象)即可。

C++中的虚函数就是用来解决这个问题的。**虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。**虚就虚在所谓“推迟联编”或者“动态联编”上，一个类函数的调用并不是在编译时刻被确定的，而是在运行时刻被确定的。由于编写代码的时候并不能确定被调用的是基类的函数还是哪个派生类的函数，所以被称为“虚”函数。

作用说明：本来基类指针是用来指向基类对象的，如果用它指向派生类对象，则进行指针类型转换，将派生类对象的指针先转换为基类的指针，所以基类指针指向的是派生类对象中的基类部分。在程序修改前，是无法通过基类指针去调用派生类对象中的成员函数的。虚函数突破了这一限制，在派生类的基类部分中，派生类的虚函数取代了基类原来的虚函数，因此在使基类指针指向派生类对象后，调用虚函数时就调用了派生类的虚函数。 要注意的是，只有用 virtual 声明了虚函数后才具有以上作用。如果不声明为虚函数，企图通过基类指针调用派生类的非虚函数是不行的。

虚函数的以上功能是很有实用意义的。在面向对象的程序设计中，经常会用到类的继承，目的是保留基类的特性，以减少新类开发的时间。但是，从基类继承来的某些成员函数不完全适应派生类的需要，例如在例 12.2 中，基类的 display 函数只输出基类的数据，而派生类的 display 函数需要输出派生类的数据。过去我们曾经使派生类的输出函数与基类的输出函数不同名（如 display 和 display1），但如果派生的层次多，就要起许多不同的函数名，很不方便。如果采用同名函数，又会发生同名覆盖。

利用虚函数就很好地解决了这个问题。可以看到：**当把基类的某个成员函数声明为虚函数后，允许在其派生类中对该函数重新定义，赋予它新的功能，并且可以通过指向基类的指针指向同一类族中不同类的对象，从而调用其中的同名函数。**由虚函数实现的动态多态性就是：同一类族中不同类的对象，对同一函数调用作出不同的响应。

虚函数的使用方法是：

1. **在基类用 virtual 声明成员函数为虚函数。**

这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。在类外定义虚函数时，不必再加 virtual。

2. **在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。**

C++规定，当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。因此在派生类重新声明该虚函数时，可以加 virtual，也可以不加，但习惯上一般在每一层声明该函数时都加 virtual，使程序更加清晰。如果在派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

3. **定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。**

4. **通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。**

通过虚函数与指向基类对象的指针变量的配合使用，就能方便地调用同一类族中不同类的同名函数，只要先用基类指针指向即可。如果指针不断地指向同一类族中不同类的对象，就能不断地调用这些对象中的同名函数。这就如同前面说的，不断地告诉出租车司

机要去的目的地，然后司机把你送到你要去的地方。

需要说明：有时在基类中定义的非虚函数会在派生类中被重新定义(如例 12.1 中的 area 函数)，如果用基类指针调用该成员函数，则系统会调用对象中基类部分的成员函数；如果用派生类指针调用该成员函数，则系统会调用派生类对象中的成员函数，这并不是多态性行为(使用的是不同类型的指针)，没有用到虚函数的功能。

19. 虚函数与纯虚函数的区别（C++）

定义一个函数为虚函数，不代表函数为不被实现的函数。定义它为虚函数是为了允许用基类的指针来调用子类的这个函数。

定义一个函数为纯虚函数，才代表函数没有被实现。定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

C++纯虚函数

（1）定义

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”

virtual void funtion1()=0

（2）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;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟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。

声明了纯虚函数的类是一个抽象类。所以，用户不能创建类的实例，只能创建它的派生类的实例。纯虚函数最显著的**特征**是：它们必须在继承类中重新声明函数（不要后面的＝0，否则该派生类也不能实例化），而且它们在抽象类中往往没有定义。定义纯虚函数的**目的**在于，使派生类仅仅只是继承函数的接口。纯虚函数的**意义**，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的缺省实现。所以类纯虚函数的声明就是在告诉子类的设计者，“你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它”。

抽象类的介绍

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。

虚函数与纯虚函数的区别总结：

1、纯虚函数声明如下： virtual void funtion1()=0; **纯虚函数一定没有定义**，纯虚函数用来规范派生类的行为，即接口。包含纯虚函数的类是抽象类，抽象类不能定义实例，但可以声明指向实现该抽象类的具体类的指针或引用。

2、虚函数声明如下：virtual ReturnType FunctionName(Parameter)；**虚函数必须实现**，如果不实现，编译器将报错，错误提示为：

error LNK****: unresolved external symbol "public: virtual void __thiscall ClassName::virtualFunctionName(void)"

3、对于虚函数来说，父类和子类都有各自的实现。由多态方式调用的时候动态绑定。

4、实现了纯虚函数的子类，该纯虚函数在子类中就变成了虚函数，子类的子类即孙子类可以覆盖该虚函数，由多态方式调用的时候动态绑定。

5、虚函数是 C++中用于实现多态(polymorphism)的机制。核心理念就是通过基类访问派生类定义的函数。

6、在有动态分配堆上内存的时候，**析构函数必须是虚函数**，但没有必要是纯虚的。

7、友元不是成员函数，只有成员函数才可以是虚拟的，因此**友元不能是虚拟函数**。但可以通过让友元函数调用虚拟成员函数来解决友元的虚拟问题。

8、**析构函数应当是虚函数**，将调用相应对象类型的析构函数，因此，**如果指针指向的是子类对象，将先调用子类的析构函数，然后自动调用基类的析构函数**。

有纯虚函数的类是抽象类，不能生成对象，只能派生。他派生的类的纯虚函数没有被改写，那么，它的派生类还是个抽象类。

定义纯虚函数就是为了让基类不可实例化化

因为实例化这样的抽象数据结构本身并没有意义。

或者给出实现也没有意义

实际上我个人认为**纯虚函数的引入，是出于两个目的**

1、为了安全，因为避免任何需要明确但是因为不小心而导致的未知的结果，提醒子类去做应做的实现。

2、为了效率，不是程序执行的效率，而是为了**编码的效率**。

20. TCP 与 UDP 的区别、应用（计算机网络）

1）**TCP**（Transmission Control Protocol，传输控制协议）是**基于连接**的协议，在正式收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次握手才能建立起来。

2）TCP 提供**可靠的数据传输**服务，通信进程能够依靠 TCP 无差错、有序交付所有数据。

3）TCP 具有**拥塞控制**机制，该服务在发送方和接收方之间的网络出现拥塞时，抑制发送进程。这可能会导致进程通信变慢，但是对网络整体通信有好处。

UDP（User Data Protocol，用户数据报协议）是与 TCP 相对应的协议。

1）它是面向**非连接**的协议，它不与对方建立连接，而是直接就把数据包发送过去！（延时小）

2）UDP 提供**不可靠数据传输**服务，不保证报文到达，也不保证有序到达。这种不可靠数据传输服务包括进程间的数据交付和差错检查两种服务，是运输层协议实现的最低限度服务。

3）UDP **没有拥塞控制**机制，所以 UDP 可以用他选定的任何速率向下层（网络层）传输数据（速度有保证）。

综合以上大致可以得出两者的适用应用：UDP 适用于对**可靠性要求不高**的应用环境，但是对**数据传输速率有最小限制且对延时有要求**，例如因特网电话。TCP 适用于对**数据传输可靠**性要求较高的应用，web 的 HTTP 协议就使用 TCP 进行数据传输。

UDP 相对 TCP 的优势：

（1）**无延时**。只要应用进程将数据传递给 UDP，UDP 就会将此数据打包进 UDP 报文段并立即传递给网络层。但是 TCP 可能会因为拥塞阻塞机制阻止运输层 TCP 发送方，并且一直等待，直到可以发送，这会增加时延。对实时应用，少量数据损耗并不影响，但延时不能太大，这最好使用 UDP，如网络电话。

（2）**UDP 无需连接建立**，相比 TCP 节省了三次握手时间。DNS 使用 UDP 协议，主要是因为 UDP 不需要建立连接，节省了握手时间，如果运行在 TCP 上，DNS 会很慢。

（3）**UDP 无连接状态**，而 TCP 需要维护连接状态，这包括维持一些参数，因此使用 UDP 可以使服务器同时支持更多用户。

（4）**分组开销小**，UDP 使用 8 字节首部开销（各两字节：源端口号、目的端口号、长度、检验和），TCP 使用 20 字节首部开销。

常见使用 **TCP 协议**的应用如下：

浏览器用的 HTTP

FlashFXP 用的 FTP

Outlook 用的 POP、SMTP

Putty 用的 Telnet、SSH

QQ 文件传输

常见使用 **UDP 协议**的应用如下：

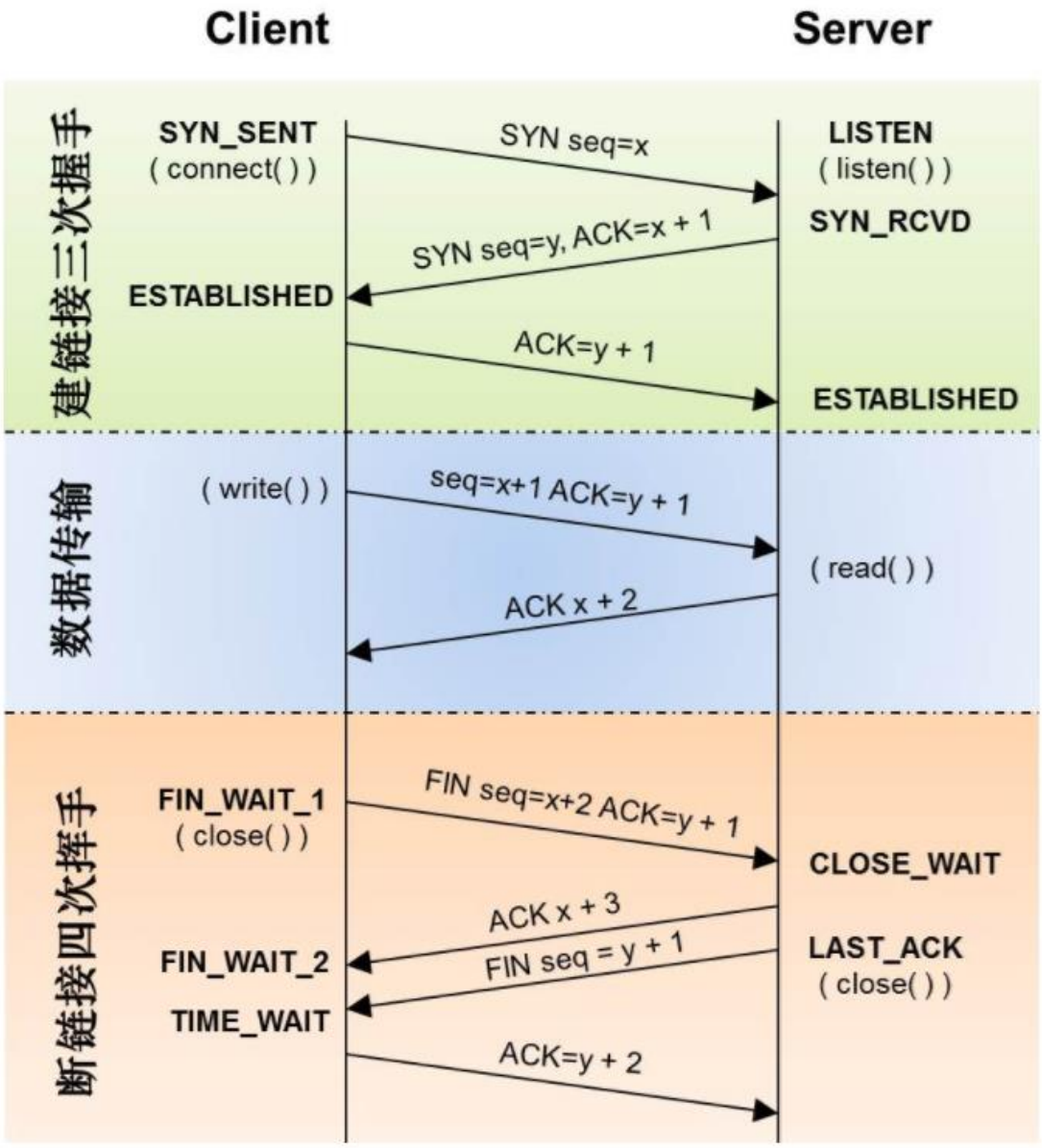
QQ 语音、QQ 视频、TFTP、DNS

表 5-1　使用 UDP 和 TCP 协议的各种应用和应用层协议

应　　用	应用层协议	运输层协议
名字转换	DNS	UDP
文件传送	TFTP	UDP
路由选择协议	RIP	UDP
IP 地址配置	BOOTP, DHCP	UDP
网络管理	SNMP	UDP
远程文件服务器	NFS	UDP
IP 电话	专用协议	UDP
流式多媒体通信	专用协议	UDP
多播	IGMP	UDP
电子邮件	SMTP	TCP
远程终端接入	TELNET	TCP
万维网	HTTP	TCP
文件传送	FTP	TCP

21. TCP 三次握手和四次挥手图，解释握手为何不能用 2 次。（计算机网络）

三次握手的过程：



三次握手的过程：

第一次握手：建立连接。客户端发送连接请求报文段，将 SYN 位置为 1，Sequence Number 为 x；然后，客户端进入 SYN_SEND 状态，等待服务器的确认；

第二次握手：服务器收到 SYN 报文段。服务器收到客户端的 SYN 报文段，需要对这个 SYN 报文段进行确认，设置 Acknowledgment Number 为 x+1(Sequence Number+1)；同时，自己自己还要发送 SYN 请求信息，将 SYN 位置为 1，Sequence Number 为 y；服务器端将上述所有信息放到一个报文段（即 SYN+ACK 报文段）中，一并发送给客户端，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 报文段。然后将 Acknowledgment Number 设置为 y+1，向服务器发送 ACK 报文段，这个报文段发送完毕以后，客户端和服务器端都进入 ESTABLISHED 状态，完成 TCP 三次握手。

完成了三次握手，客户端和服务器端就可以开始传送数据。

四次挥手的过程：

第一次挥手：主机 1（可以使客户端，也可以是服务器端），设置 Sequence Number 和 Acknowledgment Number，向主机 2 发送一个 FIN 报文段；此时，主机 1 进入 FIN_WAIT_1 状态；这表示主机 1 没有数据要发送给主机 2 了；
第二次挥手：主机 2 收到了主机 1 发送的 FIN 报文段，向主机 1 回一个 ACK 报文段，Acknowledgment Number 为 Sequence Number 加 1；主机 1 进入 FIN_WAIT_2 状态；主机 2 告诉主机 1，我“同意”你的关闭请求；
第三次挥手：主机 2 向主机 1 发送 FIN 报文段，请求关闭连接，同时主机 2 进入 LAST_ACK 状态；
第四次挥手：主机 1 收到主机 2 发送的 FIN 报文段，向主机 2 发送 ACK 报文段，然后主机 1 进入 TIME_WAIT 状态；主机 2 收到主机 1 的 ACK 报文段以后，就关闭连接；此时，主机 1 等待 2MSL 后依然没有收到回复，则证明 Server 端已正常关闭，那好，主机 1 也可以关闭连接了。

为什么要三次握手？

client 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达 server。本来这是一个早已失效的报文段。但 server 收到此失效的连接请求报文段后，就误认为是 client 再次发出的一个新的连接请求。于是就向 client 发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要 server 发出确认，新的连接就建立了。由于现在 client 并没有发出建立连接的请求，因此不会理睬 server 的确认，也不会向 server 发送数据。但 server 却以为新的运输连接已经建立，并一直等待 client 发来数据。这样，server 的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。**防止了服务器端的一直等待而浪费资源。**

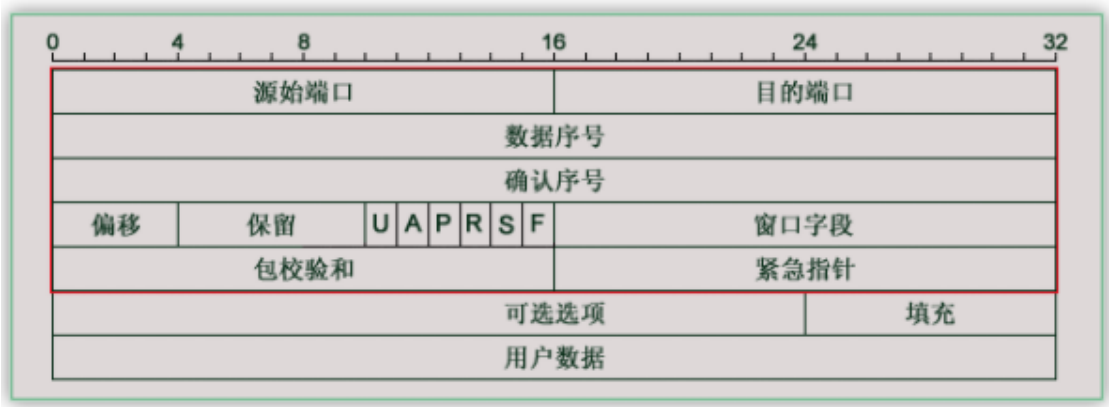
为什么要四次挥手？

- **FIN_WAIT_1:** 这个状态要好好解释一下，其实 FIN_WAIT_1 和 FIN_WAIT_2 状态的真正含义都是表示等待对方的 FIN 报文。而这两种状态的区别是：FIN_WAIT_1 状态实际上是当 SOCKET 在 ESTABLISHED 状态时，它想主动关闭连接，向对方发送了 FIN 报文，此时该 SOCKET 即进入到 FIN_WAIT_1 状态。而当对方回应 ACK 报文后，则进入到 FIN_WAIT_2 状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应 ACK 报文，所以 FIN_WAIT_1 状态一般是比较难见到的，而 FIN_WAIT_2 状态还有时常常可以用 netstat 看到。（主动方）
- **FIN_WAIT_2**：上面已经详细解释了这种状态，实际上 FIN_WAIT_2 状态下的 SOCKET，表示半连接，也即有一方要求 close 连接，但另外还告诉对方，我暂时还有点数据需要传送给你(ACK 信息)，稍后再关闭连接。（主动方）
- **CLOSE_WAIT:** 这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方 close 一个 SOCKET 后发送 FIN 报文给自己，你系统毫无疑问地会回应一个 ACK 报文给对方，此时则进入到 CLOSE_WAIT 状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以 close 这个 SOCKET，发送 FIN 报文给对方，也即关闭连接。所以你在 CLOSE_WAIT 状态下，需要完成的事情是等待你去关闭连接。（被动方）
- **LAST_ACK**: 这个状态还是比较容易好理解的，它是被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。当收到 ACK 报文后，也即可以进入到 CLOSED 可用状态了。（被动方）
- **TIME_WAIT:** 表示收到了对方的 FIN 报文，并发送出了 ACK 报文，就等 2MSL 后即可回到 CLOSED 可用状态了。如果 FINWAIT1 状态下，收到了对方同时带 FIN

标志和 ACK 标志的报文时，可以直接进入到 TIME_WAIT 状态，而无须经过 FIN_WAIT_2 状态。（主动方）

- **CLOSED:** 表示连接中断。

22. TCP 数据包结构（计算机网络）



1-1.源始端口 16 位，范围当然是 0-65535 啦。

1-2.目的端口，同上。

2-1.数据序号 32 位，TCP 为发送的每个字节都编一个号码，这里存储当前数据包数据第一个字节的序号。

3-1.确认序号 32 位，为了安全，TCP 告诉接受者希望他下次接到数据包的第一个字节的序号。

4-1.偏移 4 位，类似 IP，表明数据距包头有多少个 32 位。

4-2.保留 6 位，未使用，应置零。

4-3.紧急比特 URG—当 URG＝1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)。

4-3.确认比特 ACK—只有当 ACK＝1 时确认号字段才有效。当 ACK＝0 时，确认号无效。

4-4.复位比特 RST(Reset) —当 RST＝1 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。

4-5.同步比特 SYN—同步比特 SYN 置为 1，就表示这是一个连接请求或连接接受报文。参考 TCP 三次握手

4-6.终止比特 FIN(FINal)—用来释放一个连接。当 FIN＝1 时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

4-7.窗口字段 16 位，窗口字段用来控制对方发送的数据量，单位为字节。TCP 连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限。

5-1.包校验和 16 位，包括首部和数据这两部分。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。

5-2.紧急指针 16 位，紧急指针指出在本报文段中的紧急数据的最后一个字节的序号。

6-1.可选选项 24 位，类似 IP，是可选选项。

6-2.填充 8 位，使选项凑足 32 位。

7-1.用户数据……

23. TCP 如何保证可靠性？（计算机网络）

1、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重传。

2、数据校验

3、数据合理分片和排序：

UDP：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.

tcp 会按 MTU 合理分片，接收方会缓存未按序到达的数据，重新排序后再交给应用层。

4、流量控制：当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。

5、拥塞控制：当网络拥塞时，减少数据的发送。

24. TCP 数据校验（计算机网络）

TCP 校验和是一个端到端的校验和，由发送端计算，然后由接收端验证。其目的是为了发现 TCP 首部和数据在发送端到接收端之间发生的任何改动。如果接收方检测到校验和有差错，则 TCP 段会被直接丢弃。

