面试题 1：编码实现字符串转化为数字

编码实现函数 atoi()，设计一个程序，把一个字符串转化为一个整型数值。例如数字：“5486321”，转化成字符：5486321。

【答案】

```
int myAtoi(const char * str) {
int num = 0;      //保存转换后的数值
 int isNegative = 0;  //记录字符串中是否有负号
 int n =0;
 char *p = str;
if(p == NULL)              return -1;   //判断指针的合法性
 while(*p++ != '\0')     n++;          //计算数字符串长度
 p = str; //  指针  复原
 if(p[0] == '-')       isNegative = 1; //判断数组是否有负号
char temp = '0';
 for(int i = 0  i < n; i++)  {
       char temp = *p++;
        if(temp > '9' ||temp < '0')    continue;    //滤除非数字字符
         if(num !=0 || temp != '0')     {  //滤除字符串开始的 0 字符
           temp -= 0x30;   //将数字字符转换为数值
          num += temp *int( pow(10 , n - 1 -i) );
           }
 }

 if(isNegative)     return (0 - num);      //如果字符串中有负号，将数值取反
else                    return num;            //返回转换后的数值
}
```

注意：此段代码只是实现了十进制字符串到数字的转化，读者可以自己去实现 2 进制，8 进制，10 进制，16 进制的转化。

排序算法

```
/*
简单 选择排序                    时间复杂度 O(n2)
1】遍历每一个 待排序数组的 元素   i=0 ~ size-2
2】初始化 无序数组中 最小元素的 下标   为待排序数组的首元素   min = i
3】遍历每一个无序数组的 元素   j=i+1 ~ size-1
4】找到无序数组 中最小元素的下标 min = j
5】最小元素发生了变换 min != i 交换最小元素 min 和待排序数组首元素 i   swap(a,min,i)
*/
        void SelectSort(vector<int> data)
        {
           // 1】遍历每一个 待排序数组的 元素   i=0 ~ size-2
             for ( int i = 0; i < data.size()-1; ++I ){
                int min = i;// 2】初始化 无序数组中 最小元素的 下标   首元素
                for( int j = i+1; j < data.size(); ++j ){// 3】遍历每一个无序数组的 元素   i+1 ~ size-1
                    if( data[j]<data[min] ) min = j; // 4】找到无序数组 中最小元素的下标
                }
                if( min != i ){// 5】最小元素发生了变换   交换最小元素 和   待排序数组 首元素
                     int temp = data[i];
                     data[i] = data[min]; // 最小元素 放在 待排序数组 首元素位置
                     data[min]= temp;   // 无序数组首元素的值放到 原来最小元素的位置
                }
             }    for ( int i = 0; i < data.size(); ++I )   cout << data[i] << " ";// 打印
        }
/*堆排序*/
void HeapSort(vector<int> data)
 {   for (int i = data.size() / 2 - 1; i >= 0; --i)     Adjust(data, i, data.size());//1.构建大顶堆
     for (int i = data.size() - 1; i >= 0; --i) {
     int num = data[0];// 堆顶元素
     data[0] = data[i];// 末尾元素
     data[i] = num;//      换堆顶元素与末尾元素
     Adjust(data, 0, i);// 调整堆结构 构建大顶堆
      }
void Adjust(vector<int> &data, int i, int size)
     {
      int parent = data[i];//先取出当前元素 i 父节点 的值
      //从 i 结点的左子结点开始，也就是 2i+1 处开始
      for(int c = 2*i+1; c < size; c = 2*c + 1){
      // 找子节点中的 最大值
          if (c < size - 1 && data[c] < data[c + 1])     c++; //找到两个子节点中最大的数
          //如果左子结点小于右子结点，c 指向右子结点 2*i+1+1
          // 如果子节点中的 最大值 大于 父节点 则替换 父节点
          if ( data[c] > parent)//如果子节点 data[c] 大于父节点 parent，则将子节点值赋给父节点
          （不用进行交换）
```

```
                {   data[i] = data[c];   i = c; } //  更新  父节点   id
            else    break;
            data[i] = parent;//将 parent  值放到最终的位置
            }
}
/*  简单插入排序    时间复杂度  最坏  O(n2)
1】遍历  无序表中  的元素   i=1 ~ n-1
2】取无序表的第一个元素  取名为  temp = data[i]
3】从后向前  遍历  有序表, 选择合适的位置插入  无序表的  第一个元素   temp
4】  有序表中比需要插入的元素  temp    大的依次后移
5】在有序表中找到比需要插入的元素 temp  小的元素  data[j]
6】将需要插入的元素 temp  插入到有序表  data[j]后面一个位置  data[j+1]
*/
            void InsertSort(vector<int> data)
            {      // 1】遍历  无序表中  的元素   i=1 ~ n-1
                for (int i = 1; i < data.size(); ++i)
                {
                    int temp = data[i];// 2】取无序表的第一个元素  取名为  temp = data[i]
                    int j;
                   //3】从后向前遍历有序表, 选择合适的位置插入无序表的第一个元素 temp
                    for (j = i-1; j >= 0; --j)
                    {   // 4】  有序表中比需要插入的元素  temp    大的依次后移
                        if (data[j] > temp)   data[j + 1] = data[j];
                        else  break; // 5】在有序表中找到比需要插入的元素 temp  小的元素  data[j]
                    }
                       // 6】将需要插入的元素 temp  插入到有序表  data[j]后面一个位置  data[j+1]
                    data[j + 1] = temp; }
            }
/*  简单插入排序很循规蹈矩   希尔排序在数组中采用跳跃式分组的策略   */
void ShellSort(vector<int> data)
{
   // 1】遍历每一次  分组   int gap = data.size()/2; gap > 0; gap/=2
for (int gap = data.size()/2; gap > 0; gap/=2)   //增量 gap，并逐步缩小增量  分组数量
 { // 2】遍历  每一个  间隔分组数组   从第 gap 个元素，逐个对其所在组进行直接插入排序操作
       for ( int i = gap;   i <=data.size();   ++I ) {
            int j;
            int temp =   data[i];//无序组内的第一个元素
            for (j = i-gap; j >= 0; j -= gap) //3】从后向前  遍历  分组的  有序表
            {   // 4】有序表中比需要插入的元素  temp    大的依次后移  (间隔 gap)
                if (data[j] > temp) data[j + gap] = data[j];
               else  break; } // 5】在有序表中找到比需要插入的元素 temp  小的元素  data[j]
              // 6】将需要插入的元素 temp  插入到有序表  data[j]后面一个位置  data[j+gap]
              data[j + gap] = temp;}}}
```

有一个由大小写组成的字符串，现在需要对他进行修改，将其中的所有小写字母排在大写字母的前面(不要求保持原顺序)。
这里使用从左往后扫描的方式。字符串在调整的过程中可以分成两个部分：已排好的小写字母部分、待调整的剩余部分。用两个指针 i 和 j，其中 i 指向待调整的剩余部分的第一个元素，用 j 指针遍历待调整的部分。当 j 指向一个小写字母时，交换 i 和 j 所指的元素。向前移动 i、j，直到字符串末尾。代码如下：

```
void Proc( char *str ) {
    int i = 0, j = 0;
    //移动指针 i, 使其指向第一个大写字母
    while( str[i] != '\0' && str[i] >= 'a' && str[i] <= 'z' )   ++i;
    if( str[i] != '\0' ) {
        //指针 j 遍历未处理的部分，找到第一个小写字母
        for( j=i;   str[j] != '\0'; ++ j ) {
            if( str[j] >= 'a' && str[j] <= 'z' ) {
                char tmp = str[i];
                str[i] = str[j]; //  确保小写的放在前面
                str[j] = tmp;
                i++;   //  确保小写的放在前面
            }
        }
    }
}

/*
冒泡排序          时间复杂度  最坏  O(n2)
  1】执行每一次冒泡  总共需要  n-1 次  冒泡
  2】每次冒泡需要  n-1-i 次  比较相邻元素
  3】比较相邻元素
  4】把大的元素  移动到后面，小的元素移动到前面
*/
  void BubbleSort(vector<int> data)
   {
     for (int i =0; i < data.size()-1; ++i){ //   1】执行每一次冒泡  总共需要  n-1 次  冒泡
        bool   done = true; //  完成排序标志
        for(int j =0; j < data.size()-1-i;   ++j){ // 2】每次冒泡需要  n-1-i 次  比较相邻元素
              if( data[j] > data[j+1] ){ // 3】比较相邻元素
                int temp = data[j];   //4】把大的元素  移动到后面，小的元素移动到前面
                data[j] = data[j+1];
                data[j+1] = temp;
                done = false;   }    //本次冒泡发生  错序，排序还未完成
            } if( done ) break;//在某次冒泡中  发现  数组已经  排序完成  则结束排序
     }}
```

```
/*快速排序 是对冒泡排序的改进  基本思想是通过一趟排序将数据分成两部分，
一部分中的数据都比另一部分中的数据小，再对这两部分中的数据再排序，直到整个序列有序。
分割 使枢轴记录的 左边元素比右边元素小    平均时间复杂度均为 O(nlogn)
0】 取区间第一个元素 data[beg]  值作为 比较值  temp = data[beg];
1】从右向左开始检查。如果 end 位置的值大于 temp,end--，直到遇到一个小于 temp 的值 data[end]
2】此时将小于 temp 的这个值 data[end]  放入 data[beg]的位置 data[beg] = data[end];。
3】然后从 beg 位置开始从左向右检查，beg++，  直到遇到一个大于 temp 的值 data[beg]。
4】此时将大于 temp 的值 data[beg]   放入 end 位置  data[end] = data[beg];
5】重复第一步，直到 beg 和 end 重叠  转 6。
6】  将 temp 放入此位置  data[beg]  返回位置  beg
*/
void QuickSort(vector<int> data, int beg, int end){
    QuickSort_( data, beg, end);
}
void QuickSort_(vector<int> &data, int beg, int end)
{
      if (beg < end)
      {
            int index = Cut(data, beg, end);
            QuickSort_(data, beg, index - 1);//递归左边的区间
            QuickSort_(data, index + 1, end);//递归右边的区间
      }
}
int Cut(vector<int> &data, int beg, int end)
{
// 0】取区间第一个元素 data[beg]  值作为 比较值  temp = data[beg];
      int temp = data[beg];
      while (beg < end)//  从右找小的元素从左找大的元素调换直到碰头
      {
// 1】从右向左开始检查。如果 high 位置的值大于 temp，--end，继续往前检查，直到遇到一个
            小于 temp 的值 data[end]。
            while (beg < end && data[end] >= temp)    --end;//
// 2】此时将小于 temp 的这个值 data[end]  放入 data[beg]的位置。
            data[beg] = data[end];
// 3】然后从 beg 位置开始从左向右检查，++beg，  直到遇到一个大于 temp 的值  data[beg]  。
            while (beg < end && data[beg] <= temp) ++beg;//
// 4】此时将大于 temp 的值 data[beg]   放入 end 位置
            data[end] = data[beg];
      }
// 6】  将 temp 放入此位置  beg  返回位置  beg
      data[beg] = temp;//
      return beg;//  返回  分割位置
}
```

