程序员面霸手册

VER 1.2

黄优 2009.5

昨夜寒蛩不住鸣

惊回千里梦

已三更

起来独自绕阶行

人悄悄

帘外月胧明

白首为功名

旧山松竹老

阻归程

欲将心事付瑶琴

知音少

弦断有谁听?

岳飞

小重山

前　言

本人计算机专业毕业，找工作厉尽艰辛，面尽无数公司，深感怀才不遇，整理前人心血，以成此书。为了让后人少走我的路，把自己遇到的一些问题，以及网上一些朋友的程序员面试笔试题以及一些在面试笔试中的重要知识点都写出来同大家分享。其中有些题目已经做了答案，而有些答案在后边的知识点里已经注明，要读者自行寻找。把这些题目和知识点整理成本册子，欢迎大家来信推荐一些在自己面试笔试中遇到的问题，以完善此书，不胜感激。此书闲来所做，只期望能对后来者起到一定帮助。此电子版，免费无限分发，请勿用于商业目的，违者必究。如果你觉得好的话，将他分享给更多的朋友朋友。

书中难免出现错误和不足，还望不吝赐教，如果你在笔试或者面试中遇到了此册中没有遇到的问题，可以发电子邮件或者在本书官网给我留言，希望能把大家遇到的问题收集在一起，不胜感激。希望大家一起交流，大家帮大家，大家都能找到一个自己喜欢和满意的工作。我的电子邮件：**dotfly@hotmail.com,QQ:1140603739**,本书永久性官网：**http：//mian8king.bokee.com**处获取本书所有信息。

2009.3 黄优于西安

版权声明

版本更新说明

版本	更新内容	备注
v1.0(09.03)	无	创建
v1.1(09.04)	1. 增加了操作系统部分。 2. 数据结构部分增加了 ADT 的实现代码。 3. 数据结构部分增加了 KMP 排序算法。 4. 增加附录一 ASCII 码表。 5. 增加了附录二 ADO.NET 连接字符串。 6. 增加了附录二 ADO.NET 连接字符串。 7. 第四部分增加使用.NET 访问 MySql 数据库。 8. java 有几道题目做了答案。 9. 数据库部分新增几道华为公司问答题目。 10. 修正一些小的疏忽和错误。	更新及维护
v1.2	1. 对知识点精华部分做了索引,方便信息查阅,并且对原来档案做了简化,以保持本书的简单易读性。 2. 整理内容，对内容进行了详细分类，并修改了分类，使框架简明易懂。	更改

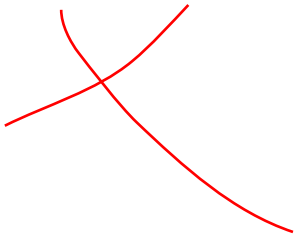
目　录

第一部分 数据结构

A. 笔试面试题集

1. 在一个单链表中 p 所指结点之前插入一个 s (值为 e)所指结点时，可执行如下操作：

```
q=head;
while (q->next!=p)
{
    q=q->next;
    s= new  Node;
    s->data=e;
    q->next=__s__;
    s->next=__p__ ;
}
```



2. 线性表的顺序存储结构是一种_____的存储结构，而链式存储结构是一种_____的存储结构。

   A．随机存取　　B．索引存取　　C．顺序存取　　D．散列存取

3. 线性表若采用链式存储结构时，要求内存中可用存储单元的地址__D__。

   A. 必须是连续的　　B. 部分地址必须是连续的

   C. 一定是不连续的　　D. 连续或不连续都可以

4. 在一个单链表中，已知 q 所指结点是 p 所指结点的前驱结点，若在 q 和 p 之间插入 s 结点，则执行____。

   A. s->next=p->next; p->next=s;　　B. p->next=s->next; s->next=p;

   C. q->next=s; s->next=p;　　D. p->next=s; s->next=q;

5. 在一个单链表中，若 p 所指结点不是最后结点，在 p 之后插入 s 所指结点，则执行_____ 。

   A. s->next=p; p->next=s;　　B. s->next=p->next; p->next=s;

   C. s->next=p->next; p=s;　　C. p->next=s; s->next=p;

6. 在一个单链表中，若删除 p 所指结点的后续结点，则执行_____。

   A. p->next= p->next->next;　　B. p= p->next; p->next= p->next->next;

   C. p->next= p->next;　　D. p= p->next->next;

7. 链表不具备的特点是__A__ 。

   A 可随机访问任何一个元素　　B 插入、删除操作不需要移动元素

   C 无需事先估计存储空间大小　　D 所需存储空间与线性表长度成正比

   注:链表是顺序结构,必须顺序访问。

8. 以下关于线性表的说法不正确的是_____。
A 线性表中的数据元素可以是数字、字符、记录等不同类型。
B 线性表中包含的数据元素个数不是任意的。
C 线性表中的每个结点都有且只有一个直接前趋和直接后继。
D 存在这样的线性表：表中各结点都没有直接前趋和直接后继。

9. 在一个长度为 n 的顺序表中删除第 i 个元素，要移动 n-i 个元素。如果要在第 i 个元素前插入一个元素，要后移 n-i+1 个元素。

10. 栈操作数据的原则是 后进先出，队列操作数据的原则是 先进先出。

11. 在栈中，可进行插入和删除操作的一端称 栈顶 。

12. 栈和队列都是_非线性_结构；对于栈只能在 栈顶 插入和删除元素；对于队列只能在_队头_插入元素和_____队尾_____删除元素。

13. 结构通常采用的两种存储结构是_____和_____。

14. 计算机在运行递归程序时，要用到 编译器 提供的栈。

15. 一个栈的入栈序列 a，b，c，d，e，则栈的不可能的输出序列是 C 。
A. edcba B. decba C. dceab D. abcde
注:此题的难点在于不必考虑完全入完后再出栈,可以边入边出,也可以入几个再出。

16. 一个队列的数据入列序列是 1，2，3，4，则队列的出队时输出序列是 B 。
A. 4，3，2，1 B. 1，2，3，4 C. 1，4，3，2 D. 3，2，4，1

17. 判断一个表达式中左右括号是否匹配，采用 ____ 实现较为方便。
A 线性表的顺序存储 B 队列 C 线性表的链式存储 D 栈

18. 栈与一般线性表区别主要在方面 C 。
A 元素个数 B 元素类型 C 逻辑结构 D 插入、删除元素的位置

19. “假上溢”现象会出现在_____中。
A 循环队列 B 队列 C 链队列 D 顺序队列

20. 在一个链队中，假设 F 和 R 分别是队首和队尾指针，则删除一个结点的运算是_____。
A R=F->next; B R=R->next; C F=F->next; D F=R->next;

21. 表达式 a*(b+c)-d 的后缀表达式是 B 。
A. abcd*+- B. abc+*d- C. abc*+d- D. -+*abcd
注:->a(b+c)*d- ->abc*+d-

22. 判断链表是否存在环型链表问题：判断一个链表是否存在环，例如下面这个链表就存在一个环：

例如 N1->N2->N3->N4->N5->N2 就是一个有环的链表，环的开始结点是 N5 这里有一个比较简单的解法。设置两个指针 p1，p2。每次循环 p1 向前走一步，p2 向前走两步。直到 p2 碰到 NULL 指针或者两个指针相等结束循环。如果两个指针相等则说明存在环。

```
struct link
{
    int data;
    link* next;
};

bool IsLoop(link* head)
{
   link* p1=head, *p2 = head;
    if (head ==NULL || head->next ==NULL)  //该链表没有节点或者只有一个节点,则不用判断.
    {
            return  false;
    }
   //先循环,再判断
   do{
       p1= p1->next;                  //p1 前进一步
       p2 = p2->next->next;           //p2 前进一步
   } while(p2 && p2->next && p1!=p2);    //  p2 不为空,p2 的下个节点不为空,p1 不等于 p2 继续循环,否则退出。
   // p2 为空则 p1p2 不相等则不存在环,p1=p2 则存在环
    if(p1 == p2)
            return true;
    else
            return false;
}
```

23. 链表反转 单向链表的反转是一个经常被问到的一个面试题，也是一个非常基础的问题。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。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)  //链表为空
        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;
}
```

还有一种利用递归的方法。这种方法的基本思想是在反转当前节点之前先调用递归函数反转后续节点。源代码如下。不过这个方法有一个缺点，就是在反转后的最后一个结点会形成一个环，所以必须将函数的返回的节点的 next 域置为 NULL。因为要 改变 head 指针，所以我用了引用。算法的源代码如下：

```
linka* reverse(linka* p,linka*& head)
{
    if(p == NULL || p->next == NULL)
    {
        head=p;
        return p;
    }
    else
    {
        linka* tmp = reverse(p->next,head);
        tmp->next = p;
        return p;
    }
}
```

24. 判断两个数组中是否存在相同的数字 给定两个排好序的数组，怎样高效得判断这两个数组中存在相同的数字？

这个问题首先最直接能想到的是一个 O(nlogn)的算法。就是任意挑选一个数组，遍历这个数组的所有元素，遍历过程中，在另一个数组中对第一个数组中的每个元素进行 binary search。用 C++实现代码如下：

```
bool findcommon(int a[],int size1,int b[],int size2)
{
    int i;
    for(i=0;i<size1;i++)
    {
```

```
            int start=0,end=size2-1,mid;
            while(start<=end)
            {
                mid=(start+end)/2;
                if(a[i]==b[mid])
                    return true;
                else if (a[i]<b[mid])
                    end=mid-1;
                else
                    start=mid+1;
            }
        }
        return false;
    }
```

有一个 O(n)算法。首先设两个下标，分别初始化为两个数组的起始地址，依次向前推进。推进的规则是比较两个数组中的数字，小的那个数组的下标向前推进一步， 直到任何一个数组的下标到达数组末尾时，如果这时还没碰到相同的数字，说明数组 中没有相同的数字。

```
    bool findcommon2(int a[], int size1, int b[], int size2)
    {
        int i=0,j=0;
        while(i<size1 && j<size2)
        {
            if(a[i]==b[j])
                    return true;
            if(a[i]>b[j])
                j++;
            if(a[i]<b[j])
                i++;
        }
        return false;
    }
```

25. 最大子序列问题：

给定一整数序列 A1， A2，... An （可能有负数），求 A1~An 的一个子序列 Ai~Aj，使得 Ai 到 Aj 的和最大

例如：

整数序列-2, 11, -4, 13, -5, 2, -5, -3, 12, -9 的最大子序列的和为 21。

对于这个问题，最简单也是最容易想到的那就是穷举所有子序列的方法。利用三重循环，依次求出所有子序列的和然后取最大的那个。当然算法复杂度会达到 O(n^3)。显然这种方法不是最优的，下面给出一个算法复杂度为 O(n)的线性算法实现，算法的来源于 Programming Pearls 一书。 在给出线性算法之前，先来看一个对穷举算法进行优化的算法，它的算法复杂度为 O(n^2)。其实这个算法只是对对穷举算法稍微做了一些修改：其实子序列的和我们并不需要每次都重新计算一遍。假设 Sum(i, j)是 A[i] ... A[j]

的和，那么 Sum(i, j+1) = Sum(i, j) + A[j+1]。利用这一个递推，我们就可以得到 下面这个算法：

```
int max_sub(int a[],int size)
{
    int i,j,v,max=a[0];
    for(i=0;i<size;i++)
    {
         v=0;
         for(j=i;j<size;j++)
         {
              v=v+a[j];//Sum(i, j+1) = Sum(i, j) + A[j+1]
              if(v>max)
                   max=v;
         }
    }
    return max;
}
```

那怎样才能达到线性复杂度呢？这里运用动态规划的思想。先看一下源代码实现：

```
int max_sub2(int a[], int size)
{
    int i,max=0,temp_sum=0;
    for(i=0;i<size;i++)
    {
         temp_sum+=a[i];
         if(temp_sum>max)
              max=temp_sum;
         else if(temp_sum<0)
              temp_sum=0;
    }
    return max;
}
```

在这一遍扫描数组当中，从左到右记录当前子序列的和 temp_sum，若这个和不断增加，那么最大子序列的和 max 也不断增加(不断更新 max)。如果往前扫描中遇到负数，那么当前子序列的和将会减小。此时 temp_sum 将会小于 max，当然 max 也就不更新。如果 temp_sum 降到 0 时，说明前面已经扫描的那一段就可以抛弃了，这时将 temp_sum 置为 0。然后，temp_sum 将从后面开始将这个子段进行分析，若有比当前 max 大的子段，继续更新 max。这样一趟扫描结果也就出来了。

26. 找出单向链表的中间结点

这道题和解判断链表是否存在环，我用的是非常类似的方法，只不过结束循环的条件和函数返回值不一样罢了。设置两个指针 p1，p2。每次循环 p1 向前走一步，p2 向前走两步。当 p2 到达链表的末尾时，p1 指向的时链表的中间。

```
link* mid(link* head)
{
```

```
        link* p1,*p2;
        p1=p2=head;
        if(head==NULL || head->next==NULL)
                return head;
        do {
                p1=p1->next;
                p2=p2->next->next;
        } while(p2 && p2->next);
        return p1;
}
```

27. 已知一个单向链表的头，请写出删除其某一个结点的算法，要求，先找到此结点，然后删除。

```
slnodetype *Delete(slnodetype *Head,int key){}中 if(Head->number==key)
{
    Head=Pointer->next;
    free(Pointer);
    break;
}
Back = Pointer;
Pointer=Pointer->next;
if(Pointer->number==key)
{
    Back->next=Pointer->next;
    free(Pointer);
    break;
}
void delete(Node* p)
{
    if(Head = Node)
    while(p)
}
```

28. 判断一个字符串是不是回文

```
int IsReverseStr(char *aStr)
{
    int i,j;
    int found=1;
    if(aStr==NULL)
        return -1;
    j=strlen(aStr);
    for(i=0;i<j/2;i++)
        if(*(aStr+i)!=*(aStr+j-i-1))
        {
            found=0;
            break;
```

```
        }
    return found;
}
```

29. 用两个栈实现一个队列的功能？要求给出算法和思路！

设 2 个栈为 A,B,一开始均为空

入队:

将新元素 push 入栈 A;

出队:

(1)判断栈 B 是否为空;

(2)如果不为空，则将栈 A 中所有元素依次 pop 出并 push 到栈 B;

(3)将栈 B 的栈顶元素 pop 出

30. Josephu 问题为：设编号为 1，2，… n 的 n 个人围坐一圈，约定编号为 k（1<=k<=n）的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

数组实现：

```
#include <stdio.h>
#include <malloc.h>
int Josephu(int n, int m)
{
    int flag, i, j = 0;
    int *arr = (int *)malloc(n * sizeof(int));
    for (i = 0; i < n; ++i)
        arr[i] = 1;
    for (i = 1; i < n; ++i)
    {
        flag = 0;
        while (flag < m)
        {
            if (j == n)
                j = 0;
            if (arr[j])
                ++flag;
            ++j;
        }
        arr[j - 1] = 0;
        printf("第%4d 个出局的人是：%4d 号\n", i, j);
    }
    free(arr);
    return j;
}
int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
```

```
    printf("最后胜利的是%d 号！\n", Josephu(n, m));
    system("pause");
    return 0;
}
```

链表实现：

```
#include <stdio.h>
#include <malloc.h>
typedef struct Node
{
    int index;
    struct Node *next;
}JosephuNode;
int Josephu(int n, int m)
{
    int i, j;
    JosephuNode *head, *tail;
    head = tail = (JosephuNode *)malloc(sizeof(JosephuNode));
    for (i = 1; i < n; ++i)
    {
        tail->index = i;
        tail->next = (JosephuNode *)malloc(sizeof(JosephuNode));
        tail = tail->next;
    }
    tail->index = i;
    tail->next = head;
    for (i = 1; tail != head; ++i)
    {
        for (j = 1; j < m; ++j)
        {
            tail = head;
            head = head->next;
        }
        tail->next = head->next;
        printf("第%4d 个出局的人是：%4d 号\n", i, head->index);
        free(head);
        head = tail->next;
    }
    i = head->index;
    free(head);
    return i;
}
int main()
{
    int n, m;
```

```
    scanf("%d%d", &n, &m);
    printf("最后胜利的是%d 号！\n", Josephu(n, m));
    system("pause");
    return 0;
}
```

31. 排序二叉树插入一个节点或双向链表的实现

32. 问答题：

1) 什么是平衡二叉树？

左右子树都是平衡二叉树 且左右子树的深度差值的绝对值不大于 1

2) 堆栈溢出一般是由什么原因导致的？

没有回收垃圾资源

3) 数组和链表的区别

数组：数据顺序存储，固定大小

链表：数据可以随机存储，大小可动态改变

4) 冒泡排序算法的时间复杂度是什么？

O(n^2)

B. 知识点精华

1. big endian,little endian

big endian 是指低地址存放最高有效字节（MSB），而 little endian 则是低地址存放最低有效字节（LSB）。
比如数字 0x12345678 在两种不同字节序 CPU 中的存储顺序如下所示：

```
Big Endian
低地址 高地址
------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 12 | 34 | 56 | 78 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Little Endian
低地址 高地址
------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 78 | 56 | 34 | 12 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

CPU 存储一个字节的数据时其字节内的 8 个比特之间的顺序是否也有 big endian 和 little endian 之分？或者说是否有比特序的不同？
实际上，这个比特序是同样存在的。下面以数字 0xB4（10110100）用图加以说明。

```
Big Endian
msb lsb
------------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Little Endian
lsb msb
------------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

2. ADT(抽象数据类型 C 语言实现)

```
#define ERROR 0
```

```
#define OK 1
struct STU
{
    char name[20];
    char stuno[10];
    int age; int score;
}stu[50];
struct LIST
{
    struct STU stu[50];
    int length;
}L;
int printlist(struct LIST L)
{
    int i;
    printf("name stuno age score\n");
    for(i=0;i<L.length;i++)
    printf("%s  %s\t%d\t%d\n",  L.stu[i].name,  L.stu[i].stuno,  L.stu[i].age,
    L.stu[i].score);
    printf("\n");
}
int listinsert(struct LIST *L,int i,struct STU e)
{
    struct STU *p,*q;
    if (i<1||i>L->length+1)
        return ERROR;
    q=&(L->stu[i-1]);
    for(p=&L->stu[L->length-1];p>=q;--p)
        *(p+1)=*p; *q=e; ++L->length;
    return OK;
}/*ListInsert Before i */
main()
{
    struct STU e;
    L.length=0;
    strcpy(e.name,"zmofun");
    strcpy(e.stuno,"100001");
    e.age=80;
    e.score=1000;
    listinsert(&L,1,e);
    printlist(L);
    printf("List length now is %d.\n\n",L.length);
    strcpy(e.name,"bobjin");
    strcpy(e.stuno,"100002");
```

```
    e.age=80;
    e.score=1000;
    listinsert(&L,1,e);
    printlist(L);
    printf("List length now is %d.\n\n",L.length);
}
```

3. KMP 排序算法 C 语言源代码

```
#include <stdio.h>
#include <string.h>
int index_KMP(char *s,char *t,int pos);
void get_next(char *t,int *next);

char s[10]="abcacbcba";
char t[4]="bca";
int next[4];
int pos=0;

int main()
{
    int n;
    get_next(t,next);
    n=index_KMP(s,t,pos);
    printf("%d",n);
    return 0;
}

int index_KMP(char *s,char *t,int pos)
{
    int i=pos,j=1;
    while (i<=(int)strlen(s)&&j<=(int)strlen(t))
    {
        if (j==0||s[i]==t[j-1])
        {
            i++;
            j++;
        }
        else j=next[j];
    }
    if (j>(int)strlen(t))
        return i-strlen(t)+1;
```

```
    else
        return 0;
}

void get_next(char *t,int *next)
{

    int i=1,j=0;
    next[0]=next[1]=0;
    while (i<(int)strlen(t))
    {
        if (j==0||t[i]==t[j])
        {
            i++;
            j++;
            next[i]=j;
        }
        else j=next[j];
    }

}
```

4. 堆与栈

```
    void f()
    {
         int*  p=new int[5];
     }
```

这条短短的一句话就包含了堆与栈，看到 new，我们首先就应该想到，我们分配了一块堆内存，那么指针 p 呢？他分配的是一块栈内存，所以这句话的意思就是：在栈内存中存放了一个指向一块堆内存的指针 p。在程序会先确定在堆中分配内存的大小，然后调用 new()分配内存，然后返回这块内存的首地址，放入栈中。

堆和栈的主要区别：

1) 管理方式：

栈由编译器自动管理，无需手工控制；堆由程序员控制，如果不加以干预会造成内存泄露。

2) 空间大小：

在 32 位系统下，堆内存可以达到 4G 的空间，也就是说堆可以使用几乎所有内存空间。但栈都是有一定的空间大小的，一般都可以在编译器中进行设置。

3) 碎片问题：

对堆的 new/delete 操作会造成内存空间的不连续，从而造成大量的内存碎片，使程序效率降低。栈则不会存在这个问题，因为栈是先进后出的，不可能有一个内存块从栈中间弹出，因而不会造成碎片。

4） 生长方向：

堆是向着内存地址增加的方向；栈向着内存地址减小的方向增长。

5） 分配方式：

堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 alloca 函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需手工操作。

6） 分配效率：

栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 C/C++函数库提供的，它的机制是很复杂的，堆的效率比栈要低得多。

无论是堆还是栈，都要防止越界现象的发生），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果。

5. 约瑟夫环

用户输入 M,N 值，从 1 至 N 开始顺序循环数数，每数到 M 输出该数值，直至最后一个元素并输出该元素的值。写出 C 程序。

其实现代码如下（在 READHAT9.0 LINUX 下测试通过）：

```
/*********************************************
*文件名：game_ouputnum.h
*文件描述：给定 M,N 值，从 1 至 N 开始顺序循环数数，每数到 M，把该 M 去掉，继续直到最后一个元数并输出该无数
            使用循环链表实现.
**********************************************/
#include <stdio.h>
#include <stdlib.h>

/*    N 值        */
#define NUM 8
#define OK 1
#define ERROR -1
/* M   值 */
#define NEXT 3
typedef struct
{
    int data;
    struct list *next;
}list;
/*******************************************************
*函数名：list *createList()
*参   数：无
*功能描述：创建一个循环链表
*返回值：成功返回链表的首指针，失败返回-1
*******************************************************/
```

```
list *createList()
{
        list *head=NULL;
        list *tail=NULL,*temp=NULL;
        int i=0;
        head=(struct list *)malloc(sizeof(list));
if( head == NULL )
{
        printf("malloc space failed!\n");
    return ERROR;
}
head->data=1;
head->next=head;
                tail = head;
for( i=2;i<=NUM; i++)
{
       temp=(struct list *)malloc(sizeof(list));
       if( temp == NULL )
       {
          printf("malloc space failed!\n");
    return ERROR;
       }
       temp->data=i;
       temp->next=head;
       tail->next = temp;
       tail = temp;
}
return head;
}
/*********************************************************
*函数名：void printList(list *head)
*参　数：链表的首地址
*功能描述：打印出链表中所有的数据
*********************************************************/
void printList(list *head)
{
       list *pList=NULL;
       pList = head;
       if( head == NULL)
{
       printf( "printList param invalid!\n" );
}
while(head->next!=pList)
{
```

```
        printf("this data is :%d\n", head->data);
        head=head->next;
}
              printf("the data is :%d\n",head->data);
return;
}
/*********************************************************
*函数名：int getLastElem(list *pList)
*参    数：链表的首地址
*功能描述：开始顺序循环数数，得到最后一个元素的值.
*返回值：正确返回最后一个元素的值，错误返回－1
*********************************************************/
int getLastElem(list *pList)
{
list *head=NULL,*temp=NULL;
int i = 1;
if(pList == NULL)
{
         printf("getLastElem param invalid\n");
     return ERROR;
}
head=temp=pList;
while(head!=head->next)
{
          if(i==NEXT)
     {
           temp=head;
       head=head->next;
                    free(temp);
                    pList->next=head;
       temp=NULL;
       i=1;
     }
     else
     {
                    pList=head;
           head=head->next;
       i++;
     }
   }
        printf("get lastElem:%d\n", head->data);
return head->data;
}
int main()
```

```
{
      list *pList=NULL;
      pList=createList();
if(pList==NULL)
{
    printf("create list error!\n");
    exit(0);
}
printList(pList);
      printf("the last elem:%d\n",getLastElem(pList));
}
```

6. 快速排序

```
#include<iostream>
usingnamespacestd;
intPartition (int*L,intlow,int high)
{
    inttemp = L[low];
    intpt = L[low];
while (low < high)
{
    while (low < high && L[high] >= pt)
        --high;
    L[low] = L[high];
    while (low < high && L[low] <= pt)
        ++low;
    L[low] = temp;
}
L[low] = temp;
returnlow;
}
voidQSort (int*L,intlow,int high)
{
if (low < high)
{
intpl = Partition (L,low,high);
QSort (L,low,pl - 1);
QSort (L,pl + 1,high);
}
}
Int main ()
```

```
{
intnarry[100],addr[100];
intsum = 1,t;
cout << "Input number:" << endl;
cin >> t;
while (t != -1)
{
narry[sum] = t;
addr[sum - 1] = t;
sum++;
cin >> t;
}
sum -= 1;
QSort (narry,1,sum);
for (int i = 1; i <= sum;i++)
cout << narry[i] << '\t';
cout << endl;
intk;
cout << "Please input place you want:" << endl;
cin >> k;
intaa = 1;
intkk = 0;
for (;;)
{
if (aa == k)
break;
if (narry[kk] != narry[kk + 1])
{
aa += 1;
kk++;
}
}
cout << "The NO." << k << "number is:" << narry[sum - kk] << endl;
cout << "And it's place is:" ;
for (i = 0;i < sum;i++)
{
if (addr[i] == narry[sum - kk])
cout << i << '\t';
}
return0;
}
```

第二部分 C/C++

A. 笔试面试题集

1. 问答题

1) static 有什么用途？
1）限制变量的作用域
2）设置变量的存储域

2) 引用与指针有什么区别？
1）引用必须被初始化，指针不必。
2）引用初始化以后不能被改变，指针可以改变所指的对象。
3）不存在指向空值的引用，但是存在指向空值的指针。

3) 全局变量和局部变量在内存中是否有区别？如果有，是什么区别？
全局变量储存在静态数据库，局部变量在堆栈

4) 什么函数不能声明为虚函数？
Constructor

5) 写出 float x 与“零值”比较的 if 语句。
if(x>0.000001&&x<-0.000001)

6) 不能做 switch()的参数类型是：
switch 的参数不能为实型，还有字符串。

7) 局部变量能否和全局变量重名？
答：能，局部会屏蔽全局。要用全局变量，需要使用"::"
局部变量可以与全局变量同名，在函数内引用这个变量时，会用到同名的局部变量，而不会用到全局变量。对于有些编译器而言，在同一个函数内可以定义多个同名的局部变量，比如在两个循环体内都定义一个同名的局部变量，而那个局部变量的作用域就在那个循环体内

8) 如何引用一个已经定义过的全局变量？
答：extern
可以用引用头文件的方式，也可以用 extern 关键字，如果用引用头文件方式来引用某个在头文件中声明的全局变理，假定你将那个变写错了，那么在编译期间会报错，如果你用 extern 方式引用时，假定你犯了同样的错误，那么在编译期间不会报错，而在连接期间报错

9) 全局变量可不可以定义在可被多个.C 文件包含的头文件中？为什么？

答：可以，在不同的 C 文件中以 static 形式来声明同名全局变量。

可以在不同的 C 文件中声明同名的全局变量，前提是其中只能有一个 C 文件中对此变量赋初值，此时连接不会出错

10) 语句 for(; 1 ;)有什么问题？它是什么意思？

答：和 while(1)相同。永真

11) do……while 和 while……do 有什么区别？

答：前一个循环一遍再判断，后一个判断以后再循环

12) tatic 全局变量与普通的全局变量有什么区别？static 局部变量和普通局部变量有什么区别？static 函数与普通函数有什么区别？

全局变量前面加 static 就构成了静态的全局变量。全局变量本身就是静态存储方式， 静态全局变量当然也是静态存储方式。两者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该变量的源文件内有效。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用， 因此可以避免在其它源文件中引起错误。static 全局变 量只初使化一次，防止在其他文件单元中被引用;

static 局部变量只被初始化一次，下一次依据上一次结果值；

static 函数与普通函数作用域不同。仅在本文件件使用。static 函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝。

13) 程序的局部变量存在于 堆栈 中，全局变量存在于 静态区 中，动态申请数据存在于 堆 中。

14) 设有以下说明和定义：

```
typedef union
{
long i; int k[5]; char c;
} DATE;
struct data { int cat; DATE cow; double dog;} too;
DATE max;
```

则语句 printf("%d",sizeof(struct date)+sizeof(max));的执行结果是：_ 52 _

答：DATE 是一个 union, 变量公用空间. 里面最大的变量类型是 int[5], 占用 20 个字节. 所以它的大小是 20

data 是一个 struct, 每个变量分开占用空间. 依次为 int4 + DATE20 + double8 = 32.所以结果是 20 + 32 = 52.当然...在某些 16 位编辑器下, int 可能是 2 字节, 那么结果是 int2 + DATE10 + double8 =20

15) 以下代码中的输出语句输出 0 吗，为什么？

```
struct CLS
{
```

```
int m_i;
    CLS( int i ) : m_i(i) {}
    CLS()
    {
     CLS(0);
    }
};
CLS obj;
cout << obj.m_i << endl;
```

答：不能。在默认构造函数内部再调用带参的构造函数属用户行为而非编译器行为，亦即仅执行函数调用，而不会执行其后的初始化表达式。只有在生成对象时，初始化表达式才会随相应的构造函数一起调用。

16) 在 c 语言库函数中将一个字符转换成整型的函数是 atool()吗，这个函数的原型是什么？

参见第二部分（c/c++）知识点精华 8。

17) 对于一个频繁使用的短小函数,在 C 语言中应用什么实现,在 C++中应用什么实现?

c 用宏定义，c++可以用宏 也可以用 inline

18) 软件测试都有那些种类?

黑盒：针对系统功能的测试 白合：测试函数功能，各函数接口

19) unsigned char *p1;
unsigned long *p2;
p1=(unsigned char *)0x801000;
p2=(unsigned long *)0x810000;
请问 p1+5= ;
p2+5= ;

20) Windows 消息调度机制是:

A.指令队列;B.指令堆栈;C.消息队列;D.消息堆栈;

21) unsigned short hash(unsigned short key)
{
return (key>>)%256
}
请问 hash(16),hash(256)的值分别是:
A.1.16;B.8.32;C.4.16;D.1.32

22) 一个 32 位的机器,该机器的指针是多少位

指针是多少位只要看地址总线的位数就行了。80386 以后的机子都是 32 的数据总线。所以指针的位数就是 4 个字节了。

23) main()
```
{
int a[5]={1,2,3,4,5};
int *ptr=(int *)(&a+1);//这个是数组的偏移
printf("%d,%d",*(a+1),*(ptr-1));
}
```
输出：2,5
(a+1）就是 a[1]，(ptr-1)就是 a[4],执行结果是 2，5
&a+1 不是首地址+1，系统会认为加一个 a 数组的偏移，是偏移了一个数组的大小（本　　例是 5 个 int）
int *ptr=(int *)(&a+1);
则 ptr 实际是&(a[5]),也就是 a+5
原因如下：
&a 是数组指针，其类型为 int (*)[5];
而指针加 1 要根据指针类型加上一定的值，
不同类型的指针+1 之后增加的大小不同
a 是长度为 5 的 int 数组指针，所以要加 5*sizeof(int)
所以 ptr 实际是 a[5]
但是 prt 与(&a+1)类型是不一样的(这点很重要)
所以 prt-1 只会减去 sizeof(int*)
a,&a 的地址是一样的，但意思不一样，a 是数组首地址，也就是 a[0]的地址，&a 是对象（数组）首地址，a+1 是数组下一元素的地址，即 a[1],&a+1 是下一个对象的地址，即 a[5].

24) 写一个“标准”宏，这个宏输入两个参数并返回较小的一个。
```
#define Min(X, Y) ((X)>(Y)?(Y):(X)) //结尾没有;
```

25) 嵌入式系统中经常要用到无限循环，你怎么用 C 编写死循环。
while(1){}或者 for(;;)

26) 关键字 const 有什么含意？
表示常量不可以修改的变量。

27) 关键字 volatile 有什么含意？并举出三个不同的例子？
提示编译器对象的值可能在编译器未监测到的情况下改变。

28) 有以下表达式：
int a=248; b=4;int const c=21;const int *d=&a;
int *const e=&b;int const *f const =&a;
请问下列表达式哪些会被编译器禁止？为什么？
*c=32;d=&b;*d=43;e=34;e=&a;f=0x321f;
*c 这是个什么东东，禁止
*d 说了是 const， 禁止
e = &a 说了是 const 禁止
const *f const =&a; 禁止

29) 交换两个变量的值，不使用第三个变量。即 a=3,b=5,交换之后 a=5,b=3;
有两种解法，一种用算术算法，一种用^(异或)

```
a = a + b;
b = a - b;
a = a - b;
```

or

```
a = a^b;// 只能对 int,char..
b = a^b;
a = a^b;
```

or

```
a ^= b ^= a;
```

30) c 和 c++中的 struct 有什么不同?
c 和 c++中 struct 的主要区别是 c 中的 struct 不可以含有成员函数，而 c++中的 struct 可以。c++中 struct 和 class 的主要区别在于默认的存取权限不同，struct 默认为 public，而 class 默认为 private

31) char szstr[10];
strcpy(szstr,"0123456789");
产生什么结果？为什么？
长度不一样，会造成非法的 OS

32) 纯虚函数如何定义？使用时应注意什么？
virtual void f()=0;
是接口，子类必须要实现

33) (void *)ptr 和 (*(void**))ptr 的结果是否相同？其中 ptr 为同一个指针
(void *)ptr 和 (*(void**))ptr 值是相同的

34) int main()

```
{
int x=3;
printf("%d",x);
return 1;
}
```

问函数既然不会被其它函数调用，为什么要返回 1?
mian 中，c 标准认为 0 表示成功，非 0 表示错误。具体的值是某中具体出错信息

35) 要对绝对地址 0x100000 赋值，我们可以用
(unsigned int*)0x100000 = 1234;
那么要是想让程序跳转到绝对地址是 0x100000 去执行，应该怎么做？
*((void (*)( ))0x100000) ( );
首先要将 0x100000 强制转换成函数指针,即:
(void (*)())0x100000

然后再调用它:
*((void (*)())0x100000)();
用 typedef 可以看得更直观些:
typedef void(*)() voidFuncPtr;
*((voidFuncPtr)0x100000)();

36) 已知一个数组 table，用一个宏定义，求出数据的元素个数
#define NTBL
#define NTBL (sizeof(table)/sizeof(table[0]))

37) 线程与进程的区别和联系? 线程是否具有相同的堆栈? dll 是否有独立的堆栈?
进程是死的，只是一些资源的集合，真正的程序执行都是线程来完成的，程序启动的时候操作系统就帮你创建了一个主线程。
每个线程有自己的堆栈。
dynamic link library
DLL 中有没有独立的堆栈，这个问题不好回答，或者说这个问题本身是否有问题。因为 DLL 中的代码是被某些线程所执行，只有线程拥有堆栈，如果 DLL 中的代码是 EXE 中的线程所调用,那么这个时候是不是说这个 DLL 没有自己独立的堆栈？如果 DLL 中的代码是由 DLL 自己创建的线程所执行，那么是不是说 DLL 有独立的堆栈?
以上讲的是堆栈，如果对于堆来说，每个 DLL 有自己的堆，所以如果是从 DLL 中动态分配的内存，最好是从 DLL 中删除，如果你从 DLL 中分配内存，然后在 EXE 中，或者另外一个 DLL 中删除，很有可能导致程序崩溃

38) unsigned short A = 10;
printf("~A = %u\n", ~A);// %u 是以无符号十进制输出

39) char c=128;
printf("c=%d\n",c);
输出多少？并分析过程
第一题，～A ＝0xffffff5,int 值 为－11，但输出的是 uint。所以输出 4294967285
第二题，c＝0x10,输出的是 int，最高位为 1，是负数，所以它的值就是 0x00 的补码就是 128，所以输出－128。
这两道题都是在考察二进制向 int 或 uint 转换时的最高位处理。

40) 给定结构 struct A
{
char t:4;
char k:4;
unsigned short i:8;
unsigned long m;
};问 sizeof(A) = ?
给定结构 struct A
{
char t:4; 4 位

```
char k:4; 4 位
unsigned short i:8; 8 位
unsigned long m; // 偏移 2 字节保证 4 字节对齐
}; // 共 8 字节
```

41) 下面的函数实现在一个数上加一个数，有什么错误？请改正。

```
int add_n ( int n )
{
static int i = 100;
i += n;
return i;
}
```

42) 求函数返回值，输入 x=9999;

```
int func ( x )
{
int countx = 0;
while ( x )
{
countx ++;
x = x&(x-1);
}
return countx;
}
```

知道了这是统计 9999 的二进制数值中有多少个 1 的函数，且有

9999＝9×1024＋512＋256＋15

9×1024 中含有 1 的个数为 2；

512 中含有 1 的个数为 1；

256 中含有 1 的个数为 1；

15 中含有 1 的个数为 4；

故共有 1 的个数为 8，结果为 8。

1000 - 1 = 0111，正好是原数取反。这就是原理。

用这种方法来求 1 的个数是很效率很高的。

不必去一个一个地移位。循环次数最少。

43) int a,b,c 请写函数实现 C=a+b ,不可以改变数据类型,如将 c 改为 long int,关键是如何处理溢出问题

```
bool add (int a, int b,int *c)
{
*c=a+b;
return (a>0 && b>0 &&(*c<a || *c<b) || (a<0 && b<0 &&(*c>a || *c>b)));
}
```

分析：

```
struct bit
```

```
{ int a:3;
int b:2;
int c:3;
};
int main()
{
bit s;
char *c=(char*)&s;
cout<<sizeof(bit)<<endl;
*c=0x99;
cout << s.a <<endl <<s.b<<endl<<s.c<<endl;
int a=-1;
printf("%x",a);
return 0;
}
```

输出为什么是

4

1

-1

-4

ffffffff

因为 0x99 在内存中表示为 100 11 001 , a = 001, b = 11, c = 100

当 c 为有符合数时, c = 100, 最高 1 为表示 c 为负数，负数在计算机用补码表示，所以 c = -4;同理

b = -1;

当 c 为有符合数时, c = 100,即 c = 4,同理 b = 3

2. 微软亚洲技术中心的面试题

1) 进程和线程的差别。
2) 测试方法
3) Heap 与 stack 的差别。
4) Windows 下的内存是如何管理的?
5) 介绍.Net 和.Net 的安全性。
6) 客户端如何访问.Net 组件实现 Web Service?
7) C/C++编译器中虚表是如何完成的?
8) 谈谈 COM 的线程模型。然后讨论进程内/外组件的差别。
9) 谈谈 IA32 下的分页机制
10) 给两个变量，如何找出一个带环单链表中是什么地方出现环的?
11) 在 IA32 中一共有多少种办法从用户态跳到内核态?
12) 如果只想让程序有一个实例运行，不能运行两个。像 winamp 一样，只能开一个窗口，怎样实现?
13) 何截取键盘的响应，让所有的‘a’变成‘b’?
14) Apartment 在 COM 中有什么用？为什么要引入？
15) 存储过程是什么？有什么用？有什么优点？
16) Template 有什么特点？什么时候用？

17) 谈谈 Windows DNA 结构的特点和优点。

3. 思科

1) 用宏定义写出 swap（x，y）

```
#define swap(x, y)\
x = x + y;\
y = x - y;\
x = x - y;
```

2) 数组 a[N]，存放了 1 至 N-1 个数，其中某个数重复一次。写一个函数，找出被重复的数字.时间复杂度必须为 o（N）函数原型：

```
int do_dup(int a[],int N)
//简单方法如下：
int do_dup(int a[], int N)
{
if (N <= 0)
return -1;
int nTemp = 1;
for (int i = 0; i < N; i++)
{
}
}
```

3) 一语句实现 x 是否为 2 的若干次幂的判断

```
int i = 512;
cout << boolalpha << ((i & (i - 1)) ? false : true) << endl;
```

4) unsigned int intvert(unsigned int x,int p,int n)实现对 x 的进行转换,p 为起始转化位,n 为需要转换的长度,假设起始点在右边.如 x=0b0001 0001,p=4,n=3 转换后 x=0b0110 0001

```
unsigned int intvert(unsigned int x,int p,int n){
unsigned int _t = 0;
unsigned int _a = 1;
for(int i = 0; i < n; ++i){
_t |= _a;
_a = _a << 1;
}
_t = _t << p;
x ^= _t;
return x;
}
```

4. 慧通：

1) 什么是预编译

2) 何时需要预编译：

总是使用不经常改动的大型代码体。

程序由多个模块组成，所有模块都使用一组标准的包含文件和相同的编译选项。在这种情况下，可以将所有包含文件预编译为一个预编译头。

3) char * const p;
char const * p
const char *p
上述三个有什么区别？

```
char * const p; //常量指针，p 的值不可以修改
char const * p; //指向常量的指针，指向的常量值不可以改
const char *p;  //和 char const *p
char str1[] = "abc";
char str2[] = "abc";
const char str3[] = "abc";
const char str4[] = "abc";
const char *str5 = "abc";
const char *str6 = "abc";
char *str7 = "abc";
char *str8 = "abc";
cout << ( str1 == str2 ) << endl;
cout << ( str3 == str4 ) << endl;
cout << ( str5 == str6 ) << endl;
cout << ( str7 == str8 ) << endl;
```

结果是：0 0 1 1
解答：str1,str2,str3,str4 是数组变量，它们有各自的内存空间；
而 str5,str6,str7,str8 是指针，它们指向相同的常量区域。

5. 改错：

```
#include <stdio.h>
int main(void) {
int **p;
int arr[100];
p = &arr;
return 0;
}
```

解答：
指针类型不同,
int **p; //二级指针
&arr; //得到的是指向第一维为 100 的数组的指针

```
#include <stdio.h>
int main(void) {
int **p, *q;
int arr[100];
q = arr;
p = &q;
```

```
return 0;
}
```

6. 下面这个程序执行后会有什么错误或者效果:

```
#define MAX 255
int main()
{
unsigned char A[MAX],i;//i 被定义为 unsigned char
for (i=0;i<=MAX;i++)
A[i]=i;
}
```

解答：死循环加数组越界访问（C/C++不进行数组越界检查）

MAX=255

数组 A 的下标范围为:0..MAX-1,这是其一..

其二.当 i 循环到 255 时,循环内执行:

A[255]=255;

这句本身没有问题..但是返回 for (i=0;i<=MAX;i++)语句时,

由于 unsigned char 的取值范围在(0..255),i++以后 i 又为 0 了..无限循环下去.

7.
```
struct name1{
char str; //为什么这个时候 str 不需要字节对齐呢
short x;
int num;
}

struct name2{
char str;
int num;
short x;
}
```

sizeof(struct name1)=8,sizeof(struct name2)=12

在第二个结构中，为保证 num 按四个字节对齐，char 后必须留出 3 字节的空间；同时为保证整个结构的自然对齐（这里是 4 字节对齐），在 x 后还要补齐 2 个字节，这样就 是 12 字节。

8. A.c 和B.c两个c文件中使用了两个相同名字的static变量,编译的时候会不会有问题?这两个 static 变量会保存到哪里（栈还是堆或者其他的）?

static 的全局变量，表明这个变量仅在本模块中有意义，不会影响其他模块。

他们都放在数据区，但是编译器对他们的命名是不同的。

如果要使变量在其他模块也有意义的话，需要使用 extern 关键字。

```
struct s1
{
int i: 8;
int j: 4;
```

```
int a: 3;
double b;
};
struct s2
{
int i: 8;
int j: 4;
double b;
int a:3;
};
printf("sizeof(s1)= %d\n", sizeof(s1));
printf("sizeof(s2)= %d\n", sizeof(s2));
```

result: 16, 24

第一个 struct s1

```
{
int i: 8;
int j: 4;
int a: 3;
double b;
};
```

理论上是这样的，首先是 i 在相对 0 的位置，占 8 位一个字节，然后，j 就在相对一个字节的位置，由于一个位置的字节数是 4 位的倍数，因此不用对齐，就放在那里了，然后是 a，要在 3 位的倍数关系的位置上，因此要移一位，在 15 位的位置上放下，目前总共是 18 位，折算过来是 2 字节 2 位的样子，由于 double 是 8 字节的，因此要在相对 0 要是 8 个字节的位置上放下，因此从 18 位开始到 8 个字节之间的位置被忽略，直接放在 8 字节的位置了，因此，总共是 16 字节。

第二个最后会对照是不是结构体内最大数据的倍数，不是的话，会补成是最大数据的倍数

9. 编程

1）读文件 file1.txt 的内容（例如）：

12

34

56

输出到 file2.txt：

56

34

12

（逆序）

2）输出和为一个给定整数的所有组合

例如 n=5

5=1+4；5=2+3（相加的数不能重复）

则输出

1，4；2，3。

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int MAX = 10;
    int *a = (int *)malloc(MAX * sizeof(int));
    int *b;
    FILE *fp1;
    FILE *fp2;
    fp1 = fopen("a.txt","r");
    if(fp1 = NULL)
    {
        printf("error1");
        exit(-1);
    }
    fp2 = fopen("b.txt","w");
    if(fp2 = NULL)
    {
        printf("error2");
        exit(-1);
    }
    int i = 0;
    int j = 0;
    while(fscanf(fp1,"%d",&a[i]) != EOF)
    {
        i++;
        j++;
        if(i >= MAX)
        {
            MAX = 2 * MAX;
            b = (int*)realloc(a,MAX * sizeof(int));
            if(b == NULL)
            {
                printf("error3");
                exit(-1);
            }
            a = b;
        }
    }
    for(;--j >= 0;)
    fprintf(fp2,"%d\n",a[j]);
    fclose(fp1);
    fclose(fp2);
    return 0;
```

```
}
```

第二题.

```
#include <stdio.h>
int main(void)
{
    unsigned long int i,j,k;
    printf("please input the number\n");
    scanf("%d",&i);
    if( i % 2 == 0)
    j = i / 2;
    else
    j = i / 2 + 1;
    rintf("The result is \n");
    for(k = 0; k < j; k++)
    printf("%d = %d + %d\n",i,k,i - k);
    return 0;
}
#include <stdio.h>
void main()
{
    unsigned long int a,i=1;
    scanf("%d",&a);
    if(a%2==0)
    {
        for(i=1;i<a/2;i++)
        printf("%d",a,a-i);
    }
    else
    for(i=1;i<=a/2;i++)
        printf(" %d, %d",i,a-i);
}
```

10. 在对齐为 4 的情况下

```
struct BBB
{
    long num;
    char *name;
    short int data;
    char ha;
    short ba[5];
}*p;
p=0x1000000;
p+0x200=____;
(Ulong)p+0x200=____;
```

(char*)p+0x200=____;
解答：假设在 32 位 CPU 上，
sizeof(long) = 4 bytes
sizeof(char *) = 4 bytes
sizeof(short int) = sizeof(short) = 2 bytes
sizeof(char) = 1 bytes
由于是 4 字节对齐，
sizeof(struct BBB) = sizeof(*p)
= 4 + 4 + 2 + 1 + 1/*补齐*/ + 2*5 + 2/*补齐*/ = 24 bytes (经 Dev-C++验证)
p=0x1000000;
p+0x200=____;
= 0x1000000 + 0x200*24
(Ulong)p+0x200=____;
= 0x1000000 + 0x200
(char*)p+0x200=____;
= 0x1000000 + 0x200*4

11. int func(int a)
```
{
    int b;
    switch(a)
    {
        case 1: 30;
        case 2: 20;
        case 3: 16;
        default: 0
    }
    return b;
}
```
则 func(1)=?
// b 定义后就没有赋值。

12. int a[3];
a[0]=0; a[1]=1; a[2]=2;
int *p, *q;
p=a;
q=&a[2];
则 a[q-p]=a[2]
解释：指针一次移动一个 int 但计数为 1

13. 线形表 a、b 为两个有序升序的线形表，编写一程序，使两个有序线形表合并成一个有序升序线形表 h;
```
Linklist *unio(Linklist *p,Linklist *q){
linklist *R,*pa,*qa,*ra;
```

```
pa=p;
qa=q;
R=ra=p;
while(pa->next!=NULL&&qa->next!=NULL){
if(pa->data>qa->data){
ra->next=qa;
qa=qa->next;
}
else{
ra->next=pa;
pa=pa->next;
}
}
if(pa->next!=NULL)
ra->next=pa;
if(qa->next!=NULL)
ra->next==qa;
return R;
}
```

14. 运用四色定理，为 N 个局域举行配色，颜色为 1、2、3、4 四种，另有数组 adj[][N]，如 adj[i][j]=1 则表示 i 区域与 j 区域相邻，数组 color[N]，如 color[i]=1,表示 i 区域的颜色为 1 号颜色。

15. 用递归算法判断数组 a[N]是否为一个递增数组。
递归的方法，记录当前最大的，并且判断当前的是否比这个还大，大则继续，否则返回 false 结束：

```
bool fun( int a[], int n )
{
if( n= =1 )
return true;
if( n= =2 )
return a[n-1] >= a[n-2];
return fun( a,n-1) && ( a[n-1] >= a[n-2] );
}
```

16. 编写算法，从 10 亿个浮点数当中，选出其中最大的 10000 个。
用外部排序

17. 编写一 unix 程序，防止僵尸进程的出现.