TCP 校验和覆盖 TCP 首部和 TCP 数据，而 IP 首部中的校验和只覆盖 IP 的首部，不覆盖 IP 数据报中的任何数据。

TCP 的校验和是必需的，而 UDP 的校验和是可选的。

TCP 和 UDP 计算校验和时，都要加上一个 12 字节的伪首部。

首先，IP、ICMP、UDP 和 TCP 报文头部都有校验和字段，大小都是 16bit，算法也基本一样：

在发送数据时，为了**计算数据包的校验和**。应该按如下步骤：

（1）把校验和字段置为 0；

（2）把需校验的数据看成以 16 位为单位的数字组成，依次进行二进制反码求和；

（3）把得到的结果存入校验和字段中。

在接收数据时，计算数据包的校验和相对简单，按如下步骤：

（1）把首部看成以 16 位为单位的数字组成，依次进行二进制反码求和，包括校验和字段；

（2）检查计算出的校验和的结果是否为 0；

（3）如果等于 0，说明被整除，校验是和正确。否则，校验和就是错误的，协议栈要抛弃这个数据包。

虽然上面四种报文的校验和算法一样，但在作用范围存在不同：IP 校验和只校验 20 字节的 IP 报头；而 ICMP 校验和覆盖整个报文（ICMP 报头+ICMP 数据）；UDP 和 TCP 校验和不仅覆盖整个报文，而且还有 12 字节的 IP 伪首部，包括源 IP 地址(4 字节)、目的 IP 地址(4 字节)、协议(2 字节，第一字节补 0)和 TCP/UDP 包长(2 字节)。另外 UDP、TCP 数据报的长度可以为奇数字节，所以在计算校验和时需要在最后增加填充字节 0（注意，填充字节只是为了计算校验和，可以不被传送）。

这里还要提一点，UDP 的校验和是可选的，当校验和字段为 0 时，表明该 UDP 报文未使用校验和，接收方就不需要校验和检查了！那如果 UDP 校验和的计算结果是 0 时怎么办呢？书上有这么一句话：“如果校验和的计算结果为 0，则存入的值为全 1（65535），这在二进制反码计算中是等效的。”

讲了这么多，那这个校验和到底是怎么算的呢？

什么是二进制反码求和(1 的补码)

对一个无符号的数，先求其反码，然后从低位到高位，按位相加，有溢出则向高位进 1（跟一般的二进制加法规则一样），若最高位有进位，则向最低位进 1。

首先这里的反码好像跟我们以前学的有符号数的反码不一样（即正数的反码是其本身，负数的反码是在其原码的基础上，符号位不变，其余各位取反），这里不分正负数，直接每个位都取反！
上面加粗的那句是跟我们一般的加法规则不太一样的地方：最高位有进位，则向最低位进 1。确实有些疑惑，为什么要这样做呢？仔细分析一下（为了方便说明，以 4bit 二进制反码求和举例），上面的这种操作，使得在发生加法进位溢出时，溢出的值并不是 10000，而是 1111。也即是当相加结果满 1111 时溢出，这样也可以说明为什么 0000 和 1111 都表示 0 了（你同样可以发现，任何数与这两个数做二进制反码求和运算结果都是原数，这恰好符合数 0 的加法意义）。

下面再举例两种二进制反码求和的运算：
原码加法运算 反码加法运算
3（0011）+ 5（0101）= 8（1000） 3（1100）+ 5（1010）= 8（0111）
8（1000）+ 9（1001）= 1（0001） 8（0111）+ 9（0110）= 2（1101）
从上面两个例子可以看出，当加法未发生溢出时，原码与反码加法运算结果一样；当有溢出时，结果就不一样了，原码是满 10000 溢出，而反码是满 1111 溢出，所以相差正好是 1。举例只是为了形象地观察二进制反码求和的运算规则，至于为什么要定义这样的规则以及该运算规则还存在其它什么特性，可能就需要涉及代数理论的东西的了。
另外关于二进制反码求和运算需要说明的一点是，先取反后相加与先相加后取反，得到的结果是一样的！（事实上我们的编程算法里，几乎都是先相加后取反。）
1 的补码（相加和补取）猜测是计算机相关语言.请参考这两段文字： It is the 1’s complement of the 1’s complement sum of all the 16-bit words in the TCP header and data. 这是关于 TCP 头部校验和字段（checksum field）的说明.句中的 complement 意思为“补码”.对于学习计算机科学的人来说,补码不算什么新鲜,现在新鲜的是这篇英语文章出现的是“1’s complement” ,翻译出来应该是“1 的补码”,对于这个笔者以前也没有碰到过,到网上查吧!网上查询的结果,“1’s complement”关键字出现的不少,但都是英文关键字,没有对应的中文翻译与解释,所以先看英语的,最后自己做解释吧. 补码：补码是计算机中二进制数表达负数的办法,这样可以在计算机中把两个数的减法变成加法.补码形式有 1 的补码和 2 的补码,其中 1 的补码用在 IP、TCP 的校验和中；平时学生在计算机科学中学习的补码是 2 的补码（即正数的补码和原码相同,负数补码按原码相应的正数按位取反再加 1）. 只是平时中文教材及中文翻译的书中,对此并不多加解释,一律翻译作补码.比如《Computer Network》(Andrew S.Tanenbaum)在中国的翻译版《计算机网络》（清华大学出版社）对于 TCP 头部的校验和是这样翻译的： （原文）The checksum algorithm is simply to add up all the 16-bit words in one's complement and then to take the one's complement of the sum. （译文）校验和的算法是简单地将所有 16 位字以补码形式相加,然后再对相加和取补. 仔细对比一下本文最上部笔者所碰到的句子,和刚才这个句子意思是一样的.这个没有注明是 1 的补码,翻译时只是以“补码”说明,也许译者并不想在这里多费口舌,因为说明 1 的补码实在是一个比较麻烦的事情,笔者在这里翻译时还是把“1 的补码”给翻译出来了,以让大家注意这个 1 的补码并不是平常学的那个 2 的补码. 笔者只是在此讨论翻译,不是在讨论补码.1 的补码较复杂,如果有兴趣,可以上网查找 RFC1071 ,这是 TCP 校验和权威的官方说明.

25. 哈希表是什么？（数据结构）

散列表（Hash table，也叫哈希表），是根据键（Key）而直接访问在内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做散列表。

26. 红黑树（数据结构）

红黑树，本质上来说就是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为 **O(log n)**。

但它是如何保证一棵 n 个结点的红黑树的高度始终保持在 h = logn 的呢？这就引出了红黑树的 **5 条性质：**

1）每个结点要么是红的，要么是黑的。

2）根结点是黑的。

3）每个叶结点（叶结点即指树尾端 NIL 指针或 NULL 结点）是黑的。

4）如果一个结点是红的，那么它的俩个儿子都是黑的。

5）对于任一结点而言，其到叶结点树尾端 NIL 指针的每一条路径都包含相同数目的黑结点。

正是红黑树的这 5 条性质，使得一棵 n 个结点是红黑树始终保持了 logn 的高度，从而也就解释了上面我们所说的“红黑树的查找、插入、删除的时间复杂度最坏为 O(log n)”这一结论的原因。

https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md

27. 若有客户打不开网站，分析原因排错（计算机网络）

OSI 参考模型的基础知识：

1、OSI 模型每一层都为上一层提供服务

2、网络出现故障从底层往高层一项项的逐步检查

（1）物理层

物理层故障：查看连接状态，查看发送和接收的数据包的具体情况（网线没有接上（断开）、网线的水晶头该重新置换，没有接触良好）

（2）数据链路层

MAC 地址冲突、ADSL 拨号上网欠费、网速没有办法协商一致、计算机连接到错误的 VLAN

（3）网络层

配置错误 IP 地址，子网掩码/配置错误的网关，路由器没有配置，到达不了目标网络的路由器

（4）应用层

应用程序配置错误

28. Malloc 和 new 的区别（C++）

1,**malloc 与 free** 是 C++/C 语言的**标准库函数**，**new/delete** 是 C++的**运算符**。它们都可用于动态申请内存和释放内存。

2,对于内置数据类型的对象而言，二者没有什么区别。但是对于类类型的数据而言，用 malloc/free 是无法满足要求的。对象在创建的同时要自动执行构造函数，对象在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。

3,因此 C++语言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。

4,malloc 申请内存的时候要指定分配内存的大小，而且不会初始化；new 申请内存时有默认的初始化，也可以指定初始化。

29. TCP 的拥塞控制（计算机网络）

http://www.cnblogs.com/hnrainll/archive/2011/10/14/2212253.html TCP/IP 协议族之运输层协议 (UDP, TCP)

在某段时间内，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏，这种情况就叫做拥塞。

从大的方面来看，拥塞控制可分为**开环控制**和**闭环控制**两种方法。开环控制方法就是在设计网络时事先将有关发生拥塞的因素考虑周到，力求网络在工作时不产生拥塞。但一旦整个系统运行起来，就不再中途进行改正了。**闭环控制是基于反馈环路**的概念。属于闭环控制的有以下几种措施：

（1）监测网络系统以便检测到拥塞在何时、何处发生。

（2）把拥塞发生的信息发送到可采取行动的地方。

（3）调整网络系统的运行以解决出现的问题。

拥塞控制与流量控制

拥塞控制就是**防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载**。拥塞控制所要做的都有一个前提，就是**网络能够承受现有的网络负荷**。拥塞控制是一个**全局性**的过程，涉及到所有的主机、所有的路由器，以及与降低网络传输性能有关的所有因素。

流量控制往往指点对点通信量的控制，是个**端到端**的问题（接收端控制发送端）。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

二者被弄混的原因是，因为某些拥塞控制算法是向发送端发送控制报文，并告诉发送端，网络已出现麻烦，必须放慢发送速率。这点和流量控制相似。

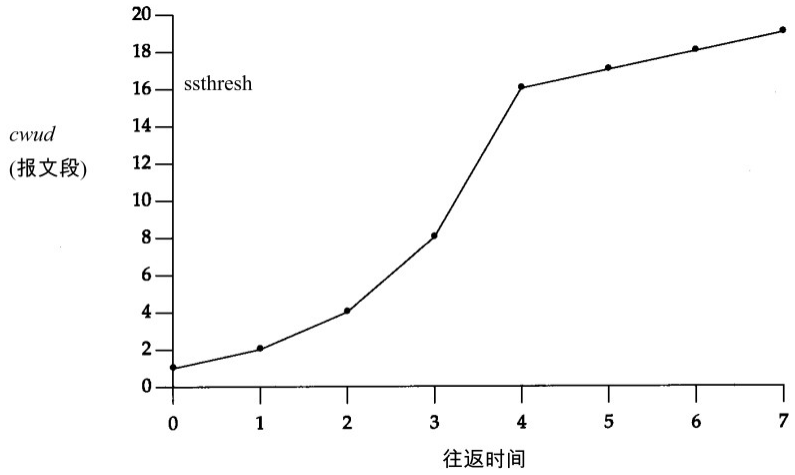
拥塞控制的方法

因特网建议标准 RFC2581 定义了进行拥塞控制的四种算法，即**慢开始**（Slow-start)，**拥塞避免**（Congestion Avoidance)，**快重传**（Fast Restrangsmit)和**快恢复**（Fast Recovery）。我们假定

1）数据是单方向传送，而另外一个方向只传送确认。

2)接收方总是有足够大的缓存空间，因为发送窗口的大小由网络的拥塞程度来决定。

下图是慢启动和拥塞避免的一个可视化描述。我们以段为单位来显示 cwnd 和 ssthresh，但它们实际上都是以字节为单位进行维护的。



拥塞窗口概念：发送报文段速率的确定，既要根据接收端的接收能力，又要从全局考虑不要使网络发生拥塞，这由接收窗口和拥塞窗口两个状态量确定。接收端窗口（Reciver Window)又称通知窗口（Advertised Window),是接收端根据目前的接收缓存大小所许诺的最新窗口值，是来自接收端的流量控制。拥塞窗口 cwnd（Congestion Window)是发送端根据自己估计的网络拥塞程度而设置的窗口值，是来自发送端的流量控制。

（1）**慢启动原理**

1）当主机开始发送数据时，如果立即将较大的发送窗口的全部数据字节都注入到网络中，那么由于不清楚网络的情况，有可能引起网络拥塞

2）比较好的方法是试探一下，即**从小到达逐渐增大发送端的拥塞控制窗口数值**

3）通常在刚刚开始发送报文段时可先将拥塞窗口 cwnd(拥塞窗口)设置为一个最大报文段的 MSS 的数值。在每收到一个对新报文段确认后，将拥塞窗口增加至多一个 MSS 的数值，当 rwind（接收窗口）足够大的时候，为了防止拥塞窗口 cwind 的增长引起网络拥塞，还需要另外一个变量---慢开始门限 ssthresh

（2）**拥塞控制**

具体过程为：

1）TCP 连接初始化，将拥塞窗口设置为 1

2）执行 慢开始算法：cwind 按指数规律增长，直到 cwind == ssthress 开始执行**拥塞避免算法：cwnd 按线性规律增长**

3）当网络发生拥塞，把 ssthresh 值更新为拥塞前 ssthresh 值的一半，cwnd 重新设置为 1，按照步骤（2）执行。

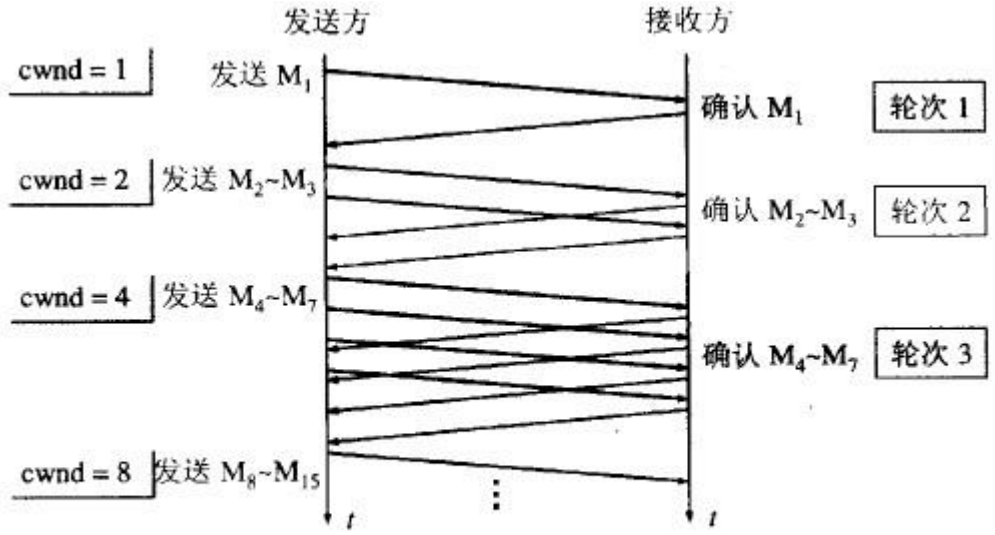


图 5-24　发送方每收到一个确认就把窗口 cwnd 加 1

（3）**快重传和快恢复**

一条 TCP 连接有时会因等待重传计时器的超时而空闲较长的时间，慢开始和拥塞避免无法很好的解决这类问题，因此提出了快重传和快恢复的拥塞控制方法。

快重传算法并非取消了重传机制，它要求接收方每收到一个失序的报文段后就立即发出重复确认，如果当发送端接收到三个重复的确认 ACK 时，则断定分组丢失，立即重传丢失的报文段，而不必等待重传计时器超时。

例如：M1，M2，M3 -----> M1,M3,缺失 M2，则接收方向发送方持续发送 M2 重复确认，当发送方收到 M2 的三次重复确认，则认为 M2 报文丢失，启动快重传机制，重传数据，其他数据发送数据放入队列，待快重传结束后再正常传输。

快恢复算法有以下两个要点：

1）当发送方连续收到接收方发来的三个重复确认时，就执行“乘法减小”算法，把慢开始门限减半(ssthresh = ssthresh/2)，这是为了预防网络发生拥塞。

2）由于发送方现在认为网络很可能没有发生拥塞，因此现在不执行慢开始算法，而是把 cwnd(拥塞窗口)值设置为慢开始门限减半后的值（cwnd=ssthresh/2），然后开始执行拥塞避免算法，使拥塞窗口缓慢地线性增大。

30. C/C++与 Java 的区别（C++）

（1）Java 中对内存的分配是动态的,它采用面向对象的机制,采用运算符 new 为每个对象分配内存空间,而且,实际内存还会随程序运行情况而改变.程序运行中,每个 Java 系统自动对内存进行扫描,对长期不用的空间作为“垃圾”进行收集,使得系统资源得到更充分地利用.按照这种机制,程序员不必关注内存管理问题,这使 Java 程序的编写变得简单明了,并且避免了由于内存管理方面的差错而导致系统出问题.而 C 语言通过 malloc()和 free()这两个库函数来分别实现分配内存和释放内存空间的,C++语言中则通过运算符 new 和 delete 来分配和释放内存.在 C 和 C++这种机制中,程序员必须非常仔细地处理内存的使用问题.一方面,如果对已释放的内存再作释放或者对未曾分配的内存作释放,都会造成死机;而另一方面,如果对长期不用的或不再使用的内存不释放,则会浪费系统资源,甚至因此造成资源枯竭.

（2）Java 不在所有类之外定义全局变量,而是在某个类中定义一种公用静态的变量来完成全局变量的功能.

（3）Java 不用 goto 语句,而是用 try-catch-finally 异常处理语句来代替 goto 语句处理出错的功能.

（4）Java 不支持头文件,面 C 和 C++语言中都用头文件来定义类的原型,全局变量,库函数等,这种采用头文件的结构使得系统的运行维护相当繁杂.

（5） Java 不支持宏定义,而是使用关键字 final 来定义常量,在 C++中则采用宏定义来实现常量定义,这不利于程序的可读性.

（6）Java 对每种数据类型都分配固定长度.比如,在 Java 中,int 类型总是 32 位的,而在 C 和 C++中,对于不同的平台,同一个数据类型分配不同的字节数,同样是 int 类型,在 PC 机中为二字节即 16 位,而在 VAX-11 中,则为 32 位.这使得 C 语言造成不可移植性,而 Java 则具有跨平台性(平台无关性).

（7）类型转换不同.在 C 和 C++中,可通过指针进行任意的类型转换,常常带来不安全性,而在 Java 中,运行时系统对对象的处理要进行类型相容性检查,以防止不安全的转换.

（8） 结构和联合的处理.在 C 和 C++中,结构和联合的所有成员均为公有,这就带来了安全性问题,而在 Java 中根本就不包含结构和联合,所有的内容都封装在类里面

（9）Java 不再使用指针.指针是 C 和 C++中最灵活,也最容易产生错误的数据类型.由指针所进行的内存地址操作常会造成不可预知的错误,同时通过指针对某个内存地址进行显式类型转换后,可以访问一个 C++中的私有成员,从而破坏安全性.而 Java 对指针进行完全地控制,程序员不能直接进行任何指针操作.

31. 寻找最小的 K 个数（数据结构与算法）

0、 咱们先简单的理解，要求一个序列中最小的 k 个数，按照惯有的思维方式，很简单，先对这个序列从小到大排序，然后输出前面的最小的 k 个数即可。

1、 至于选取什么的排序方法，我想你可能会第一时间想到快速排序，我们知道，快速排序平均所费时间为 n*logn，然后再遍历序列中前 k 个元素输出，即可，总的时间复杂度为 O（n*logn+k）=O（n*logn）。

2、 咱们再进一步想想，题目并没有要求要查找的 k 个数，甚至后 n-k 个数是有序的，既然如此，咱们又何必对所有的 n 个数都进行排序列?