例题 1、最小的 k 个数。输入 n 个整数，找出其中最小的 k 个数，例如输入
4、5、1、6、2、7、3、8、1、2，输出最小的 4 个数，则输出 1、1、2、2。 大项堆 快排

根据 Cut 返回的位置来判断，在位置左边的都小于 data[index]，右边的都大于 data[inedx]
当 k-1 小于 index，我们需要的最小的 k 个数在左边中部分，因此只需要对左边再次分割
当 k-1 大于 index，我们需要完整的左边和 右边的部分，因此需要对右边进行分割。
指导返回的 index 和 k-1 相等时，结束，数组中的前 k 个数即为最小的数。

```
void GetLeastNumbers_by_partition(vector<int> &data, int beg, int end, int k) {
      int index = Cut(data, beg, end);
      while(index != k-1){
      if(index > k-1)    index = Cut(data, beg, index-1);
      else                  index = Cut(data, index+1, end); }
      for(int i = 0; i<k;++I) cout<<data[i];
}
```

编写 打印数组中 两个元素和为 sum 的元素下标

```
void Qarray(vector<int> a, int sum){
for(int i=0;    i<a.size()-1;    ++i){
   for(int j =i+1;    j <a.size();    ++j)
     if(a[i] + a[j] == sum){
      cout <<    i <<    j    << endl;
      return;
   } } }
```

二分查找比顺序查找效率高，但它只适用于有序的数据集。二分查找也叫折半查找。

```
// 循环迭代实现   升序数组 查找的元素   范围开始   结束
int BinarySearch(int *array, int key, int low, int high)
{
      int mid;
      while (low <= high)// 范围正常
      {
            //mid = (low + high) / 2;//中间元素的 下标 low + high 可能会有整数溢出情况
            mid = low + ((high-low) >> 1);// 避免整数溢出    mid = low + (high-low) / 2;
            if (key == array[mid])    return mid;//相等 返回元素下标
            else if (key > array[mid]) low = mid + 1; //所查元素比 中间元素大则在后区间查找
            else high = mid - 1; //所查元素比 中间元素小 则 在前区间查找
      }
      return -1;
}
// 递归实现 递归思想会被经常用到，更加突出了编程解决问题的高效
int binSearch(int *a, int low, int high, int &key)
{
   int mid = low + ((high - low) >> 1);
   if(low > high) return -1;//查找完毕没有找到答案
   else{
        if(key == a[mid])          return mid;// 返回查找的下标
        else if(key > a[mid])      return binSearch(a, mid+1, high,    key);//在后半区间查找
```

```
        else                    return binSearch(a, low,    mid-1, key);//在前半区间查找
    }
}
```

常识概念：

1 C 与 C++的区别

C++ 的核心思想是抽象（Abstraction），面向对象的编程；C 的核心思想是具象（Concretion），面向具体过程的编程。

C 如何实现面向对象：面向对象无非就是 **封装 继承** 和 **多态。** 封装：可以用结构体 struct 实现；继承：指针实现，就是把父类包含在结构体中；多态：可以用指针实现。一般实现多态，父结构体必须是子结构体的第一个元素，这样就可以通过强制转换子类和父类随意转换。

```
struct parent{
        int a;   };
struct child{
        struct parent p; // 类似继承源自 结构体 parent 第一个元素
        int b;     };
//所以才有下面的转换
struct child *c=(struct child *)malloc(sizeof(struct child));   // 初始化
c->p.a=10; // ->:用来得到父类 访问
struct parent *c=(struct parent *) &(c->p);   // 父类 转换成子类
//c 语言中有很多这样通过首地址来转换类型的
```

C++的多态性用一句话概括就是：在基类的函数前加上 virtual 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数

2 const 有什么用途

- 1：定义只读变量，即常量
- 2：修饰函数的参数和函数的返回值
- 3：修饰函数的定义体，这里的函数为类的成员函数，被 const 修饰的成员函数代表不修改成员变量的值

3. 指针和引用的区别

- 1：引用是变量的一个别名，内部实现是 **只读指针 *const**
- 2：引用只能**在初始化时被赋值**，其他时候值不能被改变，指针的值可以在任何时候被改变
- 3：**引用不能为 NULL，指针可以为 NULL**
- 4：引用变量内存单元保存的是被引用变量的地址
- 5：**“sizeof 引用" = 指向变量的大小 ， "sizeof 指针"= 指针本身的大小**
- 6：引用可以取地址操作，返回的是被引用变量本身所在的内存单元地址
- 7：引用使用在源代码级相当于普通的变量一样使用，做函数参数时，**内部传递的实际是变量地址，避免拷贝**

4. C++中有了 malloc / free，为什么还需要 new / delete

- 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 **申请动态内存和释放内存**。
- 2：对于非内部数据类型的对象而言，光用 maloc/free **无法满足动态对象**的要求。对象在**创建的同时要自动执行 构造函数**，对象在**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，以一个能完成 **清理与释放内存工作** 的运算符 delete。注意 new/delete 不是库函数。

5 堆和栈的区别　一个由 c/C++编译的程序占用的内存分为以下几个部分

- 1、栈区（stack）－ **由编译器自动分配释放** ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
- 2、堆区（heap） － 一般**由程序员分配释放**， 若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。
- 3、全局区（静态区）（static）－，**全局变量和静态变量的存储**是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后有系统释放
- 4、文字常量区 － **常量字符串就是放在这里的**。 程序结束后由系统释放
- 5、程序代码区 － **存放函数体的二进制代码**。

6 不调用 C/C++ 的字符串库函数，编写 strcpy

```
char * strcpy(char * strDest, const char * strSrc)
    {   //目标字符串指针 strDest，源字符串指针 strSrc
        if ((strDest==NULL) || (strSrc==NULL))
            return NULL;// 空指针 返回 NULL
        char * strDestCopy = strDest; // 拷贝目标字符串 首地址指针
        while ( ( *strDest++ =   *strSrc++ ) != '\0' ); // 赋值源字符串内容
        *strDest = '\0';
        return strDestCopy;
    }
```

7 关键字 static 的作用

- 1. 函数体内 static 变量的作用范围为该函数体，不同于 auto 变量， **该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值。**
- 2. 在模块内的 static 全局变量可以被模块内所有函数访问，但不能被模块外其他函数访问。//文件 a.c 中定义全局变量 static　char a = 'A'; 变量 a 只在 a.c 中可见。
- 3. 在模块内的 static 函数只可被这一模块内的其他函数调用，这个函数的使用范围被限制在声明它的模块内。
- 4. 在类的 static 成员变量属于整个类所拥有，对类的所以对象只有一份拷贝。

- 5. 在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

8 在 c++程序中调用被 C 编译器编译后的函数，为什么要加 extern“C”

C++语言支持函数重载，C 语言不支持函数重载，函数被 C++编译器编译后在库中的名字与 C 语言的不同，假设某个函数原型为：void foo(int x, inty);
该函数被 C 编译器编译后在库中的名字为: _foo
而 C++编译器则会产生像: _foo_int_int 之类的名字。
为了**解决此类名字匹配的问题**，C++提供了 C 链接交换指定符号 extern "C"。