18. 有一个数组 a[1000]存放 0--1000;要求每隔二个数删掉一个数，到末尾时循环至开头继续进行，求最后一个被删掉的数的原始下标位置。
以 7 个数为例：

{0,1,2,3,4,5,6,7} 0-->1-->2（删除）-->3-->4-->5(删除)-->6-->7-->0（删除），如此循环直到最后一个数被删除。

方法 1：数组

```
#include <iostream>
using namespace std;
#define null 1000
int main()
{
int arr[1000];
for (int i=0;i<1000;++i)
arr[i]=i;
int j=0;
int count=0;
while(count<999)
{
while(arr[j%1000]==null)
j=(++j)%1000;
j=(++j)%1000;
while(arr[j%1000]==null)
j=(++j)%1000;
j=(++j)%1000;
while(arr[j%1000]==null)
j=(++j)%1000;
arr[j]=null;
++count;
}
while(arr[j]==null)
j=(++j)%1000;
cout<<j<<endl;
return 0;
}
```

方法 2：链表

```
#include<iostream>
using namespace std;
#define null 0
struct node
{
int data;
node* next;
};
int main()
{
node* head=new node;
head->data=0;
```

```
head->next=null;
node* p=head;
for(int i=1;i<1000;i++)
{
node* tmp=new node;
tmp->data=i;
tmp->next=null;
head->next=tmp;
head=head->next;
}
head->next=p;
while(p!=p->next)
{
p->next->next=p->next->next->next;
p=p->next->next;
}
cout<<p->data;
return 0;
}
```

方法 3：通用算法

```
#include <stdio.h>
#define MAXLINE 1000 //元素个数
/*
MAXLINE 元素个数
a[] 元素数组
R[] 指针场
suffix 下标
index 返回最后的下标序号
values 返回最后的下标对应的值
start 从第几个开始
K 间隔
*/
int find_n(int a[],int R[],int K,int& index,int& values,int s=0) {
int suffix;
int front_node,current_node;
suffix=0;
if(s==0) {
current_node=0;
front_node=MAXLINE-1;
}
else {
current_node=s;
front_node=s-1;
}
```

```
while(R[front_node]!=front_node) {
printf("%d\n",a[current_node]);
R[front_node]=R[current_node];
if(K==1) {
current_node=R[front_node];
continue;
}
for(int i=0;i<K;i++){
front_node=R[front_node];
}
current_node=R[front_node];
}
index=front_node;
values=a[front_node];
return 0;
}
int main(void) {
int a[MAXLINE],R[MAXLINE],suffix,index,values,start,i,K;
suffix=index=values=start=0;
K=2;
for(i=0;i<MAXLINE;i++) {
a[i]=i;
R[i]=i+1;
}
R[i-1]=0;
find_n(a,R,K,index,values,2);
printf("the value is %d,%d\n",index,values);
return 0;
}
```

19. 改错

```
void test2()
{
char string[10], str1[10];
int i;
for(i=0; i<10; i++)
{
str1[i] = 'a';
}
strcpy( string, str1 );
}
```

解答：如果面试者指出字符数组 str1 不能在数组内结束可以给 3 分；如果面试者指出 strcpy(string, str1)调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 strcpy 工作方式的给 10 分；

str1 不能在数组内结束:因为 str1 的存储为：{a,a,a,a,a,a,a,a,a,a},没有'\0'(字符串结束符)，所以不能结束
strcpy(char *s1,char *s2)他的工作原理是，扫描 s2 指向的内存，逐个字符付到 s1 所指向的内存，直到碰到'\0',因为 str1 结尾没有'\0'，所以具有不确定性，不知道他后面还会付什么东东。
正确应如下

```
void test2()
{
char string[10], str1[10];
int i;
for(i=0; i<9; i++)
{
str1[i] = 'a'+i; //把 abcdefghi 赋值给字符数组
}
str[i]='\0';//加上结束符
strcpy( string, str1 );
}
```

20. 实现 strcmp

```
int StrCmp(const char *str1, const char *str2)
int StrCmp(const char *str1, const char *str2)
{
assert(str1 && srt2);
while (*str1 && *str2 && *str1 == *str2) {
str1++, str2++;
}
if (*str1 && *str2)
return (*str1-*str2);
elseif (*str1 && *str2==0)
return 1;
elseif (*str1 = = 0 && *str2)
return -1;
else
return 0;
}
int StrCmp(const char *str1, const char *str2)
{
//省略判断空指针(自己保证)
while(*str1 && *str1++ = = *str2++);
return *str1-*str2;
}
```

21. 实现子串定位

```
int FindSubStr(const char *MainStr, const char *SubStr)
```

做是做对了，没有抄搞，比较乱

```
int MyStrstr(const char* MainStr, const char* SubStr)
{
const char *p;
const char *q;
const char * u = MainStr;
//assert((MainStr!=NULL)&&( SubStr!=NULL));//用断言对输入进行判断
while(*MainStr) //内部进行递增
{
p = MainStr;
q = SubStr;
while(*q && *p && *p++ == *q++);
if(!*q )
{
return MainStr - u +1 ;//MainStr 指向当前起始位，u 指向
}
MainStr ++;
}
return -1;
}
```

分析：

```
int arr[] = {6,7,8,9,10};
int *ptr = arr;
*(ptr++)+=123;
printf(“ %d %d ”, *ptr, *(++ptr));
```

输出：8 8

过程：对于*(ptr++)+=123;先做加法 6+123，然后++，指针指向 7；对于 printf(“ %d %d ”, *ptr, *(++ptr));从后往前执行，指针先++，指向 8，然后输出 8，紧接着再输出 8

22. 有一个 16 位的整数，每 4 位为一个数，写函数求他们的和。

解释：

整数 1101010110110111

和 1101+0101+1011+0111

答案：用十进制做参数，计算时按二进制考虑。

```
/* n 就是 16 位的数，函数返回它的四个部分之和 */
char SumOfQuaters(unsigned short n)
{
char c = 0;
int i = 4;
do
{
c += n & 15;
```

```
n = n >> 4;
} while (--i);
return c;
}
```

23. 有1,2,....一直到n的无序数组,求排序算法,并且要求时间复杂度为O(n),空间复杂度O(1),使用交换,而且一次只能交换两个数.（华为）

```
#include<iostream.h>
int main()
{
int a[] = {10,6,9,5,2,8,4,7,1,3};
int len = sizeof(a) / sizeof(int);
int temp;
for(int i = 0; i < len; )
{
temp = a[a[i] - 1];
a[a[i] - 1] = a[i];
a[i] = temp;
if ( a[i] == i + 1)
i++;
}
for (int j = 0; j < len; j++)
cout<<a[j]<<",";
return 0;
}
```

24. 写出程序把一个链表中的结点顺序倒排

```
typedef struct linknode
{
int data;
struct linknode *next;
}node;
//将一个链表逆置
node *reverse(node *head)
{
node *p,*q,*r;
p=head;
q=p->next;
while(q!=NULL)
{
r=q->next;
q->next=p;
p=q;
q=r;
```

```
}
head->next=NULL;
head=p;
return head;
}
```

25. 写出程序删除链表中的所有结点

```
void del_all(node *head)
{
node *p;
while(head!=NULL)
{
p=head->next;
free(head);
head=p;
}
cout<<"释放空间成功!"<<endl;
}
```

26. 两个字符串，s,t;把 t 字符串插入到 s 字符串中，s 字符串有足够的空间存放 t 字符串

```
void insert(char *s, char *t, int i)
{
char *q = t;
char *p =s;
if(q == NULL)return;
while(*p!='\0')
{
p++;
}
while(*q!=0)
{
*p=*q;
p++;
q++;
}
*p = '\0';
}
```

27. memcpy 拷贝一块内存，内存的大小你告诉它

strcpy 是字符串拷贝，遇到'\0'结束

```
/* memcpy ——— 拷贝不重叠的内存块 */
void memcpy(void* pvTo, void* pvFrom, size_t size)
{
void* pbTo = (byte*)pvTo;
```

```
    void* pbFrom = (byte*)pvFrom;
    ASSERT(pvTo != NULL && pvFrom != NULL); //检查输入指针的有效性
    ASSERT(pbTo>=pbFrom+size || pbFrom>=pbTo+size);//检查两个指针指向的内存是否
重叠
    while(size-->0)
    *pbTo++ == *pbFrom++;
    return(pvTo);
    }
```

28. 怎么判断链表中是否有环？

```
    bool CircleInList(Link* pHead)
    {
    if(pHead = = NULL || pHead->next = = NULL)//无节点或只有一个节点并且无自环
    return (false);
    if(pHead->next = = pHead)//自环
    return (true);
    Link *pTemp1 = pHead;//step 1
    Link *pTemp = pHead->next;//step 2
    while(pTemp != pTemp1 && pTemp != NULL && pTemp->next != NULL)
    {
    pTemp1 = pTemp1->next;
    pTemp = pTemp->next->next;
    }
    if(pTemp = = pTemp1)
    return (true);
    return (false);
    }
```

29. 两个字符串，s,t;把 t 字符串插入到 s 字符串中，s 字符串有足够的空间存放 t 字符串

```
    void insert(char *s, char *t, int i)
    {
    memcpy(&s[strlen(t)+i],&s[i],strlen(s)-i);
    memcpy(&s[i],t,strlen(t));
    s[strlen(s)+strlen(t)]='\0';
    }
```

30. 编写一个 C 函数，该函数在一个字符串中找到可能的最长的子字符串，且该字符串是由同一字符组成的。

```
    char * search(char *cpSource, char ch)
    {
    char *cpTemp=NULL, *cpDest=NULL;
    int iTemp, iCount=0;
    while(*cpSource)
    {
```

```
if(*cpSource == ch)
{
iTemp = 0;
cpTemp = cpSource;
while(*cpSource == ch)
++iTemp, ++cpSource;
if(iTemp > iCount)
iCount = iTemp, cpDest = cpTemp;
if(!*cpSource)
break;
}
++cpSource;
}
return cpDest;
}
```

31. 请编写一个 C 函数，该函数在给定的内存区域搜索给定的字符，并返回该字符所在位置索引值。

```
int search(char *cpSource, int n, char ch)
{
int i;
for(i=0; i<n && *(cpSource+i) != ch; ++i);
return i;
}
```

32. 一个单向链表，不知道头节点,一个指针指向其中的一个节点，问如何删除这个指针指向的节点？

将这个指针指向的 next 节点值 copy 到本节点，将 next 指向 next->next,并随后删除原 next 指向的节点。

```
#include <stdio.h>
void foo(int m, int n)
{
printf("m=%d, n=%d\n", m, n);
}
int main()
{
int b = 3;
foo(b+=3, ++b);
printf("b=%d\n", b);
return 0;
}
```

输出：m=7,n=4,b=7(VC6.0)

这种方式和编译器中得函数调用关系相关即先后入栈顺序。不过不同

编译器得处理不同。也是因为 C 标准中对这种方式说明为未定义，所以

各个编译器厂商都有自己得理解，所以最后产生得结果完全不同。
因为这样，所以遇见这种函数，我们首先要考虑我们得编译器会如何处理这样得函数，其次看函数得调用方式，不同得调用方式，可能产生不同得结果。最后是看编译器优化。

33. 写一函数，实现删除字符串 str1 中含有的字符串 str2.
第二个就是利用一个 KMP 匹配算法找到 str2 然后删除（用链表实现的话，便捷于数组）

34. 给定字符串 A 和 B,输出 A 和 B 中的最大公共子串。
比如 A="aocdfe" B="pmcdfa" 则输出"cdf"

```
*/
//Author: azhen
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char *commanstring(char shortstring[], char longstring[])
{
int i, j;
char *substring=malloc(256);
if(strstr(longstring, shortstring)!=NULL) //如果……，那么返回 shortstring
return shortstring;
for(i=strlen(shortstring)-1;i>0; i--) //否则，开始循环计算
{
for(j=0; j<=strlen(shortstring)-i; j++){
memcpy(substring, &shortstring[j], i);
substring[i]='\0';
if(strstr(longstring, substring)!=NULL)
return substring;
}
}
return NULL;
}
main()
{
char *str1=malloc(256);
char *str2=malloc(256);
char *comman=NULL;
gets(str1);
gets(str2);
if(strlen(str1)>strlen(str2)) //将短的字符串放前面
comman=commanstring(str2, str1);
else
comman=commanstring(str1, str2);
printf("the longest comman string is: %s\n", comman);
```

```
}
```

35. 写一个函数比较两个字符串 str1 和 str2 的大小，若相等返回 0，若 str1 大于 str2 返回 1，若 str1 小于 str2 返回－1

```
int strcmp ( const char * src,const char * dst)
{
int ret = 0 ;
while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && *dst)
{
++src;
++dst;
}
if ( ret < 0 )
ret = -1 ;
else if ( ret > 0 )
ret = 1 ;
return( ret );
}
```

36. 求 1000！的末尾有几个 0（用素数相乘的方法来做，如 72=2*2*2*3*3）；

求出 1->1000 里,能被 5 整除的数的个数 n1,能被 25 整除的数的个数 n2,能被 125 整除的数的个数 n3,能被 625 整除的数的个数 n4.1000!末尾的零的个数=n1+n2+n3+n4;

```
#include<stdio.h>
#define NUM 1000
int find5(int num){
int ret=0;
while(num%5==0){
num/=5;
ret++;
}
return ret;
}
int main(){
int result=0;
int i;
for(i=5;i<=NUM;i+=5)
{
result+=find5(i);
}
printf(" the total zero number is %d\n",result);
return 0;
}
```

37. 有双向循环链表结点定义为：

```
struct node
{ int data;
struct node *front,*next;
}
```
有两个双向循环链表 A，B，知道其头指针为：pHeadA,pHeadB，请写一函数将两链表中 data 值相同的结点删除
```
BOOL DeteleNode(Node *pHeader, DataType Value)
{
if (pHeader == NULL) return;
BOOL bRet = FALSE;
Node *pNode = pHead;
while (pNode != NULL)
{
if (pNode->data == Value)
{
if (pNode->front == NULL)
{
pHeader = pNode->next;
pHeader->front = NULL;
}
else
{
if (pNode->next != NULL)
{
pNode->next->front = pNode->front;
}
pNode->front->next = pNode->next;
}
Node *pNextNode = pNode->next;
delete pNode;
pNode = pNextNode;
bRet = TRUE;
//不要 break 或 return, 删除所有
}
else
{
pNode = pNode->next;
}
}
return bRet;
}
void DE(Node *pHeadA, Node *pHeadB)
{
if (pHeadA == NULL || pHeadB == NULL)
```

```
{
return;
}
Node *pNode = pHeadA;
while (pNode != NULL)
{
if (DeteleNode(pHeadB, pNode->data))
{
if (pNode->front == NULL)
{
pHeadA = pNode->next;
pHeadA->front = NULL;
}
else
{
pNode->front->next = pNode->next;
if (pNode->next != NULL)
{
pNode->next->front = pNode->front;
}
}
Node *pNextNode = pNode->next;
delete pNode;
pNode = pNextNode;
}
else
{
pNode = pNode->next;
}
}
}
```

38. 编程实现：找出两个字符串中最大公共子字符串,如"abccade","dgcadde"的最大子串为"cad"

```
int GetCommon(char *s1, char *s2, char **r1, char **r2)
{
int len1 = strlen(s1);
int len2 = strlen(s2);
int maxlen = 0;
for(int i = 0; i < len1; i++)
{
for(int j = 0; j < len2; j++)
{
if(s1[i] == s2[j])
```

```
{
int as = i, bs = j, count = 1;
while(as + 1 < len1 && bs + 1 < len2 && s1[++as] == s2[++bs])
count++;
if(count > maxlen)
{
maxlen = count;
*r1 = s1 + i;
*r2 = s2 + j;
}
}
}
}
```

39. 编程实现：把十进制数(long 型)分别以二进制和十六进制形式输出，不能使用 printf 系列库函数

```
char* test3(long num) {
char* buffer = (char*)malloc(11);
buffer[0] = '0';
buffer[1] = 'x';
buffer[10] = '\0';
char* temp = buffer + 2;
for (int i=0; i < 8; i++) {
temp[i] = (char)(num<<4*i>>28);
temp[i] = temp[i] >= 0 ? temp[i] : temp[i] + 16;
temp[i] = temp[i] < 10 ? temp[i] + 48 : temp[i] + 55;
}
return buffer;
}
```

40. 输入 N, 打印 N*N 矩阵

比如 N = 3，打印：

```
1 2 3
8 9 4
7 6 5
```

N = 4，打印：

```
1 2 3 4
12 13 14 5
11 16 15 6
10 9 8 7
```

解答：

```
1 #define N 15
int s[N][N];
void main()
```

```
{
int k = 0, i = 0, j = 0;
int a = 1;
for( ; k < (N+1)/2; k++ )
{
while( j < N-k ) s[i][j++] = a++; i++; j--;
while( i < N-k ) s[i++][j] = a++; i--; j--;
while( j > k-1 ) s[i][j--] = a++; i--; j++;
while( i > k ) s[i--][j] = a++; i++; j++;
}
for( i = 0; i < N; i++ )
{
for( j = 0; j < N; j++ )
cout << s[i][j] << '\t';
cout << endl;
}
}
```

41.
```
define MAX_N 100
int matrix[MAX_N][MAX_N];
/*
* (x,y)：第一个元素的坐标
* start：第一个元素的值
* n：矩阵的大小
*/
void SetMatrix(int x, int y, int start, int n) {
int i, j;
if (n <= 0) //递归结束条件
return;
if (n == 1) { //矩阵大小为 1 时
matrix[x][y] = start;
return;
}
for (i = x; i < x + n-1; i++) //矩阵上部
matrix[y][i] = start++;
for (j = y; j < y + n-1; j++) //右部
matrix[j][x+n-1] = start++;
for (i = x+n-1; i > x; i--) //底部
matrix[y+n-1][i] = start++;
for (j = y+n-1; j > y; j--) //左部
matrix[j][x] = start++;
SetMatrix(x+1, y+1, start, n-2); //递归
}
void main() {
```

```
int i, j;
int n;
scanf("%d", &n);
SetMatrix(0, 0, 1, n);
//打印螺旋矩阵
for(i = 0; i < n; i++) {
for (j = 0; j < n; j++)
printf("%4d", matrix[i][j]);
printf("\n");
}
}
```

42. 斐波拉契数列递归实现的方法如下：

```
int Funct( int n )
{
if(n==0) return 1;
if(n==1) return 1;
retrurn Funct(n-1) + Funct(n-2);
}
```

如何不使用递归，来实现上述函数？

解答：

```
int Funct( int n ) // n 为非负整数
{
int a=0;
int b=1;
int c;
if(n==0) c=1;
else if(n==1) c=1;
else for(int i=2;i<=n;i++) //应该 n 从 2 开始算起
{
c=a+b;
a=b;
b=c;
}
return c;
}
```

43. 已知 strcpy 函数的原型是：

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

1.不调用库函数，实现 strcpy 函数。

2.解释为什么要返回 char *。

解说：

1.strcpy 的实现代码

```
char * strcpy(char * strDest,const char * strSrc)
{
```

```
if ((strDest==NULL)||(strSrc==NULL)) file://[/1]
throw "Invalid argument(s)"; //[2]
char * strDestCopy=strDest; file://[/3]
while ((*strDest++=*strSrc++)!='\0'); file://[/4]
return strDestCopy;
}
```

错误的做法：

[1]

(A)不检查指针的有效性，说明答题者不注重代码的健壮性。

(B)检查指针的有效性时使用((!strDest)||(!strSrc))或(!(strDest&&strSrc))，说明答题者对 C 语言中类型的隐式转换没有深刻认识。在本例中 char *转换为 bool 即是类型隐式转换，这种功能虽然灵活，但更多的是导致出错概率增大和维护成本升高。所以 C++专门增加了 bool、true、false 三个关键字以提供更安全的条件表达式。

(C)检查指针的有效性时使用((strDest==0)||(strSrc==0))，说明答题者不知道使用常量的好处。直接使用字面常量（如本例中的 0）会减少程序的可维护性。0 虽然简单，但程序中可能出现很多处对指针的检查，万一出现笔误，编译器不能发现，生成的程序内含逻辑错误，很难排除。而使用 NULL 代替 0，如果出现拼写错误，编译器就会检查 出来。

[2]

(A)return new string("Invalid argument(s)");，说明答题者根本不知道返回值的用途，并且他对内存泄漏也没有警惕心。从函数中返回函数体内分配的内存是十分危险的做法，他把释放内存的义务抛给不知情的调用者，绝大多数情况下，调用者不会释放内存，这导致内存泄漏。

(B)return 0;，说明答题者没有掌握异常机制。调用者有可能忘记检查返回值，调用者还可能无法检查返回值（见后面的链式表达式）。妄想让返回值肩负返回正确值和异常值的双重功能，其结果往往是两种功能都失效。应该以抛出异常来代替返回值，这样可以减轻调用者的负担、使错误不会被忽略、增强程序的可维护性。

[3]

(A)忘记保存原始的 strDest 值，说明答题者逻辑思维不严密。

[4]

(A)循环写成 while (*strDest++=*strSrc++);，同[1](B)。

(B)循环写成 while (*strSrc!='\0') *strDest++=*strSrc++;，说明答题者对边界条件的检查不力。循环体结束后，strDest 字符串的末尾没有正确地加上'\0'。

44. 已知类 String 的原型为：

```
classString
{
public：
String(constchar*str=NULL);//普通构造函数
String(constString);//拷贝构造函数
~String(void);//析构函数
String&operator=(constString);//赋值构造函数
private:
char*m_data;//用于保存字符串
```

```
};
```
请编写 String 的上述 4 个函数。
答案：
版本 1
```
//String 的析构函数
String::~String(void)//3 分
{
delete[]m_data;
//由于 m_data 是内部数据类型，也可以写成 deletem_data;
}
String::String(constchar*str)
{
if(str==NULL)
{
m_data=newchar[1];//若能加 NULL 判断则更好
*m_data= ‘{post.content}’ ;
}
Else
{
intlength=strlen(str);
m_data=newchar[length+1];//若能加 NULL 判断则更好
strcpy(m_data,str);
}
}
//拷贝构造函数
String::String(constString&other)
{
intlength=strlen(other.m_data);
m_data=newchar[length+1];//若能加 NULL 判断则更好
strcpy(m_data,other.m_data);
}
//赋值函数
String&String:operate=(constString&other)
{
//(1)检查自赋值
if(this==&other)
return*this;
//(2)释放原有的内存资源
delete[]m_data;
//（3）分配新的内存资源，并复制内容
intlength=strlen(other.m_data);
m_data=newchar[length+1];//若能加 NULL 判断则更好
strcpy(m_data,other.m_data);
//（4）返回本对象的引用
```

```
return*this;
}
```
版本 2
```
String::String(constchar*str)
{
if(str){
memset(m_data,0,strlen(m_data));
strcpy(m_data,str);
}
else*m_data=0;
}
String::String(constString)
{
strcpy(m_data,copy.m_data);
}
String&String:operator=(constString)
{
if(this==)retrun*this;
strcpy(m_data,copy.m_data);
return*this;
}
```
版本 3
```
String::String(constchar*str)
{
if(m_data)
delete[]m_data;
if(str){
m_data=newchar[strlen(str)];
memset(m_data,0,strlen(m_data));
strcpy(m_data,str);
}
else*m_data=0;
}
String::String(constString)
{
if(m_data)
delete[]m_data;
m_data=newchar[strlen(copy.m_data+1)]
strcpy(m_data,copy.m_data);
}
String&String:operator=(constString)
{
if(this==)retrun*this;
if(m_data)
```

```
delete[]m_data;
m_data=newchar[strlen(copy.m_data+1)]
strcpy(m_data,copy.m_data);
return*this;
 }
~String::String(void)
{
if(m_data)
delete[]m_data;
}
```

45. 有程序片断如下

```
intmain()
{
intI=20;
pid_tpid=5;
if((pid=fork())>0)
{
I=50;
printf("%d\n",I);(1)
}
elseif(pid==0)
{
printf("%d\n",I);(2)
}
}
```

请问该程序用的是进程方式还是线程方式，并说明进程与线程的区别：

请问该程序输出什么结果？

46. 下面等价的是：

```
Ainti=0
if(i)
{
printf("hello,world");
}
Binti=1;
intj=2;
if(i==1||j==2)
{
printf("hello,world");
}
CBooleanb1=true;
Booleanb2=true;
```

```
if(b1==b2)
{
printf("hello,world");
}
Dinti=1;
intj=2;
if(i==1&|j==2)
{
printf("hello,world");
}
```

47. 指针++的含义和用法

48. 怎样避免内存泄漏的问题

49. 编程实现十进制数转化为十六进制输出，不准用任何已经定义的库函数，比方说 String，Math。inttoHex(int)

50. 论述含参数的宏与函数的优缺点。

1，程序设计（可以用自然语言来描述，不编程）：C/C++源代码中，检查花括弧（是"（"与"）"，"{"与"}"）是否匹配，若不匹配，则输出不匹配花括弧所在的行与列。

51. 巧排数字，将 1,2,...,19,20 这 20 个数字排成一排，使得相邻的两个数字之和为一个素数，且首尾两数字之和也为一个素数。编程打印出所有的排法。

52. 打印一个 N*N 的方阵，N 为每边字符的个数（3〈N〈20），要求最外层为"X"，第二层为"Y"，从第三层起每层依次打印数字 0，1，2，3，...例子：当 N=5，打印出下面的图形：

```
XXXXX
XYYYX
XY0YX
XYYYX
XXXXX
```

53. 找错题

试题 1：

```
voidtest1()
{
charstring[10];
char*str1="0123456789";
strcpy(string,str1);
}
```

试题 2：

```
voidtest2()
{
```

```
charstring[10],str1[10];
inti;
for(i=0;i＜10;i++)
{
str1[i]='a';
}
strcpy(string,str1);
}
```
试题 3：
```
voidtest3(char*str1)
{
charstring[10];
if(strlen(str1)＜=10)
{
strcpy(string,str1);
}
}
```
解答：
试题 1 字符串 str1 需要 11 个字节才能存放下（包括末尾的’\0’），而 string 只有 10 个字节的空间，strcpy 会导致数组越界；
对试题 2，如果指出字符数组 str1 不能在数组内结束可以给 3 分；如果指出 strcpy(string,str1)调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 strcpy 工作方式的给 10 分；
对试题 3，if(strlen(str1)＜=10)应改为 if(strlen(str1)＜10)，因为 strlen 的结果未统计’\0’所占用的 1 个字节。
剖析：(Coolbear：char*和 char[]都可以表示字符串，用 strlen(constchar*)函数求字符串长度时，都不计算'\0'所占的 1 个字节。不同之处在于给 char*赋值时可以加上'\0'，也可以不加，二者等价；但是对于 char[]，必须在初始化时显式加上'\0'，否则，判断字符串结束会出现问题。chararray[5]={'1','2','3','4','\0'}strlen(array)=4)考查对基本功的掌握：(1)字符串以’\0’结尾；(2)对数组越界把握的敏感度；(3)库函数 strcpy
```
voidstrcpy(char*strDest,char*strSrc)
{
while((*strDest++=*strSrc++)!=‘\0’);
}
voidstrcpy(char*strDest,constchar*strSrc)
//将源字符串加 const，表明其为输入参数，加 2 分
{
while((*strDest++=*strSrc++)!=‘\0’);
}
voidstrcpy(char*strDest,constchar*strSrc)
{
//对源地址和目的地址加非 0 断言，加 3 分
assert((strDest!=NULL)&&(strSrc!=NULL));
while((*strDest++=*strSrc++)!=‘\0’);
```

```
}
//为了实现链式操作，将目的地址返回，加 3 分！
char*strcpy(char*strDest,constchar*strSrc)
{
assert((strDest!=NULL)&&(strSrc!=NULL));
char*address=strDest;
while((*strDest++=*strSrc++)!= ‘\0’ );
returnaddress;
}
```

(4)对 strlen 的掌握，它没有包括字符串末尾的'\0'。读者看了不同分值的 strcpy 版本，应该也可以写出一个 10 分的 strlen 函数了，完美的版本为：

```
intstrlen(constchar*str)//输入参数 const
{
assert(strt!=NULL);//断言字符串地址非 0
intlen;
while((*str++)!='\0')
{
len++;
}
returnlen;
}
```

试题 4：

```
voidGetMemory(char*p)
{
p=(char*)malloc(100);
}
voidTest(void)
{
char*str=NULL;
GetMemory(str);
strcpy(str,"helloworld");
printf(str);
}
```

试题 5：

```
char*GetMemory(void)
 {
charp[]="helloworld";
returnp;
}
voidTest(void)
{
char*str=NULL;
str=GetMemory();
printf(str);
```

```
}
```
试题 6：
```
voidGetMemory(char**p,intnum)
{
*p=(char*)malloc(num);
}
voidTest(void)
{
char*str=NULL;
GetMemory(&str,100);
strcpy(str,"hello");
printf(str);
}
```
试题 7：
```
voidTest(void)
{
char*str=(char*)malloc(100);
strcpy(str,"hello");
free(str);
...//省略的其它语句
}
```
解答：

试题 4 传入中 GetMemory(char*p)函数的形参为字符串指针，在函数内部修改形参并不能真正的改变传入形参的值，执行完

```
char*str=NULL;
GetMemory(str);
```
后的 str 仍然为 NULL；

试题 5 中

```
charp[]="helloworld";
```
returnp;的 p[]数组为函数内的局部自动变量，在函数返回后，内存已经被释放。这是许多程序员常犯的错误，其根源在于不理解变量的生存期。

试题 6 的 GetMemory 避免了试题 4 的问题，传入 GetMemory 的参数为字符串指针的指针，但是在 GetMemory 中执行申请内存及赋值语句

```
*p=(char*)malloc(num);
```
后未判断内存是否申请成功，应加上：

```
if(*p==NULL)
{
.//进行申请内存失败处理
}
```
试题 7 存在与试题 6 同样的问题，在执行

```
char*str=(char*)malloc(100);
```
后未进行内存是否申请成功的判断；另外，在 free(str)后未置 str 为空，导致可能变成一个“野”指针，应加上：

```
str=NULL;
```

试题 6 的 Test 函数中也未对 malloc 的内存进行释放。
剖析：
看看下面的一段程序有什么错误：
```
swap(int*p1,int*p2)
{
int*p;
*p=*p1;
*p1=*p2;
*p2=*p;
}
```
在 swap 函数中，p 是一个“野”指针，有可能指向系统区，导致程序运行的崩溃。在 VC++中 DEBUG 运行时提示错误“AccessViolation”。该程序应该改为：
```
swap(int*p1,int*p2)
{
intp;
p=*p1;
*p1=*p2;
*p2=p;
}
```
++

54. 内功题
试题 1：分别给出 BOOL，int，float，指针变量与“零值”比较的 if 语句（假设变量名为 var）
解答：
BOOL 型变量：if(!var)
int 型变量：if(var==0)
float 型变量：
```
constfloatEPSINON=0.00001;
if((x＞=-EPSINON)&&(x＜=EPSINON)
```
指针变量：if(var==NULL)
剖析：
考查对 0 值判断的“内功”，BOOL 型变量的 0 判断完全可以写成 if(var==0)，而 int 型变量也可以写成 if(!var)，指针变量的判断也可以写成 if(!var)，上述写法虽然程序都能正确运行，但是未能清晰地表达程序的意思。
一般的，如果想让 if 判断一个变量的“真”、“假”，应直接使用 if(var)、if(!var)，表明其为“逻辑”判断;如果用 if 判断一个数值型变量(short、int、long 等)，应该用 if(var==0)，表明是与 0 进行“数值”上的比较；而判断指针则适宜用 if(var==NULL)，这是一种很好的编程习惯。
浮点型变量并不精确，所以不可将 float 变量用“==”或“！=”与数字比较，应该设法转化成“＞=”或“＜=”形式。如果写成 if(x==0.0)，则判为错，得 0 分。
试题 2：以下为 WindowsNT 下的 32 位 C++程序，请计算 sizeof 的值
```
voidFunc(charstr[100])
{
```

sizeof(str)=?
}
void*p=malloc(100);
sizeof(p)=?
解答：
sizeof(str)=4
sizeof(p)=4
剖析：
Func(charstr[100])函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。
数组名的本质如下：
（1）数组名指代一种数据结构，这种数据结构就是数组；
例如：
charstr[10];
cout＜＜sizeof(str)＜＜endl;
输出结果为 10，str 指代数据结构 char[10]。
（2）数组名可以转换为指向其指代实体的指针，而且是一个指针常量，不能作自增、自减等操作，不能被修改；
charstr[10];
str++;//编译出错，提示 str 不是左值
（3）数组名作为函数形参时，沦为普通指针。
WindowsNT32 位平台下，指针的长度（占用内存的大小）为 4 字节，故 sizeof(str)、sizeof(p)都为 4。
试题 3：写一个“标准”宏 MIN，这个宏输入两个参数并返回较小的一个。另外，当你写下面的代码时会发生什么事
least=MIN(*p++,b);
解答：
#defineMIN(A,B)((A)＜=(B)?(A):(B))
MIN(*p++,b)会产生宏的副作用
特别要注意两个问题：
（1）谨慎地将宏定义中的“参数”和整个宏用用括弧括起来。所以，严格地讲，下述解答：
#defineMIN(A,B)(A)＜=(B)?(A):(B)
#defineMIN(A,B)(A＜=B?A:B)都应判 0 分；
（2）防止宏的副作用。
宏定义#defineMIN(A,B)((A)＜=(B)?(A):(B))对 MIN(*p++,b)的作用结果是：
((*p++)＜=(b)?(*p++):(*p++))
这个表达式会产生副作用，指针 p 会作三次++自增操作。
除此之外，另一个应该判 0 分的解答是：
#defineMIN(A,B)((A)＜=(B)?(A):(B))
这个解答在宏定义的后面加“;”，显示编写者对宏的概念模糊不清
试题 4：为什么标准头文件都有类似以下的结构？
#ifndef__INCvxWorksh
#define__INCvxWorksh

```
#ifdef__cplusplus
extern"C"{
#endif
/*...*/
#ifdef__cplusplus
}
#endif
#endif/*__INCvxWorksh*/
```
解答：
头文件中的编译宏
```
#ifndef__INCvxWorksh
#define__INCvxWorksh
```
#endif 的作用是防止被重复引用。作为一种面向对象的语言，C++支持函数重载，而过程式语言 C 则不支持。函数被 C++编译后在 symbol 库中的名字与 C 语言的不同。例如，假设某个函数的原型为： voidfoo(intx,inty);该函数被 C 编译器编译后在 symbol 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。_foo_int_int 这样的名字包含了函数名和函数参数数量及类型信息，C++就是考这种机制来实现函数重载的。为了实现 C 和 C++的混合编程，C++提供了 C 连接交换指定符号 extern"C"来解决名字匹配问题，函数声明前加上 extern"C"后，则编译器就会按照 C 语言的方式将该函数编译为_foo，这样 C 语言中就可以调用 C++的函数了。

试题 5：编写一个函数，作用是把一个 char 组成的字符串循环右移 n 个。比如原来是“abcdefghi”如果 n=2，移位后应该是“hiabcdefgh”函数头是这样的：

```
//pStr 是指向以'\0'结尾的字符串的指针
//steps 是要求移动的 n
voidLoopMove(char*pStr,intsteps)
{
//请填充...
}
```
解答：
正确解答 1：
```
voidLoopMove(char*pStr,intsteps)
{
intn=strlen(pStr)-steps;
chartmp[MAX_LEN];
strcpy(tmp,pStr+n);
strcpy(tmp+steps,pStr);
*(tmp+strlen(pStr))='\0';
strcpy(pStr,tmp);
}
```
正确解答 2：
```
voidLoopMove(char*pStr,intsteps)
{
intn=strlen(pStr)-steps;
chartmp[MAX_LEN];
```

```
memcpy(tmp,pStr+n,steps);
memcpy(pStr+steps,pStr,n);
memcpy(pStr,tmp,steps);
}
```

最频繁被使用的库函数包括：

（1）strcpy

（2）memcpy

（3）memset

试题 6：已知 WAV 文件格式如下表，打开一个 WAV 文件，以适当的数据结构组织 WAV 文件头并解析 WAV 格式的各项信息。

55. 以下三条输出语句分别输出什么？

```
charstr1[]="abc";
charstr2[]="abc";
constcharstr3[]="abc";
constcharstr4[]="abc";
constchar*str5="abc";
constchar*str6="abc";
cout<cout<cout<
```

答：分别输出 false,false,true。str1 和 str2 都是字符数组，每个都有其自己的存储区，它们的值则是各存储区首地址，不等；str3 和 str4 同上，只是按 const 语义，它们所指向的数据区不能修改。str5 和 str6 并非数组而是字符指针，并不分配存储区，其后的“abc”以常量形式存于静态数据区，而它们自己仅是指向该区首地址的指针，相等。

56. 以下反向遍历 array 数组的方法有什么错误？

```
vectorarray;
array.push_back(1);
array.push_back(2);
array.push_back(3);
for(vector::size_typei=array.size()-1;i>=0;--i)//反向遍历 array 数组
{
Cout;
}
```

答：首先数组定义有误，应加上类型参数：vectorarray。其次 vector::size_type 被定义为 unsignedint，即无符号数，这样做为循环变量的 i 为 0 时再减 1 就会变成最大的整数，导致循环失去控制。

57. 以下代码有什么问题？

```
typedefvectorIntArray;
IntArrayarray;
array.push_back(1);
array.push_back(2);
array.push_back(2);
array.push_back(3);
//删除 array 数组中所有的 2
```

```
for(IntArray::iteratoritor=array.begin();itor!=array.end();++itor)
{
if(2==*itor)array.erase(itor);
}
```
答：同样有缺少类型参数的问题。另外，每次调用“array.erase(itor);”，被删除元素之后的内容会自动往前移，导致迭代漏项，应在删除一项后使 itor--，使之从已经前移的下一个元素起继续遍历。

以下代码能够编译通过吗，为什么？
```
unsignedintconstsize1=2;
charstr1[size1];
unsignedinttemp=0;
cin>>temp;
unsignedintconstsize2=temp;
charstr2[size2];
```
答：str2 定义出错，size2 非编译器期间常量，而数组定义要求长度必须为编译期常量。

58. 以下代码中的输出语句输出 0 吗，为什么？
```
structCLS
{
intm_i;
CLS(inti):m_i(i){}
CLS()
{
CLS(0);
}
};
CLSobj;
cout<
```
答：不能。在默认构造函数内部再调用带参的构造函数属用户行为而非编译器行为，亦即仅执行函数调用，而不会执行其后的初始化表达式。只有在生成对象时，初始化表达式才会随相应的构造函数一起调用。

59. C++中的空类，默认产生哪些类成员函数？
答：
```
classEmpty
{
public:
Empty();//缺省构造函数
Empty(constEmpty&);//拷贝构造函数
~Empty();//析构函数
Empty&operator=(constEmpty&);//赋值运算符
Empty*operator&();//取址运算符
constEmpty*operator&()const;//取址运算符 const
```

};

60. 非 C++内建型别 A 和 B，在哪几种情况下 B 能隐式转化为 A?
答：
a.classB:publicA{……}//B 公有继承自 A，可以是间接继承的
b.classB{operatorA();}//B 实现了隐式转化为 A 的转化
c.classA{A(constB&);}//A 实现了 non-explicit 的参数为 B(可以有其他带默认值的参数)构造函数
d.A&operator=(constA&);//赋值操作，虽不是正宗的隐式类型转换，但也可以勉强算一个

61. 不许用中间变量,把 StringABCDE 倒转
```
#include"stdafx.h"
#include
#include
usingnamespacestd;
int_tmain(intargc,_TCHAR*argv[])
{
char*ps=newchar[15];
strcpy_s(ps,15,"Iamyuchifang");
cout<<"beforereverse:"<cout<inti=0;
intj=13;
while(i{
ps[i]=ps[i]+ps[j];
ps[j]=ps[i]-ps[j];
ps[i]=ps[i]-ps[j];
i++;
j--;
}
cout<<"afterreverse"<cout<return0;
}
```

62. C++里面是不是所有的动作都是 main()引起的？如果不是，请举例。

63. C++里面如何声明 const void f(void)函数为 C 程序中的库函数？

64. 下列哪两个是等同的
```
int b;
A const int* a = &b;
B const* int a = &b;
C const int* const a = &b;
D int const* const a = &b;
```

65. 内联函数在编译时是否做参数类型检查？

66. 以下代码有什么问题？

```
struct Test
{
    Test( int ) {}
        Test() {}
        void fun() {}
    };
    void main( void )
    {
        Test a(1);
        a.fun();
        Test b();
        b.fun();
    }
```

答：变量 b 定义出错。按默认构造函数定义对象，不需要加括号。

67. 以下代码能够编译通过吗，为什么？

```
unsigned int const size1 = 2;
char str1[ size1 ];
unsigned int temp = 0;
cin >> temp;
unsigned int const size2 = temp;
char str2[ size2 ];
```

答：str2 定义出错，size2 非编译器期间常量，而数组定义要求长度必须为编译期常量。

68. 以下反向遍历 array 数组的方法有什么错误？

```
vector array;
array.push_back( 1 );
array.push_back( 2 );
array.push_back( 3 );
for( vector::size_type i=array.size()-1; i>=0; --i ) // 反向遍历 array 数组
{
   cout << array[i] << endl;
   }
```

答：首先数组定义有误，应加上类型参数：vector<int> array。其次 vector::size_type 被定义为 unsigned int，即无符号数，这样做为循环变量的 i 为 0 时再减 1 就会变成最大的整数，导致循环失去控制。

69. 写一个函数，完成内存之间的拷贝。

答：

```
void* mymemcpy( void *dest, const void *src, size_t count )
{
    char* pdest = static_cast<char*>( dest );
```

```
    const char* psrc = static_cast<const char*>( src );
    if( pdest>psrc && pdest<psrc+cout ) 能考虑到这种情况就行了
    {
            for( size_t i=count-1; i!=-1; --i )
            pdest[i] = psrc[i];
    }
    Else
   {
            for( size_t i=0; i<count; ++i )
            pdest[i] = psrc[i];
    }
    return dest;
}
```