这时，咱们想到了用选择或交换排序，即遍历 n 个数，先把**最先遍历到得 k 个数存入大小为 k 的数组之中**，对这 k 个数，利用选择或交换排序，找到 k 个数中的最大数 kmax（kmax 设为 k 个元素的数组中最大元素），用时 O（k）（你应该知道，插入或选择排序查找操作需要 O（k）的时间），后再继续遍历后 n-k 个数，x 与 kmax 比较：如果 x<kmax，则 x 代替 kmax，并再次重新找出 k 个元素的数组中最大元素 kmax‘（多谢 kk791159796 提醒修正）；如果 x>kmax，则不更新数组。这样，每次更新或不更新数组的所用的时间为 O（k）或 O（0），整趟下来，总的时间复杂度平均下来为：n*O（k）=**O（n*k）**。

3、 当然，更好的办法是**维护 k 个元素的最大堆**，原理与上述第 2 个方案一致，即用容量为 k 的最大堆存储最先遍历到的 k 个数，并假设它们即是最小的 k 个数，建堆费时 O（k）后，有 k1<k2<...<kmax（kmax 设为大顶堆中最大元素）。继续遍历数列，每次遍历一个元素 x，与堆顶元素比较，x<kmax，更新堆（用时 logk），否则不更新堆。这样下来，总费时 O（k+（n-k）*logk）=**O（n*logk）**。此方法得益于在堆中，查找等各项操作时间复杂度均为 logk（不然，就如上述思路 2 所述：直接用数组也可以找出前 k 个小的元素，用时 O（n*k））。

4、按编程之美第 141 页上解法二的所述，类似快速排序的划分方法，N 个数存储在数组 S 中，再从数组中随机选取一个数 X（随机选取枢纽元，可做到线性期望时间 O（N）的复杂度，在第二节论述），把数组划分为 Sa 和 Sb 俩部分，Sa<=X<=Sb，如果要查找的 k 个元素小于 Sa 的元素个数，则返回 Sa 中较小的 k 个元素，否则返回 Sa 中所有元素+Sb 中小的 k-|Sa|个元素。像上述过程一样，这个运用类似快速排序的 partition 的快速选择 SELECT 算法寻找最小的 k 个元素，在最坏情况下亦能做到 O（N）的复杂度。不过值得一提的是，这个快速选择 SELECT 算法是选取数组中“中位数的中位数”作为枢纽元，而非随机选取枢纽元。

5、 RANDOMIZED-SELECT，每次都是随机选取数列中的一个元素作为主元，在 O（n）的时间内找到第 k 小的元素，然后遍历输出前面的 k 个小的元素。 如果能的话，那么总的时间复杂度为线性期望时间：O（n+k）=O（n）（当 k 比较小时）。

32. 常用进程调度算法（操作系统）

一、进程调度的原因

在操作系统中，由于进程总数多于处理机，它们必然竞争处理机。为了充分利用计算机系统中的 CPU 资源，让计算机系统能够多快好省地完成我们让它做的各种任务，所以需要进行进程调度。

二、进程调度的定义

进程调度（也称 CPU 调度）是指按照某种调度算法（或原则）从就绪队列中选取进程分配 CPU，主要是协调对 CPU 的争夺使用。

通常有以下两种进程调度方式：

1) 非剥夺调度方式，又称非抢占方式。

是指当一个进程正在处理机上执行时，即使有某个更为重要或紧迫的进程进入就绪队列，仍然让正在执行的进程继续执行，直到该进程完成或发生某种事件而进入阻塞状态时，

才把处理机分配给更为重要或紧迫的进程。在非剥夺调度方式下，一旦把 CPU 分配给一个进程，那么该进程就会保持 CPU 直到终止或转换到等待状态。这种方式的优点是实现简单、系统开销小，适用于大多数的批处理系统，但它不能用于分时系统和大多数的实时系统。

2) 剥夺调度方式，又称抢占方式。

是指当一个进程正在处理机上执行时，若有某个更为重要或紧迫的进程需要使用处理机，则立即暂停正在执行的进程，将处理机分配给这个更为重要或紧迫的进程。

三、进程调度的评价指标

调度算法的评价指标通常如下：

1）CPU 利用率

CPU 是计算机系统中的稀缺资源，所以应在有具体任务的情况下尽可能使 CPU 保持忙，从而使得 CPU 资源利用率最高。

CPU 利用率 ＝CPU 利用的时间 / 开机运行的总时间

2）吞吐量

CPU 运行时的工作量大小是以每单位时间所完成的进程数目来描述的，即称为吞吐量。

3）周转时间

指从进程创建到进程结束所经过的时间，这期间包括了由于各种因素（比如等待 I/O 操作完成）导致的进程阻塞，处于就绪态并在就绪队列中排队，在处理机上运行所花时间的总和。

作业的周转时间：

周转时间 = 作业完成时间 - 作业提交时间

平均周转时间是指多个作业周转时间的平均值：

平均周转时间 ＝(作业 1 的周转时间 + … + 作业 n 的周转时间) / n

带权周转时间是指作业周转时间与作业实际运行时间的比值：

带权周转时间 = 作业周转时间 / 作业实际运行时间

平均带权周转时间是指多个作业带权周转时间的平均值：

平均带权周转时间 ＝(作业 1 的带权周转时间 + … + 作业 n 的带权周转时间) / n

4）等待时间

即进程在就绪队列中等待所花的时间总和。因此衡量一个调度算法的简单方法就是统计进程在就绪队列上的等待时间。

5）响应时间

指从事件（比如产生了一次时钟中断事件）产生到进程或系统作出响应所经过的时间。在交互式桌面计算机系统中，用户希望响应时间越快越好，但这常常要以牺牲吞吐量为代价。

四、**进程调度的常见算法（操作系统）**

1）**先来先服务调度算法**（FCFS，First Come First Server）

处于就绪态的进程按先后顺序链入到就绪队列中，而 FCFS 调度算法按就绪进程进入就绪队列的先后次序选择当前最先进入就绪队列的进程来执行，直到此进程阻塞或结束，才进行下一次的进程选择调度。FCFS 调度算法采用的是不可抢占的调度方式，一旦一个进程占有处理机，就一直运行下去，直到该进程完成其工作，或因等待某一事件而不能继续执行时，才释放处理机。

FCFS 调度算法的特点是**算法简单**，但**效率低**；对**长作业比较有利**，但对**短作业不利**（相对 SJF 和高响应比）；有利于 CPU 繁忙型作业，而不利于 I/O 繁忙型作业。

2）**短作业优先调度算法**（SJF，Short Job First）

短作业（进程）优先调度算法是指对短作业（进程）优先调度的算法。**短作业**优先(SJF)调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们**调入内存**运

行。而**短进程**优先(SPF)调度算法，则是从就绪队列中选择一个估计运行时间最短的进程，**将处理机分配给它**，使之立即执行，直到完成或发生某事件而阻塞时，才释放处理机。

SJF 又分为两种：

（1）SRTF 抢占式：又称最短剩余优先，当新进来的进程的 CPU 区间比当前执行的进程所剩的 CPU 区间短，则抢占。

（2）非抢占：称为下一个最短优先，即为在就绪队列中选择最短 CPU 区间的进程放在队头。

SJF 调度算法的特点是吞吐率高，平均等待时间、平均周转时间最少；但算法对长作业十分不利，也完全未考虑作业的紧迫程度。

3）**时间片轮转法**（RR，Round Robin）

时间片轮转调度算法主要**适用于分时系统**。在这种算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片，如 100ms。在使用完一个时间片后，即使进程并未完成其运行，它也必须释放出（被剥夺）处理机给下一个就绪的进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行。

在时间片轮转调度算法中，**时间片的大小对系统性能的影响很大**。如果时间片足够大，以至于所有进程都能在一个时间片内执行完毕，则时间片轮转调度算法就退化为先来先服务调度算法。如果时间片很小，那么处理机将在进程间过于频繁切换，使处理机的开销增大，而真正用于运行用户进程的时间将减少。因此时间片的大小应选择适当。

时间片的长短通常由以下因素确定：系统的响应时间、就绪队列中的进程数目和系统的处理能力。

时间片 q＝ 系统对相应时间的要求 RT / 最大进程数 N

（经验表明，时间片的取值，应该使得 80%的进程在时间片内完成所需的一次 CPU 运行活动。）

4）**多级反馈队列调度算法**（MLFQ，Multi-Level Feedback Queue）

设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二队次之，其余队列优先级依次降低。仅当第 1～i-1 个队列均为空时，操作系统调度器才会调度第 i 个队列中的进程运行。赋予各个队列中进程执行时间片的大小也各不相同。在优先级越高的队列中，每个进程的执行时间片就越小或越大（Linux-2.4 内核就是采用这种方式）。

当一个就绪进程需要进入就绪队列时，操作系统首先将它放入第一队列的末尾，按 FCFS 的原则排队等待调度。若轮到该进程执行且在一个时间片结束时尚未完成，则操作系统调度器便将该进程转入第二队列的末尾，再同样按先来先服务原则等待调度执行。如此下去，当一个长进程从第一队列降到最后一个队列后，在最后一个队列中，可使用 FCFS 或 RR 调度算法来运行处于此队列中的进程。

如果处理机正在第 i（i>1）队列中为某进程服务时，又有新进程进入第 k（k<i）的队列，则新进程将抢占正在运行进程的处理机，即由调度程序把正在执行进程放回第 i 队列末尾，重新将处理机分配给处于第 k 队列的新进程。

5）**高响应比优先调度算法**（HRRF，Highest Response Ratio First）

高响应比优先调度算法主要用于作业调度，该算法是对 FCFS 调度算法和 SJF 调度算法的一种综合平衡，同时考虑每个作业的等待时间和估计的运行时间。在每次进行作业调度时，先计算后备作业队列中每个作业的响应比，从中选出响应比最高的作业投入运行。

响应比 ＝ （等待时间+要求服务时间） / 要求服务时间 ＝响应时间 / 执行时间

6）**最高优先级优先调度算法**（PR，Priority First）

在作业调度中，最高优先级调度算法每次从后备作业队列中选择优先级最高的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。在进程调度中，优先级调度算法每次从就绪队列中选择优先级最高的进程，将处理机分配给它，使之投入运行。

根据新的更高优先级进程能否抢占正在执行的进程，可将该调度算法分为：

（1）非剥夺式优先级调度算法。

当某一个进程正在处理机上运行时，即使有某个更为重要或紧迫的进程进入就绪队列，仍然让正在运行的进程继续运行，直到由于其自身的原因而主动让出处理机时（任务完成或等待事件），才把处理机分配给更为重要或紧迫的进程。

（2）剥夺式优先级调度算法。

当一个进程正在处理机上运行时，若有某个更为重要或紧迫的进程进入就绪队列，则立即暂停正在运行的进程，将处理机分配给更重要或紧迫的进程。

而根据进程创建后其优先级是否可以改变，可以将进程优先级分为以下两种：

（1）静态优先级。

优先级是在创建进程时确定的，且在进程的整个运行期间保持不变。确定静态优先级的主要依据有进程类型、进程对资源的要求、用户要求。

（2）动态优先级。

在进程运行过程中，根据进程情况的变化动态调整优先级。动态调整优先级的主要依据为进程占有 CPU 时间的长短、就绪进程等待 CPU 时间的长短。

33. 面向对象的三大基本特性、五大基本原则（面向对象）

三大特性是：封装,继承,多态

所谓**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。封装是面向对象的特征之一，是对象和类概念的主要特性。 简单的说，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。

所谓**继承**是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；

所谓**多态**就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。

五大基本原则

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。

替换原则(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。

依赖原则(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(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

34. C++重载、重写、重定义区别（C++）

（1）重载：函数有同样的名称，但是参数列表不相同的情形，这样的**同名不同参数**的函数之间，互相称之为重载函数。

基本条件：

函数名必须相同；

函数参数必须不相同，可以是参数类型或者参数个数不同；

函数返回值可以相同，也可以不相同；

注意：

只能通过不同的参数样式进行重载，例如：不同的参数类型，不同的参数个数，不同的参数顺序；

不能通过访问权限、返回类型、抛出的异常进行重载；

重载的函数应该在相同的作用域下；

（2）重写：也称为**覆盖，子类重新定义父类中有相同名称和参数的虚函数**，主要在继承关系中出现。

基本条件：

重写的函数和被重写的函数必须为 virtual 函数，分别位于基类和派生类中；

重写的函数和被重写的函数函数名和函数参数必须一致；

重写的函数和被重写的函数返回值相同，或者都返回指针或引用，并且派生类虚函数所返回的指针或引用的类型是基类中被替换的虚函数所返回的指针或引用的类型的子类型。

注意：

重写的函数所抛出的异常必须和被重写的函数所抛出的异常一致，或者是其子类；

重写的函数的访问修饰符可以不同于被重写的函数，如基类的 virtual 函数的修饰符为 private，派生类改为 public 或 protected 也是可以的。

静态方法不能被重写，也就是 **static 和 virtual 不能同时使用**。

重写的函数可以带 virtual 关键字，也可以不带。

（3）**重定义：**也叫**隐藏**，子类重新定义父类中的非虚函数，屏蔽了父类的同名函数

基本条件：

被隐藏的函数之间作用域不相同

注意：

子类和父类的函数名称相同，但参数不同，此时不管父类函数是不是 virtual 函数，都将被隐藏。

子类和父类的函数名称相同，参数也相同，但是父类函数不是 virtual 函数，父类的函数将被隐藏。

35. 虚函数表是一个类一个还是一个对象一个？如果继承，子类与父类的虚函数表有何区别？虚函数表中可不可以存放非虚函数？（C++）

对 C++ 了解的人都应该知道虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。简称为 V-Table。在这个表中，主是要一个类的虚函数（**一个类有一个虚函数表**）的地址表，这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

这里我们着重看一下这张虚函数表。C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

http://blog.csdn.net/haoel/article/details/1948051

36. 图有几种存储方式？邻接矩阵与邻接表存储结构的优缺点？什么时候用什么结构？（数据结构）

常用的有两种存储方式。邻接矩阵与邻接表。

邻接矩阵的优点：容易实现图的操作，如：求某顶点的度、判断顶点之间是否有边（弧）、找顶点的邻接点等等。

邻接矩阵的缺点：n 个顶点需要 n*n 个单元存储边(弧);空间效率为 O(n^2)。 对稀疏图而言尤其浪费空间。

邻接表的优点：空间效率高；容易寻找顶点的邻接点；

邻接表的缺点：判断任意两顶点间是否有弧或边，需搜索两结点（或之一）对应的单链表，没有邻接矩阵方便。

讨论：**邻接表与邻接矩阵有什么异同**之处？

联系：邻接表中每个链表对应于邻接矩阵中的一行，链表中结点个数等于一行中非零元素的个数。

区别：

① 对于任一确定的图，邻接矩阵是唯一的（行列号与顶点编号一致），但邻接表不唯一（链接次序与顶点编号无关）。

② 邻接矩阵的空间复杂度为 O(n^2),而邻接表的空间复杂度为 O(n+e)。

用途：邻接矩阵多用于稠密图的存储（e 接近 n(n-1)/2)；而邻接表多用于稀疏图的存储（e<<n^2)

37. STL 中有哪些容器？vector 与 deque 有什么区别？（C++）

STL 是 C/C++开发中一个非常重要的模板，而其中定义的各种容器也是非常方便我们大家使用。**STL 中的常用容器**包括：**顺序性容器**（vector、deque、list）、**关联容器**（map、set）、**容器适配器**（queue、stack）。

1、顺序性容器

（1）**vector**

vector 是一种动态数组，在内存中具有连续的存储空间，支持快速随机访问。由于具有连续的存储空间，所以在插入和删除操作方面，效率比较慢。vector 有多个构造函数，

默认的构造函数是构造一个初始长度为 0 的内存空间，且分配的内存空间是以 2 的倍数动态增长的，即内存空间增长是按照 2^0,2^1,2^2,2^3.....增长的，在 push_back 的过程中，若发现分配的内存空间不足，则重新分配一段连续的内存空间，其大小是现在连续空间的 2 倍，再将原先空间中的元素复制到新的空间中，性能消耗比较大，尤其是当元素是非内部数据时(非内部数据往往构造及拷贝构造函数相当复杂)。vector 的另一个常见的问题就是 clear 操作。clear 函数只是把 vector 的 size 清为零，但 vector 中的元素在内存中并没有消除，所以在使用 vector 的过程中会发现内存消耗会越来越多，导致内存泄露，现在经常用的方法是 swap 函数来进行解决：

复制代码代码如下:

```
vector<int> V;
V.push_back(1);
V.push_back(2);
V.push_back(1);
V.push_back(2);
vector<int>().swap(V);
//或者  V.swap(vector<int>());
```

利用 swap 函数，和临时对象交换，使 V 对象的内存为临时对象的内存，而临时对象的内存为 V 对象的内存。交换以后，临时对象消失，释放内存。

（2）**deque**

deque 和 vector 类似，支持快速随机访问。**二者最大的区别**在于，vector 只能在末端插入数据，而 deque 支持双端插入数据。deque 的内存空间分布是小片的连续，小片间用链表相连，实际上内部有一个 map 的指针。deque 空间的重新分配要比 vector 快，重新分配空间后，原有的元素是不需要拷贝的。

（3）list

list 是一个双向链表，因此它的内存空间是可以不连续的，通过指针来进行数据的访问，这使 list 的随机存储变得非常低效，因此 list 没有提供[]操作符的重载。但 list 可以很好地支持任意地方的插入和删除，只需移动相应的指针即可。

（4）在实际使用时，如何选择这三个容器中哪一个，应根据你的需要而定，一般应遵循下面的原则：

1) 如果你需要高效的随即存取，而不在乎插入和删除的效率，使用 vector
2) 如果你需要大量的插入和删除，而不关心随即存取，则应使用 list
3) 如果你需要随即存取，而且关心两端数据的插入和删除，则应使用 deque

2、关联容器

（1）**map**

map 是一种关联容器，该容器用唯一的关键字来映射相应的值，即具有 key-value 功能。map 内部自建一棵红黑树（一种自平衡二叉树），这棵树具有数据自动排序的功能，所以在 map 内部所有的数据都是有序的，以二叉树的形式进行组织。

这是 map 的模板：

```
template < class Key, class T, class Compare= less<Key>, class Allocator=allocator<
pair<const Key,T> > > class map;
```

从模板中我们可以看出，再构造 map 时，是按照一定的顺序进行的。map 的插入和删除效率比其他序列的容器高，因为对关联容器来说，不需要做内存的拷贝和移动，只是指

针的移动。由于 map 的每个数据对应红黑树上的一个节点，这个节点在不保存你的数据时，是占用 16 个字节的，一个父节点指针，左右孩子指针，还有一个枚举值（标示红黑色），所以 map 的其中的一个缺点就是比较占用内存空间。

（2）**set**

set 也是一种关联性容器，它同 map 一样，底层使用红黑树实现，插入删除操作时仅仅移动指针即可，不涉及内存的移动和拷贝，所以效率比较高。set 中的元素都是唯一的，而且默认情况下会对元素进行升序排列。所以在 set 中，不能直接改变元素值，因为那样会打乱原本正确的顺序，要改变元素值必须先删除旧元素，再插入新元素。不提供直接存取元素的任何操作函数，只能通过迭代器进行间接存取。

set 模板原型：

template <class Key, class Compare=class<Key>, class Alloc=STL_DEFAULT_ALLOCATOR(Key) > class set;

set 支持集合的交(set_intersection)、差(set_difference)、并(set_union)及对称差(set_symmetric_difference) 等一些集合上的操作。

3、**容器适配器**

（1）**queue**

queue 是一个队列，实现先进先出功能，queue 不是标准的 STL 容器，却以标准的 STL 容器为基础。queue 是在 deque 的基础上封装的。之所以选择 deque 而不选择 vector 是因为 deque 在删除元素的时候释放空间，同时在重新申请空间的时候无需拷贝所有元素。

其模板为：

template < TYPENAME _Sequence="deque<_TP" typeneam _Tp,> > class queue;

（2）**stack**

stack 是实现先进后出的功能，和 queue 一样，也是内部封装了 deque，这也是为啥称为容器适配器的原因吧（纯属猜测）。自己不直接维护被控序列的模板类，而是它存储的容器对象来为它实现所有的功能。stack 的源代码原理和实现方式均跟 queue 相同。

38. 什么是智能指针？（C++）

智能指针（英语：Smart pointer）是一种抽象的数据类型。在程序设计中，它通常是经由类别模板（class template）来实现，借由模板（template）来达成泛型，通常借由类别（class）的析构函数来达成自动释放指针所指向的内存或对象。

为什么要使用智能指针：我们知道 c++的内存管理是让很多人头疼的事，当我们写一个 new 语句时，一般就会立即把 delete 语句直接也写了，但是我们不能避免程序还未执行到 delete 时就跳转了或者在函数中没有执行到最后的 delete 语句就返回了，如果我们不在每一个可能跳转或者返回的语句前释放资源，就会造成内存泄露。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域时，类会自动调用析构函数，析构函数会自动释放资源。

39. Linux 下/etc/下有哪些配置文件，有什么作用？resolve.conf 是用来做什么的。（Linux）

/etc 目录包含各种系统配置文件

/etc 是 UNIX®（LINUX）系统最基本的目录之一，因为它包含了全部特定主机的配置文件。绝不要为了回收空间而删除它！同样，如果您想要将您的文件树结构拓展到几

个分区时，请记住 /etc 一定不能放到一个另外的分区中。系统初始化需要它，因此它必须在启动时存放于根分区。

以下是其中的几个重要文件：

/etc/passwd

用户数据库，其中的域给出了用户名、真实姓名、用户起始目录、加密口令和用户的其他信息。

/etc/shadow

在安装了影子(shadow)口令软件的系统上的影子口令文件。影子口令文件将/etc/passwd 文件中的加密口令移动到/etc/shadow 中，而后者只对超级用户(root)可读。这使破译口令更困难，以此增加系统的安全性。