9 野指针(猫)：

野指针指向一个已删除的对象或未申请访问受限内存区域的指针。
设想，你家里有个物体，不知什么时候突然出现，也不知什么时候突然消失。会把你的东西乱挪位置，还时不时打碎个瓶子。
这个物体，在计算机的世界叫野指针。在现实世界，叫猫。
形成：**A 指针变量未初始化。**
任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的，它会乱指一气。应创建时初始化，设置为 NULL/合法的地址
B 指针释放后未置空。
有时指针在 free 或 delete 后未赋值 NULL，便会使人以为是合法的。别看 free 和 delete 的名字（尤其是 delete），它们只是把指针所指的内存给释放掉，但并没有把指针本身干掉。此时指针指向的就是“垃圾”内存。
C 指针操作超越变量作用域。
函数返回值为 函数内部定义的局部变量（存储在**栈内存**，离开函数就被销毁）的指针或引用时发生。
规避：A 指针初始化时置 NULL。因为任何指针变量(除了 static 修饰的指针变量)刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的。Int * p = NULL;
B 释放内存时，指针置 NULL。delete 和 free 只是把内存空间释放了，但是并没有将指针 p 的值赋为 NULL。 **delete p; p=NULL; //以防出错**

10 进程和线程：进程相当于店铺（饭店、药店），线程相当于店铺里的工人。

有了线程，一个进程就可以同时做很多事情 ,“进程是爹妈，管着众多的线程儿子”.
进程是应用程序的执行实例。 线程是执行进程中的路径。
进程：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。
只有当一个程序被 OS 加载到内存中，cpu 对其执行时，这个过程是动态的，称为进程。
更小的能独立运行的基本单位——**线程. thread： 是进程中的一条执行流程(执行路径)**
进程（店面）的缺点： 创建时间较长；共享数据不方便，需要管道；代码较复杂，
fock()一个进程，在执行 execXXX(); 等待进程结束 waitpid(pid, &pid_status, 0);
普通进程，单线程，一个人干活。
pid_t pid =fork();//会返回一个整形值，子进程返回 0，父进程返回一个正数，复制出错返回 -1
Fork()的简单实现： 对子进程分配内存，复制父进程的内存和 CPU 寄存器到子进程中，
开销大大大！ 99%调用 fork 是为了接下来调用 exec：fork 中内存复

制没用，子进程将可能关闭打开的文件和连接，开销高。

等待： wait()：wait 使父进程睡眠，当子进程调用 exit 时操作系统解锁父进程。

系统调用是被父进程用来等待子进程的结束，一个子进程向父进程返回一个值，父进程必须接受这个值并处理。

为什么要让父进程等？而不是直接结束?

当进程执行完毕退出后，几乎所有资源都回收到 OS 中。但有个资源很难回收，就是 PCB，PCB 是代表进程存在的唯一标识，OS 要依据 PCB 执行回收。这个功能由父进程完成。

如果父进程先于子进程死掉了，它释放所有的数据结构，这个进程死亡。 它的子进程将有由系统 Init 进程 来接管他，成为 init 进程的子进程，**被称为孤儿进程。**

最后清理所有等待的僵尸进程。

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程(进程号为 1)所收养，并由 init 进程对它们完成状态收集工作。孤儿进程是没有父进程的进程，孤儿进程这个重任就落到了 init 进程身上，**init 进程就好像是一个民政局，**专门负责处理孤儿进程的善后工作。每当出现一个孤儿进程的时候，内核就把孤 儿进程的父进程设置为 init，而 init 进程会循环地 wait()它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。因此孤儿进程并不会有什么危害。

僵尸进程：一个进程使用 fork 创建子进程，如果子进程退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

僵尸状态：僵尸进程 的状态

就是子进程调用了 EXIT()结束，但父进程还没有等待 wait()它的时候，它变成一个僵尸进程。子进程将死，但还没死。无法正常工作，只是等待被父进程回收。但是如果该进程的父进程已经先结束了，那么该进程就不会变成僵尸进程。

僵尸进程的危害：

每个进程在退出的时候，内核释放该进程所有的资源，但是会保留**进程号 PID，退出状态**等信息（进程号一直被占用），直到父进程通过等待 wait()/等待 waitpid()来取走时才释放。如果存在大量的僵尸进程，会占用大量的进程号，而系统的进程号是有限的，将因为没有可用的进程号而导致系统不能产生新的进程。

僵尸进程的避免：

- 1.父进程通过 wait()和 waitpid() 等函数等待子进程结束，这会导致父进程挂起。
- 2. 如果父进程很忙，那么可以用 signal 函数为 SIGCHLD 安装 handler，因为子进程结束后， 父进程会收到该信号，可以在 handler 中调用 wait 回收。
- 3. 如果父进程不关心子进程什么时候结束，那么可以用 signal（SIGCHLD,SIG_IGN）通知内核，自己对子进程的结束不感兴趣，那么子进程结束后，内核会回收， 并不再给父进程发送信号。
- 4. 还有一些技巧，就是 fork 两次，父进程 fork 一个子进程，然后继续工作，子进程 fork 一个孙进程后 该子进程退出，需要父进程等待取走进行死亡信息。那么孙进程被系统 init 进程接管，孙进程结束后，init 进程 会回收。不过子进程的回收 还要自己做。

```
pid_t pid;
if ((pid = fork()) < 0)    err_sys("fork error");//  子进程创建错误
else if (pid == 0) {    /* first child */          //  子进程运行
        if ((pid = fork()) < 0)    err_sys("fork error");//孙进程创建错误
        else if (pid > 0)    exit(0);//  子进程创建孙进程  后直接  退出  exit(0)
        else {//孙进程  运行
            //  因为孙进程的父进程死亡，孙进程由  init 进程接管(孤儿被领走)
            sleep(2);
            printf("second child, parent pid = %d ",getppid());//  获取  init 的进程号
            exit(0);//  孙进程  退出  由  init 进程等待  wait()
        }//  孙进程结束
}//子进程结束
else{//  父进程需要等待  子进程结束
   if (waitpid(pid,NULL,0) != pid)err_sys("waitpid error");//等待子进程结束错误
  exit(0);//  父进程退出
}
```

线程（员工）的优点： 启动快，又可以共享数据，可以并行执行多个线程，一个进程可以多开几个线程（多雇几名员工），并行工作。

线程执行开销小，但不利于资源的管理和保护；而进程正相反。

关系：

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。相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

区别：

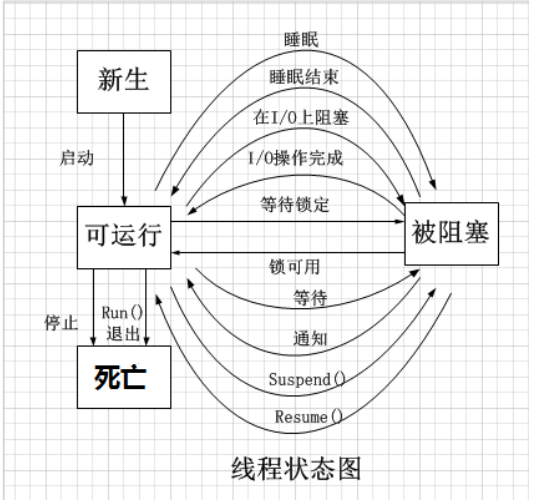
进程有独立的 **地址空间**，一个进程崩溃后，在保护模式下不会对其它进程产生影响，**而线程只是一个进程中的不同执行路径。**

线程有自己的 **堆栈和局部变量**，但线程之间没有单独的地址空间，它们共享内存，从而极大地提高了程序的运行效率。但是一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

对于一些要求 **同时进行**且要 **共享某些变量** 的 **并发操作**只能用线程不能用进程。

线程 在 **执行过程** 中与 进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

10.1 线程的基本概念、线程的基本状态及状态之间的关系？



线程状态图

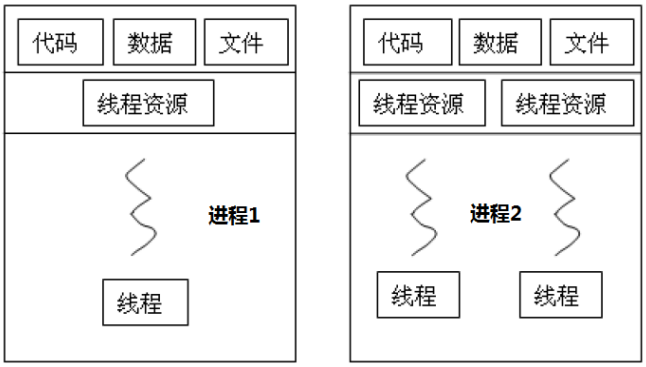
线程，有时称为轻量级进程，是 CPU 使用的基本单元；

它由线程 ID、程序计数器、寄存器集合和堆栈组成。它与属于同一进程的其他线程共享其代码段、数据段和其他操作系统资源（如打开文件和信号）。

线程有四种状态：

新生状态、可运行状态、被阻塞状态、死亡状态。状态之间的转换如左图所示。

10.2 线程与进程的区别？



- 1：**线程是进程的一部分**，所以线程有的时候被称为是轻权进程或者轻量级进程。
- 2：一个**没有线程的进程是可以被看作单线程的**，如果一个进程内拥有多个线程，线程的执行过程不是一条线（线程）的（路径），而是**多条线（线程）共同完成的。**
- 3：系统在运行的时候会**为每个进程分配不同的内存区域**，但是**不会为线程分配内存**（线程所使用的资源是它所属的进程的资源），**线程组只能共享资源**。那就是说，除了 CPU 之外（线程在运行的时候要占用 CPU 资源），计算机内部的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源。
- 4：与进程的控制块 PCB(process control block，进程控制块)相似，线程也有自己的控制表 TCB(thread control block 线程控制块)，但是 TCB 中所保存的线程状态比 PCB 表中少多了。
- 5：进程是系统所有资源分配时候的一个基本单位，拥有一个完整的虚拟空间地址，并不依赖线程而独立存在。

