# 算法数据结构

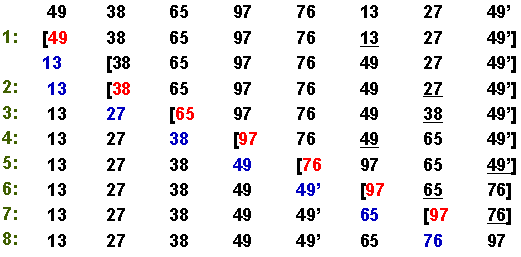
## 排序有几种方式，分别介绍其实现原理？（问）



### 冒泡排序

1. **void** bubble\_sort(**int** arr[], **int** len) {
2. **int** i, j;
3. **for** (i = 0; i < len - 1; i++)
4. **for** (j = 0; j < len - 1 - i; j++)
5. **if** (arr[j] > arr[j + 1])
6. swap(arr[j], arr[j + 1]);
7. }

### 简单选择排序

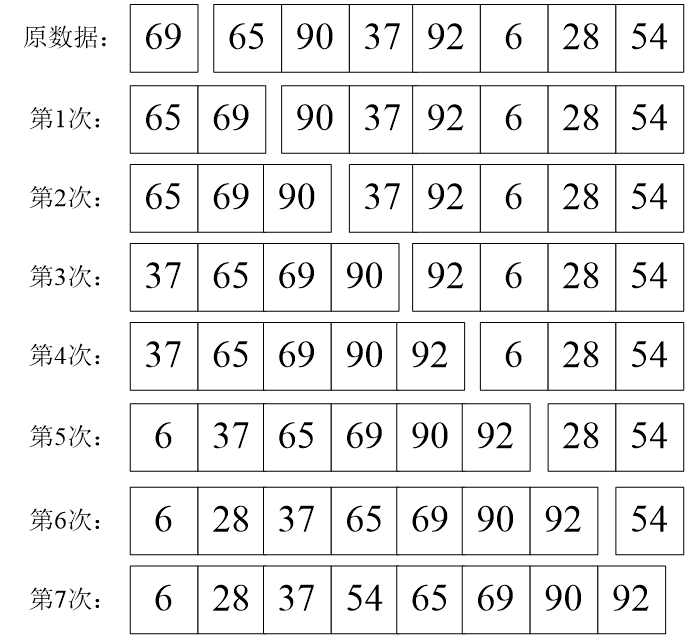


1. **void** selection\_sort(**int** arr[], **int** len) {
2. **int** i, j, min;
3. **for** (i = 0; i < len - 1; i++) {
4. min = i;
5. **for** (j = i + 1; j < len; j++)
6. **if**(arr[min] > arr[j])
7. min = j;
8. swap(arr[i], arr[min]);
9. }
10. }

### 快排（挖坑排序）（重要）

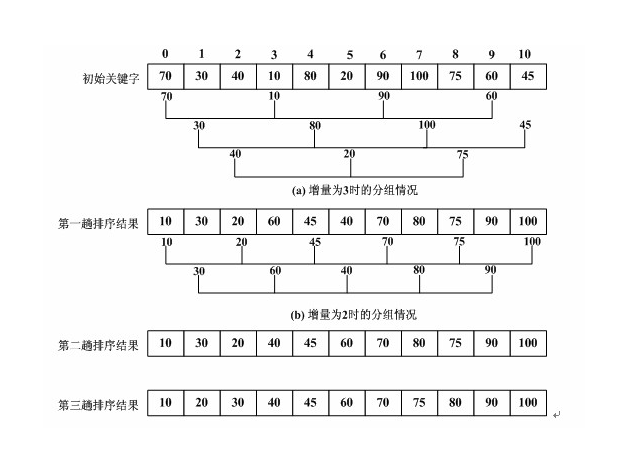
1. **void** quickSort(**int** s[], **int** l, **int** r)  {
2. **if** (l< r)   {
3. **int** i = l, j = r, x = s[l];
4. **while** (i < j)   {
5. **while**(i < j && s[j]>= x) // 从右向左找第一个小于x的数
6. j--;
7. **if**(i < j)
8. s[i++] = s[j];
9. **while**(i < j && s[i]< x) // 从左向右找第一个大于等于x的数
10. i++;
11. **if**(i < j)
12. s[j--] = s[i];
13. }
14. s[i] = x;
15. quickSort(s, l, i - 1); // 递归调用
16. quickSort(s, i + 1, r);
17. }
18. }

### (4) 插入排序



1. **void** insertion\_sort(**int** arr[], **int** len) {
2. **int** i, j, temp;
3. **for** (i = 1; i < len; i++) {
4. temp = arr[i];
5. **for** (j=i-1; j>=0 && arr[j]>temp; j--)
6. arr[j+1] = arr[j];
7. arr[j+1] = temp;
8. }
9. }

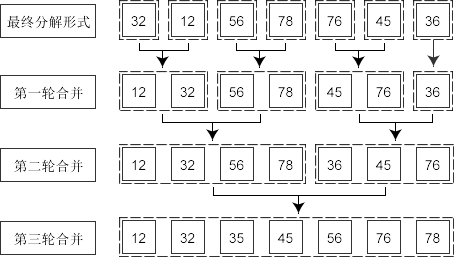
### (5)希尔排序



/\*希尔排序:先将整个待排元素序列分割成若干子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序（增量为1）。其时间复杂度为O(n^3/2),要好于直接插入排序的O(n^2) \*/

1. **void** shell\_sort(**int** arr[], **int** len) {
2. **int** gap=0, i, j;
3. **int** temp;
4. **while** (gap < len / 3)
5. gap = gap \* 3 + 1;
6. **for** (; gap > 0; gap /= 3)
7. **for** (i = gap; i < len; i++) {
8. temp = arr[i];
9. **for** (j=i; j>=gap; j-=gap) {
10. if(arr[j-gap] >temp)
11. arr[j] = arr[j-gap];
12. else
13. break;
14. }
15. arr[j] = temp;
16. }
17. }

### (6)归并排序



1. /\*假设初始序列含有n个记录,则可以看成n个有序的子序列,每个子序列的长度为1,然后两两归并,得到(不小于n/2的最小整数)个长度为2或1的有序子序列,再两两归并,...如此重复,直至得到一个长度为n的有序序列为止,这种排序方法称为2路归并排序.时间复杂度为O(nlogn),空间复杂度为O(n+logn),如果非递归实现归并,则避免了递归时深度为logn的栈空间, 空间复杂度为O(n) \*/  
   /\*lpos is the start of left half, rpos is the start of right half\*/  
   void merge(int a[], int tmp\_array[], int left\_start, int right\_start, int right\_end){  
       int i, left\_end, num\_elements, tmpos;  
       left\_end = right\_start - 1;  
       tmpos = left\_start;  
       num\_elements = right\_end – left\_start + 1;  
     
       while (left\_start <= left\_end && right\_start <= right\_end)  
           if (a[left\_start] <= a[right\_start])  
               tmp\_array[tmpos++] = a[left\_start++];  
           else  
               tmp\_array[tmpos++] = a[right\_start++];  
     
       while (left\_start <= left\_end) /\*copy rest of the first part\*/  
           tmp\_array[tmpos++] = a[left\_start++];  
       while (right\_start<=right\_end)/\*copy rest of the second part\*/  
           tmp\_array[tmpos++] = a[right\_start++];  
       /\*copy array back\*/  
       for (i = 0; i < num\_elements; i++, rightn--)  
           a[right\_end] = tmp\_array[right\_end];  
   }

void msort(int a[], int tmp\_array[], int left, int right){  
    int center;  
    if (left < right){  
        center = (right + left) / 2;  
        msort(a, tmp\_array, left, center);  
        msort(a, tmp\_array, center + 1, right);  
        merge(a, tmp\_array, left, center + 1, right);  
    }  
}

void merge\_sort(int a[], int n){  
    int \*tmp\_array;  
    tmp\_array = (int \*)malloc(n \* sizeof(int));  
    if (tmp\_array != NULL){  
        msort(a, tmp\_array, 0, n - 1);  
        free(tmp\_array);  
    }  
    else  
        printf("No space for tmp array!\n");

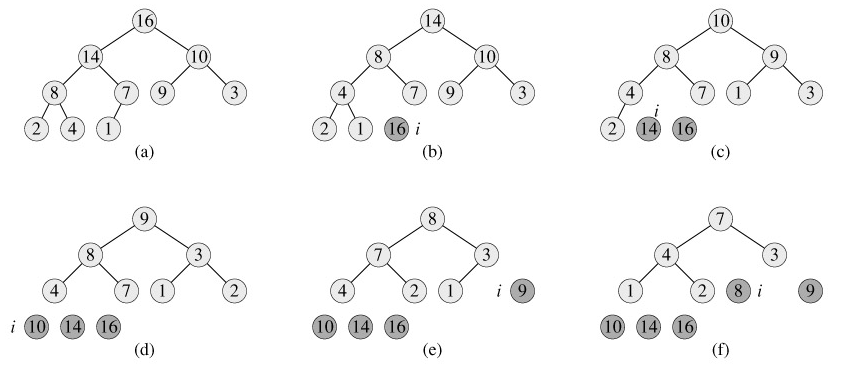
}

非递归实现：

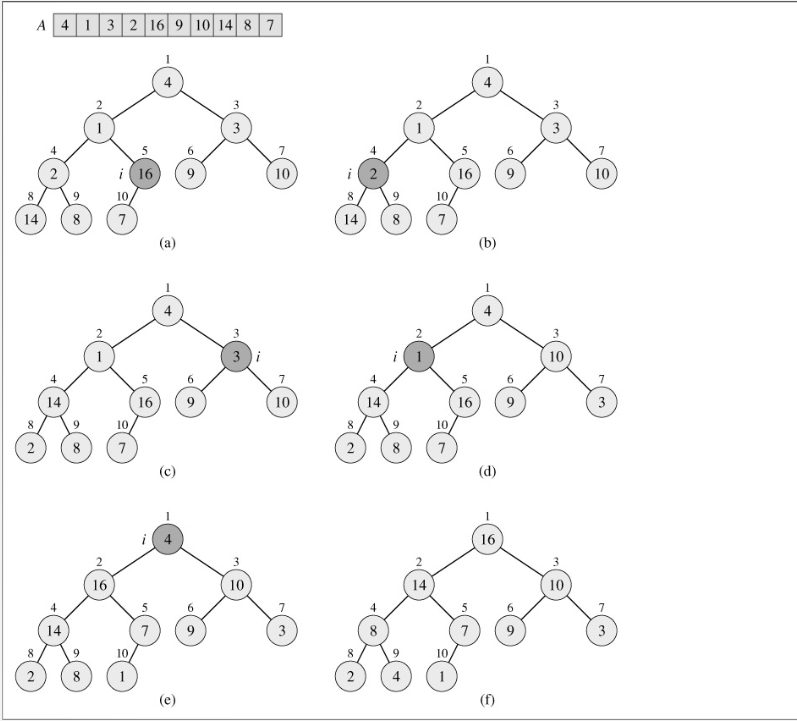
1. **void** merge\_sort(int arr[], **int** len) {
2. int\* a = arr;
3. int\* b = **new** int[len];
4. **for** (**int** seg = 1; seg < len; seg += seg) {
5. **for** (**int** start=0; start<len; start += seg+seg) {
6. **int** low=start, mid = min(start+seg, len);
7. int high = min(start+seg+seg, len), k = low;
8. **int** start1 = low, end1 = mid;
9. **int** start2 = mid, end2 = high;
10. **while** (start1 < end1 && start2 < end2)
11. b[k++] = a[start1] < a[start2] ? a[start1++] : a[start2++];
12. **while** (start1 < end1)
13. b[k++] = a[start1++];
14. **while** (start2 < end2)
15. b[k++] = a[start2++];
16. }
17. int\* temp = a;
18. a = b;
19. b = temp;
20. }
21. **if** (a != arr) {
22. **for** (**int** i = 0; i < len; i++)
23. b[i] = a[i];
24. b = a;
25. }
26. **delete**[] b;
27. }

### (7)堆排序

堆排序：



构建最大堆：



堆排序的基本思想就是：从最大（小）堆得顶部不断取走堆顶元素放到有序序列中，直到堆的元素被全部取完。堆排序完全依赖于最大（小）堆的相关操作。

① 初始化堆：将数列a[1...n]构造成最大堆。  
② 交换数据：将a[1]和a[n]交换，使a[n]是a[1...n]中的最大值；然后将a[1...n-1]重新调整为最大堆。 接着，将a[1]和a[n-1]交换，使a[n-1]是a[1...n-1]中的最大值；然后将a[1...n-2]重新调整为最大值。 依次类推，直到整个数列都是有序的。

/\* (最大)堆的向下调整算法

\* 注：数组实现的堆中，第N个节点的左孩子的索引值是(2N+1)，右孩子的索引是(2N+2)。

\* 参数说明：

\* a -- 待排序的数组

\* start -- 被下调节点的起始位置(一般为0，表示从第1个开始)

\* end -- 截至范围(一般为数组中最后一个元素的索引)

\*/

void maxheap\_down(int a[], int start, int end){

int c = start; // 当前(current)节点的位置

int left\_child = 2\*c + 1; // 左(left)孩子的位置

int tmp = a[c]; // 当前(current)节点的大小

for ( ;left\_child <= end; c=left\_child, left\_child =2\*left\_child +1){

if (left\_child < end && a[left\_child] < a[left\_child +1])

left\_child ++; // 左右两孩子中选择较大者，即m\_heap[l+1]

if (tmp >= a[left\_child])

break; // 调整结束

else{

a[c] = a[left\_child];

a[left\_child]= tmp;

}

}

}

/\*堆排序(从小到大)

\* 参数说明：

\* a -- 待排序的数组

\* n -- 数组的长度

\*/

void heap\_sort\_asc(int a[], int n) {

int i;

//从n/2-1 -->0逐次遍历。遍历之后，得到的数组实际上是一个最大堆。

for (i = n / 2 - 1; i >= 0; i--)

maxheap\_down(a, i, n-1);

// 从最后一个元素开始对序列进行调整，不断的缩小调整的范围直到第一个元素

for (i = n - 1; i > 0; i--) {

swap(a[0], a[i]);

maxheap\_down(a, 0, i-1);

}

}

heap\_sort\_asc(a, n)的作用是：对数组a进行升序排序；其中，a是数组，n是数组长度。  
heap\_sort\_asc(a, n)的操作分为两部分：初始化堆 和 交换数据。  
maxheap\_down(a, start, end)是最大堆的向下调整算法。

## 平衡二叉树（AVL）

https://www.cnblogs.com/suimeng/p/4560056.html

## 3.怎样判断一个链表有环

### 3.1怎样判断一个链表有环

（1）最常用方法：定义两个指针，同时从链表的头节点出发，一个指针一次走一步，另一个指针一次走两步。如果走得快的指针追上了走得慢的指针，那么链表就是环形链表；如果走得快的指针走到了链表的末尾（next指向 NULL）都没有追上第一个指针，那么链表就不是环形链表。

bool IsLoop(NODE \*head) {// 假设为带头节点的单链表

if (head == NULL)

return false;

node \*slow = head->next; // 初始时，慢指针从头节点开始走1步

if (slow == NULL)

return false;

node \*fast = slow->next; // 初始时，快指针从头节点开始走2步

while (fast!=NULL && slow!=NULL) //当单链表没有环时，循环到链表尾结束{

if (fast == slow)

return true;

slow = slow->next; // 慢指针每次走一步

fast = fast->next;

if (fast != NULL)

fast = fast->next;

}

return false;

}

（2）通过使用STL库中的map表进行映射。首先定义 map<NODE \*, int> m; 将一个 NODE \* 指针映射成数组的下标，并赋值为一个 int 类型的数值。然后从链表的头指针开始往后遍历，每次遇到一个指针p，就判断 m[p] 是否为0。如果为0，则将m[p]赋值为1，表示该节点第一次访问；而如果m[p]的值为1，则说明这个节点已经被访问过一次了，于是就形成了环。

map<NODE \*, int> m;

bool IsLoop\_2(NODE \*head){

if (head == NULL)

return false;

NODE \*p = head;

while (p) {

if (m[p] == 0) // 一般默认值都是0

m[p] = 1;

else if (m[p] == 1)

return true;

p = p->next;

}

return false;

}

### 3.2. 若单链表有环，如何找出环的入口节点? <1> 定义两个指针p1和p2，在初始化时都指向链表的头节点。  <2> 如果链表中的环有n个节点，指针p1先在链表上向前移动n步。  <3> 然后指针p1和p2以相同的速度在链表上向前移动直到它们相遇。  <4> 它们相遇的节点就是环的入口节点。

// 1、先求出环中的任一节点（slow）

NODE \*MeetingNode(NODE \*head) {

if (head == NULL)

return NULL;

node \*slow = head->next; // 初始时，慢指针从头节点开始走1步

if (slow == NULL)

return NULL;

node \*fast = slow->next; // 初始时，快指针从头节点开始走2步

while (fast != NULL && slow != NULL) {

if (fast == slow)

return fast;

slow = slow->next; // 慢指针每次走一步

fast = fast->next;

if (fast != NULL)

fast = fast->next;

}

return NULL;

}

// 2、从已找到的那个环中节点出发，一边继续向前移动，一边计数，当再次回到这个节点时，就可得到环中的节点数了。

NODE \*EntryNodeOfLoop(NODE \*head){

NODE \*meetingNode = MeetingNode(head); // 先找出环中的任一节点

if (meetingNode == NULL)

return NULL;

int count = 1; // 计算环中的节点数

node \*p = meetingNode;

while (p != meetingNode) {

p = p->next;

++count;

}

// p和q以相同的速度向前移动，当q指向环的入口节点时，p已经围绕着环走了一圈又回到了入口节点。

p = head;

for (int i = 0; i < count; i++)

p = p->next;

node \*q = head; // q从头节点开始

while (q != p) {

q = q->next;

p = p->next;

}

return p;

}

## 4. 两个有序的单链表如何合并成一个有序的单链表

**方法1（局部引用）**

        这种方法与上一种方法非常相似。这种方法避免使用虚拟节点(dummy node)，而是使用一个指向指针的指针，struct node\*\* lastPtrRef，这个指针指向结果链表的最后一个节点。在这个方法中，所有由虚拟节点完成的工作都有lastPtrRef完成。

1. **struct** node\* SortedMerge(**struct** node\* a, **struct** node\* b)  {
2. **struct** node\* result = NULL;
3. /\*point to the last result pointer \*/
4. **struct** node\*\* lastPtrRef = &result;
5. **while**(1)  {
6. **if**(a == NULL)   {
7. \*lastPtrRef = b;
8. **break**;
9. }
10. **else** **if**(b == NULL)   {
11. \*lastPtrRef = a;
12. **break**;
13. }
14. **if**(a->data <= b->data)   {
15. MoveNode(lastPtrRef, &a);
16. }
17. **else**   {
18. MoveNode(lastPtrRef, &b);
19. }
20. /\*tricky:advance to point to the next ".next" field \*/
21. lastPtrRef = &((\*lastPtrRef)->next);
22. }
23. **return** (result);
24. }

**方法2（递归）**

        合并操作是非常适合用递归来完成的一类操作，递归实现将会比迭代实现更加清晰且易于理解。尽管如此，你可能也不愿意使用递归来实现这个操作，因为递归方法所使用的栈空间与链表的长度成正比。

**[cpp]** [view plain](https://blog.csdn.net/lalor/article/details/7429989) [copy](https://blog.csdn.net/lalor/article/details/7429989)

1. **struct** node\* SortedMerge(**struct** node\* a, **struct** node\* b) {
2. **struct** node\* result = NULL;
3. /\*Base cases\*/
4. **if**(a == NULL)
5. **return** (b);
6. **else** **if**(b == NULL)
7. **return** (a);
9. /\*Pick either a or b, and recur \*/
10. **if**(a->data <= b->data) {
11. result = a;
12. result->next = SortedMerge(a->next, b);
13. }
14. **else**  {
15. result = b;
16. result->next = SortedMerge(a, b->next);
17. }
18. **return** (result);
19. }

## 5. 五子棋怎么判赢（检查当前落子点的一行，一列，左上右下和左下右上斜线是否有5个同样颜色的棋子）

## 6. 布龙过滤器，主要还是检查一个数据是否在海量数据中的方法（回答就是哈希，位图啥的）。

**布隆过滤器原理：**

布隆过滤器（Bloom Filter）的核心实现是一个超大的位数组和几个哈希函数。假设位数组的长度为m，哈希函数的个数为k

具体的操作流程：假设集合里面有3个元素{x, y, z}，哈希函数的个数为3。首先将位数组进行初始化，将里面每个位都设置位0。

对于集合里面的每一个元素，将元素依次通过3个哈希函数进行映射，每次映射都会产生一个哈希值，这个值对应位数组上面的一个点，然后将位数组对应的位置标记为1。

**布隆过滤器添加元素：**

将要添加的元素给k个哈希函数

得到对应于位数组上的k个位置

将这k个位置设为1

**布隆过滤器查询元素：**

将要查询的元素给k个哈希函数

得到对应于位数组上的k个位置

如果k个位置有一个为0，则肯定不在集合中

如果k个位置全部为1，则可能在集合中

注意：此处不能判断该元素是否一定存在集合中，可能存在一定的误判率。可以从图中可以看到：假设某个元素通过映射对应下标为4，5，6这3个点。虽然这3个点都为1，但是很明显这3个点是不同元素经过哈希得到的位置，因此这种情况说明元素虽然不在集合中，也可能对应的都是1，这是误判率存在的原因。

## 7. 腾讯面试题：给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？

**方案1：**oo，申请512M的内存，一个bit位代表一个unsigned int值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

**方案2：**因为2^32为40亿多，所以给定一个数可能在，也可能不在其中；这里我们把40亿个数中的每一个用32位的二进制来表示。假设这40亿个数开始放在一个文件中。

　　然后将这40亿个数分成两类: 1.最高位为0 2.最高位为1 并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20亿，而另一个>=20亿（这相当于折半了）；与要查找的数的最高位比较并接着进入相应的文件再查找

再然后把这个文件为又分成两类: 1.次最高位为0 2.次最高位为1，并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10亿，而另一个>=10亿（这相当于折半了）； 与要查找的数的次最高位比较并接着进入相应的文件再查找。 ....... 以此类推，就可以找到了,而且时间复杂度为O(logn)，方案2完。

**附：**位图法： 使用位图法判断整形数组是否存在重复、判断集合中存在重复是常见编程任务之一，当集合中数据量比较大时我们通常希望少进行几次扫描，这时双重循环法就不可取了。

　　位图法比较适合于这种情况，它的做法是按照集合中最大元素max创建一个长度为max+1的新数组，然后再次扫描原数组，遇到几就给新数组的第几位置上1，如遇到5就给新数组的第六个元素置1，这样下次再遇到5想置位时发现新数组的第六个元素已经是1了，这说明这次的数据肯定和以前的数据存在着重复。这种给新数组初始化时置零其后置一的做法类似于位图的处理方法故称位图法。它的运算次数最坏的情况是2N。如果已知数组的最大值即能事先给新数组定长的话效率还能提高一倍。