70. 将字符以 16 进制表示

```
void char2Hex( char c  // 将字符以 16 进制表示
{
   char ch = c/0x10 + '0'; if( ch > '9'  ch += ('A'-'9'-1);
   char cl = c%0x10 + '0'; if( cl > '9'  cl += ('A'-'9'-1);
   cout << ch << cl << ' ';
}
char str[] = "I love 中国";
for( size_t i=0; i<strlen(str); ++i
   char2Hex( str[i] ;
cout << endl;
```

71. int (*s[10])(int) 表示的是什么啊

int (*s[10])(int) 函数指针数组，每个指针指向一个 int func(int param)的函数。

72. 找错题:

1) 请问下面程序有什么错误?

```
int a[60][250][1000],i,j,k;
for(k=0;k<=1000;k++)
for(j=0;j<250;j++)
for(i=0;i<60;i++)
a[i][j][k]=0;
```

把循环语句内外换一下

2) #define Max_CB 500

```
void LmiQueryCSmd(Struct MSgCB * pmsg)
{
unsigned char ucCmdNum;
......
for(ucCmdNum=0;ucCmdNum<Max_CB;ucCmdNum++)
{
......;
```

```
}
```
死循环

3) 以下是求一个数的平方的程序,请找出错误:

```
#define SQUARE(a)((a)*(a))
int a=5;
int b;
b=SQUARE(a++);
```

4)
```
typedef unsigned char BYTE
int examply_fun(BYTE gt_len; BYTE *gt_code)
{
BYTE *gt_buf;
gt_buf=(BYTE *)MALLOC(Max_GT_Length);
......
if(gt_len>Max_GT_Length)
{
return GT_Length_ERROR;
}
.......
}
```

5) 找错

```
Void test1()
{
char string[10];
char* str1="0123456789";
strcpy(string, str1);// 溢出，应该包括一个存放'\0'的字符 string[11]
}
Void test2()
{
char string[10], str1[10];
for(I=0; I<10;I++)
{
str1[i] ='a';
}
strcpy(string, str1);// I，i 没有声明。
}
Void test3(char* str1)
{
char string[10];
if(strlen(str1)<=10)// 改成<10,字符溢出，将 strlen 改为 sizeof 也可以
{
strcpy(string, str1);
}
}
void g(int**);
```

```
int main()
{
int line[10],i;
int *p=line; //p 是地址的地址
for (i=0;i<10;i++)
{
*p=i;
g(&p);//数组对应的值加 1
}
for(i=0;i<10;i++)
printf("%d\n",line[i]);
return 0;
}
void g(int**p)
{
(**p)++;
(*p)++;// 无效
}
```

输出：

1
2
3
4
5
6
7
8
9
10

6) 改错题，只能在原来的基础上增加代码，不能删除代码

```
#include
#include
voidfoo(intage,char*b)
{
b=(char*)malloc(64);
sprintf(b,"YourAgeis%d",age);
}
intmain()
{
char*f;
foo(23,f);
printf("%s\n",f);
}
```

答案

版本 1

```
#include
#include
voidfoo(intage,char**b)
{
*b=(char*)malloc(64);
sprintf(*b,"YourAgeis%d",age);
}
intmain()
{
char**f;
foo(23,f);
printf("%s\n",**f);
return0;
}
```

版本 2

```
#include
#include
voidfoo(intage,char*&b)
{
b=(char*)malloc(64);
sprintf(b,"YourAgeis%d",age);
}
intmain()
{
char*f;
foo(23,f);
printf("%s\n",f);
free(f);//不要忘了 free;
}
```

73. 代码分析题：

1) 分析下面的代码：

```
char *a = "hello";
char *b = "hello";
if(a= =b)
printf("YES");
else
printf("NO");
```

2)

```
#include "stdio.h"
#include "string.h"
void main()
{
char aa[10];
```

```
printf("%d",strlen(aa));
}
```

sizeof()和初不初始化，没有关系；

strlen()和初始化有关。

```
char (*str)[20];/*str 是一个数组指针，即指向数组的指针．*/
char *str[20];/*str 是一个指针数组，其元素为指针型数据．*/
long a=0x801010;
a+5=?
```

0x801010 用二进制表示为：“1000 0000 0001 0000 0001 0000”，十进制的值为 8392720，再加上 5 就是 8392725

3) 以下代码中的两个 sizeof 用法有问题吗？

```
 void UpperCase( char str[] ) // 将 str 中的小写字母转换成大写字母
{
for( size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i )
if( 'a'<=str[i] && str[i]<='z' )
str[i] -= ('a'-'A' );
}
char str[] = "aBcDe";
cout << "str 字符长度为: " << sizeof(str)/sizeof(str[0]) << endl;
UpperCase( str );
cout << str << endl;
```

答：函数内的 sizeof 有问题。根据语法，sizeof 如用于数组，只能测出静态数组的大小，无法检测动态分配的或外部数组大小。函数外的 str 是一个静态定义的数组，因此其大小为 6，函数内的 str 实际只是一个指向字符串的指针，没有任何额外的与数组相关的信息，因此 sizeof 作用于上只将其当指针看，一个指针为 4 个字节，因此返回 4。

4) 以下代码有什么问题？

```
typedef vector IntArray;
IntArray array;
array.push_back( 1 );
array.push_back( 2 );
array.push_back( 2 );
array.push_back( 3 );
// 删除 array 数组中所有的 2
for( IntArray::iterator itor=array.begin(); itor!=array.end(); ++itor )
{
    if( 2 == *itor ) array.erase( itor );
    }
```

答：同样有缺少类型参数的问题。另外，每次调用“array.erase(itor);”，被删除元素之后的内容会自动往前移，导致迭代漏项，应在删除一项后使 itor--，使之从已经前移的下一个元素起继续遍历。

5) 以下两条输出语句分别输出什么？

```
float a = 1.0f;
cout << (int)a << endl;
```

```
cout << (int&)a << endl;
cout << boolalpha << ( (int)a == (int&)a ) << endl; // 输出什么？
float b = 0.0f;
cout << (int)b << endl;
cout << (int&)b << endl;
cout << boolalpha << ( (int)b == (int&)b ) << endl; // 输出什么？
```

答：分别输出 false 和 true。注意转换的应用。(int)a 实际上是以浮点数 a 为参数构造了一个整型数，该整数的值是 1，(int&)a 则是告诉编译器将 a 当作整数看（并没有做任何实质上的转换）。因为 1 以整数形式存放和以浮点形式存放其内存数据是不一样的，因此两者不等。对 b 的两种转换意义同上，但是 0 的整数形式和浮点形式其内存数据是一样的，因此在这种特殊情形下，两者相等（仅仅在数值意义上）。

注意，程序的输出会显示(int&)a=1065353216，这个值是怎么来的呢？前面已经说了，1 以浮点数形式存放在内存中，按 ieee754 规定，其内容为 0x0000803F（已考虑字节反序）。这也就是 a 这个变量所占据的内存单元的值。当(int&)a 出现时，它相当于告诉它的上下文：“把这块地址当做整数看待！不要管它原来是什么。”这样，内容 0x0000803F 按整数解释，其值正好就是 1065353216（十进制数）。

通过查看汇编代码可以证实“(int)a 相当于重新构造了一个值等于 a 的整型数”之说，而(int&)的作用则仅仅是表达了一个类型信息，意义在于为 cout<<及==选择正确的重载版本。

6) 以下三条输出语句分别输出什么？

```
char str1[]       = "abc";
char str2[]       = "abc";
const char str3[] = "abc";
const char str4[] = "abc";
const char* str5  = "abc";
const char* str6  = "abc";
cout << boolalpha << ( str1==str2 ) << endl; // 输出什么？
cout << boolalpha << ( str3==str4 ) << endl; // 输出什么？
cout << boolalpha << ( str5==str6 ) << endl; // 输出什么？
```

答：分别输出 false,false,true。str1 和 str2 都是字符数组，每个都有其自己的存储区，它们的值则是各存储区首地址，不等；str3 和 str4 同上，只是按 const 语义，它们所指向的数据区不能修改。str5 和 str6 并非数组而是字符指针，并不分配存储区，其后的“abc”以常量形式存于静态数据区，而它们自己仅是指向该区首地址的指针，相等。

7) 以下代码有什么问题

```
cout<<(true?1:"1")<endl;
```

答：三元表达式“？：”问号后面的两个操作数必须为同一类　　型。

8) 分析下列程序的执行结果

```
#include<iostream.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <memory.h>
typedef struct AA
{
int b1:5;
int b2:2;
}AA;
void main()
{
AA aa;
char cc[100];
strcpy(cc,"0123456789abcdefghijklmnopqrstuvwxyz");
memcpy(&aa,cc,sizeof(AA));
cout << aa.b1 <<endl;
cout << aa.b2 <<endl;
}
```

答案是 -16 和 1

首先 sizeof(AA)的大小为 4,b1 和 b2 分别占 5bit 和 2bit.

经过 strcpy 和 memcpy 后,aa 的 4 个字节所存放的值是:

0,1,2,3 的 ASC 码，即 00110000,00110001,00110010,00110011

所以，最后一步：显示的是这 4 个字节的前 5 位，和之后的 2 位

分别为：10000,和 01 因为 int 是有正负之分　　所以：答案是-16 和 1

9)　分析下面的程序：

```
void GetMemory(char **p,int num)
{
*p=(char *)malloc(num);
}
int main()
{
char *str=NULL;
GetMemory(&str,100);
strcpy(str,"hello");
free(str);
if(str!=NULL)
{
strcpy(str,"world");
}
printf("\n str is %s",str);
getchar();
}
```

输出 str is world。

free 只是释放的 str 指向的内存空间,它本身的值还是存在的.

所以 free 之后，有一个好的习惯就是将 str=NULL.

此时 str 指向空间的内存已被回收,如果输出语句之前还存在分配空间的操作的话,这段存储空间是可能被重新分配给其他变量的,

尽管这段程序确实是存在大大的问题（上面各位已经说得很清楚了），但是通常会打印出 world 来。

这是因为，进程中的内存管理一般不是由操作系统完成的，而是由库函数自己完成的。

当你 malloc 一块内存的时候，管理库向操作系统申请一块空间（可能会比你申请的大一些），然后在这块空间中记录一些管理信息（一般是在你申请的内存前面一点），并将可用内存的地址返回。但是释放内存的时候，管理库通常都不会将内存还给操作系统，因此你是可以继续访问这块地址的 char a[10],strlen(a)

10）写出程序运行结果

```
int sum(int a)
{
auto int c=0;
static int b=3;
c+=1;
b+=2;
return(a+b+c);
}
void main()
{
int I;
int a=2;
for(I=0;I<5;I++)
{
printf("%d,", sum(a));
}
}
```

// static 会保存上次结果，记住这一点，剩下的自己写

输出：8,10,12,14,16,

11）请写出下列代码的输出内容

```
#include<stdio.h>
main()
{
int a,b,c,d;
a=10;
b=a++;
c=++a;
d=10*a++;
printf("b，c，d：%d，%d，%d", b，c，d）;
return 0;
}
```

答：10，12，120

12）写出下列代码的输出内容

```
#include<stdio.h>
int inc(int a)
{
```

```
return(++a);
}
int multi(int*a,int*b,int*c)
{
return(*c=*a**b);
}
typedef int(FUNC1)(int in);
typedef int(FUNC2) (int*,int*,int*); //函数指针，注意之 !!
void show(FUNC2 fun,int arg1, int*arg2)
{
INCp=&inc;
int temp =p(arg1);
fun(&temp,&arg1, arg2);
printf("%d\n",*arg2);
}
main()
{
int a;
show(multi,10,&a);
return 0;
}
```

答：110

13) 请找出下面代码中的所以错误 说明：以下代码是把一个字符串倒序，如“abcd”倒序后变为“dcba”

```
1、#include"string.h" //改成为 #include <string.h>
//缺少#include <stdio.h>
2、main() //int main()
3、{
4、 char*src="hello,world";
5、 char* dest=NULL;
6、 int len=strlen(src);
7、 dest=(char*)malloc(len); // dest=(char*)malloc(len+1);
8、 char* d=dest;
9、 char* s=src[len]; //char* s = &src[len-1];
10、 while(len--!=0) //while(len-- >= 0)
11、 d++=s--; // *d++=*s--
12、 printf("%s",dest); //别忘了释放空间 free(dest);
13、 return 0;
14、}
```

答：方法 1：

```
int main(){
char* src = "hello,world";
int len = strlen(src);
char* dest = (char*)malloc(len+1);//要为\0 分配一个空间
```

```
char* d = dest;
char* s = &src[len-1];//指向最后一个字符
while( len-- != 0 )
*d++=*s--;
*d = 0;//尾部要加\0
printf("%s\n",dest);
free(dest);// 使用完，应当释放空间，以免造成内存汇泄露
return 0;
}
```

方法 2：

```
#include <stdio.h>
#include <string.h>
main()
{
char str[]="hello,world";
int len=strlen(str);
char t;
for(int i=0; i<len/2; i++)
{
t=str[i];
str[i]=str[len-i-1]; str[len-i-1]=t;
}
printf("%s",str);
return 0;
}
```

14) 请问以下代码有什么问题：

```
int main()
{
char a;
char *str=&a;
strcpy(str,"hello");
printf(str);
return 0;
}
```

没有为 str 分配内存空间，将会发生异常

问题出在将一个字符串复制进一个字符变量指针所指地址。虽然可以正确输出结果，但因为越界进行内在读写而导致程序崩溃。

```
char* s="AAA";
printf("%s",s);
s[0]='B';
printf("%s",s);
```

有什么错？

"AAA"是字符串常量。s 是指针，指向这个字符串常量，所以声明 s 的时候就有问题。

```
cosnt char* s="AAA";
```

然后又因为是常量，所以对是 s[0]的赋值操作是不合法的。

15) #include <stdio.h>

```
#include <stdlib.h>
void getmemory(char *p)
{
p=(char *) malloc(100);
strcpy(p,"hello world");
}
int main( )
{
char *str=NULL;
getmemory(str);
printf("%s/n",str);
free(str);
return 0;
}
```

程序崩溃，getmemory 中的 malloc 不能返回动态内存， free（）对 str 操作很危险

B. 知识点精华

1. 变量的作用域

一个 C 程序在运行时，用户区可以分为动态存储区、静态存储区和程序区三大块。
程序区，用来存放 C 程序运行代码。
静态存储区，存储的变量被称作静态变量，如全局变量。
动态存储区，用来存放变量以及进行函数调用时的现场信息和函数返回地址等，在这个区域存储的变量称之为动态变量，如形参变量、函数体内部定义的局部变量等。
在 C 语言中，每一个变量都有两个属性：数据类型和存储类型。数据类型即常说的字符型、整型、浮点型；存储类型则指变量在内存中的存储方式，它决定了变量的作用域和生存期。
变量的存储类型有以下四种：auto（自动）、register（寄存器）、extern（外部）和 static（静态）。其中 auto 和 register 用于声明内部变量，auto 变量是存储在栈中的，register 变量是存储在寄存器中的。static 用于声明内部变量或外部变量，extern 用于声明外部变量，它们是存储在静态存储区的。
变量声明的一般形式：〈存储类型〉〈数据类型〉〈变量名列表〉
当声明变量时未指定存储类型，则内部变量的存储类型默认为 auto 型，外部变量的存储类型默认为 extern 型。
外部变量有两种声明方式：定义性声明和引用性声明。
定义性声明是为了创建变量，及变量需要分配内存。引用性声明是为了建立变量与内存单元之间的关系，表示要引用的变量已在程序源文件中其他地方进行过定义性声明。定义性声明只能放在函数外部，而引用性声明可放在函数外部，也可放在函数内部。

```
extern int b;//引用性声明，也可放在函数 fun 中
void fun()
{
    printf("d%",b);//输出
}
extern int b=5;//定义性声明，可以省略关键字 extern
```

变量内存有三种分配方式：静态分配、自动分配和动态分配。
系统可以为每个程序开辟一个固定的静态存储区，静态分配是指在这个固定的静态存储区为变量分配内存空间。对于静态分配的变量，在编译时就分配了内存地址（相对地址），在程序开始执行时变量就占用内存，直到程序结束时变量才释放内存。
程序运行后，系统将为程序开辟一块称为栈的活动存储区，栈按“后进先出”的方式使用内存空间。自动分配是指在栈（stack）中为变量临时分配内存空间。对于自动分配内存空间的变量（一般方式声明的变量，如[auto] int x,y=4;），程序运行后，在变量作用域开始时由系统自动为变量分配内存，在作用域结束后即释放内存。
动态分配是指利用一个称之为堆（heap）的内存块为变量分配内存空间，堆使用了静态存储区和栈之外的部分内存。动态分配是一种完全有程序本身控制内存的使用的分配方式。例如 C/C++中的 malloc/new 运算符分配内存，利用 free/delete 运算符释放内存，这样就能实现对内存的动态分配。
变量的作用域是指一个范围，是从代码空间的角度考虑问题，它决定了变量的

可见性，说明变量在程序的哪个区域可用，即程序中哪些行代码可以使用变量。作用域有三种：局部作用域、全局作用域、文件作用域，相对应于局部变量（local variable）、全局变量和静态变量（global variable）。

(1)局部变量

大部分变量具有局部作用域，它们声明在函数（包括 main 函数）内部，因此局部变量又称为内部变量。在语句块内部声明的变量仅在该语句块内部有效，也属于局部变量。局部变量的作用域开始于变量被声明的位置，并在标志该函数或块结束的右花括号处结束。函数的形参也具有局部作用域。

```
void MyFun1(int x)
{                   //形参 x 的作用域开始于此
    int y=3;        //局部变量 y 的作用域开始于此
    {
        int z=x+y;//块内部变量 z 的作用域开始于此，x 和 y 在该块内可用
        ……
    }               //块内部变量 z 的作用域结束
    int k;          //局部变量 k 的作用域开始于此
    ……
}                   //局部变量 y、k 和形参 x 的作用域结束
void MyFun2()
{
    int x=1;
    {
        int x=2;  //变量“x=1”失去作用域
        printf("x=%d\n",x); //输出 x=2
    }
}
```

(2) 全局变量及 extern 关键字

全局变量声明在函数的外部，因此又称外部变量，其作用域一般从变量声明的位置起，在程序源文件结束处结束。全局变量作用范围最广，甚至可以作用于组成该程序的所有源文件。当将多个独立编译的源文件链接成一个程序时，在某个文件中声明的全局或函数，在其他相链接的文件中也可以使用它们，但是用前必须进行 extern 外部声明。extern 可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。

以下是 MSDN 对 C/C++中 extern 关键字的解释：

The extern Storage-Class Specifier (C)

A variable declared with the extern storage-class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The internal extern declaration is used to make the external-level variable definition visible within the block. Unless otherwise declared at the external level, a variable declared with the extern keyword is visible only in the block in which it is declared.

The extern Storage-Class Specifier (C++)

The extern keyword declares a variable or function and specifies that it has external linkage (its name is visible from files other than the one in which it's defined).

When modifying a variable, extern specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends). The variable or function may be defined in another source file, or later in the same file. Declarations of variables and functions at file scope are external by default.

在 A.cpp 文件中，有一个全局变量 a,和一个函数：func();

```
//A.cpp 文件：
……
int a;   //全局变量 a 的作用域开始于此，结束与整个程序源文件
void func()
{
  ……
}
……
```

我们希望在 B.cpp 或更多其它文件可以使用到变量 a 和函数 func(),必须在“合适的位置”声明二者：

```
//B.cpp 文件：
……
extern int a;         //a 由另一源文件(A.cpp)定义
extern void func();  //func 由另一源文件(A.cpp)定义
a = 100;
func();
……
```

这里例子中，“合适的位置”是在 B.cpp 文件里。其它合适的位置，比如在头文件里的例子。需要强调的是函数的定义默认就是外部的,所以上面 func()之前的 extern 也可以省略。

在使用 extern 声明全局变量或函数时，一定要注意：所声明的变量或函数必须在，且仅在一个源文件中实现定义。如果你的程序声明了一个外部变量，但却没有在任何源文件中定义它，程序将可以通编译，但无法链接通过。

另外，extern 也可用来进行链接指定。C++中的 extern "C"声明是为了实现 C++与 C 及其它语言的混合编程，其中被 extern "C"修饰的变量和函数是按照 C 语言方式编译和连接的。如果 C++调用一个 C 语言编写的.DLL 时，当包括.DLL 的头文件或声明接口函数时，应加 extern "C" { }。

In C++, when used with a string, extern specifies that the linkage conventions of another language are being used for the declarator(s). C functions and data can be accessed only if they are previously declared as having C linkage. However, they must be defined in a separately compiled translation unit.

Microsoft C++ supports the strings "C" and "C++" in the string-literal field. All of the standard include files use the extern "C" syntax to allow the run-time library functions to be used in C++ programs.

The following example shows alternative ways to declare names that have C linkage:

```
// testExtern.cpp
// compile with: /c
// Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
// 包含 C 语言头文件
```

```
extern "C" {
   #include <stdio.h>
}
//  声明函数 ShowChar 和 GetChar 为按 C 方式编译链接函数
extern "C" {
   char ShowChar( char ch );
   char GetChar( void );
}
//调用已有 C 函数，定义函数 ShowChar 和 GetChar
extern "C" char ShowChar( char ch ) {
   putchar( ch );
   return ch;
}
extern "C" char GetChar( void ) {
   char ch;
   ch = getchar();
   return ch;
}
// Declare a global variable, errno, with C linkage.
extern "C" int errno;
```

(3) 静态变量及 static 关键字

文件作用域是指在函数外部声明的变量只在当前文件范围内（包括该文件内所有定义的函数）可用，但不能被其他文件中的函数访问。一般在具有文件作用域的变量或函数的声明前加上 static 修饰符。

static 静态变量可以是全局变量，也可以是局部变量，但都具有全局的生存周期，即生命周期从程序启动到程序结束时才终止。

```
#include <stdio.h>
void fun()
{
    static int a=5;//静态变量 a 是局部变量，但具有全局的生存期
    a++;
    printf("a=%d\n",a);
}
void main()
{
    int i;
    for(i=0;i<2;i++)
    fun();
    getchar();
}
```

输出结果为：

```
a=6
a=7
```

static 操作符后面生命的变量其生命周期是全局的，而且其定义语句即 static int a=5;

只运行一次，因此之后再调用 fun()时，该语句不运行。所以 f 的值保留上次计算所得,因此是 6,7.

由于静态变量或静态函数只在当前文件（定义它的文件）中有效，所以我们完全可以在多个文件中，定义两个或多个同名的静态变量或函数。这样当将多个独立编译的源文件链接成一个程序时，static 修饰符避免一个文件中的外部变量由于与其他文件中的变量同名而发生冲突。

比如在 A 文件和 B 文件中分别定义两个静态变量 a：

A 文件中：static int a;

B 文件中：static int a;

这两个变量完全独立，之间没有任何关系，占用各自的内存地址。你在 A 文件中改 a 的值，不会影响 B 文件中那个 a 的值。

2 变量

可以分为：全局变量、静态全局变量、静态局部变量和局部变量。

按存储区域分，全局变量、静态全局变量和静态局部变量都存放在内存的静态存储区域，局部变量存放在内存的栈区。

按作用域分，全局变量在整个工程文件内都有效；静态全局变量只在定义它的文件内有效；静态局部变量只在定义它的函数内有效，只是程序仅分配一次内存，函数返回后，该变量不会消失；局部变量在定义它的函数内有效，但是函数返回后失效。

全局变量和静态变量如果没有手工初始化，则由编译器初始化为 0。局部变量的值不可知。当编译一个 C++程序时，计算机的内存被分成了 4 个区域，一个包括程 序的代码，一个包括所有的全局变量，一个是堆栈，还有一个是堆（heap）,我们称堆是自由的内存区域，我们可以通过 new 和 delete 把对象放在这个 区域。你可以在任何地方分配和释放自由存储区。

局部静态变量

在局部变量之前加上关键字 static，局部变量就被定义成为一个局部静态变量。

1）内存中的位置：静态存储区

2）初始化：未经初始化的全局静态变量会被程序自动初始化为 0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域随之结束。

[注]当 static 用来修饰局部变量的时候，它就改变了局部变量的存储位置，从原来的栈中存放改为静态存储区。但是局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问。

全局静态变量

在全局变量之前加上关键字 static，全局变量就被定义成为一个全局静态变量。

1）内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）

2）初始化：未经初始化的全局静态变量会被程序自动初始化为 0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：全局静态变量在声明他的文件之外是不可见的。准确地讲从定义之处开始到文件结尾。

[注]当 static 用来修饰全局变量的时候，它就改变了全局变量的作用域（在声明他的文件之外是不可见的），但是没有改变它的存放位置，还是在静态存储区中。

3 函数

静态函数

在函数的返回类型前加上关键字 static，函数就被定义成为静态函数。
函数的定义和声明默认情况下是 extern 的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。
静态数据成员

1） 内存中的位置：静态存储区

2） 初始化和定义：

〈1〉静态数据成员定义时要分配空间，所以不能在类声明中定义。

〈2〉静态数据成员在程序中只能提供一个定义，所以静态数据成员的初始化不能在类的头文件中。

3) 访问：

〈1〉类对象名.静态数据成员

〈2〉类类型名::静态数据成员

4） 说明：

a. static 数据成员和普通数据成员一样遵 public, protected, private 访问规则。

b. 对于非静态数据成员，每个类对象都有自己的拷贝。静态数据成员被当作类的全局对象，无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份拷贝，由该类类型的所有对象共享访问。

5） 同全局对象相比，使用静态数据成员有两个优势：

〈1〉静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其他全局名字冲突的可能性。

〈2〉可以实现信息隐藏。静态成员可以是 private 成员，而全局对象不能。

6 ） 静态数据成员的“唯一性”本质（独立于类的任何对象而存在的唯一实例），使他能够以独特的方式被使用，这些方式对于非 static 数据成员来说是非法的。

静态成员函数

1） 声明：在类的成员函数返回值之前加上关键字 static，他就被声明为一个静态成员函数。静态成员函数不能声明为 const 或 volatile，这与非静态成员函数不同。

2） 定义：出现在类体外的函数定义不能指定关键字 static。

3） 作用：主要用于对静态数据成员的操作

4） 静态成员函数与类相联系，不与类的对象相联系。

5） 静态成员函数不能访问非静态数据成员。因为非静态数据成员属于特定的类实例。

6） 静态成员函数没有 this 指针，因此在静态成员函数中隐式或显示的引用这个指针都将导致编译时刻错误。试图访问隐式引用 this 指针的非静态数据成员也会导致编译时刻错误。

7） 访问：可以用成员访问操作符(.)和箭头(->)为一个类对象或指向类对象的指针调用静态成员函数，也可以用限定修饰符名直接访问或调用静态成员函数，而无需声明类对象。

2. 位域

有些信息在存储时，并不需要占用一个完整的字节， 而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态， 用一位二进位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为“位域”或“位段”。所谓“位域”是把一个字节中的二进位划分为几个不同的区域， 并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。 这样就可以把几个不同的对象用一个字节的二进制位域来表示。一、位域的定义和位域变量的说明位域定义与结构定义相仿，其形式为：

```
struct 位域结构名
{ 位域列表 };
```
其中位域列表的形式为： 类型说明符 位域名：位域长度
例如：
```
struct bs
{
    int a:8;
    int b:2;
    int c:6;
};
```
位域变量的说明与结构变量说明的方式相同。 可采用先定义后说明，同时定义说明或者直接说明这三种方式。例如：
```
struct bs
{
    int a:8;
    int b:2;
    int c:6;
}data;
```
说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。
对于位域的定义尚有以下几点说明：
1. 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：
```
struct bs
{
    unsigned a:4
    unsigned :0 /*空域*/
    unsigned b:4 /*从下一单元开始存放*/
unsigned c:4
}
```
在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。
2. 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进位。
3. 位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：
```
struct k
{
    int a:1
    int :2 /*该 2 位不能使用*/
    int b:3
    int c:2
};
```
从以上分析可以看出，位域在本质上就是一种结构类型， 不过其成员是按二进位分配的。
二、位域的使用位域的使用和结构成员的使用相同，其一般形式为： 位域变量名•位域名 位域允许用各种格式输出。

```
main(){
struct bs
{
  unsigned a:1;
  unsigned b:3;
  unsigned c:4;
} bit,*pbit;
bit.a=1;
bit.b=7;
bit.c=15;
pri
```

3. 用 C 连接数据库

```
// test.cpp : Defines the entry point for the console application.
//
#define DBNTWIN32
#include <stdio.h>
#include <windows.h>
#include <sqlfront.h>
#include <sqldb.h>
int err_handler(PDBPROCESS, INT, INT, INT, LPCSTR, LPCSTR);
int      msg_handler(PDBPROCESS,       DBINT,       INT,       INT,       LPCSTR,
LPCSTR,                  LPCSTR, DBUSMALLINT);
main()
{
     PDBPROCESS   dbproc;     // The connection with SQL Server.
     PLOGINREC    login;      // The login information.
     DBCHAR       name[100];
     DBCHAR       city[100];
     // Install user-supplied error- and message-handling functions.
     dberrhandle (err_handler);
     dbmsghandle (msg_handler);
     // Initialize DB-Library.
     dbinit ();
     // Get a LOGINREC.
     login = dblogin ();
     DBSETLUSER (login, "sa");
     DBSETLPWD (login, "carso");
     DBSETLAPP (login, "pubs");
     // Get a DBPROCESS structure for communication with SQL Server.
     dbproc = dbopen (login, ".");
```

```
    // Retrieve some columns from the "authors" table in the
    // "pubs" database.
    // First, put the command into the command buffer.
    dbcmd (dbproc, "select au_lname, city from pubs..authors");
    dbcmd (dbproc, " where state = 'CA' ");
    // Send the command to SQL Server and start execution.
    dbsqlexec (dbproc);
    // Process the results.
    if (dbresults (dbproc) == SUCCEED)
    {
        // Bind column to program variables.
        dbbind (dbproc, 1, NTBSTRINGBIND, 0,(LPBYTE)name);
        dbbind (dbproc, 2, NTBSTRINGBIND, 0,(unsigned char *)city);
        // Retrieve and print the result rows.
        while (dbnextrow (dbproc) != NO_MORE_ROWS)
        {
            printf ("%s from %s\n", name, city);
        }
    }
    // Close the connection to SQL Server.
    dbexit ();
    return (0);
}
int err_handler (PDBPROCESS dbproc, INT severity,      INT dberr, INT oserr, LPCSTR
dberrstr, LPCSTR oserrstr)
{
    printf ("DB-Library Error %i: %s\n", dberr, dberrstr);
    if (oserr != DBNOERR)
    {
        printf ("Operating System Error %i: %s\n", oserr, oserrstr);
    }
    return (INT_CANCEL);
}
int msg_handler (PDBPROCESS dbproc, DBINT msgno, INT msgstate, INT severity, LPCSTR
msgtext, LPCSTR server, LPCSTR procedure, DBUSMALLINT line)
{
    printf ("SQL Server Message %ld: %s\n", msgno, msgtext);
    return (0);
}
```

4. 存储分配

在 C++中，内存分成 5 个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存

储区。

栈，就是那些由编译器在需要的时候分配，在不需要的时候自动清楚的变量的存储区。里面的变量通常是局部变量、函数参数等。

堆，就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

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

全局/静态存储区，全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量又分为初始化的和未初始化的，在 C++里面没有这个区分了，他们共同占用同一块内存区。

常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改，而且方法很多，在《const 的思考》一文中，我给出了 6 种方法）

5. 函数指针

函数指针是指向函数的指针变量。

因而“函数指针”本身首先应是指针变量，只不过该指针变量指向函数。这正如用指针变量可指向整型变量、字符型、数组一样，这里是指向函数。如前所述，C 在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，在这些概念上一致的。函数指针有两个用途：调用函数和做函数的参数。函数指针的说明方法为:

数据类型标志符 （*指针变量名）(参数);

注：函数括号中的参数可有可无，视情况而定。

下面的程序说明了函数指针调用函数的方法：

```
#include
int max(int x,int y){ return(x>y?x:y); }
void main()
{
int (*ptr)();
int a,b,c;
ptr=max;
scanf("%d,%d",&a,&b);
c=(*ptr)(a,b);
printf("a=%d,b=%d,max=%d",a,b,c);
}
```

ptr 是指向函数的指针变量，所以可把函数 max()赋给 ptr 作为 ptr 的值，即把 max()的入口地址赋给 ptr,以后就可以用 ptr 来调用该函数，实际上 ptr 和 max 都指向同一个入口地址，不同就是 ptr 是一个指针变量，不像函数名称那样是死的，它可以指向任何函数，就看你像怎么做了。在程序中把哪个函数的地址赋给它，它就指向哪个函数。而后用指针变量调用它，因此可以先后指向不同的函数，不过注意，指向函数的指针变量没有++和--运算，用时要小心。

不过，在某些编译器中这是不能通过的。这个例子的补充如下。

应该是这样的：

1.定义函数指针类型：

```
typedef int (*fun_ptr)(int,int);
```

2.申明变量，赋值:

```
fun_ptr max_func=max;
```

也就是说，赋给函数指针的函数应该和函数指针所指的函数原型是一致的。

Pro *C/C++学习笔记(Pro *C/C++的编译环境的设置)

6. extern 使用方法总结

```
#i nclude "stdafx.h"
```

1.extern 用在变量声明中常常有这样一个作用，你在*.c 文件中声明了一个全局的变量，这个全局的变量如果要被引用，就放在*.h 中并用 extern 来声明。

2.如果函数的声明中带有关键字 extern，仅仅是暗示这个函数可能在别的源文件里定义，没有其它作用。即下述两个函数声明没有区别：

```
extern int f(); 和 int f();
```

==========

如果定义函数的 c/cpp 文件在对应的头文件中声明了定义的函数，那么在其他 c/cpp 文件中要使用这些函数，只需要包含这个头文件即可。

如果你不想包含头文件，那么在 c/cpp 中声明该函数。一般来说，声明定义在本文件的函数不用“extern”，声明定义在其他文件中的函数用“extern”，这样在本文件中调用别的文件定义的函数就不用包含头文件

include “*.h”来声明函数，声明后直接使用即可。

举个例子：

```
//extern.cpp 内容如下
#i nclude "stdafx.h"
extern print(char *p);
int main(int argc, char* argv[])
{
 char *p="hello world!";
 print(p);
 return 0;
}
//print.cpp 内容如下
#i nclude "stdafx.h"
#i nclude "stdio.h"
print(char *s)
{
 printf("The string is %s\n",s);
}
```

结果程序可以正常运行，输出结果。如果把“extern”去掉，程序依然可以正常运行。

由此可见，“extern”在函数声明中可有可无，只是用来标志该函数在本文件中定义，还是在别的文件中定义。只要你函数在使用之前声明了，那么就可以不用包含头文件了。

extern "C" 链接指示符 extern C

如果程序员希望调用其他程序设计语言尤其是 C 写的函数那么调用函数时必须告诉编译器使用不同的要求例如当这样的函数被调用时函数名或参数排列的顺序可能不同无论是 C++函数调用它还是用其他语言写的函数调用它

程序员用链接指示符 linkage directive 告诉编译器该函数是用其他的程序设计语言编写的链接指示符有两种形式既可以是单一语句 single statement 形式也可以是复合语句 compound statement 形式

```
// 单一语句形式的链接指示符
extern "C" void exit(int);
// 复合语句形式的链接指示符
extern "C" {
int printf( const char* ... );
int scanf( const char* ... );
}
// 复合语句形式的链接指示符
extern "C" {
#include <cmath>
}
```

链接指示符的第一种形式由关键字 extern 后跟一个字符串常量以及一个普通的函数声明构成虽然函数是用另外一种语言编写的但调用它仍然需要类型检查例如编译器会检查传递给函数 exit()的实参的类型是否是 int 或者能够隐式地转换成 int 型多个函数声明可以用花括号包含在链接指示符复合语句中这是链接指示符的第二种形式花招号被用作分割符表示链接指示符应用在哪些声明上在其他意义上该花括号被忽略所以在花括号中声明的函数名对外是可见的就好像函数是在复合语句外声明的一样例如在前面的例子中复合语句 extern "C"表示函数 printf()和 scanf()是在 C 语言中写的函数因此这个声明的意义就如同 printf()和 scanf()是在 extern "C"复合语句外面声明的一样当复合语句链接指示符的括号中含有#include 时在头文件中的函数声明都被假定是用链接指示符的程序设计语言所写的在前面的例子中在头文件<cmath>中声明的函数都是 C 函数链接指示符不能出现在函数体中下列代码段将会导致编译错误

```
int main()
{
// 错误: 链接指示符不能出现在函数内
extern "C" double sqrt( double );
```

如果我们希望 C++函数能够为 C 程序所用又该怎么办呢我们也可以使用 extern "C" 链接指示符来使 C++函数为 C 程序可用例如

```
// 函数 calc() 可以被 C 程序调用
extern "C" double calc( double dparm ) { /* ... */ }
```

如果一个函数在同一文件中不只被声明一次则链接指示符可以出现在每个声明中它也可以只出现在函数的第一次声明中在这种情况下第二个及以后的声明都接受第一个声明中链接指示符指定的链接规则例如

```
// ---- myMath.h ----
extern "C" double calc( double );
```

```
// ---- myMath.C ----
// 在 Math.h 中的 calc() 的声明
#include "myMath.h"
// 定义了 extern "C" calc() 函数
// calc() 可以从 C 程序中被调用
double calc( double dparm ) { // ...
```

在本节中我们只看到为 C 语言提供的链接指示 extern "C" extern "C"是惟一被保证由所有 C++实现都支持的每个编译器实现都可以为其环境下常用的语言提供其他链接指示例如 extern "Ada"可以用来声明是用 Ada 语言写的函数 extern "FORTRAN"用来声明是用 FORTRAN 语言写的函数等等因为其他的链接指示随着具体实现的不同而不同

7. virtual 关键字

一、 基本概念

首先，C++通过虚函数实现多态."无论发送消息的对象属于什么类，它们均发送具有同一形式的消息，对消息的处理方式可能随接手消息的对象而变"的处理方式被称为多态性。"在某个基类上建立起来的类的层次构造中，可以对任何一个派生类的对象中的同名过程进行调用，而被调用的过程提供的处理可以随其所属的类而变。"虚函数首先是一种成员函数，它可以在该类的派生类中被重新定义并被赋予另外一种处理功能。二、 虚函数的定义与派生类中的重定义

```
class 类名{
public:
virtual 成员函数说明;
}
class 类名：基类名{
public:
virtual 成员函数说明;
}
```

三、 虚函数在内存中的结构

1．我们先看一个例子：

```
#include "iostream.h"
#include "string.h"
class A {
public:
virtual void fun0() { cout << "A::fun0" << endl; }
};
int main(int argc, char* argv[])
{
A a;
cout << "Size of A = " << sizeof(a) << endl;
return 0;
}
```

结果如下：Size of A = 4

2．如果再添加一个虚函数：virtual void fun1() { cout << "A::fun" << endl;}
得到相同的结果。如果去掉函数前面的 virtual 修饰符 class A {
public:
void fun0() { cout << "A::fun0" << endl; }
};
int main(int argc, char* argv[])
{
A a;
cout << "Size of A = " << sizeof(a) << endl;
return 0;
}
结果如下：Size of A = 1
3．在看下面的结果： class A {
public:
virtual void fun0() { cout << "A::fun0" << endl; }
int a;
int b;
};
int main(int argc, char* argv[])
{
A a;
cout << "Size of A = " << sizeof(a) << endl;
return 0;
}
结果如下：Size of A = 12
在 window2000 下指针在内存中占 4 个字节，虚函数在一个虚函数表(VTABLE)中保存函数地址。在看下面例子。 class A {
public:
virtual void fun0() { cout << "A::fun0" << endl; }
virtual void fun1() { cout << "A::fun1" << endl; }
int a;
int b;
};
int main(int argc, char* argv[])
{
A a;
cout << "Size of A = " << sizeof(a) << endl;
return 0;
}
结果如下：结果如下：
Size of A = 4
虚函数的内存结构如下，你也可以通过函数指针，先找到虚函数表(VTABLE),然后访问每个函数地址来验证这种结构，在国外网站作者是：Zeeshan Amjad 写的"ATL on the Hood 中有详细介绍"

4．我们再来看看继承中虚函数的内存结构，先看下面的例子 class A {
public:
virtual void f() { }
};
class B {
public:
virtual void f() { }
};
class C {
public:
virtual void f() { }
};
class Drive : public A, public B, public C {
};
int main() {
Drive d;
cout << "Size is = " << sizeof(d) << endl;
return 0;
}
结果如下：Size is = 12
5.我们再来看看用虚函数实现多态性，先看个例子：class A {
public:
virtual void f() { cout << "A::f" << endl; }
};
class B :public A{
public:
virtual void f() { cout << "B::f" << endl;}
};
class C :public A {
public:
virtual void f() { cout << "C::f" << endl;}
};
class Drive : public C {
public:
virtual void f() { cout << "D::f" << endl;}
};
int main(int argc, char* argv[])
{
A a;
B b;
C c;
Drive d;
a.f();
b.f();

```
c.f();
d.f();
return 0;
}
```
结果：A::f
B::f
C::f
D::f
不用解释，相信大家一看就明白什么道理！注意：多态不是函数重载
6.用虚函数实现动态连接在编译期间，C++编译器根据程序传递给函数的参数或者函数返回类型来决定程序使用那个函数，然后编译器用正确的的函数替换每次启动。这种基于编译器的替换被称为静态连接，他们在程序运行之前执行。另一方面，当程序执行多态性时，替换是在程序执行期进行的，这种运行期间替换被称为动态连接。如下例子：
```
class A{
public:
virtual void f(){cout << "A::f" << endl;};
};
class B:public A{
public:
virtual void f(){cout << "B::f" << endl;};
};
class C:public A{
public:
virtual void f(){cout << "C::f" << endl;};
};
void test(A *a){
a->f();
};
int main(int argc, char* argv[])
{
B *b=new B;
C *c=new C;
char choice;
do{
cout<<"type B for class B,C for class C:"<<endl;
cin>>choice;
if(choice==''b'')
test(b);
else if(choice==''c'')
test(c);
}while(1);
cout<<endl<<endl;
return 0;
}
```
在上面的例子中，如果把类 A,B,C 中的 virtual 修饰符去掉，看看打印的结果，然后再看下

面一个例子想想两者的联系。如果把 B 和 C 中的 virtual 修饰符去掉，又会怎样，结果和没有去掉一样。

7．在基类中调用继承类的函数（如果此函数是虚函数才能如此）还是先看例子： class A {

```
virtual void fun() {
cout << "A::fun" << endl;
}
void show() {
fun();
}
};
class B : public A {
public:
virtual void fun() {
cout << "B::fun" << endl;
}
};
int main() {
A a;
a.show();
return 0;
}
```

打印结果：A::fun

在 6 中的例子中，test(A *a)其实有一个继承类指针向基类指针隐式转化的过程。可以看出利用虚函数我们可以在基类调用继承类函数。但如果不是虚函数，继承类指针转化为基类指针后只可以调用基类函数。反之，如果基类指针向继承类指针转化的情况怎样，这只能进行显示转化，转化后的继承类指针可以调用基类和继承类指针。如下例子： class A {

```
void fun() {
cout << "A::fun" << endl;
}
};
class B : public A {
public:
void fun() {
cout << "B::fun" << endl;
}
void fun0() {
cout << "B::fun0" << endl;
}
};
int main() {
A *a=new A;
B *b=new B;
A *pa;
B *pb;
```

```
pb=static_cast<B *>(a); //基类指针向继承类指针进行显示转化
pb->fun0();
pb->fun();
return 0;
}
```

8. itoa 和 atoi 函数 c 语言实现

函数名: itoa
功能: 把整型数转换成字符串
用 法: void itoa(unsigned int val,char *buf,int is_neg)
程序:

```
void itoa (
unsigned int val,
char *buf,
int is_neg
)
{
char *p;
char *firstdig;
char temp;
unsigned digval;
p = buf;
if (is_neg) {
*p++ = '-';
val = (unsigned int)(-(int)val);
}
firstdig = p;
do {
digval = (unsigned) (val % 10);
val /= 10;
*p++ = (char) (digval + '0');
} while (val > 0);


*p-- = '\0';
do {
temp = *p;
*p = *firstdig;
*firstdig = temp;
--p;
++firstdig;
```

```
} while (firstdig < p);
}
```

函数名: atoi
功能: 把字符串转换成整型数
用 法: int atoi(const char * str);
程序:

```
int atoi(const char * nptr)
{
 int total = 0;
 int c;
 int sign;
 while (isspace((int)(unsigned char)*nptr))
        ++nptr;
 c =(int)(unsigned char)*nptr++;
    sign=c;
    if (c == '-' || c == '+')
        c = (int)(unsigned char)*nptr++;
    total = 0;
    while (isdigit(c))
 {
  total = 10*total + (c - '0');
  c = (int)(unsigned char)*nptr++;
 }
 if (sign == '-')
  return   -total;
 else
  return   total;
}
```

第三部分 JAVA

A. 笔试面试题集

1. 面向对象的特征有哪些方面

（1.抽象：

抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用部分细节。抽象包括两个方面，一是过程抽象，二是数据抽象。

（2.继承：

继承是一种联结类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。新类继承了原始类的特性，新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。派生类可以从它的基类那里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。

（3.封装：

封装是把过程和数据包围起来，对数据的访问只能通过已定义的界面。面向对象计算始于这个基本概念，即现实世界可以被描绘成一系列完全自治、封装的对象，这些对象通过一个受保护的接口访问其他对象。

（4. 多态性：

多态性是指允许不同类的对象对同一消息作出响应。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好的解决了应用程序函数同名问题。

2. String 是最基本的数据类型吗?

基本数据类型包括 byte、int、char、long、float、double、boolean 和 short。

java.lang.String 类是 final 类型的，因此不可以继承这个类、不能修改这个类。为了提高效率节省空间，我们应该用 StringBuffer 类

3. int 和 Integer 有什么区别

Java 提供两种不同的类型：引用类型和原始类型（或内置类型）。Int 是 java 的原始数据类型，Integer 是 java 为 int 提供的封装类。Java 为每个原始类型提供了封装类。

原始类型封装类

boolean Boolean

char Character

byte Byte

short Short

int Integer

long Long

float Float

double Double

引用类型和原始类型的行为完全不同，并且它们具有不同的语义。引用类型和原始类型具有不同的特征和用法，它们包括：大小和速度问题，这种类型以哪种类型的数据结构存储，当引用类型和原始类型用作某个类的实例数据时所指定的缺省值。对象引用实例变量的缺省值为 null，而原始类型实例变量的缺省值与它们的类型有关。

4. String 和 StringBuffer 的区别

AVA 平台提供了两个类：String 和 StringBuffer，它们可以储存和操作字符串，即包含多个字符的字符数据。这个 String 类提供了数值不可改变的字符串。而这个 StringBuffer 类提供的字符串进行修改。当你知道字符数据要改变的时候你就可以使用 StringBuffer。典型地，你可以使用 StringBuffers 来动态构造字符数据。

5. 运行时异常与一般异常有何异同?