进程间的通信方式

无名管道、有名管道、信号、共享内存、消息队列、信号量(信号灯，访问共享内存)、套接字、文件。

10.3 多线程有几种实现方法？

1. 继承 Thread 类
2. 实现 Runnable 接口再 new Thread(YourRunnableOjbect)

- **多线程实现：**继承 Thread 类，重写 run()；实现 Runnable 接口，重写 run()；实现 Callable 接口，重写 call 函数
- **同步方式：**synchronized 修饰,wait(),notify()

10.4 多线程同步和互斥有几种实现方法，都是什么？

- 线程间的同步方法大体可分为两类：**用户模式和内核模式。**

内核模式 就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，

用户模式 就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。

内核模式下的方法有：事件，信号量，互斥量。

- **线程同步** 是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应**等待，直到消息到达时才被唤醒。**
- **线程互斥** 是指对于**共享的进程系统资源，在各单个线程访问时的排它性**。当有若干个线程**都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，**其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可

以看成是一种特殊的线程同步（下文统称为同步）。

10.41 通过锁机制实现线程间的同步：

- **1：初始化锁**
  在 Linux 下，线程的互斥量数据类型是 pthread_mutex_t 使用前,要对它进行初始化。
  静态分配： **pthread_mutex_t** mutex = PTHREAD_MUTEX_INITIALIZER;
  动态分配： int **pthread_mutex_init**(pthread_mutex_t *mutex, const pthread_mutex_attr_t *mutexattr);
- **2：加锁** 对共享资源的访问，要对互斥量进行加锁，如果互斥量已经上了锁，调用线程会阻塞，直到互斥量被解锁。
  int **pthread_mutex_lock**(pthread_mutex *mutex);
  int **pthread_mutex_trylock**(pthread_mutex_t *mutex);
- **3：解锁** 在完成了对共享资源的访问后，要对互斥量进行解锁。
  int **pthread_mutex_unlock**(pthread_mutex_t *mutex);
- **4：销毁锁** 锁在是使用完成后，需要进行销毁以释放资源。
  int **pthread_mutex_destroy**(pthread_mutex *mutex);

10.42 条件变量(cond) 实现线程间的同步： condition

互斥锁不同，条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。条件变量分为两部分: 条件和变量。

条件本身是由互斥量保护的。线程在改变条件状态前先要锁住互斥量。

一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。

条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

- **1：初始化条件变量。**
  静态态初始化， **pthread_cond_t** cond = PTHREAD_COND_INITIALIER;
  动态初始化， int **pthread_cond_init**(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
- **2：等待条件成立。**释放锁,同时阻塞等待条件变量为真才行。timewait()设置等待时间,仍未 signal,返回 ETIMEOUT(加锁保证只有一个线程 wait)
  int **pthread_cond_wait**(pthread_cond_t *cond, pthread_mutex_t *mutex);
  int pthread_cond_timewait(pthread_cond_t *cond,pthread_mutex *mutex,const timespec *abstime);
- **3：激活条件变量。** pthread_cond_broadcast（激活所有等待线程）
  int **pthread_cond_signal(**pthread_cond_t *cond);// 激活单个条件变量
  int **pthread_cond_broadcast**(pthread_cond_t *cond); //解除所有线程的阻塞
- 4：清除条件变量。无线程等待,否则返回 EBUSY
  int **pthread_cond_destroy**(pthread_cond_t *cond);

10.43 信号量(sem) 实现线程间的同步： semaphore 信号灯

如同进程一样，线程也可以通过 **信号量** 来实现通信，虽然是轻量级的。信号量函数的名字都以"sem_"打头。线程使用的基本信号量函数有四个。

- **1：信号量初始化**。
  int sem_init (sem_t *sem , int pshared, unsigned int value);
  这是对由 sem 指定的信号量进行初始化，设置好它的共享选项(linux 只支持为 0，即表示它是当前进程的**局部信号量**)，然后给它一个初始值 VALUE。
- **2：等待信号量。**给信号量减 1，然后等待直到信号量的值大于 0。
  int sem_wait(sem_t *sem);
- **3：释放信号量**。信号量值加 1。并通知其他等待线程。
  int sem_post(sem_t *sem);
- **4：销毁信号量**。我们用完信号量后都它进行清理。归还占有的一切资源。
  int sem_destroy(sem_t *sem);

10.5 操作系统（Operating System，简称 OS）

是管理和控制计算机硬件与软件资源的计算机程序，是直接运行在“裸机”上的最基本的系统软件，任何其他软件都必须在操作系统的支持下才能运行。操作系统是用户和计算机的接口，同时也是计算机硬件和其他软件的接口。

11 竞争条件（race condition）：多个进程或线程在读写某些共享数据，而最后的结果取决于进程运行的精确时序，这种情形叫做竞争。

例如：考虑下面的例子

假设两个进程 P1 和 P2 共享了变量 a。在某一执行时刻，P1 更新 a 为 1，在另一时刻，P2 更新 a 为 2。

因此两个任务竞争地写变量 a。在这个例子中，竞争的“失败者”(最后更新的进程）决定了变量 a 的最终值。

多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关，称为竞争条件。

12 TCP/IP 计算机 之间 进行通信交流的工具

OSI(Open System Interconnection 开放式系统互联)。

12.1 OSI 七层模型

OSI 层	功能 function	协议 protocol
应用层 Application layer	文件传输 电子邮件 文件服务 虚拟终端	HTTP 超文本传输协议 FTP 文件传输协议 DNS
表示层 Presentation layer	数据格式化，代码转换， 数据加密	无协议
会话层 Session layer	解除或建立与其他节点的联系	无协议
传输层 Transport layer	提供端对端的接口	TCP UDP
网络层 Network layer	为数据包选择路由（接力者）	检测 IP 地址 ICMP IPV4 IPV6
数据链路层 Data link layer	传输有地址的帧，错误检测功能	APR 地址检测
物理层 Physical layer	以二进制数据形式在物理 媒介上传输数据	IEEE 802.2 到 802.11

12.2 TCP/IP 五层模型的协议

应用层(http ftp)→传输层(TCP UDP) →网络层(IP) →数据链路层(CRC 编码) →物理层
TCP (Transmission Control Protocol) 传输控制协议
UDP 协议（User Datagram Protocol），即用户数据报协议

与 TCP 协议不同，UDP 协议是一个无连接协议，发送端和接收端不建立连接；UDP 协议不提供数据传送的保证机制，可以说它是一种不可靠的传输协议；UDP 协议也不能确保数据的发送和接收顺序，实际上，这种乱序性很少出现，通常只是在网络非常拥挤的情况下才可能发生。**TCP 协议植入的各种安全保障功能加大了执行过程中的系统开销，使速度受到严重的影响；UDP 协议执行速度快，适合视频、音频、文件等大规模数据的网络传输。**

互联网地址(ip 地址):网络上每个节点都必须有一个独立的 Internet 地址（也叫 IP 地址）网络号码+子网号+主机号。例如一个 B 类地址：210.30.109.134

域名系统 DNS: Domain Name System 是个分布的数据库，提供将主机名（网址）转换成 IP 地址的服务.