## 8. 搜索的时候输入一个字开始提示怎么实现的

## 9. 在我们使用淘点点时，怎么查找最近100家商户，数据量大；说出算法思想，要求效率高；

## 10.找出一个字符串中第一个出现次数为一次的字符的下标,如何优化算法使得遍历次数更少？

哈希表统计出现次数

## 11.int 数组中保存了一个整数，如{1,2,3}，实现一个对数组加1的函数，不考虑{0,1,2,3}这种情况

## 12. 双向链表排序用什么排序算法比较好

归并排序

## 13. 红黑树、平衡二叉树、B树的区别

https://blog.csdn.net/v\_july\_v/article/details/6530142

## 14. 栈用数组怎么实现

1. #define MAXSIZE 10;
3. **template**<**class** T>
4. **class** Stack {
5. **public**:
6. //默认构造函数
7. Stack();
8. Stack(**size\_t** maxElements);
9. Stack(T data[],**size\_t** maxElments);
10. ~Stack() {
11. **delete**[] arrays; }
12. //入栈
13. **void** Push(T data) {
14. **if**(isFull())
15. **throw** runtime\_error("Full stack");
16. **else**  {
17. top++;//指向栈顶
18. arrays[top]=data;
19. }
20. }
21. //出栈并返回
22. T Pop(){
23. **if**(isEmpty())
24. **throw** runtime\_error("No elements in the stack");
25. **else**  {
26. T data=arrays[top];
27. top--;
28. **return** data;
29. }
30. }
31. //返回栈顶元素
32. T Top(){
33. **if**(isEmpty())
34. **throw** runtime\_error("No elements in the stack");
35. **else**
36. **return** arrays[top];
37. }
38. //判断是否为空栈
39. **bool** isEmpty(){
40. **return** top==-1;
41. }
42. //栈是否已满
43. **bool** isFull(){
44. **return** top==maxSize-1;
45. }
46. //清空栈
47. **void** Clear(){
48. **while** (top!=-1)
49. top--;
50. }
51. //获得栈里元素的个数
52. **size\_t** GetSize(){
53. **return** top+1;
54. }
55. **private**:
56. **size\_t** top;
57. T \*arrays;
58. **size\_t** maxSize;
59. };
61. **template**<**class** T>
62. Stack<T>::Stack():  maxSize(MAXSIZE),top(-1) {
63. arrays=**new** T[maxSize];
64. **if**(arrays==NULL)
65. cout<<"动态分配内存失败";
66. }
68. **template**<**class** T>
69. Stack<T>::Stack(**size\_t** maxElements): maxSize(maxElements),top(-1)
70. {
71. arrays=**new** T[maxSize];//创建存储栈的数组
72. }
74. **template**<**class** T>
75. Stack<T>::Stack(T data[],**size\_t** maxElements): maxSize(maxElements),top(-1) {   arrays=**new** T[maxSize];//创建存储栈的数组
76. **for**(**size\_t** i=0;i<maxSize;i++)
77. arrays[i]=data[i];
78. top+=maxSize;
79. }

## 15．两个数进行交换，不额外使用变量，怎么做

a = a+b; a ^= b

b = a – b; 或者 b ^= a

a = a-b; a ^= b

## 16. 一个英文文章，含有各种单词和标点符号，从中找出A-Z都出现过至少一次的最小子串大小写不限，有点蒙了

## 17. 链表和数组分别适用于那些排序，为什么

## 18. 找数组中最大和最小元素，能否优化这个O（n ）

最简单的写法是这样：

**void** **naive**(**const** **int\*** x, **size\_t** n, **int\*** out\_min, **int** **\***out\_max) {

**int** min **=** x[0], max **=** x[0];

**for** (**size\_t** i **=** 1; i **<** n; i**++**) {

**if** (x[i] **<** min)

min **=** x[i];

**if** (x[i] **>** max)

max **=** x[i];

}

**\***out\_min **=** min;

**\***out\_max **=** max;

}

就是逐对处理 [1]：

**void** **pairwise**(**const** **int\*** x, **size\_t** n, **int\*** out\_min, **int** **\***out\_max){

**if** (n **==** 1) {

**\***out\_min **=** **\***out\_max **=** x[0];

**return**;

}

//初始化min和max

**int** min, max;

**if** (x[0] **<** x[1]) {

min **=** x[0]; max **=** x[1];

}

**else** {

min **=** x[1]; max **=** x[0];

}

**size\_t** i;

**for** (i **=** 2; i **<** n **-** 1; i **+=** 2)

**if** (x[i] **<** x[i **+** 1]) {

**if** (x[i] **<** min)

min **=** x[i];

**if** (x[i **+** 1] **>** max)

max **=** x[i **+** 1];

}

**else** { *// x[i] >= x[i + 1]*

**if** (x[i **+** 1] **<** min)

min **=** x[i **+** 1];

**if** (x[i] **>** max)

max **=** x[i];

}

}

**if** (i **!=** n) { *// 还有一个元素（因为奇数 n）*

**if** (x[i] **<** min)

min **=** x[i];

**else** **if** (x[i] **>** max) *// 加入 else*

max **=** x[i];

}

**\***out\_min **=** min;

**\***out\_max **=** max;

}

核心部分每次先比较两个相连的元素，然后把较小的与 min 比较，较大的与 max 比较，所以每对元素需要 3 次比较（(1)，(2A)或(2B)，(3A)或(3B)）。在偶数 n 的情况下，共需要3n/2次比较，奇数 n 则是(3n+1)\*/2或(3n+2)/2次。  
也可以写一个空间复杂度O(logn)的分治递归解：

**void** **divide**(**const** **int\*** x, **size\_t** n, **int\*** out\_min, **int** **\***out\_max){

assert(size **>** 0);

**if** (size **==** 1)

**\***out\_min **=** **\***out\_max **=** x[0];

**else** **if** (size **==** 2) {

**if** (x[0] **<** x[1]) {

**\***out\_min **=** x[0];

**\***out\_max **=** x[1];

}

**else** {

**\***out\_min **=** x[1];

**\***out\_max **=** x[0];

}

}

**else** {

**size\_t** half **=** n **/** 2;

**int** min1, max1, min2, max2;

divide(x, half, **&**min1, **&**max1);

divide(x **+** half, n **-** half, **&**min2, **&**max2);

**\***out\_min **=** min1 **<** min2 **?** min1 : min2;

**\***out\_max **=** max1 **>** max2 **?** max1 : max2;

}

}

这种分治方式，时间复杂度仍然是不变的，即O(n)。但如果有n/2个并行单元，最理想的延迟是O(logn)。

## 19. 求二叉树的最大距离（即相距最远的两个叶子节点），写代码。（动态规划）

https://my.oschina.net/wizardpisces/blog/116425

## 20.求一个单链表的中间节点，要求安全检查，能直接运行的程序。

## 21.求二叉树的宽度，先简介思路再写代码。

把二叉树中每层的节点依次放入一个队列中。设置一个变量width用于存储树的宽度。每一层的节点入队时计算该层节点的数目，如果该层次节点的数目大于width的值，那么把该层次节点的数目赋给width。如此，对二叉树按层遍历一遍之后width中保存的就是该二叉树的宽度。

1. **int** WidthOfTheTree(Node\* pRoot)  {
2. **if**(pRoot==NULL)
3. **return** 0;
4. queue<Node\*> MyQueue2;
5. MyQueue2.push(pRoot);
6. **int** width=1;
7. **int** curwidth=1;
8. **int** nextwidth=0;
9. **while**(!MyQueue2.empty())   {
10. **while**(curwidth!=0)   {
11. Node\* pTmp=MyQueue2.front();
12. MyQueue2.pop();
13. curwidth--;
14. **if**(pTmp->pLeft!=NULL)   {
15. MyQueue2.push(pTmp->pLeft);
16. nextwidth++;
17. }
18. **if** (pTmp->pRight!=NULL)   {
19. MyQueue2.push(pTmp->pRight);
20. nextwidth++;
21. }
22. }
23. **if**(nextwidth > width)
24. Width = nextwidth;
25. Curwidth = nextwidth;
26. Nextwidth = 0;
27. }
28. **return** width;
29. }

## 22.求二叉树的深度，先简介思路再写代码。

1. int Deep(TreeNode \*pRoot){
2. **if**(pRoot==NULL)
3. **return** 0;
4. **int** DL=DeepthOfTheTree(pRoot->pLeft);
5. **int** DR=DeepthOfTheTree(pRoot->pRight);
6. **int** Depth=DL>DR? DL:DR;
7. **return** Depth+1;
8. }

# 语言特性

## C++

### Volatile

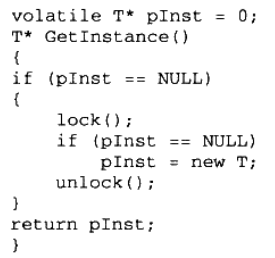
提醒编译器它后面所定义的变量随时都可能改变，因此编译后的程序每次需要存储或读取该变量时，都从变量地址中读取数据。

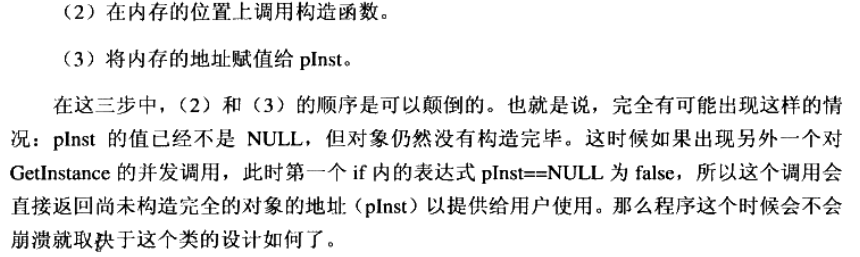
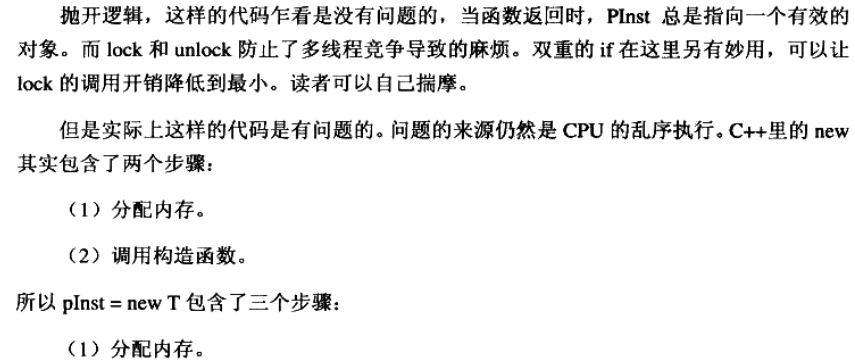
Volatile的作用：

* 1. 组织编译器为了提高速度将一个变量缓存到寄存器内不写回
  2. 阻止编译器调整操作volatile变量的指令顺序

但是volatile能够阻止编译器调整顺序，也无法阻止CPU动态调度换序，例

如：





解决方法：barrier指令

应用场景：

1. 中断服务程序中修改的供其他程序检测的变量
2. 多任务环境下个任务见共享的标志
3. 存储器映射的硬件存储器

### 四大转型操作符

static\_cast, const\_cast, dynamic\_cast, reinterpret\_cast

语法为：\*\*\*\_cast<type>(expression)

1. static\_cast

基本与拥有与C旧式转型相同的威力与意义，以及相同的限制。如：

int firstNum,secondNum;

double res =(double)firstNum / secondNum; //旧式C语法

double res =static\_cast<double>(firstNum) / secondNum; //新式C++转型符

1. const\_cast

用来改变表达式中的常量性(constness)或易变性(volatileness)。如：

int num;

const int \*cpNum =&num;

int \*pNum =cpNum; //error:cannot convert from 'const int \*' to 'int \*'

int \*pNum = (int \*)cpNum; //旧式C

int \*pNum =const\_cast<int \*>(cpNum); //新式C++const\_cast移除常量性

1. dynamic\_cast

用来执行继承体系中“安全的向下转型或跨系转型动作”。如：可以利用dynamic\_cast将“指向base class object 的pointer或reference”转型为“指向derived class object的pointer或reference”。如果转型失败，会以一个null指针或一个exception 表现出来（指针转型失败返回null,引用转型失败抛出异常）

class CBase { };

class CDerived:public CBase { };

CDerived dc;

CDerived \*dp =&dc;

CBase \*bp =dynamic\_cast<CBase \*>(dp); //使用dynamic\_cast将指向继承类的指针转化为指向基类的指针

CBase &br =dynamic\_cast<CBase &>(dc); //使用dynamic\_cast将指向继承类的引用转化为指向基类的引用

1. reinterpret\_cast

最常用的用途是转换"函数指针"类型。

typedef void(\*funcPtr)(); //funcPtr是个无参数返回值为void型的函数指针类型

int iFunc(){ return0; } //iFunc为一个无参数返回值为int型的函数

void func( funcPtrf ){} //func函数的参数是一个类型为funcPtr类型的函数指针

int main(){

func(iFunc()); //error:cannotconvert parameter 1 from 'int' to 'void (\_\_cdecl \*)(void)'

func(reinterpre\_cast<funcPtr>(iFunc); //right!reinterpre\_cast将返回值为int 的函数转化为返回值为void的函数

}

补充及修正：

1.static\_cast与dynamic\_cast

Static\_cast 常用于数据类型之间的转换，它也可以实现dynamic\_cast中的转换效果，当实现上行转换（继承类转换基类）时，二者可以实现相同的转换效果，当实现下行转换（基类转继承类）时，由于static\_cast没有动态类型检查机制，转换可能不安全。而正是由于static\_cast不作类型检查，它的转换比dynamic\_cast要快。

2. reinterpret\_cast

reinterpret\_cast字面意思是重新解释，主要是将数据从一种类型的转换为另一种类型。所谓“通常为操作数的位模式提供较低层的重新解释”也就是说将数据以二进制存在形式的重新解释。

int i;

char \*p = "Thisis a example.";

i =reinterpret\_cast<int>(p);

此时结果，i与p的值是完全相同的。reinterpret\_cast的作用是说将指针p的值以二进制（位模式）的方式被解释为整型，并赋给i，一个明显的现象是在转换前后没有数位损失。因此，reinterpret\_cast允许转换一个指针为其他类型的指针，也允许将一个指针转换为整数类型，反之亦然。这个操作符能够在非相关的类型之间进行转换。操作结果只是简单的从一个指针到别的指针的值的二进制拷贝，在类型之间指向的内容不做任何类型的检查和转换。这是一个强制转换。使用时有很大的风险，慎用之。比如：

class A { public:int m\_a; };

class B { public:int m\_b; };

class C : public A,public B {};

C c;

printf("%p, %p,%p\r\n", &c, reinterpret\_cast<B\*>(&c), static\_cast<B\*>(&c));

前两个的输出值是相同的，最后一个则会在原基础上偏移4个字节，这是因为static\_cast计算了父子类指针转换的偏移量，并将之转换到正确的地址（c里面有m\_a,m\_b，转换为B\*指针后指到m\_b处），而reinterpret\_cast却不会做这一层转换。因此, 你需要谨慎使用 reinterpret\_cast.

### 用C实现面向对象(例如多态)

### C++多态、虚函数、重载、多态是什么？虚函数实现原理

多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。（1）静态多态：由重载和模板实现。

（2）动态多态：由虚函数实现的。

虚函数的实现原理：虚函数表（vtbl），表中每一项指向一个虚函数地址，虚函数表具有继承性和多态性。虚函数表是在编译时就建立了。

### C++三大特性(继承、多态、封装及具体解释)

### New和malloc都能创建新的对象，区别在哪？哪个效率高

1. malloc/free是C++的标准库函数，new/delete是运算符
2. 不能讲执行构造函数和析构函数的任务强加给malloc/free，只能用new/delete
3. New返回指定类型的指针，malloc返回空指针再进行类型转换
4. new操作符从**自由存储区（free store）**上为对象动态分配内存空间，而malloc函数从**堆**上动态分配内存。
5. new内存分配失败时，会抛出bad\_alloc异常，它**不会返回NULL**；malloc分配内存失败时返回NULL。
6. 使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算，而malloc则需要显式地指出所需内存的尺寸。
7. operator new /operator delete的实现可以基于malloc，而malloc的实现不可以去调用new。

### 文件存储，文件删除了内存里面还有木有，能不能找回来

有，可以找回

### 内存分配（腾讯内推）

内存分配方式有三种：

[1] 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量, static 变量。

[2] 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中 ，效率很高，但是分配的内存容量有限。

[3] 从堆上分配，亦称动态内存分配 。程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。动态内存的生存期由程序员决定 ，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。

程序的内存空间：

代码区 (code area) ：程序内存空间

全局数据区 (data area)

堆区 (heap area)

栈区 (stack area)

一个由 C/C++ 编译的程序占用的内存分为以下几个部分 ：

[1]栈区（ stack ） 由编译器自动分配释放 ，存放为运行函数而分配的局部变量、函数参数、返回数据、返回地址等。其操作方式类似于数据结构中的栈。

[2]堆区（ heap ）一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收 。分配方式类似于链表。

[3]全局区（静态区）（ static ）存放全局变量、静态数据、常量。程序结束后有系统释放

[4]文字常量区: 常量字符串就是放在这里的。 程序结束后由系统释放。

堆与栈的比较（腾讯实习内推）:

[1]申请方式: stack由系统自动分配。heap需要程序员自己申请，并指明大小。堆:频繁new/delete会造成内存空间的不连续，造成大量的碎片，程序效率降低

[2]申请后系统的响应:栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。堆：首先应该知道操作系统有一个记录空闲内存地址的链表 ，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。（伙伴算法、SLAB算法）

[3]申请大小的限制: 堆向着内存地址增加的方向；栈是向着内存地址减小的方向增长。栈：在 Windows 下 , 栈是由高向低地址扩展的数据结构，是一块连续的内存的区域。 这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，能从栈获得的空间较小。 堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

[4]申请效率的比较: 栈由系统自动分配，速度较快。但程序员是无法控制的 。

堆是由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片 , 不过用起来最方便 。

### 接口、抽象类、集合

接口描述了类的行为和功能，而不需要完成类的特定实现。接口和抽象类都是继承树的上层，他们的共同点如下：  
 1) 都是上层的抽象层。  
 2) 都不能被实例化  
 3) 都能包含抽象的方法，这些抽象的方法用于描述类具备的功能，但是不比提供具体的实现。  
他们的区别如下：  
 1) 在抽象类中可以写非抽象的方法，从而避免在子类中重复书写他们，这样可以提高代码的复用性，这是抽象类的优势；接口中只能有抽象的方法。  
 2) 一个类只能继承一个直接父类，这个父类可以是具体的类也可是抽象类；但是一个类可以实现多个接口。

### 构造函数和析构函数是否可以定义为虚函数（构造函数不能，析构函数尽可能定义成虚函数）

构造函数不能：虚函数表是在构造函数中改写的

析构函数：虚函数

### 内部类（包含在一个类里面的类）的作用

1）内部类主要是为了避免命名冲突；（内部类定义为public）

2）为了隐藏名称（内部类定义为private/protected）

### ==和equals区别

==是一个**逻辑运算符**，判断两个值是否相等，而对于对象的引用来说，它的值就是它的指向；而equals()方法是一个普通的**方法**，它里面包含了自定义比较的成份，也就是说，你可以定义你自己的相等的标准，而不一定是比较引用的指向.

### 右值引用的概念

作用：右值引用引用的是寄存器

（1）消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。

（2）能够更简洁明确地定义泛型函数

<https://www.cnblogs.com/qicosmos/p/4283455.html>（经典）

### Auto关键字

1. 直观地，auto的一个最大的优势在于简化代码。由于C++的发展，变量类型变得越来越复杂。但是很多时候，名字空间、模板成为类型的一部分，导致了程序员在使用库的时候如履薄冰。
2. 可以避免类型声明时的麻烦而且避免类型声明时的错误。

使用时应该注意：

①可以使用valatile，pointer（\*），reference（&），rvalue reference（&&） 来修饰auto

1. auto k = 5;
2. auto\* pK = **new** auto(k);
3. auto\*\* ppK = **new** auto(&k);
4. **const** auto n = 6;

②用auto声明的变量必须初始化

auto m; // m should be intialized

③auto不能与其他类型组合连用

auto int p; // 这是旧auto的做法。

④函数和模板参数不能被声明为auto

1. **void** MyFunction(auto parameter){} // no auto as method argument
2. **template**<auto T> // utter nonsense - not allowed
3. **void** Fun(T t){}

⑤定义在堆上的变量，使用了auto的表达式必须被初始化

1. **int**\* p = **new** auto(0); //fine
2. **int**\* pp = **new** auto(); // should be initialized
3. auto x = **new** auto(); // Hmmm ... no intializer
4. auto\* y = **new** auto(9); // Fine. Here y is a int\*
5. auto z = **new** auto(9); //Fine. Here z is a int\* (It is not just an int)

⑥auto是一个占位符，并不是一个他自己的类型，因此不能用于类型转换或其他一些操作，如sizeof和typeid

1. **int** value = 123;
2. auto x2 = (auto)value; // no casting using auto
3. auto x3 = **static\_cast**<auto>(value); // same as above

⑦定义在一个auto序列的变量必须始终推导成同一类型

1. auto x1 = 5, x2 = 5.0, x3='r';  // This is too much....we cannot combine like this

⑧auto不能自动推导成CV-qualifiers（constant & volatile qualifiers），除非被声明为引用类型

1. **const** **int** i = 99;
2. auto j = i;       // j is int, rather than const int
3. j = 100           // Fine. As j is not constant
4. // Now let us try to have reference
5. auto& k = i;      // Now k is const int&
6. k = 100;          // Error. k is constant
7. // Similarly with volatile qualifer

⑨auto会退化成指向数组的指针，除非被声明为引用

1. **int** a[9];
2. auto j = a;
3. cout<<**typeid**(j).name()<<endl; // This will print int\*
4. auto& k = a;
5. cout<<**typeid**(k).name()<<endl; // This will print int [9]

### C++的lambda函数实现原理

原理：编译器会把一个lambda表达式生成一个匿名类的匿名对象，并在类中重载函数调用运算符。

#### 15.1无捕获列表和参数列表

**[cpp]** [view plain](https://blog.csdn.net/zhangxiangdavaid/article/details/44064765) [copy](https://blog.csdn.net/zhangxiangdavaid/article/details/44064765)

1. auto print = []{cout << "zhangxiang" << endl; };

编译器会把这一句翻译成如下情形：

1. //用给定的lambda表达式生成相应的类
2. **class** print\_class{
3. **public**:
4. **void** operator()(**void**) **const** {
5. cout << "zhangxiang" << endl;
6. }
7. };
8. //用构造的类创建对象，print此时就是一个函数对象
9. auto print = print\_class();

#### 15.2 无捕获列表但有参数列表

1. auto add = [](**int** a, **int** b){**return** a + b; };

编译器会把这一句翻译成如下情形：

1. **class** add\_class  {
2. **public**:
3. auto operator()(**int** a, **int** b) **const**   {
4. **return** a + b;
5. }
6. };
7. auto add = add\_class();

#### 15.3 有捕获列表，参数列表可选

由于捕获方式分为两种：引用捕获、值捕获。

#### （1） 值捕获

1. **int** year = 19900212;
2. **char** \*name = "zhangxiang";
3. //采用值捕获，捕获所有的已定义的局部变量，如year，name
4. auto print = [=](){
5. cout << year << ends << name << endl;
6. };

翻译

1. **int** year = 19900212;
2. **char** \*name = "zhangxiang";
3. **class** print\_class  {
4. **public**:
5. //根据捕获列表来决定构造函数的参数列表形式
6. print\_class(**int** year, **char** \*name) :year(year), name(name)   {
7. }
8. **void** operator()(**void**) **const**   {
9. cout << year << ends << name << endl;
10. }
11. **private**:
12. **int** year;
13. **char** \*name;
14. };
15. auto print = print\_class(a, str);

#### （2）引用捕获

1. **int** year = 19900212;
2. **char** \*name = "zhangxiang";
3. auto print = [&](){
4. year++;
5. cout << year << ends << name << endl;
6. };