异常表示程序运行过程中可能出现的非正常状态,运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误。java 编译器要求方法必须声明抛出可能发生的非运行时异常，但是并不要求必须声明抛出未被捕获的运行时异常。

6. 说出 Servlet 的生命周期，并说出 Servlet 和 CGI 的区别。

Servlet 被服务器实例化后，容器运行其 init 方法，请求到达时运行其 service 方法，service 方法自动派遣运行与请求对应的 doXXX 方法（doGet，doPost）等，当服务器决定将实例销毁的时候调用其 destroy 方法。

与 cgi 的区别在于 servlet 处于服务器进程中,它通过多线程方式运行其 service 方法，一个实例可以服务于多个请求，并且其实例一般不会销毁，而 CGI 对每个请求都产生新的进程，服务完成后就销毁，所以效率上低于 servlet。

7. 说出 ArrayList,Vector, LinkedList 的存储性能和特性

ArrayList 和 Vector 都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector 由于使用了 synchronized 方法（线程安全），通常性能上较 ArrayList 差，而 LinkedList 使用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

8. EJB 是基于哪些技术实现的？并说出 SessionBean 和 EntityBean 的区别,StatefulBean 和 StatelessBean 的区别。

EJB 包括 Session Bean、Entity Bean、Message Driven Bean，基于 JNDI、RMI、JAT 等技术实现。

SessionBean 在 J2EE 应用程序中被用来完成一些服务器端的业务操作，例如访问数据库、调用其他 EJB 组件。EntityBean 被用来代表应用系统中用到的数据。

对于客户机，SessionBean 是一种非持久性对象，它实现某些在服务器上运行的业务逻辑。对于客户机，EntityBean 是一种持久性对象，它代表一个存储在持久性存储器中的实体的对象视图，或是一个由现有企业应用程序实现的实体。

Session Bean 还可以再细分为 Stateful Session Bean 与 Stateless Session Bean，这两种的 Session Bean 都可以将系统逻辑放在 method 之中执行，不同的是 Stateful Session Bean 可以记录呼叫者的状态，因此通常来说，一个使用者会有一个相对应的

Stateful Session Bean 的实体。Stateless Session Bean 虽然也是逻辑组件，但是他却不负责记录使用者状态，也就是说当使用者呼叫 Stateless Session Bean 的时候，EJB Container 并不会找寻特定的 Stateless Session Bean 的实体来执行这个 method。换言之，很可能数个使用者在执行某个 Stateless Session Bean 的 methods 时，会是同一个 Bean 的 Instance 在执行。从内存方面来看，Stateful Session Bean 与 Stateless Session Bean 比较，Stateful Session Bean 会消耗 J2EE Server 较多的内存，然而 Stateful Session Bean 的优势却在于他可以维持使用者的状态。

9. Collection 和 Collections 的区别。

Collection 是集合类的上级接口，继承与他的接口主要有 Set 和 List.

Collections 是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

10. &和&&的区别。

&是位运算符，表示按位与运算，&&是逻辑运算符，表示逻辑与（and）。

11. HashMap 和 Hashtable 的区别。

HashMap 是 Hashtable 的轻量级实现（非线程安全的实现），他们都完成了 Map 接口，主要区别在于 HashMap 允许空（null）键值（key）,由于非线程安全，效率上可能高于 Hashtable。

HashMap 允许将 null 作为一个 entry 的 key 或者 value，而 Hashtable 不允许。HashMap 把 Hashtable 的 contains 方法去掉了，改成 containsvalue 和 containsKey。因为 contains 方法容易让人引起误解。 Hashtable 继承自 Dictionary 类，而 HashMap 是 Java1.2 引进的 Map interface 的一个实现。最大的不同是，Hashtable 的方法是 Synchronize 的，而 HashMap 不是，在多个线程访问 Hashtable 时，不需要自己为它的方法实现同步，而 HashMap 就必须为之提供外同步。Hashtable 和 HashMap 采用的 hash/rehash 算法都大概一样，所以性能不会有很大的差异。

12. final, finally, finalize 的区别。

final 用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承。

finally 是异常处理语句结构的一部分，表示总是执行。

Finalize 是 Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。

13. sleep() 和 wait() 有什么区别?

sleep 是线程类（Thread）的方法，导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复。调用 sleep 不会释放对象锁。

wait 是 Object 类的方法，对此对象调用 wait 方法导致本线程放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象发出 notify 方法（或 notifyAll）后本线程才 进入对象锁定池准备获得对象锁进入运行状态。

14. Overload 和 Override 的区别。Overloaded 的方法是否可以改变返回值的类型?

方法的重写 Overriding 和重载 Overloading 是 Java 多态性的不同表现。重写 Overriding 是父类与子类之间多态性的一种表现，重载 Overloading 是一个类中多态

性的一种表现。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被"屏蔽"了。如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载(Overloading)。Overloaded 的方 法是可以改变返回值的类型。

15. error 和 exception 有什么区别?

error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。

exception 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况。

16. 同步和异步有何异同，在什么情况下分别使用他们？举例说明。

如果数据将在线程间共享。例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了,那么这些数据就是共享数据,必须进行同步存取。

当应用程序在对象上调用了一个需要花费很长时间来执行的方法,并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。

17. abstract class 和 interface 有什么区别?

声明方法的存在而不去实现它的类被叫做抽象类（abstract class），它用于要创建一个体现某些基本行为的类，并为该类声明方法，但不能在该类中实现该类的情况。不能创建 abstract 类的实例。然而可以创建一个变量，其类型是一个抽象类，并让它指向 具体子类的一个实例。不能有抽象构造函数或抽象静态方法。Abstract 类的子类为它 们父类中的所有抽象方法提供实现，否则它们也是抽象类为。取而代之，在子类中实现 该方法。知道其行为的其它类可以在类中实现这些方法。

接口（interface）是抽象类的变体。在接口中，所有方法都是抽象的。多继承性可通过实现这样的接口而获得。接口中的所有方法都是抽象的，没有一个有程序体。接口只可以定义 static final 成员变量。接口的实现与子类相似，除了该实现类不能从接口 定义中继承行为。当类实现特殊接口时，它定义（即将程序体给予）所有这种接口的方 法。然后，它可以在实现了该接口的类的任何对象上调用接口的方法。由于有抽象类， 它允许使用接口名作为引用变量的类型。通常的动态联编将生效。引用可以转换到接口 类型或从接口类型转换，instanceof 运算符可以用来决定某对象的类是否实现了接口。

18. heap 和 stack 有什么区别。

栈是一种线形集合，其添加和删除元素的操作应在同一段完成。栈按照后进先出的方式进行处理。

堆是栈的一个组成元素

19. forward 和 redirect 的区别

forward 是服务器请求资源，服务器直接访问目标地址的 URL，把那个 URL 的响应内容读取过来，然后把这些内容再发给浏览器，浏览器根本不知道服务器发送的内容是从哪儿来的，所以它的地址栏中还是原来的地址。

redirect 就是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址，一般来说浏览器会用刚才请求的所有参数重新请求，所以 session,request 参数都可以获

取。

20. EJB 与 JAVA BEAN 的区别？

Java Bean 是可复用的组件，对 Java Bean 并没有严格的规范，理论上讲，任何一个 Java 类都可以是一个 Bean。但通常情况下，由于 Java Bean 是被容器所创建(如 Tomcat)的，所以 Java Bean 应具有一个无参的构造器，另外，通常 Java Bean 还要实现 Serializable 接口用于实现 Bean 的持久性。Java Bean 实际上相当于微软 COM 模型中的本地进程内 COM 组件，它是不能被跨进程访问的。Enterprise Java Bean 相当于 DCOM，即分布式组件。它是基于 Java 的远程方法调用（RMI）技术的，所以 EJB 可以被远程访问(跨进程、跨计算机)。但 EJB 必须被布署在诸如 Webspere、WebLogic 这样的容器中，EJB 客户从不直接访问真正的 EJB 组件，而是通过其容器访问。EJB 容器是 EJB 组件的代理，EJB 组件由容器所创建和管理。客户通过容器来访问真正的 EJB 组件。

21. Static Nested Class 和 Inner Class 的不同。

Static Nested Class 是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化。

22. JSP 中动态 INCLUDE 与静态 INCLUDE 的区别？

动态 INCLUDE 用 jsp:include 动作实现 <jsp:include page="included.jsp" flush="true" />它总是会检查所含文件中的变化，适合用于包含动态页面，并且可以带参数。

静态 INCLUDE 用 include 伪码实现,定不会检查所含文件的变化，适用于包含静态页面 <%@ include file="included.htm" %>

23. 什么时候用 assert。

assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。在实现中，assertion 就是在程序中的一条语句，它对一个 boolean 表达式进行检查，一个正确程序必须保证这个 boolean 表达式的值为 true；如果该值为 false，说明程序已经处于不正确的状态下，系统将给出警告或退出。一般来说，assertion 用于保证程序最基本、关键的正确性。assertion 检查通常在开发和测试时开启。为了提高性能，在软件发布后，assertion 检查通常是关闭的。

24. GC 是什么? 为什么要有 GC?

GC 是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

25. short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 += 1;有什么错?

short s1 = 1; s1 = s1 + 1; （s1+1 运算结果是 int 型，需要强制转换类型）

short s1 = 1; s1 += 1;（可以正确编译）

26. Math.round(11.5)等於多少? Math.round(-11.5)等於多少?

Math.round(11.5)==12

Math.round(-11.5)==-11
round 方法返回与参数最接近的长整数，参数加 1/2 后求其 floor.

27. String s = new String("xyz");创建了几个 String Object?
两个，一个字符对象，一个字符对象引用对象

28. 设计 4 个线程，其中两个线程每次对 j 增加 1，另外两个线程对 j 每次减少 1。写出程序。

```
public class ThreadTest1{
private int j;
public static void main(String args[]){
ThreadTest1 tt=new ThreadTest1();
Inc inc=tt.new Inc();
Dec dec=tt.new Dec();
for(int i=0;i<2;i++){
Thread t=new Thread(inc);
t.start();
t=new Thread(dec);
t.start();
}
}
private synchronized void inc(){
j++;
System.out.println(Thread.currentThread().getName()+"-inc:"+j);
}
private synchronized void dec(){
j--;
System.out.println(Thread.currentThread().getName()+"-dec:"+j);
}
class Inc implements Runnable{
public void run(){
for(int i=0;i<100;i++){
inc();
}
}
}
class Dec implements Runnable{
public void run(){
for(int i=0;i<100;i++){
dec();
}
}
}
}
```

29. Java 有没有 goto?

java 中的保留字，现在没有在 java 中使用。

30. 启动一个线程是用 run()还是 start()？

启动一个线程是调用 start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行。这并不意味着线程就会立即运行。run()方法可以产生必须退出的标志来停止一个线程。

31. EJB 包括（SessionBean,EntityBean）说出他们的生命周期，及如何管理事务的？

SessionBean：Stateless Session Bean 的生命周期是由容器决定的，当客户机发出请求要建立一个 Bean 的实例时，EJB 容器不一定要创建一个新的 Bean 的实例供客户机调用，而是随便找一个现有的实例提供给客户机。当客户机第一次调用一个 Stateful Session Bean 时，容器必须立即在服务器中创建一个新的 Bean 实例，并关联到客户机上，以后此客户机调用 Stateful Session Bean 的方法时容器会把调用分派到与此客户机相关联的 Bean 实例。EntityBean：Entity Beans 能存活相对较长的时间，并且状 态是持续的。只要数据库中的数据存在，Entity beans 就一直存活。而不是按照应用 程序或者服务进程来说的。即使 EJB 容器崩溃了，Entity beans 也是存活的。Entity Beans 生命周期能够被容器或者 Beans 自己管理。EJB 通过以下技术管理实务：对象管理组织(OMG)的对象实务服务(OTS),Sun Microsystems 的 Transaction Service(JTS)、Java Transaction API（JTA），开发组（X/Open）的 XA 接口。

32. 应用服务器有那些？

BEA WebLogic Server，IBM WebSphere Application Server，Oracle9i Application Server，jBoss，Tomcat

33. 给我一个你最常见到的 runtime exception。

ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DOMException, EmptyStackException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IndexOutOfBoundsException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NullPointerException, ProfileDataException, ProviderException, RasterFormatException, SecurityException, SystemException, UndeclaredThrowableException, UnmodifiableSetException, UnsupportedOperationException

34. 接口是否可继承接口？抽象类是否可实现(implements)接口？抽象类是否可继承实体类(concrete class)？

接口可以继承接口。抽象类可以实现(implements)接口，抽象类是否可继承实体类，但前提是实体类必须有明确的构造函数。

35. List, Set, Map 是否继承自 Collection 接口?

List，Set 是，Map 不是

36. 说出数据连接池的工作机制是什么?

J2EE 服务器启动时会建立一定数量的池连接，并一直维持不少于此数目的池连接。客户端程序需要连接时，池驱动程序会返回一个未使用的池连接并将其表记为忙。如果当前没有空闲连接，池驱动程序就新建一定数量的连接，新建连接的数量有配置参数决定。当使用的池连接调用完成后，池驱动程序将此连接表记为空闲，其他调用就可以使用这个连接。

37. abstract 的 method 是否可同时是 static,是否可同时是 native，是否可同时是 synchronized?

都不能

38. 数组有没有 length()这个方法? String 有没有 length()这个方法?

数组没有 length()这个方法，有 length 的属性。String 有有 length()这个方法。

39. Set 里的元素是不能重复的,那么用什么方法来区分重复与否呢? 是用==还是 equals()? 它们有何区别?

Set 里的元素是不能重复的，那么用 iterator()方法来区分重复与否。equals()是判读两个 Set 是否相等。equals()和==方法决定引用值是否指向同一对象 equals()在类中被覆盖，为的是当两个分离的对象的内容和类型相配的话，返回真值。

40. 构造器 Constructor 是否可被 override?

构造器 Constructor 不能被继承，因此不能重写 Overriding，但可以被重载 Overloading。

41. 是否可以继承 String 类?

String 类是 final 类故不可以继承。

42. swtich 是否能作用在 byte 上，是否能作用在 long 上，是否能作用在 String 上?

switch（expr1）中，expr1 是一个整数表达式。因此传递给 switch 和 case 语句的参数应该是 int、 short、 char 或者 byte。long,string 都不能作用于 swtich。

43. try {}里有一个 return 语句，那么紧跟在这个 try 后的 finally {}里的 code 会不会被执行，什么时候被执行，在 return 前还是后?

会执行，在 return 前执行。

44. 编程题: 用最有效率的方法算出 2 乘以 8 等於几?

2 << 3

45. 两个对象值相同(x.equals(y) == true)，但却可有不同的 hash code，这句话对不对?

不对，有相同的 hash code。

46. 当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递?

是值传递。Java 编程语言只有值传递参数。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的内容可以在被调用的方法中改变，但对象的引用是永远不会改变的。

47. 当一个线程进入一个对象的一个 synchronized 方法后，其它线程是否可进入此对象的其它方法?

不能，一个对象的一个 synchronized 方法只能由一个线程访问。

48. 编程题: 写一个 Singleton 出来。

Singleton 模式主要作用是保证在 Java 应用程序中，一个类 Class 只有一个实例存在。

一般 Singleton 模式通常有几种种形式:

第一种形式: 定义一个类，它的构造函数为 private 的，它有一个 static 的 private 的该类变量，在类初始化时实例话，通过一个 public 的 getInstance 方法获取对它的 引用,继而调用其中的方法。

```
public class Singleton {
    private Singleton(){}
    private static Singleton instance = new Singleton();
    public static Singleton getInstance() {
        return instance;
    }
}
```

第二种形式:

```
public class Singleton {
    private static Singleton instance = null;
    public static synchronized Singleton getInstance() {
        if (instance==null)
        instance＝new Singleton();
        return instance; }
    }
```

其他形式:

定义一个类，它的构造函数为 private 的，所有方法为 static 的。

一般认为第一种形式要更加安全些

49. Java 的接口和 C++的虚类的相同和不同处。

由于 Java 不支持多继承，而有可能某个类或对象要使用分别在几个类或对象里面的方法或属性，现有的单继承机制就不能满足要求。与继承相比，接口有更高的灵活性，因为接口中没有任何实现代码。当一个类实现了接口以后，该类要实现接口里面所有的方法和属性，并且接口里面的属性在默认状态下面都是 public static,所有方法默认情 况下是 public.一个类可以实现多个接口

50. 作用域 public,private,protected,以及不写时的区别

答：区别如下：

作用域	当前类	同一 package	子孙类	其他 package
public	√	√	√	√
protected	√	√	√	×
friendly	√	√	×	×
private	√	×	×	×

不写时默认为 friendly

51. Anonymous Inner Class (匿名内部类) 是否可以 extends(继承)其它类，是否可以 implements(实现)interface(接口)

答：匿名的内部类是没有名字的内部类。不能 extends(继承) 其它类，但一个内部类可以作为一个接口，由另一个内部类实现

52. 描述 Struts 体系结构？对应各个部分的开发工作主要包括哪些？

53. XML 包括哪些解释技术，区别是什么？

54. JSP 有哪些内置对象和动作？它们的作用分别是什么？

request 表示 HttpServletRequest 对象。它包含了有关浏览器请求的信息，并且提供了几个用于获取 cookie, header, 和 session 数据的有用的方法。

response response 表示 HttpServletResponse 对象，并提供了几个用于设置送回 浏览器的响应的方法（如 cookies,头信息等）

out 对象是 javax.jsp.JspWriter 的一个实例，并提供了几个方法使你能用于向浏览器回送输出结果。

pageContext 表示一个 javax.servlet.jsp.PageContext 对象。它是用于方便存取各种范围的名字空间、servlet 相关的对象的 API，并且包装了通用的 servlet 相关功能 的方法。

session 表示一个请求的 javax.servlet.http.HttpSession 对象。Session 可以存贮用户的状态信息

applicaton 表示一个 javax.servle.ServletContext 对象。这有助于查找有关 servlet 引擎和 servlet 环境的信息

config 表示一个 javax.servlet.ServletConfig 对象。该对象用于存取 servlet 实例的初始化参数。

page 表示从该页面产生的一个 servlet 实例

55. 编程练习题库

1) 编译运行如下程序的结果是什么 ？

```
class InvalidIndexException extends Exception{
private int i;
 InvalidIndexException(int a){
    i=a;
}
 public String toString(){
    return i+" is out of boundary--0 < i < 8";
}
```

```
}
public class Weekdays{
 public static void main(String args[]){
    try{
        for (int i=1; i < 9; i++) System.out.println(i+"---"+giveName(i));
    }catch(InvalidIndexException e){
     System.out.println(e.toString());
     }finally{ System.out.println("These days makes up a week.");}
 }
 public static String giveName(int d) {
String name;
    switch(d){
    case 1: name="Monday"; break;
    case 2: name="Tuesday"; break;
    case 3: name="Wednesday"; break;
    case 4: name="Thursday"; break;
    case 5: name="Friday"; break;
    case 6: name="Saturday"; break;
    case 7: name="Sunday"; break;
    default: throw new InvalidIndexException(d);
    }
    return name;
}
}
```

(A) 1---Monday
2---Tuesday
3---Wednesday
4---Thursday
5---Friday
6---Saturday
7---Sunday
These days makes up a week.

(B) 1---Monday
2---Tuesday
3---Wednesday
4---Thursday
5---Friday
6---Saturday
7---Sunday
8 is out of boundary--0 < i < 8
These days makes up a week.

(C) 1---Monday
2---Tuesday
3---Wednesday

4---Thursday
5---Friday
6---Saturday
7---Sunday
（D）编译不能通过。
答案:(B)
2) 有如下一段Java程序：

```
import java.io.*;
public class Quiz1{
public static void main(String arg[]){
        int i;
        System.out.print("Go ");
        try{
            System.out.print("in ");
            i=System.in.read();
         if (i=='0') {throw new MyException();}
            System.out.print("this ");
     }
        catch(IOException e){}
        catch(MyException e){
            System.out.print("that ");
        }
        System.out.print("way.\n");
    }
}
class MyException extends Exception{}
```

运行该程序后输入字符'0', 请问运行结果为何？
（A）Go in this way
（B）Go in that this way
（C）Go in that
（D）Go in that way
答案:(D)
3) 下面程序的输出是什么？

```
public class Quiz2{
    public static void main(String args[]){
        try {throw new MyException();
        }catch(Exception e){
            System.out.println("It's caught!");
        }finally{
            System.out.println("It's finally caught!");
        }
    }
}
class MyException extends Exception{}
```

(A) It's finally caught!
(B) It's caught!
(C) It's caught!
    It's finally caught!
(D) 无 输 出
答 案:(C)

4) 下 面 的 程 序 是 一 个 嵌 套 例 外 处 理 的 例 子, 请 选 择 其 运 行 结 果:

```
public class Quiz3{
    public static void main(String args[]){
        try{
            try{
                int i;
                int j=0;
                i=1/j;
            }catch(Exception e){
                System.out.print("Caught ");
                throw e;
            }finally{
            System.out.print("Inside ");
            }
        }catch(Exception e){
            System.out.print("Caught ");
        }finally{
            System.out.print("Outside\n");
        }
    }
}
```

(A) Caught Outside
(B) Caught Inside
(C) Caught Caught Outside
(D) Caught Inside Caught Outside
答 案:(D)

5) Java 运 行 时 例 外 是 在 运 行 Java 程 序 时 由 Java 运 行 时 系 统 负 责 抛 出 的 一 系 列 例 外。 本 章 例 程 中 所 提 到 的 许 多 例 外 就 是 Java 运 行 时 例 外(详 见 例 8.2 等)。请 详 细 阅 读 例 程, 选 择 对 于 如 下 的 程 序, 系 统 将 抛 出 哪 个 运 行 时 例 外。

```
class Quiz4{
int a[]=new int[10];
    a[10]=0;
}
```

(A) ArithmeticException
(B) ArrayIndexOutOfBoundsException
(C) NegativeArraySizeException

(D) IllegalArgumentException

答 案:(B)

6) 为了编程需要，现需自己编写一个例外类。一般说来，下面声明哪个最为合适？

(A) class myClass extentds Exception{...

(B) class myException extends Error{...

(C) class myException extends RuntimeException{...

(D) class myException extends Exception{...

答 案:(D)

56. JAVA 代码查错

1)

```
abstract class Name {
    private String name;
    public abstract boolean isStupidName(String name) { }
}
```

答案: 错。abstract method 必须以分号结尾，且不带花括号。

2)

```
public class Something {
    void doSomething () {
    private String s = "";
    int l = s.length();
    }
}
```

答案: 错。局部变量前不能放置任何访问修饰符 (private，public，和 protected)。final 可以用来修饰局部变量

(final 如同 abstract 和 strictfp，都是非访问修饰符，strictfp 只能修饰 class 和 method 而非 variable)。

3)

```
abstract class Something {
    private abstract String doSomething ();
}
```

答案: 错。abstract 的 methods 不能以 private 修饰。abstract 的 methods 就是让子类 implement(实现)具体细节的，怎么可以用 private 把 abstract method 封锁起来呢? (同理，abstract method 前不能加 final)。

4)

```
public class Something {
    public int addOne(final int x) {
    return ++x;
    }
}
```

答案: 错。int x 被修饰成 final，意味着 x 不能在 addOne method 中被修改。

5)

```
public class Something {
    public static void main(String[] args) {
```

```
    Other o = new Other();
    new Something().addOne(o);
    }
    public void addOne(final Other o) {
        o.i++;
    }
}
class Other {
    public int i;
}
```

答案: 正确。在 addOne method 中，参数 o 被修饰成 final。如果在 addOne method 里我们修改了 o 的 reference
(比如: o = new Other()，那么如同上例这题也是错的。但这里修改的是 o 的 member variable
成员变量)，而 o 的 reference 并没有改变。

6)

```
class Something {
    int i;
    public void doSomething() {
    System.out.println("i = " + i);
    }
}
```

答案: 正确。输出的是"i = 0"。int i 属於 instant variable (实例变量，或叫成员变量)。instant variable 有 default value。int 的 default value 是 0。

7)

```
    class Something {
        final int i;
        public void doSomething() {
            System.out.println("i = " + i);
        }
    }
```

答案: 错。final int i 是个 final 的 instant variable (实例变量，或叫成员变量)。final 的 instant variable 没有 default value，必须在 constructor (构造器)结束之前被赋予一个明确的值。可以修改为"final int i = 0;"。

8)

```
    public class Something {
        public static void main(String[] args) {
            Something s = new Something();
            System.out.println("s.doSomething() returns " + doSomething());
        }
        public String doSomething() {
            return "Do something ...";
         }
    }
```

答案: 错。看上去在 main 里 call doSomething 没有什么问题，毕竟两个 methods 都在同一个 class 里。但仔细看，main 是 static 的。static method 不能直接 call non-static methods 。可改成"System.out.println("s.doSomething() returns " + s.doSomething());"。同理，static method 不能访问 non-static instant variable。

9)

此处，Something 类的文件名叫 OtherThing.java

```
class Something {
     private static void main(String[] something_to_do) {
        System.out.println("Do something ...";
    }
}
```

答案: 正确。

10)

```
interface  A{
    int x = 0;
}
class B{
    int x =1;
}
class C extends B implements A {
    public void pX(){
        System.out.println(x);
    }
    public static void main(String[] args) {
        new C().pX();
    }
}
```

答案：错误。在编译时会发生错误(错误描述不同的 JVM 有不同的信息，意思就是未明确的 x 调用，两个 x 都匹配（就象在同时 import java.util 和 java.sql 两个包时直接声明 Date 一样)。对于父类的变量,可以用 super.x 来明确，而接口的属性默认隐含为 public static final.所以可以通过 A.x 来明确。

11)

```
interface Playable {
    void play();
}
interface Bounceable {
    void play();
}
interface Rollable extends Playable, Bounceable {
    Ball ball = new Ball("PingPang");
}
class Ball implements Rollable {
    private String name;
```

```
    public String getName() {
        return name;
    }
    public Ball(String name) {
    this.name = name;
    }
    public void play() {
    ball = new Ball("Football";
    System.out.println(ball.getName());
    }
}
```

答案：错。"interface Rollable extends Playable, Bounceable"没有问题。interface 可继承多个 interfaces，所以这里没错。问题出在 interface Rollable 里的 "Ball ball = new Ball("PingPang";" 。任何在 interface 里声明的 interface variable (接口变量，也可称成员变量)，默认为 public static final。也就是说 "Ball ball = new Ball("PingPang")" 实际上是 "public static final Ball ball = new Ball("PingPang";"。在 Ball 类的 Play()方法中，"ball = new Ball("Football";"改变了 ball 的 reference，而这里的 ball 来自 Rollable interface，Rollable interface 里的 ball 是 public static final 的，final 的 object 是不能被改变 reference 的。因此编译器将在 "ball = new Ball("Football";"这里显示有错。

57. JAVA 编程题

1) 现在输入 n 个数字，以逗号，分开；然后可选择升或者降序排序；按提交键就在另一页面显示按什么排序，结果为，提供 reset

```
import java.util.*;
public class bycomma{
    public static String[] splitStringByComma(String source){
        if(source==null||source.trim().equals("")
        return null;
        StringTokenizer commaToker = new StringTokenizer(source,",";
        String[] result = new String[commaToker.countTokens()];
        int i=0;
        while(commaToker.hasMoreTokens()){
            result[i] = commaToker.nextToken();
            i++;
        }
        return result;
    }
    public static void main(String args[]){
        String[] s = splitStringByComma("5,8,7,4,3,9,1";
        int[] ii = new int[s.length];
        for(int i = 0;i<s.length;i++){
        ii[i] =Integer.parseInt(s[i]);
```

```
        }
        Arrays.sort(ii);
         //asc
        for(int i=0;i<s.length;i++){
            System.out.println(ii[i]);
        }
        //desc
            for(int i=(s.length-1);i>=0;i--){
                System.out.println(ii[i]);
             }
        }
    }
```

2) 金额转换，阿拉伯数字的金额转换成中国传统的形式如：（￥1011）—>（一千零一拾一元整）输出。

```
package test.format;
import java.text.NumberFormat;
import java.util.HashMap;
public class SimpleMoneyFormat {
public static final String EMPTY = "";
public static final String ZERO = "零";
public static final String ONE = "壹";
public static final String TWO = "贰";
public static final String THREE = "叁";
public static final String FOUR = "肆";
public static final String FIVE = "伍";
public static final String SIX = "陆";
public static final String SEVEN = "柒";
public static final String EIGHT = "捌";
public static final String NINE = "玖";
public static final String TEN = "拾";
public static final String HUNDRED = "佰";
public static final String THOUSAND = "仟";
public static final String TEN_THOUSAND = "万";
public static final String HUNDRED_MILLION = "亿";
public static final String YUAN = "元";
public static final String JIAO = "角";
public static final String FEN = "分";
public static final String DOT = ".";
private static SimpleMoneyFormat formatter = null;
private HashMap chineseNumberMap = new HashMap();
private HashMap chineseMoneyPattern = new HashMap();
private NumberFormat numberFormat = NumberFormat.getInstance();
private SimpleMoneyFormat() {
  numberFormat.setMaximumFractionDigits(4);
```

```
  numberFormat.setMinimumFractionDigits(2);
  numberFormat.setGroupingUsed(false);
  chineseNumberMap.put("0", ZERO);
  chineseNumberMap.put("1", ONE);
  chineseNumberMap.put("2", TWO);
  chineseNumberMap.put("3", THREE);
  chineseNumberMap.put("4", FOUR);
  chineseNumberMap.put("5", FIVE);
  chineseNumberMap.put("6", SIX);
  chineseNumberMap.put("7", SEVEN);
  chineseNumberMap.put("8", EIGHT);
  chineseNumberMap.put("9", NINE);
  chineseNumberMap.put(DOT, DOT);
  chineseMoneyPattern.put("1", TEN);
  chineseMoneyPattern.put("2", HUNDRED);
  chineseMoneyPattern.put("3", THOUSAND);
  chineseMoneyPattern.put("4", TEN_THOUSAND);
  chineseMoneyPattern.put("5", TEN);
  chineseMoneyPattern.put("6", HUNDRED);
  chineseMoneyPattern.put("7", THOUSAND);
  chineseMoneyPattern.put("8", HUNDRED_MILLION);
 }
public static SimpleMoneyFormat getInstance() {
  if (formatter == null)
    formatter = new SimpleMoneyFormat();
  return formatter;
 }
public String format(String moneyStr) {
  checkPrecision(moneyStr);
  String result;
  result = convertToChineseNumber(moneyStr);
  result = addUnitsToChineseMoneyString(result);
  return result;
 }
public String format(double moneyDouble) {
  return format(numberFormat.format(moneyDouble));
 }
public String format(int moneyInt) {
  return format(numberFormat.format(moneyInt));
 }
public String format(long moneyLong) {
  return format(numberFormat.format(moneyLong));
 }
public String format(Number moneyNum) {
```

```
    return format(numberFormat.format(moneyNum));
  }
 private String convertToChineseNumber(String moneyStr) {
    String result;
    StringBuffer cMoneyStringBuffer = new StringBuffer();
    for (int i = 0; i < moneyStr.length(); i++) {
      cMoneyStringBuffer.append(chineseNumberMap.get(moneyStr.substring(i, i + 1
)));
}
    //拾佰仟万亿等都是汉字里面才有的单位，加上它们
    int indexOfDot = cMoneyStringBuffer.indexOf(DOT);
    int moneyPatternCursor = 1;
    for (int i = indexOfDot - 1; i > 0; i--) {
      cMoneyStringBuffer.insert(i, chineseMoneyPattern.get(EMPTY + moneyPatternC
ursor));
      moneyPatternCursor = moneyPatternCursor == 8 ? 1 : moneyPatternCursor + 1;
     }
    String fractionPart = cMoneyStringBuffer.substring(cMoneyStringBuffer.indexO
f(".");
    cMoneyStringBuffer.delete(cMoneyStringBuffer.indexOf(".", cMoneyStringBuffer
.length());
    while (cMoneyStringBuffer.indexOf("零拾" != -1) {
      cMoneyStringBuffer.replace(cMoneyStringBuffer.indexOf("        零        拾
", cMoneyStringBuffer.indexOf("零拾" + 2, ZERO);
}
    while (cMoneyStringBuffer.indexOf("零佰" != -1) {
      cMoneyStringBuffer.replace(cMoneyStringBuffer.indexOf("        零        佰
", cMoneyStringBuffer.indexOf("零佰" + 2, ZERO);
     }
    while (cMoneyStringBuffer.indexOf("零仟" != -1) {
      cMoneyStringBuffer.replace(cMoneyStringBuffer.indexOf("        零        仟
", cMoneyStringBuffer.indexOf("零仟" + 2, ZERO);
     }
    while (cMoneyStringBuffer.indexOf("零万" != -1) {
      cMoneyStringBuffer.replace(cMoneyStringBuffer.indexOf("        零        万
", cMoneyStringBuffer.indexOf("零万" + 2, TEN_THOUSAND);
     }
    while (cMoneyStringBuffer.indexOf("零亿" != -1) {
      cMoneyStringBuffer.replace(cMoneyStringBuffer.indexOf("        零        亿
", cMoneyStringBuffer.indexOf("零亿" + 2, HUNDRED_MILLION);
     }
    while (cMoneyStringBuffer.indexOf("零零" != -1) {
      cMoneyStringBuffer.replace(cMoneyStringBuffer.indexOf("        零        零
", cMoneyStringBuffer.indexOf("零零" + 2, ZERO);
```

```
    }
   if (cMoneyStringBuffer.lastIndexOf(ZERO) == cMoneyStringBuffer.length() - 1)
     cMoneyStringBuffer.delete(cMoneyStringBuffer.length() - 1, cMoneyStringBuf
fer.length());
   cMoneyStringBuffer.append(fractionPart);
   result = cMoneyStringBuffer.toString();
   return result;
  }

 private String addUnitsToChineseMoneyString(String moneyStr) {
   String result;
   StringBuffer cMoneyStringBuffer = new StringBuffer(moneyStr);
   int indexOfDot = cMoneyStringBuffer.indexOf(DOT);
   cMoneyStringBuffer.replace(indexOfDot, indexOfDot + 1, YUAN);
```

58. 英文题

QUESTION NO:1

```
1、public class Test {
  public static void changeStr(String str){
str="welcome";
  }
  public static void main(String[] args) {
    String str="1234";
    changeStr(str);
    System.out.println(str);
  }
}
```

Please write the output result：

QUESTION NO:2

```
1. public class Test {
2. static boolean foo(char c) {
3. System.out.print(c);
4. return true;
5. }
6. public static void main( String[] argv ) {
7. int i =0;
8. for ( foo(’A’); foo(’B’)&&(i<2); foo(’C’)){
9. i++ ;
10. foo(’D’);
12. }
13. }
14. }
```

What is the result?

A. ABDCBDCB

B. ABCDABCD
C. Compilation fails.
D. An exception is thrown at runtime.
QUESTION NO: 3
1. class A {
2. protected int method1(int a, int b) { return 0; }
3. }
Which two are valid in a class that extends class A? (Choose two)
A. public int method1(int a, int b) { return 0; }
B. private int method1(int a, int b) { return 0; }
C. private int method1(int a, long b) { return 0; }
D. public short method1(int a, int b) { return 0; }
E. static protected int method1(int a, int b) { return 0; }
QUESTION NO: 4
1. public class Outer{
2. public void someOuterMethod() {
3. // Line 3
4. }
5. public class Inner{}
6. public static void main(String[]argv) {
7. Outer o = new Outer();
8. // Line 8
9. }
10. }
Which instantiates an instance of Inner?
A. new Inner(); // At line 3
B. new Inner(); // At line 8
C. new o.Inner(); // At line 8
D. new Outer.Inner(); // At line 8//new Outer().new Inner()
QUESTION NO: 5
Which method is used by a servlet to place its session ID in a URL that is written to the servlet's response output stream?
A. The encodeURL method of the HttpServletRequest interface.
B. The encodeURL method of the HttpServletResponse interface.
C. The rewriteURL method of the HttpServletRequest interface.
D. The rewriteURL method of the HttpServletResponse interface.
QUESTION NO: 6
Which of the following statements regarding the lifecycle of a session bean are correct?
1. java.lang.IllegalStateException is thrown if SessionContext.getEJBObject() is invoked when a stateful session bean instance is passivated.
2. SessionContext.getRollbackOnly() does not throw an exception when a session bean with bean-managed transaction demarcation is activated.
3. An exception is not thrown when SessionContext.getUserTransaction() is called

in the afterBegin method of a bean with container-managed transactions.
4. JNDI access to java:comp/env is permitted in all the SessionSynchronization methods of a stateful session bean with container-managed transaction demarcation.
5. Accessing resource managers in the SessionSynchronization.afterBegin method of a stateful session bean with bean-managed transaction does not throw an exception.
QUESTION NO:7
Module 1 - Getting Started
Q1.What will happen when you compile and run the following code?
(4)
```
public class MyClass{
static int i;
public static void main(String argv[]){
System.out.println(i);
}
}
```
1) Error Variable i may not have been initialized
2) null
3) 1
4) 0
Q2.Which of the following will compile without error (2)(3)
1)
```
import java.awt.*;
package Mypackage;
class Myclass {}
```
2)
```
package MyPackage;
import java.awt.*;
class MyClass{}
```
3)
```
/*This is a comment */
package MyPackage;
import java.awt.*;
class MyClass{}
```
Q3.What will happen if you try to compile and run the following code (1)
```
public class MyClass {
public static void main(String arguments[]) {
amethod(arguments);
}
public void amethod(String[] arguments) {
System.out.println(arguments);
System.out.println(arguments[1]);
}
}
```
1) error Can't make static reference to void amethod.