端口号(port): 是用在 TCP，UDP 上的一个逻辑号码，并不是一个硬件端口.
ARP (Address Resolution Protocol)叫做地址解析协议，是用 IP 地址换 MAC 地址(物理地址)的一种协议，而 RARP 则叫做逆地址解析协议
ICMP 协议: (Internet Control Message Protocol)Internet 控制信息协议。当传送 IP 数据包发生错误－－比如主机不可达，路由不可达等等，ICMP 协议将会把错误信息封包，然后传送回给主机。
ICMP 的应用—ping: 它利用 ICMP 协议包来侦测另一个主机是否可达。
原理是用类型码为 0 的 ICMP 发请求，收到请求的主机则用类型码为 8 的 ICMP 回应。ping 程序来计算间隔时间，并计算有多少个包被送达。
ICMP 的应用—Traceroute: 用来侦测主机到目的主机之间所经路由情况。不断增加 TTL 的值发送 UDP 数据给目标主机。

Traceroute 的原理是非常非常的有意思，它收到目的主机的 IP 后，首先给目的主机发送一个 TTL=1（**TTL 是 Time To Live 的缩写，该字段指定 IP 包被路由器丢弃之前允许通过的最大网段数量(路由次数)**）的 UDP(后面就 知道 UDP 是什么了)数据包，而经过的第一个路由器收到这个数据包以后，就自动把 TTL 减 1，而 TTL 变为 0 以后，路由器就把这个包给抛弃了，并同时产生 一个主机不可达的 ICMP 数据报给主机。主机收到这个数据报以后再发一个 TTL=2 的 UDP 数据报给目的主机,然后刺激第二个路由器给主机发 ICMP 数据 报。如此往复直到到达目的主机。这样，traceroute 就拿到了所有的路由器 ip。从而避开了 ip 头只能记录有限路由 IP 的问题。
而 traceroute 发送的是端口号>30000(真变态)的 UDP 报，所以到达目的主机的时候，目的 主机只能发送一个端口不可达的 ICMP 数据报给主机。主机接到这个报告以后就知道，主机到了，所以，说 Traceroute 是一个骗子一点也不为过:)

UDP 是传输层协议，和 TCP 协议处于一个分层中，但是与 TCP 协议不同，UDP 协议并

不提供超时重传，出错重传等功能，也就是说其是不可靠的协议。由于很多软件需要用到 UDP 协议，所以 UDP 协议必须通过某个标志用以区分不同的程序所需要的数据包。端口号的功能就在于此。

TCP 和 UDP 处在同一层----**传输层**，但是 TCP 和 UDP 最不同的地方是，TCP 提供了一种可靠的数据传输服务，TCP 是面向连接的，也就是说，利用 TCP 通信的两台主机首先要经历一个“拨打电话”的过程，等到通信准备结束才开始传输数据，最后结束通话。所以 TCP 要比 UDP 可靠的多，UDP 是把数据直接发出去，而不管对方是不是在收信，就算是 UDP 无法送达，也不会产生 ICMP 差错报文，这一经时重申了很多遍了。

TCP 中保持可靠性的方式就是超时重发，最可靠的方式就是只要不得到确认，就重新发送数据报，直到得到对方的确认为止。

一个 TCP 数据的发送过程：

- 双方建立连接**(三次握手协议建立连接)**
- 发送方给接受方传 TCP 数据报，然后等待对方的确认 TCP 数据报，如果没有，就重新发，如果有，就发送下一个数据报。
- 接受方等待发送方的数据报，如果得到数据报并检验无误，就发送 ACK(确认)数据报，并等待下一个 TCP 数据报的到来。直到接收到 FIN(发送完成数据报)
- 中止连接 **(四次挥手协议断开连接)**

HTTP 连接：HTTP 协议即超文本传送协议(Hypertext Transfer Protocol)，是 Web 联网的基础，也是手机联网常用的协议之一，HTTP 协议是建立在 TCP 协议之上的一种应用。

HTTP 连接最显著的特点是 客户端发送的每次请求 都需要 服务器回送响应，在请求结束后，会主动释放连接。从建立连接到关闭连接的过程称为“一次连接”。

由于 HTTP 在每次请求结束后都会主动释放连接，因此 HTTP 连接是一种“短连接”，要保持客户端程序的在线状态，需要不断地向服务器发起连接请求。通常的做法是，即时不需要获得任何数据，客户端也保持每隔一段固定的时间向服务器发送一次“保持连接”的请求，服务器在收到该请求后对客户端进行回复，表明知道客户端“在线”。若服务器长时间无法收到客户端的请求，则认为客户端“下线”，若客户端长时间无法收到服务器的回复，则认为网络已经断开。

12.3 TCP 和 UDP 有什么区别

TCP---传输控制协议,

- 提供的是面向连接、可靠的字节流服务。
- 当客户和服务器彼此交换数据前，必须先在双方之间建立一个 TCP 连接，之后才能传输数据。
- **TCP 提供超时重发**，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。

UDP---用户数据报协议，

- 是一个简单的面向数据报的运输层协议。
- UDP 不提供可靠性，它只是把应用程序传给 IP 层的数据报发送出去，但是并不能保证它们能到达目的地。
- 由于 UDP 在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，**故而传输速度很快。适合视频、音频、文件等大规模数据的网络传输。**

12.4 TCP 建立连接 三次握手协议建立连接

A TCP 客户端 主动打开 TCP 进程　　B TCP 服务器 被动打开 TCP 进程

1】B 服务器进程先创建传输控制模块 TCB，准备接受客户端进程的连接请求，然后 服务器进程就处于 **LISTEN(监听)状态**，等待客户的连接请求。

2】A 客户端 进程向 B 服务器**发出连接请求报文段**，首部中的同步位 SYN=1，同时选择一个初始序号 seq=x，这时，A 客户端就进入 **SYN-SENT(同步已发送)状态**

3】B 服务器 收到连接请求报文段后，向客户端 A 发送确认。在确认报文段中把 SYN 和 ACK 位都置为 1，确认号是 ack=x+1,同时也为自己选择一个初始序号 seq=y。这时 B 服务器 就进入 **SYN-RCVD(同步已收到)状态。**

4】A 客户端 收到服务器 B 的确认后，**还要向服务器 B 给出确认**。确认报文段的 ACK 置为 1，确认号 ack=y+1，而自己的序号 seq=x+1。这时，TCP 连接已经建立，A 进入 **ESTABLISHED(已建立连接)状态**，（防止已失效的连接请求报文段突然又传送到了 B，因而产生错误）

5】当服务器 B 收到客户端 A 的确认后，也会进入 ESTABLISHED 状态。

12.5 TCP 断开连接 四次挥手协议断开连接

数据传输结束后，通信的双方都可释放连接。

1】现在 A 和 B 都处于 ESTABLISHED 状态。客户端 A 先向 TCP 发出连接释放报文段，主动关闭 TCP 连接。这时 A 进入 **FIN-WAIT-1(终止等待 1)状态**，等待 B 的确认。

2】服务器 B 收到连接释放报文段后即发出确认(发出的不是连接释放报文段)，然后 B 就进入 **CLOSE-WAIT(关闭等待)状态**。

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

3】若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接。这时 B 发出的连接释放报文段必须使用 FIN=1，这是 B 就进入 **LAST-ACK 状态**，等待 A 的确认。

4】在 A 收到 B 的连接释放报文段后，必须对此发出确认。然后进入 TIME-WAIT 状态。必须再经过 2MSL 后，A 才进入到 CLOSED 状态。MSL 叫最长报文段寿命，一般为 2 分钟。

5】当 B 收到 A 发出的确认，就进入 CLOSED 状态。由此可见 **服务器 B 结束 TCP 连接的时间要比 客户端 A 早一些。**

12.6 socket（套接字）连接过程

建立 Socket 连接至少需要一对套接字，其中一个运行于客户端，称为 ClientSocket ，另一个运行于服务器端，称为 ServerSocket 。

服务器监听，客户端请求，连接确认。

1.服务器监听：是服务器端套接字并不指定具体的客户端套接字，而是一直处于等待连接的状态，实时监控网络状态。

2.客户端请求：是指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** 到或者接收到 **客户端套接字的连接请求**，

它就响应该请求，**建立一个新的线程**，把服务器端套接字的**描述发给客户端**(回应)，一旦客户端**确认此描述**(回应确认)，连接就建立好了。

注意：此时，**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。**

13 代码 编译 调试

13.1 GCC 编译器：

gcc 编译程序的流程：

源文件 -> 预处理器程序 -> 编译器程序 -> 汇编器程序 -> 连接器程序 -> 可执行文件

源文件类型有： .c 为 C 语言源文件 .C .cc .cpp .cxx 为 C++源文件 .m 为 Object-C 源文件

预处理之后的文件类型：.i 为 c 程序预处理后 .ii 为 C++程序预处理后

汇编之后的文件类型： .s .S

预处理头文件： .h

目标文件 ：.o

gcc 连接器将目标文件链接为一个可执行文件，一个大致的编译流程结束

存档文件、归档文件：.a

```
// filename: hello.c
#include <stdio.h>
int main(int argc, char **argv)
{   printf("Hello, Shi-Yan-Lou!");   }
```

// 编译 gcc hello.c –o hello 增加执行权限 chmod u+x hello 执行 ./hello

一个 –o 选项省略了中间很多步骤 –E 选项查看预处理之后的文件 .i

13.2GDB 代码调试器：

(GNU debugger) GDB 可以让你看到程序在执行过程时的内部流程，并帮你明确问题的所在。

```
// filename: hello.c
#include <stdio.h>
int func(int n)
{
    int sum=0,i;
    for(i=0; i<n; i++)   sum+=i;
    return sum;
}
int main(void)
{
    int i;     long result = 0;
    for(i=1; i<=100; i++)   result += i;
printf("result[1-100] = %ld \n", result );
printf("result[1-250] = %d \n", func(250) );}
```