翻译

1. **int** year = 19900212;
2. **char** \*name = "zhangxiang";
3. **class** print\_class  {
4. **public**:
5. //由于是引用捕获，参数列表采用引用的方式
6. print\_class(**int** &year, **char** \*&name) :year(year), name(name)    {
7. }
8. **void** operator()(**void**) **const**   {
9. year++;   //编译通过，const对引用类型无效
10. cout << year << ends << name << endl;
11. }
12. **private**:
13. **int** &year;
14. **char** \*&name;
15. };

（3）混合捕获

1. **int** year = 19900212;
2. **int** shoes = 42;
3. **char** \*name = "zhangxiang";
4. auto show = [&, shoes]()**mutable**{
5. shoes++;
6. year++;
7. cout << year << ends << shoes << ends << name << endl;
8. };

翻译

**[cpp]** [view plain](https://blog.csdn.net/zhangxiangdavaid/article/details/44064765) [copy](https://blog.csdn.net/zhangxiangdavaid/article/details/44064765)

1. **int** year = 19900212;
2. **int** shoes = 42;
3. **char** \*name = "zhangxiang";
4. **class** show\_class  {
5. **private**:
6. **int** &year;
7. **mutable** **int** shoes;
8. **char** \*&name;
9. **public**:
10. show\_class(**int** &year, **int** shoes, **char** \*&name) :year(year), shoes(shoes), name(name)  {
11. }
12. **void** operator()(**void**)**const**{
13. shoes++;
14. year++;
15. cout << year << ends << shoes << ends << name << endl;
16. }
17. };
18. auto show = show\_class(year, shoes, name);
19. show();

默认情况下，经过值捕获的变量是不可以被修改的，除非在参数列表后加关键字mutable。以上代码展示了，对应类中mutable是如何加的。当然还有另一种实现方法，只要lambda表达式加了mutable，重载函数调用运算符时就不加const修饰。

### C++的函数对象是如何实现的

类的对象跟括号结合，表现出函数一般的行为，这个对象可以称作是函数对象(类重载了()运算符)。引到Lambda

1. #include <iostream>
2. **using** **namespace** std;
3. **class** MyClass  {
4. **public**:
5. //重载函数调用运算符()
6. **int** operator()(**int** i)  {
7. **return** i;
8. }
9. };
10. **int** main() {
11. MyClass my;
12. //my()的调用行为似同函数
13. **int** i = my(1);     //本质是调用 my.operator()(1)
14. cout << "i = " << i << endl;
15. cin.get();
16. **return** 0;
17. }

运行

https://img-blog.csdn.net/20150304234000971

### 17.头文件include “”和<>区别

#include <> ：引用标准库头文件，编译器从标准库目录开始搜索

#incluce " "  ：引用非标准库的头文件，编译器从用户的工作目录开始搜索

### 18.一个空类多大，一个空类的子类如果也是空，多大，为什么

类的大小：

１．为类的非静态成员数据的类型大小之和．

２．有编译器额外加入的成员变量的大小，用来支持语言的某些特性（如：指向虚函数的指针）．

３．为了优化存取效率，进行的边缘调整．

４　与类中的构造函数，析构函数以及其他的成员函数无关

空类的大小：（空类同样可以被实例化），每个实例在内存中都有一个独一无二的地址，为了达到这个目的，编译器往往会给一个空类隐含的加一个字节，这样空类在实例化后在内存得到了独一无二的地址．所以空类的大小为１．

### 19.如果你定义一个string类，里面会有哪些函数

### 20.指针和引用的区别

（1）初始化要求不同：引用在创建时必须初始化，指针不必

（2）可修改性不同：引用一旦被初始化为指向一个对象，他就不能改变为另一个对象的引用，指针则不一样。

（3）不存在空引用，但是存在空指针

（4）测试需要的区别：由于引用不会指向空值，使用引用之前不需要测试其合法性，而指针则需要。因此引用代码效率比指针高。

### 21.static的用法（区分C语言和C++，两种语言下作用有所不同）。

在C语言中，static的用法：

* 1. 在函数体，一个被声明为静态的变量在函数调用过程中值不变
  2. 在模块内（函数体外），一个被声明为静态的变量可以被模块内所有函数访问，但不能被模块外其他函数访问，他是一个本地全局变量
  3. 在模块内，一个被声明为静态的函数只能被该模块内其他函数调用

在C++中，是static的用法：

（1）类的静态成员或方法不属于类实例，而属于类本身，并在所有实例间共享，调用时应用类名+::来引用

### 22.栈帧是什么？

栈帧也叫过程活动记录，是编译器用来实现过程/函数调用的一种数据结构”。可以简单理解为：栈帧就是存储在用户栈上的（当然内核栈同样适用）每一次函数调用涉及的相关信息的记录单元。

（未完待续）

### 23.函数传参是怎么传的？如何入栈？

### 24.多线程的适用场景是什么？为啥要用多线程？

### 25.Explicit的作用：防止执行隐式类型转换

### 26.C和c++中struct的区别，struct和class区别

C++中的struct对C中的struct进行了扩充，它已经不再只是一个包含不同数据类型的数据结构了，它已经获取了太多的功能。

C++中struct能包含成员函数吗？ 能！

C++中struct能继承吗？能！！

C++中struct能实现多态吗？能！！！

C语言中struct智能定义成员变量，不能定义成员函数，但是可以有函数指针。

struct和class有什么区别？

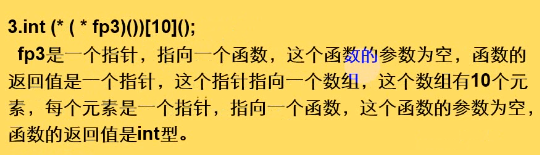
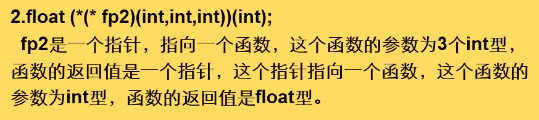
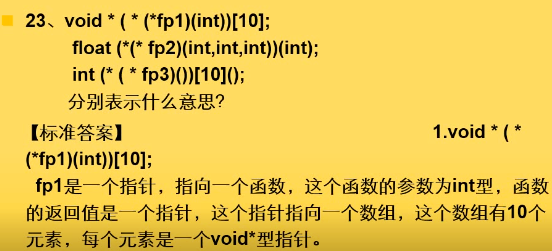
1）默认的继承访问权限。struct是public的，class是private的。

2）默认的数据访问控制。struct是public的，class是private的。

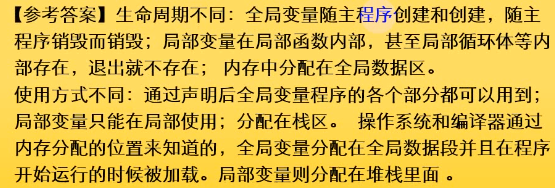
3）“class”还用于定义模板参数，就像“typename”。但关键字“struct”不用于定义模板参数。

### 27.函数assert（断言）的用法

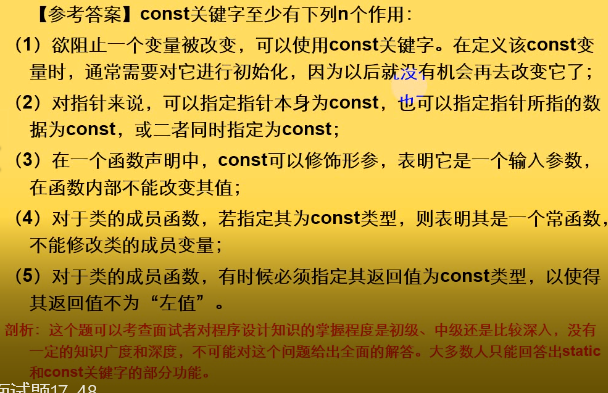
### 28.指针



### 29.全局变量和局部变量有什么区别，是怎么实现的？操作系统和编译器是怎么知道的？



### 30. const关键字的作用



与define相比，const内存效率更高，编译器通常将const 变量保存在符号表中，而不会分配存储空间，这使得它成为一个编译期间的常量，没有存储和读取的操作

### 31.智能指针

智能指针是一种资源管理类，通过对原始指针进行封装，在资源管理对象进

行析构时对指针指向的内存进行释放；通常使用引用计数方式进行管理，一

个基本实现如下：

### 32.内联函数、构造函数、静态成员函数可以是虚函数吗？

都不可以。

内联函数需要在编译阶段展开，而虚函数是运行时动态绑定的，编译时无法展开；

构造函数在进行调用时还不存在父类和子类的概念，父类只会调用父类的构造函数，子类调用子类的，因此不存在动态绑定的概念；

静态成员函数是以类为单位的函数，与具体对象无关，虚函数是与对象动态绑定的，因此是两个不冲突的概念。

## STL

### 什么是STL？

标准模板库（STL），它的基本概念就是把数据和操作分离，含有容器、算法、迭代器组件等。迭代器是容器和算法之间的粘合剂，使任何算法都可以和任何容器进行交互操作。

在STL中体现了泛型程序设计的思想，是以类型参数化的方式实现的（模板）。

STL中的容器：

序列容器：vector string deque list

关联容器：set map multiset multimap

适配容器：stack queue priority\_queue（优先队列）

### 2.Vector的底层是怎样的

vector就是动态数组.

在堆中分配内存,元素连续存放,有保留内存,如果减少大小后，内存也不会释放.如果新值>当前大小时才会再分配内存.当该数组后的内存空间不够时，需要重新申请一块足够大的内存并进行内存的拷贝。内部使用allocator类进行内存管理，程序员不需要自己操作内存。

对 vector 的任何操作，一旦引起空间重新配置，指向原 vector 的所有迭代器就都失效了。

注：size\_t是无符号整数！！！！

**Vector提供的函数：**

v.begin() v.end() v.size() v.capacity()

v.push\_back() v.pop\_back()

v.erase(iterator position) v.erase(iterator begin, end) ：执行后后面元素自动前移

v.clear()

v.insert(position, size\_type n, const &x)

### 3. list的实现，vector和list的区别

List是不连续的内存空间。

list和vector的区别：

（1）实现方式：vector是基于动态数组，内存空间是连续的，因此支持下标访问；list是基于双向链表，内存是不连续的，不支持下标访问，支持指针访问。

（2）如果需要高效的存取，而不在乎插入和删除的效率，选取vector。如果需要大量的插入删除而关心随机存取，选用list。

List.erase(it)执行后，迭代器it指向的内存已失效，再访问违规

### 4. Map, Set 等实现原理

Map和set是基于红黑树实现的

### 5. Deque

Deque支持随机访问和存取，支持下标访问，插入删除的效率也很高。是由一段一段定量的连续空间组成的。

实现：map+buffer

插入元素时随时可以重新增加一段新的空间并连接起来。

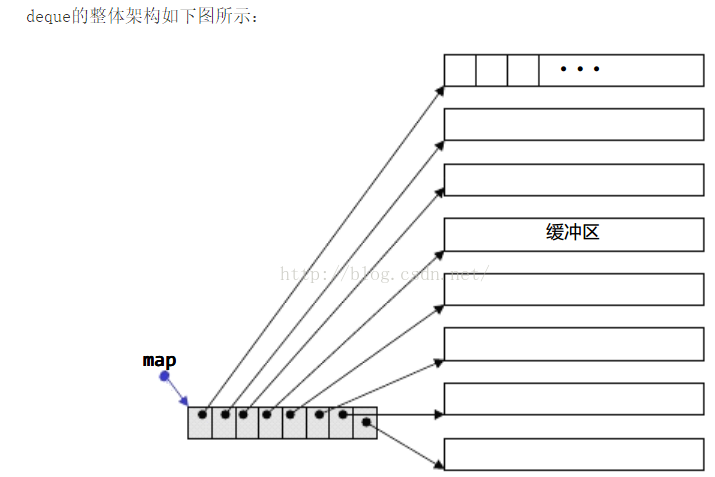
提供的函数：

Push\_back() pop\_back() push\_front() pop\_front()

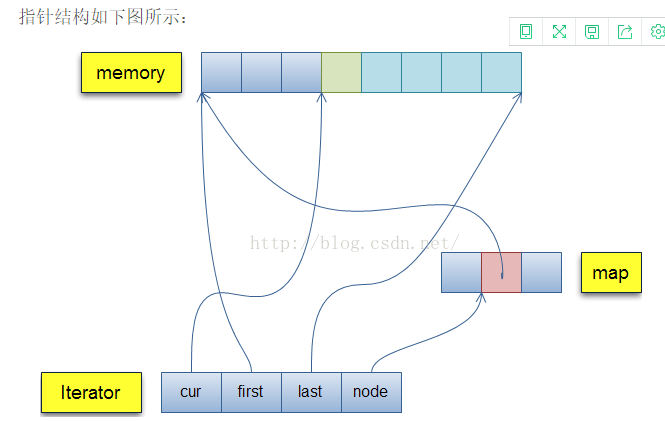
Clear() erase() insert()

Deque.at() deque.front() deque.back()

Deque.begin() deque.end() deque.rbegin() deque.rend()



Deque的迭代器：



首先，它必须能够指出分段连续空间（亦即缓冲区）在哪里，其次它必须能够判断自己是否已经处于其所在缓冲区的边缘，如果是，一旦前进或后退就必须跳跃至下一个或上一个缓冲区。为了能够正确跳跃，deque必须随时掌握管控中心（map）。所以在迭代器中需要定义：当前元素的指针，当前元素所在缓冲区的起始指针，当前元素所在缓冲区的尾指针，指向map中指向所在缓区地址的指针，分别为cur, first, last, node。

### 6.Stack queue

Stack和 queue是基于deque实现的 。

Stack： top() push() pop()

Queue： front() back() push() pop()

### 7.Unordered\_map（补充）

Unordered\_map是哈希表，各项操作的平均时间复杂度接近常数。

可以自定义hash函数和比较函数

哈希函数：线性探测：H+i

二次探测：H+i^2

开链法

哈希函数用质数取模这就是为了使得有特征的数据集也能均匀映射到映射空间。 在这种情况下，有时hash函数还要结合数据特征，让出现概率较大的数据集有较小的碰撞概率，出现概率较小的数据集有较大的碰撞概率。这样，就可以减少整体数据集的碰撞概率。

### 8.空间适配器

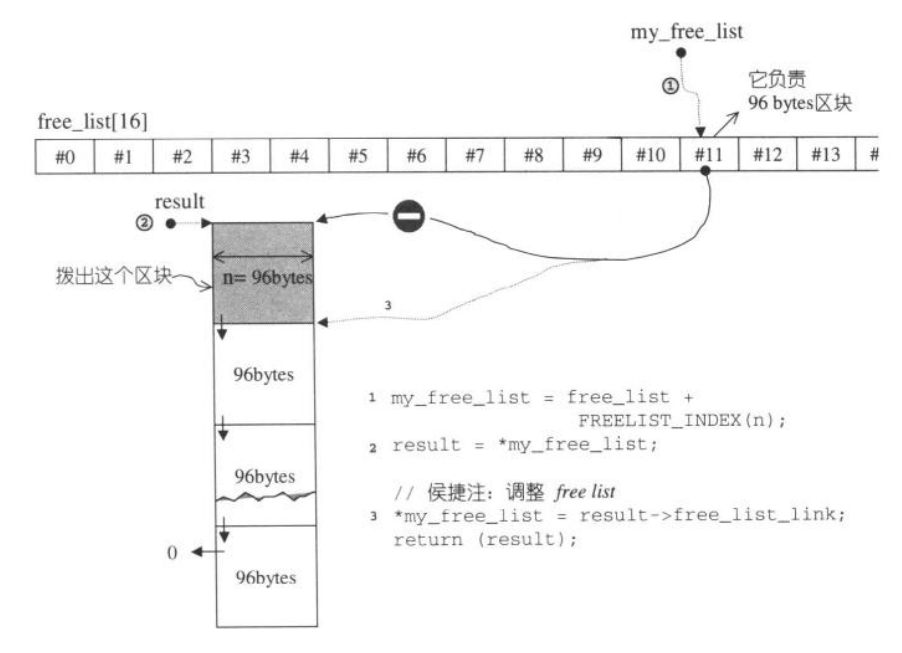
当请求的内存大于128b时，就用第一层配置器分配内存，当请求的内存小于等于128b时就调用第二层配置器。

第一级配置器相对简单，因为使用的正是我们平常使用的malloc，dealloc，free等，但是这个配置器提供了当内存配置错误时的处理函数oom\_malloc，这个函数会调用\_\_malloc\_alloc\_oom\_handler)() 这个函数，去企图释放内存，然后重新调用malloc分配内存。这个函数默认是0，所以malloc调用失败默认操作是返回\_THROW\_BAD\_ALLOC。

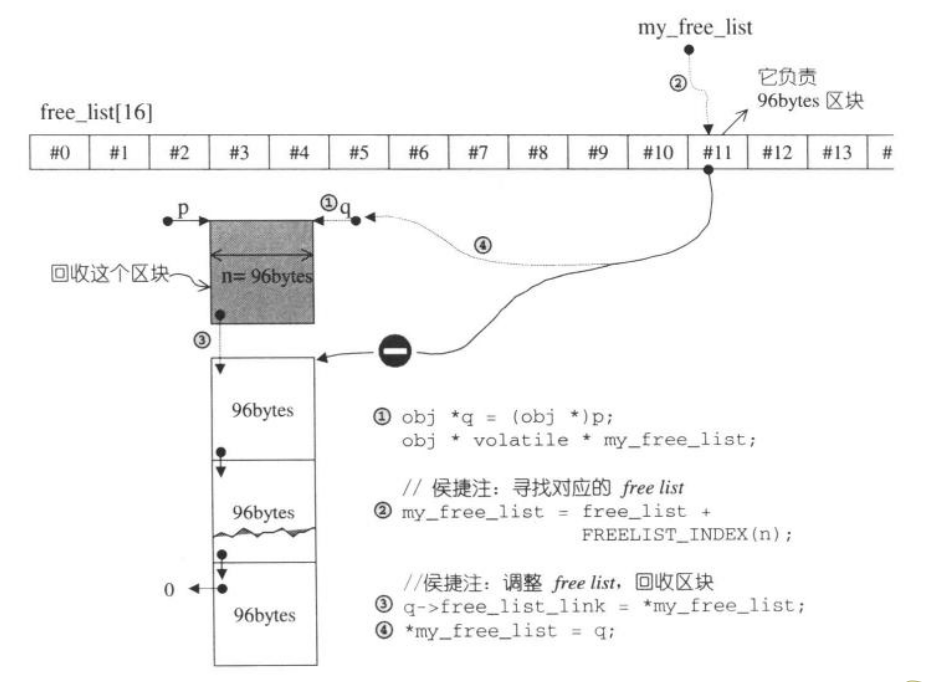
第二层配置器有有一个内存池，用一个union obj数组free\_list来存储内存的地址，数组的每一个元素都指向一个obj链表，也就是内存链表。数组从小到大表示负责8b,16b,24b,...,120b,128b内存请求。当请求的内存为n时，会将请求上条至2的指数大小，并从数组相应位置获取内存。例如如果请求为20b，则请求会上调至24b 。（伙伴算法）

内存池模型：

这个图展示了当有内存请求到达时，先找到负责这个内存大小的数组元素指向的内存链表，取出第一块内存，然后把数组元素(obj指针)指向第二块内存的首地址，

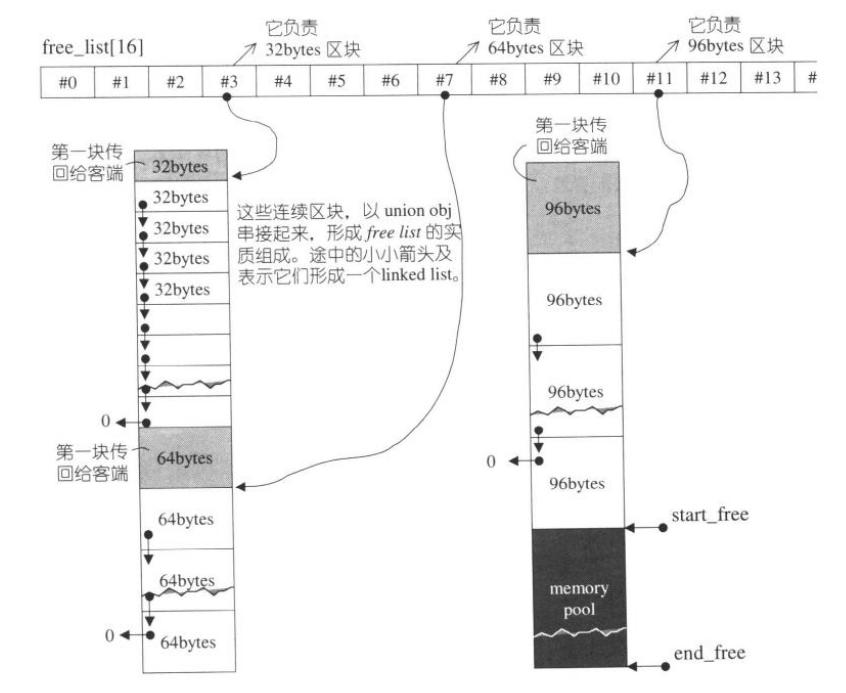


当程序释放这块内存时，第二级配置器还负责回收这块内存，等下次有请求时，可以直接使用这块内存。示意图如下，从《STL源码剖析》摘取:

[](http://7xjnip.com1.z0.glb.clouddn.com/ldw-%E9%80%89%E5%8C%BA_020.png);

先计算机这块内存属于哪个数组元素负责，然后将这块回收的内存放置链表的第一个位置，这块内存的下一块内存为这个链表原先的第一块内存

之前分析分配内存时，当free\_list没有可用的内存时，会调用refill来从内存池分配内存。例如，如果请求内存为32b，此时内存链表中没有足够的内存了，那么refill会分配20块32b的内存块，然后把第一块返回给程序，其他19块由数组相应链表管理。



当free\_list没有内存返回给用户时，refill函数会调用chunk\_alloc从内存池获取内存，如果内存池剩余的内存(end\_free-start\_free)满足需求的内存(size\*nobjs)，则直接从从内存池获取内存，返回给程序；当内存池的块不能满足20块时，返回一个以上的内存块；当内存池一块都不能满足时，先是回收剩余的内存，然后调用malloc从系统获取内存。如果系统内存不足，则先从数组其他链表获取内存，如果其他链表也不足够内存的话，就调用第一级内存来分配，因为第一级内存分配失败，有处理函数来解决。

### 9.萃取器

如何让编译器推导出返回值？普通的做法是使用typedef，但直接将迭代器传入的类型做typedef并不能解决泛型算法对裸指针的兼容。

### 10.STL算法

Reversr< vector<string>::iterator>(v.begin(), v.end());

For\_each()泛型算法

Find()函数

### 11.函数适配器

# 系统实现（linux）

## 1.linux命令、查cpu使用率

答：a.方法一：top命令

在终端中直接执行top命令，即可看到cpu的使用率。(h查看命令帮助，q退出top). 如果是多核CPU，希望查询每个CPU的使用率，可以在top命令的界面，按数字键1,即可看到其它CPU的使用率

b. 方法二：vmstat命令

vmstat是一个虚拟内存命令，但是本身也可以看到cpu的使用率，可以执行vmstat -w-w参数可以等宽输出信息，便于用户查询。

c. 方法三：sysstate工具包

首先要先安装sysstate工具包，如果系统中没有安装，则需要自行安装，以deepin linux为例，执行如下命令安装

sudo apt-get install sysstate

在sysstate工具包中，有一个sar命令，可以根据时间和次数来查看cpu使用率，在一般的测试中是非常有用的。比如执行下面的命令：

sar -u 1 5

使用sar命令，如果要获取某个CPU的使用率，可以通过-P参数来指明。

在sysstate工具包中，还有一个mpstat，它与sar命令的使用方法类似，但是却可以看到nice值等更多的信息。

d.方法四：dstat命令

如果没有dstat命令，则需要进行安装。以deepin linux为例:

sudo apt-get install dstat

安装完成后，使用dstat命令的-c参数，即只显示cpu状态信息。

## 2.指针的寻址过程

数组是相同数据类型的数据集合，以线性方式连续存储在内存中，而指针只是保存一个地址值的4字节变量。在使用中，数组名是一个地址常量值，保存数据首元素地址，不可修改，因此只能以基地址访问内存数据。指针是一个变量，可以修改，随意访问内存数据。