/etc/inittab

init 的配置文件。

/etc/profile 、/etc/csh.login、/etc/csh.cshrc

登录或启动时 bourne 或 c shells 执行的文件。这允许系统管理员为所有用户建立全局缺省环境。

passwd 和 shadow：这些文本文件包含所有系统用户及其加密后的密码。只有当您使用 shadow 密码，您才会见到 shadow 文件。出于安全考虑，这是安装时的默认选项。

inittab：这是 init 的配置文件，而它在系统启动时扮演了一个十分重要的角色。

services：该文件保存了现有的网络服务。

profile：这是 shell 的配置文件。某些 shells 使用其他的文件，比如 bash 就使用 .bashrc。

crontab：cron (定期执行命令的程序)的配置文件。

某些需要大量配置文件的程序会将它们保存在某些子目录中。比如 X Window 系统 将其所有配置文件保存于 /etc/X11 目录中

linux 根文件系统 /etc/resolv.conf 文件详解

它是 DNS 客户机配置文件，用于设置 DNS 服务器的 IP 地址及 DNS 域名，还包含了主机的域名搜索顺序。该文件是由域名解析器（resolver，一个根据主机名解析 IP 地址的库）使用的配置文件。它的格式很简单，每行以一个关键字开头，后接一个或多个由空格隔开的参数。

resolv.conf 的关键字主要有四个，分别是：

nameserver //定义 DNS 服务器的 IP 地址

domain //定义本地域名

search //定义域名的搜索列表

sortlist //对返回的域名进行排序

下面我们给出一个/etc/resolv.conf 的示例：

domain 51osos.com

search www.51osos.com 51osos.com

nameserver 202.102.192.68

nameserver 202.102.192.69

最主要是 nameserver 关键字，如果没指定 nameserver 就找不到 DNS 服务器，其它关键字是可选的。

nameserver 表示解析域名时使用该地址指定的主机为域名服务器。其中域名服务器是按照文件中出现的顺序来查询的,且只有当第一个 nameserver 没有反应时才查询下面的 nameserver。

domain　声明主机的域名。很多程序用到它，如邮件系统；当为没有域名的主机进行 DNS 查询时，也要用到。如果没有域名，主机名将被使用，删除所有在第一个点(.)前面的内容。

search　它的多个参数指明域名查询顺序。当要查询没有域名的主机，主机将在由 search 声明的域中分别查找。

domain 和 search 不能共存；如果同时存在，后面出现的将会被使用。

sortlist　允许将得到域名结果进行特定的排序。它的参数为网络/掩码对，允许任意的排列顺序。

40. Linux 下网卡配置在哪修改？（Linux）

http://www.cnblogs.com/shuaixf/archive/2011/11/29/2267863.html

iptables 命令是 Linux 上常用的防火墙软件，是 netfilter 项目的一部分。可以直接配置，也可以通过许多前端和图形界面配置。

41. Linux 的七个运行级别（Linux）

Linux 系统有 7 个运行级别(runlevel)

运行级别 0：系统停机状态，系统默认运行级别不能设为 0，否则不能正常启动

运行级别 1：单用户工作状态，root 权限，用于系统维护，禁止远程登陆

运行级别 2：多用户状态(没有 NFS)

运行级别 3：完全的多用户状态(有 NFS)，登陆后进入控制台命令行模式

运行级别 4：系统未使用，保留

运行级别 5：X11 控制台，登陆后进入图形 GUI 模式

运行级别 6：系统正常关闭并重启，默认运行级别不能设为 6，否则不能正常启动

运行级别的原理(以 ubuntu 说明)：

1　在目录/etc/init.d 下有许多服务器脚本程序，一般称为服务(service)

2　在/etc 下有 7 个名为 rcN.d 的目录，对应系统的 7 个运行级别

3　rcN.d 目录下都是一些符号链接文件，这些链接文件都指向 init.d 目录下的 service 脚本文件，命名规则为 K+nn+服务名或 S+nn+服务名，其中 nn 为两位数字。

4　系统会根据指定的运行级别进入对应的 rcN.d 目录，并按照文件名顺序检索目录下的链接文件

对于以 K 开头的文件，系统将终止对应的服务

对于以 S 开头的文件，系统将启动对应的服务

5　查看运行级别用：runlevel

6　进入其它运行级别用：init N

7　另外 init0 为关机，init 6 为重启系统

42. 内核态与用户态的区别？从用户态切换到内核态有哪几种方式？　（Linux）

2）**特权级**

熟悉 Unix/Linux 系统的人都知道，fork 的工作实际上是以系统调用的方式完成相应功能的，具体的工作是由 sys_fork 负责实施。其实无论是不是 Unix 或者 Linux，对于任何操作系统来说，创建一个新的进程都是属于核心功能，因为它要做很多底层细致地工作，消耗系统的物理资源，比如分配物理内存，从父进程拷贝相关信息，拷贝设置页目录页表等等，这些显然不能随便让哪个程序就能去做，于是就自然引出特权级别的概念，显然，**最关键性的权力必须由高特权级的程序来执行，这样才可以做到集中管理，减少有限资源的访问和使用冲突**。

特权级显然是非常有效的管理和控制程序执行的手段，因此在硬件上对特权级做了很多支持，就 Intel x86 架构的 CPU 来说一共有 0~3 四个特权级，0 级最高，3 级最低，硬件上在

执行每条指令时都会对指令所具有的特权级做相应的检查，相关的概念有CPL、DPL和RPL，这里不再过多阐述。硬件已经提供了一套特权级使用的相关机制，软件自然就是好好利用的问题，这属于操作系统要做的事情，对于 Unix/Linux 来说，只使用了 0 级特权级和 3 级特权级。也就是说**在 Unix/Linux 系统中，一条工作在 0 级特权级的指令具有了 CPU 能提供的最高权力**，而**一条工作在 3 级特权级的指令具有 CPU 提供的最低或者说最基本权力**。

3）用户态和内核态

现在我们从特权级的调度来理解用户态和内核态就比较好理解了，**当程序运行在 3 级特权级上时，就可以称之为运行在用户态**，因为这是**最低特权级，是普通的用户进程运行的特权级**，大部分用户直接面对的程序都是运行在用户态；反之，**当程序运行在 0 级特权级上时，就可以称之为运行在内核态。**

虽然用户态下和内核态下工作的程序有很多差别，但**最重要的差别就在于特权级的不同，**即权力的不同。运行在用户态下的程序不能直接访问操作系统内核数据结构和程序，比如上面例子中的 testfork()就不能直接调用 sys_fork()，因为前者是工作在用户态，属于用户态程序，而 sys_fork()是工作在内核态，属于内核态程序。

当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态，比如 testfork()最初运行在用户态进程下，当它调用 fork()最终触发 sys_fork()的执行时，就切换到了内核态。

2. 用户态和内核态的转换（Linux）

1）用户态切换到内核态的 3 种方式

a. 系统调用

这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中 fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如 Linux 的 int 80h 中断。

b. 异常

当 CPU 在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

c. 外围设备的中断

当外围设备完成用户请求的操作后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

这 3 种方式是系统在运行时由用户态转到内核态的最主要方式，其中**系统调用可以认为是用户进程主动**发起的，**异常和外围设备中断则是被动**的。

2）具体的切换操作

从触发方式上看，可以认为存在前述 3 种不同的类型，但是从最终实际完成由用户态到内核态的切换操作上来说，涉及的关键步骤是完全一致的，没有任何区别，都相当于执行了一个中断响应的过程，因为**系统调用实际上最终是中断机制实现的**，而异常和中断的处理机制基本上也是一致的，关于它们的具体区别这里不再赘述。关于中断处理机制的细节和步骤这里也不做过多分析，涉及到由用户态切换到内核态的步骤主要包括：

[1] 从当前进程的描述符中提取其内核栈的 ss0 及 esp0 信息。

[2] 使用 ss0 和 esp0 指向的内核栈将当前进程的 cs,eip,eflags,ss,esp 信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。

[3] 将先前由中断向量检索得到的中断处理程序的 cs,eip 信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。

43. 深究递归和迭代的区别、联系、优缺点及实例对比（C++）

1.概念区分

递归的基本概念:程序调用自身的编程技巧称为递归,是函数自己调用自己.

一个函数在其定义中直接或间接调用自身的一种方法,它通常把一个大型的复杂的问题转化为一个与原问题相似的规模较小的问题来解决,可以极大的减少代码量.递归的能力在于用有限的语句来定义对象的无限集合.

使用递归要注意的有两点:

1)递归就是在过程或函数里面调用自身;

2)在使用递归时,必须有一个明确的递归结束条件,称为递归出口.

递归分为两个阶段:

1)递推:把复杂的问题的求解推到比原问题简单一些的问题的求解;

2)回归:当获得最简单的情况后,逐步返回,依次得到复杂的解.

利用递归可以解决很多问题:如背包问题,汉诺塔问题,...等.

斐波那契数列为:0,1,1,2,3,5...

由于递归引起一系列的函数调用,并且有可能会有一系列的重复计算,递归算法的执行效率相对较低.

迭代:利用变量的原值推算出变量的一个新值.如果递归是自己调用自己的话,迭代就是 A 不停的调用 B.

2.辩证看递归和迭代

所谓递归，简而言之就是应用程序自身调用自身，以实现层次数据结构的查询和访问。递归的使用可以使代码更简洁清晰，可读性更好（对于初学者到不见得），但由于递归需要系统堆栈，所以空间消耗要比非递归代码要大很多，而且，如果递归深度太大，可能系统资源会不够用。

往往有这样的观点：能不用递归就不用递归，递归都可以用迭代来代替。

诚然，在理论上，递归和迭代在时间复杂度方面是等价的（在不考虑函数调用开销和函数调用产生的堆栈开销），但实际上递归确实效率比迭代低，既然这样，递归没有任何优势，那么是不是就，没有使用递归的必要了，那递归的存在有何意义呢？

万物的存在是需要时间的检验的，递归没有被历史所埋没，即有存在的理由。从理论上说，所有的递归函数都可以转换为迭代函数，反之亦然，然而代价通常都是比较高的。但从算法结构来说，递归声明的结构并不总能够转换为迭代结构，原因在于结构的引申本身属于递归的概念，用迭代的方法在设计初期根本无法实现，这就像动多态的东西并不总是可以用静多态的方法实现一样。这也是为什么在结构设计时，通常采用递归的方式而不是采用迭代的方式的原因，一个极典型的例子类似于链表，使用递归定义及其简单，但对于内存定义(数组方式)其定义及调用处理说明就变得很晦涩，尤其是在遇到环链、图、网格等问题时，使用迭代方式从描述到实现上都变得不现实。因而可以从实际上说，所有的迭代可以转换为递归，但递归不一定可以转换为迭代。

采用递归算法需要的前提条件是，当且仅当一个存在预期的收敛时，才可采用递归算法，否则，就不能使用递归算法。

递归其实是方便了程序员难为了机器，递归可以通过数学公式很方便的转换为程序。其优点就是易理解，容易编程。但递归是用栈机制实现的，每深入一层，都要占去一块栈数据区域，

对嵌套层数深的一些算法，递归会力不从心，空间上会以内存崩溃而告终，而且递归也带来了大量的函数调用，这也有许多额外的时间开销。所以在深度大时，它的时空性就不好了。而迭代虽然效率高，运行时间只因循环次数增加而增加，没什么额外开销，空间上也没有什么增加，但缺点就是不容易理解，编写复杂问题时困难。
因而，“能不用递归就不用递归，递归都可以用迭代来代替”这样的理解，还是辩证的来看待，不可一棍子打死。*/
3.个人总结

	定义	优点	缺点
递归	程序调用自身的编程技巧称为递归	1）大问题化为小问题,可以极大的减少代码量； 2）用有限的语句来定义对象的无限集合.； 3）代码更简洁清晰，可读性更好	1）递归调用函数,浪费空间； 2）递归太深容易造成堆栈的溢出；
迭代	利用变量的原值推算出变量的一个新值，迭代就是 A 不停的调用 B.	1）迭代效率高，运行时间只因循环次数增加而增加； 2）没什么额外开销，空间上也没有什么增加，	1） 不容易理解； 2） 代码不如递归简洁； 3） 编写复杂问题时困难。
二者关系	1） 递归中一定有迭代,但是迭代中不一定有递归,大部分可以相互转换。 2） 能用迭代的不用递归,递归调用函数,浪费空间,并且递归太深容易造成堆栈的溢出./*相对*/		

Const 的作用（C++）

Const 是 C++中常用的类型修饰符,**常类型是指使用类型修饰符 const 说明的类型，常类型的变量或对象的值是不能被更新的。**

No.	作用	说明	参考代码
1	可以定义const常量		const int Max = 100;
2	便于进行类型检查	const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误	void f(const int i) { .....} //对传入的参数进行类型检查，不匹配进行提示

3	可以保护被修饰的东西	防止意外的修改，增强程序的健壮性。	void f(const int i) { i=10;//error! }
			//如果在函数体内修改了i，编译器就会报错

4	可以很方便地进行参数的调整和修改	同宏定义一样，可以做到不变则已，一变都变	

5	为函数重载提供了一个参考		class A
			{
			.....
			void f(int i) {.....} //一个函数
			void f(int i) const {.....} //上一个函数的重载
			.....
			};

6	可以节省空间，避免不必要的内存分配	const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝	#define PI 3.14159 //常量宏
			const doulbe Pi=3.14159; //此时并未将Pi放入ROM中
			.....
			double i=Pi; //此时为Pi分配内存，以后不再分配!
			double I=PI; //编译期间进行宏替换，分配内存
			double j=Pi; //没有内存分配
			double J=PI; //再进行宏替换，又一次分配内存!

7	提高了效率	编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高	

静态成员函数有特点呢？（C++）

1.静态成员之间可以相互访问，包括静态成员函数访问静态数据成员和访问静态成员函数；

2.非静态成员函数可以任意地访问静态成员函数和静态数据成员；

3.静态成员函数不能访问非静态成员函数和非静态数据成员；

4.调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指针调用静态成员函数,也可以用类名::函数名调用(因为他本来就是属于类的，用类名调用很正常)

44. 指针与引用的区别？常用指针还是引用？（C++）

1.指针是一个变量，程序会为指针分配内存，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。

2.指针可以有多级，但是引用只能是一级；

3.指针的值可以为空，也可能指向一个不确定的内存空间，但是引用的值不能为空，并且引用在定义的时候必须初始化为特定对象；（因此引用更安全）

4.指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变引用对象了；

5.sizeof 引用得到的是所指向的变量(对象)的大小，而 sizeof 指针得到的是指针本身的大小；

6.指针和引用的自增(++)运算意义不一样；

7.引用不能 const，指针能用 const，const 的指针不可变。

8.指针传递和引用传递（C++）

指针传递参数本质上是值传递的方式，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，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**。

引用传递过程中，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。**

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。

先理解值传递：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，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拷贝。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。

指针传递本质上值传递，只不过它所传递的是一个地址值。然后把上面那段话「翻译」一下：指针传递时，形参是一个指针变量，该变量拷贝了实参的地址值，然后作为被调函数的局部变量（传递过来的实参的地址值不会变），再然后我们可以用*操作符访问实参，从而对实参进行操作。

在引用传递过程中，被调函数的形式参数虽然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（不必通过*操作符），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（此时的形参其实就是实参）。

例子见 http://geekplux.com/2013/01/02/pointer_references.html

45. 常使用指针还是引用（C++）

More Effective C++ 一书的总结

这本书的第一条就是区分指针和引用。它们**两者之间的最大区别是引用必须指向某个对象而指针可以是 NULL，此外引用一旦指定不能更改而指针可以**。

这两个区别点导致引用有更加安全和高效的特性，但是指针却有无可比拟的灵活性。大部分人出于安全性的考虑会推荐使用引用，这其实也是它设计的主要目的，但是如果你想要灵活

的设计，大部分时候你只能选用指针，比如设计模式种的大部分设计都是使用指针而不是使用引用。引用在参数传递的时候用得多一些，而类内部的组合中可能会使用指针来提高设计的灵活性（毕竟一旦设定就无法改变对于灵活性来说是个灾难）。

网上观点的总结#

1. 尽量避免使用指针，可以用引用的时候尽量不要使用指针。
2. 指针参数可以在调用的时候传递 NULL 而引用则不可以。所以如果你的参数是可选的话选择传递指针。
3. 指针参数在调用的时候会比引用要明显一些：

int fun(val);

int fun(&val);

前者比较难以看出是传递引用还是直接传递值，而第二个很明显是传递指针。

4. 如果你需要在函数中重新绑定改变参数，你只能用指针。因为你没有办法重新绑定一个引用。不过需要这么做的情况好像比较少。
6. 如果你的参数需要传递数组的话，你只能使用指针。
7. 其他情况下尽可能的使用引用。因为引用从语义上来说更直白一些，也更不容易出错。引用一定是指向一个合法的对象，而指针需要在使用之前检查是否为 NULL
8. 还有一些人觉得参数的传递如果是传递引用的话只使用 const refercence，把引用的作用限制在避免参数拷贝的开销上。然后把改变变量内容的任务交给指针。这也是一个非常不错的建议。

46. C++虚析构函数的作用，如果析构函数不是析构函数有什么后果？（C++）

为了**当用一个基类的指针删除一个派生类的对象时，派生类的析构函数会被调用**。

当基类的析构函数为虚函数时，无论指针指的是同一类族中的哪一个类对象，系统会采用动态关联，调用相应的析构函数，对该对象进行清理工作。

如果将基类的析构函数声明为虚函数时，由该基类所派生的所有派生类的析构函数也都自动成为虚函数，即使派生类的析构函数与基类的析构函数名字不相同。

最好把基类的析构函数声明为虚函数。这将使所有派生类的析构函数自动成为虚函数。这样，如果程序中显式地用了 delete 运算符准备删除一个对象，而 delete 运算符的操作对象用了**指向派生类对象的基类指针**，则系统会调用相应类的析构函数，先调用派生类析构函数，再调用基类析构函数。

专业人员一般都习惯声明虚析构函数，即使基类并不需要析构函数，也显式地定义一个函数体为空的虚析构函数，以保证在撤销动态分配空间时能得到正确的处理。

构造函数不能声明为虚函数。这是因为在**执行构造函数时类对象还未完成建立过程**，当然谈不上函数与类对象的绑定。

一般情况下类的析构函数里面都是释放内存资源，而析构函数不被调用的话就会造成内存泄漏。

当然，并不是要把所有类的析构函数都写成虚函数。因为当类里面有虚函数的时候，编译器会给类添加一个虚函数表，里面来存放虚函数指针，这样就会增加类的存储空间。所以，只有当一个类被用来作为基类的时候，才把析构函数写成虚函数。

47. C++虚基类是什么？纯虚函数可不可以有实现？什么情况下会对纯虚函数进行实现？（C++）

当在多条继承路径上有一个公共的基类,在这些路径中的某几条汇合处，这个公共的基类就会产生多个实例(或多个副本)，若只想保存这个基类的一个实例，可以将这个**公共基类说明为虚基类**。

在继承中产生歧义的原因有可能是继承类继承了基类多次，从而产生了多个拷贝，即不止一次的通过多个路径继承类在内存中创建了基类成员的多份拷贝。虚基类的基本原则是在内存中只有基类成员的一份拷贝。这样，通过把基类继承声明为虚拟的，就只能继承基类的一份拷贝，从而消除歧义。用 virtual 限定符把基类继承说明为虚拟的。

http://c.biancheng.net/cpp/biancheng/view/238.html 多继承和虚基类

http://www.cnblogs.com/fly1988happy/archive/2012/09/25/2701237.html C++ 虚函数&纯虚函数&抽象类&接口&虚基类

纯虚函数

许多情况下，在基类中不能对虚函数给出有意义的实现，则把它声明为纯虚函数，它的实现留给该基类的派生类去做。

纯虚函数的声明格式：virtual <函数返回类型说明符> <函数名> (<参数表>)=0;

纯虚函数的作用是为派生类提供一个一致的接口。

纯虚函数可以有实现，具有函数体的纯虚函数同时还提供了缺省实现。

48. hash 表中为了防止冲突过多常用素数，为什么？（数据结构）

首先来说假如关键字是随机分布的，那么无所谓一定要模质数。但在实际中往往关键字有某种规律，例如大量的等差数列，那么公差和模数不互质的时候发生碰撞的概率会变大，而用质数就可以很大程度上回避这个问题。

49. static 的作用（C++）

1、隐藏：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。

static 可以用作函数和变量的前缀，对于函数来讲，static 的作用仅限于隐藏.