// 首先在编译时，必须要把调试信息加到可执行文件中。

// 对于 c 程序 gcc **-g** hello.c -o hello 对于 c++程序 g++ **-g** hello.cpp -o hello

gcc –g gdb.c –o testGDB 编译时加入 调试信息

gdb testgdb <---------- 启动 gdb

l <---------- 键入 l 相当于 **list 列出源码**

break 16 <---------- **设置断点，在源程序 16 行处 break 16**

break func <---------- 设置断点，在调用函数 func() 的地方 即函数入口处

info break <---------- **查看设置的断点信息 info break**

r <---------- 继续运行程序 run 命令 的缩写
Breakpoint 1, main () at test.c:16 <---------- 会在设置的断点 1 处停住。
n <---------- 单条语句执行，next 命令简写
c <---------- 继续运行程序，continue 命令简写。
result[1-100] = 5050 <----------程序输出。
Breakpoint 2, func (n=250) at test.c:5 <---------- 会在设置的断点 2 处停住。
n
p i <---------- 打印变量 i 的值，print 命令简写。
bt <---------- backtrace 回溯 查看函数堆栈。通过 gdb 的堆栈跟踪，可以看到所有已调用的函数列表，以及每个函数在栈中的信息
finish <---------- 退出函数
c <---------- 继续运行程序，continue 命令简写。
q <---------- 退出 gdb，quit。

14 使用过的 shell 命令

复制修改文件名 cp 移动 mv 创建目录 mkdir 删除目录 rmdir 创建文件 touch
删除文件 rm 打印当前路径 pwd 切换当前路径 cd 列出文件列表 ls
查看文件/连接文件 cat 分页查看文件 more/less 查看文件顶部 head 查看文件顶部 tail
文件文字统计 wc 显示消息 echo 查找信息 grep 查看磁盘使用情况 du df
查看帮助信息 man 搜索文件 find 杀死进程 kill 超级用户权限 sudo

15 public/private/protected 的具体区别

对于继承自己的 class，base class 可以认为他们都是自己的子女，而对于和自己一个目录下的 classes，认为都是自己的朋友，在类内被 friend 修饰的函数或类被称为友元，是有元函数、友元类。
public：public 表明该数据成员、成员函数是对所有用户开放的，所有用户都可以直接进行调用。
private：private 表示私有，私有的意思就是**除了 class 自己和 友元** 之外，任何人都不可以直接使用，私有财产神圣不可侵犯嘛，即便是子女(子类)，朋友，都不可以使用。
在 A 类内部 定义的友元类 B 类，B 类可以访问 A 内部的包括私有的所有成员。
A 类的友元函数也可以访问 A 的 private 成员。

protected：protected 对于子女、朋友来说，就是 public 的，可以自由使用，没有任何限制，而对于其他的外部 class，protected 就变成 private，不可使用。

16 静态链表和动态链表

静态链表 在某些语言中指针是不被支持的,只能使用数组来模拟线性链表的结构.在数组中每个元素不但保存了当前元素的值,还保存了一个”伪指针域”,一般是 int 类型,用于指向下一个元素的内存地址.

```
#define MAXSIZE 100;
typedef struct{
   ElemType data;
```

```
    int cur;
}component, SLinkList[MAXSIZE];   // 结构体变量   结构体数组
```

这种链表在初始时必须分配足够的空间，**也就是空间大小是静态的，**在进行插入和删除时则不需要移动元素，修改指针域即可,所以仍然具有链表的主要优点(快速插入和删除)。

动态链表：如果程序支持指针,则可按照我们的一般形式实现链表，需要时分配,不需要时回收即可.

```
typedef struct    node{
  EleType data;
  struct node * pNext;
}Node;
```

链表结构可以是动态地分配存储的，即在需要时才开辟结点的存储空间，实现动态链接。

malloc 函数

malloc(sizeof(int));开辟 2 个字节的存储空间，
molloc(sizeof(struct student));开辟 10 个(4+4+2)字节。
该函数返回值是 void* 类型，因此调用时需要强制转换成需要的类型。

```
int * p = (int *) malloc(sizeof(int));
struct people * p = (struct people *) malloc(sizeof(struct student));
```

free 函数

其作用是释放由 p 指向的内存区（堆栈区），即将这部分内存还给系统。我们要注意动态开辟的内存在不用之后应及时还给系统，以免造成内存“遗漏”。free 函数无返回值。释放后需要将指针置位 p = NULL; 避免出现也指针。

17 什么时候要用虚析构函数

通过基类的指针来删除派生类的对象时，基类的析构函数应该是虚的。否则其删除效果将无法实现。
一般情况下，这样的删除只能够删除基类对象，而不能删除子类对象，形成了删除一半形象，从而千万内存泄漏。
原因：
在公有继承中，基类对派生类及其对象的操作，只能影响到那些从基类继承下来的成员。如果想要用基类对非继承成员进行操作，则要把基类的这个操作（函数）定义为虚函数。那么，析构函数自然也应该如此：如果它想析构子类中的重新定义或新的成员及对象，当然也应该声明为虚的。
注意：
如果不需要 基类对 派生类 及对象进行操作，则不能定义虚函数（包括虚析构函数），因为这样会增加内存开销。

18 c++怎样让返回对象的函数不调用拷贝构造函数

我们知道拷贝构造函数有两种“默默”的方式被调用：

- 向函数传入 值参数
- 函数返回 值类型
- 讨论函数返回值类型的情况，得到结论是：

1. 当对象有拷贝构造函数（系统为我们生成、或者我们自己写拷贝构造函数）可以被隐式调用时，函数返回时会使用拷贝构造函数。

- 2. 当对象的拷贝构造函数声明成为 explicit（不能被隐式调用时），函数返回时会使用 move 构造函数。

```
/*
计算两个字符串之间的编辑距离
从 s2 字符串 到 s1 字符串 需要多少次 插入、删除、替换操作
*/
int minStrEditDis(char* s1, char* s2){
    int i,j;
    // 两个字符串的长度
    int n,m;
    n = strlen(s1);
    m= strlen(s2);
    int dis[n+1][m+1];// 创建一个二维数组
    for(i=0;i<=n;i++)
        for(j=0;j<=m;j++)
            dis[i][j]=INF;//赋值为 INF
    dis[0][0]=0;// 第一个 认为相等 距离为 0
    for(i=1; i<=n; i++)
        for(j=1; j<=m; j++)
        {
            dis[i][j] = min(dis[i][j], dis[i-1][j]+1); //delete 删除
            dis[i][j] = min(dis[i][j], dis[i][j-1]+1);//insert 插入
            //substitute 替换
            if(s1[i-1] != s2[j-1])// 不 相等 距离是 上一匹配到 dis[i-1][j-1]的距离 + 1
                dis[i][j] = min(dis[i][j], dis[i-1][j-1]+1);
            else // 相等 的话，距离还是 上一匹配到 dis[i-1][j-1]为止的距离
                dis[i][j] = min(dis[i][j], dis[i-1][j-1]);
        }
    return dis[n][m];
}
```

1 旋转字符串

将一个字符串 XY 分成 X(m 个字符)和 Y(n-m 个字符)两个部分，在每部分字符串上定义反转操作，如 X^T，即把 X 的所有字符反转（如，X="abc"，那么 X^T="cba"），那么就得到下面的结论：(X^TY^T)^T=YX，显然就解决了字符串的反转问题。

```
void ReverseString(char *s, int form, int to)
{
    while(from < to)// 从首尾开始交换首位位置的值，直到相遇
    {
        char tep  = s[from];// 保存左边
        s[from++] = s[to];  //右边 交换给左边 同时左边向右移动一个位置
        s[to--]   = tep;    // 前右边保存的值 交换给左边 同时 右边向左移动一个位置
```

```
        }
    }


// 前 m 后 放在尾部
    // 反转[0..m - 1] 反转[m..n - 1] 反转[0..n - 1]
    void LeftRotateString(char* s, int n, int m)
    {
    m   %= n;// 避免左移数量大于 n 的情况
    ReverseString(s, 0, m - 1);// 反转[0..m - 1]，套用到上面举的例子中，就是 X->X^T，即 abc->cba
    ReverseString(s, m, n - 1);// 反转[m..n - 1]，例如 Y->Y^T，即 def->fed
    ReverseString(s, 0, n - 1);// 反转[0..n - 1]，即如整个反转，(X^TY^T)^T=YX，即 cbafed->defabc。
    }
```

2、单词翻转。输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变，句子中单词以空格符隔开。

为简单起见，标点符号和普通字母一样处理。例如，输入“I am a student.”，则输出“student. a am I”。

注解：先将整个字符串反转，在将其中每个单词反转回来

```
void Reverse_word(char *str)
{
    if(str == NULL)// 空指针 返回
        return;
    int len = strlen(str);// 字符串长度
    // 反转整个字符串
    ReverseString(str, 0, len - 1);
    int s = 0;//每个单词的 起始 index
    int e = 0;// 每个单词的 后 index
    for(int i = 0; i < len; i++) // 遍历字符串 查找每一个单词
    {
        e = i;
        if(str[e] == ' ')// 此处 遇到空格
        {
            ReverseString(str, s, e-1); // 反转这个单词
            s = e+1;// 更新 单词的起始指针
        }
    }
   ReverseString(str, s, len - 1); // 反转最后一个单词 因为最后一个单词后面无空格
}
```