数组作为参数和返回值，都是将数组的首地址以指针的方式进行传递，但是它们也有不同。将数组作为参数时，其定义所在的作用域必然在函数调用以外，在调用之前已经存在。而数组作为函数的返回值则有一定的风险。当数组作为局部变量数据时，便出现了稳定性问题。

P.S. 字符类型的数组的赋值（初始化）其实是复制字符串的过程。这里并不是单字节的复制，而是每次复制4字节的数据。

访问数组的方式有两种：通过下标寻址和通过指针寻址。指针寻址没有下标寻址便利，效率也低。因为指针是存放地址数据的变量类型，因此在数据访问的过程中，要先取出指针变量中的数据，然后再针对数据进行地址偏移计算，从而寻址到目标数据。而数组名本身就是常量地址，可直接对数据名所代替的地址进行偏移计算。

1. 当下标值为整型常量时，由于类型大小已知，编译器可直接计算出数据所在的地址。其寻址过程与局部变量相同。

2. 当下标值为整型变量时，编译器无法计算出相应的地址，只能先进行地址偏移计算，然后得出目标数据所在的地址

3. 当下标值为整型表达式时，先计算出表达式的结果，然后将其结果作为下标值。如果是常量计算，常量会被折叠，最后还是以常量为下标寻址。

## 3.Linux文件的三个时间（acm）

修改时间(mtime)：文件的内容被最后一次修改的时间，我们经常用的ls -l命令显示出来的文件时间就是这个时间，当用vim对文件进行编辑之后保存，它的mtime就会相应的改变；

访问时间(atime)：对文件进行一次读操作，它的访问时间就会改变。例如像：cat、more等操作，但是像之前的state还有ls命令对atime是不会有影响的；

状态时间(ctime)：当文件的状态被改变的时候，状态时间就会随之改变，例如当使用chmod、chown等改变文件属性的操作是会改变文件的ctime的。

除了可以通过stat来查看文件的mtime,ctime,atime等属性，也可以通过ls命令来查看，具体如下:

ls -lc filename 列出文件的 ctime （最后更改时间）

ls -lu filename 列出文件的 atime（最后存取时间）

ls -l filename 列出文件的 mtime （最后修改时间）

## 4.常用的Linux命令

Find

awk 命令：awk 是一个强大的文本分析工具，使用 awk 命令处理一个或者多个文件时，它会依次读取文件的每一行内容, 然后对其进行处理，awk 其实就是一个脚本编程语言。

## 5.进程管理的一些命令

ps -l命令能查看当前 bash 下的相关进程全部信息

pstree

监控进程的变化：top

Kill

我们需要通过以下的方法来查看进程的详细信息：

Linux在启动一个进程时，系统会在/proc下创建一个以PID命名的文件夹，在该文件夹下会有我们的进程的信息，其中包括一个名为exe的文件即记录了绝对路径，通过ll或ls –l命令即可查看。

ll /proc/PID

## 6. linux下如何修改进程优先级？（nice命令的使用）。

1、一开始执行程序就指定nice值：nice

nice -n -5 /usr/local/mysql/bin/mysqld\_safe &

语　　法：nice [-n <优先等级>][--help][--version][执行指令]

参　　数：-n<优先等级>或-<优先等级>或–adjustment=<优先等级> 　设置欲执行的指令的优先权等级。等级的范围从-20-19，其中-20最高，19最低，只有系统管理者可以设置负数的等级。

2、调整已存在进程的nice：renice

renice -5 -p 5200

#PID为5200的进程nice设为-5

语　　法：renice [优先等级][-g <程序群组名称>...][-p <程序识别码>...][-u <用户名称>...]

参　　数：

-g <程序群组名称> 　使用程序群组名称，修改所有隶属于该程序群组的程序的优先权。

-p <程序识别码> 　改变该程序的优先权等级，此参数为预设值。

-u <用户名称> 　指定用户名称，修改所有隶属于该用户的程序的优先权。

3、也可以用top命令更改已存在进程的nice：

top#进入top后按“r”–>输入进程PID–>输入nice值

## 7.linux下性能监控命令uptime介绍，平均负载的具体含义是什么？建议看server load概念。

uptime命令能够打印系统总共运行了多长时间和系统的平均负载。uptime命令可以显示的信息显示依次为：现在时间、系统已经运行了多长时间、目前有多少登陆用户、系统在过去的1分钟、5分钟和15分钟内的平均负载。

系统平均负载是指在特定时间间隔内运行队列中的平均进程数。如果每个CPU内核的当前活动进程数不大于3的话，那么系统的性能是良好的。如果每个CPU内核的任务数大于5，那么这台机器的性能有严重问题。

如果你的linux主机是1个双核CPU的话，当Load Average 为6的时候说明机器已经被充分使用了。

## 8.linux下如何调试程序？说到gdb，具体如何调试？如何查看core文件中的堆栈信息等（bt指令）。

Bt：看函数调用栈信息

## 9. Linux Cpu占用高调试方法

1.用top命令确定 有问题的pid

2.用top -H -p pid命令查看进程内各个线程占用的CPU百分比

>top -H -p 14094

3.用gdb来attach目标进程

gdb

gdb>attach 2907

调试完指定进程后，可以运行detach命令来让GDB释放该进程，该进程得以继续运行。

## 10.select 和epoll（重要）

（1）Select：不断地轮询所负责的所有socket，当某个socket有数据到达时，就通知用户进程。（默认是1024）

实现：

<1>使用copy\_from\_user从用户态拷贝fd\_set到内核态

<2>注册回调函数\_pollwait

<3>遍历所有fd，调用其对应的poll函数

<4>poll函数返回一个描述读写是否就绪的mass掩码，根据掩码对fd\_set赋值

<5>若遍历完所有的fd，没有可读可写的mask掩码，则调用schedule\_timeout使调用select的进程进入睡眠。若超过一定时间仍未唤醒，则该进程被重新唤醒获得CPU，重新遍历fd。

<6>吧fd\_set从内核态拷贝到用户态。

（2）epoll

Int epoll\_creat( int size); 创建一个epoll fd。

Int epoll\_ctl( int epfd, int op, int fd, struct epoll\_evevt \*event);

Int epoll\_wait( int epfd, struct epoll\_evevt \*events, int maxevents, int timeout);

Epoll高效的原因：

<1>select/poll每次调用都要将监控的fd传递给select/poll系统调用，需要从用户态拷贝到内核态，而调用epoll\_wait时不需要。Epoll\_creat在内核态准备数据结构存放fd，epoll\_ctl对此数据结构进行监控。

<2>内核使用了slab机制，为epoll提供了快速的数据结构。

在内核里，一切皆文件。Epoll向内核注册了一个文件系统，用于存储上述被监控的fd。当调用epoll\_creat时，就会在这个虚拟的epoll文件系统里创建一个file结点。Epoll在被内核初始化时，开辟出epoll自己的高速cache区，用来安置每一个我们想要监控的fd，以红黑树保存在内核中。

<3>当调用epoll\_ctl塞入百万个fd时，epoll\_wait仍然可以快速返回，并有效的将发生时间的fd给用户（建立了一个list链表，存储准备就绪的事件。）

List链表维护：epoll\_ctl将fd放到epoll文件系统里file对应的红黑树外，还在内核中断处理程序注册了一个回调函数，告诉内核如果该fd中断到了，就把它放到list链表里。Epoll\_wait只需要观察该链表有没有数据，如果有数据就返回就绪链表里面的数据。

**Epoll的两种模式：**

1. 水平触发（LT）：使用此种模式，当数据可读的时候，epoll\_wait()将会一直返回就绪事件。如果你没有处理完全部数据，并且再次在该epoll实例上调用epoll\_wait()才监听描述符的时候，它将会再次返回就绪事件，因为有数据可读。ET只支持非阻塞socket。

2. 边缘触发（ET）：使用此种模式，只能获取一次就绪通知，如果没有处理完全部数据，并且再次调用epoll\_wait()的时候，它将会阻塞，因为就绪事件已经释放出来了。

ET的效能更高，但是对程序员的要求也更高。在ET模式下，我们必须一次干净而彻底地处理完所有事件。LT两种模式的socket都支持。

# 网络

## 1.HTTP协议

HTTP：超文本传输协议，基于TCP/IP

主要特点：

（1）简单快速：客户向服务器请求服务时，只需要传递请求方法和路径

（2）灵活：允许传输任意类型的数据对象

（3）无连接：每次连接只处理一个请求，服务器处理完客户端请求并收到客户端应答后立即断开

（4）无状态协议：协议对事物处理没有记忆能力

（5）支持B/S和C/S模式

HTTP1.0的请求方法：GET POST HEAD

HTTP1.1的请求方法：OPTIONS PUT DELETE TRACE CONNECT

## 2.在浏览器地址栏输入URL，按下回车后的过程：

（1）浏览器向DNS服务器请求解析该URL对应的IP地址

（2）根据该IP地址和端口号，和服务器建立TCP协议

（3）浏览器发出读取文件的http请求，该请求报文作为TCP三次握手的第三个报文数据发送给服务器

* 1. 服务器做出相应，并将对应的HTML文本发送给浏览器
  2. 释放TCP连接
  3. 浏览器显示该内容

## 3.http和https的区别

HTTPS：是以安全为目标的HTTP通道，即HTTP下加入SSL层，HTTPS的安全基础是SSL。

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

HTTPS和HTTP的区别主要如下：

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

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

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## 4. TCP长连接的保持消耗流量不，有什么优劣势

（1）长/短连接操作过程

短连接的操作步骤是：

建立连接——数据传输——关闭连接...建立连接——数据传输——关闭连接

长连接的操作步骤是：

建立连接——数据传输...（保持连接）...数据传输——关闭连接

（2）TCP保活功能

保活功能主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，  
从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将应远等待客户端的数据，保活功能就是试图在服务器端检测到这种半开放的连接。

如果一个给定的连接在2小时内没有任何的动作，则服务器就向客户发一个探测报文段，客户主机必须处于以下4个状态之一：

客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。

客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在75秒后超时。服务器总共发送10个这样的探测 ，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

客户主机崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。

客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探查的响应。

（3）长/短连接的优点和缺点

长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户来说，较适用长连接。

不过这里存在一个问题，存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，  
 client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在TCP的建立和关闭操作上浪费时间和带宽。

（4）长/短连接的应用场景

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，再次处理时直接发送数据包就OK了，不用建立TCP连接。  
 例如：数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，  
而且频繁的socket 创建也是对资源的浪费。  
 而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

## 5.ping命令过程

Ping命令是基于ICMP的查询报文，分为回送请求和回送应答。请求类型为8，应答类型为0.

Ping [www.baidu.com](http://www.baidu.com)的过程：

（1）通过DNS服务，将域名解析成ip地址

（2）piing命令发送一个带有32字节数据的ICMP请求包，收到回复后显示结果

ping的原理是用类型码为0的ICMP发请求，收到请求的主机则用类型码为8的ICMP回应。通过计算ICMP应答报文数量和与接受与发送报文之间的时间差，判断当前的网络状态。

这个往返时间的计算方法是：ping命令在发送ICMP报文时将当前的时间值存储在ICMP报文中发出，当应答报文返回时，使用当前时间值减去存放在ICMP报文数据中存放发送请求的时间值来计算往返时间。ping返回接受到的数据报文字节大小、TTL值以及往返时间。

Unix系统在实现ping程序时是把ICMP报文中的标识符字段置成发送进程的 ID号。这样 即使在同一台主机上同时运行了多个 ping程序实例，ping程序也可以识别出返回的信息。

Traceroute命令：追踪网络数据包的路由途径，也是基于ICMP协议的

## 6.HTTP各个状态码代表的含义

1\*\*：请求已接收，继续处理

2\*\*：成功

3\*\*：重定向

4\*\*：客户端错误

5\*\*：服务器错误

常见的状态码：

100 (继续) 请求者应当继续提出请求。服务器返回此代码表示已收到请求的第一部分，正在等待其余部分。

101 (切换协议) 请求者已要求服务器切换协议，服务器已确认并准备切换。

200 (成功) 服务器已成功处理了请求。

201 (已创建) 请求成功并且服务器创建了新的资源。

202 (已接受) 服务器已接受请求，但尚未处理。

203 (非授权信息) 服务器已成功处理了请求，但返回的信息可能来自另一来源。

204 (无内容) 服务器成功处理了请求，但没有返回任何内容。

205 (重置内容) 服务器成功处理了请求，但没有返回任何内容。

206 (部分内容) 服务器成功处理了部分 GET 请求。

300 (多种选择) 针对请求，服务器可执行多种操作。 服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。

301 (永久移动) 请求的网页已永久移动到新位置。 服务器返回此响应(对 GET 或 HEAD 请求的响应)时，会自动将请求者转到新位置。

302 (临时移动) 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

303 (查看其他位置) 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。

304 (未修改) 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。

305 (使用代理) 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理。

307 (临时重定向) 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

400 (错误请求) 服务器不理解请求的语法。

401 (未授权) 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403 (禁止) 服务器拒绝请求。

404 (未找到) 服务器找不到请求的网页。

405 (方法禁用) 禁用请求中指定的方法。

406 (不接受) 无法使用请求的内容特性响应请求的网页。

407 (需要代理授权) 此状态代码与 401(未授权)类似，但指定请求者应当授权使用代理。

408 (请求超时) 服务器等候请求时发生超时。

409 (冲突) 服务器在完成请求时发生冲突。 服务器必须在响应中包含有关冲突的信息。

410 (已删除) 如果请求的资源已永久删除，服务器就会返回此响应。

411 (需要有效长度) 服务器不接受不含有效内容长度标头字段的请求。

412 (未满足前提条件) 服务器未满足请求者在请求中设置的其中一个前提条件。

413 (请求实体过大) 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。

414 (请求的 URI 过长) 请求的 URI(通常为网址)过长，服务器无法处理。

415 (不支持的媒体类型) 请求的格式不受请求页面的支持。

416 (请求范围不符合要求) 如果页面无法提供请求的范围，则服务器会返回此状态代码。

417 (未满足期望值) 服务器未满足"期望"请求标头字段的要求。

500 (服务器内部错误) 服务器遇到错误，无法完成请求。

501 (尚未实施) 服务器不具备完成请求的功能。 例如，服务器无法识别请求方法时可能会返回此代码。

502 (错误网关) 服务器作为网关或代理，从上游服务器收到无效响应。

503 (服务不可用) 服务器目前无法使用(由于超载或停机维护)。 通常，这只是暂时状态。

504 (网关超时) 服务器作为网关或代理，但是没有及时从上游服务器收到请求。

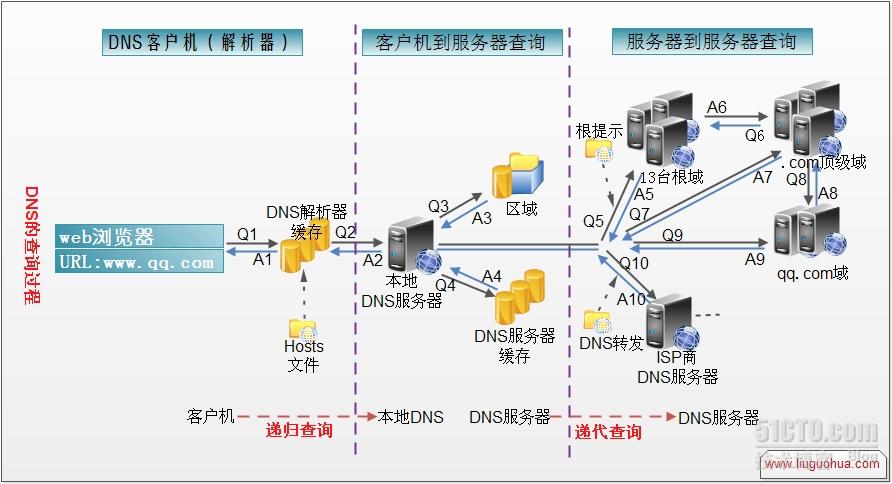
505 (HTTP 版本不受支持) 服务器不支持请求中所用的 HTTP 协议版本。

## 7.DNS解析过程

DNS( Domain Name System)是“域名系统”的英文缩写，是一种组织成域层次结构的计算机和网络服务命名系统，它用于TCP/IP网络，它用来将主机名和域名转换为IP地址。

DNS域名称  
    域名系统作为一个层次结构和分布式数据库，包含各种类型的数据，包括主机名和域名。DNS数据库中的名称形成一个分层树状结构称为域命名空间。  
[](http://blog.51cto.com/attachment/201203/171409287.jpg) 一些常见的DNS域名称如下图：  
[](http://blog.51cto.com/attachment/201203/171425985.jpg)

资源记录  
    DNS 数据库中包含的资源记录 (RR)。 每个 RR 标识数据库中的特定资源。我们在建立DNS服务器时，经常会用到SOA,NS,A之类的记录，在维护DNS服务器时，会用到MX，CNAME记录。   
常见的RR见下图：  
[](http://blog.51cto.com/attachment/201203/171440482.jpg)

DNS服务的工作过程  
[](http://blog.51cto.com/attachment/201203/175333937.jpg)

1、在浏览器中输入www.qq.com域名，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。

2、如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

3、如果hosts与本地DNS解析器缓存都没有相应的网址映射关系，首先会找TCP/IP参数中设置的首选DNS服务器，在此我们叫它本地DNS服务器，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。

4、如果要查询的域名，不由本地DNS服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。

5、如果本地DNS服务器本地区域文件与缓存解析都失效，则根据本地DNS服务器的设置（是否设置转发器）进行查询，如果未用转发模式，本地DNS就把请求发至全球13台根DNS，根DNS服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址(qq.com)给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找qq.com域服务器，重复上面的动作，进行查询，直至找到www.qq.com主机。

6、如果用的是转发模式，此DNS服务器就会把请求转发至上一级DNS服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地DNS服务器用是是转发，还是根提示，最后都是把结果返回给本地DNS服务器，由此DNS服务器再返回给客户机。   
  
    从客户端到本地DNS服务器是属于递归查询，而DNS服务器之间就是的交互查询就是迭代查询。

## 8.Cookie与Session

### （1）Cookies

 Cookies是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器。网络服务器用HTTP头向客户端发送Cookies，在客户终端，浏览器解析这些Cookies并将它们保存为一个本地文件，它会自动将同一服务器的任何请求缚上这些Cookies。

     Cookie机制采用的是在客户端保持状态的方案。它是在用户端的会话状态的存贮机制，他需要用户打开客户端的Cookie支持。Cookie的作用就是为了解决HTTP协议无状态的缺陷所作的努力。（HTTP无状态协议是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。）

     正统的Cookie分发是通过扩展HTTP协议来实现的，服务器通过在HTTP的响应头中加上一行特殊的指示以提示浏览器按照指示生成相应的Cookie。然而纯粹的客户端脚本如JavaScript也可以生成Cookie。而Cookie的使用是由浏览器按照一定的原则在后台自动发送给服务器的。浏览器检查所有存储的Cookie，如果某个Cookie所声明的作用范围大于等于将要请求的资源所在的位置，则把该Cookie附在请求资源的HTTP请求头上发送给服务器。

     Cookie的内容主要包括：名字，值，过期时间，路径和域。路径与域一起构成cookie的作用范围。若不设置过期时间，则表示这个Cookie的生命期为浏览器会话期间，关闭浏览器窗口，cookie就消失。这种生命期为浏览器会话期的Cookie被称为会话Cookie。会话Cookie一般不存储在硬盘上而是保存在内存里，当然这种行为并不是规范规定的。若设置了过期时间，浏览器就会把Cookie保存到硬盘上，关闭后再次打开浏览器，这些Cookie仍然有效直到超过设定的过期时间。存储在硬盘上的Cookie可以在不同的浏览器进程间共享，比如两个IE窗口。而对于保存在内存里的cookie，不同的浏览器有不同的处理方式。

     而Session机制采用的是一种在服务器端保持状态的解决方案。同时我们也看到，由于采用服务器端保持状态的方案在客户端也需要保存一个标识，所以Session机制可能需要借助于Cookie机制来达到保存标识的目的。而Session提供了方便管理全局变量的方式。

     Session是针对每一个用户的，变量的值保存在服务器上，用一个SessionID来区分是哪个用户Session变量,这个值是通过用户的浏览器在访问的时候返回给服务器，当客户禁用Cookie时，这个值也可能设置为由get来返回给服务器。

     就安全性来说：当你访问一个使用Session 的站点，同时在自己机子上建立一个Cookie，建议在服务器端的Session机制更安全些，因为它不会任意读取客户存储的信息。

### （2）Session机制

     Session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。

     当程序需要为某个客户端的请求创建一个Session时，服务器首先检查这个客户端的请求里是否已包含了一个Session标识（称为Session id），如果已包含则说明以前已经为此客户端创建过Session，服务器就按照Session id把这个session检索出来使用，如果客户端请求不包含Session id，则为此客户端创建一个Session并且生成一个与此Session相关联的Session id，Session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id将被在本次响应中返回给客户端

保存。

     保存这个Session id的方式可以采用Cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器。一般这个Cookie的名字都是类似于SEEESIONID。但Cookie可以被人为的禁止，则必须有其他机制以便在Cookie被禁止时仍然能够把Session id传递回服务器。

     经常使用的一种技术叫URL重写，就是把Session id直接附加在URL路径的后面。还有一种技术叫做表单隐藏字段。就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把Session id传递回服务器。

     Cookie与Session都能够进行会话跟踪，但是完成的原理不太一样。普通状况下二者均能够满足需求，但有时候不能够运用Cookie，有时候不能够运用Session。

### （3）Cookie与Session的区别：

     ①存取方式的不同（Cookie只能保存ASCII，Session可以存任意数据类型）

     Cookie中只能保管ASCII字符串，假如需求存取Unicode字符或者二进制数据，需求先进行编码。Cookie中也不能直接存取Java对象。若要存储略微复杂的信息，运用Cookie是比拟艰难的。

     而Session中能够存取任何类型的数据，包括而不限于String、Integer、List、Map等。Session中也能够直接保管Java Bean乃至任何Java类，对象等，运用起来十分便当。能够把Session看做是一个Java容器类。

     ②隐私策略的不同（Cookie存储在客户端阅读器中，Session存储在服务器上）

     Cookie存储在客户端阅读器中，对客户端是可见的，客户端的一些程序可能会窥探、复制以至修正Cookie中的内容。而Session存储在服务器上，对客户端是透明的，不存在敏感信息泄露的风险。

     假如选用Cookie，比较好的方法是，敏感的信息如账号密码等尽量不要写到Cookie中。最好是像Google、Baidu那样将Cookie信息加密，提交到服务器后再进行解密，保证Cookie中的信息只要本人能读得懂。而假如选择Session就省事多了，反正是放在服务器上，Session里任何隐私都能够有效的保护。

     ③有效期不同（Cookie可设长时间，Session关闭浏览器就失效）

     使用过Google的人都晓得，假如登录过Google，则Google的登录信息长期有效。用户不用每次访问都重新登录，Google会持久地记载该用户的登录信息。要到达这种效果，运用Cookie会是比较好的选择。只需要设置Cookie的过期时间属性为一个很大很大的数字。