2) error method main not correct
3) error array must include parameter
4) amethod must be declared with String
Q4.Given the following code (2)
```
public class Sytch{
int x=2000;
public static void main(String argv[]){
System.out.println("Ms "+argv[1]+"Please pay $"+x);
}
```
What will happen if you attempt to compile and run this code with the command line
java Sytch Jones Diggle
1) Compilation and output of Ms Diggle Please pay $2000
2) Compile time error
3) Compilation and output of Ms Jones Please pay $2000
4) Compilation but runtime error
Q5.You have a public class called myclass with the main method defined as follows(4)
```
public static void main(String parm[]){
System.out.println(parm[0]);
}
```
If you attempt to compile the class and run the program as follows
java myclass hello
What will happen?
1) Compile time error, main is not correctly defined
2) Run time error, main is not correctly defined
3) Compilation and output of java
4) Compilation and output of hello
QUESTION NO:8
(1、Examine the following code which includes an inner class:
```
 public final class Test4{
  class Inner{
void test(){
      if (Test4.this.flag);{
     sample();
      }
    }
 }
 private boolean flag=false;
 public void sample(){
 System.out.println("Sample");
 }
 public Test4(){
 (new Inner()).test();
 }
 public static void main(String args[]){
```

new Test4();
}
}
What is the result:
A.Print out “Sample”
B.Program produces no output but termiantes correctly.
C. Program does not terminate.
D.The program will not compile
答案:A
(2、What will happen when you attempt to compile and run the following code?
```
class Base {
int i = 99;
    public void amethod() {
        System.out.println("Base.amethod()");
    }
    Base() {
        amethod();
    }
}
public class Derived extends Base{
int i = -1;
    public static void main(String argv[]) {
        Base b = new Derived();
        System.out.println(b.i);
        b.amethod();
    }
     public void amethod() {
        System.out.println("Derived.amethod()");
    }
}
```
A. Derived.amethod()
-1
Derived.amethod()
B. Derived.amethod()
99
Derived.amethod()
C. 99
Derived.amethod()
D.
Compile time error
答案:B
(3、
```
public class Test{
 public static void main(String[] args){
 StringBuffer a=new StringBuffer("A");
```

```
     StringBuffer b=new StringBuffer("B");
     operate(a,b);
     System.out.pintln(a+","+b);
}
   public static void operate(StringBuffer x, StringBuffer y){
    x.append(y);
    y=x;
   }
```
what is the output?
答案:"AB,B"
(4、
```
public class Test{
   public static void stringReplace(String text){
     text=text.replace('j','l');
}
    public static void bufferReplace(StringBuffer text){
      text=text.append("c");
     }
   public static void main(String args[]){
      String textString=new String("java");
      StringBuffer textBuffer=new StringBuffer("java");
      stringReplace(textString);
      bufferReplace(textBuffer);
      System.out.println(textString+textBuffer);
    }
 }
```
what is the output?
答案:"javajavac"
(5、
```
public class Test{
  static void leftshift(int i, int j){
      i<<=j;
  }
  public static void main(String args[]){
    int i=4, j=2;
   leftshift(i,j);
   System.out.println(i);
 }
}
```
what is the result?
答案:4

B. 知识点精华

1. Get 与 Post 的区别

HTTP 请求：GET 与 POST 方法的区别

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。事实上 GET 适用于多数请求，而保留 POST 仅用于更新站点。根据 HTTP 规范，GET 用于信息获取，而且应该是安全的和幂等的。所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。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。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）；

在 FORM 提交的时候，如果不指定 Method，则默认为 GET 请求，Form 中提交的数据将会附加在 url 之后，以?分开与 url 分开。字母数字字符原样发送，但空格转换为“+“号，其它符号转换为%XX,其中 XX 为该符号以 16 进制表示的 ASCII（或 ISO Latin-1）值。GET 请求请提交的数据放置在 HTTP 请求协议头中，而 POST 提交的数据则放在实体数据中； GET 方式提交的数据最多只能有 1024 字节，而 POST 则没有此限制

表单提交中 Get 和 Post 方式的区别：

1. get 是把参数数据队列加到提交表单的 ACTION 属性所指的 URL 中，值和表单内各个字段一一对应，在 URL 中可以看到。post 是通过 HTTP post 机制，将表单内各个字段与其内容放置在 HTML HEADER 内一起传送到 ACTION 属性所指的 URL 地址。用户看不到这个过程。
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。
3. get 传送的数据量较小，不能大于 2KB。post 传送的数据量较大，一般被默认为不受限制。但理论上，IIS4 中最大量为 80KB，IIS5 中为 100KB。
4. get 安全性非常低，post 安全性较高。如用 get 方法提交用户名密码的时候，密码和用户名会显示在浏览器中。登陆页面可以被浏览器缓存，其他人可以访问客户的这台机器。那么，别人即可以从浏览器的历史记录中，读取到此客户的账号和密码。所以，在某些情况下，get 方法会带来严重的安全性问题。

建议：除非你肯定你提交的数据可以一次性提交，否则请尽量用 Post 方法

2. AJAX

AJAX 全称为“Asynchronous JavaScript and XML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。

主要包含了以下几种技术

Ajax（Asynchronous JavaScript + XML）的定义

基于 web 标准（standards-based presentation）XHTML+CSS 的表示；

使用 DOM（Document Object Model）进行动态显示及交互；

使用 XML 和 XSLT 进行数据交换及相关操作；

使用 XMLHttpRequest 进行异步数据查询、检索；

使用 JavaScript 将所有的东西绑定在一起。英文参见 Ajax 的提出者 Jesse James Garrett 的原文,原文题目(Ajax: A New Approach to Web Applications)。

类似于 DHTML 或 LAMP，AJAX 不是指一种单一的技术，而是有机地利用了一系列相关的技术。事实上，一些基于 AJAX 的“派生/合成”式（derivative/composite）的技术正在出现，如“AFLAX”。

AJAX 前景非常乐观，可以提高系统性能，优化用户界面。AJAX 现有直接框架 AjaxPro，可以引入 AjaxPro.2.dll 文件，可以直接在前台页面 JS 调用后台页面的方法。但此框架与 FORM 验证有冲突。另微软也引入了 AJAX 组建，需要添加 AjaxControlToolkit.dll 文件，可以在控件列表中出现相关控件。

优点和缺点

传统的web应用允许用户填写表单(form)，当提交表单时就向web服务器发送一个请求。服务器接收并处理传来的表单，然後返回一个新的网页。这个做法浪费了许多带宽，因为在前後两个页面中的大部分 HTML 代码往往是相同的。由于每次应用的交互都需要向服务器发送请求，应用的响应时间就依赖于服务器的响应时间。这导致了用户界面的响应比本地应用慢得多。

与此不同，AJAX 应用可以仅向服务器发送并取回必需的数据，它使用 SOAP 或其它一些基于 XML 的 web service 接口，并在客户端采用 JavaScript 处理来自服务器的响应。因为在服务器和浏览器之间交换的数据大量减少，结果我们就能看到响应更快的应用。同时很多的处理工作可以在发出请求的客户端机器上完成，所以 Web 服务器的处理时间也减少了。

使用 Ajax 的最大优点，就是能在不更新整个页面的前提下维护数据。这使得 Web 应用程序更为迅捷地回应用户动作，并避免了在网络上发送那些没有改变过的信息。

Ajax 不需要任何浏览器插件，但需要用户允许 JavaScript 在浏览器上执行。就像 DHTML 应用程序那样，Ajax 应用程序必须在众多不同的浏览器和平台上经过严格的测试。随着 Ajax 的成熟，一些简化 Ajax 使用方法的程序库也相继问世。同样，也出现了另一种辅助程序设计的技术，为那些不支持 JavaScript 的用户提供替代功能。

对应用 Ajax 最主要的批评就是，它可能破坏浏览器后退按钮的正常行为[4]。在动态更新页面的情况下，用户无法回到前一个页面状态，这是因为浏览器仅能记下历史记录中的静态页面。一个被完整读入的页面与一个已经被动态修改过的页面之间的差别非常微妙；用户通常都希望单击后退按钮，就能够取消他们的前一次操作，但是在 Ajax 应用程序中，却无法这样做。不过开发者已想出了种种办法来解决这个问题，当中大多数都是在用户单击后退按钮访问历史记录时，通过建立或使用一个隐藏的 IFRAME 来重现页面上的变更。（例如，当用户在 Google Maps 中单击后退时，它在一个隐藏的 IFRAME 中进行搜索，然后将搜索结果反映到 Ajax 元素上，以便将应用程序状态恢复到当时的状态。）

一个相关的观点认为，使用动态页面更新使得用户难于将某个特定的状态保存到收藏夹中。该问题的解决方案也已出现，大部分都使用 URL 片断标识符（通常被称为锚点，即 URL 中#后面的部分）来保持跟踪，允许用户回到指定的某个应用程序状态。（许多浏览器允许 JavaScript 动态更新锚点，这使得 Ajax 应用程序能够在更新显示内容的同时更新锚点。）这些解决方案也同时解决了许多关于不支持后退按钮的争论。

进行 Ajax 开发时，网络延迟——即用户发出请求到服务器发出响应之间的间隔——需要慎重考虑。不给予用户明确的回应 [5]，没有恰当的预读数据 [6]，或者对 XMLHttpRequest 的不恰当处理[7]，都会使用户感到延迟，这是用户不欲看到的，也是他们无法理解的[8]。通常的解决方案是，使用一个可视化的组件来告诉用户系统正在进行后台操作并且正在读取数据和内容。

一些手持设备（如手机、PDA 等）现在还不能很好的支持 Ajax;

用 JavaScript 作的 Ajax 引擎，JavaScript 的兼容性和 DeBug 都是让人头痛的事;

Ajax 的无刷新重载，由于页面的变化没有刷新重载那么明显，所以容易给用户带来困扰－－用户不太清楚现在的数据是新的还是已经更新过的;现有的解决有：在相关位置提示、数据更新的区域设计得比较明显、数据更新后给用户提示等;

对串流媒体的支持没有 FLASH、Java Applet 好;

基础应用

创建 XMLHttpRequest 方法如下

XMLHttpRequest 类首先由 Internet Explorer 以 ActiveX 对象引入，被称为 XMLHTTP。后来 Mozilla、Netscape、Safari 和其他浏览器也提供了 XMLHttpRequest 类，不过它们创建 XMLHttpRequest 类的方法不同。

对于 Internet Explorer 浏览器：

```
xmlhttp_request = new ActiveXObject("Msxml2.XMLHTTP.3.0"); //3.0 或 4.0, 5.0
xmlhttp_request = new ActiveXObject("Msxml2.XMLHTTP");
xmlhttp_request = new ActiveXObject("Microsoft.XMLHTTP");
```

由于在不同 Internet Explorer 浏览器中 XMLHTTP 版本可能不一致，为了更好的兼容不同版本的 Internet Explorer 浏览器，因此我们需要根据不同版本的 Internet Explorer 浏览器来创建 XMLHttpRequest 类，上面代码就是根据不同的 Internet Explorer 浏览器创建 XMLHttpRequest 类的方法。

对于 Mozilla、Netscape、Safari 等浏览器

创建 XMLHttpRequest 方法如下：`xmlhttp_request = new XMLHttpRequest();`

如果服务器的响应没有 XML mime-type header，某些 Mozilla 浏览器可能无法正常工作。 为了解决这个问题，如果服务器响应的 header 不是 text/xml，可以调用其它方法修改该 header。

```
xmlhttp_request = new XMLHttpRequest();
xmlhttp_request.overrideMimeType('text/xml');
```

在实际应用中，为了兼容多种不同版本的浏览器，一般将创建 XMLHttpRequest 类的方法写成如下形式：

```
try{
if( window.ActiveXObject ){
for( var i = 5; i; i-- ){
try{
if( i == 2 ){
xmlhttp_request = new ActiveXObject( "Microsoft.XMLHTTP" ); }
else{
xmlhttp_request = new ActiveXObject( "Msxml2.XMLHTTP." + i + ".0" );
xmlhttp_request.setRequestHeader("Content-Type","text/xml");
xmlhttp_request.setRequestHeader("Charset","gb2312"); }
break;}
```

```
catch(e){
xmlhttp_request = false; } } }
else if( window.XMLHttpRequest )
{ xmlhttp_request = new XMLHttpRequest();
if (xmlhttp_request.overrideMimeType)
{ xmlhttp_request.overrideMimeType('text/xml'); } } }
catch(e){ xmlhttp_request = false; }
```

发送请求了

可以调用 HTTP 请求类的 open()和 send()方法，如下所示：

```
xmlhttp_request.open('GET', URL, true);
xmlhttp_request.send(null);
```

open()的第一个参数是 HTTP 请求方式—GET，POST 或任何服务器所支持的您想调用的方式。 按照 HTTP 规范，该参数要大写；否则，某些浏览器(如 Firefox)可能无法处理请求。

第二个参数是请求页面的 URL。

第三个参数设置请求是否为异步模式。如果是 TRUE，JavaScript 函数将继续执行，而不等待服务器响应。这就是"AJAX"中的"A"。

服务器的响应

这需要告诉 HTTP 请求对象用哪一个 JavaScript 函数处理这个响应。可以将对象的 onreadystatechange 属性设置为要使用的 JavaScript 的函数名，如下所示：

```
xmlhttp_request.onreadystatechange =FunctionName;
```

FunctionName 是用 JavaScript 创建的函数名，注意不要写成 FunctionName()，当然我们也可以直接将 JavaScript 代码创建在 onreadystatechange 之后，例如：

```
xmlhttp_request.onreadystatechange = function(){
// JavaScript 代码段
};
```

首先要检查请求的状态。只有当一个完整的服务器响应已经收到了，函数才可以处理该响应。XMLHttpRequest 提供了 readyState 属性来对服务器响应进行判断。

readyState 的取值如下：

0 (未初始化)

1 (正在装载)

2 (装载完毕)

3 (交互中)

4 (完成)

所以只有当 readyState=4 时，一个完整的服务器响应已经收到了，函数才可以处理该响应。具体代码如下：

```
if (http_request.readyState == 4) { // 收到完整的服务器响应 }
else { // 没有收到完整的服务器响应 }
```

当 readyState=4 时，一个完整的服务器响应已经收到了，接着，函数会检查 HTTP 服务器响应的状态值。完整的状态取值可参见 W3C 文档。当 HTTP 服务器响应的值为 200 时，表示状态正常。

处理从服务器得到的数据

有两种方式可以得到这些数据：

(1) 以文本字符串的方式返回服务器的响应

(2) 以 XMLDocument 对象方式返回响应

3. java 异常处理

throws 是用来声明一个方法可能抛出的所有异常信息
throw 则是指抛出的一个具体的异常类型。
通常在一个方法（类）的声明处通过 throws 声明方法（类）可能抛出的异常信息，而在方法（类）内部通过 throw 声明一个具体的异常信息。
throws 通常不用显示的捕获异常，可由系统自动将所有捕获的异常信息抛给上级方法；
throw 则需要用户自己捕获相关的异常，而后在对其进行相关包装，最后再将包装后的异常信息抛出。
对异常处理方式不同.throws 对异常不处理,谁调用谁处理,throws 的 Exception 的取值范围要大于方法内部异常的最大范围,而 cathch 的范围又要大于 throws 的 Exception 的范围;throw 主动抛出自定义异常类对象. throws 抛出的是类,throw 抛出的是对象.
在方法定义中表示的是陈述语气，第三人称单数，throw 显然要加 s。(throws 一般用作方法定义的子句)
在函数体中要用 throw，实际上是祈使句+强调，等价于 DO throw ....,do +动词原形
throw 用于引发异常，可引发预定义异常和自定义异常。
I)异常中“throws”和“throw”的区别:
throw 是个“动词”，紧接在 try 语句块之后。
而 throws 是“名词” ，用在函数方法名后 function A () throws Exception e {}
throw 用在程序中明确表示这里抛出一个异常。throws 用在方法声明的地方，表示这个方法可能会抛出某异常。
throw 用来抛出实际的异常, 后面要跟一个异常对象(实例), 是一个实际的语句
throws 是用来声明的, 加在方法声明的后面, 后面跟一个异常类的名字, 表示一般性动作而不是特指某一个动作.
使用 throws 是来说明，当前的函数会抛出一个异常。
在一般的情况下，你调用的一些函数会抛出一些异常。但是你又不想在当前的 context 中去处理它，就可以声明该函数会抛出该异常，这样你就不用去 try-catch 它了。当出现该异常，该函数会抛出此异常，让上一层的函数去处理。throws 也称异常规范

```
public static h() throws
{
     Try
     {
         a.g();
     }
     catch(Exception e)
     {
         throw e;
     }
}
```

因为你没有对这个异常处理，而是简单的抛出了。
而单纯的声明

```
public static h() throws
{
```

```
    a.g();
}
```
也会抛出这个异常

II)try catch \throws \throw

throws 如果发生了对应的错误后，下边的的确不会被执行；

try catch 的理解应该辩证点看：如果 catch 后没有再次 throw 出去，那会继续执行；要想不执行必须 throw 处理

throws　抛出异常，解决不了再向上,直道碰到能解决这个异常的处理程序，就好像你的上司让你执行一项任务，中途你遇到问题不知道如何解决，你把问题返还给你的上司，认为既然是 T 分配的任务就该知道如何解决这个问题，你的上司无法解决同样把它送给经理解决，依次向上，直到有人能解决这个问题为止（不想自己另外写代码处理异常时非常有用）

try catch　　则是考虑到 try 包含这段代码可能会遇到这种异常，直接用 catch 捕获处理，catch 包含的代码为处理代码

throws 只是把一个异常抛出去了，如果你的上层代码有处理方式，就由上层代码来处理这个异常。

而 try/catch 对是清楚的知道该操作可能出现什么异常,同时在 catch 块中应该有处理的方法。

而且还有一种方式就是 try/catch/finaly 的方式。

Throws 是把异常返回给调用者，由调用者处理，调用者还是要 try/catch，跑不掉的

catch 中就一个简单的 SYSTEM.OUT.PRINTLN（……）；还有，连接数据库时会连接不上，你也不知道是驱动问题、电脑问题还是网络问题,发给用户,用户也看不懂,所以统统 throws 给 catch，提示请与管理员联系。。。。。。这就简单多了

throws 写在方法签名后，

throw 写在方法体内，可以写在 if()....

也可以 catch 住一个 exception 后立刻又把他 throw 出去，什么处理也不做，还可以 catch 住后 throw new 一个你自己定义的 exception ....

throws 就是把异常抛出，但是在以后要不有个 catch 接受，要不就抛给主函数.就是逐级往上一级抛，直到有一个接受他的

Throws 抛出异常交给调用该方法的方法 处理，即：

```
public class Test{
    public static void main(String[] args){
            Test2 test2 = new Test2();
            try{
               System.out.println("invoke the method begin!");
               test2.method();
               System.out.println("invoke the method end!");
            }catch(Exception e){
               System.out.println("catch Exception!");
            }
        }
    }
class Test2{
    public void method() throws Exception{
            System.out.println("method begin!");
```

```
            int a = 10;
            int b = 0;
            int c = a/b;
            System.out.println("method end!");
        }
    }
```

很明显，答案出来了：

```
invoke the method begin!
method begin!
catch Exception!
```

finally 语句是任选的，try 语句后至少要有一个 catch 或一个 finally,finally 语句为异常处理提供一个统一的出口，不论 try 代码块是否发生了异常事件，finally 块中的语句都会被执行

在覆盖的方法中声明异常

在子类中，如果要覆盖父类的一个方法，或父类中的方法声明了 throws 异常，则子类的方法也可以抛出异常，但切记子类方法抛出的异常只能是父类方法抛出的异常的同类或子类。如：

```
import java.io.*;
class A {
public void methodA()throws IOException{
.....
}
}
class B1 extends A {
public void methodA()throws FileNotFoundException{
....}
}
class B2 extends A {
public void methodA()throws Exception{//Error
....}
}
public void method() throws Exception {
try {
      具体程序
} catch(Exception ex) {
}
}
```

如果具体程序出错的话，将处理下面程序体中 catch 的地方,这个时候 throws Exception 其实是没有意义的。

```
public void method() throws Exception {
try {
      } catch(FileNotFoundException ex) {
}
}
```

如果具体程序出错的话，且是 FileNotFoundException 的情况下，将处理下面程序体中 catch 的地方处理。
这个时候 FileNotFoundException 以外的 Exception 将通过 throws Exception ，throw 到上一层。
throw 写在方法体内, throws 写在方法名的后面
throw 关键字的格式：throw new ArithmeticException(); 抛出一个异常，这些异常可以使 unchecked exception(也就是 RuntimeException),也可以是 checked execption. throw 必须有一个捕获该异常的 try/catch 语句
throws 关键字的格式

```
private void arrayMethod(int[] arr)
           throws ArrayIndexOutOfBoundsException,
                  ArithmeticException {
     //   Body
}
```

throws 子句列出了方法可能抛出的异常类型，除了 Error 和 RuntimeException 异常,方法中可能抛出的异常必须在 throws 列表中声明,否则就会出现编译错误。
例如：假如方法中可能抛出 IllegalAccessException(属于 checked execption)则必须在 throws 列表中声明。
系统异常是默认抛出的,自己定义的异常要显示抛出
还有一些是库方法只 throw 没有处理的,所以表面上你看到没有 throw 也 catch 到异常

第四部分 .NET

A. 笔试面试题集

1. 填空:
   1) 面向对象的语言具有__继承__性、__封装__性、__多态__性。
   2) 能用 foreach 遍历访问的对象需要实现 ___ IEnumerable ____接口或声明_ GetEnumerator _方法的类型。
   3) 列举 ADO.net 中的五个主要对象_ connection__、 __command__、 _dataadapter__、__dataset__、__datareader__。
2. 不定项选择：
   1) 以下叙述正确的是：__B,C__
      A. 接口中可以有虚方法。　　B. 一个类可以实现多个接口。
      C. 接口不能被实例化。　　D. 接口中可以包含已实现的方法。
   2) 从数据库读取记录，你可能用到的方法有：__B、C、D__
      A. ExecuteNonQuery　　B. ExecuteScalar
      C. Fill　　D. ExecuteReader
3. 简述 private、 protected、 public、 internal 修饰符的访问权限。
   private : 私有成员, 在类的内部才可以访问。
   protected : 保护成员，该类内部和继承类中可以访问。
   public : 公共成员，完全公开，没有访问限制。
   internal: 在同一命名空间内可以访问。
4. 列举 ASP.NET 页面之间传递值的几种方式。
   1) 使用 QueryString, 如....?id=1; response. Redirect()....
   2) 使用 Session 变量
   3) 使用 Server.Transfer
5. 写出程序的输出结果

```
class Class1  {
   private string str = "Class1.str";
   private int i = 0;
   static void StringConvert(string str)  {
       str = "string being converted.";
   }
   static void StringConvert(Class1 c)  {
       c.str = "string being converted.";
   }
   static void Add(int i)  {
       i++;
   }
   static void AddWithRef(ref int i)  {
       i++;
```

```
        }
        static void Main()  {
            int i1 = 10;
            int i2 = 20;
            string str = "str";
            Class1 c = new Class1();
            Add(i1);
            AddWithRef(ref i2);
            Add(c.i);
            StringConvert(str);
            StringConvert(c);
            Console.WriteLine(i1);
            Console.WriteLine(i2);
            Console.WriteLine(c.i);
            Console.WriteLine(str);
            Console.WriteLine(c.str);
        }
    }
```

结果如下：

10

21

0

str

string being converted.

6. 写出程序的输出结果

```
    public abstract class A
    {
        public A()
        {
            Console.WriteLine('A');
        }
        public virtual void Fun()
        {
            Console.WriteLine("A.Fun()");
        }
    }
    public class B: A
    {
        public B()
        {
            Console.WriteLine('B');
        }
        public new void Fun()
        {
```

```
        Console.WriteLine("B.Fun()");
    }
    public static void Main()
    {
        A a = new B();
        a.Fun();
    }
}
```

结果如下：

A

B

A.Fun()

7. 写出程序的输出结果：

```
public class A
{
    public virtual void Fun1(int i)
    {
        Console.WriteLine(i);
    }
    public void Fun2(A a)
    {
        a.Fun1(1);
        Fun1(5);
    }
}
public class B : A
{
    public override void Fun1(int i)
    {
        base.Fun1 (i + 1);
    }
    public static void Main()
    {
        B b = new B();
        A a = new A();
        a.Fun2(b);
        b.Fun2(a);
    }
}
```

结果如下：

2

5

1

6

8. 一列数的规则如下: 1、1、2、3、5、8、13、21、34.....
求第 30 位数是多少， 用递归算法实现。(C#语言)
注：观察后观察从第三位开始，每个数字都是前两个数字之和（斐波那契数列），于是有以下程序：

```
public class MainClass
    {
        public static void Main()
        {
            Console.WriteLine(Foo(30));
        }
        public static int Foo(int i)
        {
            if (i <= 0)
                return 0;
            else if(i > 0 && i <= 2)
                return 1;
            else return Foo(i -1) + Foo(i - 2);
        }
    }
```

9. 程序设计: 猫大叫一声，所有的老鼠都开始逃跑，主人被惊醒。(C#语言)
要求：
1.要有联动性，老鼠和主人的行为是被动的。
2.考虑可扩展性，猫的叫声可能引起其他联动效应。
<1>.构造出 Cat、Mouse、Master 三个类，并能使程序运行
<2>从 Mouse 和 Master 中提取抽象
<3>联动效应，只要执行 Cat.Cryed()就可以使老鼠逃跑，主人惊醒。

```
public interface Observer
    {
        void Response();    //观察者的响应，如是老鼠见到猫的反映
    }
    public interface Subject
    {
        void AimAt(Observer obs);  //针对哪些观察者，这里指猫的要扑捉的对象---老鼠
    }
    public class Mouse : Observer
    {
        private string name;
        public Mouse(string name, Subject subj)
        {
            this.name = name;
            subj.AimAt(this);
        }
        public void Response()
```

```
    {
        Console.WriteLine(name + " attempt to escape!");
    }
}
public class Master : Observer
{
    public Master(Subject subj)
    {
        subj.AimAt(this);
    }
    public void Response()
    {
        Console.WriteLine("Host waken!");
    }
}
public class Cat : Subject
{
    private ArrayList observers;
    public Cat()
    {
        this.observers = new ArrayList();
    }
    public void AimAt(Observer obs)
    {
        this.observers.Add(obs);
    }
    public void Cry()
    {
        Console.WriteLine("Cat cryed!");
        foreach (Observer obs in this.observers)
        {
            obs.Response();
        }
    }
}
class MainClass
{
    static void Main(string[] args)
    {
        Cat cat = new Cat();
        Mouse mouse1 = new Mouse("mouse1", cat);
        Mouse mouse2 = new Mouse("mouse2", cat);
        Master master = new Master(cat);
        cat.Cry();
```

```
        }
    }
```

10. C#中 property 与 attribute 的区别，他们各有什么用处，这种机制的好处在哪里？

11. 讲一讲你理解的 web service,在 dotnet framework 中，怎么很好的结合 xml?

12. C#, Java 和 c++的特点，有什么相同的地方，不同的地方，C#分别从 c++和 java 中吸取了他们那些优点？

13. C#可否对内存进行直接的操作？

14. 用 Visual C++ 6.0 编写的代码(unmanaged code)，如何在 CLR 下和其他 dot net component 结合？

15. ADO。NET 相对于 ADO 等主要有什么改进？

16. ASP.NET 与 ASP 相比，主要有哪些进步？

17. C#中的委托是什么？事件是不是一种委托？

18. 描述一下 C#中索引器的实现过程，是否只能根据数字进行索引？

19. C#中要使一个类支持 FOREACH 遍历，实现过程怎样？

20. 写一个 HTML 页面，实现以下功能，左键点击页面时显示“您好”，右键点击时显示“禁止右键”。并在 2 分钟后自动关闭页面。

21. 你对 XMLHTTP、WEBSERVICE 了解吗？简单描述其特点、作用

22. 接口和抽象类有什么区别？你选择使用接口和抽象类的依据是什么？

23. 定义控件和一般用户控件的异同？如果要用这两者之一，你会选择哪种？为什么

24. 大概描述一下 ASP。NET 服务器控件的生命周期

25. 在 C#中，string str = null 与 string str = “” 请尽量使用文字或图象说明其中的区别。回答要点：说明详细的空间分配。
    答：string str = null 是不给他分配内存空间,而 string str = "" 给它分配长度为空字符串的内存空间.

26. 请详述在 dotnet 中类(class)与结构(struct)的异同：
    答：Class 可以被实例化,属于引用类型,是分配在内存的堆上的,Struct 属于值类型,是分配在内存的栈上的.

27. 根据委托(delegate)的知识，请完成以下用户控件中代码片段的填写

```
namespace test
{
public delegate void OnDBOperate();
public class UserControlBase : System.Windows.Forms.UserControl
{
public event OnDBOperate OnNew;
privatevoidtoolBar_ButtonClick(objectsender,System.Windows.Forms.ToolBarBut
tonClickEventArgs e)
{
if(e.Button.Equals(BtnNew))
{
//请在以下补齐代码用来调用 OnDBOperate 委托签名的 OnNew 事件。
}
}
}
```

答：if(OnNew != null)
OnNew(this, e);

28. 分析以下代码，完成填空

```
string strTmp = "abcdefg 某某某";
int i= System.Text.Encoding.Default.GetBytes(strTmp).Length;
int j= strTmp.Length;
```

以上代码执行完后，i=　　j=
答：i=13,j=10

29. 根据线程安全的相关知识,分析以下代码,当调用 test 方法时 i>10 时是否会引起死锁?并简要说明理由。

```
public void test(int i)
{
lock(this)
{
if (i>10)
{
i--;
test(i);
}
}
}
```

答：不会发生死锁，(但有一点 int 是按值传递的，所以每次改变的都只是一个副本，因此不会出现死锁。但如果把 int 换做一个 object，那么死锁会发生)

30. 分析以下代码。

```
public static void test(string  ConnectString)
```

```
{
System.Data.OleDb.OleDbConnection conn = new
System.Data.OleDb.OleDbConnection();
conn.ConnectionString = ConnectString;
try
{
conn.Open();
…….
}catch(Exception Ex)
{
MessageBox.Show(Ex.ToString());
}finally
{
if (!conn.State.Equals(ConnectionState.Closed))
conn.Close();
}
```

请问

1)以上代码可以正确使用连接池吗？

答：回答：如果传入的 connectionString 是一模一样的话，可以正确使用连接池。不过一模一样的意思是，连字符的空格数，顺序完全一致。

2)以上代码所使用的异常处理方法，是否所有在 test 方法内的异常都可以被捕捉并显示出来？

答：只可以捕捉数据库连接中的异常吧. （finally 中，catch 中，如果有别的可能引发异常的操作，也应该用 try,catch。所以理论上并非所有异常都会被捕捉。）

31. 简要谈一下您对微软.NET 构架下 remoting 和 webservice 两项技术的理解以及实际中的应用。

答：WS 主要是可利用 HTTP，穿透防火墙。而 Remoting 可以利用 TCP/IP，二进制传送提高效率。

32. 公司要求开发一个继承 System.Windows.Forms.ListView 类的组件，要求达到以下的特殊功能：点击 ListView 各列列头时，能按照点击列的每行值进行重排视图中的所有行(排序的方式如 DataGrid 相似)。根据您的知识，请简要谈一下您的思路：

答：根据点击的列头,包该列的 ID 取出,按照该 ID 排序后,在给绑定到 ListView 中

33. 给定以下 XML 文件，完成算法流程图。

```
<FileSystem>
< DriverC >
<Dir DirName=”MSDOS622”>
<File FileName =” Command.com” ></File>
</Dir>
<File FileName =”MSDOS.SYS” ></File>
<File FileName =” IO.SYS” ></File>
</DriverC>
```

</FileSystem>
请画出遍历所有文件名（FileName）的流程图(请使用递归算法)。
答：

```
void FindFile( Directory d )
{
    FileOrFolders = d.GetFileOrFolders();
    foreach( FileOrFolder fof in FileOrFolders )
    {
        if( fof is File )
          You Found a file;
        else if ( fof is Directory )
          FindFile( fof );
    }
}
```

从根节点开始遍历找子节点,在从找到的子节点找它的子节点,一层层下去

B. 知识点精华

1. WEB SERVICE

当前，WebService 是一个热门话题。但是，WebService 究竟是什么？什么情况下应该用 WebService？什么情况下不应该用 WebService？是需要我们正确认识的。

Web Service 是一种新的 web 应用程序分支，他们是自包含、自描述、模块化的应用，可以发布、定位、通过 web 调用。Web Service 可以执行从简单的请求到复杂商务处理的任何功能。一旦部署以后，其他 Web Service 应用程序可以发现并调用它部署的服务。

实际上，WebService 的主要目标是跨平台的可互操作性。为了达到这一目标，WebService 完全基于 XML（可扩展标记语言）、XSD（XMLSchema）等独立于平台、独立于软件供应商的标准，是创建可互操作的、分布式应用程序的新平台。由此可以看出，在以下三种情况下，使用 WebService 会带来极大的好处。

长项一：跨防火墙的通信

如果应用程序有成千上万的用户，而且分布在世界各地，那么客户端和服务器之间的通信将是一个棘手的问题。因为客户端和服务器之间通常会有防火墙或者代理服务器。在这种情况下，使用 DCOM 就不是那么简单，通常也不便于把客户端程序发布到数量如此庞大的每一个用户手中。传统的做法是，选择用浏览器作为客户端，写下一大堆 ASP 页面，把应用程序的中间层暴露给最终用户。这样做的结果是开发难度大，程序很难维护。

举个例子，在应用程序里加入一个新页面，必须先建立好用户界面(Web 页面)，并在这个页面后面，包含相应商业逻辑的中间层组件，还要再建立至少一个 ASP 页面，用来接受用户输入的信息，调用中间层组件，把结果格式化为 HTML 形式，最后还要把“结果页”送回浏览器。要是客户端代码不再如此依赖于 HTML 表单，客户端的编程就简单多了。

如果中间层组件换成 WebService 的话，就可以从用户界面直接调用中间层组件，从而省掉建立 ASP 页面的那一步。要调用 WebService，可以直接使用 MicrosoftSOAPToolkit 或.NET 这样的 SOAP 客户端，也可以使用自己开发的 SOAP 客户端，然后把它和应用程序连接起来。不仅缩短了开发周期，还减少了代码复杂度，并能够增强应用程序的可维护性。同时，应用程序也不再需要在每次调用中间层组件时，都跳转到相应的“结果页”。

从经验来看，在一个用户界面和中间层有较多交互的应用程序中，使用 WebService 这种结构，可以节省花在用户界面编程上 20%的开发时间。另外，这样一个由 WebService 组成的中间层，完全可以在应用程序集成或其它场合下重用。最后，通过 WebService 把应用程序的逻辑和数据“暴露”出来，还可以让其它平台上的客户重用这些应用程序。

长项二：应用程序集成

企业级的应用程序开发者都知道，企业里经常都要把用不同语言写成的、在不同平台上运行的各种程序集成起来，而这种集成将花费很大的开发力量。应用程序经常需要从运行在 IBM 主机上的程序中获取数据；或者把数据发送到主机或 UNIX 应用程序中去。即使在同一个平台上，不同软件厂商生产的各种软件也常常需要集成起来。通过 WebService，应用程序可以用标准的方法把功能和数据“暴露”出来，供其它应用程序使用。

例如，有一个订单登录程序，用于登录从客户来的新订单，包括客户信息、发货地址、数量、价格和付款方式等内容；还有一个订单执行程序，用于实际货物发送的管理。这两个程序来自不同软件厂商。一份新订单进来之后，订单登录程序需要通知订单执行程序发送货

物。通过在订单执行程序上面增加一层 WebService，订单执行程序可以把“AddOrder”函数“暴露”出来。这样，每当有新订单到来时，订单登录程序就可以调用这个函数来发送货物了。

长项三：B2B 的集成

用 WebService 集成应用程序，可以使公司内部的商务处理更加自动化。但当交易跨越供应商和客户、突破公司的界限时会怎么样呢？跨公司的商务交易集成通常叫做 B2B 集成。

WebService 是 B2B 集成成功的关键。通过 WebService，公司可以把关键的商务应用“暴露”给指定的供应商和客户。例如，把电子下单系统和电子发票系统“暴露”出来，客户就可以以电子的方式发送订单，供应商则可以以电子的方式发送原料采购发票。当然，这并不是一个新的概念，EDI(电子文档交换)早就是这样了。但是，WebService 的实现要比 EDI 简单得多，而且 WebService 运行在 Internet 上，在世界任何地方都可轻易实现，其运行成本就相对较低。不过，WebService 并不像 EDI 那样，是文档交换或 B2B 集成的完整解决方案。WebService 只是 B2B 集成的一个关键部分，还需要许多其它的部分才能实现集成。

用 WebService 来实现 B2B 集成的最大好处在于可以轻易实现互操作性。只要把商务逻辑“暴露”出来，成为 WebService，就可以让任何指定的合作伙伴调用这些商务逻辑，而不管他们的系统在什么平台上运行，使用什么开发语言。这样就大大减少了花在 B2B 集成上的时间和成本，让许多原本无法承受 EDI 的中小企业也能实现 B2B 集成。

长项四：软件和数据重用

软件重用是一个很大的主题，重用的形式很多，重用的程度有大有小。最基本的形式是源代码模块或者类一级的重用，另一种形式是二进制形式的组件重用。

当前，像表格控件或用户界面控件这样的可重用软件组件，在市场上都占有很大的份额。但这类软件的重用有一个很大的限制，就是重用仅限于代码，数据不能重用。原因在于，发布组件甚至源代码都比较容易，但要发布数据就没那么容易，除非是不会经常变化的静态数据。

WebService 在允许重用代码的同时，可以重用代码背后的数据。使用 WebService，再也不必像以前那样，要先从第三方购买、安装软件组件，再从应用程序中调用这些组件；只需要直接调用远端的 WebService 就可以了。举个例子，要在应用程序中确认用户输入的地址，只需把这个地址直接发送给相应的 WebService，这个 WebService 就会帮你查阅街道地址、城市、省区和邮政编码等信息，确认这个地址是否在相应的邮政编码区域。WebService 的提供商可以按时间或使用次数来对这项服务进行收费。这样的服务要通过组件重用来实现是不可能的，那样的话你必须下载并安装好包含街道地址、城市、省区和邮政编码等信息的数据库，而且这个数据库还是不能实时更新的。

另一种软件重用的情况是，把好几个应用程序的功能集成起来。例如，要建立一个局域网上的门户站点应用，让用户既可以查询联邦快递包裹，查看股市行情，又可以管理自己的日程安排，还可以在线购买电影票。现在 Web 上有很多应用程序供应商，都在其应用中实现了这些功能。一旦他们把这些功能都通过 WebService“暴露”出来，就可以非常容易地把所有这些功能都集成到你的门户站点中，为用户提供一个统一的、友好的界面。

将来，许多应用程序都会利用 WebService，把当前基于组件的应用程序结构扩展为组件/WebService 的混合结构，可以在应用程序中使用第三方的 WebService 提供的功能，也可以把自己的应用程序功能通过 WebService 提供给别人。两种情况下，都可以重用代码和代码背后的数据。

从以上论述可以看出，WebService 在通过 Web 进行互操作或远程调用的时候是最有用的。不过，也有一些情况，WebService 根本不能带来任何好处。

短处一：单机应用程序

目前，企业和个人还使用着很多桌面应用程序。其中一些只需要与本机上的其它程序通信。在这种情况下，最好就不要用 WebService，只要用本地的 API 就可以了。COM 非常适合于在这种情况下工作，因为它既小又快。运行在同一台服务器上的服务器软件也是这样。最好直接用 COM 或其它本地的 API 来进行应用程序间的调用。当然 WebService 也能用在这些场合，但那样不仅消耗太大，而且不会带来任何好处。

短处二：局域网的同构应用程序

在许多应用中，所有的程序都是用 VB 或 VC 开发的，都在 Windows 平台下使用 COM，都运行在同一个局域网上。例如，有两个服务器应用程序需要相互通信，或者有一个 Win32 或 WinForm 的客户程序要连接局域网上另一个服务器的程序。在这些程序里，使用 DCOM 会比 SOAP/HTTP 有效得多。与此相类似，如果一个.NET 程序要连接到局域网上的另一个.NET 程序，应该使用.NETremoting。有趣的是，在.NETremoting 中，也可以指定使用 SOAP/HTTP 来进行 WebService 调用。不过最好还是直接通过 TCP 进行 RPC 调用，那样会有效得多。

总之，只要从应用程序结构的角度看，有别的方法比 WebService 更有效、更可行，那就不要用 WebService

2. C#中的委托

委托是 C＃中的一种引用类型，类似于 C/C++中的函数指针。与函数指针不同的是，委托是面向对象、类型安全的，而且委托可以引用静态方法和实例方法，而函数指针只能引用静态函数。委托主要用于 .NET Framework 中的事件处理程序和回调函数。

一个委托可以看作一个特殊的类，因而它的定义可以像常规类一样放在同样的位置。与其他类一样，委托必须先定义以后，再实例化。与类不同的是，实例化的委托没有与之相应的术语(类的实例化称作对象)，作为区分我们将实例化的委托称为委托实例。

函数指针

一个函数在编译时被分配给一个入口地址，这个入口地址就称为函数的指针，正如同指针是一个变量的地址一样。

函数指针的用途很多，最常用的用途之一是把指针作为参数传递到其他函数。我们可以参考下面的例子进一步理解函数指针作为参数的情况：

```
# include<stdio.h>
int max(int x,int y)
{
	return (x>y?x:y);
}
int min(int x,int y)
{
	return(x<y?x:y);
}
int sub(int x, int y)
{
	return(x+y);
}
int minus(int x,int y)
{
```

```
        return(x-y);
}
void test(int (*p)(int,int),int (*q)(int,int),int a,int b)
{
        int Int1,Int2;
        Int1=(*p)(a,b);
        Int2=(*q)(a,b);
        printf("%d,\t%d\n",Int1,Int2);
}
void main()
{
        test(max,min,10,3);
        test(sub,minus,10,3);
}
```

客观的讲，使用函数指针作为其参数的函数如果直接调用函数或是直接把调用的函数的函数体放在这个主函数中也可以实现其功能。那么为什么还要使用函数指针呢？我们仔细看一下上面的 main()函数就可以发现，main()函数两次调用了 test 函数，前一次求出最大最小值，后一次求出两数的和与差。如果我们 test 函数不用函数指针，而是采用直接在 test 函数中调用函数的方法，使用一个 test 函数还能完成这个功能吗？显然不行，我们必须写两个这样的 test 函数供 main()函数调用，虽然大多数代码还是一样的，仅仅是调用的函数名不一样。上面仅仅是一个简单的例子，实际生活中也许 main()函数会频繁的调用 test()，而每次的差别仅仅是完成的功能不一样，也许第一次调用会要求求出两数的和与差，而下一次会要求求出最大值以及两数之和，第三次呢，也许是最小值和最大值，……，如果不用函数指针，我们需要写多少个这样的 test()函数？显然，函数指针为我们的编程提供了灵活性。

另外，有些地方必须使用到函数指针才能完成给定的任务，特别是异步操作的回调和其他需要匿名回调的结构。另外，像线程的执行，事件的处理，如果缺少了函数指针的支持也是很难完成的。

类型安全

从上面的介绍可以看出，函数指针的提出还是有其必要的，上面的介绍也同时说明了委托存在的必要性。那么为什么 C＃中不直接用函数指针，而是要使用委托呢？这就涉及到另外一个问题：C＃是类型安全的语言。何谓类型安全？这里的类型安全特指内存类型安全，即类型安全代码只访问被授权可以访问的内存位置。如果代码以任意偏移量访问内存，该偏移量超出了属于该对象的公开字段的内存范围，则它就不是类型安全的代码。显然指针不属于类型安全代码，这也是为什么 C＃使用指针时必须申明 unsafe 的缘故。

那么类型不安全代码可能会带来什么不良的后果呢？相信对于安全技术感兴趣的朋友一定十分熟悉缓冲区溢出问题，通过缓冲区溢出攻击者可以运行非法的程序获得一定的权限从而攻击系统或是直接运行恶意代码危害系统，在 UNIX 下这是一个十分普遍的问题。那么缓冲区溢出又和函数指针有什么关系呢？事实上，攻击者就是通过缓冲区溢出改变返回地址的值到恶意代码地址来执行恶意代码的。我们可以看看下面的代码：

```
void copy()
            {
                char buffer[128];
                ........
                    strcpy (buffer,getenv("HOME"));//HOME 为 UNIX 系统中的 HOME
```

环境变量

```
                ........
        }
```

上面的代码中如果 HOME 环境变量的字符数大于 128，就会产生缓冲区溢出，假如这个缓冲区之前有另一个函数的返回地址，那么这一是地址就有可能覆盖，而覆盖这一地址的字符有可能就是恶意代码的地址，攻击者就有可能攻击成功了！

上面的例子仅仅是指针问题中的一种，除此以外，还可能由于错误的管理地址，将数据写入错误地址，造成程序的崩溃；还可能由于对指针不恰当的赋值操作产生悬浮指针；还可能产生内存越界，内存泄漏等等问题。

由此可见，指针不是类型安全的，函数指针当然也不例外，所以 C＃里面没有使用函数指针，而且不建议使用指针变量。

前面的说明充分证明了委托存在的必要性，那么我们再谈谈为什么委托是类型安全的。C＃中的委托和指针不一样，指针不通过 MSIL 而是直接和内存打交道，这也是指针不安全的原因所在，当然也是采用指针能够提高程序运行速度的缘故；委托不与内存打交道，而是把这一工作交给 CLR 去完成。CLR 无法阻止将不安全的代码调用到本机（非托管）代码中或执行恶意操作。然而当代码是类型安全时，CLR 的安全性强制机制确保代码不会访问本机代码，除非它有访问本机代码的权限。

委托派生于基类 System.Delegate，不过委托的定义和常规类的定义方法不太一样。委托的定义通过关键字 delegate 来定义：

```
public delegate int myDelegate(int x,int y);
```

上面的代码定义了一个新委托，它可以封装任何返回为 int，带有两个 int 类型参数的方法。任何一个方法无论是实例方法还是静态方法，只要他们的签名（参数类型在一个方法中的顺序）和定义的委托是一样的，都可以把他们封装到委托中去。这种签名方法正是保证委托是类型安全的手段之一。

产生委托实例和产生类实例（对象）差不多，假如我们有如下的方法：

```
public int sub(int x,int y)
{
       return(x+y);
}
```

我们就可以使用如下的代码得到一个委托实例：

```
myDelegate calculatin=new myDelegate(sub);
```

接下来我们就可以直接使用 calculation 调用 sub 方法了：

```
calculation(10,3);
```

下面我们将用委托重写上面的一个程序来看一下在 C＃中如何通过委托实现由函数指针实现的功能：

```
using System;
class MathClass
{
       public static int max(int a,int b)
       {
              return(a>b?a:b);
       }
       public static int min(int a,int b)
       {
```

```
            return(a<b?a:b);
        }
        public static int sub(int a,int b)
        {
            return (a+b);
        }
        public static int minus(int a,int b)
        {
            return (a-b);
        }
}
class Handler
{
        private delegate int Calculation(int a, int b);
        private static Calculation[] myCalculation=new Calculation[2];
        public static void EventHandler(int i,int a,int b)
        {
            switch (i)
            {
                case 1:
                    myCalculation[0]=new Calculation(MathClass.max);
                    myCalculation[1]=new Calculation(MathClass.min);
                    Console.WriteLine(myCalculation[0](a,b));
                    Console.WriteLine(myCalculation[1](a,b));
                    break;
                case 2:
                    myCalculation[0]=new Calculation(MathClass.sub);
                    myCalculation[1]=new Calculation(MathClass.minus);
                    Console.WriteLine(myCalculation[0](a,b));
                    Console.WriteLine(myCalculation[1](a,b));
                    break;
                default:
                    return;
            }
        }
}
class Test
{
        static void Main()
        {
            Handler.EventHandler(1,10,3);
            Handler.EventHandler(2,10,3);
        }
}
```

我们还可以声明一个委托数组，就像声明一个对象数组一样，上面的例子中就使用到了委托数组；一个委托还可以封装多个方法（多路广播委托，经常与事件处理程序结合使用），只要这些方法的签名是正确的。多路广播委托的返回值一般为 void，这是因为一个委托只能有一个返回值，如果一个返回值不为 void 的委托封装了多个方法时，只能得到最后封装的方法的返回值，这可能和用户初衷不一致，同时也会给管理带来不方便。如果你想通过委托返回多个值，最好是使用委托数组，让每个委托封装一个方法，各自返回一个值。

事件

在 C＃中，委托的最基本的一个用处就是用于事件处理。事件是对象发送的消息，以发信号通知操作的发生，通俗一点讲，事件就是程序中产生了一件需要处理的信号。

事件的定义用关键字 event 声明,不过声明事件之前必须存在一个多路广播委托：

```
public delegate void Calculate(int x,int y);//返回值为 void 的委托自动成为多路广播委托;
public event calculate OnCalculate;
```

从上节的委托实例和上面的事件的声明可以看出,事件的声明仅仅是比委托实例的声明多了个关键字 event，事实上事件可以看作是一个为事件处理过程定制的多路广播委托。因此，定义了事件后，我们就可以通过向事件中操作符+=添加方法实现事件的预定或者是通过-=取消一个事件，这些都与委托实例的处理是相同的。与委托实例不同的是，操作符＝对于事件是无效的，即

```
OnCalculate=new calculate(sub) ;//无效
```

只是因为上面的语句会删除由 OnCalculate 封装的所有其他方法,指封装了由此语句指定的唯一方法，而且一个预定可以删除其他所有方法，这会导致混乱。

回调函数

回调函数是在托管应用程序中可帮助非托管 DLL 函数完成任务的代码。对回调函数的调用将从托管应用程序中，通过一个 DLL 函数，间接地传递给托管实现。在用平台调用调用的多种 DLL 函数中，有些函数要求正确地运行托管代码中的回调函数。关于回调函数只是使用到委托，在此不加过多说明，具体实现可参考

怎么理解事件委托 ？

理解一：事件委托的概念

.NET 框架的事件委托遵循特定的签名和命名约定。这种约定依赖于可视化设计工具，为客户端代码提供了一致性的模型。为了理解这种约定，下面来看看.NET 框架中一个常见的可访问事件委托 System.EventHandler:

```
public delegate void EventHandler(object sender,EventArgs e);
```

下面是一些特定的事件委托签名约定：

◎ 事件委托的返回类型是 void

◎ 一个事件委托带有两个参数。第一个参数是 object 类型，表示事件的发送者。第二个参数

描述事件的数据，是由 System.EventArgs 派生出的类的实例。

应该根据.NET 框架的命名约定来给事件数据类和事件委托命名。事件数据类由事件名再添加

后缀 EventArgs 构成，如 MonthChangeEventArgs。

事件委托由事件名再添加 EventHandler 后缀构成，如 MonthChangeEventHandler。事件委托

用事件处理程序(Event Handler)命名是因为她们将绑定到处理事件的方法上。

理解二：Wiring 事件

将事件处理程序和事件相关联的过程(添加委托给 invocation 列表)叫做事件布线(event wiring).
而从事件中删除事件处理程序的过程叫做事件撤线(event unwring)。
在 C#中，对于一个事件的布线和撤线事件处理程序的语法如下：
button.Click += new EventHandler(this.Button_Clicked);
button.Click -= new EventHandler(this.Button_Clicked) ;
这里，button 是 Button 控件的一个实例，并建立于具有 Button_Clicked 方法的类中，该方法处理按钮的 Click 事件。
理解三：事件委托的实现
为了在类中实现事件，需要一个事件数据的类、事件委托、在类中拥有 invocation 列表的委托成员，以及一个发布事件通知的方法。
具体实现过程如下：
1)如果类没有任何关联的事件数据，对事件数据使用 EventArgs 类。或者可以使用其他事先已经存在的事件数据类。如果不存在一个合适的事件数据类，则定义一个事件来包含事件数据。这个类必须从 System.EventArgs 中派生。按照规则它的名字应该是事件名加上 EventArgs 得到。例如，AdCreatedEventArgs,MonthChangedEventArgs.
下面的代码声明了一个事件数据类：
public class LowChargeEventArgs:EventArgs{...}
2) 如果事件没有关联数据，使用第一步的 EventArgs,用 System.EventHandler 作为事件委托或者使用能匹配事件的预先存在的其他委托。如果不存在一个合适的事件委托，则定义一个事件委托，该委托的第二个参数具备来自第一步的事件数据类的类型。根据规则，事件委托的名字是在事件后附加 EventHandler. 例如，AdCreateEventHandler,MonthChangedEventHandler。下面的代码用来定义事件委托：
public delegate void LowChargeEventHandler(object sender,LowChargeEventArgs e);
3) 在类里，用 event 关键字定义事件成员。将事件名传递给事件成员。成员的类型是第二步中事件委托的类型。
如下面的例子：
public event LowChargeEventHandler LowCharge;
事件成员包含订阅事件的委托的列表。当调用这个成员时，它通过调用委托来分配事件。
4) 在类里，定义一个受保护的虚拟方法，在检查是否有事件监听器存在之后调用事件委托。其参数就是第一步中定义的事件数据×EventArgs。方法的名字是在事件名前加上前缀 On。
例如：

```
protected virtual void OnLowCharge(LowChargeEventArgs e){
       if (LowCharge != null){
          LowCharge(this,e);
          }
}
```