3. 字符串包含

Hash 签名查找法

事实上，可以先把长字符串 a 中的所有字符都放入一个 Hashtable 里，
然后轮询短字符串 b，看短字符串 b 的每个字符是否都在 Hashtable 里，
如果都存在，说明长字符串 a 包含短字符串 b，否则，说明不包含。

再进一步，我们可以对字符串 A，用位运算（26bit 整数表示)计算出一个“签名”，再用 B 中的字符到 A 里面进行查找。

```
// “最好的方法”，时间复杂度 O(n + m)，空间复杂度 O(1)
// 这个方法的实质是用一个整数代替了 hashtable，空间复杂度为 O(1)，时间复杂度还是 O(n + m)。
    bool StringContain(string &a,string &b)
    {
        int hash = 0;
        for (int i = 0; i < a.length(); ++i)
        {
            hash |= (1 << (a[i] - 'A'));
// (a[i] - 'A')为 0~25 右移动一位 位或上签名 hash 得到 长串 a 的最后签名
// 相当于把 a 的每一个不同的 字符 放入 每一个二进行位上
        }
        for (int i = 0; i < b.length(); ++i)
        {
            if ((hash & (1 << (b[i] - 'A'))) == 0)
// 位与 上(1 << (b[i] - 'A')) 的 0 的话就说明对应的位上（字符）与 a 中无匹配
            {
                return false;// 对应位上 a 中未出现相应的字符 则查找失败
            }
        }
        return true;
    }
```

4. 字符串转换成整数

```
    int StrToInt(const char *str)
    {
        if(str == NULL)     return -1; // 空指针返回
        int n = 0;
        int isNeg = 0;      // 负数标志
        if(str[0] == '-')   isNeg = 1;
        while (*str != 0) // 遍历到字符串末尾
        {
            if(*str > '9' || *str < '0') {continue; ++str;}
// 跳过非数字字符 或者 直接 退出 return -1;
            int c = *str - '0';
            n = n * 10 + c;// 累加和
            ++str;          // 移动指针
        }
        if(isNeg) return -n;
        else      return n;
    }
```

5. 实现 string 到 double 的转换

```
double StrToDou(const char *str)
{
    if(str == NULL)    return -1; // 空指针返回
    double result = 0.0;          // 最后的值
    double dec = 10.0;// 小数位比值
    int isNeg = 0;       // 负数标志
    int isDec = 0;
    if(str[0] == '-')    isNeg = 1;
    while (*str != 0) // 遍历到字符串末尾
    {
        if(*str == '.') {isDec = 1; ++str;}//是小数
        else if(*str > '9' || *str < '0') {continue; ++str;}
// 跳过非数字字符 或者 直接 退出 return -1;
        if(!isDec) // 整数部分
            result = result * 10 + *str - '0';// 整数部分 累加和
        else //识别小数点之后进入 这个分支
            {
              result = result + (*str - '0')/dec;// 小数部分 算法
              dec *= 10;// 小数位比值递增
            }
        ++str;              // 移动指针
    }
    if(isNeg) return -result;
    else          return result;
}
```

6. 字符串 回文 判断 首尾并进判断法 分治法

```
// 输入字符指针 s 和 字符串大小 num
bool IsPalindrome(const char* s, int num)
{
    if(s == NULL || num < 2)    return false;//指针为空 / 字符串长度小于 2
    // 定义并初始化 首尾指针
    const char* low, *high;// 首尾指针
    low = s;// 首指针
    high = s + num - 1;// 尾指针
    while(low < high){
        if(*low != *high) return false;// 出现一次相对应位置的字符不相同 就返回错误
        ++low;
        --high;
    }
    return true; // 到这里的话 就证明 相对应位置上的 字符相同 为 回文
}
```

7. 利用快排思想 (得到大小数 不同类别)的 分区中枢

```
int Cut(int a[], int low, int high){
   int temp = a[low];// 区间第一个元素 可以随机选择 在替换到 low 的位置
   while(low < high){
      while((low < higt)&&(a[high] > temp)) --high;
//从右边向左寻找比 temp 小的元素 a[high]
      a[low] = a[high];// 把比 temp 小的元素 放在 原来 temp 的位置(左边小的区域)
      while((low < high)&&(a[low] < temp))   ++low;
// 从左边寻找比 temp 大的元素 a[low] 放在右边 high 的位置
      a[high] = a[low];
    }
   a[low] = temp; // 临时变量 放在 中枢位置
   return low;    // 返回中枢位置
}
```

// 在数组 a 中选择 k 个最小的元素 块选 非递归调用

```
void QuickSelest(int a[], int low, int high, int k){
   int index =   Cut(a, 0, n-1);// 首先得到一个中枢
   while(index != k-1){// 非递归调用
   if(index > k - 1) index = Cut(a, low, index -1);
// 左半部分小的元素个数大于 k 在左半部分找 中枢
    else   index = Cut(a, index+1, high);
// 左半部分小的元素个数小于 k 右半部分还有，在右半部分找 中枢
    }
  // 打印
  for(int i = 0; i<k; ++i) cout<<a[i];
  cout << endl;
}
```

// 而快排为 是递归调用

```
void QuickSort(int a[], int low, int high){
   int index;
   if(low < high){
      index = Cut(a, low, high);//选取中枢位置
      QuickSort(a, low, index-1);//对左边快排
      QuickSort(a, index+1, high);//对右边快排
   }
}
```

// 二分查找　　非递归版本

```
int BinarySearch(int a[], int low, int high, int key){
int mid;//中间元素
while(low < high){
int mid = low + ((high - low)>>1);     //中间元素的 下标 在循环体内 不停的被改变
if( key < a[mid] )          high = mid - 1;// 查找的元素小于中值   高区间移至 mid-1
elae if(key > a[mid] ) low    = mid + 1;// 查找元素大于中值， 将低区间移至 mid+1
else return mid;
}
return -1;//未找打 返回-1
}
```

// 二分查找　　递归版本

```
int BinarySearch(int a[], int low, int higt, int key){
int mid = low + ((high - low)>>1);//中间元素的 下标
if(low < high){
   if(key < a[mid])           return BinarySearch(a, low, mid-1, key);
   else if(key > a[mid]) return BinarySearch(a, mid+1, high, key);
   else return mid;
}
return -1;
    }
```

8. 寻找和为定值的两个数

```
// 二分（若无序，先排序后二分），时间复杂度总为 O(N log N)，空间复杂度为 O（1）;
     void findTwo2(int a[], int low, int high, int sum){
 //sort(a, a+high+1);     如果数组非有序的，那就事先排好序 O(N log N)
                  while(low < high){
                           if(a[low] + a[high] > sum)        --high; //太大 right 减少
                           else if(a[low] + a[high] < sum) ++low; //太小 low 增加
                           else cout << a[low] << a[high] < endl;
                  }
                 cout << "Can't found" < endl;
        }
```

1．说明形参、局部变量及局部静态变量的区别。编写一个函数，同时用到这三种形式。

形参　　相当于函数的自变量，其作用域为**函数声明与定义的范围**。
局部变量 是在函数体中所声明并定义的变量，其**作用域在该函数体内**。
局部静态变量 为在函数体中声明的静态变量，但其作用域是在此函数体及函数体后面程序段落中，生物周期自它被声明之时起，至整个程序结束终止。

函数示例及变量说明如下：

```
void func(int intpara) //intpara 为形参。
{
    int localvar;    //localvar 为局部变量。
    static int localstavar;    //localstavar 为局部静态变量。
    return;
}
```

类的 static 变量在什么时候初始化？函数的 static 变量在什么时候初始化？
答：**类的静态成员变量**在类实例化之前就已经存在了，并且分配了内存，**属于类，不属于任何对象**，而**非静态类成员变量则是属于对象所有的**。**类静态数据成员只有一个拷贝，为所有此类的对象所共享**。 **类静态成员函数属于整个类**，不属于某个对象，由该类所有对象共享。
函数的 static 变量在执行此函数时进行初始化，程序运行期间只初始化一次。
1、static 成员变量实现了 **同类对象间信息共享**。
2、static 成员在类外存储(在静态区)，**求类大小，并不包含在内。**
3、static 成员只能类外初始化。
1、静态成员函数的意义，不在于信息共享，数据沟通，而在于管理静态数据成员，完成对静态数据成员的封装。
2、静态成员函数只能访问静态数据成员。原因：非静态成员函数，在调用时 this 指针时被当作参数传进。而**静态成员函数属于类，而不属于对象**，没有 this 指针。
3、**在所有函数体外定义的 static 变量表示在该文件中有效**，不能 extern 到别的文件用，在函数体内定义的 static 表示只在该函数体内有效。另外，函数中的"adgfdf"这样的字符串存放在常量区。

2．将“引用”作为函数返回值类型的格式、好处和需要遵守的规则?