     由于Session依赖于名为JSESSIONID的Cookie，而Cookie JSESSIONID的过期时间默许为–1，只需关闭了阅读器该Session就会失效，因而Session不能完成信息永世有效的效果。运用URL地址重写也不能完成。而且假如设置Session的超时时间过长，服务器累计的Session就会越多，越容易招致内存溢出。(这时使用Local Storage本地存储)

     ④服务器压力不同（Cookie压力小于Session）

     Session是保管在服务器端，每个用户都会产生一个Session。假如并发访问的用户十分多，会产生十分多的Session，耗费大量的内存。因而像Google、Baidu、Sina这样并发访问量极高的网站，是不太可能运用Session来追踪客户会话的。

     而Cookie保管在客户端，不占用服务器资源。假如并发阅读的用户十分多，Cookie是很好的选择。关于Google、Baidu、Sina来说，Cookie或许是唯一的选择。

     ⑤存储大小不同

     单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

     ⑤浏览器支持的不同

     Cookie是需要客户端浏览器支持的。假如客户端禁用了Cookie，或者不支持Cookie，则会话跟踪会失效。关于WAP上的应用，常规的Cookie就派不上用场了。

     假如客户端浏览器不支持Cookie，需要运用Session以及URL地址重写。需要注意的是一切的用到Session程序的URL都要进行URL地址重写，否则Session会话跟踪还会失效。关于WAP应用来说，Session+URL地址重写或许是它唯一的选择。

     假如客户端支持Cookie，则Cookie既能够设为本浏览器窗口以及子窗口内有效（把过期时间设为–1），也能够设为一切阅读器窗口内有效（把过期时间设为某个大于0的整数）。但Session只能在本阅读器窗口以及其子窗口内有效。假如两个浏览器窗口互不相干，它们将运用两个不同的Session。（IE8下不同窗口Session相干）

     ⑥跨域支持上的不同

     Cookie支持跨域名访问，例如将domain属性设置为“.biaodianfu.com”，则以“.biaodianfu.com”为后缀的一切域名均能够访问该Cookie。跨域名Cookie如今被普遍用在网络中，例如Google、Baidu、Sina等。而Session则不会支持跨域名访问。Session仅在他所在的域名内有效。

### （4）总结

     仅运用Cookie或者仅运用Session可能完成不了理想的效果。这时应该尝试一下同时运用Cookie与Session。Cookie与Session的搭配运用在实践项目中会完成很多意想不到的效果。

     将登陆信息等重要信息存放为Session，

     其他信息如果需要保留，可以放在Cookie中

## 9.get和 post 区别

GET使用URL或Cookie传参。而POST将数据放在BODY中。（错）

GET的URL会有长度上的限制，则POST的数据则可以非常大。（错）

POST比GET安全，因为数据在地址栏上不可见。（错）

GET和POST与数据如何传递没有关系

GET和POST是由HTTP协议定义的。在HTTP协议中，Method和Data（URL， Body， Header）是正交的两个概念，也就是说，使用哪个Method与应用层的数据如何传输是没有相互关系的。

HTTP没有要求，如果Method是POST数据就要放在BODY中。也没有要求，如果Method是GET，数据（参数）就一定要放在URL中而不能放在BODY中。

那么，网上流传甚广的这个说法是从何而来的呢？我在HTML标准中，找到了相似的描述。这和网上流传的说法一致。但是这只是HTML标准对HTTP协议的用法的约定。怎么能当成GET和POST的区别呢？

而且，现代的Web Server都是支持GET中包含BODY这样的请求。虽然这种请求不可能从浏览器发出，但是现在的Web Server又不是只给浏览器用，已经完全地超出了HTML服务器的范畴了。

HTTP协议对GET和POST都没有对长度的限制

HTTP协议明确地指出了，HTTP头和Body都没有长度的要求。而对于URL长度上的限制，有两方面的原因造成：

浏览器。据说早期的浏览器会对URL长度做限制。据说IE对URL长度会限制在2048个字符内（流传很广，而且无数同事都表示认同）。但我自己试了一下，我构造了90K的URL通过IE9访问live.com，是正常的。网上的东西，哪怕是Wikipedia上的，也不能信。

服务器。URL长了，对服务器处理也是一种负担。原本一个会话就没有多少数据，现在如果有人恶意地构造几个几M大小的URL，并不停地访问你的服务器。服务器的最大并发数显然会下降。另一种攻击方式是，把告诉服务器Content-Length是一个很大的数，然后只给服务器发一点儿数据，嘿嘿，服务器你就傻等着去吧。哪怕你有超时设置，这种故意的次次访问超时也能让服务器吃不了兜着走。有鉴于此，多数服务器出于安全啦、稳定啦方面的考虑，会给URL长度加限制。但是这个限制是针对所有HTTP请求的，与GET、POST没有关系。

安全不安全和GET、POST没有关系

## 10.7层 TCP、IP协议分别在哪层

第三层和第四层

## 11.TCP中time\_wait作用

主动关闭的Socket端会进入TIME\_WAIT状态，并且持续2MSL时间长度，MSL就是maximum segment lifetime(最大分节生命期），这是一个IP数据包能在互联网上生存的最长时间，超过这个时间将在网络中消失。MSL在RFC 1122上建议是2分钟，而源自berkeley的TCP实现传统上使用30秒，因而，TIME\_WAIT状态一般维持在1-4分钟。

TIME\_WAIT状态存在的理由：

（1）可靠地实现TCP全双工连接的终止

在进行关闭连接四路握手协议时，最后的ACK是由主动关闭端发出的，如果这个最终的ACK丢失，服务器将重发最终的FIN，因此客户端必须维护状态信息允许它重发最终的ACK。如果不维持这个状态信息，那么客户端将响应RST分节，服务器将此分节解释成一个错误（在java中会抛出connection reset的SocketException)。因而，要实现TCP全双工连接的正常终止，必须处理终止序列四个分节中任何一个分节的丢失情况，主动关闭 的客户端必须维持状态信息进入TIME\_WAIT状态。

（2）允许老的重复分节在网络中消逝

TCP分节可能由于路由器异常而“迷途”，在迷途期间，TCP发送端可能因确认超时而重发这个分节，迷途的分节在路由器修复后也会被送到最终目的地，这个原来的迷途分节就称为lost duplicate。在关闭一个TCP连接后，马上又重新建立起一个相同的IP地址和端口之间的TCP连接，后一个连接被称为前一个连接的化身 （incarnation)，那么有可能出现这种情况，前一个连接的迷途重复分组在前一个连接终止后出现，从而被误解成从属于新的化身。为了避免这个情况，TCP不允许处于TIME\_WAIT状态的连接启动一个新的化身，因为TIME\_WAIT状态持续2MSL，就可以保证当成功建立一个TCP连接的时候，来自连接先前化身的重复分组已经在网络中消逝。

## 12.TCP与UDP编程区别

1）TCP面向连接（三次握手机制），通信前需要先建立连接；UDP面向无连接，通信前不需要建立连接；

2）TCP保障可靠传输（按序、无差错、不丢失、不重复）；UDP不保障可靠传输，使用最大努力交付；

3）TCP面向字节流的传输，UDP面向数据报的传输。

## 13.两个文件，每个文件中有50亿URL，找出两个文件中相同的URL

Hash+hash

可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

    遍历文件a，对每个url求取hash(url)%1000，然后根据所取得的值将url分别存储到1000个小文件（记为a0,a1,...,a999）中。这样每个小文件的大约为300M。

    遍历文件b，采取和a相同的方式将url分别存储到1000小文件（记为b0,b1,...,b999）。这样处理后，所有可能相同的url都在对应的小文件（a0vsb0,a1vsb1,...,a999vsb999）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。

    求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash\_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。

## 14.tcp syn丢包，重连的时间

## 15.socket编程相关，如果服务器这边调用write写了100个字节的数据，客户端想要获得这个数据，是直接用read系统调用，参数也是100吗？

不是。对于read 调用，如果接收缓冲区中有 20字节，请求读 100个字节，就会返回 20；对于 write调用，如果请求写 100个字节，而发送缓冲区中只有 20个字节的空闲位置，那么 write会阻塞，直到把 100个字节全部交给发送缓冲区才返回。但如果 socket文件描述符有 O\_NONBLOCK标志，则 write不阻塞，直接返回 20；此时非阻塞地read 也直接返回20。

## 16.TCP/IP协议栈各个层次及分别的功能

## 17.Servlet 与 CGI 的比较：

和CGI程序一样，Servlet可以响应用户的指令(提交一个FORM等等)，也可以象CGI程序一样，收集用户表单的信息并给予动态反馈(简单的注册信息录入和检查错误)。  
 然而，Servlet的机制并不仅仅是这样简单的与用户表单进行交互。传统技术中，动态的网页建立和显示都是通过CGI来实现的，但是，有了Servlet,您可以大胆的放弃所有CGI(perl?php?甚至asp!)，利用Servlet代替CGI,进行程序编写。  
    对比一：当用户浏览器发出一个Http/CGI的请求，或者说 调用一个CGI程序的时候，服务器端就要新启用一个进程 (而且是每次都要调用)，调用CGI程序越多(特别是访问量高的时候)，就要消耗系统越多的处理时间，只剩下越来越少的系统资源，对于用户来说，只能是漫长的等待服务器端的返回页面了，这对于电子商务激烈发展的今天来说，不能不说是一种技术上的遗憾。  
而Servlet充分发挥了服务器端的资源并高效的利用。每次调用Servlet时并不是新启用一个进程 ，而是在一个Web服务器的进程敏感词享和分离线程，而线程最大的好处在于可以共享一个数据源，使系统资源被有效利用。  
    对比二：传统的CGI程序，不具备平台无关性特征，系统环境发生变化，CGI程序就要瘫痪，而Servlet具备Java的平台无关性，在系统开发过程中保持了系统的可扩展性、高效性。  
    对比三：传统技术中，一般大都为二层的系统架构，即Web服务器+数据库服务器，导致网站访问量大的时候，无法克服CGI程序与数据库建立连接时速度慢的瓶颈，从而死机、数据库死锁现象频繁发生。而我们的Servlet有连接池的概念，它可以利用多线程的优点，在系统缓存中事先建立好若干与数据库的连接，到时候若想和数据库打交道可以随时跟系统"要"一个连接即可，反应速度可想而知。

## 18. connect方法会阻塞，如何避免其长时间阻塞？

（1）考虑采用异步传输机制，同步传输与异步传输的主要区别在于同步传输中，如果调用recvfrom后会一致阻塞运行，从而导致调用线程暂停运行；异步传输机制则不然，会立即返回。

（2）当在一个非阻塞的TCP套接字上调用connect时，connect将立即返回一个EINPROGRESS错误，不过已经发起的TCP三路握手继续进行。然后用select检测连接是否成功。给select设置超时值，就可以避免长时间阻塞。

非阻塞connect有三个用途:

(1)在三路握手期间，可以做其他操作。

(2)同时建立多个连接。这个用途已随着Web浏览器流行起来。

(3)缩短connect的超时，避免长时间阻塞。

select+非阻塞connect：两个规则：

（1）当连接成功建立时，描述符可写

（2）当连接建立遇到错误时，描述符变为既可读又可写。

所以可以用getsockopt查看是否connect出错。

## 19. 网络中客户端突然掉线或者重启，服务器端怎么样才能立刻知道？

如果server用select，客户重启，进程结束时发送FIN，server可读，读取长度为0 。

如果客户端突然掉线，不会发送FIN，服务器需要主动发送数据才能知道。

## 20. 心跳

### 20.1当TCP两端A、B建立了连接后，如果一端拔掉网线或者拔掉电源，那么另一端能够收到通知吗？ 答案是不会。

原因：

TCP是一种有连接的协议，但是这个连接并不是指有一条实际的电路，而是一种虚拟的电路。TCP的建立连接和断开连接都是通过发送数据实现的，也就是我们常说的三次握手、四次挥手。TCP两端保存了一种数据的状态，就代表这种连接，TCP两端之间的路由设备只是将数据转发到目的地，并不知道这些数据实际代表了什么含义，也并没有在其中保存任何的状态信息，也就是说中间的路由设备没有什么连接的概念，只是将数据转发到目的地，只有数据的发送者和接受者两端真正的知道传输的数据代表着一条连接。

但是这就说明了一点，如果不发送数据那么是无法断开连接的。正常情况下当TCP的一端A调用了SOCKET的close或者进程结束，操作系统就会按照TCP协议发送FIN数据报文。B端收到后就会断开连接。但是当出现了上文所说的异常情况时：被拔掉网线或者断掉电源，总结起来就是没有机会发出断开的FIN数据报文。那么和A直连的路由设备虽然知道A设备已经断开了，但是路由设备并没有保存连接的状态信息，所以路由设备也就不可能去通知B端A端的断开。而B端没有收到断开的数据报文就会依然保持连接。所以A端拔掉网线或者断掉电源后B端是没办法收到断开连接的通知的。

解决方案：

保持连接并不是毫无代价的，如果这种异常断开的连接有很多，那么势必会耗费大量的资源，必须要想办法检测出这种异常连接。

检测的方法很简单，只要让B端主动通过这个连接向A端继续发送数据即可。上文说过，A端异常断开后，和A端直接连接的路由器是知道的。当B端发送的数据经过转发后到达这个路由器后，必然最终会返回B端一个目的不可达。此时B端立刻就会知道这条连接其实已经异常断开了。

但是B端不可能知道什么时候会出现这种异常，所以B端必须定时发送数据来检测连接是否异常断开。数据的内容无关紧要，任何数据都能达到这个效果。这个数据就是我们经常在TCP编程中所说的心跳。

### 20.2 KEEP\_ALIVE（TCP本身提供的保活机制）

TCP协议本身就提供了一种这样的机制来探测对端的存活。TCP协议有一个KEEP\_LIVE开关，只要打开这个开关就会定时发送一些数据长度为零的探测心跳包，发送的频率和次数都可以设置，具体的方法在网上搜索tcp keepalive即可，网上有很多文章，这里不再赘述。

### 20.3应用层心跳（主动发送心跳数据包）

区别：（1）应用层的心跳数据包会耗费更多的带宽，因为TCP协议的保活机制发送的是数据长度为零心跳包，而应用层的心跳数据包长度则必然会大于0。因此，由TCP协议提供的检活, 其发的探测包, 理论上实现的会更精妙(用更少的字节完成更多的目标), 耗费更少的流量.

（2）由应用自己实现的应用层的心跳, 为心跳消息额外定义一个消息类型就可以了. 就是应用正常的消息包, 只是这个包特殊点, 专门用来检活而已, 通常比较小。应用层的心跳数据包可以带一些应用所需要的数据，随应用自己控制，而TCP协议的保活机制则是对于应用层透明的，无法利用心跳携带数据。

应用层心跳的好处我个人的理解有两点:

一是比较灵活, 因为协议层的心跳只能提供最纯粹的检活功能, 但是应用层自己可以随意控制, 包括协议可能提供的是秒级的, 但是你想做成毫秒级的都任意(虽然实际几乎不会有这种时间级别的心跳), 包里还甚至可以携带额外的信息, 这些都是灵活之处.

二是通用, 应用层的心跳不依赖协议. 如果有一天不用TCP要改为UDP了, 协议层不提供心跳机制了, 但是你应用层的心跳依旧是通用的, 可能只需要做少许改动就可以继续使用.

应用层心跳的不好的地方也很显而易见, 增加开发工作量, 由于应用特定的网络框架, 还可能很增加代码结构的复杂度.

### 20.4 双向心跳

那么是否只是一端向另一端发送心跳就行了呢？显然不行。因为两端都有可能发生异常断开的情况。所以TCP连接的两端必须都向对端发送心跳。

## 21.TTL是什么？有什么用处，通常那些工具会用到它？（ping? traceroute? ifconfig? netstat?）

TTL（time to live）生存时间字段（8位，在IP首部）设置了ip数据报可以经过的最多路由器数。TTL的初始值由主机设置，一旦经过一个处理它的路由器，它的值减去1。当该字段的值为0时，数据报就被丢弃，并发送ICMP报文通知源主机。

TTL字段的用处：目的是防止数据报在选路时无休止地在网络中流动。（例如，当路由器瘫痪或者两个路由器之间连接丢失，选路协议有时会检测丢失的路由并一直进行下去。）

Traceroute会用到TTL，当路由器接收IP数据报中的TTL是1或者0，不转发，丢弃后给信源发ICMP“超时”信息。包含这份ICMP信息的IP报文的信源地址是该路由器的IP地址。当Traceroute发送TTL为1的IP数据报，得到第一个路由器的地址，发送TTL为2则得到第二个路由器地址，最后IP数据报到达目的主机时，由于Tracerout发送的UDP数据报中的端口选择了一个不可能的值，使得目的主机的UDP模块产生一份“端口不可达”错误的ICMP报文，Tracerout收到后结束。

Ping： TCP/IP协议，发送ICMP报文

Traceroute：查看路由路径

Ifconfig：用于配置和显示Linux内核中网络接口的网络参数。用ifconfig命令配置的网卡信息，在网卡重启后机器重启后，配置就不存在。要想将上述的配置信息永远的存的电脑里，那就要修改网卡的配置文件了。

Netstat：netstat命令用来打印Linux中网络系统的状态信息，可让你得知整个Linux系统的网络情况。

Arp：查看或修改IP到MAC转换表

## 22.路由表的作用？linux中怎么来配置一条默认路由？

路由表是用来决定如何将包从一个子网传送到另一个子网的。

在Linux上可以用 “route add default gw <默认路由器IP>”来配置一条默认路由。

route命令用来显示并设置Linux内核中的网络路由表，route命令设置的路由主要是静态路由。

要实现两个不同的子网之间的通信，需要一台连接两个网络的路由器，或者同时位于两个网络的网关来实现。

在Linux系统中设置路由通常是为了解决以下问题：该Linux系统在一个局域网中，局域网中有一个网关，能够让机器访问Internet，那么就需要将这台机器的ip地址设置为Linux机器的默认路由。要注意的是，直接在命令行下执行route命令来添加路由，不会永久保存，当网卡重启或者机器重启之后，该路由就失效了；可以在/etc/rc.local中添加route命令来保证该路由设置永久有效。

显示当前路由：route

添加网关/设置网关：

route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0 #增加一条到达244.0.0.0的路由。

屏蔽一条路由：

route add -net 224.0.0.0 netmask 240.0.0.0 reject #增加一条屏蔽的路由，目的地址为224.x.x.x将被拒绝。

删除路由记录：

route del -net 224.0.0.0 netmask 240.0.0.0

route del -net 224.0.0.0 netmask 240.0.0.0 reject

删除和添加设置默认网关：

route del default gw 192.168.120.240

route add default gw 192.168.120.240

## 23.在网络中有两台主机A和B，并通过路由器和其他交换设备连接起来，已经确认物理连接正确无误，怎么来测试这两台机器是否连通？如果不通，怎么来判断故障点？怎么排除故障？

从一台机器ping另一台机器，测试是否连通。

若ping不通，用traceroute可以确定是哪个路由器不能连通，然后再找问题是在交换设备、hup、cable等。

## 24.deamon进程如何实现？

关于deamon进程：deamon进程，也就是守护进程，是在UNIX后台运行不受终端控制的进程。守护进程常常在系统引导装入时启动，系统关闭时终止。由于没有控制终端，syslog函数是输出消息的标准方式。守护进程通过读取环境变量而获得配置信息。

守护进程的创建，首先调用fork，为避免挂起控制终端，将守护进程放入后台执行。调用setsid函数脱离控制终端，登录会话和进程组，使该进程成为无终端的会话组长。通过不让进程成为会话组长来禁止进程重新打开控制终端，这就需要第二次调用fork，父进程（会话组长）退出，子进程继续执行，并不再拥有打开控制终端的能力。

Fork()函数返回值：若成功调用一次则返回两个值，子进程返回0，父进程返回子进程标记；否则，出错返回-1。

[复制代码](javascript:void(0);)

// unistd.h syslog.h signal.h fcntl.h

int deamon\_init(const char \*pname,int facility)

{

int i;

int deamon\_proc;

pid\_t pid;

if((pid = fork())<0)//出错

return -1;

else if(pid)//父进程，结束父进程，子进程继续

exit(0);

if(setsid()<0)

return -1;

signal(SIGHUP,SIG\_IGN);

if((pid = fork())<0)

return -1;

else if(pid) //结束第一子进程，第二子进程继续（第二子进程不再是会话组长）

exit(0);

daemon\_proc = 1;

chdir("/"); //改变工作路径

//进程从创建它的父进程那里继承了打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。按如下方法关闭它们：

for(i=0;i<64;i++)

close(i);

open("/dev/null",O\_RDONLY);

open("/dev/null",O\_RDWR);

open("/dev/null",O\_RDWR);

openlog(pname,LOG\_PID,facility);

return 0;

}

[复制代码](javascript:void(0);)

if(setsid()<0) return -1; 当进程是会话组长时setsid()调用失败。但第一点已经保证进程不是会话组长。setsid()调用成功后，进程成为新的会话组长和新的进程组长，并与原来的登录会话和进程组脱离。由于会话过程对控制终端的独占性，进程同时与控制终端脱离。

处理SIGCHLD信号并不是必须的。但对于某些进程，特别是服务器进程往往在请求到来时生成子进程处理请求。如果父进程不等待子进程结 束，子进程将成为僵尸进程（zombie）从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在Linux下可以简单地将 SIGCHLD信号的操作设为SIG\_IGN。

signal(SIGCHLD,SIG\_IGN);

这样，内核在子进程结束时不会产生僵尸进程。这一点与BSD4不同，BSD4下必须显式等待子进程结束才能释放僵尸进程。

## 25.用UDP协议通讯时怎样得知目标机是否获得了数据包

可以在每个数据包中插入一个唯一的ID，比如timestamp或者递增的int。

发送方在发送数据时将此ID和发送时间记录在本地。

接收方在收到数据后将ID再发给发送方作为回应。

发送方如果收到回应，则知道接收方已经收到相应的数据包;如果在指定时间内没有收到回应，则数据包可能丢失，需要重复上面的过程重新发送一次，直到确定对方收到。

## 26.TCP为什么不是两次连接？而是三次握手？

如果A与B两个进程通信，如果仅是两次连接。可能出现的一种情况就是：A发送完请报文以后，由于网络情况不好，出现了网络拥塞，即B延时很长时间后收到报文，即此时A将此报文认定为失效的报文。B收到报文后，会向A发起连接。此时两次握手完毕，B会认为已经建立了连接可以通信，B会一直等到A发送的连接请求，而A对失效的报文回复自然不会处理。依次会陷入B忙等的僵局，造成资源的浪费。

## 27.网络编程中设计并发服务器，使用多进程与多线程，请问有什么区别？

答案一:

1，进程：子进程是父进程的复制品。子进程获得父进程数据空间、堆和栈的复制品。

2，线程：相对与进程而言，线程是一个更加接近与执行体的概念，它可以与同进程的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。两者都可以提高程序的并发度，提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

答案二:

根本区别就一点：用多进程每个进程有自己的地址空间(address space)，线程则共享地址空间。所有其它区别都是由此而来的：

1。速度：线程产生的速度快，线程间的通讯快、切换快等，因为他们在同一个地址空间内。

2。资源利用率：线程的资源利用率比较好也是因为他们在同一个地址空间内。

3。同步问题：线程使用公共变量/内存时需要使用同步机制还是因为他们在同一个地址空间内。

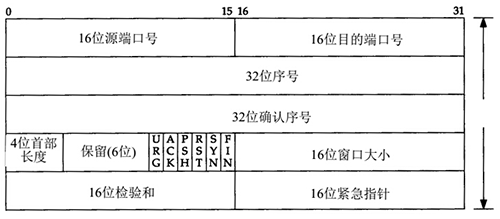
## 28.TIME\_WAIT状态的产生、危害、如何避免？

【答】TCP协议在关闭连接的四次挥手中，为了应对最后一个 ACK 丢失的情况，Client（即主动关闭连接的一方）需要维持 time\_wait 状态并停留 2 个MSL的时间。