2、static 的第二个作用是保持变量内容的持久：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。虽然这种用法不常见

3、static 的第三个作用是默认初始化为 0（static 变量）

4、C++中的作用

1）不能将静态成员函数定义为虚函数。

2）静态数据成员是静态存储的，所以必须对它进行初始化。（程序员手动初始化，否则编译时一般不会报错，但是在 Link 时会报错误）

3）静态数据成员在<定义或说明>时前面加关键字 static。

50. 两个文件，一个是 class A，一个是 class B，如果 A 跟 B 相互引用，如何 include？（C++）

一、问题描述

现在有两个类 A 和 B 需要定义，定义 A 的时候需要用到 B，定义 B 的时候需要用到 A。

二、分析

A 和 B 的定义和调用都放在一个文件中肯定是不可以的，这样就会造成两个循环调用的死循环。

根本原因是：定义 A 的时候，A 的里面有 B，所以就需要去查看 B 的占空间大小，但是查看的时候又发现需要知道 A 的占空间大小，造成死循环。

解决方法：

（1）写两个头文件 A.h 和 B.h 分别用于声明类 A 和 B;

（2）写两个.cpp 文件分别用于定义类 A 和 B;

（3）在 A 的头文件中导入 B 的头文件；
（4）在 B 的头文件中不导入 A 的头文件，但是用 extern 的方式声明类 A，并且，在 B 中使用 A 的时候要用指针的形式。
原理：在 B 中用指针调用 A，那么在 A 需要知道 B 占空间大小的时候，就会去找到 B 的定义文件，虽然 B 的定义文件中并没有导入 A 的头文件，不知道 A 的占空间大小，但是由于在 B 中调用 A 的时候用的指针形式，B 只知道指针占 4 个字节就可以，不需要知道 A 真正占空间大小，也就是说，A 也是知道 B 的占空间大小的。

代码见 http://blog.csdn.net/xiqingnian/article/details/41214539

51. STL vector 的扩容问题（C++）

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；

扩容时的操作流程为：开辟新内存 -> copy 数据 -> 释放旧内存。因此频繁导致 vector 扩容(如 for 循环持续 push_back)会使得程序效率降低。因此，如有需要，可以提前通过初始化或者 resize、reserve 来预先开辟较大的容量。

vector 在 push_back 以成倍增长可以在均摊后达到 O(1)常数的时间复杂度，相对于增长指定大小的 O(n)时间复杂度更好。

为了防止申请内存的浪费，现在使用较多的有 2 倍与 1.5 倍的增长方式，而 1.5 倍的增长方式可以更好的实现对内存的重复利用，因此更好。

默认在 vector 析构的时候才会清空所有内存。如果想要提前释放掉 vector 开辟的内存，可以使其与一个空 vector 进行交换，如下：

```
// 创建一个 vector
vector<int> data;
for(int i = 0; i < 10; ++i)
    data.push_back(i);
// 与空 vector 进行交换
    vector<int>().swap(data);
// 或者
    data.swap(vector<int>());
```

52. 堆和栈的区别（C++）

从**内存角度**来说：**栈区**（stack）由编译器自动分配释放，存放函数的参数值，局部变变量的值等，其操作方式类似于数据结构中的栈，**可静态亦可动态分配**。

堆区（heap）一般由程序员分配释放，若程序员不释放，可能造成内存泄漏，程序结束时可能由 OS 回收。**只可动态分配**，分配方式类似于链表。

从**数据结构**角度来说：堆和栈都是一种数据项按序排列的数据结构。

堆像一棵倒过来的树

堆是一种经过排序的树形数据结构，每个结点都有一个值。

通常我们所说的堆的数据结构，是指二叉堆。

堆的特点是根结点的值最小（或最大），且根结点的两个子树也是一个堆。

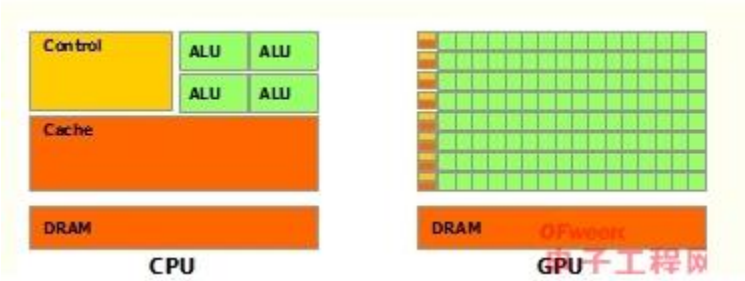
由于堆的这个特性，常用来**实现优先队列**。

而**栈**是一种先进后出的数据结构。

53. CPU 和 GPU 的区别（计算机系统结构）

CPU 和 GPU 之所以大不相同，是由于其**设计目标**的不同，它们分别针对了两种不同的应用场景。**CPU 需要很强的通用性来处理各种不同的数据类型，同时又要逻辑判断又会引入大量的分支跳转和中断的处理**。这些都使得 **CPU 的内部结构异常复杂**。而 **GPU 面对的则是类型高度统一的、相互无依赖的大规模数据和不需要被打断的纯净的计算环境**。

于是 CPU 和 GPU 就呈现出非常不同的架构（示意图）：



GPU 采用了数量众多的计算单元和超长的流水线，但只有非常简单的控制逻辑并省去了 Cache。而 CPU 不仅被 Cache 占据了大量空间，而且还有有复杂的控制逻辑和诸多优化电路，相比之下计算能力只是 CPU 很小的一部分。

所以与 **CPU 擅长逻辑控制和通用类型数据运算**不同，**GPU 擅长的是大规模并发计算**，这也正是密码破解等所需要的。所以 GPU 除了图像处理，也越来越多的参与到计算当中来。

CPU 和 GPU 因为最初用来处理的任务就不同，所以设计上有不小的区别。而某些任务和 GPU 最初用来解决的问题比较相似，所以用 GPU 来算了。GPU 的运算速度取决于雇了多少小学生，CPU 的运算速度取决于请了多么厉害的教授。教授处理复杂任务的能力是碾压小学生的，但是对于没那么复杂的任务，还是顶不住人多。当然现在的 GPU 也能做一些稍微复杂的工作了，相当于升级成初中生高中生的水平。但还需要 CPU 来把数据喂到嘴边才能开始干活，究竟还是靠 CPU 来管的。

什么类型的程序**适合在 GPU 上运行**？

（1）**计算密集型的程序**。所谓计算密集型(Compute-intensive)的程序，就是其大部分运行时间花在了寄存器运算上，寄存器的速度和处理器的速度相当，从寄存器读写数据几乎没有延时。可以做一下对比，读内存的延迟大概是几百个时钟周期；读硬盘的速度就不说了，即便是 SSD, 也实在是太慢了。

（2）**易于并行的程序**。GPU 其实是一种 SIMD(Single Instruction Multiple Data)架构， 他有成百上千个核，每一个核在同一时间最好能做同样的事情。

54. 熟悉的 Linux 命令　查看磁盘空间的命令，如何找到一个文件中含有 aaa 和 不含有 bbb 的行

利用 Linux 所提供的管道符“|”将两个命令隔开，管道符左边命令的输出就会作为管道符右边命令的输入。连续使用管道意味着第一个命令的输出会作为第二个命令的输入，第二个命令的输出又会作为第三个命令的输入，依此类推。（Linux）

grep -rl “aaa” * | grep -v “bbb”

1、cd 命令

这是一个非常基本，也是大家经常需要使用的命令，它用于切换当前目录，它的参数是要切换到的目录的路径，可以是绝对路径，也可以是相对路径。如：

```
1. cd /root/Docements # 切换到目录/root/Docements
2. cd ./path      # 切换到当前目录下的 path 目录中，“.”表示当前目录
3. cd ../path     # 切换到上层目录中的 path 目录中，“..”表示上一层目录
```

2、ls 命令

这是一个非常有用的查看文件与目录的命令，list 之意，它的参数非常多，下面就列出一些我常用的参数吧，如下：

1. -l ：列出长数据串，包含文件的属性与权限数据等
2. -a ：列出全部的文件，连同隐藏文件（开头为.的文件）一起列出来（常用）
3. -d ：仅列出目录本身，而不是列出目录的文件数据
4. -h ：将文件容量以较易读的方式（GB，kB 等）列出来
5. -R ：连同子目录的内容一起列出（递归列出），等于该目录下的所有文件都会显示出来
1. ls -l #以长数据串的形式列出当前目录下的数据文件和目录
2. ls -lR #以长数据串的形式列出当前目录下的所有文件

3、grep 命令

该命令常用于分析一行的信息，若当中有我们所需要的信息，就将该行显示出来，该命令通常与管道命令一起使用，用于对一些命令的输出进行筛选加工等等，它的简单语法为

grep [-acinv] [--color=auto] '查找字符串' filename

1. -a ：将 binary 文件以 text 文件的方式查找数据
2. -c ：计算找到‘查找字符串’的次数
3. -i ：忽略大小写的区别，即把大小写视为相同
4. -v ：反向选择，即显示出没有‘查找字符串’内容的那一行
5. # 例如：
6. # 取出文件/etc/man.config 中包含 MANPATH 的行，并把找到的关键字加上颜色
7. grep --color=auto 'MANPATH' /etc/man.config
8. # 把 ls -l 的输出中包含字母 file（不区分大小写）的内容输出
9. ls -l | grep -i file

4、find 命令

find 是一个基于查找的功能非常强大的命令，相对而言，它的使用也相对较为复杂，参数也比较多，所以在这里将给把它们分类列出，它的基本语法如下：

1. find [PATH] [option] [action]
2.
3. # 与时间有关的参数：
4. -mtime n : n 为数字，意思为在 n 天之前的“一天内”被更改过的文件；
5. -mtime +n : 列出在 n 天之前（不含 n 天本身）被更改过的文件名；
6. -mtime -n : 列出在 n 天之内（含 n 天本身）被更改过的文件名；
7. -newer file : 列出比 file 还要新的文件名
8. # 例如：
9. find /root -mtime 0 # 在当前目录下查找今天之内有改动的文件
10.
11. # 与用户或用户组名有关的参数：
12. -user name : 列出文件所有者为 name 的文件
13. -group name : 列出文件所属用户组为 name 的文件
14. -uid n : 列出文件所有者为用户 ID 为 n 的文件
15. -gid n : 列出文件所属用户组为用户组 ID 为 n 的文件
16. # 例如：
17. find /home/ljianhui -user ljianhui # 在目录/home/ljianhui 中找出所有者为 ljianhui 的文件
18.

```
# 与文件权限及名称有关的参数：
-name filename ：找出文件名为 filename 的文件
-size [+-]SIZE ：找出比 SIZE 还要大（+）或小（-）的文件
-tpye TYPE ：查找文件的类型为 TYPE 的文件，TYPE 的值主要有：一般文件（f)、设备文件（b、c）、
        目录（d）、连接文件（l）、socket（s）、FIFO 管道文件（p）；
-perm mode ：查找文件权限刚好等于 mode 的文件，mode 用数字表示，如 0755；
-perm -mode ：查找文件权限必须要全部包括 mode 权限的文件，mode 用数字表示
-perm +mode ：查找文件权限包含任一 mode 的权限的文件，mode 用数字表示
# 例如：
find / -name passwd # 查找文件名为 passwd 的文件
find . -perm 0755 # 查找当前目录中文件权限的 0755 的文件
find . -size +12k # 查找当前目录中大于 12KB 的文件，注意 c 表示 byte
```

5、cp 命令

该命令用于复制文件，copy 之意，它还可以把多个文件一次性地复制到一个目录下，它的常用参数如下：

```
-a ：将文件的特性一起复制
-p ：连同文件的属性一起复制，而非使用默认方式，与-a 相似，常用于备份
-i ：若目标文件已经存在时，在覆盖时会先询问操作的进行
-r ：递归持续复制，用于目录的复制行为
-u ：目标文件与源文件有差异时才会复制
cp -a file1 file2 #连同文件的所有特性把文件 file1 复制成文件 file2
cp file1 file2 file3 dir #把文件 file1、file2、file3 复制到目录 dir 中
```

6、mv 命令

该命令用于移动文件、目录或更名，move 之意，它的常用参数如下：

```
-f ：force 强制的意思，如果目标文件已经存在，不会询问而直接覆盖
-i ：若目标文件已经存在，就会询问是否覆盖
-u ：若目标文件已经存在，且比目标文件新，才会更新
```

注：该命令可以把一个文件或多个文件一次移动一个文件夹中，但是最后一个目标文件一定要是“目录”。

```
mv file1 file2 file3 dir # 把文件 file1、file2、file3 移动到目录 dir 中
mv file1 file2 # 把文件 file1 重命名为 file2
```

7、rm 命令

该命令用于删除文件或目录，remove 之意，它的常用参数如下：

```
-f ：就是 force 的意思，忽略不存在的文件，不会出现警告消息
-i ：互动模式，在删除前会询问用户是否操作
-r ：递归删除，最常用于目录删除，它是一个非常危险的参数
rm -i file # 删除文件 file，在删除之前会询问是否进行该操作
rm -fr dir # 强制删除目录 dir 中的所有文件
```

8、ps 命令

该命令用于将某个时间点的进程运行情况选取下来并输出，process 之意，它的常用参数如下：

```
-A ：所有的进程均显示出来
-a ：不与 terminal 有关的所有进程
```

3. -u：有效用户的相关进程
4. -x：一般与 a 参数一起使用，可列出较完整的信息
5. -l：较长，较详细地将 PID 的信息列出

其实我们只要记住 ps 一般使用的命令参数搭配即可，它们并不多，如下：

```
1. ps aux # 查看系统所有的进程数据
2. ps ax # 查看不与 terminal 有关的所有进程
3. ps -lA # 查看系统所有的进程数据
4. ps axjf # 查看连同一部分进程树状态
```

9、kill 命令

该命令用于向某个工作（%jobnumber）或者是某个 PID（数字）传送一个信号，它通常与 ps 和 jobs 命令一起使用，它的基本语法如下：

```
kill -signal PID
```

signal 的常用参数如下：

注：最前面的数字为信号的代号，使用时可以用代号代替相应的信号。

1. 1：SIGHUP，启动被终止的进程
2. 2：SIGINT，相当于输入 ctrl+c，中断一个程序的进行
3. 9：SIGKILL，强制中断一个进程的进行
4. 15：SIGTERM，以正常的结束进程方式来终止进程
5. 17：SIGSTOP，相当于输入 ctrl+z，暂停一个进程的进行

```
1. # 以正常的结束进程方式来终止第一个后台工作，可用 jobs 命令查看后台中的第一个工作进程
2. kill -SIGTERM %1
3. # 重新改动进程 ID 为 PID 的进程，PID 可用 ps 命令通过管道命令加上 grep 命令进行筛选获得
4. kill -SIGHUP PID
```

10、killall 命令

该命令用于向一个命令启动的进程发送一个信号，它的一般语法如下：

```
killall [-iIe] [command name]
```

它的参数如下：

```
1. -i：交互式的意思，若需要删除时，会询问用户
2. -e：表示后面接的 command name 要一致，但 command name 不能超过 15 个字符
3. -I：命令名称忽略大小写
4. # 例如：
5. killall -SIGHUP syslogd # 重新启动 syslogd
```

11、file 命令

该命令用于判断接在 file 命令后的文件的基本数据，因为在 Linux 下文件的类型并不是以后缀为分的，所以这个命令对我们来说就很有用了，它的用法非常简单，基本语法如下：

```
1. file filename
2. #例如：
3. file ./test
```

12、tar 命令

该命令用于对文件进行打包，默认情况并不会压缩，如果指定了相应的参数，它还会调用相应的压缩程序（如 gzip 和 bzip 等）进行压缩和解压。它的常用参数如下：

1. -c：新建打包文件

```
-t：查看打包文件的内容含有哪些文件名
-x ：解打包或解压缩的功能，可以搭配-C（大写）指定解压的目录，注意-c,-t,-x 不能同时出现在同一条命令中
-j ：通过 bzip2 的支持进行压缩/解压缩
-z ：通过 gzip 的支持进行压缩/解压缩
-v ：在压缩/解压缩过程中，将正在处理的文件名显示出来
-f filename ： filename 为要处理的文件
-C dir ：指定压缩/解压缩的目录 dir
```

通常我们只需要记住下面三条命令即可：

```
压缩：tar -jcv -f filename.tar.bz2 要被处理的文件或目录名称
查询：tar -jtv -f filename.tar.bz2
解压：tar -jxv -f filename.tar.bz2 -C 欲解压缩的目录
```

注：文件名并不定要以后缀 tar.bz2 结尾，这里主要是为了说明使用的压缩程序为 bzip2

13、cat 命令

该命令用于查看文本文件的内容，后接要查看的文件名，通常可用管道与 more 和 less 一起使用，从而可以一页页地查看数据。例如：

```
cat text | less # 查看 text 文件中的内容
# 注：这条命令也可以使用 less text 来代替
```

14、chgrp 命令

该命令用于改变文件所属用户组，它的使用非常简单，它的基本用法如下：

```
chgrp [-R] dirname/filename
-R ：进行递归的持续对所有文件和子目录更改
# 例如：
chgrp users -R ./dir # 递归地把 dir 目录下中的所有文件和子目录下所有文件的用户组修改为 users
```

15、chown 命令

该命令就是改变文件拥有者和所在用户组。每个文件都属于一个用户组和一个用户。在你的目录下，使用"ls -l",你就会看到像这样的东西。

```
root@tecmint:~# ls -l

drwxr-xr-x 3 server root 4096 May 10 11:14 Binary
drwxr-xr-x 2 server server 4096 May 13 09:42 Desktop
```

在这里，目录 Binary 属于用户"server",和用户组"root",而目录"Desktop"属于用户“server”和用户组"server"

“chown”命令用来改变文件的所有权，所以仅仅用来管理和提供文件的用户和用户组授权。

```
root@tecmint:~# chown server:server Binary

drwxr-xr-x 3 server server 4096 May 10 11:14 Binary
drwxr-xr-x 2 server server 4096 May 13 09:42 Desktop
```

16、chmod 命令

该命令用于改变文件的权限，一般的用法如下：

```
1. chmod [-R] xyz 文件或目录
2. -R：进行递归的持续更改，即连同子目录下的所有文件都会更改
```

同时，chmod 还可以使用 u（user）、g（group）、o（other）、a（all）和+（加入）、-（删除）、=（设置）跟 rwx 搭配来对文件的权限进行更改。

```
1. # 例如：
2. chmod 0755 file # 把 file 的文件权限改变为-rxwr-xr-x
3. chmod g+w file # 向 file 的文件权限中加入用户组可写权限
4. Read (r)=4
5. Write(w)=2
6. Execute(x)=1
```

3 种用户和用户组权限。**第一个是拥有者，然后是用户所在的组，最后是其它用户**。

18、vim 命令

该命令主要用于文本编辑，它接一个或多个文件名作为参数，如果文件存在就打开，如果文件不存在就以该文件名创建一个文件。vim 是一个非常好用的文本编辑器，它里面有很多非常好用的命令。

19. ping

查看网络是连通

> ping 127.16.8.66　　#查看当前机器与目录机器是否能正常通信(主要是看 TCP/IP 协议是否正常)

20. ipconfig

查看 ip 地址

> ipconfig -all　　#可以看到各个网络适配器(本地网卡、无线网卡)的 IP 地址和 MAC 地址

21. df

df 命令是 Linux 查看磁盘空间系统以磁盘分区为单位查看文件系统，可以加上参数查看磁盘剩余空间信息，命令格式：

df -hl

```
df -hl 查看磁盘剩余空间

df -h 查看每个根路径的分区大小

du -sh [目录名] 返回该目录的大小

du -sm [文件夹] 返回该文件夹总 M 数
```

55. **白盒测试和黑盒测试方法（软件测试）**

简单来说，**白盒测试指的是代码层级的测试，黑盒测试指的是针对软件的功能进行测试，**不涉及代码逻辑结构。常见的**黑盒测试方法为等价类划分、边界值划分、因果图法、场景法**等。**白盒测试方法可分为逻辑覆盖、循环覆盖和基本路径测试**。

56. **顺序表和链表的区别（数据结构）**

数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要快速访问数据，很少或不插入和删除元素，就应该用数组。

链表恰好相反，链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。比如：上一个元素有个指针指到下一个元素，以此类推，直到最后一个元素。如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表数据结构了

一：**顺序表**的特点是逻辑上相邻的数据元素，物理存储位置也相邻，并且顺序表的存储空间需要预先分配。

它的优点是：

（1）方法简单，各种高级语言中都有数组，容易实现。

（2）不用为表示节点间的逻辑关系而增加额外的存储开销。

（3）顺序表具有按元素序号随机访问的特点。

缺点：

（1）在顺序表中做插入、删除操作时，平均移动表中的一半元素，因此对 n 较大的顺序表效率低。

（2）需要预先分配足够大的存储空间，估计过大，可能会导致顺序表后部大量闲置；预先分配过小，又会造成溢出。

二、在链表中逻辑上相邻的数据元素，物理存储位置不一定相邻，它使用指针实现元素之间的逻辑关系。并且，链表的存储空间是动态分配的。

链表的最大特点是：

插入、删除运算方便。

缺点：

（1）要占用额外的存储空间存储元素之间的关系，存储密度降低。存储密度是指一个节点中数据元素所占的存储单元和整个节点所占的存储单元之比。

（2）链表不是一种随机存储结构，不能随机存取元素。

57. OSI 七层协议分别是什么，每一层功能 （计算机网络）

OSI 模型，即开放式通信系统互联参考模型（Open System Interconnection Reference Model），是国际标准化组织提出的一个试图使各种计算机在世界范围内互连为网络的标准框架，简称 OSI。