3. C#执行存储过程

1. C#中执行存储过程
2. 存储过程 p_sys_Login 定义如下：
3. CREATE PROCEDURE p_sys_Login

```
4.  @argUserID varchar(20), --用户名
5.  @argPassword varchar(20), --密码
6.  @argResult varchar(50) OUTPUT --登录结果
7.  AS
8.  /*
9.  ... ...
10. */
```

11. 下面演示如何在 C#中用最简洁有效的代码执行该存储过程并返回数据：

```
12. /// <summary>
13. /// 用户登录验证
14. /// </summary>
15. /// <param name="userID">用户名</param>
16. /// <param name="password">密码</param>
17. public void Login(string userID, string password)
18. {
19. //数据库连接字符串存储在 Web.config 中
20. string cnnString = ConfigurationSettings.AppSettings["ConnectionString"];SqlConnection cnn = new SqlConnection(cnnString);
21. string sql = string.Format("EXEC p_sys_Login '{0}', '{1}', @Result OUTPUT",userID, password);
22. SqlCommand cmd = new SqlCommand(sql, cnn);
23. //建立并添加和“@Result OUTPUT”对应的参数
24. SqlParameter paramResult = new SqlParameter("@Result", SqlDbType.VarChar, 50);
25. paramResult.Direction = ParameterDirection.Output;
26. cmd.Parameters.Add(paramResult);
27. cnn.Open();
28. cmd.ExecuteNonQuery();
29. cnn.Close();
30. //获取存储过程返回的结果
31. string result = paramResult.Value.ToString();
32. //... ...
33. }
```

当前开发过程中，微软的.NET 以其易用和对网络的支持性好等而倍受开发人员的青睐，不少项目使用 Oralce 数据库作为后台数据库，但是在开发过程中需要通过前台程序调用数据库中的一些对象，本文将以一个实例的形式，对

1、在数据库中建一用户表及用户 ID 的序列：

```
create sequence seq_user_information
increment by 1
start with 1
nomaxvalue
nocycle
cache 10
```

```
create table user_information
(
  user_id number primary key, --用户序号
  user_login_name varchar2(30) not null, --登陆名
  user_password varchar2(50) not null, --用户密码
  user_name varchar2(20), --用户姓名
  user_telephone varchar2(20), --用户电话
  user_type number(5) , --用户类型
  creation_date date not null, --创建日期
  last_update_date date not null --最后修改日期
)
```

2、在 Oracle 中建执行插入操作的存储过程：

```
create or replace procedure insert_user_information
(
  p_user_login_name in varchar2,
  p_user_password in varchar2,
  p_user_name in varchar2,
  p_user_telephone in varchar2,
  p_user_type in number,
  p_out out number
) as
  v_count number;
begin
  if p_user_login_name is null or p_user_password is null then
    p_out:=-1; --用户名和密码不能为空
        return ;
      end if;
      if p_user_type is null then p_out:=-2; --用户类型不能为空
        return ;
      end if;
         select    count(*)    into    v_count    from    user_information    a
where .user_login_name=upper(p_user_login_name);
      if v_count>0 then
        p_out:=-3; --该用户名已经存在
        return ;
      end if;
      insert into user_information
                  values(seq_user_information.nextval,upper(p_user_login_name),
p_user_password,p_user_name,p_user_telephone,p_user_type, sysdate,sysdate);
      commit;
      p_out:=0; --操作成功
      return ;
    exception
      when others then
```

```
        p_out:=-4; --插入过程中出现异常
        return ;
    end ;
```

3、在.NET 项目建一个到数据库的联结：

在项目中新增加一个类文件。名称：clsPublic，

```
//添加引用：
using System;
using System.Data;
using System.Data.OleDb;
//连接字符串
private string connectora="Provider=MSDAORA.1;Password=fran;User ID=fran;Data Source=demo;Persist Security Info=True";
//连接 Oracle 数据库
public OleDbConnection ConnectDB()
{
  try
  {
    OleDbConnection conn=new OleDbConnection();
        conn.ConnectionString=connectora;
        conn.Open();
        return conn;
      }
      Catch
      {
        return null;
      }
    }
```

4、在类文件中添加如下内容，用来执行 Oracle 中的过程：

```
Public int Insert_User_Information(string v_user_login_name,string v_user_password,string v_user_name,string v_user_telephone,int v_user_type,string proc_name)
{
  int i;
  cmdOra.Parameters.Clear();
  cmdOra.CommandText=proc_name;
  cmdOra.CommandType=CommandType.StoredProcedure;
  cmdOra.Connection=new clsPublic().ConnectDB();
  cmdOra.Parameters.Add("p_user_login_name",OleDbType.VarChar);
  cmdOra.Parameters.Add("p_user_password",OleDbType.VarChar);
  cmdOra.Parameters.Add("p_user_name",OleDbType.VarChar);
  cmdOra.Parameters.Add("p_user_telephone",OleDbType.VarChar);
  cmdOra.Parameters.Add("p_user_type",OleDbType.Integer);
  cmdOra.Parameters.Add("p_out",OleDbType.Integer);
  cmdOra.Parameters["p_user_login_name"].Value=v_user_login_name;
```

```
  cmdOra.Parameters["p_user_password"].Value=v_user_password;
  cmdOra.Parameters["p_user_name"].Value=v_user_name;
  cmdOra.Parameters["p_user_telephone"].Value=v_user_telephone;
  cmdOra.Parameters["p_user_type"].Value=v_user_type;
  cmdOra.Parameters["p_user_login_name"].Direction=ParameterDirection.Input;
  cmdOra.Parameters["p_user_password"].Direction=ParameterDirection.Input;
  cmdOra.Parameters["p_user_name"].Direction=ParameterDirection.Input;
  cmdOra.Parameters["p_user_telephone"].Direction=ParameterDirection.Input;
  cmdOra.Parameters["p_user_type"].Direction=ParameterDirection.Input;
  cmdOra.Parameters["p_out"].Direction=ParameterDirection.ReturnValue;
  try
  {
    cmdOra.ExecuteNonQuery();
        i=(int)cmdOra.Parameters["p_out"].Value;
      }
      catch
      {
        i=-88;
      }
      finally
      {
        if(cmdOra.Connection.State==ConnectionState.Open)
      {
        cmdOra.Connection.Close();
        cmdOra.Connection.Dispose();
        cmdOra.Parameters.Clear();
        cmdOra.Dispose();
      }
      }
      return i;
}
```

5、在窗体界面中调用执行存储过程

```
private void button3_Click_1(object sender, System.EventArgs e)
{
                    int        i=new        clsPublic().Insert_User_Information
("dinya","111","DINYA","13877778888",0,
"cux_franchiser.insert_user_information"); MessageBox.Show(i.ToString());
}
```

在本例第二步中，定义一个输出参数用来存储执行的结果，Oracle 存储过程允许给输出参数直接赋值，在。NET 中设置该参数类型：

```
cmdOra.Parameters["p_out"].Direction=ParameterDirection.ReturnValue;
```

将其设置为返回值，这样在调用 Insert_User_Information 执行存储过程后，该参数将执行结果返回给用户。

在第五步窗体界面中调用存储过程的时候，最后一个参数为

"cux_franchiser.insert_user_information"，该值中 cux_franchiser 为自己定义的一个包，insert_user_information 为包中的过程。(具体对包的使用请参考 Oracle 相关书籍中有关包的使用一节)。

其中需要指出的是,用来连接 Oracle 数据库的联结串会因为 OleDB 的厂家不同而不同，本例中使用微软的，您在开发过程也可以使用 Oracle 公司的，可以到 Oralce 网站下载,地址: http://www.oracle.com/technology/software/tech/windows/ole_db/index.html。

4. Session

问：为什么 Session 在有些机器上偶尔会丢失?

答：可能和机器的环境有关系，比如：防火墙或者杀毒软件等，尝试关闭防火墙。

问：为什么当调用 Session.Abandon 时并没有激发 Session_End 方法?

答：首先 Session_End 方法只支持 InProc（进程内的）类型的 Session。其次要激发 Session_End 方法，必须存在 Session（即系统中已经使用 Session 了），并且至少要完成一次请求（在这次请求中会调用该方法）。

问：为什么当我在 InProc 模式下使用 Session 会经常丢失?

答：该问题通常是由于应用程序被回收导致的，因为当使用进程内 Session 时，Session 是保存在 aspnet_wp 进程中，当该进程被回收 Session 自然也就没有了，确定该进程是否被回收可以通过查看系统的事件查看器获得信息。

具体信息请参考：

Session variables are lost intermittently in ASP.NET applications

http://support.microsoft.com/default.aspx?scid=kb;en-us;Q316148

在 1.0 的时候也有一个 bug 会导致工作进程被回收并重启，该 bug 已经在 1.1 和 sp2 中修复。

关于该 bug 的详细信息请参考：

ASP.NET Worker Process (Aspnet_wp.exe) Is Recycled Unexpectedly.

http://support.microsoft.com/default.aspx?scid=kb;en-us;Q321792

问：为什么当 Session 超时或者 Abandoned 后，新 Session 的 ID 和原来的相同?

答：因为 SessionID 是保存在客户端浏览器的实例里，当 Session 超时在服务器重新建立 Session 时，将使用浏览器传来的 SessionID，所以当 Session 超时后，再重新建立后 SessionID 并不变。

问：为什么每次请求的 SessionID 都不相同?

答：该问题可能是没有在 Session 里面保存任何信息引起的，即程序中任何地方都没有使用 Session。当 Session 中保存信息之后 SessionID 将一直和浏览器相关，此时的 SessionID 将不会在变化。

问：ASP 和 ASP.NET 之间是否可以共享 Session?

答：可以。但是这是一个比较复杂的过程，微软提供了官方的解决方案，请参考：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/ConvertToASPNET.asp

问：什么类型的对象可以保存在 Session 里?

答：这依赖使用的 Session 的模式，当使用的是进程内（InProc）的 Session 那么可以轻松的保存任何对象。如果你使用了非 InProc 的模式，则只能保存可以序列化和反序列化的对象，如果此时保存的对象不支持序列化，则不能保存到这种模式（非 InProc）的 Session

里。
问：为什么在 Session_End 中不能使用 Response.Redirect 和 Server.Transfer 方法跳转页面？
答：Session_End 是一个在服务器内部激发的事件处理函数。它是基于一个服务器内部的计时器的，在激发该事件时服务器上并没有相关的 HttpRequest 对象，因此此时并不能使用 Response.Redirect 和 Server.Transfer 方法。
问：在 Session_End 中是否可以获得 HttpContext 对象？
答：不行，因为这个事件并没有和任何的请求（Request）相关联，没有基于请求的上下文。
问：在 Web Service 中该如何使用 Session？
答：为了在 Web Service 中使用 Session，需要在 Web Service 的调用方做一些额外的工作，必须保存和存储调用 Web Service 时使用的 Cookie。详细信息请参考 MSDN 文档的 HttpWebClientProtocol.CookieContainer 属性。然而，如果你使用代理服务器访问 Web Service 由于框架的限制，两者不能共享 Session。
问：在自定义自己的 HttpHandler 的时候，为什么不能使用 Session？
答：在实现自己的 HttpHandler 的时候，如果希望使用 Session 必须实现下面的两个标记接口中的一个： IRequiresSessionState 和 IReadOnlySessionState，这些接口没有任何方法需要实现，只是一个标记接口和使用 INamingContainer 接口的方法一样。
问：当我使用 webfarm 时，当我重定向到其他的 Web 服务器时 Session 为什么会丢失？
答：详细信息请参考：
PRB: Session State Is Lost in Web Farm If You Use SqlServer or StateServer Session Mode
http://support.microsoft.com/default.aspx?scid=kb;en-us;325056
问：为什么我的 Session 在 Application_OnAcquireRequestState 方法中无效？
答：Session 只有在 HttpApplication.AcquireRequestState 事件调用以后才会有效。
详细信息请参考：
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhandlingpublicevents.asp
问：如果使用了 cookieless，我该如何从 HTTP 页面定向到 HTTPS？
答：请尝试下面的方法：

```
String originalUrl = "/fxtest3/sub/foo2.aspx";
String modifiedUrl = "https://localhost" + Response.ApplyAppPathModifier(originalUrl);
Response.Redirect(modifiedUrl);
```

问：Session 在 global.asax 中的那些事件中有效？
答：Session 只有在 AcquireRequestState 事件之后有效，该事件之后的事件都可以使用 Session。
问：如何获得当前 Session 中保存的所有对象？
答：可以通过遍历所有的 Session.Keys 来获得。代码如下：

```
ArrayList sessionCollection = new ArrayList();
foreach (string strKey in Session.Keys){
  sessionCollection.Add(Session[strKey]);
}
```

问：是否可以在不同的应用程序中共享 Session？
答：不能直接共享。可以参考如何在 ASP 和 ASP.NET 之间共享 Session。

问：Session.Abandon 和 Session.Clear 有何不同？
答：主要的不同之处在于当使用 Session.Abandon 时，会调用 Session_End 方法（InProc 模式下）。当下一个请求到来时将激发 Session_Start 方法。而 Session.Clear 只是清除 Session 中的所有数据并不会中止该 Session，因此也不会调用那些方法。
问：为了可以顺序访问 Session 的状态值，Session 是否提供了锁定机制？
答：Session 实现了 Reader/Writer 的锁机制：
当页面对 Session 具有可写功能(即页面有<%@ Page EnableSessionState="True" %>标记)，此时直到请求完成该页面的 Session 持有一个写锁定。
当页面对 Session 具有只读功能（即页面有<%@ Page EnableSessionState="ReadOnly" %>标记），此时知道请求完成该页面的 Session 持有一个读锁定。
读锁定将阻塞一个写锁定；读锁定不会阻塞读锁定；写锁定将阻塞所有的读写锁定。这就是为什么两个框架中的同一个页面都去写同一个 Session 时，其中一个要等待另一个（稍快的那个）完成后，才开始写。
问：Session 平滑超时意味着什么？
答：Session 平滑超时意味着只要你的页面访问（使用）了 Session，超时时间将被刷新（可以理解为重新计时），即从该页面请求开始，将重新计算超时时间。但是，该页面不能禁用 Session。它会自动的访问当前页面的 Session，刷新超时时间。
问：在 global.asax 中的事件处理函数中 Session 为什么无效？
答：依赖于在哪个事件处理函数中使用 Session，Session 在 AcquireRequestState 事件之后才有效，该事件之后的所有事件处理函数都可以使用 Session，之前的则不能。
问：当我写一个依赖于当前应用的 Session 的组件时，为什么不能直接使用 Session["Key"]获得其值？
答：Session["Key"]实际上是 this.Session["Key"]，它是作为 Page 的一个属性提供的，所以在你的组件中不能直接使用这个属性。你可以通过下面的方式使用 Session：
HttpContext.Current.Session["Key"] = "My Seesion Value";
问：当我使用 InProc 模式保存 Session 时，此时的 Session 是保存在哪里？
答：不同的 IIS 的处理方式不同，
当使用的是 IIS5 的时候 Session 是保存在 aspnet_wp.exe 的进程空间里的。
当使用的是 IIS6 时，默认情况下所有的应用程序共享应用程序池，Session 保存在 w3wp.exe 的进程空间中。
问：Session 的超时设置是分钟还是秒？
答：是分钟，默认为 20 分钟。
问：当页面出现错误后我的 Session 是否将被保存？我需要在 Session_End 中处理一些清理工作，但是失败了，为什么？
答： Session_End 只有在 Session 运行在 InProc 模式下才会被执行。Session_End 使用的帐号是运行 aspnet_wp 工作进程的帐号（这个可以在 machine.config 中设置）。因此，如果在 Session_End 方法里，使用集成安全性链接到 SQL，它将使用 aspnet_wp 进程的帐号打开链接，此时成功与否则依赖于你的 SQL 的安全性设置。
问：为什么当我设置 cookieless 为 true 是我在重定向的时候会丢失 Session？
答：当使用 cookieless 时，你必须使用相对路径替换程序中的绝对路径，如果使用绝对路径 ASP.NET 将无法在 URL 中保存 SessionID。
例如：将\myDir\mySubdir\default.aspx 换成..\default.aspx 即可。
问：如何将 SortedList 存储到 Session 或者 Cache 里？
答：请参考下面的方法：

SortedList x = new SortedList();
x.Add("Key1", "ValueA");
x.Add("Key2", "ValueB");
保存到 Session 中:
Session["SortedList1"] = x;
使用下面方法获得之：
SortedList y = (SortedList) Session["SortedList1"];
Chahe 则同理。

问：我为什么会获得这样的错误信息“Session state can only be used when enableSessionState is set to true, either in a configuration file or in the Page directive”？

答：这个问题可能在一个已经安装了 Microsoft Visual Studio .NET 开发环境的机器上，再安装 Window Sharepoint Server（WSS）后出现。

WSS ISAPI 过滤器会处理所有的请求。当你通过虚拟目录浏览一个 ASP.NET 的应用程序时，ISAPI 过滤器不会给文件夹目录分配 URL。

解决方法是：不要再安装了 WSS 的机器上使用 Session。

详细信息请参考：

Session state cannot be used in ASP.NET with Windows SharePoint Services
http://support.microsoft.com/default.aspx?scid=kb;en-us;837376

问：如何删除 Session 变量?

答：想要删除 Session 变量可以使用 HttpSessionState.Remove()方法。

问：是否有办法知道应用程序的 Session 在运行时占用了多少内存？

答：没有。目前这个值时无法考证的，至少我现在还没有看到这方面的资料。但是可以通过性能监视器以及程序代码大概估算出来一个值。

问：当页面中是否了 frameset，发现在每个 frame 中显示页面的 SessionID 在第一次请求时都不相同，为什么？

答：原因是你的 frameset 是放在一个 htm 页面上而不是 ASPX 页面。

在一般情况下，如果 frameset 是 aspx 页面，当你请求页面时，它首先将请求发送到 Web 服务器，此时已经获得了 SessionID，接着浏览器会分别请求 Frame 中的其他页面，这样所有页面的 SessionID 就是一样的，就是 FrameSet 页面的 SessionID。

然而如果你使用 Html 页面做 FrameSet 页面，第一个请求将是 HTML 页面，当该页面从服务器上返回是并没有任何 Session 产生，接着浏览器会请求 Frame 里面的页面，这样这些页面都会产生自己的 SessionID，所以在这种情况下就会出现这种问题。当你重新刷新页面时，SessionID 就会一样，并且是最后一个请求页面的 SessionID。

问：是否可以将不同应用程序的 Session 保存在相同的 SQL Server 服务器的不同数据库上。

答：可以，请参考：

FIX: Using one SQL database for all applications for SQL Server session state may cause a bottleneck
http://support.microsoft.com/default.aspx?scid=kb;en-us;836680

问：在 Session_End 是我是否可以获得有效的 HttpSessionState 和 HttpContext 对象？

答：你可以在这个方法中获得 HttpSessionState 对象，可以直接使用 Session 来访问即可。但是不能获得 HttpContext 对象，因为该事件并没有和任何请求相关联，因此不存在上下文对象。

问：在 SQLServer 模式下使用 Session，为什么我的 Session 不过期？

答：在 SqlServer 模式下，Session 的过期是通过 SQL Agent 的注册工作完成的，请检查你的 SQL Agent 是否运行？
问：当我设置 EnableSessionState 为“ReadOnly”后，但是我在 InProc 模式下依然可以修改 Session 的值，这是为什么？
答：即使 EnableSessionState 标示为 ReadOnly，但是在 InProc 模式下用户依然可以编辑 Session。唯一不同的是，在请求过程中 Session 将不会被锁住。
问：我如何才能避免在链接 SQL 时指定密码？
答：使用信任链接或者使用加密的链接串。有关这方面的详细信息请参考：
How To Use the ASP.NET Utility to Encrypt Credentials and Session State Connection Strings
http://support.microsoft.com/default.aspx?scid=kb;en-us;329290
问：我在我自己的类中该如何使用 Session 呢？
答：可以使用 HttpContext.Current.Session 方式使用，具体方法如下：
HttpContext.Current.Session["SessionKey"] = "SessionValue";
类似的你还可以使用这种方式使用 Application 对象。
问：为什么在切换成 SQLServer 模式后我的请求被挂起了？
答：检查在 Session 里面是否都保存的是可以保存在 SQLServer 模式下的对象，即这些对象必须支持序列化。
问：当 Session 设置成 cookieless 后会有什么影响？
答：当把 cookieless 设置成 true 时，主要会有下面的约束：
1、在页面中不能使用绝对链接
2、在应用程序中在除了 Http 和 Https 之间的切换时需要完成一些其他的步骤。
如果发送一个链接给其他人，此时的 URL 里面将包含 Session ID 的信息，所以两个人将公用一个 Session。
问：是否可以将 Session 保存在数据库中？
答：当然可以，详细信息请参考：http://support.microsoft.com/default.aspx?scid=kb;en-us;311209
从 array 拷贝到 arrarylist 中
使用 String t[]={"1","2","3"}; List list=Arrays.asList(t); 就可以得到这个列表。

5. 密封类 sealed 关键字

密封类不能被继承，密封方法会重写基类中的方法，但本身不能在任何类中进一步重写

```
1   sealed class SealedClass
2    {
3        public int x;
4        public int y;
5     }
6     class MainClass
7    {
8        static void Main()
9        {
```

```
10              SealedClass sc = new SealedClass();
11              sc.y = 110;
12              sc.x = 150;
13              Console.WriteLine("x={0},y={1}", sc.x, sc.y);
14          }
15      }
```

密封类只要是为了防止第三方未经授权的扩展自身的类

6. Remoting

提供了一个功能强大、高效的处理远程对象的方法，从结构上而言，.NET Remote 对象非常适合通过网络访问资源，而又无需处理由基于 SOAP 的 WebServices 所带来的难题。.NET Remoting 使用起来比 Java 的 RMI 简单，但要比创建 Web Service 难度大一些。

在本篇文章中，我们将创建一个从数据库读入内容的远程对象。文中还包括了一个忽略数据库功能的替补对象，以使没有数据库可以使用的读者仍然能够使用.NET Remoting。

第一步：创建共享库

依次点击“文件”->“新创建”->“工程”，选择创建一个 C# Library，并将其命名为 ResumeServerLibrary，然后点击 OK 按钮。这将创建一个我们的.NET Remote 客户端和服务器端用来通讯的“共享命令集”。

正面是完整的代码，如果要跳过数据库访问部分，可以使用下面的代码替换 ResumeLoader 对象：

```
public class ResumeLoader : System.MarshalByRefObject
{
  public ResumeLoader()
  {
   System.Console.WriteLine("New Referance Added!");
}
public Resume GetResumeByUserID(decimal userID)
  {
return new Resume(1);
  }
}
```

名字空间是对象所需要的。请记住，如果得到 System.Runtime.Remoting.Channels.Tcp 名字空间不存在的信息，请检查是否象上面的代码那样添加了对 System.Runtime.Remoting.dll 的引用。

```
using System;
using System.Runtime;
using System.Data.SqlClient;
```

我们为对象使用的名字空间是 DotNetRemoteTest，下面的对象是 MarshalByRefObject，在其中我们创建了一个引用和包括服务器端数据库操作全部完成所需要的所有工作。

```
namespace DotNetRemoteTest
{
public class ResumeLoader : System.MarshalByRefObject
```

```
{
private SqlConnection dbConnection;
public ResumeLoader()
{
this.dbConnection = new System.Data.SqlClient.SqlConnection();
this.dbConnection.ConnectionString = "data source=GRIMSAADO2K;initial
catalog=underground;integrated security=SSPI;pers" + "ist security
info=True;workstation id=GRIMSAADO2K;packet size=4096";
/*具体的连接字符串会有所不同，这超出了本篇文章的范围。
如果不清楚如何创建一个数据库连接，请使用这一对象的另一个版本。*/
System.Console.WriteLine("New Referance Added!");
}
public Resume GetResumeByUserID(decimal userID)
 {
Resume resume = new Resume();
Try
{
dbConnection.Open();
SqlCommand cmd = new SqlCommand(
"SELECT ResumeID, UserID, Title,
  Body FROM Resume as theResume WHERE theResume.UserID="+ userID +"", dbConnection
);
SqlDataReader aReader = cmd.ExecuteReader();
if(aReader.Read())
{
resume.ResumeID=aReader.GetDecimal(0);
resume.UserID=aReader.GetDecimal(1);
resume.Title=aReader.GetString(2);
resume.Body=aReader.GetString(3);
}
aReader.Close();
dbConnection.Close();
}
catch(Exception x) { resume.Title="Error:"+x; }
return resume;
}
}
```

Resume 需要能够被串行化，以便能作为被远程调用的.NET Remote 对象的返回类型，原因是该对象将被转换为通过网络传输的原始数据，然后在网络的另一端再被装配成一个对象。该对象非常简单，为了使本篇文章看起来更简单，其中的构造器甚至使用缺省的内容初始化其中的一些域。

```
[Serializable]
 public class Resume
{
```

```
private decimal resumeID, userID;
private String body, title;
public Resume(decimal resumeID)
{
this.ResumeID=resumeID;
this.UserID=1;
this.Body="This is the default body of the resume";
this.Title="This is the default Title";
}
public decimal ResumeID
{
get { return resumeID; }
set { this.resumeID=value; }
}
public decimal UserID
{
get { return userID; }
set { this.userID=value; }
}
public String Body
{
get { return body; }
set { this.body=value;}
}
public String Title
{
get { return title; }
set { this.title=value; }
}
}//RESUME 对象结束
}//DotNetRemoteTest 名字空间结束
```

BR>编译创建的工程，就会得到一个 DLL 文件，并可以在其他的工程中使用它。

第二步：创建 Server 对象

有几种方法可以创建 Server 对象，最直观的方法是下面的方法：在 Visual Studio.NET 中，依次点击“文件”->“新创建”->“工程”，选择创建一个“Command Line Application”（命令行应用程序），并将它命名为 ResumeSuperServer。最最重要的是，我们需要添加对刚才在第一步中所创建的 DLL 文件的应用，该应用程序才能正确地运行。依次点击“工程”->“添加引用”，然后通过点击“浏览”按钮添加一个对在第一步中所创建的 DLL 文件的引用。为了使用.NET remote 功能，必须通过选择“工程”->“添加引用”，添加对 DLL 文件的引用。在.NET 标签中选择 System.Runtime.Remoting.DLL，然后点击“OK”按钮。然后，需要象我们在第一步中那样添加对 System.Runtime.Remoting.dll 的引用。下面的对象相当的简单和直观，我将就真正与.NET remoting 相关的 3 行代码中的每一行进行解释。TcpServerChannel 是.NET remoting 支持的二种信道类型中的一种，它将设置我们希望我们的对象对来自哪一个端口的请求进行回应，ChannelServices.RegisterChannel 将把该端口

号与操作系统中的 TCP/IP 栈绑定。

```
TcpServerChannel channel = new TcpServerChannel(9932);
ChannelServices.RegisterChannel(channel);
```

另一种可以设置的信道类型是 HTTP，只要简单地使用 System.Runtime.Remoting.Channels.Http 名字空间中的 HttpServerChannel 对象即可搞定。使用 HTTP 和 TCP 信道之间的区别可以简单的归结为：如果应用程序是在局域网上运行，则最好使用 TCP 信道，因为它的性能要好于 HTTP 信道；如果应用程序是在互联网上运行，则有时候根据防火墙的配置，HTTP 是唯一的选择。需要记住的是，如果使用了防火墙软件，则防火墙应该配置成允许 TCP 数据流量通过你为对象选择的端口。

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(ResumeLoader),
"ResumeLoader", WellKnownObjectMode.SingleCall);
```

这行代码设置了服务中的一些参数和把欲使用的对象名字与远程对象进行绑定，第一个参数是绑定的对象，第二个参数是 TCP 或 HTTP 信道中远程对象名字的字符串，第三个参数让容器知道，当有对对象的请求传来时，应该如何处理对象。尽管 WellKnownObjectMode.Single 对所有的调用者使用一个对象的实例，但它为每个客户生成这个对象的一个实例。完整的对象代码如下所示：

```
using System;
using System.Runtime;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Data.SqlClient;
using DotNetRemoteTest;
namespace ResumeServerServer
{
public class ResumeSuperServer
{
public static void Main(String[] args)
{
TcpServerChannel channel = new TcpServerChannel(9932);
ChannelServices.RegisterChannel(channel);
RemotingConfiguration.RegisterWellKnownServiceType(typeof(ResumeLoader),
"ResumeLoader", WellKnownObjectMode.SingleCall);
System.Console.WriteLine("Press Any Key");
System.Console.ReadLine();
}
}
}
```

编译这一程序并注意生成的.EXE 文件的位置。

第三步：创建 Remote 客户端程序 ResumeClinet 是我们为对在上面创建的 ResumeSuperServer 远和对象进行测试而创建的。要创建这一工程，可以依次点击“文件”->“创建”->“工程”，然后选择创建一个 Console Application 类型、名字为 ResumeClient 的工程名。象在第二步中那样，我们需要添加对在第一步中创建的 DLL 文件和 System.Runtime.Remoting DLL 的引用。下面的代码中有二行对于.NET remoting 而言是特别重要的。第一行创建了一个 TCP 客户端信道，该信道并不是绑定在一个端口上的；第二

行获取了一个对远程的 ResumeLoader 对象的引用。Activator.GetObject 方法返回一个对象类型的值，我们随后会将它返回的值赋予 ResumeLoader。我们传给它的参数与在服务器工程中传递给 RemotingConfiguration 的参数非常地相似，第一个参数是对象类型的，第二个参数是远程对象的 URI 。 ChannelServices.RegisterChannel(new TcpClientChannel());ResumeLoader loader = (ResumeLoader)Activator.GetObject(typeof(ResumeLoader), "tcp://localhost:9932/ResumeLoader");

ResumeClient 的全部代码如下所示：

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using DotNetRemoteTest;
namespace ResumeClient
{
public class ResumeClient
{
public static void Main(string[] args)
{
ChannelServices.RegisterChannel(new TcpClientChannel());
ResumeLoader loader = (ResumeLoader)Activator.GetObject(
typeof(ResumeServer), "tcp://localhost:9932/ResumeLoader");
if(rs==null)
{ Console.WriteLine("Unable to get remote referance"); }
Else
{
Resume resume = loader.GetResumeByUserID(1);
Console.WriteLine("ResumeID:"+ resume.ResumeID);
Console.WriteLine("UserID:"+ resume.UserID);
Console.WriteLine("Title:"+ resume.Title);
Console.WriteLine("Body:"+ resume.Body);
}
Console.ReadLine();//在能够看到结果前不让窗口关闭
}//END OF MAIN METHOD
}//END OF ResumeClient Object
}//END OF ResumeClientNamespace
```

测试

在数据库中创建一个具有如下结构的表：

```
Table Name-Resume
ResumeID, numeric (autonumber)
UserID, numeric
Title, Char(30)
Body, Text
```

双击我们在第二步中创建的 Server.exe，然后双击在第三步中创建的 Client 可执行文件。

如果一切正常的话，我们应该能够看到数据库中 ResumeID 的值为 1 的记录行。
总之，.NET Remoting 使用起来很简单，而且为处理局域网甚至互联网范围内的资源提供了一个绝佳的方法。

7. foreach

foreach 语句为数组或对象集合中的每个元素重复一个嵌入语句组 foreach 语句用于循环访问集合以获取所需信息，但不应用于更改集合内容以避免产生不可预知的副作用。此语句的形式如下：

foreach (type identifier in expression) statement

其中：

type

identifier 的类型。

identifier

表示集合元素的迭代变量。如果迭代变量为值类型，则无法修改的只读变量也是有效的。

expression

对象集合或数组表达式。集合元素的类型必须可以转换为 identifier 类型。请不要使用计算为 null 的表达式。

而应计算为实现 IEnumerable 的类型或声明 GetEnumerator 方法的类型。在后一种情况中，GetEnumerator 应该要么返回实现 IEnumerator 的类型，要么声明 IEnumerator 中定义的所有方法。

statement

要执行的嵌入语句。

在 c#中 foreach 与 for 的区别
for 和 foreach 有本质的区别
foreach 的具体过程是由目标对象控制的
所以并非集合内容更改了就不可以使用 foreach
而是 Array 和 Collection 对象如此实现了它而已
你却可以完全按照自己的想法去实现这一过程
也许你可以去翻一下 Gof 的 Visitor 模式，foreach 看起来就是它的简化版本
我们应该把 foreach 当作一种设计模式而不是简单的一个循环语句去理解
目前 C#里的 foreach 是以 IEnumerable 的方式去实现，这只是保留 COM 的实现方式
对集合使用 foreach
若要循环访问集合，集合必须满足特定的要求。例如，在下面的 foreach 语句中：

```
foreach (ItemType item in myCollection)
```

myCollection 必须满足下列要求：

集合类型：必须是 interface、class 或 struct。必须包括返回类型的名为 GetEnumerator 的实例方法。

Enumerator 类型（类或结构）必须包含：一个名为 Current 的属性，它返回 ItemType 或者可以转换为此类型的类型。属性访问器返回集合的当前元素。一个名为 MoveNext 的 bool 方法，它递增项计数器并在集合中存在更多项时返回 true。
有三种使用集合的方法：

使用上述指导创建一个集合。此集合只能用于 C# 程序。
使用上述指导创建一个一般集合，另外实现 IEnumerable 接口。此集合可用于其他语言(如 Visual Basic)。

8. 托管

自动内存管理是公共语言运行库在托管执行过程过程中提供的服务之一。公共语言运行库的垃圾回收器为应用程序管理内存的分配和释放。对开发人员而言，这就意味着在开发托管应用程序时不必编写执行内存管理任务的代码。自动内存管理可解决常见问题，例如，忘记释放对象并导致内存泄漏，或尝试访问已释放对象的内存。本节描述垃圾回收器如何分配和释放内存。

分配内存

初始化新进程时，运行时会为进程保留一个连续的地址空间区域。这个保留的地址空间被称为托管堆。托管堆维护着一个指针，用它指向将在堆中分配的下一个对象的地址。最初，该指针设置为指向托管堆的基址。托管堆上部署了所有引用类型。应用程序创建第一个引用类型时，将为托管堆的基址中的类型分配内存。应用程序创建下一个对象时，垃圾回收器在紧接第一个对象后面的地址空间内为它分配内存。只要地址空间可用，垃圾回收器就会继续以这种方式为新对象分配空间。

从托管堆中分配内存要比非托管内存分配速度快。由于运行时通过为指针添加值来为对象分配内存，所以这几乎和从堆栈中分配内存一样快。另外，由于连续分配的新对象在托管堆中是连续存储，所以应用程序可以快速访问这些对象。

释放内存

垃圾回收器的优化引擎根据所执行的分配决定执行回收的最佳时间。垃圾回收器在执行回收时，会释放应用程序不再使用的对象的内存。它通过检查应用程序的根来确定不再使用的对象。每个应用程序都有一组根。每个根或者引用托管堆中的对象，或者设置为空。应用程序的根包含全局对象指针、静态对象指针、线程堆栈中的局部变量和引用对象参数以及 CPU 寄存器。垃圾回收器可以访问由实时 (JIT) 编译器和运行时维护的活动根的列表。垃圾回收器对照此列表检查应用程序的根，并在此过程中创建一个图表，在其中包含所有可从这些根中访问的对象。

不在该图表中的对象将无法从应用程序的根中访问。垃圾回收器会考虑无法访问的对象垃圾，并释放为它们分配的内存。在回收中，垃圾回收器检查托管堆，查找无法访问对象所占据的地址空间块。发现无法访问的对象时，它就使用内存复制功能来压缩内存中可以访问的对象，释放分配给不可访问对象的地址空间块。在压缩了可访问对象的内存后，垃圾回收器就会做出必要的指针更正，以便应用程序的根指向新地址中的对象。它还将托管堆指针定位至最后一个可访问对象之后。请注意，只有在回收发现大量的无法访问的对象时，才会压缩内存。如果托管堆中的所有对象均未被回收，则不需要压缩内存。

为了改进性能，运行时为单独堆中的大型对象分配内存。垃圾回收器会自动释放大型对象的内存。但是，为了避免移动内存中的大型对象，不会压缩此内存。

级别和性能

为了优化垃圾回收器的性能，托管堆分为三个生成级别：0、1 和 2。运行时的垃圾回收算法基于以下几个普遍原理，这些垃圾回收方案的原理已在计算机软件业通过实验得到了证实。首先，压缩托管堆的一部分内存要比压缩整个托管堆速度快。其次，较新的对象生存期较短，而较旧的对象生存期则较长。最后，较新的对象趋向于相互关联，并且大致同时由应

用程序访问。
运行时的垃圾回收器将新对象存储在第 0 级托管堆中。在应用程序生存期的早期创建的对象如果未被回收，则被升级并存储在第 1 级和第 2 级托管堆中。对象升级的过程将在本主题的后面介绍。因为压缩托管堆的一部分要比压缩整个托管堆速度快，所以此方案允许垃圾回收器在每次执行回收时释放特定级别的内存，而不是整个托管堆的内存。
实际上，垃圾回收器在第 0 级托管堆已满时执行回收。如果应用程序在第 0 级托管堆已满时尝试新建对象，垃圾回收器将会发现第 0 级托管堆中没有可分配给该对象的剩余地址空间。垃圾回收器执行回收，尝试为对象释放第 0 级托管堆中的地址空间。垃圾回收器从检查第 0 级托管堆中的对象（而不是托管堆中的所有对象）开始执行回收。这是最有效的途径，因为新对象的生存期往往较短，并且期望在执行回收时，应用程序不再使用第 0 级托管堆中的许多对象。另外，单独回收第 0 级托管堆通常可以回收足够的内存，这样，应用程序便可以继续创建新对象。
垃圾回收器执行第 0 级托管堆的回收后，会压缩可访问对象的内存，如本主题前面的释放内存中所述。然后，垃圾回收器升级这些对象，并考虑第 1 级托管堆的这一部分。因为未被回收的对象往往具有较长的生存期，所以将它们升级至更高的级别很有意义。因此，垃圾回收器在每次执行第 0 级托管堆的回收时，不必重新检查第 1 级和第 2 级托管堆中的对象。
在执行第 0 级托管堆的首次回收并把可访问的对象升级至第 1 级托管堆后，垃圾回收器将考虑第 0 级托管堆的其余部分。它将继续为第 0 级托管堆中的新对象分配内存，直至第 0 级托管堆已满并需执行另一回收为止。这时，垃圾回收器的优化引擎会决定是否需要检查较旧的级别中的对象。例如，如果第 0 级托管堆的回收没有回收足够的内存，不能使应用程序成功完成创建新对象的尝试，垃圾回收器就会先执行第 1 级托管堆的回收，然后再执行第 0 级托管堆的回收。如果这样仍不能回收足够的内存，垃圾回收器将执行第 2、1 和 0 级托管堆的回收。每次回收后，垃圾回收器都会压缩第 0 级托管堆中的可访问对象并将它们升级至第 1 级托管堆。第 1 级托管堆中未被回收的对象将会升级至第 2 级托管堆。由于垃圾回收器只支持三个级别，因此第 2 级托管堆中未被回收的对象会继续保留在第 2 级托管堆中，直到在将来的回收中确定它们为无法访问为止。
为非托管资源释放内存
对于应用程序创建的大多数对象，可以依赖垃圾回收器自动执行必要的内存管理任务。但是，非托管资源需要显式清除。最常用的非托管资源类型是包装操作系统资源的对象，例如，文件句柄、窗口句柄或网络连接。虽然垃圾回收器可以跟踪封装非托管资源的托管对象的生存期，但却无法具体了解如何清理资源。创建封装非托管资源的对象时，建议在公共 Dispose 方法中提供必要的代码以清理非托管资源。通过提供 Dispose 方法，对象的用户可以在使用完对象后显式释放其内存。使用封装非托管资源的对象时，应该了解 Dispose 并在必要时调用它。有关清理非托管资源的更多信息和实现 Dispose 的设计模式示例，请参见垃圾回收。

9. 早期绑定和后期绑定

将对象分配给对象变量时，Visual Basic 编译器会执行一个名为 binding 的进程。如果将对象分配给声明为特定对象类型的变量，则该对象为“早期绑定”。早期绑定对象允许编译器在应用程序执行前分配内存以及执行其他优化。例如，下面的代码片段将一个变量声明为 FileStream 类型：

Visual Basic 复制代码

```
'  Create a variable to hold a new object.
Dim FS As System.IO.FileStream
' Assign a new object to the variable.
FS = New System.IO.FileStream("C:\tmp.txt", _
System.IO.FileMode.Open)
```

因为 FileStream 是一种特定的对象类型，所以分配给 FS 的实例是早期绑定。

相反，如果将对象分配给声明为 Object 类型的变量，则该对象为“后期绑定”。这种类型的对象可以存储对任何对象的引用，但没有早期绑定对象的很多优越性。例如，下面的代码段声明一个对象变量来存储由 CreateObject 函数返回的对象：

Visual Basic 复制代码

```
' To use this example, you must have Microsoft Excel installed on your computer.
' Compile with Option Strict Off to allow late binding.
Sub TestLateBinding()
    Dim xlApp As Object
    Dim xlBook As Object
    Dim xlSheet As Object
    xlApp = CreateObject("Excel.Application")
    ' Late bind an instance of an Excel workbook.
    xlBook = xlApp.Workbooks.Add
    ' Late bind an instance of an Excel worksheet.
    xlSheet = xlBook.Worksheets(1)
    xlSheet.Activate()
    ' Show the application.
    xlSheet.Application.Visible = True
    ' Place some text in the second row of the sheet.
    xlSheet.Cells(2, 2) = "This is column B row 2"
End Sub
```

早期绑定的优点

应当尽可能使用早期绑定对象，因为它们允许编译器进行重要优化，从而生成更高效的应用程序。早期绑定对象比后期绑定对象快很多，并且能通过确切声明所用的对象种类使代码更易于阅读和维护。早期绑定的另一个优点是它启用了诸如自动代码完成和动态帮助等有用的功能，这是因为 Visual Studio 集成开发环境 (IDE) 可以在您编辑代码时准确确定您所使用的对象类型。由于早期绑定使编译器可以在编译程序时报告错误，所以它减小了运行时错误的数量和严重度。

注意

后期绑定只能用于访问声明为 Public 的类型成员。访问声明为 Friend 或 Protected Friend 的成员会导致运行时错误。

10. c#与 vb 的语法区别

	C#	VB
1. 变量声	int x;	Dim x As Integer

名	String s; String s1, s2; Object o; Object obj = new Object(); public String name;	Dim s As String Dim s1, s2 As String Dim o 'Implicitly Object Dim obj As New Object() Public name As String
2. 注释语句	//…… /* …… ' …… */	'……
3. 获得 URL 传递的变量	String s = Request.QueryString["Name"]; String value = Request.Cookies["key"];	Dim s, value As String s = Request.QueryString("Name") value = Request.Cookies("Key").Value

4. 声明属性	public String name {
	get {
	...
	return ...;
	}
	set {
	... = value;
	}
	}
		Public Property Name As String
		Get
		...
		Return ...;
		End Get
		Set
		... = Value;
		End Set
		End Property

5. 数组	String[] a = new String[3]; a[0] = "1"; a[1] = "2"; a[2] = "3"; //二维数组 String[][] a = new String[3][3]; a[0][0] = "1"; a[1][0] = "2"; a[2][0] = "3";	Dim a(3) As String a(0) = "1" a(1) = "2" a(2) = "3" Dim a(3,3) As String a(0,0) = "1" a(1,0) = "2" a(2,0) = "3" Dim a() As String a(0,0) = "1" a(1,0) = "2" a(2,0) = "3" Dim a(,) As String a(0,0) = "1" a(1,0) = "2" a(2,0) = "3"

6. If 语句	if (Request.QueryString != null) {
	...
	}
		If Not (Request.QueryString = Null)
		...
		End If

7. 分支语句	switch (FirstName) {
	case "John" :
	...
	break;
	case "Paul" :
	...
	break;
	case "Ringo" :
	...
	break;
	}
		Select (FirstName)
		case "John" :
		...
		case "Paul" :
		...
		case "Ringo" :
		...
		End Select

8. For 循环	for (int i=0; i<3; i++) a(i) = "test";	Dim I As Integer For I = 0 To 2 step 1 a(I) = "test" Next

9. While 循环	int i = 0;
	while (i<3) {
	Console.WriteLine(i.ToString());
	i += 1;
	}
		Dim I As Integer
		I = 0
		Do While I < 3
		Console.WriteLine(I.ToString())
		I = I + 1
		Loop