危害：Linux分配给一个用户的文件句柄是有限的，如果系统中存在大量的 time\_wait 状态，一旦达到句柄数上限，新的请求就无法被处理了，而且大量 time\_wait 连接占用资源影响性能。

如何避免：在/etc/sysctl.conf文件中开启 net.ipv4.tcp\_tw\_reuse重用和net.ipv4.tcp\_tw\_recycle快速回收。

## 29.请画出TCP头部



## 30.TCP的拥塞控制机制是什么

我们知道TCP通过一个定时器（timer）采样了RTT（路的传播时间（propagation delay）、末端系统的处理时间、路由器缓存中的排队和处理时间）并计算RTO（超时重传时间），但是，如果网络上的延时突然增加，那么，TCP对这个事做出的应对只有重传数据，然而重传会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这就导致了恶性循环，最终形成“网络风暴” —— TCP的拥塞控制机制就是用于应对这种情况。

首先需要了解一个概念，为了在发送端调节所要发送的数据量，定义了一个“拥塞窗口”（Congestion Window），在发送数据时，将拥塞窗口的大小与接收端ack的窗口大小做比较，取较小者作为发送数据量的上限。

拥塞控制主要是四个算法：

慢启动：意思是刚刚加入网络的连接，一点一点地提速，不要一上来就把路占满。

连接建好的开始先初始化cwnd = 1，表明可以传一个MSS大小的数据。

每当收到一个ACK，cwnd++; 呈线性上升

每当过了一个RTT，cwnd = cwnd\*2; 呈指数让升

阈值ssthresh（slow start threshold），是一个上限，当cwnd >= ssthresh时，就会进入“拥塞避免算法”

拥塞避免：当拥塞窗口 cwnd 达到一个阈值时，窗口大小不再呈指数上升，而是以线性上升，避免增长过快导致网络拥塞。

每当收到一个ACK，cwnd = cwnd + 1/cwnd

每当过了一个RTT，cwnd = cwnd + 1

拥塞发生：当发生丢包进行数据包重传时，表示网络已经拥塞。分两种情况进行处理：

等到RTO超时，重传数据包

sshthresh = cwnd /2

cwnd 重置为 1

进入慢启动过程

在收到3个duplicate ACK时就开启重传，而不用等到RTO超时

sshthresh = cwnd = cwnd /2

进入快速恢复算法——Fast Recovery

快速恢复：至少收到了3个Duplicated Acks，说明网络也不那么糟糕，可以快速恢复。

cwnd = sshthresh + 3 \* MSS （3的意思是确认有3个数据包被收到了）

重传Duplicated ACKs指定的数据包

如果再收到 duplicated Acks，那么cwnd = cwnd +1

如果收到了新的Ack，那么，cwnd = sshthresh ，然后就进入了拥塞避免的算法了。

## 31.常用的TCP选项

1. .窗口扩大因子  
2. SACK选择确认选项  
3. .MSS（最大分段大小）

## 32.udp调用connect有什么作用

1.因为UDP可以是一对一，多对一，一对多，或者多对多的通信，所以每次调用sendto()/recvfrom()时都必须指定目标IP和端口号。通过调用connect()建立一个端到端的连接，就可以和TCP一样使用send()/recv()传递数据，而不需要每次都指定目标IP和端口号。但是它和TCP不同的是它没有三次握手的过程。

2.还可以通过在已建立连接的UDP套接字上，再次调用connect()实现以下功能：

a.指定新的IP地址和端口号。

b.断开连接。

这也与TCP有所不同，TCP套接字只能调用一次connect()函数。

## 33.socket可读或可写的情况

一、 下列四个条件中的任何一个满足时,socket准备好读:

1.socket接收缓冲区中已经接收的数据的字节数大于等于socket接收缓冲区低潮限度的当前值;对这样的socket的读操作不会阻塞,并返回一个大于0的值(即:准备好读入的数据的字节数).我们可以用socket选项SO\_RCVLOWAT来设置此低潮限度,对于TCP和UDPsocket,其缺省值为1;

2.连接的读这一半关闭(即:接收到对方发过来的FIN的TCP连接).对于这样的socket的读操作将不阻塞,并且返回0(即:文件结束符,FIN包体长度为0字节);

3.socket是一个用于监听的socket,并且已经完成的连接数为非0.这样的soocket处于可读状态,是因为socket收到了对方的connect请求,执行了三次握手的第一步:对方发送SYN请求过来,使监听socket处于可读状态;正常情况下,这样的socket上的accept操作不会阻塞;

4.有一个socket有异常错误条件待处理.对于这样的socket的读操作将不会阻塞,并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可通过指定socket选项SO\_ERROR调用getsockopt来取得并清除;

二、 下列三个条件中的任何一个满足时,socket准备好写 :

1.socket发送缓冲区中的可用空间字节数大于等于socket发送缓冲区低潮限度的当前值,且(i):socket已连接(TCP socket),或者(ii):socket不要求连接(如:UDP socket).这意味着,如果我们将这样的socket设置为非阻塞模式,写操作将不会阻塞,并且返回一个正值(如:由传输层接收的字节数).我们可以用socket选项SO\_SNDLOWAT来设置此低潮限度,对于TCP和UDP socket,其缺省值一般是2048Bytes;

2.连接的写这一半关闭.对于这样的socket的的写操作将产生信号SIGPIPE;

3.有一个socket异常错误条件待处理.对于这样的socket的写操作将不会阻塞并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可以通过指定socket选项SO\_ERROR调用getsockopt函数来取得并清除;

## UDP怎么模拟TCP可靠传输

## x86计算机和网络字节序分别是什么

## 4.写一个get请求包头

## 5.https的理解，http的各个方法的使用场景

## 6.滑动窗口协议、滑动窗口的大小是怎么分配的

# 操作系统

## 线程的同步/通信与进程的同步/通信有区别吗？

同步机制：

进程的同步：信号量、管程、互斥

线程的同步：信号量、互斥量、消息、条件变量

通信机制：

进程的通信：管道、FIFO、消息队列、信号量、共享内存、SOCKET、mmap。

教材上没有线程的通信机制这样的说法

## 1.进程和线程的基本概念

**进程(process)**：

狭义定义：进程就是一段程序的执行过程。

广义定义：进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。进程是系统进行资源分配和调度的一个独立单元。

1）进程是一个实体，每个进程都有自己的地址空间，一般情况下，包含文本区域、数据区域、堆栈

2）进程是执行中的程序，程序是一个没有生命的实体，只有处理器赋予程序生命时，他才能成为一个活动的实体，我们称之为进程

3）进程本身不会运行，是线程的容器。线程不能单独执行，必须组成进程

**进程状态：**

1）就绪：获取出CPU外的所有资源、只要处理器分配资源就可以马上执行

2）运行：获得处理器分配的资源，程序开始执行

3）阻塞：当程序条件不够的时候，需要等待提交满足的时候才能执行。

**线程**：

线程是进程的实体，是CPU调度和分派的基本单元。

1）在一个进程内部，要同时干多件事情，就需要同时运行多个子任务，我们把进程内的这些子任务叫做线程  
 2）多线程就是为了同步完成多项任务(在单个程序中同时运行多个线程完成不同的任务和工作)，不是为了提高运行效率，而是为了提高资源使用效率来提高系统的效率   
 3）线程是程序执行流的最小单元。一个标准的线程由当前的线程ID、当前指令指针、寄存器和堆栈组成

**线程状态**：

1）就绪：指线程具备运行的所有条件，逻辑上可以运行，在等待处理机

2）运行：指线程占用处理机正在运行

3）阻塞：线程在等待一个事件，逻辑上不可执行

**如果我们要同时执行多个任务怎么办？**

1）启动多个进程，每个进程虽然只有一个线程，但是多个进程可以一块执行多个任务

2）启动一个进程，在一个进程内启动多个线程，这样多个线程也可以一块执行多个任务

## 2.线程同步的方法（4种）

**（1）互斥量：**采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。互斥对象和临界区对象非常相似，只是其允许在进程间使用，而临界区只限制与同一进程的各个线程之间使用。

1.初始化互斥量

互斥量是一个pthread\_mutex\_t类型的变量。

1.1：用宏常量初始化：

1 pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;

1.2：用函数初始化：

1 #include <pthread.h>

2 int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

mutex：互斥量结构指针

attr：互斥量的属性结构指针

2.设置互斥量属性

1 #include <pthread.h>

2 int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int type);

attr：互斥量的属性结构指针

type：PTHREAD\_MUTEX\_NORMAL(默认属性)，PTHREAD\_MUTEX\_ERRORCHECK(会进行错误检查，速度比较慢)，PTHREAD\_MUTEX\_RECURSIVE(递归锁)。

对于递归锁，同一个线程对一个递归锁加锁多次，会有一个锁计数器，解锁的时候也需要解锁这个次数才能释放该互斥量。

3.加锁与解锁

1 #include <pthread.h>

2 int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

3 int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

4 int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

pthread\_mutex\_lock()得不到锁会阻塞

int pthread\_mutex\_trylock()得不到锁会立即返回，并返回EBUSY错误。

pthread\_mutex\_timedlock()会根据时间来等待加锁，如果这段时间得不到锁会返回ETIMEDOUT错误！

1 #include <pthread.h>

2 #include <time.h>

3 int pthread\_mutex\_timedlock(pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict abs\_timeout);

4.销毁互斥量

1 #include <pthread.h>

2 int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

**（2）信号量：**它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。

当需要一个计数器来限制可以使用某共享资源的线程数目时，可以使用“信号量”对象。CSemaphore类对象保存了对当前访问某一个指定资源的线程的计数值，该计数值是当前还可以使用该资源的线程数目。如果这个计数达到了零，则所有对这个CSemaphore类对象所控制的资源的访问尝试都被放入到一个队列中等待，直到超时或计数值不为零为止。

CSemaphore 类的构造函数原型及参数说明如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CSemaphore(     LONG lInitialCount = 1,     LONG lMaxCount = 1,     LPCTSTR pstrName = NULL,     LPSECURITY\_ATTRIBUTES lpsaAttributes = NULL  ); |

* lInitialCount:信号量对象的初始计数值，即可访问线程数目的初始值；
* lMaxCount：信号量对象计数值的最大值，该参数决定了同一时刻可访问由信号量保护的资源的线程最大数目；
* 后两个参数在同一进程中使用一般为NULL，不作过多讨论；

一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就减1，只要当前可用资源计数大于0，就可以发出信号量信号。如果为0，则放入一个队列中等待。线程在处理完共享资源后，应在离开的同时通过ReleaseSemaphore（）函数将当前可用资源数加1。

|  |  |
| --- | --- |
| 1  2  3 | BOOL ReleaseSemaphore(  HANDLE hSemaphore,       // hSemaphore:信号量句柄  　　　　　　　　　　　　　　LONG lReleaseCount,      // lReleaseCount：信号量计数值  　　　　　　　　　　　　　　LPLONG lpPreviousCount   // 参数一般为NULL); |

**（3）消息**

允许一个线程在处理完一个任务后，主动唤醒另外一个线程执行任务。通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作。

**（4）条件变量(cond)**

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

条件变量分为两部分: 条件和变量。条件本身是由互斥量保护的。线程在改变条件状态前先要锁住互斥量。条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待”条件变量的条件成立”而挂起；另一个线程使”条件成立”（给出条件成立信号）。条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。   
1. 初始化条件变量。   
静态初始化，pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIER;   
动态初始化，int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr);   
2. 等待条件成立。释放锁,同时阻塞等待条件变量为真才行。timewait()设置等待时间,仍未signal,返回ETIMEOUT(加锁保证只有一个线程wait)

int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);

int pthread\_cond\_timewait(pthread\_cond\_t \*cond,pthread\_mutex \*mutex,const timespec \*abstime);

激活条件变量。pthread\_cond\_signal,pthread\_cond\_broadcast（激活所有等待线程）

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond); //解除所有线程的阻塞

1. 清除条件变量。无线程等待,否则返回EBUSY

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

## 3.进程同步的方法

**一、信号量**

理解PV：   
    P(S):①S=S-1；  
           ②如果S>0，则该进程继续执行；否则该进程置为等待状态，排入等待队列。  
    V(S):①S=S+1；  
           ②如果S>0，则该进程继续执行；否则释放队列中第一个等待信号量的进程。

信号量（semaphore）的数据结构为一个值和一个指针，指针指向等待该信号量的下一个进程。  
     一般来说，信号量S>0时，S表示可用资源的数量。执行一次P操作意味着请求分配一个单位资源，因此S的值减1；当S<0时，表示已经没有可用资源，请求者必须等待别的进程释放该类资源，它才能运行下去。而执行一个V操作意味着释放一个单位资源，因此S的值加1；若S=0，表示有某些进程正在等待该资源，因此要唤醒一个等待状态的进程，使之运行下去。

利用信号量和PV操作实现进程互斥的一般模型是：  
进程P1              进程P2           ……          进程Pn  
……                  ……                           ……  
P（S）；              P（S）；                         P（S）；  
临界区；             临界区；                        临界区；  
V（S）；              V（S）；                        V（S）；  
……                  ……            ……           ……

    其中信号量S用于互斥，初值为1。

利用信号量和PV操作实现进程同步

    使用PV操作实现进程同步时应该注意的是：

    （1）分析进程间的制约关系，确定信号量种类。在保持进程间有正确的同步关系情况下，哪个进程先执行，哪些进程后执行，彼此间通过什么资源（信号量）进行协调，从而明确要设置哪些信号量。  
    （2）信号量的初值与相应资源的数量有关，也与P、V操作在程序代码中出现的位置有关。  
    （3）同一信号量的P、V操作要成对出现，但它们分别在不同的进程代码中。

【例1】生产者-消费者问题

（1）一个生产者，一个消费者，公用一个缓冲区。  
定义两个同步信号量：  
empty——表示缓冲区是否为空，初值为1。  
   full——表示缓冲区中是否为满，初值为0。  
生产者进程  
while(TRUE){  
生产一个产品;  
     P(empty);  
     产品送往Buffer;  
     V(full);  
}  
消费者进程  
while(True){  
P(full);  
   从Buffer取出一个产品;  
   V(empty);  
   消费该产品;  
   }

（2）一组生产者，一组消费者，公用n个环形缓冲区  
    在这个问题中，不仅生产者与消费者之间要同步，而且各个生产者之间、各个消费者之间还必须互斥地访问缓冲区。  
定义四个信号量：  
empty——表示缓冲区是否为空，初值为n。  
full——表示缓冲区中是否为满，初值为0。  
mutex1——生产者之间的互斥信号量，初值为1。  
mutex2——消费者之间的互斥信号量，初值为1。

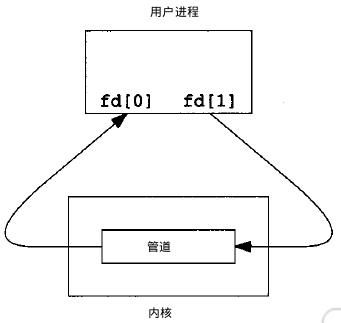
设缓冲区的编号为1～n-1，定义两个指针in和out，分别是生产者进程和消费者进程使用的指针，指向下一个可用的缓冲区。  
生产者进程  
while(TRUE){  
     生产一个产品;  
     P(empty);  
     P(mutex1)；  
     产品送往buffer（in）；  
     in=(in+1)mod n；  
     V(mutex1);  
     V(full);  
}  
消费者进程  
while(TRUE){  
 P(full)  
   P(mutex2)；  
   从buffer（out）中取出产品；  
   out=(out+1)mod n；  
   V（mutex2）；  
   V(empty);  
   消费该产品;

## 4.进程间通信的方法

主要分为：管道（pipe）、命名管道(FIFO)、消息队列、共享内存（传统的与文件映射（mmap））、SOCKET、

（1）普通管道是一种半双工的通信方式，数据只能单项流动，并且只能在具有亲缘关系的进程间流动，进程的亲缘关系通常是父子进程。命名管道也是半双工的通信方式，它允许无亲缘关系的进程间进行通信

当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。如下图：



（2）消息队列是消息的链表，存放在内核中，一个消息队列由一个标识符（即队列ID）来标识。

消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

复制代码

1 #include <sys/msg.h>

2 // 创建或打开消息队列：成功返回队列ID，失败返回-1

3 int msgget(key\_t key, int flag);

4 // 添加消息：成功返回0，失败返回-1

5 int msgsnd(int msqid, const void \*ptr, size\_t size, int flag);

6 // 读取消息：成功返回消息数据的长度，失败返回-1

7 int msgrcv(int msqid, void \*ptr, size\_t size, long type,int flag);

8 // 控制消息队列：成功返回0，失败返回-1

9 int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

复制代码

在以下两种情况下，msgget将创建一个新的消息队列：

* 如果没有与键值key相对应的消息队列，并且flag中包含了IPC\_CREAT标志位。
* key参数为IPC\_PRIVATE。

函数msgrcv在读取消息队列时，type参数有下面几种情况：

* type == 0，返回队列中的第一个消息；
* type > 0，返回队列中消息类型为 type 的第一个消息；
* type < 0，返回队列中消息类型值小于或等于 type 绝对值的消息，如果有多个，则取类型值最小的消息。

可以看出，type值非 0 时用于以非先进先出次序读消息。也可以把 type 看做优先级的权值。

（3）信号用于通知接收进程某个事件已经发生。

a、这种通信可携带的信息极少。不适合需要经常携带数据的通信。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

b、不具备同步机制，类似于中断，什么时候产生信号，进程是不知道的。

1 #include <sys/sem.h>

2 // 创建或获取一个信号量组：若成功返回信号量集ID，失败返回-1

3 int semget(key\_t key, int num\_sems, int sem\_flags);

4 // 对信号量组进行操作，改变信号量的值：成功返回0，失败返回-1

5 int semop(int semid, struct sembuf semoparray[], size\_t numops);

6 // 控制信号量的相关信息

7 int semctl(int semid, int sem\_num, int cmd, ...);

（4）共享内存就是映射一段能被其它进程访问的内存，这段共享内存由一个进程创建，但是多个进程可以访问。

a、最快的一种通信方式，多个进程可同时访问同一片内存空间，相对其他方式来说具有更少的数据拷贝，效率较高。

b、需要结合信号或其他方式来实现多个进程间同步，自身不具备同步机制。

c、随内核持续，相比于随进程持续生命力更强。

（5）SOCKET通信：

a、实现起来简单，可以使用因特网域和UNIX域来实现，使用因特网域可以实现不同主机之间的进出通信。

b、该方式自身携带同步机制，不需要额外的方式来辅助实现同步。

c、随进程持续。

## 5.伙伴系统 SLAB算法

伙伴系统，其思想是：把内存块分成不同的组(1,2,4,8,16,32....)；分配内存时找到能够满足条件 的最小的块；如果找不到，就找大的块，然后一分为 2，分配一块，留一块；回收时：如果有相邻的同样大小的块，则合并。

Malloc 实现原理：可以基于伙伴系统实现，也可以使用基于链表的实现

• 将所有空闲内存块连成链表，每个节点记录空闲内存块的地址、大小等信息

• 分配内存时，找到大小合适的块，切成两份，一分给用户，一份放回空闲链表

•free 时，直接把内存块返回链表

解决外部碎片：将能够合并的内存块进行合并

## 6.什么是缓冲区溢出？有什么危害？其原因是什么？

缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖在合法数据上。

危害有以下两点：

程序崩溃，导致拒绝额服务

跳转并且执行一段恶意代码

造成缓冲区溢出的主要原因是程序中没有仔细检查用户输入。缓冲区攻击的最终目的就是希望系统能执行这块可读写内存中已经被蓄意设定好的恶意代码。

## 7.死锁、死锁 产生的条件、怎么处理死锁、怎么预防死锁

**死锁：**在两个或者多个并发进程中，如果每个进程持有某种资源而又等待其它进程释放它或它们现在保持着的资源，在未改变这种状态之前都不能向前推进，称这一组进程产生了死锁。通俗的讲就是两个或多个进程无限期的阻塞、相互等待的一种状态。

**产生死锁的原因主要：**  
（1） 因为系统资源不足。  
（2） 进程运行推进的顺序不合适。  
（3） 资源分配不当等。

**死锁的必要条件：**

（1）互斥条件：一个资源一次只能被一个进程使用

（2）请求与保持条件：一个进程因请求资源而阻塞时，对已获得资源保持不放

（3）不剥夺条件：进程获得的资源，在未完全使用完之前，不能强行剥夺

（4）循环等待条件：若干进程之间形成一种头尾相接的环形等待资源关系

**解决死锁的方法：分为死锁的预防，避免，检测与恢复三种，当然还一种是忽略该问题，**例如鸵鸟算法，该算法可以应用在极少发生死锁的情况下。

**（1）死锁的预防（静态）**

  死锁的预防是保证系统不进入死锁状态的一种策略。它的基本思想是要求进程申请资源时遵循某种协议，从而打破产生死锁的四个必要条件中的一个或几个，保证系统不会进入死锁状态。

**〈1〉破坏互斥条件。**即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。

**〈2〉破坏不可剥夺条件。**即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这种预防死锁的方法实现起来困难，会降低系统性能。

**〈3〉破坏请求与保持条件。**可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。但是，这种策略也有如下缺点：

（1）在许多情况下，一个进程在执行之前不可能知道它所需要的全部资源。这是由于进程在执行时是动态的，不可预测的；

（2）资源利用率低。无论所分资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况。这显然是一种极大的资源浪费；

（3）降低了进程的并发性。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就必然少了。

**< 4 >破坏循环等待条件，实行资源有序分配策略。**采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但是也存在以下缺点：

（1）限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并增加了系统开销；

（2）为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。

**（2）死锁的避免（动态）**

**死锁的避免，它不限制进程有关申请资源的命令，而是对进程所发出的每一个申请资源命令加以动态地检查，并根据检查结果决定是否进行资源分配。**

**<1>安全序列**

  我们首先引入安全序列的定义：所谓系统是安全的，**是指系统中的所有进程能够按照某一种次序分配资源，并且依次地运行完毕，**这种进程序列{P1，P2，...，Pn}就是安全序列。

虽然存在安全序列时一定不会有死锁发生，但是系统进入不安全状态（四个死锁的必要条件同时发生）也未必会产生死锁。当然，产生死锁后，系统一定处于不安全状态。

**<2>银行家算法**Dijstra

 优点：

银行家算法允许死锁必要条件中的互斥条件，占有且申请条件，不可抢占条件的存在，这样，它与预防死锁的几种方法相比较，限制条件少了，资源利用程度提高了。

 缺点：

   〈1〉这个算法要求客户数保持固定不变，这在多道程序系统中是难以做到的。

   〈2〉这个算法保证所有客户在有限的时间内得到满足，但实时客户要求快速响应，所以要考虑这个因素。

   〈3〉由于要寻找一个安全序列，实际上增加了系统的开销

**（3）死锁检测与恢复（实际中使用）**

常利用资源分配图、进程等待图来协助这种检测。系统设有专门的机构，当死锁发生时，该机构能够检测到死锁发生的位置和原因，并能通过外力破坏死锁发生的必要条件，从而使得并发进程从死锁状态中恢复出来。

    （1）最简单，最常用的方法就是进行系统的重新启动，不过这种方法代价很大。

    （2）撤消进程，剥夺资源。终止参与死锁的进程，收回它们占有的资源，从而解除死锁。这时又分两种情况：一次性撤消参与死锁的全部进程，剥夺全部资源；或者逐步撤消参与死锁的进程，逐步收回死锁进程占有的资源。一般来说，选择逐步撤消的进程时要按照一定的原则进行，目的是撤消那些代价最小的进程，比如按进程的优先级确定进程的代价；考虑进程运行时的代价和与此进程相关的外部作业的代价等因素。

（3）进程回退策略，即让参与死锁的进程回退到没有发生死锁前某一点处，并由此点处继续执行，以求再次执行时不再发生死锁。虽然这是个较理想的办法，但是操作起来系统开销极大，要有堆栈这样的机构记录进程的每一步变化，以便今后的回退，有时这是无法做到的。

## 8.分页和分段有什么区别？

分段：将用户程序地址空间分成若干个**大小不等**的段，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现了离散分配。

分页：用户程序的地址空间被划分成若干固定大小的区域，称为“页”，相应地，内存空间分成若干个物理块，页和块的大小相等。可将用户程序的任一页放在内存的任一块中，实现了离散分配。

(1)页是信息的物理单位,分页是为了实现非连续分配,以便解决内存碎片问题,或者说分页是由于系统管理的需要。段是信息的逻辑单位,它含有一组意义相对完整的信息,分段的目的是为了更好地实现存储保护和信息的共享，满足用户的需要。

(2)页的大小固定,由系统确定,将逻辑地址划分为页号和页内地址是由机器硬件实现的.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.分段的地址空间是二维的.