OSI 分层（7 层）：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层

TCP/IP 分层（4 层）：网络接口层、网际层、传输层、应用层

五层协议：物理层、数据链路层、网络层、传输层、应用层

每层的作用及协议：

物理层：通过媒介传输比特，确定机械及电气规范。RJ45,CLOCK,IEEE802.3（中继器、集线器）

数据链路层：将比特组装成帧和点到点的传递帧。PPP,FR,VLAN,MAC（网桥、交换机）

网络层：负责数据包从源到目的端的传递和网际互联（包 packet）,路由寻址。IP,ICMP,ARP,RARP,OSPF,IPX,RIP,IGRP（路由器）

传输层：提供端到端的可靠报文传递和错误恢复（段 Segment）。TCP,UDP

会话层：建立、管理和终止会话（会话协议数据单元 SPDU）。NFS,SQL,NETBIOS,RPC

表示层：对数据进行翻译、加密和压缩（表示协议数据单元 PPDU）。JPEG，MPEG

应用层：允许访问 OSI 环境的手段（应用协议数据单元 APDU）。FTP,DNS,Telnet,SMTP,HTTP,WWW,NFS

58. 数据完整性（数据库）

数据完整性可以分为四类。

1、实体完整性，实体完整性的目的是确保数据库中所有实体的唯一性，也就是不应出现完全相同的数据记录。

2、区域完整性，匹配完整性要求数据表中的数据位于某一个特定的允许范围内。

3、参考完整性，是用来维护相关数据表之间数据一致性的手段，通过实现参考完整性，可以避免因一个数据表的记录改变而造成另一个数据表内的数据变成无效值。

4、用户自定义完整性，用户自定义由用户根据实际应用中的需要自行定义。

59. 会出现内存泄漏的情况（C++）

不再用到的内存，没有及时释放，就叫做**内存泄漏。内存溢出(out of memory)：**程序在申请内存时，没有足够的内存空间供其使用。一般发生内存溢出时，程序讲无法进行，强制终止。内存泄露的积累将导致内存溢出。

导致内存泄漏的情况：

申请和释放不一致

由于 C++ 兼容 C，而 C 与 C++ 的内存申请和释放函数是不同的，因此在 C++ 程序中，就有两套动态内存管理函数。一条不变的规则就是采用 C 方式申请的内存就用 C 方式释放；用 C++ 方式申请的内存，用 C++ 方式释放。也就是用 malloc/alloc/realloc 方式申请的内存，用 free 释放；用 new 方式申请的内存用 delete 释放。在上述程序中，用 malloc 方式申请了内存却用 delete 来释放，虽然这在很多情况下不会有问题，但这绝对是潜在的问题。

申请和释放不匹配

申请了多少内存，在使用完成后就要释放多少。如果没有释放，或者少释放了就是内存泄露；多释放了也会产生问题。

释放后仍然读写

本质上说，系统会在堆上维护一个动态内存链表，如果被释放，就意味着该块内存可以继续被分配给其他部分，如果内存被释放后再访问，就可能覆盖其他部分的信息，这是一种严重的错误。

避免内存泄露的方法：养成良好的单元测试习惯，将内存泄露消灭在初始阶段。另外我们可以使用各种检测工具，对代码进行动态检测和静态检测，查出存在的泄露点或潜在泄露点。所谓**静态检测**，就是不运行程序，在程序的编译阶段进行检测，主要原理就是对 new 与 delete，malloc 与 free 进行匹配检测，基本上能检测出大部分coding 中因为粗心导致的问题。静态检测包括**手动检测**和**静态工具**分析。**动态检测**，就是运行程序的过程中，对程序的内存分配情况进行记录并判定。

60. 预编译的作用（C++）

C++ 编译器工作原理

简单地说，一个编译器就是一个程序，它可以阅读以某一种语言（源语言）编写的程序，并把该程序翻译成一个等价的、用另一种语言（目标语言）编写的程序。

C/C++编译系统将一个程序转化为可执行程序的过程包含：

1. 预处理(preprocessing)：根据已放置的文件中的预处理指令来修改源文件的内容。

编译(compilation)：通过词法分析和语法分析，在确认所有指令都是符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。

汇编(assembly)：把汇编语言代码翻译成目标机器指令的过程。

链接(linking)：找到所有用到的函数所在的目标文件，并把它们链接在一起合成为可执行文件(executable file)。

预处理（预编译）

预处理器是在程序源文件被编译之前根据预处理指令对程序源文件进行处理的程序。**预处理器指令以#号开头标识，末尾不包含分号**。预处理命令不是 C/C++语言本身的组成部分，不能直接对它们进行编译和链接。C/C++语言的一个重要功能是可以使用预处理指令和具有预处理的功能。预处理主要将源程序中的宏定义指令、条件编译指令、头文件包含指令以及特殊符号完成相应的替换工作。

头文件包含

预处理指令 #include 用于包含头文件，有两种形式：#include <xxx.h>，#include "xxx.h"。

尖括号形式表示被包含的文件在系统目录中。如果被包含的文件不一定在系统目录中，应该用双引号形式。在双引号形式中可以指出文件路径和文件名。如果在双引号中没有给出绝对路径，则默认为用户当前目录中的文件，此时系统首先在用户当前目录中寻找要包含的文件，若找不到再在系统目录中查找。

对于用户自己编写的头文件，宜用双引号形式。对于系统提供的头文件，既可以用尖括号形式，也可以用双引号形式，都能找到被包含的文件，但显然用尖括号形式更直截了当，效率更高。

宏替换

宏定义：一般用一个短的名字代表一个长的代码序列。宏定义包括无参数宏定义和带参数宏定义两类。宏名和宏参数所代表的代码序列可以是任何意义的内容，如类型、常量、变量、操作符、表达式、语句、函数、代码块等。

宏定义在源文件中必须单独另起一行，换行符是宏定义的结束标志，因此宏定义以换行结束，不需要分号等符号作分隔符。如果一个宏定义中代码序列太长，一行不够时，可采用续行的方法。续行是在键入回车符之前先键入符号\，注意回车要紧接在符号\之后，中间不能插入其它符号，当然代码序列最后一行结束时不能有\。

预处理器在处理宏定义时，会对宏进行展开（即宏替换）。宏替换首先将源文件中在宏定义随后所有出现的宏名均用其所代表的代码序列替换之，如果是带参数宏则接着将代码序列中的宏形参名替换为宏实参名。宏替换只作代码字符序列的替换工作，不作任何语法的检查，也不作任何的中间计算，一切其它操作都要在替换完后才能进行。如果宏定义不当，错误要到预处理之后的编译阶段才能发现。

条件编译

一般情况下，在进行编译时对源程序中的每一行都要编译，但是有时希望程序中某一部分内容只在满足一定条件时才进行编译，如果不满足这个条件，就不编译这部分内容，这就是条件编译。

条件编译主要是进行编译时进行有选择的挑选，注释掉一些指定的代码，以达到多个版本控制、防止对文件重复包含的功能。if, #ifndef, #ifdef, #else, #elif, #endif 是比较常见条件编译预处理指令，可根据表达式的值或某个特定宏是否被定义来确定编译条件。

此外，还有 #pragma 指令，它的作用是设定编译器的状态或指示编译器完成一些特定的动作。

2. 编译

编译过程的第一个步骤称为词法分析（lexical analysis）或扫描（scanning），词法分析器读入组成源程序的字符流，并且将它们组织成有意义的词素的序列，对于每个词素，词法分析器产生一个词法单元（token），传给下一个步骤：语法分析。

语法分析（syntax analysis）或解析（parsing）是编译的第二个步骤，使用词法单元来创建树形的中间表示，该中间表示给出了词法分析产生的词法单元流的语法结构。一

个常用的表示方法是语法树（syntax tree），树中每个内部结点表示一个运算，而该结点的子结点表示该运算的分量。

接下来是语义分析（semantic analyzer），使用语法树和符号表中的信息来检测源程序是否和语言定义的语义一致。

在源程序的语法分析和语义分析之后，生成一个明确的低级的或者类机器语言的中间表示。接下来一般会有一个机器无关的代码优化步骤，试图改进中间代码，以便生成更好的目标代码。

3. 汇编

对于被翻译系统处理的每一个 C/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机器语言代码。目标文件由段组成，通常一个目标文件中至少有两个段：代码段和数据段。

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。

4. 链接

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按操作系统装入执行的统一整体。主要有静态链接和动态链接两种方式：

静态链接：在链接阶段，会将汇编生成的目标文件.o 与引用到的库一起链接打包到可执行文件中，程序运行的时候不再需要静态库文件。

动态链接：把调用的函数所在文件模块（DLL）和调用函数在文件中的位置等信息链接进目标程序，程序运行的时候再从 DLL 中寻找相应函数代码，因此需要相应 DLL 文件的支持。

这里的库是写好的现有的，成熟的，可以复用的代码。现实中每个程序都要依赖很多基础的底层库，不可能每个人的代码都从零开始，因此库的存在意义非同寻常。本质上来说库是一种可执行代码的二进制形式，可以被操作系统载入内存执行。库有两种：静态库（.a、.lib）和动态库（.so、.dll），所谓静态、动态是指链接方式的不同。

静态链接库与动态链接库都是共享代码的方式。如果采用静态链接库，程序在运行时与函数库再无瓜葛，移植方便。但是会浪费空间和资源，因为所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。此外，静态库对程序的更新、部署和发布也会带来麻烦。如果静态库更新了，所有使用它的应用程序都需要重新编译、发布给用户。

动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入。不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题。动态库在程序运行是才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，增量更新。

此外，还要注意静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。

例子见：https://github.com/xuelangZF/CS_Offer/blob/master/C%2B%2B/Compiler.md

61. 解决哈希冲突的方法（数据结构）

当关键字值域远大于哈希表的长度，而且事先并不知道关键字的具体取值时。冲突就难免会发生。另外，当关键字的实际取值大于哈希表的长度时，而且表中已装满了记

录，如果插入一个新记录，不仅发生冲突，而且还会发生溢出。因此，处理冲突和溢出是哈希技术中的两个重要问题。

1、开放定址法

用开放定址法解决冲突的做法是：当冲突发生时，使用某种探测技术在散列表中形成一个探测序列。沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址(即该地址单元为空)为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。查找时探查到开放的地址则表明表中无待查的关键字，即查找失败。

开放寻址法；Hi=(H(key) + di) MOD m,i=1,2,…，k(k<=m-1)，其中 H(key)为散列函数，m 为散列表长，di 为增量序列

使用开放寻址法，非同义词也可能产生冲突。

注意：

①用开放定址法建立散列表时，建表前须将表中所有单元(更严格地说，是指单元中存储的关键字)置空。

②空单元的表示与具体的应用相关。

按照形成**探查序列**的方法不同，可将**开放定址法**区**分为线性探查法、线性补偿探测法、随机探测**等。

（1）**线性探查法**(Linear Probing)

di=1,2,3,…，m-1，称线性探测再散列；该方法的基本思想是：

将散列表 T[0..m-1]看成是一个循环向量，若初始探查的地址为 d(即 h(key)=d)，则最长的探查序列为：

d，d+l，d+2，…，m-1，0，1，…，d-1

即:探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+1]，…，直到 T[m-1]，此后又循环到 T[0]，T[1]，…，直到探查到 T[d-1]为止。

探查过程**终止于三种情况**：

(1)若当前探查的单元为空，则表示查找失败（若是插入则将 key 写入其中）；

(2)若当前探查的单元中含有 key，则查找成功，但对于插入意味着失败；

(3)若探查到 T[d-1]时仍未发现空单元也未找到 key，则无论是查找还是插入均意味着失败(此时表满)。

用线性探测法处理冲突，思路清晰，算法简单，但存在下列**缺点**：

① 处理溢出需另编程序。一般可另外设立一个溢出表，专门用来存放上述哈希表中放不下的记录。此**溢出表最简单的结构是顺序表，查找方法可用顺序查找**。

② 按上述算法建立起来的哈希表，删除工作非常困难。假如要从哈希表 HT 中删除一个记录，按理应将这个记录所在位置置为空，但我们不能这样做，而只能标上已被删除的标记，否则，将会影响以后的查找。

③ 线性探测法很容易产生堆聚现象。所谓堆聚现象，就是存入哈希表的记录在表中连成一片。按照线性探测法处理冲突，如果生成哈希地址的连续序列愈长 (即不同关键字值的哈希地址相邻在一起愈长) ，则当新的记录加入该表时，与这个序列发生冲突的可能性愈大。因此，哈希地址的较长连续序列比较短连续序列生长得快，这就意味着，一旦出现堆聚 (伴随着冲突) ，就将引起进一步的堆聚。

（2）**线性补偿探测法**

di=1^2,(-1)^2,2^2,(-2)^2,(3)^2,…，±(k)^2,(k<=m/2)称**二次探测再散列**；

线性补偿探测法的基本思想是：

将线性探测的步长从 1 改为 Q ，即将上述算法中的 j ＝ (j ＋ 1) % m 改为： j ＝ (j

＋ Q) % m ，而且要求 Q 与 m 是互质的，以便能探测到哈希表中的所有单元。

【例】 PDP-11 小型计算机中的汇编程序所用的符合表，就采用此方法来解决冲突，所用表长 m ＝ 1321 ，选用 Q ＝ 25 。

（3）**随机探测**

di=伪随机数序列，称伪随机探测再散列。每个 key 下次的步长是一样的，不同的 key 之间的步长是随机的。随机探测的基本思想是：

将线性探测的步长从常数改为随机数，即令： j ＝ (j ＋ RN) % m ，其中 RN 是一个随机数。在实际程序中应预先用随机数发生器产生一个随机序列，将此序列作为依次探测的步长。这样就能使不同的关键字具有不同的探测次序，从而可以**避免或减少堆聚**。基于与线性探测法相同的理由，在线性补偿探测法和随机探测法中，删除一个记录后也要打上删除标记。

2、拉链法

（1）拉链法解决冲突的方法

拉链法解决冲突的**做法**是：将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为 m，则可将散列表定义为一个由 m 个头指针组成的指针数组 T[0..m-1]。凡是散列地址为 i 的结点，均插入到以 T[i]为头指针的单链表中。T 中各分量的初值均应为空指针。在拉链法中，装填因子 α 可以大于 1，但一般均取 α≤1。

（2）**拉链法的优点**

与开放定址法相比，拉链法有如下几个优点：

①拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；

②由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况；

③开放定址法为减少冲突，要求装填因子 α 较小，故当结点规模较大时会浪费很多空间。而拉链法中可取 α≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间；

④在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。而对开放地址法构造的散列表，删除结点不能简单地将被删结点的空间置为空，否则将截断在它之后填人散列表的同义词结点的查找路径。这是因为各种开放地址法中，空地址单元(即开放地址)都是查找失败的条件。因此在用开放地址法处理冲突的散列表上执行删除操作，只能在被删结点上做删除标记，而不能真正删除结点。

（3）**拉链法的缺点**

拉链法的缺点是：指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间，而若将节省的指针空间用来扩大散列表的规模，可使装填因子变小，这又减少了开放定址法中的冲突，从而提高平均查找速度。

3、再散列法

Hi=RHi(key),i=1,2,…，k RHi 均是不同的散列函数，即在同义词产生地址冲突时计算另一个散列函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算时间。

4． 公共溢出区

这也是处理冲突的一种方法。假设哈希函数的值域为[0,m-1]，则设向量 HashTable[0 ·· m-1]为基本表，每个分量存放一个记录，另设立向量 OverTable[0 ·· v]为溢出表。所有关键字和基本表中关键字为同义词的记录，不管它们由哈希函数得到的哈希地址是什么，一旦发生冲突，都能填入溢出表。

62. Linux 递归删除最近 7 天的文件（Linux）

find . -type f -mtime +7 -exec rm -f {} \;

-mtime +7 最近七天

-exec rm -f {} \ find 指令提供的"-exec"选项可以调用外部指令完成对查找到的文件的操作。

63. 银行家算法避免死锁（操作系统）

系统安全状态的定义

1．安全状态

在避免死锁的方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，令进程等待。虽然并非所有的不安全状态都必然会转为死锁状态，但当系统进入不安全状态后，便有可能进而进入死锁状态；反之，只要系统处于安全状态，系统便可避免进入死锁状态。因此，**避免死锁的实质在于：系统在进行资源分配时，如何使系统不进入不安全状态。**

利用银行家算法避免死锁

1．银行家算法中的数据结构

(1) **可利用资源向量 Available**。这是一个含有 m 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 Available[j]=K，则表示系统中现有 Rj 类资源 K 个。

(2) **最大需求矩阵 Max**。这是一个 n×m 的矩阵，它定义了系统中 n 个进程中的每一个进程对 m 类资源的最大需求。如果 Max[i,j]=K，则表示进程 i 需要 Rj 类资源的最大数目为 K。

(3) **分配矩阵 Allocation。**这也是一个 n×m 的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 Allocation[i,j]=K，则表示进程 i 当前已分得 R j 类资源的数目为 K。

(4) **需求矩阵 Need。**这也是一个 n×m 的矩阵，用以表示每一个进程尚需的各类资源数。如果 Need[i,j]=K，则表示进程 i 还需要 R j 类资源 K 个，方能完成其任务。

上述三个矩阵间存在下述关系：

Need[i, j]=Max[i, j]-Allocation[i, j]

2．**银行家算法**

设 Request i 是进程 Pi 的请求向量，如果 Request i[j]=K，表示进程 P i 需要 K 个 R j 类型的资源。当 P i 发出资源请求后，系统按下述步骤进行检查：

(1) 如果 Request i[j]≤Need[i,j]，便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

(2) 如果 Request i[j]≤Available[j]，便转向步骤(3)；否则，表示尚无足够资源，Pi 须等待。

(3) 系统试探着把资源分配给进程 P i，并修改下面数据结构中的数值：

Available[j]:= Available[j]-Request i[j]；

Allocation[i,j]:= Allocation[i,j]+Request i[j]；

Need[i,j]:= Need[i,j]-Request i[j]；

(4) 系统执行安全性算法，检查此次资源分配后系统是否处于安全状态。**若安全，才正式将资源分配给进程 Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程 Pi 等待。**

3．**安全性算法**

系统所执行的安全性算法可描述如下：

(1) **设置两个向量：**

① 工作向量 Work，它表示系统可提供给进程继续运行所需的各类资源数目，它含有 m 个元素，在执行安全算法开始时，Work:=Available。

② Finish，它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做 Finish[i]:=false；当有足够资源分配给进程时，再令 Finish[i]:=true。

(2) **从进程集合中找到一个能满足下述条件的进程：**

① Finish[i]=false；

② Need[i,j]≤Work[j]；若找到，执行步骤(3)，否则，执行步骤(4)。

(3) **当进程 Pi 获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：**

Work[j]:= Work[j]+Allocation[i,j]；

Finish[i]:=true；

go to step （2）；

(4) **如果所有进程的 Finish[i]=true 都满足，则表示系统处于安全状态；否则，系统处于不安全状态。**

64. 中断与异常（操作系统）

在操作系统中引入核心态和用户态这两种工作状态后，就需要考虑这两种状态之间如何切换。**操作系统内核工作在核心态，而用户程序工作在用户态**。但系统不允许用户程序实现核心态的功能，而它们又必须使用这些功能。因此，需要在核心态建立一些“门”，实现从用户态进入核心态。在实际操作系统中，CPU 运行上层程序时唯一能进入这些“门”的途径就是通过中断或异常。当中断或异常发生时，运行用户态的 CPU 会立即进入核心态，这是通过硬件实现的（例如，用一个特殊寄存器的一位来表示 CPU 所处的工作状态，0 表示核心态，1 表示用户态。若要进入核心态，只需将该位置 0 即可)。中断是操作系统中非常重要的一个概念，对一个运行在计算机上的实用操作系统而言，缺少了中断机制，将是不可想象的。

中断(Interruption)，**也称外中断，指来自 CPU 执行指令以外的事件的发生**，如设备发出的 **I/O 结束中断**，表示设备输入/输出处理已经完成，希望处理机能够向设备发下一个输入 / 输出请求，同时让完成输入/输出后的程序继续运行。**时钟中断**，表示一个固定的时间片已到，让处理机处理计时、启动定时运行的任务等。这一类中断通常是与当前程序运行无关的事件，即它们与当前处理机运行的程序无关。

异常(Exception)，**也称内中断、例外或陷入**(Trap)，指**源自 CPU 执行指令内部的事件**，如程序的非法操作码、 地址越界、算术溢出、虚存系统的缺页以及专门的陷入指令等引起的事件。对异常的处理一般要依赖于当前程序的运行现场，而且异常不能被屏蔽，一旦出现应立即处理。关于内中断和外中断的联系与区别如图 1-2 所示。