10. 数据类型转换	int i = 3;
	String s = i.ToString();
	double d = Double.Parse(s);
		Dim i As Integer
		Dim s As String
		Dim d As Double
		i = 3
		s = i.ToString()
		d = CDbl(s)

11. 类的声明和继承	using System;
	namespace MySpace {
	public class Foo : Bar {
	int x;
	public Foo() { x = 4; }
	public void Add(int x) { this.x += x; }
	public int GetNum() { return x; }
	}
	}
		Imports System
		Namespace MySpace
		Public Class Foo : Inherits Bar
		Dim x As Integer
		Public Sub New()
		MyBase.New()
		x = 4
		End Sub
		Public Sub Add(x As Integer)
		Me.x = Me.x + x
		End Sub
		Public Function GetNum() As Integer
		Return x
		End Function
		End Class
		End Namespace

12. 声明类的主函数	using System;
	public class ConsoleCS {
	public ConsoleCS() {
	Console.WriteLine("Object
	Created");
	}
	public static void Main (String[]
	args) {
	Console.WriteLine("Hello
	World");
	ConsoleCS ccs = new ConsoleCS();
	}
	}
		Imports System
		Public Class ConsoleVB
		Public Sub New()
		M
		yBase.New()
		Console.WriteLine("Object
		Created")
		End Sub
		Public Shared Sub Main()
		Console.WriteLine("Hello World")
		Dim cvb As ConsoleVB
		cvb = New ConsoleVB()
		End Sub
		End Class

11. 反射（C# 编程指南）

反射提供了封装程序集、模块和类型的对象（Type 类型）。可以使用反射动态创建类型的实例，将类型绑定到现有对象，或从现有对象获取类型并调用其方法或访问其字段和属性。如果代码中使用了属性，可以利用反射对它们进行访问。有关更多信息，请参见属性。

下面是使用静态方法 GetType-- 从 Object 基类派生的所有类型都继承该方法 -- 获取变量类型的简单反射示例：

```
// Using GetType to obtain type information:
int i = 42;
System.Type type = i.GetType();
System.Console.WriteLine(type);
```

输出为：

```
System.Int32
```

此示例使用反射获取已加载的程序集的完整名称：

```
// Using Reflection to get information from an Assembly:
System.Reflection.Assembly o = System.Reflection.Assembly.Load("mscorlib.dll");
System.Console.WriteLine(o.GetName());
```

输出为：

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

反射概述

反射在下列情况下很有用：

需要访问程序元数据的属性。检查和实例化程序集中的类型。在运行时构建新类型。使用 System.Reflection.Emit 中的类。

执行后期绑定，访问在运行时创建的类型的方法。请参见主题动态加载和使用类型。

程序集包含模块，而模块包含类型，类型又包含成员。反射则提供了封装程序集、模块和类型的对象。您可以使用反射动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取类型。然后，可以调用类型的方法或访问其字段和属性。反射通常具有以下用途：

1.使用 Assembly 定义和加载程序集，加载在程序集清单中列出的模块，以及从此程序

集中查找类型并创建该类型的实例。

2.使用 Module 了解如下的类似信息：包含模块的程序集以及模块中的类等。您还可以获取在模块上定义的所有全局方法或其他特定的非全局方法。

3.使用 ConstructorInfo 了解如下的类似信息：构造函数的名称、参数、访问修饰符（如 public 或 private）和实现详细信息（如 abstract 或 virtual）等。

4.使用 Type 的 GetConstructors 或 GetConstructor 方法来调用特定的构造函数。

5.使用 MethodInfo 来了解如下的类似信息：方法的名称、返回类型、参数、访问修饰符(如 public 或 private)和实现详细信息(如 abstract 或 virtual)等。使用 Type 的 GetMethods 或 GetMethod 方法来调用特定的方法。

6.使用 FieldInfo 来了解如下的类似信息：字段的名称、访问修饰符（如 public 或 private）和实现详细信息（如 static）等；并获取或设置字段值。

7.使用 EventInfo 来了解如下的类似信息：事件的名称、事件处理程序数据类型、自定义属性、声明类型和反射类型等；并添加或移除事件处理程序。

8.使用 PropertyInfo 来了解如下的类似信息：属性的名称、数据类型、声明类型、反射类型和只读或可写状态等；并获取或设置属性值。

9.使用 ParameterInfo 来了解如下的类似信息：参数的名称、数据类型、参数是输入参数还是输出参数，以及参数在方法签名中的位置等。

12. 使用.NET 访问 MySql 数据库

在.NET 中访问 MySQL 数据库却并没有想象中那么方便,因为.NET OleDb Data Provider 和 MySQL 的兼容性并不理想。本文介绍了两种在.NET 中访问 MySQL 数据库的方法，并对这两种方法做了一个简单的性能对比。 引言 如果你不是只在大集团公司工作过的话，你一定会有机会接触到 MySQL，虽然它并不支持事务处理，存储过程，但是它提供的功能一定能满足你的大部分需求，另外，简洁的 MySQL 也有一些它独到的优势，在有些时候，它的速度甚至超过大型数据库。 那么如何在.NET 中访问 MySQL 数据库呢？也许很多人马上会说：用 OLEDB 嘛，但是事实上采用.NET OleDb Data Provider 并不能访问 MySQL，如果你使用的话，系统会提示你："Net Data OLE DB 提供程序 (System.Data.Odbc) 不支持 MSDASQL 提供程序（用于 Odbc 驱动程序的 Microsoft OLE DB 提供程序）。"，是什么原因我并不知道，按照 MySQLDriverCS 的作者的说法就是它被"abandoned by the owner"，呵呵，兴许还有些故事。 幸好，我们还有其它的选择，这里就要介绍两种访问 MySQL 数据库的办法。 使用 ODBC.NET ODBC.NET（全称 ODBC .NET Data Provider）是一个免费的.NET Framework 附加组件，需要到微软公司的网站上去下载，下载地址为：http://download.microsoft.com/download/dasdk/Install/1.0.4030.0/W98NT42KMeXP/EN-US/odbc_net.msi，它需要系统已经安装 MDAC 2.7 或者更高版本。另外，还需要安装 MySQL 的 ODBC 驱动程序，下载地址为：http://www.mysql.com/downloads/api-myodbc-2.50.html，还需要在"ODBC 数据源管理器"中配置一下 DSN。 在对象的设计上，ODBC.NET 也跟 OLEDB，SQL 等一样，分别为 OdbcConnection, OdbcCommand, OdbcDataAdapter, OdbcDataReader,用法也完全一样，如果你希望用 ODBC .NET 来代替以前的 OleDb .NET Data Provider，事实上完全可以通过查找替换的办法来修改你的程序。 以下是一段代码示例：

```
try
{
```

```
    string constr = "DSN=MySQL;" + "UID=;" +"PWD="; ;
    conn =  new OdbcConnection(constr);
    conn.Open();
    string query = "insert into test.dbtable values10,'disksidkfsdi',
'asdfaf', 'adsfasdf')";
    string tmp = null;
    OdbcCommand cmd = new OdbcCommand(query, conn);
    for(int i = 0; i < 100000; i++)
    {
        cmd.ExecuteNonQuery();
    }
    cmd.Dispose();
    conn.Close();
    query = "select * from test.dbtable";
    OdbcCommand cmd2 = newOdbcCommand(query, conn);
    conn.Open();
    OdbcDataReader reader = cmd2.ExecuteReader();
    while(reader.Read())
    {
        tmp = reader[0].ToString();
        tmp = reader[1].ToString();
        tmp = reader[2].ToString();
        tmp = reader[3].ToString();
    }
    conn.Close();
    query = "delete from test.dbtable";
    OdbcCommand cmd3 = newOdbcCommand(query, conn);
    conn.Open();
    cmd3.ExecuteNonQuery();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
    finally
    {
    conn.Close();
}
```

只要是用 C#写过数据库应用的人一定能知道，上面的代码执行了十万次插入数据和读取数据，最后将数据记录全部删除的操作。MySQLDriverCS 是 MySQL 数据库的一个免费开源的.NET 驱动程序。和 Sql .NET Data Provider 是为 Sql Server 一样，它是专门为 MySQL 设计的，可以叫做 MySQL .NET Data Provider。使用他不需要额外的去设置 ODBC 数据源，基本上只要能连接到 MySQL 就能通过 MySQLDriverCS 来访问。 MySQLDriverCS 是 SourceForge.NET 上的一个项目，不过不知道什么原因，这个网站在国内访问不到。 下面是

使用 MySQLDriverCS 的代码示例：

```
MySQLConnection conn = null;
try
{
    string    connstr    =    "Data    Source=MySQL;Password=root;User
ID=root;Location=localhost";
    conn =  new MySQLConnection(constr);
    conn.Open();
    string query = "insert into test.dbtable values(10, 'disksidkfsdi',
'asdfaf', 'adsfasdf')";
    string tmp = null;
    MySQLCommand cmd = new MySQLCommand(query, conn);
    for(int i = 0; i < 100000; i++)
    {
             cmd.ExecuteNonQuery();
    }
    cmd.Dispose();
    conn.Close();
    query = "select * from test.dbtable";
    MySQLCommand cmd2 = new MySQLCommand(query, conn);
    conn.Open();
    MySQLDataReader reader = cmd2.ExecuteReaderEx();
    while(reader.Read())
    {
        tmp = reader[0].ToString();
        tmp = reader[1].ToString();
        tmp = reader[2].ToString();
        tmp = reader[3].ToString();
    }
    conn.Close();
    query = "delete from test.dbtable";
    MySQLCommand cmd3 = new MySQLCommand(query, conn);
    conn.Open();
    cmd3.ExecuteNonQuery();
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
finally
{
    conn.Close();
}
```

和上面的那段代码几乎一模一样，所不同的是 Odbc 变成了 MySQL，另外，需要注意的一点是 Command 的 ExecuteReader 方法在 MySQLDriverCS 中变成了 ExecuteReaderEx，还有些细微的差别请参考附带的文档详细的介绍。性能测试 有些读者其实已经看出来我以上写的那段代码的用意，对了，其实目的就是用来进行性能测试的。以上两段代码的执行时间分别是：ODBC.NET 为 24 秒左右，MySQLDriverCS 为 17 秒左右。结果并不出人意外，作为 MySQL 的专用数据驱动程序，MySQLDriverCS 的速度大大快于 ODBC.NET 是在情理之中的。 总结 本文介绍了两种 MySQL 数据库访问的方法，同时对它们的性能做了一个简单的测试，希望能为各位读者在采用 MySQL 数据库开发.NET 应用的时候提

第五部分 数据库

A. 笔试面试题集

1. 选择题

1) 下面叙述正确的是_____
A、算法的执行效率与数据的存储结构无关
B、算法的空间复杂度是指算法程序中指令(或语句)的条数
C、算法的有穷性是指算法必须能在执行有限个步骤之后终止
D、以上三种描述都不对

2) 以下数据结构中不属于线性数据结构的是_____。
A、队列 B、线性表 C、二叉树 D、栈

3) 在一棵二叉树上第 5 层的结点数最多是_____。
A、8 B、16 C、32 D、15

4) 下面描述中，符合结构化程序设计风格的是_____。
A、使用顺序、选择和重复(循环)三种基本控制结构表示程序的控制逻辑
B、模块只有一个入口，可以有多个出口
C、注重提高程序的执行效率 D、不使用 goto 语句

5) 下面概念中，不属于面向对象方法的是_____。
A、对象 B、继承 C、类 D、过程调用

6) 在结构化方法中，用数据流程图(DFD)作为描述工具的软件开发阶段是_____。
A、可行性分析 B、需求分析 C、详细设计 D、程序编码

(7. 在软件开发中，下面任务不属于设计阶段的是_____。
A、数据结构设计 B、给出系统模块结构 C、定义模块算法 D、定义需求并建立系统模型

(8. 数据库系统的核心是_____。
A、数据模型 B、数据库管理系统 C、软件工具 D、数据库

(9. 下列叙述中正确的是_____。
A、数据库是一个独立的系统，不需要操作系统的支持
B、数据库设计是指设计数据库管理系统
C、数据库技术的根本目标是要解决数据共享的问题
D、数据库系统中，数据的物理结构必须与逻辑结构一致

(10. 下列模式中，能够给出数据库物理存储结构与物理存取方法的是_____。
A、内模式 B、外模式 C、概念模式 D、逻辑模式

(11. Visual FoxPro 数据库文件是_____。
A、存放用户数据的文件 B、管理数据库对象的系统文件
C、存放用户数据和系统的文件 D、前三种说法都对

(12. SQL 语句中修改表结构的命令是_____。
A、MODIFY TABLE B、MODIFY STRUCTURE C、ALTER TABLE D、ALTER STRUCTURE

(13. 如果要创建一个数据组分组报表，第一个分组表达式是"部门"，第二个分组表达式是"性别"，第三个分组表达式是"基本工资"，当前索引的索引表达式应当是_____。

A、部门+性别+基本工资 B、部门+性别+STR(基本工资)

C、STR(基本工资)+性别+部门 D、性别+部门+STR(基本工资)

(14. 把一个项目编译成一个应用程序时，下面的叙述正确的是_____。

A、所有的项目文件将组合为一个单一的应用程序文件

B、所有项目的包含文件将组合为一个单一的应用程序文件

C、所有项目排除的文件将组合为一个单一的应用程序文件

D、由用户选定的项目文件将组合为一个单一的应用程序文件

(15. 数据库 DB、数据库系统 DBS、数据库管理系统 DBMS 三者之间的关系是_____。

A、DBS 包括 DB 和 DBMS B、DBMS 包括 DB 和 DBS

C、DB 包括 DBS 和 DBMS D、DBS 就是 DB，也就是 DBMS

(16. 在"选项"对话框的"文件位置"选项卡中可以设置_____。

A、表单的默认大小 B、默认目录

C、日期和时间的显示格式 D、程序代码的颜色

(17. 要控制两个表中数据的完整性和一致性可以设置"参照完整性"，要求这两个表_____。

A、是同一个数据库中的两个表 B、不同数据库中的两个表

C、两个自由表 D、一个是数据库表另一个是自由表

(18. 定位第一条记录上的命令是_____。

A、GO TOP B、GO BOTTOM C、GO 6 D、SKIP

(19. 在关系模型中，实现"关系中不允许出现相同的元组"的约束是通过_____。

A、候选键 B、主键 C、外键 D、超键

(20. 设当前数据库有 10 条记录(记录未进行任何索引)，在下列三种情况下，当前记录号为 1 时;EOF()为真时;BOF()为真时，命令?RECN()的结果分别是_____。

A、1,11,1 B、1,10,1 C、1,11,0 D、1,10,0

(21. 下列表达式中结果不是日期型的是_____。

A、CTOD("2000/10/01") B、{^99/10/01}+365 C、VAL("2000/10/01") D、DATE()

(22. 只有满足联接条件的记录才包含在查询结果中，这种联接为_____。

A、左联接 B、右联接 C、内部联接 D、完全联接

(23. 索引字段值不唯一，应该选择的索引类型为_____。

A、主索引 B、普通索引 C、候选索引 D、唯一索引

(24. 执行 SELECT 0 选择工作区的结果是_____。

A、选择了 0 号工作区 B、选择了空闲的最小号工作区

C、关闭选择的工作区 D、选择已打开的工作区

(25. 从数据库中删除表的命令是_____。

A、DROP TABLE B、ALTER TABLE C、DELETE TABLE D、USE

(26. DELETE FROM S WHERE 年龄>60 语句的功能是_____。

A、从 S 表中彻底删除年龄大于 60 岁的记录 B、S 表中年龄大于 60 岁的记录被加上删除标记

C、删除 S 表 D、删除 S 表的年龄列

(27. SELECT-SQL 语句是_____。

A、选择工作区语句 B、数据查询语句 C、选择标准语句 D、数据修改语句

(28. SQL 语言是_____语言。A、层次数据库 B、网络数据库 C、关系数据库 D、非

数据库

(29. 在 SQL 中，删除视图用_____。

A、DROP SCHEMA 命令 B、CREATE TABLE 命令 C、DROP VIEW 命令 D、DROP INDEX 命令

(30. 以下属于非容器类控件的是_____。A、Form B、Label C、page D、Container

(31. 将查询结果放在数组中应使用_____短语。

A、INTO CURSOR B、TO ARRAY C、INTO TABLE D、INTO ARRAY

(32. 在命令窗口执行 SQL 命令时，若命令要占用多行，续行符是_____。

A、冒号(:) B、分号(;) C、逗号(,) D、连字符(-)

(33. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

对于图书管理数据库，查询 0001 号借书证的读者姓名和所借图书的书名。

SQL 语句正确的是_____。

SELECT 姓名,书名 FROM 借阅,图书,读者 WHERE;

借阅.借书证号="0001" AND;

_____

_____

A、图书.总编号=借阅.总编号 AND;

读者.借书证号=借阅.借书证号

B、图书.分类号=借阅.分类号 AND;

读者.借书证号=借阅.借书证号

C、读者.总编号=借阅.总编号 AND;

读者.借书证号=借阅.借书证号

D、图书.总编号=借阅.总编号 AND;

(34. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

对于图书管理数据库，分别求出各个单位当前借阅图书的读者人次。下面的 SQL 语句正确的是_____。

SELECT 单位,_____ FROM 借阅,读者 WHERE;

借阅.借书证号=读者.借书证号 _____

A、COUNT(借阅.借书证号) GROUP BY 单位 B、SUM(借阅.借书证号) GROUP BY 单位

C、COUNT(借阅.借书证号) ORDER BY 单位 D、COUNT(借阅.借书证号) HAVING 单位

（35. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

对于图书管理数据库，检索借阅了《现代网络技术基础》一书的借书证号。下面 SQL 语句正确的是_____。

SELECT 借书证号 FROM 借阅 WHERE 总编号=;

_____

A、(SELECT 借书证号 FROM 图书 WHERE 书名="现代网络技术基础")

B、(SELECT 总编号 FROM 图书 WHERE 书名="现代网络技术基础")

C、(SELECT 借书证号 FROM 借阅 WHERE 书名="现代网络技术基础")

D、(SELECT 总编号 FROM 借阅 WHERE 书名="现代网络技术基础")

（36. 以下数据结构中不属于线性数据结构的是_____。

A、队列 B、线性表 C、二叉树 D、栈

（37. 在结构化方法中，用数据流程图(DFD)作为描述工具的软件开发阶段是_____。

A、可行性分析 B、需求分析 C、详细设计 D、程序编码

（38. 结构化程序设计主要强调的是_____。

A、程序的规模 B、程序的易读性 C、程序的执行效率 D、程序的可移植性

（39. 在软件生命周期中，能准确地确定软件系统必须做什么和必须具备哪些功能的阶段是_____。

A、概要设计 B、详细设计 C、可行性分析 D、需求分析

（40. 下列关于栈的叙述中正确的是_____。A、在栈中只能插入数据 B、在栈中只能删除数据

C、栈是先进先出的线性表 D、栈是先进后出的线性表

（41. 下面不属于软件设计原则的是_____。A、抽象 B、模块化 C、自底向上 D、信息隐蔽

（42. 对长度为 N 的线性表进行顺序查找，在最坏情况下所需要的比较次数为_____。

A、N+1 B、N C、(N+1)/2 D、N/2

（43. 视图设计一般有 3 种设计次序，下列不属于视图设计的是_____。

A、自顶向下 B、由外向内 C、由内向外 D、自底向上

（44. 下列有关数据库的描述，正确的是_____。A、数据库是一个 DBF 文件 B、数据库是一个关系

C、数据库是一个结构化的数据集合 D、数据库是一组文件

（45. 下列说法中，不属于数据模型所描述的内容的是_____。

A、数据结构 B、数据操作 C、数据查询 D、数据约束

（46. 在下面的 Visual FoxPro 表达式中，运算结果是逻辑真的是_____。

A、EMPTY(.NULL.) B、LIKE('acd','ac?') C、AT('a','123abc') D、EMPTY(SPACE(2))

（47. 表达式 VAL(SUBS("奔腾 586",5,1))*Len("visual foxpro")的结果是_____。

A、13.00 B、14.00 C、45.00 D、65.00

（48. 以下关于自由表的叙述，正确的是_____。

A、全部是用以前版本的 FOXPRO(FOXBASE)建立的表

B、可以用 Visual FoxPro 建立，但是不能把它添加到数据库中

C、自由表可以添加到数据库中，数据库表也可以从数据库中移出成为自由表

D、自由表可以添加到数据库中，但数据库表不可从数据库中移出成为自由表

（49. 下面关于数据环境和数据环境中两个表之间的关系的陈述中，_____是正确的。

A、数据环境是对象，关系不是对象 B、数据环境不是对象，关系是对象

C、数据环境是对象，关系是数据环境中的对象 D、数据环境和关系均不是对象

（50. 在"报表设计器"中，可以使用的控件是_____。

A、标签、域控件和线条 B、标签、域控件和列表框

C、标签、文本框和列表框 D、布局和数据源

（51. 用二维表数据来表示实体及实体之间联系的数据模型称为_____。

A、实体--联系模型 B、层次模型 C、网状模型 D、关系模型

（52. 用来指明复选框的当前选中状态的属性是_____。A、Selected B、Caption C、Value D、ControlSource

（53. 使用菜单操作方法打开一个在当前目录下已经存在的查询文件 zgjk.qpr 后，在命令窗口生成的命令是____。

A、OPEN QUERY zgjk.qpr B、MODIFY QUERY zgjk.qpr

C、DO QUERY zgjk.qpr D、CREATE QUERY zgjk.qpr

（54. 可以伴随着表的打开而自动打开的索引是_____。

A、单一索引文件(IDX) B、复合索引文件(CDX)C、结构化复合索引文件 D、非结构化复合索引文件

（55. 在数据库设计器中，建立两个表之间的一对多联系是通过以下索引实现的_____。

A、"一方"表的主索引或候选索引，"多方"表的普通索引

B、"一方"表的主索引，"多方"表的普通索引或候选索引

C、"一方"表的普通索引，"多方"表的主索引或候选索引

D、"一方"表的普通索引，"多方"表的候选索引或普通索引

（56. 下列函数中函数值为字符型的是_____。 A、DATE() B、TIME() C、YEAR() D、DATETIME()

（57. 下面对控件的描述正确的是_____。

A、用户可以在组合框中进行多重选择 B、用户可以在列表框中进行多重选择

C、用户可以在一个选项组中选中多个选项按钮 D、用户对一个表单内的一组复选框只能选中其中一个

（58. 确定列表框内的某个条目是否被选定应使用的属性是_____。

A、Value B、ColumnCount C、ListCount D、Selected

（59.. 设有关系 R1 和 R2，经过关系运算得到结果 S，则 S 是_____。

A、一个关系 B、一个表单 C、一个数据库 D、一个数组

（60. DBAS 指的是_____。A、数据库管理系统 B、数据库系统 C、数据库应用系统 D、数据库服务系统

（61. 设 X="ABC"，Y="ABCD"，则下列表达式中值为.T.的是_____。A、X=Y B、X==Y C、X$Y D、AT(X,Y)=0

（62.在表结构中，逻辑型、日期型、备注型字段的宽度分别固定为_____。

A、3，8，10 B、1，6，4 C、1，8，任意 D、1，8，4

（63. 在标准 SQL 中，建立视图的命令是_____。

A、CREATE SCHEMA 命令 B、CREATE TABLE 命令 C、CREATE VIEW 命令 D、CREATE INDEX 命令

（64. 有关 SCAN 循环结构，叙述正确的是_____。

A、SCAN 循环结构中的 LOOP 语句，可将程序流程直接指向循环开始语句 SCAN，首先判断 EOF()函数的真假

B、在使用 SCAN 循环结构时，必须打开某一个数据库

C、SCAN 循环结构的循环体中必须写有 SKIP 语句

D、SCAN 循环结构，如果省略了子句\FOR 和 WHILE 条件子句，则直接退出循环

（65. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

对于图书管理数据库，要查询所藏图书中，各个出版社的图书最高单价、平均单价和册数，下面 SQL 语句正确的是_____。

SELECT 出版单位,_____,_____,_____;

FROM 图书管理!图书 _____ 出版单位

A、MIN(单价) AVGAGE(单价) COUNT(*) GROUP BY B、MAX(单价) AVG(单价) COUNT(*) ORDER BY

C、MAX(单价) AVG(单价) SUM(*) ORDER BY D、MAX(单价) AVG(单价) COUNT(*) GROUP BY

（66. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

对于图书管理数据库，求 CIE 单位借阅图书的读者的人数。

下面 SQL 语句正确的是_____。

SELECT _____ FROM 借阅 WHERE;

借书证号 _____

A、COUNT (DISTINCT 借书证号)

IN (SELECT 借书证号 FROM 读者 WHERE 单位="CIE")

B、COUNT (DISTINCT 借书证号)

IN (SELECT 借书证号 FROM 借阅 WHERE 单位="CIE")

C、SUM (DISTINCT 借书证号)

IN (SELECT 借书证号 FROM 读者 WHERE 单位="CIE")

D、SUM (DISTINCT 借书证号)

IN (SELECT 借书证号 FOR 借阅 WHERE 单位="CIE")

（67. 查询订购单号(字符型，长度为 4)尾字符是"1"的错误命令是_____。

A、SELECT * FROM 订单 WHERE SUBSTR(订购单号,4)="1"

B、SELECT * FROM 订单 WHERE SUBSTR(订购单号,4,1)="1"

C、SELECT * FROM 订单 WHERE "1"$订购单号

D、SELECT * FROM 订单 WHERE RIGHT(订购单号,1)="1"

（68. 在关系模型中，为了实现"关系中不允许出现相同元组"的约束应使用_____。

A、临时关键字 B、主关键字 C、外部关键字 D、索引关键字

（69. 根据"职工"项目文件生成 emp_sys.exe 应用程序的命令是_____。

A、BUILD EXE emp_sys FROM 职工 B、BUILD APP emp_sys.exe FROM 职工

C、LIKE EXE emp_sys FROM 职工 D、LIKE APP emp_sys.exe FROM 职工

（70. 当前盘当前目录下有数据库：学院.dbc，其中有"教师"表和"学院"表。

"教师"表：

"学院"表：

有 SQL 语句：

SELECT DISTINCT 系号 FROM 教师 WHERE 工资>=;

ALL (SELECT 工资 FROM 教师 WHERE 系号="02")

与如上语句等价的 SQL 语句是_____。

A、SELECT DISTINCT 系号 FROM 教师 WHERE 工资>=;

(SELECT MAX(工资) FROM 教师 WHERE 系号="02")

B、SELECT DISTINCT 系号 FROM 教师 WHERE 工资>=;
(SELECT MIN(工资) FROM 教师 WHERE 系号="02")
C、SELECT DISTINCT 系号 FROM 教师 WHERE 工资>=;
ANY(SELECT 工资 FROM 教师 WHERE 系号="02")
D、SELECT DISTINCT 系号 FROM 教师 WHERE 工资>=;
SOME (SELECT 工资 FROM 教师 WHERE 系号="02")

答案：1-5 CCBAD 6-10 BDBCA 11-15 DCBAA 16-20 BAABA 21-25 CCBBA 26-30 BBCCB 31-35 DDAAB 36-40 CBBDD 41-45 CBBCC 46-50 DDCCA 51-55 DCBCA 56-60 BBDAC 61-65 CDCBD 66-70 ACBBA

2. 填空

（1. 算法的复杂度主要包括___时间___复杂度和空间复杂度。

（2. 数据的逻辑结构在计算机存储空间中的存放形式称为数据的___模式或逻辑模式___。

（3. 若按功能划分，软件测试的方法通常分为白盒测试方法和___黑盒___测试方法。

（4. 如果一个工人可管理多个设施，而一个设施只被一个工人管理，则实体"工人"与实体"设备"之间存在___一对多 或 1 对多 或 一对 n 或 1：N 或 1:n 或 1：n 或 1:N 或 一对 m 或 1：M 或 1:m 或 1：m 或 1:N ___联系。

（5. 关系数据库管理系统能实现的专门关系运算包括选择、连接和__投影____。

（6. 命令?LEN("THIS IS MY BOOK")的结果是__15____。

（7. SQL SELECT 语句为了将查询结果存放到临时表中应该使用___ Into cursor 或 Into cursor cursorname ___短语。

（8. 多栏报表的栏目数可以通过__页面设置 或 列数____来设置。

44. 在打开项目管理器之后再打开"应用程序生成器"，可以通过按 ALT+F2 键，快捷菜单和"工具"菜单中的___应用程序生成器___。

（9. 数据库系统的核心是__数据库管理系统 或 DBM____。

（10. 查询设计器中的"联接"选项卡，可以控制___联接类型 或 联接条件___选择。

（11. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

用 SQL 的 CREATE 命令建立借阅表(字段顺序要相同)，请对下面的 SQL 语句填空：

____ CREATE TABLE 借阅 (借书证号 C(4),总编号 C(6),借书日期 D(8)) 或 CREA TABL 借阅 (借书证号 C(4),总编号 C(6),借书日期 D(8)) 或 CREATE TABLE 借阅 (借书证号 C(4),总编号 C(6),借书日期 D) 或 CREA TABL 借阅 (借书证号 C(4),总编号 C(6),借书日期 D) __

（12. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

对图书管理数据库，查询由"清华大学出版社"或"电子工业出版社"出版，并且单价不超出 20 元的书名。请对下面的 SQL 语句填空：

SELECT 书名,出版单位,单价 FROM 图书;

WHERE____单价<=20 或 (出版单位="清华大学出版社" OR 出版单位="电子工业出版社") 或 (出版单位="电子工业出版社" OR 出版单位="清华大学出版社") 或 (出版单位='清华大学出版社' OR 出版单位='电子工业出版社')___ AND;

___ (出版单位="清华大学出版社" OR 出版单位="电子工业出版社") 或 (出版单位='清华大学出版社‘)____

（13. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

对图书管理数据库，求共借出多少种图书。请对下面的 SQL 语句填空：

SELECT ___ COUNT(DISTINCT 总编号) 或 COUN(DISTINCT 总编号) 或 COUNT(DIST 总编号) 或 COUN(DIST 总编号)____ FROM 借阅

（14. 数据库系统的三级模式分别为___概念或概念级___模式、内部级模式与外部级模式。

（15. 关系模型的数据操纵即是建立在关系上的数据操纵，一般有___查询___、增加、删除和修改四种操作。

（16. TIME( )的返回值的数据类型是___字符 或 C ___类型。

（17. 在定义字段有效性规则中，在规则框中输入的表达式中类型是___逻辑表达式_____。

（18. 设计报表通常包括两部分内容：___.数据源___和布局。

（19. ___内部联接__是指只有满足联接条件的记录才包含在查询结果中。

（20. 设有图书管理数据库：

图书(总编号 C(6),分类号 C(8),书名 C(16),作者 C(6),出版单位 C(20),单价 N(6,2))

读者(借书证号 C(4),单位 C(8),姓名 C(6),性别 C(2),职称 C(6),地址 C(20))

借阅(借书证号 C(4),总编号 C(6),借书日期 D(8))

检索书价在 15 元至 25 元(含 15 元和 25 元)之间的图书的书名、作者、书价和分类号，结果按分类号升序排序。

SELECT 书名,作者,单价,分类号 FROM 图书;

WHERE_____;

ORDER BY_____;

答案：单价 BETWEEN 15 AND 25 或 单价 BETW 15 AND 25 或 单价 BETWE 15 AND 25 或 单价>=15 and 单价<=25 或 单价>=15 and 单价=<25 或 单价=>15 and 单价<=25 或 单价=>15 and 单价=<25 与 分类号 ASC 或 分类号

（21. 设有如下关系表 R、S 和 T：

R(BH,XM,XB,DWH)

S(SWH,DWM)

T(BH,XM,XB,DWH)

实现 R∪T 的 SQL 语句是___ SELECT * FROM R UNION SELECT * FROM T 或 SELE * FROM R UNIO SELE * FROM T 或 SELECT * FROM R UNIO SELECT * FROM T 或 SELE * FROM R UNION SELE * FROM T ____。

（22. 设有如下关系表 R：

R(NO,NAME,SEX,AGE,CLASS)

主关键字是 NO

其中 NO 为学号，NAME 为姓名，SEX 为性别，AGE 为年龄，CLASS 为班号。写出实现下列功能的 SQL 语句。

插入"95031"班学号为 30，姓名为"郑和"的学生记录;___ INSERT INTO R(NO,NAME,CLASS) VALUES(30,"郑和","95031") 或 INSE INTO R(NO,NAME,CLASS) VALUES(30,"郑和","95031")____。

（23. 设有如下关系表 R：

R(NO,NAME,SEX,AGE,CLASS)

主关键字是 NO

其中 NO 为学号(数值型)，NAME 为姓名，SEX 为性别，AGE 为年龄，CLASS 为班号。写出实现下列功能的 SQL 语句。

删除学号为 20 的学生记录;___ DELETE FROM R WHERE NO=20 或 DELE FROM R WHERE NO=20 或 DELE FROM R WHER NO=20 或 DELETE FROM R WHER NO=20 ___。

（23.写出一条 Sql 语句： 取出表 A 中第 31 到第 40 记录（SQLServer, 以自动增长的 ID 作为主键, 注意：ID 可能不是连续的。）

解 1: select top 10 * from A where id not in (select top 30 id from A)

解 2: select top 10 * from A where id > (select max(id) from (select top 30 id from A)as A)

3. Oracle 专集

（1. 解释冷备份和热备份的不同点以及各自的优点

解答：热备份针对归档模式的数据库，在数据库仍旧处于工作状态时进行备份。而冷备份指在数据库关闭后，进行备份，适用于所有模式的数据库。热备份的优点在于当备份时，数据库仍旧可以被使用并且可以将数据库恢复到任意一个时间点。冷备份的优点在于它的备份和恢复操作相当简单，并且由于冷备份的数据库可以工作在非归档模式下,数据库性能会比归档模式稍好。(因为不必将 archive log 写入硬盘)

（2. 你必须利用备份恢复数据库，但是你没有控制文件，该如何解决问题呢?

解答：重建控制文件，用带 backup control file 子句的 recover 命令恢复

数据库。

（3. 如何转换 init.ora 到 spfile?

解答：使用 create spfile from pfile 命令.

（4. 解释 data block , extent 和 segment 的区别(这里建议用英文术语)

解答：data block 是数据库中最小的逻辑存储单元。当数据库的对象需要更多的物理存储空间时，连续的 data block 就组成了 extent . 一个数据库对象

拥有的所有 extents 被称为该对象的 segment.

（5. 给出两个检查表结构的方法

解答：1。DESCRIBE 命令

2. DBMS_METADATA.GET_DDL 包

（6. 怎样查看数据库引擎的报错

解答：alert log.

（7. 比较 truncate 和 delete 命令

解答：两者都可以用来删除表中所有的记录。区别在于：truncate 是 DDL 操作，它移动 HWK，不需要 rollback segment .而 Delete 是 DML 操作, 需要 rollback segment 且花

费较长时间.

（8. 使用索引的理由

解答：快速访问表中的 data block

（9. 给出在 STAR SCHEMA 中的两种表及它们分别含有的数据

解答：Fact tables 和 dimension tables. fact table 包含大量的主要的信息而 dimension tables 存放对 fact table 某些属性描述的信息

（10. FACT Table 上需要建立何种索引?

解答：位图索引 (bitmap index)

（11. 给出两种相关约束?

解答：主键和外键

（12. 如何在不影响子表的前提下，重建一个母表

解答：子表的外键强制实效，重建母表，激活外键

（13. 解释归档和非归档模式之间的不同和它们各自的优缺点

解答：归档模式是指你可以备份所有的数据库 transactions 并恢复到任意一个时间点。非归档模式则相反，不能恢复到任意一个时间点。但是非归档模式可以带来数据库性能上的少许提高.

（14. 如何建立一个备份控制文件?

解答：Alter database backup control file to trace.

（15. 给出数据库正常启动所经历的几种状态 ？

解答：

STARTUP NOMOUNT - 数据库实例启动

STARTUP MOUNT - 数据库装载

STARTUP OPEN - 数据库打开

（16. 哪个 column 可以用来区别 V$视图和 GV$视图?

解答： INST_ID 指明集群环境中具体的 某个 instance 。

（17. 如何生成 explain plan?

解答：运行 utlxplan.sql. 建立 plan 表

针对特定 SQL 语句，使用 explain plan set statement_id = 'tst1' into plan_table

运行 utlxplp.sql 或 utlxpls.sql 察看 explain plan

（18. 如何增加 buffer cache 的命中率?

解答:在数据库较繁忙时,适用 buffer cache advisory 工具,查询 v$db_cache_advice . 如果有必要更改，可以使用 alter system set db_cache_size 命令

（19. ORA-01555 的应对方法?

解答：具体的出错信息是 snapshot too old within rollback seg , 通常可以通过增大 rollback seg 来解决问题。当然也需要察看一下具体造成错误的 SQL 文本

（20. 解释$ORACLE_HOME 和$ORACLE_BASE 的区别?

解答：ORACLE_BASE 是 oracle 的根目录，ORACLE_HOME 是 oracle 产品的目录。

（21. 如何判断数据库的时区?

解答：SELECT DBTIMEZONE FROM DUAL;

（22. 解释 GLOBAL_NAMES 设为 TRUE 的用途

解答：GLOBAL_NAMES 指明联接数据库的方式。如果这个参数设置为 TRUE,在建立数据库链接时就必须用相同的名字连结远程数据库

（23。如何加密 PL/SQL 程序?

解答：WRAP

（24. 解释 FUNCTION,PROCEDURE 和 PACKAGE 区别

解答：function 和 procedure 是 PL/SQL 代码的集合，通常为了完成一个任务。procedure 不需要返回任何值而 function 将返回一个值在另一

方面，Package 是为了完成一个商业功能的一组 function 和 proceudre 的集合

（25. 解释 TABLE Function 的用途

解答：TABLE Function 是通过 PL/SQL 逻辑返回一组纪录，用于普通的表/视图。他们也用于 pipeline 和 ETL 过程。

（26. 举出 3 种可以收集 three advisory statistics

解答：Buffer Cache Advice, Segment Level Statistics, Timed Statistics

（27. Audit trace 存放在哪个 oracle 目录结构中?

解答：unix $ORACLE_HOME/rdbms/audit Windows the event viewer

（28. 解释 materialized views 的作用

解答：Materialized views 用于减少那些汇总，集合和分组的信息的集合数量。它们通常适合于数据仓库和 DSS 系统。

（29. 当用户进程出错，哪个后台进程负责清理它

解答： PMON

（30. 哪个后台进程刷新 materialized views?

解答：The Job Queue Processes.

（31. 如何判断哪个 session 正在连结以及它们等待的资源?

解答：V$SESSION / V$SESSION_WAIT

（32. 描述什么是 redo logs

解答：Redo Logs 是用于存放数据库数据改动状况的物理和逻辑结构。可以用来修复数据库.

（33. 如何进行强制 LOG SWITCH?

解答：ALTER SYSTEM SWITCH LOGFILE;

（34. 举出两个判断 DDL 改动的方法?

解答：你可以使用 Logminer 或 Streams

（35. Coalescing 做了什么?

解答：Coalescing 针对于字典管理的 tablespace 进行碎片整理，将临近的小 extents 合并成单个的大 extent.

（36. TEMPORARY tablespace 和 PERMANENT tablespace 的区别是?

解答：A temporary tablespace 用于临时对象例如排序结构而 permanent tablespaces 用来存储那些'真实'的对象(例如表，回滚段等)

（37. 创建数据库时自动建立的 tablespace 名称?

解答：SYSTEM tablespace.

（38. 创建用户时，需要赋予新用户什么权限才能使它联上数据库。

解答：CONNECT

（39. 如何在 tablespace 里增加数据文件?

解答：ALTER TABLESPACE ADD DATAFILE SIZE

（40. 如何变动数据文件的大小?

解答：ALTER DATABASE DATAFILE RESIZE ;

（41. 哪个 VIEW 用来检查数据文件的大小?

解答： DBA_DATA_FILES

（42. 哪个 VIEW 用来判断 tablespace 的剩余空间

解答：DBA_FREE_SPACE

（43. 如何判断谁往表里增加了一条纪录?

解答：auditing

（44. 如何重构索引?

解答： ALTER INDEX REBUILD;

（45. 解释什么是 Partitioning(分区)以及它的优点。

解答：Partition 将大表和索引分割成更小，易于管理的分区。

（46. 你刚刚编译了一个 PL/SQL Package 但是有错误报道，如何显示出错信息?

解答：SHOW ERRORS

（47. 如何搜集表的各种状态数据?

解答： ANALYZE

The ANALYZE command.

（48. 如何启动 SESSION 级别的 TRACE

解答: DBMS_SESSION.SET_SQL_TRACE

ALTER SESSION SET SQL_TRACE = TRUE;

（49. IMPORT 和 SQL*LOADER 这 2 个工具的不同点

解答：这两个 ORACLE 工具都是用来将数据导入数据库的。

区别是：IMPORT 工具只能处理由另一个 ORACLE 工具 EXPORT 生成的数据。而 SQL*LOADER 可以导入不同的 ASCII 格式的数据源

（50. 用于网络连接的 2 个文件?

解答： TNSNAMES.ORA and SQLNET.ORA

4. SQL 问答题

（1. SELECT * FROM TABLE
和
SELECT * FROM TABLE
WHERE NAME LIKE '%%' AND ADDR LIKE '%%'
AND (1_ADDR LIKE '%%' OR 2_ADDR LIKE '%%'
OR 3_ADDR LIKE '%%' OR 4_ADDR LIKE '%%')
的检索结果为何不同?