格式：类型标识符 &函数名（形参列表及类型说明）{ //函数体 }
好处：在内存中不产生被返回值的副本;（注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生 runtime error!
注意事项：
1） 不能返回局部变量的引用。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了”无所指”的引用，程序会进入未知状态。
2） 不能返回函数内部 new 分配的内存的引用。虽然不存在局部变量的被动销毁问题，

可对于这种情况(返回函数内部 new 分配内存的引用),又面临其它尴尬局面。例如,被函数返回的引用只是作为一个临时变量出现,而没有被赋予一个实际的变量,那么这个引用所指向的空间(由 new 分配)就无法释放,造成 memory leak。

3) 可以返回类成员的引用,但最好是 const。

4) 流操作符重载返回值申明为“引用”的作用:

5) 在另外的一些操作符中,却千万不能返回引用:+-*/ 四则运算符。它们不能返回引用。

2 extern “C”的惯用法

1) 在**C++中引用 C 语言中的函数和变量**,在**包含 C 语言头文件**(假设为 cExample.h)时,需进行下列处理:

```
extern “C”
{
#include “cExample.h” }
```

在 C 语言的头文件中,对其外部函数只能指定为 extern 类型,C 语言中不支持 extern “C”声明,在.c 文件中包含了 extern “C”时会出现编译语法错误。

C++引用 C 函数例子工程中包含的三个文件的源代码如下:

```
/* c 语言头文件:cExample.h */
#ifndef C_EXAMPLE_H
#define C_EXAMPLE_H
extern int add(int x,int y);
#endif

/* c 语言实现文件:cExample.c */
#include “cExample.h”
int add( int x, int y )
{  return x + y;  }
```

```
// c++实现文件,调用 add,包含 cExample.h
文件:cppFile.cpp
extern “C”
{   #include “cExample.h”   }
int main(int argc, char* argv[])
{
add(2,3);
return 0;   }
```

如果 C++调用一个 C 语言编写的.DLL 动态链接库时,当包括.DLL 的头文件或声明接口函数时,应加 extern “C” { }。

2) 在 C 中引用 C++语言中的函数和变量时,C++的头文件需添加 extern “C”,但是在 C 语言中不能直接引用声明了 extern “C”的该头文件,应该仅在 C 文件中将 C++中定义的 extern “C”函数声明为 extern 类型。

```
//C++头文件 cppExample.h
#ifndef CPP_EXAMPLE_H
#define CPP_EXAMPLE_H
extern “C” int add( int x, int y );
#endif

//C++实现文件 cppExample.cpp
#include “cppExample.h”
int add( int x, int y )
{  return x + y;  }
```

```
/* C 实现文件 cFile.c
/* 这样会编译出错:#include “cExample.h” */
extern int add( int x, int y );// 不能不能直接引用声明 extern “C”的该头文件
int main( int argc, char* argv[] )
{
add( 2, 3 );
return 0;
```

3. 重载（overload)和重写(overwried，有的书也叫做“覆盖”）的区别?

重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。
重写：是指子类 重新定义 基类虚函数(virtual) 的方法。
从实现原理上来说：
重载：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：int add(int t, int d);和 double add(double t, double d);。那么编译器做过修饰后的函数名称可能是这样的：int_add_int_int、double_add_double_double。对于这两个函数的调用，在编译器间就已经确定了，是静态的。也就是说，它们的地址在编译期就绑定了（早绑定），**因此，重载和多态无关！**
重写：和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚绑定）。

4. 面向对象的三个基本特征，并简单叙述之?

1. 封装：将客观事物**抽象成类**，每个类对自身的 **数据和方法** 实行保护 protection (私有类型 private, 保护类型 protected, 公有类型 public)
2. 继承：广义的继承有三种实现形式：**实现继承**（指使用基类的属性和方法而无需额外编码的能力）、**可视继承**（子窗体使用父窗体的**外观和实现代码**）、**接口继承**（仅使用属性和方法，**实现**滞后到子类实现）。前两种（类继承）和后一种（对象组合=>接口继承以及纯虚函数）构成了功能复用的两种方式。
3. 多态：是将 **父对象** 设置成为 和 一个或更多的他的 **子对象** 相等的技术，赋值之后，**父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。**简单的说，就是一句话：允许将 **子类类型的指针 赋值给 基类类型的指针**。

5、单向链表的反转

比如一个链表是这样的： 1->2->3->4->5 通过反转后成为 5->4->3->2->1。

1. 最容易想到的方法遍历一遍链表，利用一个辅助指针，存储遍历过程中当前指针指向的下一个元素，然后将当前节点元素的指针反转后，利用已经存储的指针往后面继续遍历。源代码如下：

```
struct linka {
int data;
linka* next;
};
void reverse(linka*& head) {
  if(head ==NULL)     return;
  linka *pre, *cur, *ne;
  pre = head;
  cur = head->next;
  while(cur){
    ne = cur->next;
    cur->next = pre;
    pre = cur;
    cur = ne;
  }
  head->next = NULL;
  head = pre;
}
```

6. 类实例

```
class String
{
public:
String(const char *str = NULL); // 通用构造函数　默认构造函数
String(const String &another); // 拷贝构造函数　拷贝初始化　String S3(S2);
~ String(); // 析构函数
String & operater =(const String &rhs); // 赋值函数　　String S3; S3 = S2;
private:
char *m_data; // 用于保存字符串
};
```

7. STL 库用过吗？常见的 STL 容器有哪些？算法用过哪几个？

在 STL 中，容器分为两类：序列式容器和关联式容器。

序列式容器，其中的元素不一定有序，但都可以被排序。
如：vector、list、deque（双端队列）、stack、queue、heap、priority_queue、slist；

关联式容器，内部结构基本上是一颗平衡二叉树。所谓关联，指每个元素都有一个键值和一个实值，元素按照一定的规则存放。如：RB-tree、set、map、multiset、multimap、hashtable、hash_set、hash_map、hash_multiset、hash_multimap。
关联式容器又分为 set(集合)和 map(映射表)两大类，以及这两大类的衍生体 multiset(多键集合)和 multimap(多键映射表)，这些容器均以 RB-tree（red-black tree，红黑树）完成。

此外，还有第 3 类关联式容器，如 hashtable(散列表)，以及以 hashtable 为底层机制完成的 hash_set(散列集合)/hash_map(散列映射表)/hash_multiset(散列多键集合)/hash_multimap(散列多键映射表)。
也就是说，set/map/multiset/multimap 都内含一个 **RB-tree（red-black tree，红黑树）**，而 hash_set/hash_map/hash_multiset/hash_multimap 都内含一个 **hashtable(散列表)**。
所谓关联式容器，类似关联式数据库，每笔数据或每个元素都有一个键值(key)和一个实值(value)，即所谓的 Key-Value(键-值对)。当元素被插入到关联式容器中时，**容器内部结构(RB-tree/hashtable)便依照其键值大小**，以某种特定规则将这个元素放置于适当位置。

所以，综上，说白了，什么样的结构决定其什么样的性质，
因为 set/map/multiset/multimap 都是基于 RB-tree 之上，所以有自动排序功能，
而 hashset/hash_map/hash_multiset/hash_multimap 都是基于 hashtable 之上，所以不含有自动排序功能，至于加个前缀 multi 无非就是允许键值重复而已。

8. 海量数据处理-分而治之

对于海量数据而言，由于无法一次性装进内存处理，导致我们不得不把海量的数据通过 **hash 映射** 分割成相应的 **小块数据**，然后再针对各个小块数据通过 **hash_map** 进行统计或其它操作，在**通过 堆排序 或者 快速排序** 选取 TOP K。
那什么是**hash 映射** 呢？简单来说，就是为了便于计算机在有限的内存中处理 big 数据，我们通过一种 **映射散列** 的方式让数据均匀分布在对应的内存位置(如**大数据通过取余的方式映射成小数存放在内存中**，或**大文件映射成多个小文件**)，而这个映射散列方式便是我们通常所说的 **hash 函数**，设计的好的 hash 函数能让数据均匀分布而减少冲突。

9. simhash 算法

simhash 作为 locality sensitive hash（局部敏感哈希）的一种：
其主要思想是 **降维**，将 **高维的特征向量映射成低维的特征向量** ，通过两个向量的 Hamming Distance 来确定文章是否重复或者高度近似。

文章分词—>得到特征向量—>哈希函数→得到 hash 值→加权→合并→降维→汉明距离

其中，**Hamming Distance，又称汉明距离**，在信息论中，两个等长字符串之间的汉明距离是两个字符串对应位置的不同字符的个数。也就是说，它就是将一个字符串变换成另外一个字符串所需要替换的字符个数。例如：1011101 与 1001001 之间的汉明距离是 2。
至于我们常说的字符串编辑距离则是一般形式的汉明距离。
如此，通过比较多个文档的 simHash 值的海明距离，可以获取它们的相似度。