65. 数据库事务的隔离级别（数据库）

数据库是要被广大客户所共享访问的，那么在数据库操作过程中很可能出现以下几种不确定情况。

更新丢失

两个事务都同时更新一行数据，一个事务对数据的更新把另一个事务对数据的更新覆盖了。这是因为系统没有执行任何的锁操作，因此并发事务并没有被隔离开来。

脏读

一个事务读取到了另一个事务未提交的数据操作结果。这是相当危险的，因为很可能所有的操作都被回滚。

不可重复读

不可重复读（Non-repeatable Reads）：一个事务对同一行数据重复读取两次，但是却得到了不同的结果。包括以下情况：

（1） **虚读**：事务 T1 读取某一数据后，事务 T2 对其做了修改，当事务 T1 再次读该数据时得到与前一次不同的值。

（2）**幻读**（Phantom Reads）：事务在操作过程中进行两次查询，第二次查询的结果包含了第一次查询中未出现的数据或者缺少了第一次查询中出现的数据（这里并不要求两次查询的 SQL 语句相同）。这是因为在两次查询过程中有另外一个事务插入数据造成的。

为了避免上面出现的几种情况，在标准 SQL 规范中，定义了 4 个事务隔离级别，不同的隔离级别对事务的处理不同。

未授权读取也称为读未提交（**Read Uncommitted**）：允许脏读取，但不允许更新丢失。如果一个事务已经开始写数据，则另外一个事务则不允许同时进行写操作，但允许其他事务读此行数据。该隔离级别可以通过“排他写锁”实现。

授权读取也称为读提交（**Read Committed**）：允许不可重复读取，但不允许脏读取。这可以通过“瞬间共享读锁”和“排他写锁”实现。读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行。

可重复读取（Repeatable Read）禁止不可重复读取和脏读取，但是有时可能出现幻读数据。这可以通过“共享读锁”和“排他写锁”实现。读取数据的事务将会禁止写事务（但允许读事务），写事务则禁止任何其他事务。

序列化（Serializable）：提供严格的事务隔离。它要求事务序列化执行，事务只能一个接着一个地执行，不能并发执行。仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机制保证新插入的数据不会被刚执行查询操作的事务访问到。

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为 Read Committed。它能够避免脏读取，而且具有较好的并发性能。尽管它会导致不可重复读、幻读和第二类丢失更新这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

隔离等级	脏读	不可重复读	幻读
读未提交	YES	YES	YES
读已提交	NO	YES	YES
可重复读	NO	NO	YES
串行化	NO	NO	NO

66. map 是怎么实现的？（C++）

hash_map 原理

hash_map 基于 hash table（哈希表）。哈希表最大的优点是：把数据存储和查询消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。然后在当前可利用内存越来越多的情况下，用空间换时间的做法是值得的。另外，编码比较容易也是它的特点之一。

其基本原理是：使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数，也叫散列函数），使得每个元素的关键字都与一个函数值（即数组下标，hash 值）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照关键字为每一个元素“分类”，然后将这个元素存储在相应“类”所对应的地方，称为桶。

但是，这不能够保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对不不同的元素，却计算出相同的函数值，这样就产生了“冲突”。换句话说，就是把不同的元素分在了相同的“类”中。总的来说，“直接定址”和“解决冲突”是哈希表的两大特点。

hash_map，首先分配一大片内存，形成许多桶。是利用 hash 函数，对 key 进行映射到不同区域进行保存。其**插入过程：**

1、得到 key；

2、通过 hash 函数得到 hash 值；

3、得到桶号（一般都为 hash 值对桶数求模）；

4、存放 key 和 value 在桶内；

其取值过程是：

1、得到 key；

2、通过 hash 函数得到 hash 值；

3、得到桶号；

4、比较桶的内部元素是否与 key 相等，若不相等，则没有找到；

5、取出相等的记录的 value；

hash_map 中直接地址用 hash 函数生成，解决冲突，用比较函数解决。这里可以看出，如果每个桶内部只有一个元素，那么查找的时候只有一次比较。当许多桶没有值时，许多查询就会更快了（指查不到的时候）。

hash_map 和 map 的区别在哪里？

（1）构造函数 hash_map 需要 hash 函数，等于函数；map 只需要比较函数（小于函数）。

（2）存储结构 hash_map 采用 hash 表存储，map 一般采用红黑树实现。因此内存数据结构是不一样的。

什么时候需要使用 hash_map，什么时候需要 map?

总体来说，hash_map 查找速度会比 map 快，而且查找速度基本和数据数据量大小，属于常数级别;而 map 的查找速度是 log(n)级别。并不一定常数就比 log(n)小， hash 还有 hash 函数的耗时，明白了吧，如果你考虑效率，特别是在元素达到一定数量级时，考虑考虑 hash_map。但若你对内存使用特别严格，希望 程序尽可能少消耗内存，那么一定要小心，hash_map 可能会让你陷入尴尬，特别是当你的 hash_map 对象特别多时，你就更无法控制了，而且 hash_map 的构造速度较慢。

现在知道如何选择了吗？权衡三个因素: 查找速度, 数据量, 内存使用。

67. http1.0 与 http2 的区别（计算机网络）

HTTP1.0

HTTP 协议老的标准是 HTTP/1.0，为了提高系统的效率，HTTP 1.0 规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个 TCP 连接，服务器完成请求处理后立即断开 TCP 连接，服务器不跟踪每个客户也不记录过去的请求。但是，这也造成了一些性能上的缺陷，例如，一个包含有许多图像的网页文件中并没有包含真正的图像数据内容，而只是指明了这些图像的 URL 地址，当 WEB 浏览器访问这个网页文件时，浏览器首先要发出针对该网页文件的请求，当浏览器解析 WEB 服务器返回的该网页文档中的 HTML 内容时，发现其中的图像标签后，浏览器将根据标签中的 src 属性所指定的 URL 地址再次向服务器发出下载图像数据的请求。显然，访问一个包含有许多图像的网页文件的整个过程包含了多次请求和响应，每次请求和响应都需要建立一个单独的连接，每次连接只是传输一个文档和图像，上一次和下一次请求完全分离。即使图像文件都很小，但是客户端和服务器端每次建立和关闭连接却是一个相对比较费时的过程，并且会严重影响客户机和服务器的性能。当一个网页文件中包含 JavaScript 文件，CSS 文件等内容时，也会出现类似上述的情况。

同时，带宽和延迟也是影响一个网络请求的重要因素。在网络基础建设已经使得带宽得到极大的提升的当下，大部分时候都是延迟在于响应速度。基于此会发现，http1.0 被抱怨最多的就是连接无法复用，和 head of line blocking 这两个问题。理解这两个问题有一个十分重要的前提：客户端是依据域名来向服务器建立连接，一般 PC 端浏览器会针对单个域名的 server 同时建立 6～8 个连接，手机端的连接数则一般控制在 4～6 个。显然连接数并不是越多越好，资源开销和整体延迟都会随之增大。连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。head of line blocking 会导致带宽无法被充分利用，以及后续健康请求被阻塞。

head of line blocking(holb)会导致健康的请求会被不健康的请求影响，而且这种体验的损耗受网络环境影响，出现随机且难以监控。为了解决 holb 带来的延迟，协议设计者设计了一种新的 pipelining 机制。pipelining 只能适用于 http1.1,而且由于使用苛刻，很多浏览器厂商并不支持。

HTTP1.1

为了克服 HTTP 1.0 的这个缺陷，HTTP 1.1 支持持久连接（HTTP/1.1 的默认模式使用带流水线的持久连接），在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟。一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。HTTP 1.1 还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间。

在 http1.1，request 和 reponse 头中都有可能出现一个 connection 的头，此 header 的含义是当 client 和 server 通信时对于长链接如何进行处理。

在 http1.1 中，client 和 server 都是默认对方支持长链接的， 如果 client 使用 http1.1 协议，但又不希望使用长链接，则需要在 header 中指明 connection 的值为 close；如果 server 方也不想支持长链接，则在 response 中也需要明确说明 connection 的值为 close。不论 request 还是 response 的 header 中包含了值为 close 的 connection，都表明当前正在使用的 tcp 链接在当天请求处理完毕后会被断掉。以后 client 再进行新的请求时就必须创建新的 tcp 链接了。

HTTP 1.1 在继承了 HTTP 1.0 优点的基础上，也克服了 HTTP 1.0 的性能问题。HTTP 1.1 通过增加更多的请求头和响应头来改进和扩充 HTTP 1.0 的功能。如，HTTP 1.0 不支持 Host 请求头字段，WEB 浏览器无法使用主机头名来明确表示要访问服务器上的哪个 WEB 站点，这样就无法使用 WEB 服务器在同一个 IP 地址和端口号上配置多个虚拟 WEB 站点。在 HTTP 1.1 中增加 Host 请求头字段后，WEB 浏览器可以使用主机头名来明确表示要访问服务器上的哪个 WEB 站点，这才实现了在一台 WEB 服务器上可以在同一个 IP 地址和端口号上使用不同的主机名来创建多个虚拟 WEB 站点。HTTP 1.1 的持续连接，也需要增加新的请求头来帮助实现，例如，Connection 请求头的值为 Keep-Alive 时，客户端通知服务器返回本次请求结果后保持连接；Connection 请求头的值为 close 时，客户端通知服务器返回本次请求结果后关闭连接。HTTP 1.1 还提供了与身份认证、状态管理和 Cache 缓存等机制相关的请求头和响应头。HTTP/1.0 不支持文件断点续传，RANGE:bytes 是 HTTP/1.1 新增内容，HTTP/1.0 每次传送文件都是从文件头开始，即 0 字节处开始。RANGE:bytes=XXXX 表示要求服务器从文件 XXXX 字节处开始传送，这就是我们平时所说的断点续传！

由上，**HTTP/1.1 相较于 HTTP/1.0 协议的区别**主要体现在：

1 缓存处理

2 带宽优化及网络连接的使用

3 错误通知的管理

4 消息在网络中的发送

5 互联网地址的维护

6 安全性及完整性

常用的请求方式

GET 请求获取 Request-URI 所标识的资源

POST 在 Request-URI 所标识的资源后附加新的数据

HEAD 请求获取由 Request-URI 所标识的资源的响应消息报头

PUT 请求服务器存储一个资源，并用 Request-URI 作为其标识

DELETE 请求服务器删除 Request-URI 所标识的资源

TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断

CONNECT 保留将来使用

OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求

GET 方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用 GET 方法向服务器获取资源，POST 方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。**GET 是用于获取数据的，POST 一般用于将数据发给服务器之用。**

相比 **HTTP/1.x，HTTP/2 在底层传输做了很大的改动和优化：**

（1） **HTTP/2 采用二进制格式传输数据**，而非 HTTP/1.x 的文本格式。二进制格式在协议的解析和优化扩展上带来更多的优势和可能。

（2） **HTTP/2 对消息头采用 HPACK 进行压缩传输**，能够节省消息头占用的网络的流量。而 HTTP/1.x 每次请求，都会携带大量冗余头信息，浪费了很多带宽资源。头压缩能够很好的解决该问题。

（3） **多路复用**，直白的说就是所有的请求都是通过一个 TCP 连接并发完成。HTTP/1.x 虽然通过 pipeline 也能并发请求，但是多个请求之间的响应会被阻塞的，所以 pipeline 至今也没有被普及应用，而 HTTP/2 做到了真正的并发请求。同时，流还支持优先级和流量控制。

（4） **Server Push**：服务端能够更快的把资源推送给客户端。例如服务端可以主动把 JS 和 CSS 文件推送给客户端，而不需要客户端解析 HTML 再发送这些请求。当客户端需要的时候，它已经在客户端了。

HTTP/2 主要是 HTTP/1.x 在底层传输机制上的完全重构，HTTP/2 是基本兼容 HTTP/1.x 的语义的(详细兼容性说明请戳这里)。Content-Type 仍然是 Content-Type，只不过它不再是文本传输了。

----------**分割线**----------

以下主要是一些面经里提到过的设计测试用例的题，我这是总结整理从网上找答案，所以也要谢谢原作者。

1. 给你一个自动贩售机，你进行测试，编写测试计划和测试用例

（1）首先可能先要测试一下，外观设计的是否合理，是否符合大众审美观点。（可用性）

（2）测试操作是否简单便捷，还有操作说明是否简单易懂无歧义。（可用性）

（3）对于正确流程的测试，有零钱找：1.投入足够纸币，选择商品，看能否正确投出所选货物以及正确找零。2.投入足够硬币，重复以上过程。3.纸币和硬币配合来购买物品的情况。（功能）

（4）错误流程（不能正确购买物品）1.没有足够的零钱找，投入超出物品价格的纸币或硬币，看能否正确的提示以及吐出钱。2.投入的钱不够买所选的物品，能否正确提示，以及吐出钱 3.投入的不是合适的纸币或者硬币，如缺角的纸币，游戏币等。（功能）

（5）异常处理，如断电，系统出错等，能否恢复，并且正确处理未完成的业务。（可靠性）

（6）安全性，货物的存放和钱币的存放是否安全。（安全性）

从测试的六种类型着手，分别测试。对于自动售货机的测试 ，最好还是根据需求说明书来测，根据需求说明，测试用户的需求是否得到了满足，同时测试系统有没有做出没让他做的事。

2. 给你一个 ATM 取款机，你怎么测试

设想自己使用 ATM 取款机的整个流程，对整个流程中的每个步骤进行测试，给出输入及预期输出结果，测试结果是否与预期一致。

本用例的开端是 ATM 处于准备就绪状态。

准备提款 - 客户将银行卡插入 ATM 机的读卡机。

验证银行卡 - ATM 机从银行卡的磁条中读取帐户代码，并检查它是否属于可以接收的银行卡。

输入 PIN - ATM 要求客户输入 PIN 码（4 位）

验证帐户代码和 PIN - 验证帐户代码和 PIN 以确定该帐户是否有效以及所输入的 PIN 对该帐户来说是否正确。对于此事件流，帐户是有效的而且 PIN 对此帐户来说正确无误。

ATM 选项 - ATM 显示在本机上可用的各种选项。在此事件流中，银行客户通常选择“提款”。

输入金额 - 要从 ATM 中提取的金额。对于此事件流，客户需选择预设的金额（10 美元、20 美元、50 美元或 100 美元）。

授权 - ATM 通过将卡 ID、PIN、金额以及帐户信息作为一笔交易发送给银行系统来启动验证过程。对于此事件流，银行系统处于联机状态，而且对授权请求给予答复，批准完成提款过程，并且据此更新帐户余额。

出钞 - 提供现金。

返回银行卡 - 银行卡被返还。

收据 - 打印收据并提供给客户。ATM 还相应地更新内部记录。

用例结束时 ATM 又回到准备就绪状态。

备选流 1 - 银行卡无效 在基本流步骤 2 中 - 验证银行卡，如果卡是无效的，则卡被退回，同时会通知相关消息。

备选流 2 - ATM 内没有现金 在基本流步骤 5 中 - ATM 选项，如果 ATM 内没有现金，则“提款”选项将无法使用。

备选流 3 - ATM 内现金不足 在基本流步骤 6 中- 输入金额，如果 ATM 机内金额少于请求提取的金额，则将显示一则适当的消息，并且在步骤 6 - 输入金额处重新加入基本流。

备选流 4 - PIN 有误 在基本流步骤 4 中- 验证帐户和 PIN，客户有三次机会输入 PIN。

如果 PIN 输入有误，ATM 将显示适当的消息；如果还存在输入机会，则此事件流在步骤 3 - 输入 PIN 处重新加入基本流。

如果最后一次尝试输入的 PIN 码仍然错误，则该卡将被 ATM 机保留，同时 ATM 返回到准备就绪状态，本用例终止。

备选流 5 - 帐户不存在 在基本流步骤 4 中 - 验证帐户和 PIN，如果银行系统返回的代码表明找不到该帐户或禁止从该帐户中提款，则 ATM 显示适当的消息并且在步骤 9 - 返回银行卡处重新加入基本流。

备选流 6 - 帐面金额不足 在基本流步骤 7 - 授权中，银行系统返回代码表明帐户余额少于在基本流步骤 6 - 输入金额内输入的金额，则 ATM 显示适当的消息并且在步骤 6 - 输入金额处重新加入基本流。

备选流 7 - 达到每日最大的提款金额 在基本流步骤 7 - 授权中，银行系统返回的代码表明包括本提款请求在内，客户已经或将超过在 24 小时内允许提取的最多金额，则 ATM 显示适当的消息并在步骤 6 - 输入金额上重新加入基本流。

备选流 x - 记录错误 如果在基本流步骤 10 - 收据中，记录无法更新，则 ATM 进入“安全模式”，在此模式下所有功能都将暂停使用。同时向银行系统发送一条适当的警报信息表明 ATM 已经暂停工作。

备选流 y - 退出 客户可随时决定终止交易（退出）。交易终止，银行卡随之退出。

备选流 z - “翘起” ATM 包含大量的传感器，用以监控各种功能，如电源检测器、不同的门和出入口处的测压器以及动作检测器等。在任一时刻，如果某个传感器被激活，则警报信号将发送给警方而且 ATM 进入“安全模式”，在此模式下所有功能都暂停使用，直到采取适当的重启/重新初始化的措施。

在第一次迭代中，根据迭代计划，我们需要核实提款用例已经正确地实施。此时尚未实施整个用例，只实施了下面的事件流：

基本流 - 提取预设金额（10 美元、20 美元、50 美元、100 美元）

备选流 2 - ATM 内没有现金

备选流 3 - ATM 内现金不足

备选流 4 - PIN 有误

备选流 5 - 帐户不存在/帐户类型有误

备选流 6 - 帐面金额不足

可以从这个用例生成下列场景

场景 1 - 成功的提款 基本流
场景 2 - ATM 内没有现金 基本流 备选流 2
场景 3 - ATM 内现金不足 基本流 备选流 3
场景 4 - PIN 有误（还有输入机会） 基本流 备选流 4
场景 5 - PIN 有误（不再有输入机会） 基本流 备选流 4
场景 6 - 帐户不存在/帐户类型有误 基本流 备选流 5
场景 7 - 帐户余额不足 基本流 备选流 6

注：为方便起见，备选流 3 和 6（场景 3 和 7）内的循环以及循环组合未纳入上表。

对于这 7 个场景中的每一个场景都需要确定测试用例。可以采用矩阵或决策表来确定和管理测试用例。下面显示了一种通用格式，其中各行代表各个测试用例，而各列则代表 测试用例的信息。本示例中，对于每个测试用例，存在一个测试用例 ID、条件（或说明）、测试用例中涉及的所有数据元素（作为输入或已经存在于数据库中）以及预期结果。

通过从确定执行用例场景所需的数据元素入手构建矩阵。然后，对于每个场景，至少要确定包含执行场景所需的适当条件的测试用例。例如，在下面的矩阵中，V（有效）用于表明这个条件必须是 VALID（有效的）才可执行基本流，而 I（无效）用于表明这种条件下将激活所需备选流。下表中使用的“n/a”（不适用）表明这个条件不适用于测试用例。

测试用例
ID 号 场景/条件 PIN
帐号 输入的金额
（或选择的金额） 帐面金额 ATM 内的金额 预期结果
CW1. 场景 1 - 成功的提款 V V V V V 成功的提款。
CW2. 场景 2 - ATM 内没有现金 V V V V I 提款选项不可用，用例结束
CW3. 场景 3 - ATM 内现金不足 V V V V I 警告消息，返回基本流步骤 6 - 输入金额
CW4. 场景 4 - PIN 有误（还有不止一次输入机会） I V n/a V V 警告消息，返回基本流步骤 4，输入 PIN
CW5. 场景 4 - PIN 有误（还有一次输入机会） I V n/a V V 警告消息，返回基本流步骤 4，输入 PIN
CW6. 场景 4 - PIN 有误（不再有输入机会） I V n/a V V 警告消息，卡予保留，用例结束

在上面的矩阵中，六个测试用例执行了四个场景。对于基本流，上述测试用例 CW1 称为正面测试用例。它一直沿着用例的基本流路径执行，未发生任何偏差。基本流的全面测试必须包括负面测试用例，以确保只有在符合条件的情况下才执行基本流。这些负面测试用例由 CW2 至 6 表示（阴影单元格表明这种条件下需要执行备选流）。虽然 CW2 至 6 对于基本流而言都是负面测试用例，但它们相对于备选流 2 至 4 而言是正面测试用例。而且对于这些备选流中的每一个而言，至少存在一个负面测试用例（CW1 - 基本流）。

每个场景只具有一个正面测试用例和负面测试用例是不充分的，场景 4 正是这样的一个示例。要全面地测试场景 4 - PIN 有误，至少需要三个正面测试用例（以激活场景 4）：

* 输入了错误的 PIN，但仍存在输入机会，此备选流重新加入基本流中的步骤 3 - 输入 PIN。

* 输入了错误的 PIN，而且不再有输入机会，则此备选流将保留银行卡并终止用例。

* 最后一次输入时输入了“正确”的 PIN。备选流在步骤 5 - 输入金额处重新加入基本流。

注：在上面的矩阵中，无需为条件（数据）输入任何实际的值。以这种方式创建测试用例矩阵的一个优点在于容易看到测试的是什么条件。由于只需要查看 V 和 I（或此处采用

的阴影单元格），这种方式还易于判断是否已经确定了充足的测试用例。从上表中可发现存在几个条件不具备阴影单元格，这表明测试用例还不完全，如场景 6 - 不存在的帐户/帐户类型有误和场景 7 - 帐户余额不足就缺少测试用例。