（2. 表结构：

1 表名：g_cardapply
字段(字段名/类型/长度)：
g_applyno varchar 8；//申请单号（关键字）
g_applydate bigint 8；//申请日期
g_state varchar 2；//申请状态

2 表名：g_cardapplydetail
字段(字段名/类型/长度)：
g_applyno varchar 8；//申请单号（关键字）
g_name varchar 30；//申请人姓名
g_idcard varchar 18；//申请人身份证号
g_state varchar 2；//申请状态
其中，两个表的关联字段为申请单号。

题目：
1.查询身份证号码为 440401430103082 的申请日期
2.查询同一个身份证号码有两条以上记录的身份证号码及记录个数
3.将身份证号码为 440401430103082 的记录在两个表中的申请状态均改为 07
4.删除 g_cardapplydetail 表中所有姓李的记录

（3．SQLSERVER 服务器中，给定表 table1 中有两个字段 ID、LastUpdateDate，ID 表示更新的事务号， LastUpdateDate 表示更新时的服务器时间，请使用一句 SQL 语句获得最后更新的事务号。(10)
答：SELECT ID
FROM table1
WHERE LastUpdateDate = (SELECT MAX(LastUpdateDate) FROM table1)

5. 问答题

1. 存储过程和函数的区别

存储过程是用户定义的一系列 sql 语句的集合，涉及特定表或其它对象的任务，用户可以调用存储过程，而函数通常是数据库已定义的方法，它接收参数并返回某种类型的值并且不涉及特定用户表。

2. 事务是什么？

事务是作为一个逻辑单元执行的一系列操作，一个逻辑工作单元必须有四个属性，称为 ACID（原子性、一致性、隔离性和持久性）属性，只有这样才能成为一个事务：

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。

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。

隔离性：由并发事务所作的修改必须与任何其它并发事务所作的修改隔离。事务查看数据时数据所处的状态，要么是另一并发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，事务不会查看中间状态的数据。这称为可串行性，因为它能够重新装载起始数据，并且重播一系列事务，以使数据结束时的状态与原始事务执行的状态相同。

持久性 ：事务完成之后，它对于系统的影响是永久性的。该修改即使出现系统故障也将一直保持。

3. 游标的作用？如何知道游标已经到了最后？

游标用于定位结果集的行，通过判断全局变量@@FETCH_STATUS 可以判断是否到了最后，通常此变量不等于 0 表示出错或到了最后。

4. 触发器分为事前触发和事后触发，这两种触发有和区别。语句级触发和行级触发有何区别。

事前触发器运行于触发事件发生之前，而事后触发器运行于触发事件发生之后。通常事前触发器可以获取事件之前和新的字段值。

语句级触发器可以在语句执行前或后执行，而行级触发在触发器所影响的每一行触发一次。

B. 知识点精华

1. SQL

SQL 全称是“结构化查询语言(Structured Query Language)

SQL 语言包含 4 个部分：

※ 数据定义语言(DDL)，例如：CREATE、DROP、ALTER 等语句。

※ 数据操作语言(DML)，例如：INSERT、UPDATE、DELETE 语句。

※ 数据查询语言，例如：SELECT 语句。

※ 数据控制语言，例如：GRANT、REVOKE、COMMIT、ROLLBACK 等语句。

SQL 语言包括三种主要程序设计语言类别的陈述式：数据定义语言(DDL)，数据操作语言(DML)及数据控制语言(DCL)。

DDL

DDL 用于定义和管理物件，例如资料库、资料表以及检视表（ 第 18 章 将会解释何谓检视表）。DDL 陈述式通常包括每个物件的 CREATE、ALTER 以及 DROP 命令。举例来说，CREATE TABLE、ALTER TABLE 以及 DROP TABLE 这些陈述式便可以用来建立新资料表、修改其属性（如新增或删除资料行）、删除资料表等，下面我们会一一介绍。

CREATE TABLE 陈述式

使用 DDL 在 MyDB 资料库建立一个名为 Customer_Data 的范例资料表，本章后面的例子我们会使用到这个资料表。如前所述，CREATE TABLE 陈述式可以用来建立资料表。这个范例资料表被定义成四个资料行，如下所示：

```
Use MyDB
CREATE TABLE Customer_Data
(customer_id smallint,
first_name char(20),
last_name char(20),
phone char(10))
GO
```

这个陈述式能产生 Customer_Data 资料表，这个资料表会一直是空的直到资料被填入资料表内。

ALTER TABLE 陈述式

ALTER TABLE 陈述式用来变更资料表的定义与属性。在下面的例子中，我们利用 ALTER TABLE 在已经存在的 Customer_Data 资料表中新增 middle_initial 资料行。

```
ALTER TABLE Customer_Data
ADD middle_initial char(1)
GO
```

现在资料表的定义包括了五个资料行，而不是之前的四个资料行。关於使用 ALTER TABLE 的更多细节，请参阅 第 15 章 。

DROP TABLE 陈述式

DROP TABLE 陈述式用来删除资料表定义以及所有的资料、索引、触发程序、条件约束以及资料表的权限。要删除我们的 Customer_Data 资料表，可利用下列命令：

DROP TABLE Customer_Data

GO

DML

DML 利用 INSERT、SELECT、UPDATE 及 DELETE 等陈述式来操作资料库物件所包含的资料。

INSERT 陈述式

INSERT 陈述式用来在资料表或检视表中插入一列资料。例如，如果要在 Customer_Data 资料表中新增一个客户，可使用类似以下的 INSERT 陈述式：

```
INSERT INTO Customer_Data
(customer_id, first_name, last_name, phone)
VALUES (777, "Frankie", "Stein", "4895873900")
```

请注意 SQL 陈述式中第二行的资料行名称清单，清单上资料行名称的次序决定了资料数值将被放在哪个资料行。举例来说，第一个资料数值将被放在清单列出的第一个资料行 customer_id、第二个资料数值放在第二个资料行，依此类推。由于我们在建立资料表时，定义资料资料行填入数值的次序与现在相同，因此我们不必特意指定栏位名称。我们可以用以下的 INSERT 陈述式代替：

```
INSERT INTO Customer_Data
VALUES (777, "Frankie", "Stein", "4895873900")
```

注意

如果使用这种形式的 INSERT 陈述式，但被插入的数值次序上与建立资料表时不同，数值将被放入错误的资料行。如果资料的型别与定义不符，则会收到一个错误讯息。

UPDATE 陈述式

UPDATE 陈述式用来更新或改变一列或多列中的值。例如，一位名称为 Frankie Stein 的客户想要在记录中改变他的姓氏为 Franklin，可使用以下 UPDATE 陈述式：

```
UPDATE Customer_Data
SET first_name = "Franklin"
WHERE last_name = "Stein" and customer_id= 777
```

我们在 WHERE 子句中加入 customer_id 的项目来确定其他名称为 Stein 的客户不会被影响－只有 customer_id 为 777 的客户，姓氏会有所改变。

说明

当您使用 UPDATE 陈述式时，要确定在 WHERE 子句提供充分的筛选条件，如此才不会不经意地改变了一些不该改变的资料。

DELETE 陈述式

DELETE 陈述式用来删除资料表中一列或多列的资料，您也可以删除资料表中的所有资料列。要从 Customer_Data 资料表中删除所有的列，您可以利用下列陈述式：

```
DELETE FROM Customer_Data
```

或

```
DELETE Customer_Data
```

资料表名称前的 FROM 关键字在 DELETE 陈述式中是选择性的。除此之外，这两个陈述式完全相同。

要从 Customer_Data 资料表中删除 customer_id 资料行的值小於 100 的列，可利用下列陈述式：

```
DELETE FROM Customer_Data
WHERE customer_id < 100
```

现在我们已经快速浏览了 SQL 提供的 DDL 与 DML 陈述式，接著，下面将介绍 T-SQL。
DCL
DCL 是用来管理数据库的语言。包含管理权限及数据更改。
SELECT 陈述式

SELECT 陈述式用来检索资料表中的资料，而哪些资料被检索由列出的资料行与陈述式中的 WHERE 子句决定。例如，要从之前建立的 Customer_Data 资料表中检索 customer_id 以及 first_name 资料行的资料，并且只想取出每列中 first_name 资料行值为 Frankie 的资料，那麼可以利用以下的 SELECT 陈述式：

SELECT customer_id, first_name FROM Customer_Data

WHERE first_name = "Frankie"

如果有一列符合 SELECT 陈述式中的标准，则结果将显示如下：

customer_id first_name

777 Frankie

SQL 中的五种数据类型

简要描述一下 SQL 中的五种数据类型:字符型，文本型，数值型，逻辑型和日期型

字符型

VARCHAR VS CHAR

VARCHAR 型和 CHAR 型数据的这个差别是细微的，但是非常重要。他们都是用来储存字符串长度小于 255 的字符。

假如你向一个长度为四十个字符的 VARCHAR 型字段中输入数据 BIll GAtES。当你以后从这个字段中取出此数据时，你取出的数据其长度为十个字符——字符串 Bill Gates 的长度。 现在假如你把字符串输入一个长度为四十个字符的 CHAR 型字段中，那么当你取出数据时，所取出的数据长度将是四十个字符。字符串的后面会被附加多余的空格。

当你建立自己的站点时，你会发现使用 VARCHAR 型字段要比 CHAR 型字段方便的多。使用 VARCHAR 型字段时，你不需要为剪掉你数据中多余的空格而操心。

VARCHAR 型字段的另一个突出的好处是它可以比 CHAR 型字段占用更少的内存和硬盘空间。当你的数据库很大时，这种内存和磁盘空间的节省会变得非常重要

文本型

TEXT

使用文本型数据,你可以存放超过二十亿个字符的字符串。当你需要存储大串的字符时，应该使用文本型数据。

注意文本型数据没有长度，而上一节中所讲的字符型数据是有长度的。一个文本型字段中的数据通常要么为空，要么很大。

当你从 HTML fORM 的多行文本编辑框(TEXTAREA)中收集数据时，你应该把收集的信息存储于文本型字段中。但是，无论何时，只要你能避免使用文本型字段，你就应该不适用它。文本型字段既大且慢，滥用文本型字段会使服务器速度变慢。文本型字段还会吃掉大量的磁盘空间。

一旦你向文本型字段中输入了任何数据(甚至是空值)，就会有 2K 的空间被自动分配给该数据。除非删除该记录，否则你无法收回这部分存储空间。

数值型

SQL 支持许多种不同的数值型数据。你可以存储整数 INT 、小数 NUMERIC、和钱数 MONEY。

INT VS SMALLINT VS TINYINT

他们的区别只是字符长度:

INT 型数据的表数范围是从-2，147，483，647 到 2，147，483，647 的整数

SMALLINT 型数据可以存储从-32768 到 32768 的整数

TINYINT 型的字段只能存储从 0 到 255 的整数,不能用来储存负数

通常，为了节省空间，应该尽可能的使用最小的整型数据。一个 TINYINT 型数据只占用一个字节;一个 INT 型数据占用四个字节。这看起来似乎差别不大，但是在比较大的表中，字节数的增长是很快的。另一方面，一旦你已经创建了一个字段，要修改它是很困难的。因此，为安全起见，你应该预测以下，一个字段所需要存储的数值最大有可能是多大，然后选择适当的数据类型。

NUMERIC

为了能对字段所存放的数据有更多的控制,你可以使用 NUMERIC 型数据来同时表示一个数的整数部分和小数部分。NUMERIC 型数据使你能表示非常大的数——比 INT 型数据要大得多。一个 NUMERIC 型字段可以存储从-1038 到 1038 范围内的数。NUMERIC 型数据还使你能表示有小数部分的数。例如，你可以在 NUMERIC 型字段中存储小数 3.14。

当定义一个 NUMERIC 型字段时，你需要同时指定整数部分的大小和小数部分的大小。如:MUNERIC(23,0)

一个 NUMERIC 型数据的整数部分最大只能有 28 位，小数部分的位数必须小于或等于整数部分的位数，小数部分可以是零。

MONEY VS SMALLMONEY

你可以使用 INT 型或 NUMERIC 型数据来存储钱数。但是，专门有另外两种数据类型用于此目的。如果你希望你的网点能挣很多钱，你可以使用 MONEY 型数据。如果你的野心不大，你可以使用 SMALLMONEY 型数据。MONEY 型数据可以存储从-922，337，203，685，477.5808 到 922，337，203，685，477.5807 的钱数。如果你需要存储比这还大的金额，你可以使用 NUMERIC 型数据。

SMALLMONEY 型数据只能存储从-214，748.3648 到 214，748.3647 的钱数。同样，如果可以的话，你应该用 SMALLMONEY 型来代替 MONEY 型数据，以节省空间。

逻辑型

BIT

如果你使用复选框(CHECKBOX)从网页中搜集信息，你可以把此信息存储在 BIT 型字段中。BIT 型字段只能取两个值:0 或 1。

当心，在你创建好一个表之后，你不能向表中添加 BIT 型字段。如果你打算在一个表中包含 BIT 型字段，你必须在创建表时完成。

日期型

DATETIME VS SMALLDATETIME

一个 DATETIME 型的字段可以存储的日期范围是从 1753 年 1 月 1 日第一毫秒到 9999 年 12 月 31 日最后一毫秒。

如果你不需要覆盖这么大范围的日期和时间，你可以使用 SMALLDATETIME 型数据。它与 DATETIME 型数据同样使用，只不过它能表示的日期和时间范围比 DATETIME 型数据小，而且不如 DATETIME 型数据精确。一个 SMALLDATETIME 型的字段能够存储从 1900 年 1 月 1 日到 2079 年 6 月 6 日的日期，它只能精确到秒。

DATETIME 型字段在你输入日期和时间之前并不包含实际的数据，认识这一点是重要的。

1.SQL SERVER 的数据类型

数据类弄是数据的一种属性，表示数据所表示信息的类型。任何一种计算机语言都定义了自己的数据类型。当然，不同的程序语言都具有不同的特点，所定义的数据类型的各类和名称都或多或少有些不同。SQLServer 提供了 25 种数据类型：

- Binary [(n)]
- Varbinary [(n)]
- Char [(n)]
- Varchar[(n)]
- Nchar[(n)]
- Nvarchar[(n)]
- Datetime
- Smalldatetime
- Decimal[(p[,s])]
- Numeric[(p[,s])]
- Float[(n)]
- Real
- Int
- Smallint
- Tinyint
- Money
- Smallmoney
- Bit
- Cursor
- Sysname
- Timestamp
- Uniqueidentifier
- Text
- Image
- Ntext

(1)二进制数据类型

二进制数据包括 Binary、Varbinary 和 Image

Binary 数据类型既可以是固定长度的(Binary),也可以是变长度的。

Binary[(n)] 是 n 位固定的二进制数据。其中，n 的取值范围是从 1 到 8000。其存储窖的大小是 n + 4 个字节。

Varbinary[(n)] 是 n 位变长度的二进制数据。其中，n 的取值范围是从 1 到 8000。其存储窖的大小是 n + 4 个字节，不是n 个字节。

在 Image 数据类型中存储的数据是以位字符串存储的，不是由 SQL Server 解释的，必须由应用程序来解释。例如，应用程序可以使用 BMP、TIEF、GIF 和 JPEG 格式把数据存储在 Image 数据类型中。

(2)字符数据类型

字符数据的类型包括 Char，Varchar 和 Text

字符数据是由任何字母、符号和数字任意组合而成的数据。

Varchar 是变长字符数据，其长度不超过 8KB。Char 是定长字符数据，其长度最多为 8KB。超过 8KB 的 ASCII 数据可以使用 Text 数据类型存储。例如，因为 Html 文档全部都是 ASCII 字符，并且在一般情况下长度超过 8KB，所以这些文档可以 Text 数据类型存储在 SQL Server 中。

(3)Unicode 数据类型

Unicode 数据类型包括 Nchar,Nvarchar 和 Ntext

在 Microsoft SQL Server 中，传统的非 Unicode 数据类型允许使用由特定字符集定义的字符。在 SQL Server 安装过程中，允许选择一种字符集。使用 Unicode 数据类型，列中可以存储任何由Unicode 标准定义的字符。在 Unicode 标准中，包括了以各种字符集定义的全部字符。使用Unicode 数据类型，所战胜的窖是使用非 Unicode 数据类型所占用的窖大小的两倍。

在 SQL Server 中，Unicode 数据以 Nchar、Nvarchar 和 Ntext 数据类型存储。使用这种字符类型存储的列可以存储多个字符集中的字符。当列的长度变化时，应该使用 Nvarchar 字符类型，这时最多可以存储 4000 个字符。当列的长度固定不变时，应该使用 Nchar 字符类型，同样，这时最多可以存储 4000 个字符。当使用 Ntext 数据类型时，该列可以存储多于 4000 个字符。

(4)日期和时间数据类型

日期和时间数据类型包括 Datetime 和 Smalldatetime 两种类型

日期和时间数据类型由有效的日期和时间组成。例如，有效的日期和时间数据包括“4/01/98 12:15:00:00:00 PM”和“1:28:29:15:01AM 8/17/98”。前一个数据类型是日期在前，时间在后一个数据类型是雯时间在前，日期在后。在 Microsoft SQL Server 中，日期和时间数据类型包括 Datetime 和 Smalldatetime 两种类型时，所存储的日期范围是从 1753 年 1 月 1 日开始，到 9999 年 12 月 31 日结束(每一个值要求 8 个存储字节)。使用 Smalldatetime 数据类型时，所存储的日期范围是 1900 年 1 月 1日 开始，到 2079 年 12 月 31 日结束(每一个值要求 4 个存储字节)。

日期的格式可以设定。设置日期格式的命令如下：

Set DateFormat {format | @format _var|

其中，format | @format_var 是日期的顺序。有效的参数包括 MDY、DMY、YMD、YDM、MYD 和 DYM。在默认情况下，日期格式为 MDY。

例如，当执行 Set DateFormat YMD 之后，日期的格式为年 月 日 形式；当执行 Set DateFormat DMY 之后，日期的格式为日 月有年 形式

（5）数字数据类型

数字数据只包含数字。数字数据类型包括正数和负数、小数（浮点数）和整数

整数由正整数和负整数组成，例如 39、25、0-2 和 33967。在 Micrsoft SQL Server 中，整数存储的数据类型是 Int，Smallint 和 Tinyint。Int 数据类型存储数据的范围大于 Smallint 数据类型存储数据的范围，而 Smallint 据类型存储数据的范围大于 Tinyint 数据类型存储数据的范围。使用 Int 数据狗昔存储数据的范围是从 -2 147 483 648 到 2 147 483 647（每一个值要求 4 个字节存储空间)。使用 Smallint 数据类型时，存储数据的范围从 -32 768 到 32 767（每一个值要求 2 个字节存储空间)。使用 Tinyint 数据类型时，存储数据的范围是从 0 到 255（每一个值要求 1 个字节存储空间)。

精确小娄数据在 SQL Server 中的数据类型是 Decimal 和 Numeric。这种数据所占的存储空间根据该数据的位数后的位数来确定。

在 SQL Server 中，近似小数数据的数据类型是 Float 和 Real。例如，三分之一这个分数记作。3333333，当使用近似数据类型时能准确表示。因此，从系统中检索到的数据可能与存储在该列中数据不完全一样。

（6）货币数据表示正的或者负的货币数量 。

在 Microsoft SQL Server 中，货币数据的数据类型是 Money 和 Smallmoney

Money 数据类型要求 8 个存储字节，Smallmoney 数据类型要求 4 个存储字节。

（7）特殊数据类型

特殊数据类型包括前面没有提过的数据类型。特殊的数据类型有 3 种，即 Timestamp、

Bit 和 Uniqueidentifier。
Timestamp 用于表示 SQL Server 活动的先后顺序，以二进投影的格式表示。Timestamp 数据与插入数据或者日期和时间没有关系。
Bit 由 1 或者 0 组成。当表示真或者假、ON 或者 OFF 时，使用 Bit 数据类型。例如，询问是否是每一次访问的客户机请求可以存储在这种数据类型的列中。
Uniqueidentifier 由 16 字节的十六进制数字组成，表示一个全局唯一的。当表的记录行要求唯一时，GUID 是非常有用。例如，在客户标识号列使用这种数据类型可以区别不同的客户。
2.用户定义的数据类型
用户定义的数据类型基于在 Microsoft SQL Server 中提供的数据类型。当几个表中必须存储同一种数据类型时，并且为保证这些列有相同的数据类型、长度和可空性时，可以使用用户定义的数据类型。例如，可定义一种称为
postal_code 的数据类型，它基于 Char 数据类型。
当创建用户定义的数据类型时，必须提供三个数：数据类型的名称、所基于的系统数据类型和数据类型的可空性。
（1）创建用户定义的数据类型
创建用户定义的数据类型可以使用 Transact-SQL 语句。系统存储过程 sp_addtype 可以来创建用户定义的数据类型。其语法形式如下：
sp_addtype {type},[,system_data_bype][,'null_type']
其中，type 是用户定义的数据类型的名称。system_data_type 是系统提供的数据类型，例如 Decimal、Int、Char 等等。 null_type 表示该数据类型是如何处理空值的，必须使用单引号引起来，例如'NULL'、'NOT NULL'或者'NONULL'。
例子：

```
Use cust
Exec sp_addtype ssn,'Varchar(11)',"Not Null'
```

创建一个用户定义的数据类型 ssn，其基于的系统数据类型是变长为 11 的字符，不允许空。
例子：

```
Use cust
Exec sp_addtype birthday,datetime,'Null'
```

创建一个用户定义的数据类型 birthday，其基于的系统数据类型是 DateTime，允许空。
例子：

```
Use master
Exec sp_addtype telephone,'varchar(24),'Not Null'
Eexc sp_addtype fax,'varchar(24)','Null'
```

创建两个数据类型，即 telephone 和 fax
（2）删除用户定义的数据类型
当用户定义的数据类型不需要时，可删除。删除用户定义的数据类型的命令是 sp_droptype {'type'}。
例子：

```
Use master
Exec sp_droptype 'ssn'
```

注意：当表中的列还正在使用用户定义的数据类型时，或者在其上面还绑定有默认或者规则时，这种用户定义的数据类型不能删除。
高级特性

本文介绍 LINQ 的高级特性，其包括大家都关心的动态查询的用法，另外简单提下 ID 标识这个知识。

动态查询

有这样一个场景：应用程序可能会提供一个用户界面，用户可以使用该用户界面指定一个或多个谓词来筛选数据。这种情况在编译时不知道查询的细节，动态查询将十分有用。

在 LINQ 中，Lambda 表达式是许多标准查询运算符的基础，编译器创建 lambda 表达式以捕获基础查询方法（例如 Where、Select、Order By、Take While 以及其他方法）中定义的计算。表达式目录树用于针对数据源的结构化查询，这些数据源实现 IQueryable<T>。例如，LINQ to SQL 提供程序实现 IQueryable<T>接口，用于查询关系数据存储。C#和 Visual Basic 编译器会针对此类数据源的查询编译为代码，该代码在运行时将生成一个表达式目录树。然后，查询提供程序可以遍历表达式目录树数据结构，并将其转换为适合于数据源的查询语言。表达式目录树在 LINQ 中用于表示分配给类型为 Expression<TDelegate>的变量的 Lambda 表达式。还可用于创建动态 LINQ 查询。

System.Linq.Expressions 命名空间提供用于手动生成表达式目录树的 API。Expression 类包含创建特定类型的表达式目录树节点的静态工厂方法，例如，ParameterExpression（表示一个已命名的参数表达式）或 MethodCallExpression（表示一个方法调用）。编译器生成的表达式目录树的根始终在类型 Expression<TDelegate>的节点中，其中 TDelegate 是包含至多五个输入参数的任何 TDelegate 委托；也就是说，其根节点是表示一个 lambda 表达式。下面几个例子描述如何使用表达式目录树来创建动态 LINQ 查询。

1.Select

下面例子说明如何使用表达式树依据 IQueryable 数据源构造一个动态查询，查询出每个顾客的 ContactName，并用 GetCommand 方法获取其生成 SQL 语句。

```
//依据 IQueryable 数据源构造一个查询
IQueryable<Customer> custs = db.Customers;
//组建一个表达式树来创建一个参数
ParameterExpression param =
    Expression.Parameter(typeof(Customer), "c");
//组建表达式树:c.ContactName
Expression selector = Expression.Property(param,
    typeof(Customer).GetProperty("ContactName"));
Expression pred = Expression.Lambda(selector, param);
//组建表达式树:Select(c=>c.ContactName)
Expression expr = Expression.Call(typeof(Queryable), "Select",
    new Type[] { typeof(Customer), typeof(string) },
    Expression.Constant(custs), pred);
//使用表达式树来生成动态查询
IQueryable<string> query = db.Customers.AsQueryable()
    .Provider.CreateQuery<string>(expr);
//使用 GetCommand 方法获取 SQL 语句
System.Data.Common.DbCommand cmd = db.GetCommand(query);
Console.WriteLine(cmd.CommandText);
```

生成的 SQL 语句为：

```
SELECT [t0].[ContactName] FROM [dbo].[Customers] AS [t0]
```

2.Where

下面一个例子是“搭建”Where 用法来动态查询城市在伦敦的顾客。

```
IQueryable<Customer> custs = db.Customers;
//创建一个参数 c
ParameterExpression param = Expression.Parameter(typeof(Customer), "c");
//c.City=="London"
Expression left = Expression.Property(param,
typeof(Customer).GetProperty("City"));
Expression right = Expression.Constant("London");
Expression filter = Expression.Equal(left, right);

Expression pred = Expression.Lambda(filter, param);
//Where(c=>c.City=="London")
Expression expr = Expression.Call(typeof(Queryable), "Where", new Type[]
{ typeof(Customer) }, Expression.Constant(custs), pred);
//生成动态查询
IQueryable<Customer> query =
db.Customers.AsQueryable() .Provider.CreateQuery<Customer>(expr);
```

生成的 SQL 语句为：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0] WHERE [t0].[City] = @p0
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
```

3.OrderBy

本例既实现排序功能又实现了过滤功能。

```
IQueryable<Customer> custs = db.Customers;
//创建一个参数 c
ParameterExpression param =
   Expression.Parameter(typeof(Customer), "c");
//c.City=="London"
Expression left = Expression.Property(param,
    typeof(Customer).GetProperty("City"));
Expression right = Expression.Constant("London");
Expression filter = Expression.Equal(left, right);
Expression pred = Expression.Lambda(filter, param);
//Where(c=>c.City=="London")
MethodCallExpression whereCallExpression = Expression.Call(
    typeof(Queryable), "Where",
    new Type[] { typeof(Customer) },
    Expression.Constant(custs), pred);
//OrderBy(ContactName => ContactName)
MethodCallExpression orderByCallExpression = Expression.Call(
    typeof(Queryable), "OrderBy",
    new Type[] { typeof(Customer), typeof(string) },
```

```
    whereCallExpression,
    Expression.Lambda(Expression.Property
    (param, "ContactName"), param));
//生成动态查询
IQueryable<Customer> query = db.Customers.AsQueryable()
    .Provider.CreateQuery<Customer>(orderByCallExpression);
```

动态生成动态查询

生成的 SQL 语句为：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0] WHERE [t0].[City] = @p0
ORDER BY [t0].[ContactName]
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
```

4.Union

下面的例子使用表达式树动态查询顾客和雇员同在的城市。

```
//e.City
IQueryable<Customer> custs = db.Customers;
ParameterExpression param1 =
Expression.Parameter(typeof(Customer), "e");
Expression left1 = Expression.Property(param1,
    typeof(Customer).GetProperty("City"));
Expression pred1 = Expression.Lambda(left1, param1);
//c.City
IQueryable<Employee> employees = db.Employees;
ParameterExpression param2 =
Expression.Parameter(typeof(Employee), "c");
Expression left2 = Expression.Property(param2,
    typeof(Employee).GetProperty("City"));
Expression pred2 = Expression.Lambda(left2, param2);
//Select(e=>e.City)
Expression expr1 = Expression.Call(typeof(Queryable), "Select",
    new Type[] { typeof(Customer), typeof(string) },
    Expression.Constant(custs), pred1);
//Select(c=>c.City)
Expression expr2 = Expression.Call(typeof(Queryable), "Select",
    new Type[] { typeof(Employee), typeof(string) },
    Expression.Constant(employees), pred2);
//生成动态查询
IQueryable<string> q1 = db.Customers.AsQueryable()
    .Provider.CreateQuery<string>(expr1);
IQueryable<string> q2 = db.Employees.AsQueryable()
    .Provider.CreateQuery<string>(expr2);
//并集
```

```
var q3 = q1.Union(q2);
```
生成的 SQL 语句为：
```
SELECT [t2].[City]
FROM (
    SELECT [t0].[City] FROM [dbo].[Customers] AS [t0]
    UNION
    SELECT [t1].[City] FROM [dbo].[Employees] AS [t1]
    ) AS [t2]
```
ID 标识

在前面这一点没有说到，在这里作为高级特性单独说下 ID 标识。

这个例子说明我们存储一条新的记录时候，ContactID 作为主键标识，系统自动分配，标识种子为 1，所以每次自动加一。
```
//ContactID 是主键 ID，插入一条数据，系统自动分配 ID
Contact con = new Contact()
{
    CompanyName = "New Era",
    Phone = "(123)-456-7890"
};
db.Contacts.InsertOnSubmit(con);
db.SubmitChanges();
```

第六部分 操作系统

A. 笔试面试题集

1. 在下列系统中，( )是实时系统。
A.计算机激光照排系统 B.航空定票系统 C．办公自动化系统 D.计算机辅助设计系统
答案：B

2. 操作系统是一种( )。
A.应用软件 B．系统软件 C．通用软件 D．工具软件
答案：B

3. 引入多道程序的目的在于( )。
A.充分利用 CPU，减少 CPU 等待时间 B．提高实时响应速度
C.有利于代码共享，减少主、辅存信息交换量 D．充分利用存储器
答案：A

4. 已经获得除( )以外的所有运行所需资源的进程处于就绪状态
A.存储器 B．打印机 C．CPU D．磁盘空间
答案：C

5. 进程调度的关键问题：一是选择合理的( )，二是恰当地进行代码转换
A．时间片间隔 B.调度算法 C．CPU 速度 D．内存空间
答案：B

6. 采用轮转法调度是为了( )：
A.多个终端都能得到系统的及时响应 B．先来先服务
C．优先级较高的进程得到及时调度 D．需 CPU 最短的进程先做
答案：A

7. 在一段时间内只允许一个进程访问的资源，称为( )
A.共享资源 B．临界区 C．临界资源 D．共享区
答案：C

8. 并发性是指若干事件在( )发生
A．同一时刻 B．同一时间间隔内 C．不同时刻 D．不同时间间隔内
答案：B

9. 在单一处理器上，将执行时间有重叠的几个程序称为( )
A．顺序程序 B．多道程序 C.并发程序 D．并行程序
答案：C

10. 程序运行时，独占系统资源，只有程序本身能改变系统资源状态，这是指( )
A．程序顺序执行的再现性 B．程序顺序执行的封闭性
C．并发程序失去封闭性 D．并发程序失去再现性
答案：B

11. 引人多道程序技术以后，处理器的利用率( )
A.降低了 B．有所改善 C．大大提高 D．没有变化，只是程序的执行方便了.
答案：C

12. 在单一处理器上执行程序，多道程序的执行是在( )进行的。
A．同一时刻 B．同一时间间隔内 C．某一固定时刻 D．某一固定时间间隔内
答案：B

13. 为了使多个进程能有效地同时处理输入和输出，最好使用( )
A．缓冲区 B．闭缓冲区环 C．多缓冲区 D．双缓冲区
答案：A

14. 在进程通信中，( )常通过变量、数组形式来实现。
A．高级通信 B．消息通信 C．低级通信 D．管道通信
答案：C

15. 管道通信是以( )进行写入和读出。
A．消息为单位 B．自然字符流 C．文件 D．报文
答案：B

16. 系统出现死锁的原因是( )
A.计算机系统发生了重大故障 B．有多个封锁的进程同时存在
C．若干进程因竞争资源而无休止的等待着，它方释放已占有的资源
D．资源数大大少于进程数，或进程同时申请的资源数大大超过资源总数
答案：C

17. 解决死锁的途径是( )
A．立即关机排除故障 B．立即关机再重新开机
C.不要共享资源，增加独占资源 D．设计预防死锁，运行检测并恢复
答案：D

18. 进程 P1 使用资源情况：申请资源 S1.. • 申请资源 S2，…释放资源 S1；进程凹使用资源情况：申请资源 S2，…申请资源 S1，…释放资源 S2，系统并发执行进程 P1，P2，系统将( )
A.必定产生死锁 B．可能产生死锁 C．不会产生死锁 D．无法确定是否会产生死锁
答案：B

19. 现代操作系统的两个基本特征是( )和资源共享。
A.多道程序设计 B．中断处理 C．程序的并发执行 D．实现分时与实时处理

答案：C

20. 为了描述进程的动态变化过程，采用了一个与进程相联系的( )系统，根据它而感知进程的存在。
A.进程状态字 B．进程优先数 C．进程控制块 D．进程起始地址
答案：C

21. 上题中所指是进程的唯一( )。
A.关联 B．实体 C．状态 D．特征
答案：B

22. 操作系统中采用缓冲技术的目的是为了增强系统( )的能力。
A.串行操作 B．重执操作 C.控制操作 D．并行操作
答案：D

23. 操作系统中采用缓冲技术，能够减少对 CPU 的( )的次数，从而提高资源的利用率。
A.中断 B.访问 C．控制 D．依赖
答案：A

24. 已经获得除 CPU 以外的所有所需资源的进程处于( )状态。
A.运行状态 B．就绪状态 C．自由状态 D．阻塞状态
答案：B

25. 顺序程序和并发程序的执行相比 ( )
A.基本相同 B．有点不同
C.并发现程序执行总体上执行时间快 D.顺序程序执行总体上执行时间快
答案：C

26. 进程是( )
A.与程序等效的概念 B．行进中的程序 C.一个系统软件 D．存放在内存中的程序
答案：B

27. 进程具有并发性和( )两大重要属性。
A．动态性 B．静态性 C 易用性 D．封闭性
答案：A

28. 操作系统在控制和管理进程过程中，涉及到( )这一重要数据结构，这是进程存在的唯一标志。
A．FCB B．FIFO C. FDT D．PCB
答案：D

29. 磁盘的读写单位是( )
A.块 B．扇区 C．簇 D．字节
答案：B

30. 在单处理机系统中，处于运行状态的进程( )
A.只有一个 B．可以有多个 C．不能被挂起 D．必须在执行完成后才能被撤下
答案：A

31. 如果某一进程获得除 CPU 以外的所有所需运行资源，经调度，分配 CPU 给它，该进程将进入( )
A.就绪状态 B．运行状态 C．阻塞状态 D．活动状态
答案：B

32. 如果某一进程在运行时，因某种原因暂停，此时将脱离运行状态，而进入( )
A.自由状态 B．停止状态 C．阻塞状态 D．静止状态
答案：C

33. 在操作系统中同时存在多个进程，它们( )
A.不能共享系统资源 B．不能调用同一段程序代码
C.可以共享允许共享的系统资源 D．可以共享所有的系统资源
答案：C

34. 操作系统中有一组常称为特殊系统调用．它们不能被系统中断，在操作系统中称为( )
A.初始化程序 B．原语 C．子程序 D.控制模块
答案：B

35. 如果某一进程处于就绪状态要将其投入运行，应使用( )
A.挂起原语 B．创建原浯 C.调度原语 D．终止原语
答案：C

36. 当一进程运行时，系统可基于某种原则，强行将其掇下，把处理器分配给其他进程，这种调 度方式是( )
A.非剥夺方式 B．剥夺方式 C．中断方式 D．查询方式
答案：C

37. 为了照顾短作业用户，进程调度采用( )
A.先进先出调度算法 B．短执行优先调度 C．优先级调度 D．轮转法
答案：B

38. 为了对紧急进程或重要进程进行调度，调度算法采用( )
A.先进先出调度算法 B．短执行优先调度 C．优先级调度 D．轮转法
答案：B

39. 如果某些进程优先级别相同，应采用( )算法较为适应。
A．FIFO B．SCBF C.FDF D．轮转法
答案：A

40. 如果要照顾所有进程，让它们都有执行的机会，最好采用( )算法。
A．SCBF B．FIFO C．轮转法 D．FPF
答案：C

41. 在下列情况( )，要进行进程调度。
A.某一进程正访问一临界资源 B．某一进程运行时因缺乏资源进入阻塞状态
C．某一进程处于运行状态，而另一进程处于自由状态
D．某一进程正在访问打印机，而另一进程处于就绪状态
答案：B

42. 操作系统中，( )负责对进程进行调度。
A.处理机管理 B．作业管理 C，高级高度管理 D．存储和设备管理
答案：A

43. 进程间的基本关系为( )
A.相互独立与互相制约 B．同步与互斥 C.并行执行与资源共享 D．信息传递与信息缓冲
答案：B

44. 进程间的同步与互斥，分别表示了各进程间的( )
A.相互独立与互相制约 B．协调与竞争 C．不同状态 D．动态性与独立性
答案：B
45. 操作系统对临界区调用的原则之一是( )
A.当无进程处于临界区时 B．当有进程处于临界区时
C．当进程处于就绪状态时 D．当进程开始创建时
答案：A

46. 两个进程合作完成一个任务，在并发执行中，一个进程要等待其合作伙伴发来信息，或者建立某个条件后再向前执行，这种关系是进程间的( )关系。
A.同步 B．互斥 C．竞争 D．合作
答案：A

47. ( )是一种能由 P 和 V 操作所改变的整型变量。
A.控制变量 B．锁 C．整型信号量 D．记录型信号量
答案：C

48. 在一单用户操作系统中，当用户编辑好一个程序要存放到磁盘上去的时候，他使用操作系统提供的( )这一接口。
A．键盘命令 B．作业控制命令 C.鼠标操作 D．原语
答案：A

49. ( )存储管理支持多道程序设计，算法简单，但存储碎片多。
A.段式 B．页式 C.固定分区 D．段页式
答案：C

50. 虚拟存储技术是( )。
A．补充内存物理空间的技术 B．补充相对地址空间的技术
C．扩充外存空间技术 D. 扩充输人输出缓冲区的技术
答案：B

51. 虚拟内存的容量只受( )的限制。
A．物理内存的大小 B．磁盘空间的大小 C．数据存放的实际地址 D．计算机地址位数
答案：D

52. 动态页式管理中的( )是：当内存中没有空闲帧时，如何将已占据的帧释放。
A.调入策略 B．地址变换 C.替换策略 D．调度算法
答案：C

53. 分区管理要求对每一个作业都分配( )的内存单元。
A.地址连续 B．若干地址不连续 C.若干连续的帧 D．若干不连续的帧
答案：B

54. 缓冲技术用于( )。
A.提高主机和设备交换信息的速度 B．提供主、辅存接口
C.提高设备利用率 D.扩充相对地址空间
答案：A

55. 段页式管理每取一数据，要访问( )次内存。
A．1 B．2 C．3 D．4
答案：C

56. 分段管理提供( )维的地址结构。
A．1 B．2 C．3 D．4
答案：B

57. 系统抖动是指( )
A. 使用机器时，千万屏幕闪烁的现象
B．刚被调出的帧又立刻被调入所形成的频繁调入调出的现象
C．系统盘不净，千万系统不稳定的现象
D．由于内存分配不当，偶然造成内存不够的现象
答案：B

58. 在( )中，不可能产生系统抖动现象。
A．静态分区管理 B．请求页式管理 C．段式分区管理 D．机器中不存在病毒时
答案：A

59. 当内存碎片容量大于某一作业所申请的内存容量时，( )
A.可以为这一作业分配内存 B．不可以为这一作业分配内存
C．拼接后，可以为这一作业分配内存 D.一定能够为这一作业分配内存

答案：D

60. 在分段管理中( )
A．以段为单位分配，每段是一个连续存储区 B．段与段之间必定不连续
C．段与段之间必定连续 D．每段是等长的
答案：A

61. 进程与线程有什么区别？

（1）每个进程实体中包含了程序段、数据段这两个部分，因此说进程和程序是紧密相关的。但从结构上看，进程实体中除了程序段和数据段外，还必须包含一个数据结构，即进程控制块 PCB。

（2）进程是程序的一次执行过程，因此是动态的；动态性还表现在进程由创建产生、由调度而执行、由撤销而消亡，即它具有一定的生命周期。而程序则只是一组指令的有序集合，并可永久地存放在某种介质上，其本身不具有动态的含义，因此是静态的。

（3）多个进程实体可同时存放在内存中并发执行，其实这正是引入进程的目的。而程序的并发执行具有不可再现性，因此程序不能正确地并发执行。

（4）进程是一个能够独立运行、独立分配资源和独立接受调度的基本单位。而因程序不具有 PCB，所以它是不可能在多道程序环境下独立运行的。

（5）进程和程序不一一对应。同一个程序的多次运行，将形成多个不同的进程；同一个程序的一次执行也可以产生多个进程；而一个进程也可以执行多个程序。

62. AND 信号量集机制的基本思想是什么,它能解决什么问题？

答：AND 同步机制的基本思想是，将进程在整个运行过程中所需要的所有临界资源一次性全部分配给进程，待该进程使用完后再一起释放。只要尚有一个资源未能分配给该进程，其他所有可能为之分配的资源也不分配给它。亦即，对若干个临界资源的分配采取原子操作方式，要么全部分配到进程，要么一个也不分配。它能解决的问题：避免死锁的发生

63. 试述分页和分段的主要区别。
答：区别：（1）页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。分页仅仅是由于系统管理的需要而不是用户的需要。段则是信息的逻辑单位，它喊有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。
（2）页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而段的长度却不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。
（3）分页的作业地址空间是一维的，即单一的线性地址空间，程序员只需利用一个记忆符，即可以表示一个地址；而分段的作业地址空间则是二维的，程序员在标识一个地址时，既需要给出段名，，又需给出段内地址。

64. 什么是虚拟存储器，其实现方式有哪些？
答：虚拟存储器，是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储齐系统。
实现方式：1）分页请求系统 2）请求分段系统

65. 什么是临界资源？什么是临界区？

答：（1）临界资源是指每次仅允许一个进程访问的资源。
属于临界资源有硬件打印机、磁带机等,软件在消息缓冲队列、变量、数组、缓冲区等。
（2）不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问。每个进程中访问临界资源的那段代码称为临界区

66. 请说明 SPOOLing 系统的组成及特点？
SPOOLing 系统的组成：输入井和输出井、输入缓冲区和输出缓冲区、输入进程和输出进程。
SPOOLing 技术的特点：(1)提高了 I/O 速度. (2)将独占设备改造为共享设备。 (3)实现了虚拟设备功能.多个进程同时使用一独享设备,而对每一进程而言,都认为自己独占这一设备,不过,该设备是逻辑上的设备.

67. 内存管理有那些功能？
1）内存分配 2）内存保护 3）地址映射 4）内存扩充

68. 操作系统有哪几个特征？其最主要的特征是什么？虚拟存储器有哪些特征？其中最本质的特征是什么？
操作系统四个特征：并发，共享，虚拟和异步。其中最主要的特征是并发。
虚拟存储器特征：1）离散性 2）多次性 3）对换性 4）虚拟性其中最本质的特征是离散性

69. 请从调度性、并发行、拥有资源及系统开销四个方面对线程与进程作简单比较。

答:从调度,并发性,系统开销,拥有资源等方面来比较线程和进程:⑴调度.在传统的操作系统中,独立调度,分派的基本单位是进程.而在引入线程的操作系统中,则把线程作为调度和分派的基本单位.⑵并发性.在引入线程的操作系统中,不仅进程之间可以并发执行,而且在一个进程中的多个线程之间亦可并发执行,因而使操作系统具有更好的并发性,从而能更有效地使用系统资源和提高系统吞吐量.⑶拥有资源.不论是传统的操作系统,还是设有线程的操作系统,进程都是拥有资源的一个独立单位,它可以拥有自己的资源.一般地说,线程自己不拥有系统资源(也有一点必不可少的资源),但它可以访问其隶属进程的资源.⑷系统开销.由于在创建,撤销或切换进程时,系统都要为之分配或回收资源,保存 CPU 现场.因此,操作系统所付出的开销将显著地大于在创建,撤销或切换线程时的开销

70. 产生死锁的原因？产生死锁的必要条件是什么？
答：原因：1）竞争资源。2） 进程间推进顺序非法
必要条件：（1）互斥条件（2）请求和保持条件（3）不剥夺条件（4）环路等待条件

71. PCB 有何作用？为什么说 PCB 是进程存在的唯一标志？
PCB 的作用是使一个在多道程序环境下不能独立运行的程序，成为一个能独立运行的基本单位，一个能与其他进程并发执行的进程。进程创建时，操作系统首先就要为它分配一个 PCB，并通过 PCB 对进程实施有效的管理和控制，进程终止时，系统必须收回其 PCB，因为进程与 PCB 是一一对应的，系统通过 PCB “感知”到某个进程的存在。所以 PCB 是进程存在的唯一标志。

72. 引入缓冲的原因是什么？常见的缓冲区机制有哪些？
答：主要原因：（1）缓和 CPU 与 I/O 设备间速度不匹配的矛盾。（2）减少对 CPU 的中断频

率，放宽 CPU 中断响应时间的限制。（3）提高 CPU 和 I/O 设备之间的并行性。
最常见的缓冲区机制有单缓冲机制、能实现双向同时传送数据的双缓冲机制，以及能供多个设备同时使用的公用缓冲池机制

73. 为什么说并发和共享是操作系统的基本特征？

74. 为什么要在操作系统中引入线程？
引入线程的目的是为了进一步提高系统的并发程度，有效地提高系统的性能。

75. 简述基于索引节点共享方式的优缺点？
优点：文件的索引节点包括文件的物理地址及其他文件属性等信息,这些内容不放在目录项中,文件目录只设置文件名和指向索引节点的指针,用户对文件的添加和修改只引起索引节点内容的改变,对其它用户是可见的,从而能提供给其它用户共享
缺点：索引节点中设有一链接计数 count,表示共享此文件的用户数,当 count>1 时,文件主删除文件就会出现悬空指针,可能使其它用户的操作半途而废,若不删除文件,文件主必须为其它用户的操作付费.

76. 名词解释
临界区：不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问。每个进程中访问临界资源的那段代码称为临界区
临界资源：临界资源是指每次仅允许一个进程访问的资源
多道程序设计：为了进一步提高资源的利用率和系统吞吐量，引入了多道程序设计技术，由此形成了多道批处理系统。该系统中，用户所提交的作业都先存放在外存上并排成一个队列，称为“后备队列”；然后，由作业调度程序按一定的算法从后备队列中选择若干个作业调入内存，使它们共享 CPU 和系统中的各种资源。
并行：指两个或多个事件在同一时刻发生
并发：指两个或多个事件在同一时间间隔内发生
进程：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 ：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.进程控制块：进程控制块 PCB 是一个记录进程属性信息的数据结构，是进程实体的一部分，是操作系统中最重要的数据结构
动态重定位：把原来多个分散的小分区拼接成一个大分区，并对移动了的程序或数据进行重定位
虚拟存储器：是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储齐系统
文件控制块：文件控制块是操作系统为管理文件而设置的数据结构，存放了为管理文件所需的所有有关信息。文件控制块是文件存在的标志
文件目录：是一种数据结构，用语标识系统中的文件及其去里地址，供检索时使用

附录一 ASCII 码表

ASCII（美国信息交换标准编码）表											
字符	ASCII 代码			字符	ASCII 代码			字符	ASCII 代码		
	二进制	十进制	十六进制		二进制	十进制	十六进制		二进制	十进制	十六进制
回车	0001101	13	0D	?	0111111	63	3F	a	1100001	97	61
ESC	0011011	27	1B	@	1000000	64	40	b	1100010	98	62
空格	0100000	32	20	A	1000001	65	41	c	1100011	99	63
!	0100001	33	21	B	1000010	66	42	d	1100100	100	64
"	0100010	34	22	C	1000011	67	43	e	1100101	101	65
#	0100011	35	23	D	1000100	68	44	f	1100110	102	66
$	0100100	36	24	E	1000101	69	45	g	1100111	103	67
%	0100101	37	25	F	1000110	70	46	h	1101000	104	68
&	0100110	38	26	G	1000111	71	47	i	1101001	105	69
,	0100111	39	27	H	1001000	72	48	j	1101010	106	6A
(	0101000	40	28	I	1001001	73	49	k	1101011	107	6B
)	0101001	41	29	J	1001010	74	4A	l	1101100	108	6C
*	0101010	42	2A	K	1001011	75	4B	m	1101101	109	6D
+	0101011	43	2B	L	1001100	76	4C	n	1101110	110	6E
,	0101100	44	2C	M	1001101	77	4D	o	1101111	111	6F
-	0101101	45	2D	N	1001110	78	4E	p	1110000	112	70
.	0101110	46	2E	O	1001111	79	4F	q	1110001	113	71
/	0101111	47	2F	P	1010000	80	50	r	1110010	114	72
0	0110000	48	30	Q	1010001	81	51	s	1110011	115	73
1	0110001	49	31	R	1010010	82	52	t	1110100	116	74
2	0110010	50	32	S	1010011	83	53	u	1110101	117	75
3	0110011	51	33	T	1010100	84	54	v	1110110	118	76
4	0110100	52	34	U	1010101	85	55	w	1110111	119	77
5	0110101	53	35	V	1010110	86	56	x	1111000	120	78
6	0110110	54	36	W	1010111	87	57	y	1111001	121	79
7	0110111	55	37	X	1011000	88	58	z	1111010	122	7A
8	0111000	56	38	Y	1011001	89	59				
9	0111001	57	39	Z	1011010	90	5A	{	1111011	123	7B
:	0111010	58	3A	[	1011011	91	5B	|	1111100	124	7C
;	0111011	59	3B	\	1011100	92	5C	}	1111101	125	7D
<	0111100	60	3C	]	1011101	93	5D	~	1111110	126	7E
=	0111101	61	3D	^	1011110	94	5E				
>	0111110	62	3E	-	1011111	95	5F				