## 9.进程调度算法，有哪些算法比较难实现？

一、先来先服务调度算法和短作业(进程)优先调度算法

1．先来先服务调度算法

2．短作业(进程)优先调度算法

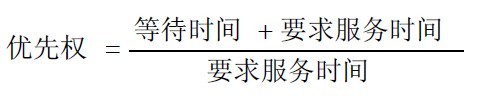
二、高优先权优先调度算法

1．优先权调度算法的类型：批处理系统、实时系统中。

1) 非抢占式优先权算法：主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2) 抢占式优先权调度算法：常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

2．高响应比优先调度算法



三、基于时间片的轮转调度算法

1．时间片轮转法

2．多级反馈队列调度算法（较好）

调度算法的实施过程如下所述。

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。

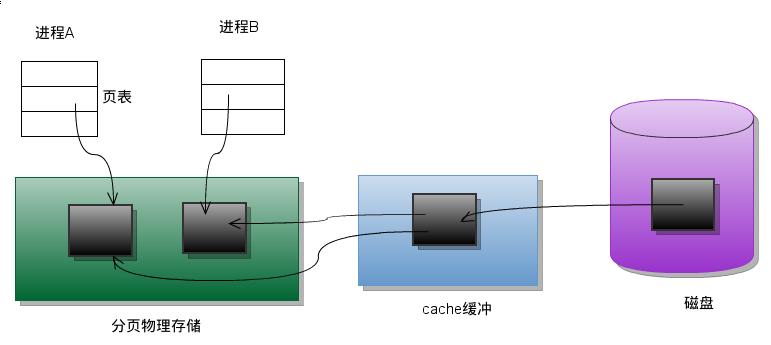
(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n 队列便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

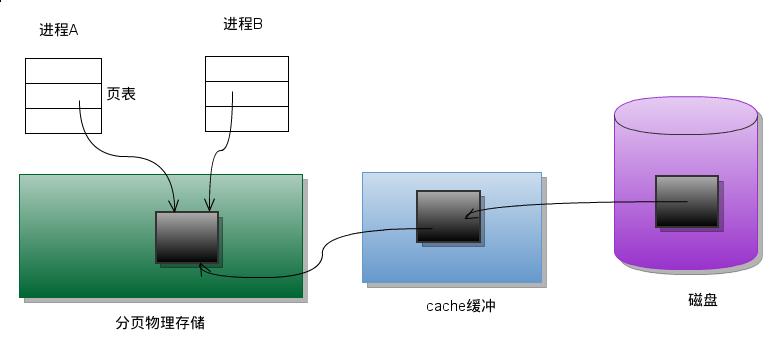
## 10.使用 mmap 读写文件为什么比普通读写函数要快？

像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据: 一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，共享内存中的内容往往是在解除映射时才写回文件的。

UNIX访问文件的传统方法是用open打开它们, 如果有多个进程访问同一个文件, 则每一个进程在自己的地址空间都包含有该文件的副本,这不必要地浪费了存储空间. 下图说明了两个进程同时读一个文件的同一页的情形. 系统要将该页从磁盘读到高速缓冲区中, 每个进程再执行一个存储器内的复制操作将数据从高速缓冲区读到自己的地址空间.



**共享存储映射**：进程A和进程B都将该页映射到自己的地址空间, 当进程A第一次访问该页中的数据时, 它生成一个缺页中断. 内核此时读入这一页到内存并更新页表使之指向它.以后, 当进程B访问同一页面而出现缺页中断时, 该页已经在内存, 内核只需要将进程B的页表登记项指向此页即可. 如下图所示:



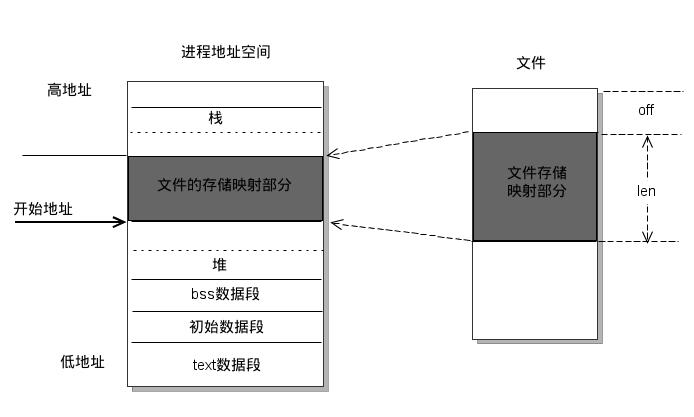
**mmap()及其相关系统调用**

mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用read()，write（）等操作。

 mmap()系统调用形式如下：

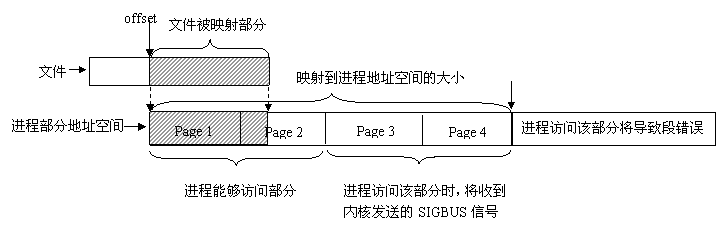
void\* mmap ( void \* addr , size\_t len , int prot , int flags , int fd , off\_t offset )

mmap的作用是映射文件描述符fd指定文件的 [off,off + len]区域至调用进程的[addr, addr + len]的内存区域, 如下图所示:



**对mmap()返回地址的访问**

linux采用的是页式管理机制。对于用mmap()映射普通文件来说，进程会在自己的地址空间新增一块空间，空间大小由mmap()的len参数指定，注意，进程并不一定能够对全部新增空间都能进行有效访问。进程能够访问的有效地址大小取决于文件被映射部分的大小。简单的说，能够容纳文件被映射部分大小的最少页面个数决定了进程从mmap()返回的地址开始，能够有效访问的地址空间大小。超过这个空间大小，内核会根据超过的严重程度返回发送不同的信号给进程。可用如下图示说明：



总结一下就是, 文件大小, mmap的参数 len 都不能决定进程能访问的大小, 而是容纳文件被映射部分的最小页面数决定进程能访问的大小。

## 11.缓存处理的算法有哪些，讲一讲LRU算法

常用的页面置换算法：

（1）最常不使用法：LFU（踢掉频率最小的）

（2）最近最少使用置换法（LRU）:当需要置换一页时，选择在最近一段时间里最久没有使用过的页面予以淘汰

浏览器就是使用了LRU作为缓存算法。

最近最少使用 twice（LRU2）：把被两次访问过的对象放入缓存池，当缓存池满了之后，我会把有两次最少使用的缓存对象踢走。因为需要跟踪对象2次，访问负载就会随着缓存池的增加而增加。

Two Queues（2Q）：把被访问的数据放到 LRU 的缓存中，如果这个对象再一次被访问，我就把他转移到第二个、更大的 LRU 缓存。踢走缓存对象是为了保持第一个缓存池是第二个缓存池的1/3。

（3）Adaptive Replacement Cache（ARC）（性能最好，可自调，低负载）：

由2个 LRU 组成，第一个，也就是 L1，包含的条目是最近只被使用过一次的，而第二个 LRU，也就是 L2，包含的是最近被使用过两次的条目。

（4）最近最常使用置换Most Recently Used（MRU）：

常用于数据库内存缓存中，每当一次缓存记录的使用，我会把它放到栈的顶端。当栈满了的时候，把栈顶的对象给换成新进来的对象！

（5）最近未使用置换法（NUR）：是LRU算法的近似方法，选择在最近一段时间里未被访问过的页面予以淘汰

（6）先进先出法：（置换次数比较多）

（7）Second Chance：像FIFO 一样在观察队列的前端，但是很FIFO的立刻踢出不同，我会检查即将要被踢出的对象有没有之前被使用过的标志（1一个 bit 表示），没有被使用过，我就把他踢出；否则，我会把这个标志位清除，然后把这个缓存对象当做新增缓存对象加入队列。

（8）CLock：有一个装有缓存对象的环形列表，头指针指向列表中最老的缓存对象。当缓存 miss 发生并且没有新的缓存空间时，我会问问指针指向的缓存对象的标志位去决定我应该怎么做。如果标志是0，我会直接用新的缓存对象替代这个缓存对象；如果标志位是1，我会把头指针递增，然后重复这个过程，知道新的缓存对象能够被放入。

（9）最佳置换法（OPT）：选择将来不再使用或在最远的将来才被访问的页调换出去（不便于实现）

## 12.进程阻塞状态是怎么引起的

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

阻塞原语的执行过程是：

找到将要被阻塞进程的标识号对应的PCB。

若该进程为运行状态，则保护其现场，将其状态转为阻塞状态，停止运行。

把该PCB插入到相应事件的等待队列中去。

当被阻塞进程所期待的事件出现时，如它所启动的I/O操作已完成或其所期待的数据已到达，则由有关进程（比如，提供数据的进程）调用唤醒原语(Wakeup)，将等待该事件的进程唤醒。

唤醒原语的执行过程是：

在该事件的等待队列中找到相应进程的PCB。

将其从等待队列中移出，并置其状态为就绪状态。

把该PCB插入就绪队列中，等待调度程序调度。

## 13.wait函数和waitpid函数

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid, int \*status, int options);

wait()会暂时停止目前进程的执行, 直到有信号来到或子进程结束. 如果在调用wait()时子进程已经结束, 则wait()会立即返回子进程结束状态值. 而子进程的进程识别码也会一快返回.

wait等待第一个终止的子进程，而waitpid可以通过pid参数指定等待哪一个子进程。当pid=-1、option=0时，waitpid函数等同于wait。

waitpid函数提供了wait函数没有提供的三个功能：

1、waitpid等待一个特定的进程，而wait则返回任一终止子进程的状态 。

2、waitpid提供了一个 wait的非阻塞版本，有时希望取得一个子进程的状态， 但不想进程阻塞。

3、waitpid支持作业控制。

## 14.缓存memrycache、缓存的作用

缓存的三大作用，就是预读取（预先读取将要载入的数据）、存储临时访问过的数据和对写入的数据进行暂时存放。

让缓存夹在CPU与内存之间，CPU缓存的存在，主要是为了解决内存的“慢速”与CPU的"快速"之间的矛盾。

## 15.线程、线程池使用

线程池技术把线程的创建和销毁分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段(在应用程序启动之后，就马上创建一定数量的线程，放入空闲的队列中。这些线程都是处于阻塞状态，这些线程只占一点内存，不占用CPU。

当任务到来后，线程池将选择一个空闲的线程，将任务传入此线程中运行。当所有的线程都处在处理任务的时候，线程池将自动创建一定的数量的新线程，用于处理更多的任务。执行任务完成之后线程并不退出，而是继续在线程池中等待下一次任务。当大部分线程处于阻塞状态时，线程池将自动销毁一部分的线程，回收系统资源)。

## 16. 用户态和内核态

\*特权指令：在内核态运行的指令，对内存空间的访问范围基本不受限制，不仅能访问用户存储空间也能访问系统存储空间。特权指令只允许OS使用。

\*非特权指令：在用户态运行的指令，只能完成一般性的操作和任务，不能对系统中的硬件和软件直接进行访问，其对内存的访问范围也局限于用户空间。

系统调用与过程调用的区别：

\*运行在不同系统状态：一般过程调用，其调用程序和被调用进程都运行在相同的状态(内核态或用户态)；而系统调用与一般过程调用的最大区别：调用程序是运行在用户态，而被调用程序是运行在系统态。

\*嵌套调用：系统调用的嵌套深度有限制，而一般过程调用对嵌套无限制。

用户态切换到内核态的3种方式

(1)系统调用 这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作。

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

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

线程的实现可以分为两类：用户级线程(User-Level Thread)和内核级线程(Kernel-Level Thread)

用户级线程（必须借助于某种形式的中间系统的帮助获得内核的服务）指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。不需要用户态/核心态切换，速度快，操作系统内核不知道多线程的存在，因此一个线程阻塞将使得整个进程（包括它的所有线程）阻塞。由于这里的处理器时间片分配是以进程为基本单位，所以每个线程执行的时间相对减少。

内核支持线程：由操作系统内核创建和撤销。内核维护进程及线程的上下文信息以及线程切换。一个内核线程由于I/O操作而阻塞，不会影响其它线程的运行。可以是用户进程中的线程或者系统的线程。

用户线程运行在一个中间系统上面。目前中间系统实现的方式有两种，即运行时系统（Runtime System）和内核控制线程。“运行时系统”实质上是用于管理和控制线程的函数集合，包括创建、撤销、线程的同步和通信的函数以及调度的函数。这些函数都驻留在用户空间作为用户线程和内核之间的接口。用户线程不能使用系统调用，而是当线程需要系统资源时，将请求传送给运行时，由后者通过相应的系统调用来获取系统资源。内核控制线程：系统在分给进程几个轻型进程（LWP），LWP可以通过系统调用来获得内核提供的服务，而进程中的用户线程可通过复用来关联到LWP，从而得到内核的服务。

以下是用户级线程和内核级线程的区别：

（1）内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的。

（2）用户级线程的创建、撤消和调度不需要OS内核的支持，是在语言（如Java）这一级处理的；而内核支持线程的创建、撤消和调度都需OS内核提供支持，而且与进程的创建、撤消和调度大体是相同的。

（3）用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。

（4）在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内核支持线程的系统内，CPU调度则以线程为单位，由OS的线程调度程序负责线程的调度。

（5）用户级线程的程序实体是运行在用户态下的程序，而内核支持线程的程序实体则是可以运行在任何状态下的程序。

内核线程的优点：

（1）当有多个处理机时，一个进程的多个线程可以同时执行。

（2）如果进程中的一个线程被阻塞，内核可以调度该进程的其他线程或者其他进程中的线程运行。

缺点：

（1）由内核进行调度。

（2）对于用户的线程而言，其模式切换的开销较大，在同一进程中，从一个线程切换到另一个线程时，需要从用户态转到内核态进行，这是因为用户进程的线程在用户态进行，而线程调度和管理是在内核实现的，系统开销较大。

用户线程的优点：

（1） 线程的调度不需要内核直接参与，控制简单。

（2） 可以在不支持线程的操作系统中实现。

（3） 创建和销毁线程、线程切换代价等线程管理的代价比内核线程少得多。

（4） 允许每个进程定制自己的调度算法，线程管理比较灵活。

（5） 线程能够利用的表空间和堆栈空间比内核级线程多。

缺点：

（1）资源调度按照进程进行，多个处理机下，同一个进程中的线程只能在同一个处理机下分时复用，多线程不能利用多处理器多重处理的优点。

（2） 同一进程中只能同时有一个线程在运行，如果有一个线程使用了系统调用而阻塞，那么整个进程都会被挂起。另外，页面失效也会产生同样的问题。

## 17.inode节点，文件名是否存放在inode节点里，

inode包含文件的元信息，具体来说有以下内容：

　　\* 文件的字节数

　　\* 文件拥有者的User ID

　　\* 文件的Group ID

　　\* 文件的读、写、执行权限

　　\* 文件的时间戳，共有三个：ctime指inode上一次变动的时间，mtime指文件内容上一次变动的时间，atime指文件上一次打开的时间。

　　\* 链接数，即有多少文件名指向这个inode

　　\* 文件数据block的位置

可以用stat命令，查看某个文件的inode信息：

stat example.txt

查看每个硬盘分区的inode总数和已经使用的数量，可以使用df命令。

df -i

查看每个inode节点的大小，可以用如下命令：

sudo dumpe2fs -h /dev/hda | grep "Inode size"

使用ls -i命令，可以看到文件名对应的inode号码：

ls -i example.txt

每个inode都有一个号码，操作系统用inode号码来识别不同的文件。Unix/Linux系统内部不使用文件名，而使用inode号码来识别文件。对于系统来说，文件名只是inode号码便于识别的别称或者绰号。

硬链接

Unix/Linux系统允许，多个文件名指向同一个inode号码。这意味着，可以用不同的文件名访问同样的内容；对文件内容进行修改，会影响到所有文件名；但是，删除一个文件名，不影响另一个文件名的访问。这种情况就被称为"硬链接"（hard link）。

ln命令可以创建硬链接：

ln 源文件 目标文件

目录文件的"链接数"。创建目录时，默认会生成两个目录项："."和".."。前者的inode号码就是当前目录的inode号码，等同于当前目录的"硬链接"；后者的inode号码就是当前目录的父目录的inode号码，等同于父目录的"硬链接"。所以，任何一个目录的"硬链接"总数，总是等于2加上它的子目录总数（含隐藏目录）,这里的2是父目录对其的“硬链接”和当前目录下的".硬链接“。

软链接

文件A和文件B的inode号码虽然不一样，但是文件A的内容是文件B的路径。读取文件A时，系统会自动将访问者导向文件B。因此，无论打开哪一个文件，最终读取的都是文件B。这时，文件A就称为文件B的"软链接"（soft link）或者"符号链接（symbolic link）。

这意味着，文件A依赖于文件B而存在，如果删除了文件B，打开文件A就会报错："No such file or directory"。这是软链接与硬链接最大的不同：文件A指向文件B的文件名，而不是文件B的inode号码，文件B的inode"链接数"不会因此发生变化。

ln -s命令可以创建软链接。

ln -s 源文文件或目录 目标文件或目录

## 18.如何创建进程，如何创建守护进程

## 19.套接字从write写入数据后，数据是如何发送到远端主机的

## 20.举个会发生死锁的例子

## 22.线程安全

# 数据库

## 1. 数据库范式

第一范式：列不可分，

eg:【联系人】（姓名，性别，电话），一个联系人有家庭电话和公司电话，那么这种表结构设计就没有达到 1NF；

第二范式：有主键，保证完全依赖。

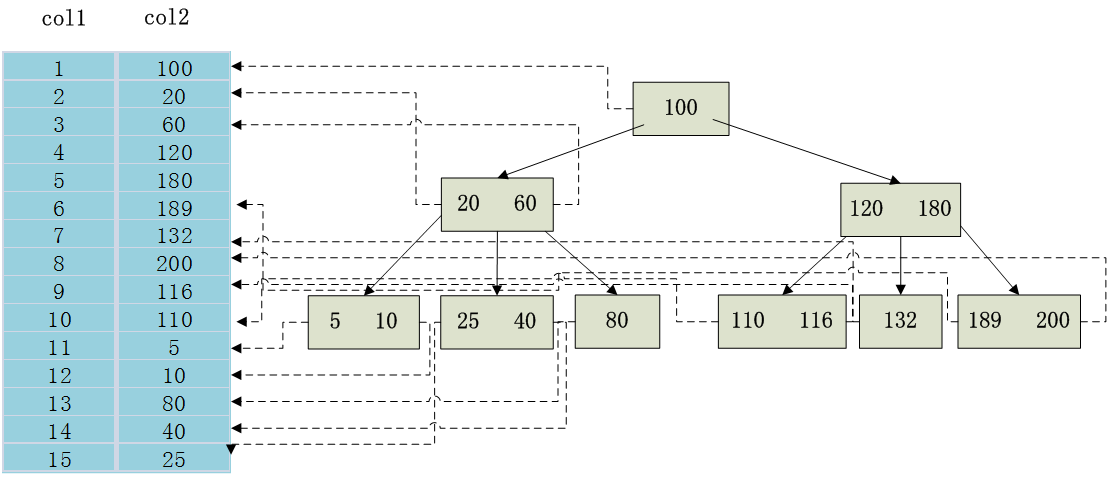
eg:订单明细表【OrderDetail】（OrderID，ProductID，UnitPrice，Discount，Quantity，ProductName），Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID，不符合2NF；

第三范式：无传递依赖(非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况)，

eg:订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID），CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。

## 数据库索引

**索引是对数据库表中一个或多个列的值进行排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B\_TREE及其变种。索引加速了数据访问，因为存储引擎不会再去扫描整张表得到需要的数据；相反，它从根节点开始，根节点保存了子节点的指针，存储引擎会根据指针快速寻找数据。



　　上图显示了一种索引方式。左边是数据库中的数据表，有col1和col2两个字段，一共有15条记录；右边是以col2列为索引列的B\_TREE索引，每个节点包含索引的键值和对应数据表地址的指针，这样就可以都过B\_TREE在 **O(logn)** 的时间复杂度内获取相应的数据，这样明显地加快了检索的速度。

## 索引的底层实现（B树）和优化（B+树）

(1). B-Tree(平衡多路查找树)

　　B\_TREE是一种平衡多路查找树，是一种动态查找效率很高的树形结构。一颗m阶的B\_TREE或是一颗空树，或者是满足下列条件的m叉树：

树中每个结点最多有m个孩子结点；

若根结点不是叶子节点，则根结点至少有2个孩子结点；

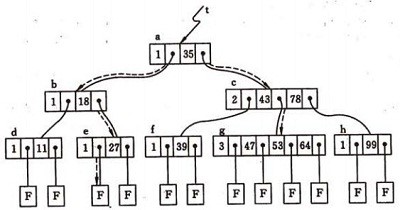
除根结点外，其它结点至少有(m/2的上界)个孩子结点；

结点的结构如下图所示，其中，n为结点中关键字个数，(m/2的上界)-1 <= n <= m-1；di(1<=i<=n)为该结点的n个关键字值的第i个，且di< d(i+1)；ci(0<=i<=n)为该结点孩子结点的指针，且ci所指向的节点的关键字均大于或等于di且小于d(i+1)；

B-Tree结点的结构.png-1.7kB

所有的叶结点都在同一层上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

　　下图是一棵4阶B\_TREE，4叉树结点的孩子结点的个数范围[2,4]。其中，有2个结点有4个孩子结点，有1个结点有3个孩子结点，有5个结点有2个孩子结点。



B\_TREE的查找类似二叉排序树的查找，所不同的是B-树每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码。由于B\_TREE的高检索效率，B-树主要应用在文件系统和数据库中，对于存储在硬盘上的大型数据库文件，可以极大程度减少访问硬盘次数，大幅度提高数据检索效率。