一旦确定了所有的测试用例，则应对这些用例进行复审和验证以确保其准确且适度，并取消多余或等效的测试用例。

测试用例一经认可，就可以确定实际数据值（在测试用例实施矩阵中）并且设定测试数据：

测试用例
ID 号 场景/条件 PIN 帐号 输入的金额
（或选择的金额） 帐面金额 ATM 内的金额 预期结果

CW1. 场景 1 - 成功的提款 4987 809 - 498 50.00 500.00 2,000 成功的提款。帐户余额被更新为 450.00

CW2. 场景 2 - ATM 内没有现金 4987 809 - 498 100.00 500.00 0.00 提款选项不可用，用例结束

CW3. 场景 3 - ATM 内现金不足 4987 809 - 498 100.00 500.00 70.00 警告消息，返回基本流步骤 6 - 输入金额

CW4. 场景 4 - PIN 有误（还有不止一次输入机会） 4978 809 - 498 n/a 500.00 2,000 警告消息，返回基本流步骤 4，输入 PIN

CW5. 场景 4 - PIN 有误（还有一次输入机会） 4978 809 - 498 n/a 500.00 2,000 警告消息，返回基本流步骤 4，输入 PIN

CW6. 场景 4 - PIN 有误（不再有输入机会） 4978 809 - 498 n/a 500.00 2,000 警告消息，卡予保留，用例结束

以上测试用例只是在本次迭代中需要用来验证提款用例的一部分测试用例。需要的其他测试用例包括：

* 场景 6 - 帐户不存在/帐户类型有误：未找到帐户或帐户不可用
* 场景 6 - 帐户不存在/帐户类型有误：禁止从该帐户中提款
* 场景 7 - 帐户余额不足：请求的金额超出帐面金额

在将来的迭代中，当实施其他事件流时，在下列情况下将需要测试用例：

* 无效卡（所持卡为挂失卡、被盗卡、非承兑银行发卡、磁条损坏等）
* 无法读卡（读卡机堵塞、脱机或出现故障）
* 帐户已消户、冻结或由于其他方面原因而无法使用
* ATM 内的现金不足或不能提供所请求的金额（与 CW3 不同，在 CW3 中只是一种币值不足，而不是所有币值都不足）
* 无法联系银行系统以获得认可
* 银行网络离线或交易过程中断电

在确定功能性测试用例时，确保满足下列条件：

* 已经为每个用例场景确定了充足的正面和负面测试用例。
* 测试用例可以处理用例所实施的所有业务规则，确保对于业务规则，无论是在内部、外部还是在边界条件/值上都存在测试用例。
* 测试用例可以处理所有事件或动作排序（如在设计模型的序列图中确定的内容），还应能处理用户界面对象状态或条件。
* 测试用例可以处理为用例所指定的任何特殊需求，如最佳/最差性能，有时这些特殊需求会与用例执行过程中的最小/最大负载或数据容量组合在一起。

3. 给你一个水杯、凳子，你怎么测试

测试项目：**杯子**
需求测试:查看杯子使用说明书
界面测试:查看杯子外观
功能度：用水杯装水看漏不漏；水能不能被喝到
安全性：杯子有没有毒或细菌
可*性：杯子从不同高度落下的损坏程度
可移植性：杯子再不同的地方、温度等环境下是否都可以正常使用
兼容性：杯子是否能够容纳果汁、白水、酒精、汽油等
易用性：杯子是否烫手、是否有防滑措施、是否方便饮用
用户文档：使用手册是否对杯子的用法、限制、使用条件等有详细描述
疲劳测试：将杯子盛上水（案例一）放 24 小时检查泄漏时间和情况；盛上汽油（案例二）放 24 小时检查泄漏时间和情况等
压力测试：用根针并在针上面不断加重量，看压强多大时会穿透
跌落测试: 杯子加包装(有填充物),在多高的情况摔下不破损
震动测试: 杯子加包装(有填充物),六面震动,检查产品是否能应对恶劣的铁路/公路/航空运输
测试数据：测试数据具体编写此处略（最讨厌写测试数据了）。其中应用到：场景法、等价类划分法、因果图法、错误推测法、边界值法等方法

4. 测试项目：椅子

1、首先要了解对**椅子**的需求（显示、隐式需求），不明确时要与需求方进行确认，需求可能包含椅子的材质、适用场合、是否可折叠、可调高度等。【一个椅子最基本的需求是可以让人坐。那么第一条冒烟测试的用例就是是否可以坐人。】【然后研究这把椅子的适用场合。根据它要摆放的场合来看他的隐性需求，比如可坐人数，材质，样式，甚至价格。】
2、分析需求中涵盖的所有的功能点。
3、针对不同的功能点，选取适当的测试方法，编写测试用例。
4、最后应该就是准备好测试环境、被测椅子、测试工具、对照着测试用例执行功能测试。
注：
1、如果面试官提出没有需求的话，那就按照椅子通常具有的功能测吧（好比行业的标准）。
2、还要考虑测试的时间和成本的要求，保证优先级别高的先测完。

功能测试：

1.能不能供人坐，即能不能供人使用。
2.坐上去是否摇晃。
3.坐人后是否会发出响声。
4.椅子上会不会掉颜色，即坐上去，来回摩擦椅子上的颜色会不会粘到衣服上
5.有水撒到椅子上的时候，用布子或纸擦的时候会不会掉颜色。能不能擦干净水。
6.坐上去会不会有塌陷的感觉。
7.从椅子上离开的时候会不会发出响声。
8.椅子会不会轻易挂到衣服。
9.靠在椅背上的时候会不会，发出响声，椅子会不会摇晃。
10.椅子脏了是能易清理干净。

性能测试：

1.椅子能承受多大的重量，不会发出响声；能承受多大的重量不被压坏。
2.椅子是否怕水
3.椅子是否怕火
4.椅子是否能在压了重物的情况下,然后摇晃,能坚持不长时间不响\不坏.

5.椅背,用力向后靠椅背,检测椅背的向后的承受能力.

安全性测试：

1.椅子的材质是否与用户说明书或质量保证书上的一样。

2.椅子的材料是否对人体有危害。

3.在撒到椅子上水/饮料等液体的时候，椅子会不会产生什么有害的物质。

4.在椅子被磨损的时候，会不会有划伤或擦伤用户的可能。

5.坐在椅子上的时候，是否安全，例如在只坐到椅子最前端的一部分时，椅子会不会失去平衡等等。

6.在与椅子摩擦的时候，会产生一定的容量，在摩擦的比较厉害的时候，会不会，产生有害的气体或物质。例如，产生难闻的气味等等。

7.在人坐或踩在椅子上时椅子是否稳固,即不摇晃等。

外观/适用性测试（界面/适用性测试）：

1.椅子的外观是否美观实用。

2.是否与用户说明书或质量保证书上的一样出现的实物图相同。

3.椅子的气味/扶手/坐垫及靠垫的软硬度是否合适。

4.椅子是否容易挪动。

5.椅子的高度/重量/材质是否合适。

6.椅子的适用场合是否合适。

5.如何测试一个三角形是否是一个等腰三角形，写出测试用例

首先它是一个三角形（a,b,c），考虑所有情况：

编号	有效等价类	编号	无效等价类
1	a = b != c(有一组边相等)	3	a!=b!=c(三条边都不相等)
2	a=b=c(三条边相等)		

共三组测试用例

6. 怎样测一张 A4 纸

如测试一张 A4 纸的，可以从物理、化学、经济、使用性能等方面考察，具体如 A4 纸的材料、用途、价格、规格（尺寸、克数）、白度、韧性、吸水性、油墨扩散性、平滑度、腐化周期、掺石粉比例、纤维长度，可折曲次数、抗拉力度、抗皱效果。。。

7. 对百度页面进行测试

先来一个一般上测试人员最喜欢最常用的测试方法，边界值法。

文本框边界值，一般可以测试一下输入字符的数量。

探索过程：

1.不输入文字，直接按搜索----->页面刷新，无变化---->结论 1

2.复制粘贴一段很长的中文进入文本框----->被百度自动截取其中前 100 个字-->结论 2

3.按下搜索按钮，百度跳转到搜索结果页面，并提示“"××" 及其后面的字词均被忽略，因为百度的查询限制在 38 个汉字以内”，将被自动截取的内容复制粘贴到 word，统计字数为 38--->结论 3

5.复制粘贴一段很长的中文、英文、空格、符号混合文本进入文本框----->被百度自动截取其中一段内容。---->结果截取了 100 个字--->结论 4

6.复制粘贴 38 个汉字进入搜索文本框，并中间加入 62 个连续空格后按下搜索-----> 搜索结果里最后一个汉字被忽略，因为前面有 37 个汉字加 1 个合并后的空格长度已达 38. 并且在文本框里原来 62 个空格的位置现在显示一个空格--->结论 5

关于文本框字符数的结论：

1.最小输入值为 0 个字；

2.百度搜索文本框内可输入的最大字数是 100 个汉字；

3.百度搜索文本框的输入值在点击搜索按钮后，会被截取前 38 个汉字，其后面的字词均被忽略；

4.任意一个中文、英文、符号、空格在输入进文本框内第一次计数时均视同一个汉字；

5.连续的空格在点击搜索按钮后进行搜索时会自动合并，并转化一个单独的空格。

根据上述探索结果设计的文本框**边界值测试用例（思路）**：

1.输入 0 个汉字：

什么也不输入直接点击搜索按钮，点击后应刷新首页

2.输入 38 个汉字：

输入 38 个汉字后点击搜索按钮，成功跳转到搜索结果页面

3.输入 39 个汉字：

输入 39 个汉字后点击搜索按钮，跳转到搜索结果页面，并在结果页面上显示“"×"及其后面的字词均被忽略，因为百度的查询限制在 38 个汉字以内”

4.输入 100 个汉字：

输入 100 个汉字后点击搜索按钮，跳转到搜索结果页面，并在结果页面上显示“"×"及其后面的字词均被忽略，因为百度的查询限制在 38 个汉字以内”

5.尝试输入 100 个以上的汉字：

尝试输入 101 个汉字，预期结果为尝试失败，只能输入 100 个汉字。

6.英文、符号的测试：

以英文、符号作为输入值，重复用例 1 到 5，预期结果应与汉字相同。

7.空格的测试：

复制粘贴 38 个汉字进入搜索文本框，并中间加入 62 个连续空格后按下搜索，预期搜索结果里最后一个汉字被忽略，并在结果页面上显示“"×" 及其后面的字词均被忽略，因为百度的查询限制在 38 个汉字以内”同时，连续的空格应在搜索后的文本框内显示为一个空格。

然后是另一个测试人员最爱的测试，等价类法。

等价类可以这么划：

按区间划分。

按数值划分。

按数值集合划分。

按限制条件或规划划分。

按处理方式划分。

三角形问题显然按照数值和区间划分了。但百度首页的话就难划了。

按区间划 1.有意义的关键词做输入值，预期能搜出结果

2.无意义的关键词做输入值（比如用脸滚键盘来输入一些乱七八糟的关键字），预期搜不出任何结果

第三种方法：写 case 就是**按照需求和标准来写**嘛

对于**搜索引擎的测试需求和评价指标**随便百度了一下就找到了：

1）搜索覆盖的网站或网页数目及范围；

2）结果的准确性，或者说相关度；

3）结果的全面性；

4）结果的时效性，比如说期望搜到最新的结果；

5）搜索的速度或者响应时间

6）易用性

7）链接有效性、稳定性等

对于这些我们可以一个个设计用例来测

比如

1.找一个很偏僻的小网站看看能不能被搜到。

2.挑一系列常用关键字，然后人工检查搜索结果的相关度。

3.挑一系列常用关键字，然后人工检查搜索结果的排序等等

其中的问题是，

1.这个偏僻的小网站应不应该被搜到呢。

2.我怎么知道这个搜索结果的相关度哪个应该算高，哪个应该算低。

3.我怎么知道这个搜索结果的哪个应该排第一哪个应该排第二。等等

这里应该超出黑盒测试的范围了。

假如能用白盒测试/自动化测试。常用方法有：

3.设计一些有一定通用性的规则，然后校验。（推荐）

比如说，给你一个数据库，告诉你里面所有记录都是数字，那么写脚本检查数据库的记录，当发现有字母时，脚本可以报异常。

同样，在百度里搜索一个关键字，然后根据预先定义好的某个规则，比如搜索结果页面在相关度一样的时候应当以时间为排序标准，检查出结果里有明显违背时，脚本可以报异常。

5.还有特别提一下易用性。

百度里面有：

1.下拉框提示

2.搜索结果页提示”要找的是不是 xxxx“

3.搜索结果页提示”关键字里去掉引号可以找到更多 xxx“

4.搜索结果页提示”您输入的网址是不是 xxx“

等等，都可以用探索性测试的方法试出来。然后针对他们设计对应的 case。

百度首页还有用户登录、导航、天气预报、随心听、自定义主页、各种链接等等。就光从功能测试角度来看也有很多东西需要测。

8. 美团有一个 API 用于创建团购订单，地址如下

https://open.meituan.com/order/createorder?token=1234567890abcdefghijklmnopqrstuvwxyz

其中，token 用于验证用户身份

请求方法：POST

参数类型：application/json

参数列表（隐去无关参数）：

```
{
"dealid": 90,
"quantity": 5
}
```

传入 deal ID（要购买的团购券的 ID）和数量后，返回新生成的订单 ID（隐去无关参数）。

例如：

{

"success": 0, // 正常情况为 0
"msg": "", // 正常情况为空
"orderid": 2910100100, // 订单 id
}
设计测试用例进行测试，尽可能覆盖的完备。

考察测试用例设计思路，从功能、性能、安全等多方面思考；结合测试用例设计方法回答。 答案要点： 功能测试 1. 正向功能； 2. 参数为空； 3. dealid 不存在； 4. dealid 为非数字的值；5. quantity 为 0 或负值；6. quantity 大于库存量；7. token 无效 8. 入参不是 JSON 性能测试 1. 压力测试，考察系统在极限压力下的处理能力 2. 狭义性能测试，验证系统能够达到一定的处理能力 3. 并发测试，测试数据库和应用服务器对并发请求的处理 安全性测试 1. 伪造 token 攻击 2. 订单潮水攻击 3. deal 遍历攻击 4. SQL 注入攻击 加分项 订单复用：当同一个用户提交的 dealid、quantity 相同时，返回的 orderID 总是一样（没有重复创建订单）

你的缺点是什么。

1 工作的时候，顾虑过多，可能会因此错过一些机会，所以我特别看重团队合作的重要性，善用集体的智慧，排查风险，抓住机遇。

2 不会拒绝别人，有时候会因为帮助别人完成任务而耽误自己的进度，我已经在慢慢学着拒绝，会讲清楚我的状况让对方理解；

为什么要做测试?

总结：1，从以前参加过的一些项目来看，对于软件测试有浓厚的兴趣，享受发现问题验证问题的乐趣。

2，软件测试是为软件产品的质量把关的，目前软件测试的工业化时代还没有来临，自动化软件测试工具还没有能统一起来的模式，大部分还是靠人工测试，所以软件测试有很大的发展空间和前景。

3，软件测试贯穿于整个软件开发的生命周期，交流面广，方便以后做更明确的职业规划啦……

4，我的性格细致耐心，逻辑思维较强，感觉我的性格更适合做测试。

测试开发面试经验

第一次面试，滴滴一面（26 分钟）

1. HTTP 协议
2. 熟悉的协议，TCP 协议，TCP 连接与断开
3. 长连接
4. 多线程，并发
5. 数据库查询语句，滴滴的订单创建一个表会选择哪些键，如何查询按日期排列的前 10 个订单 SQL 语句
6. 项目（先说的项目，自我介绍引出来的）
7. Linux，Apache
8. 自己的项目中（记事本），如何做的测试

第二次面试，阿里一面（40+分钟）

1.面试官自我介绍完就让讲讲研究生的研究方向和项目（项目介绍的太详细，所以时间有点

长，以后要精简），项目中遇到的问题，如何解决的，项目的应用
2.计算机网络：TCP 连接与断开
3.C++:多态是如何实现的
4.操作系统：多线程与多进程（资源、空间）
5.数据库：死锁以及死锁的解决办法
6.软件测试：测试用例的设计
7.自己的项目中（富文本编辑器），如何做的测试
8.学得好的课程和学得不好的课程（说了编程不好，想了觉得不太好）
9.一道数据结构题：1-100000 这些数字乱序，从中抽出两个，怎么找到抽出的是哪两个？我想了想，说了一种方法是：先将这些数排序，再两两相减，相隔两数差不是 1 的就是差了中间这个数，但是显然要排序的话，会增加时间复杂度，面试官又让我想另外一种思路，不通过排序的，我说想不起来，他说没事不着急慢慢想，我还是没想出来，他就提示我用索引的方法，用一个等长数组存 1-100000，遍历乱序数组将遍历到的每个数字在索引数组中置为 0，然后不为 0 的则是要找的数字；之后他说再改变一下如果是抽出一个数字呢？我立马就想到这些数字的和是固定的并且可以通过公式计算，所以就想到算出和之后再将乱序数组中每个数字减掉，剩下的则是要找的数，我没想到更合适的方法，面试官也觉得还可以，因为求和比较容易，有公式
阿里二面（20 多分钟）
1.项目
2.TCP 与 UDP 的区别，传输层有几个协议
3.TCP/IP 有几层，分别是什么
4.研究生期间学的课程，人工智能学的啥
5.C++的存储方式
6.C++常引用什么时候用
7.求一个二叉平衡树
8.堆栈排序
9.说一下自己常用的排序，说明冒泡排序
10.软件测试常用的方法
11.会什么脚本语言
12.最近关注的技术，从哪里获取
13.自动化测试工具及其功能
14.针对百度搜索页面设计测试用例
京东一面：
1 自我介绍
2 先问你是西安的，能接受去北京吗
3 你是做 c++的 以后做的工作可能用 Java 开发（愿意学）
照着简历
4 说一说起泡排序的实现
数据库的左连接右连接
5 计算机网络中 OSI 七层模型分别是什么
6 http 协议中 get 与 post 的区别
7 Linux 了解多少 （知道一些常用命令）说一些 如何查看文件权限（ls -l） 如何修改文件权限 权限 drwx d 代表什么 分别对应数字几
8 软件测试了解多少 软件开发的 V 模型

9 自动化测试工具知道多少
10 自己项目中做的测试
11 科研项目中负责的东西
12 探讨自动化测试 及 测试工程师职业发展
京东二面：
1 自我介绍
2 你是西安的 愿意去北京吗 家人同意吗 西安不算发达城市 交通怎么还拥堵 人行道规划也不合理 来西安的火车不停穿隧道
3 当团支书都组织什么活动了 喜欢户外运动都有什么 我说上山 问我上过华山吗 我说上过太白山
4 你每年都拿奖学金 奖学金很好拿吗
5 你怎么理解测试开发的 有偏差 解释一番
6 面试官介绍业务线 部门活动 对我们的工作有兴趣吗
7 对自己职业生涯的规划 三年内的 还有技术规划 才知道做测试也有架构师
8 提问加班的问题 解释加班的两种原因 个人问题 耽误了工作 公司大型活动需要加班 只是值班
9 如果又有一个 bat 的 offer 怎么选 我说没投
10 探讨京东自营的问题
面试官问了好几次我有什么问题 我都不知道有什么问题了
整个面试几乎都是面试官在介绍我在听 中间插入几句看法 疑惑呀 技术问题基础知识几乎完全没有问到
最后再确认我没有问题之后就结束了，实习就只拿到了京东一个 offer，结束了实习招聘
美团一面：
为什么要做测试？
给你一个手机 app 你如何测试，写测试计划？
一个数组如何逆序？
Linux 进程与线程有什么区别，进程线程的通信方式？
TCP/IP 五层结构和对应协议？
Session 和 cookie？
http 状态码？
数据库 update 命令，写一个复杂的 SQL 命令
Linux 复杂的命令 管道
Tcp 与 udp 区别。
会 shell 吗？
冒泡排序改进

美团二面
1，自我介绍
2，测试：
1）怎么学习测试的？设计测试用例的方法？给你一个 http 的 API，get 方法有两个参数 id 与时间 ，如何设计测试用例？
3，数据结构
知道什么树，二叉树都有哪些，平衡二叉树有哪些特性，与满二叉树有什么区别
链表有什么特性，什么时候用链表

循环队列和顺序队列有什么区别，什么时候用循环队列
有哪些遍历二叉树的方法什么是中序遍历，不用递归实现前序遍历
4，计算机网络
http 工作在哪一层，这一层下面是哪一层
http 有状态吗？Cookie 与 session 有什么区别？Cookie 有时间期限吗？访问不同网站的 session 能相互 get 对方吗？
5，操作系统
大端小端？（有点懵，操作系统里面没听过这个词）
虚拟内存？硬盘和内存有什么区别？
6，数据库
数据库特性？持久性指什么？索引用来干什么？
7 C++
C++与 C 语言有什么区别？C++好在哪里？重载？C++多态？抽象类的关键字？
虚函数的关键字？

祝看到本文的宝宝都能找到满意的工作！