(2). B+Tree ： InnoDB存储引擎的索引实现

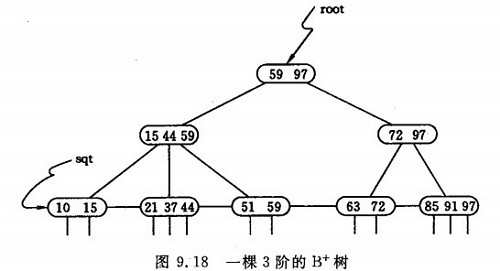
　　B+Tree是应文件系统所需而产生的一种B\_TREE树的变形树。一棵m阶的B+树和m阶的B\_TREE的差异在于以下三点：

n 棵子树的结点中含有n个关键码；

所有的叶子结点中包含了全部关键码的信息，及指向含有这些关键码记录的指针，且叶子结点本身依关键码的大小自小而大的顺序链接；

非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键码。

　　下图为一棵3阶的B+树。通常在B+树上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点。因此可以对B+树进行两种查找运算：一种是从最小关键字起顺序查找，另一种是从根节点开始，进行随机查找。 在B+树上进行随机查找、插入和删除的过程基本上与B-树类似。只是在查找时，若非终端结点上的关键码等于给定值，并不终止，而是继续向下直到叶子结点。因此，对于B+树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。



## 为什么B+树比B树更适合实际应用中操作系统的文件索引和数据库索引？

B+tree的磁盘读写代价更低：B+tree的内部结点并没有指向关键字具体信息的指针(红色部分)，因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多，相对来说IO读写次数也就降低了；

文件与数据库都是需要较大的存储，也就是说，它们都不可能全部存储在内存中，故需要存储到磁盘上。而所谓索引，则为了数据的快速定位与查找，那么索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数，因此B+树相比B树更为合适。

B+tree的查询效率更加稳定：任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；

**数据库索引采用B+树而不是B树的主要原因：**B+树只要遍历叶子节点就可以实现整棵树的遍历，而且在数据库中基于范围的查询是非常频繁的，而B树只能中序遍历所有节点，效率太低。

数据库系统巧妙利用了局部性原理与磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入，而红黑树这种结构，高度明显要深的多，并且由于逻辑上很近的节点(父子)物理上可能很远，无法利用局部性。

## 索引的优点和缺点

优点：

（1）大大加快数据的检索速度，这也是创建索引的最主要的原因；

（2）加速表和表之间的连接；

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

（4）通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性；

缺点：

（1）时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度；

（2）空间方面：索引需要占物理空间。

## 6. 索引的分类（唯一索引）

（1）主键、自增主键、主键索引与唯一索引概念区别

主键：指字段 唯一、不为空值 的列；

主键索引：指的就是主键，主键是索引的一种，是唯一索引的特殊类型。创建主键的时候，数据库默认会为主键创建一个唯一索引；

自增主键：字段类型为数字、自增、并且是主键；

唯一索引：索引列的值必须唯一，但允许有空值。主键是唯一索引，这样说没错；但反过来说，唯一索引也是主键就错误了，因为唯一索引允许空值，主键不允许有空值，所以不能说唯一索引也是主键。

（2）索引分类

普通索引和唯一索引：索引列的值的唯一性

单个索引和复合索引：索引列所包含的列数

聚集索引与非聚集索引：聚集索引按照数据的物理存储进行划分的。对于一堆记录来说，使用聚集索引就是对这堆记录进行堆划分，主要描述的是物理上的存储。这种划分方法，导致聚集索引必须是唯一的。聚集索引可以帮助把很大的范围，迅速减小范围。但是查找该记录，就要从这个小范围中Scan了；而非聚集索引是把一个很大的范围，转换成一个小的地图，然后你需要在这个小地图中找你要寻找的信息的位置，最后通过这个位置，再去找你所需要的记录。

（3）主键就是聚集索引吗？主键和索引有什么区别？

**主键是一种特殊的唯一性索引，其可以是聚集索引，也可以是非聚集索引。**在SQLServer中，主键的创建必须依赖于索引，默认创建的是聚集索引，但也可以显式指定为非聚集索引。InnoDB作为MySQL存储引擎时，默认按照主键进行聚集，如果没有定义主键，InnoDB会试着使用唯一的非空索引来代替。如果没有这种索引，InnoDB就会定义隐藏的主键然后在上面进行聚集。所以，对于聚集索引来说，你创建主键的时候，自动就创建了主键的聚集索引。

## 7. 什么情况下设置了索引但无法使用？

以“%(表示任意0个或多个字符)”开头的LIKE语句，模糊匹配；

OR语句前后没有同时使用索引；

数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）；

对于多列索引，必须满足**最左匹配原则** (eg：多列索引col1、col2和col3，则 索引生效的情形包括 col1或col1，col2或col1，col2，col3)。

## 8. 什么样的字段适合创建索引？

经常作查询选择的字段

经常作表连接的字段

经常出现在order by, group by, distinct 后面的字段

## 9. 创建索引时需要注意什么？

非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。应该用0、一个特殊的值或者一个空串代替空值；

取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多，字段的离散程度越高；

索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多，一次IO操作获取的数据越大，效率越高。

## 10. mysql的底层存储的数据结构是啥

## 11. 数据库事务，事物的特性，隔离性的四个级别

事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。

### (1) 事务的特征（ACID）

原子性(Atomicity)：事务所包含的一系列数据库操作要么全部成功执行，要么全部回滚；

一致性(Consistency)：事务的执行结果必须使数据库从一个一致性状态到另一个一致性状态；

隔离性(Isolation)：因为数据库允许多个并发十五同时对数据进行读写，隔离性要求并发执行的事务之间不能相互影响，避免多事务并发执行时由于交叉执行导致数据不一致；

持久性(Durability)：事务一旦提交，对数据库中数据的改变是永久性的。

### (2) 隔离性的4个级别

READ UNCOMMITTED（读未提交）：最低级别的隔离，通常又称为dirty read，它允许一个事务读取另一个事务还没commit的数据，这样可能会提高性能，但是会导致脏读问题；

READ COMMITTED（读提交）：在一个事务中只允许对其它事务已经commit的记录可见，该隔离级别不能避免不可重复读问题；

REPEATABLE READ（可重复读）：在一个事务开始后，其他事务对数据库的修改在本事务中不可见，直到本事务commit或rollback。但是，其他事务的insert/delete操作对该事务是可见的，也就是说，该隔离级别并不能避免幻读问题。在一个事务中重复select的结果一样，除非本事务中update数据库。

SERIALIZABLE（串行化）：最高级别的隔离，只允许事务串行执行。

脏读：一个事务读取了另一个事务未提交的数据；

不可重复读：不可重复读的重点是修改，同样条件下两次读取结果不同，也就是说，被读取的数据可以被其它事务修改；

幻读：幻读的重点在于新增或者删除，同样条件下两次读出来的记录数不一样。

MySQL默认的隔离级别是REPEATABLE READ。

### (3) mysql的事务支持

　　MySQL的事务支持不是绑定在MySQL服务器本身，而是与存储引擎相关：

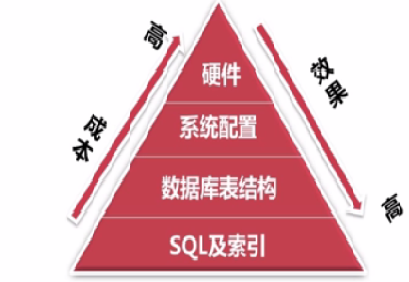
MyISAM：不支持事务，用于只读程序提高性能；

InnoDB：支持ACID事务、行级锁、并发；

Berkeley DB：支持事务。

## 12. 数据库优化

实践中，MySQL的优化主要涉及SQL语句及索引的优化、数据表结构的优化、系统配置的优化和硬件的优化四个方面，如下图所示：



　　　　　　　　　　　　　　Mysql性能优化-82.5kB

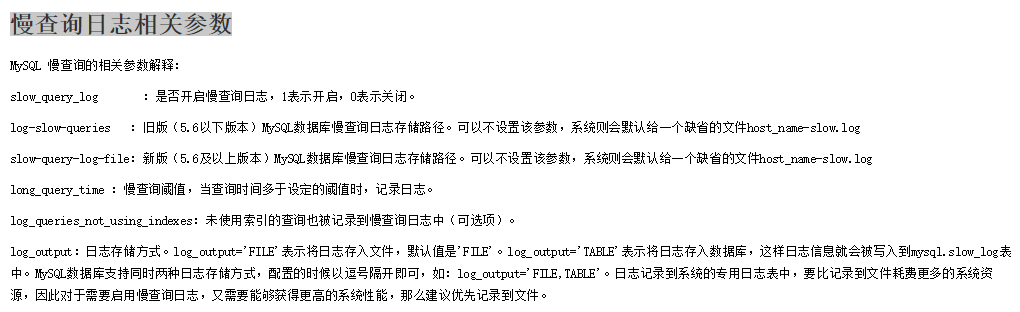
### 1）SQL语句及索引的优化

（1）SQL语句的优化

　　SQL语句的优化主要包括三个问题，即如何发现有问题SQL、如何分析SQL的执行计划以及如何优化SQL。

a. 怎么发现有问题的SQL?（通过MySQL慢查询日志对有效率问题的SQL进行监控）

　　MySQL的慢查询日志是MySQL提供的一种日志记录，它用来记录在MySQL中响应时间超过阀值的语句，具体指运行时间超过long\_query\_time值的SQL，则会被记录到慢查询日志中。long\_query\_time的默认值为10，意思是运行10s以上的语句。慢查询日志的相关参数如下所示：



　　　　　　　　慢查询日志相关参数.png-20.4kB

　　通过MySQL的慢查询日志，我们可以查询出执行的次数多占用的时间长的SQL、可以通过pt\_query\_disgest(一种mysql慢日志分析工具)分析Rows examine(MySQL执行器需要检查的行数)项去找出IO大的SQL以及发现未命中索引的SQL，对于这些SQL，都是我们优化的对象。

b. 通过explain查询和分析SQL的执行计划

　　使用 EXPLAIN 关键字可以知道MySQL是如何处理你的SQL语句的，以便分析查询语句或是表结构的性能瓶颈。通过explain命令可以得到表的读取顺序、数据读取操作的操作类型、哪些索引可以使用、哪些索引被实际使用、表之间的引用以及每张表有多少行被优化器查询等问题。当扩展列extra出现Using filesort和Using temporay，则往往表示SQL需要优化了。

c. SQL语句的优化

优化insert语句：一次插入多值；

应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描；

应尽量避免在 where 子句中对字段进行null值判断，否则将导致引擎放弃使用索引而进行全表扫描；

优化嵌套查询：子查询可以被更有效率的连接(Join)替代；

很多时候用 exists 代替 in 是一个好的选择。

(2) 索引优化

　　建议在经常作查询选择的字段、经常作表连接的字段以及经常出现在order by、group by、distinct 后面的字段中建立索引。

### 2) 数据库表结构的优化

　　数据库表结构的优化包括选择合适数据类型、表的范式的优化、表的垂直拆分和表的水平拆分等手段。

(1). 选择合适数据类型

使用较小的数据类型解决问题；

使用简单的数据类型(mysql处理int要比varchar容易)；

尽可能的使用not null 定义字段；

尽量避免使用text类型，非用不可时最好考虑分表；

(2). 表的范式的优化: 表的设计应该遵循三大范式。

(3). 表的垂直拆分

　　把含有多个列的表拆分成多个表，解决表宽度问题，具体包括以下几种拆分手段：

把不常用的字段单独放在同一个表中；

把大字段独立放入一个表中；

把经常使用的字段放在一起；

这样做的好处是非常明显的，具体包括：拆分后业务清晰，拆分规则明确、系统之间整合或扩展容易、数据维护简单。

(4). 表的水平拆分:解决数据表中数据过大的问题

水平拆分每一个表的结构都是完全一致的。一般地，将数据平分到N张表中的常用方法包括以下两种：

对ID进行hash运算，如果要拆分成5个表，mod(id,5)取出0~4个值；

针对不同的hashID将数据存入不同的表中；

问题和挑战：跨分区表的数据查询、统计及后台报表的操作等问题

优点：

表分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询速度；

表中的数据本来就有独立性，例如表中分别记录各个地区的数据或不同时期的数据，特别是有些数据常用，而另外一些数据不常用。

需要把数据存放到多个数据库中，提高系统的总体可用性(分库，鸡蛋不能放在同一个篮子里)。

### 3). 系统配置的优化

操作系统配置的优化：增加TCP支持的队列数

mysql配置文件优化：Innodb缓存池设置(innodb\_buffer\_pool\_size，推荐总内存的75%)和缓存池的个数（innodb\_buffer\_pool\_instances）

### 4). 硬件的优化

CPU：核心数多并且主频高的

内存：增大内存

磁盘配置和选择：磁盘性能

## 13. 数据库插入数据时怎么判断是否重复操作

（1）对于主键和唯一索引，可以用IGNORE关键字，遇到重复记录会直接忽略插入记录，返回0。

insert ignore into table\_name ('id','name') values (1,'eddy')

（2）replace关键字：REPLACE的运行与INSERT很相像,但是如果旧记录与新记录有相同的值，则在新记录被插入之前，旧记录被删除。REPLACE返回受影响的行数。

replace into table\_name ('id','name') values (1,'eddy')

（3）ON DUPLICATE KEY UPDATE：遇到重复的记录则更新指定的字段。如果行作为新记录被插入，则受影响行的值为1；如果原有的记录被更新，则受影响行的值为2。

insert ignore into table\_name ('id','name') values (1,'eddy') on

duplicate key update id = 100

## 14. 什么是存储过程？有哪些优缺点？

存储过程是事先经过编译并存储在数据库中的一段SQL语句的集合。进一步地说，存储过程是由一些T-SQL语句组成的代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。

存储过程具有以下特点：存储过程只在创建时进行编译，以后每次执行存储过程都不需再重新编译，而一般 SQL 语句每执行一次就编译一次，所以使用存储过程可提高数据库执行效率；

当SQL语句有变动时，可以只修改数据库中的存储过程而不必修改代码；

减少网络传输，在客户端调用一个存储过程当然比执行一串SQL传输的数据量要小；

通过存储过程能够使没有权限的用户在控制之下间接地存取数据库，从而确保数据的安全。

## 15. drop、delete与truncate的区别

Delete用来删除表的全部或者一部分数据行，执行delete之后，用户需要提交(commmit)或者回滚(rollback)来执行删除或者撤销删除，delete命令会触发这个表上所有的delete触发器；

Truncate删除表中的所有数据，这个操作不能回滚，也不会触发这个表上的触发器，TRUNCATE比delete更快，占用的空间更小；

Drop命令从数据库中删除表，所有的数据行，索引和权限也会被删除，所有的DML触发器也不会被触发，这个命令也不能回滚。

因此，在不再需要一张表的时候，用drop；在想删除部分数据行时候，用delete；在保留表而删除所有数据的时候用truncate。

## 16. 什么叫视图？游标是什么？触发器？

　　视图是一种虚拟的表，通常是有一个表或者多个表的行或列的子集，具有和物理表相同的功能，可以对视图进行增，删，改，查等操作。特别地，对视图的修改不影响基本表。相比多表查询，它使得我们获取数据更容易。

游标（cursor）是一个存储在MySQL服务器上的数据库查询，它不是一条 SELECT语句，而是被该语句检索出来的结果集。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。在存储了游标之后，应用程序可以根据需要滚动或浏览其中的数据。游标主要用于交互式应用，其中用户需要滚动屏幕上的数据，并对数据进行浏览或做出更改。

在操作mysql的时候，我们知道MySQL检索操作返回一组称为结果集的行。这组返回的行都是与 SQL语句相匹配的行（零行或多行）。使用简单的 SELECT语句，例如，没有办法得到第一行、下一行或前 10行，也不存在每次一行地处理所有行的简单方法（相对于成批地处理它们）。有时，需要在检索出来的行中前进或后退一行或多行。这就是使用游标的原因。

　　触发器是与表相关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用在数据库端确保数据库的完整性。

## 17. MySQL中的悲观锁与乐观锁的实现（资源并发锁）

(1). 悲观锁

悲观锁的特点是先获取锁，再进行业务操作，即“悲观”的认为所有的操作均会导致并发安全问题，因此要先确保获取锁成功再进行业务操作。

通常来讲，在数据库上的悲观锁需要数据库本身提供支持，即通过常用的select … for update操作来实现悲观锁。当数据库执行select … for update时会获取被select中的数据行的行锁，因此其他并发执行的select … for update如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

这里需要特别注意的是，不同的数据库对select… for update的实现和支持都是有所区别的，例如oracle支持select for update no wait，表示如果拿不到锁立刻报错，而不是等待，mysql就没有no wait这个选项。另外，mysql还有个问题是: select… for update语句执行中所有扫描过的行都会被锁上，这一点很容易造成问题。因此，如果在mysql中用悲观锁务必要确定使用了索引，而不是全表扫描。

(2). 乐观锁

乐观锁的特点先进行业务操作，只在最后实际更新数据时进行检查数据是否被更新过，若未被更新过，则更新成功；否则，失败重试。乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。一般的做法是在需要锁的数据上增加一个版本号或者时间戳，然后按照如下方式实现：

SELECT data AS old\_data, version AS old\_version FROM …;

2. 根据获取的数据进行业务操作，得到new\_data和new\_version

3. UPDATE SET data = new\_data, version = new\_version WHERE version = old\_version

if (updated row > 0) {

// 乐观锁获取成功，操作完成

} else {

// 乐观锁获取失败，回滚并重试

}

乐观锁是否在事务中其实都是无所谓的，其底层机制是这样：在数据库内部update同一行的时候是不允许并发的，即数据库每次执行一条update语句时会获取被update行的写锁，直到这一行被成功更新后才释放。因此在业务操作进行前获取需要锁的数据的当前版本号，然后实际更新数据时再次对比版本号确认与之前获取的相同，并更新版本号，即可确认这其间没有发生并发的修改。如果更新失败，即可认为老版本的数据已经被并发修改掉而不存在了，此时认为获取锁失败，需要回滚整个业务操作并可根据需要重试整个过程。

(3). 悲观锁与乐观锁的应用场景

　　一般情况下，读多写少更适合用乐观锁，读少写多更适合用悲观锁。乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能。

## MySQL存储引擎中的MyISAM和InnoDB区别详解

　　在MySQL 5.5之前，MyISAM是mysql的默认数据库引擎，其由早期的ISAM（Indexed Sequential Access Method：有索引的顺序访问方法）所改良。虽然MyISAM性能极佳，但却有一个显著的缺点： 不支持事务处理。不过，MySQL也导入了另一种数据库引擎InnoDB，以强化参考完整性与并发违规处理机制，后来就逐渐取代MyISAM。

InnoDB是MySQL的数据库引擎之一，其由Innobase oy公司所开发，2006年五月由甲骨文公司并购。与传统的ISAM、MyISAM相比，**InnoDB的最大特色就是支持****ACID（事务的四大特性）兼容的事务功能**，类似于PostgreSQL。目前InnoDB采用双轨制授权，一是GPL授权，另一是专有软件授权。具体地，MyISAM与InnoDB作为MySQL的两大存储引擎的差异主要包括：

存储结构：每个MyISAM在磁盘上存储成三个文件：第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义，数据文件的扩展名为.MYD (MYData)，索引文件的扩展名是.MYI (MYIndex)。InnoDB所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。

存储空间：MyISAM可被压缩，占据的存储空间较小，支持静态表、动态表、压缩表三种不同的存储格式。InnoDB需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

可移植性、备份及恢复：MyISAM的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便，同时在备份和恢复时也可单独针对某个表进行操作。InnoDB免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

事务支持：MyISAM强调的是性能，每次查询具有原子性，其执行数度比InnoDB类型更快，但是不提供事务支持。InnoDB提供事务、外键等高级数据库功能，具有事务提交、回滚和崩溃修复能力。

AUTO\_INCREMENT：在MyISAM中，可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，它可以根据前面几列进行排序后递增。InnoDB中必须包含只有该字段的索引，并且引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

表锁差异：MyISAM只支持表级锁，用户在操作MyISAM表时，select、update、delete和insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。InnoDB支持事务和行级锁。行锁大幅度提高了多用户并发操作的新能，但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。

全文索引：MyISAM支持 FULLTEXT类型的全文索引；InnoDB不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。

表主键：MyISAM允许没有任何索引和主键的表存在，索引都是保存行的地址。对于InnoDB，如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。

表的具体行数：MyISAM保存表的总行数，select count(*) from table;会直接取出出该值；而InnoDB没有保存表的总行数，如果使用select count(*) from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，myisam和innodb处理的方式都一样。

CURD操作：在MyISAM中，如果执行大量的SELECT，MyISAM是更好的选择。对于InnoDB，如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。DELETE从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。

外键：MyISAM不支持外键，而InnoDB支持外键。

　　通过上述的分析，基本上可以考虑使用InnoDB来替代MyISAM引擎了，原因是InnoDB自身很多良好的特点，比如事务支持、存储过程、视图、行级锁、外键等等。尤其在并发很多的情况下，相信InnoDB的表现肯定要比MyISAM强很多。另外，必须需要注意的是，任何一种表都不是万能的，合适的才是最好的，才能最大的发挥MySQL的性能优势。如果是不复杂的、非关键的Web应用，还是可以继续考虑MyISAM的，这个具体情况具体考虑。

## join语句和数据库移植问题

环境：

A机器：cenos6.5、mysql是5.5

B机器：cenos7、mysql是5.5

移植前的准备：

1、找到mysql数据库数据文件的位置：find / -name 'mysql'

找到所有mysql的目录查看A、B机器的数据文件的位置。 一般mysql5.5默认的位置是/var/lib/mysql

2、查看linux系统中内存大小，是否能符合移植的需要：df -h

3、初步了解数据文件的格式

1）从文件的扩展名是myisam类型的。

.frm 是描述表结构的文件

.MYD 是表的数据文件

.MYI 是表数据文件中任何索引的数据树。

2）而有些数据文件没有.MYD是数据存储在ibdata1。

因此我们所要做的是将整个/var/lib/mysql整个目录拷贝到B机器上

准备移植

1、首先关闭A、B机器的mysql服务：service mysql stop

2、备份B机器的mysql数据（以防移植后出现问题）

3、使用scp指令传输文件：scp -r [root@192.168.1.1:/var/lib/mysql](mailto:root@192.168.1.1:/var/lib/mysql) /var/lib

其中192.168.1.1是A机器的ip地址。由于文件110G左右，传输速度基本稳定在10M左右，传输时间基本可以接受。

Scp –r；递归复制整个目录

4、这个时候从A机器拷贝的mysql 数据文件由于是在root权限下执行的，因此数据文件的权限是root，需要将权限改为mysql

chown -r mysql:mysql

5、若复制后的文件中有mysql-bin-xxx格式的文件，会导致启动mysql服务失败，若有，需删除。

6、确认不缺失mysql数据库的前提下，开启mysql服务：service mysql start

## 20. 用MySQL语法建 一个学生表，包括学生姓名、性别、年龄、班级信息。

CREAT TABLE student (

　　ID int primary key not null,

　　NAME varchar(50),

　　sex int,

　　age int,

　　classNo int )

## 21. char()与varchar()的区分，什么情况下用char()？（两者区别很重要）

char的长度是不可变的，以空间换取时间效率，而varchar的长度是可变的，也就是说，定义一个char[10]和varchar[10],如果存进去的是‘csdn’,那么char所占的长度依然为10，除了字符‘csdn’外，后面跟六个空格，而varchar就立马把长度变为4了，取数据的时候，char类型的要用trim()去掉多余的空格，而varchar是不需要的。

char的存储方式是，对英文字符（ASCII）占用1个字节，对一个汉字占用两个字节；而varchar的存储方式是，对每个英文字符占用2个字节，汉字也占用2个字节。

两者的存储数据都非unicode的字符数据。

## 22. 在sql语句中加上字符集的方法。

在连接数据库之后，读取数据之前，先执行一项查询“SET NAMES UTF8”

mysql\_query("set names 'utf8'");

## 23. 密码传输 如何保持安全性。

## 24. 有A,B用户，从A转100块钱到B用户，写出sql语句，需要注意的问题

start transaction

update from account set money=money+100 where name='b';

update from account set money=money-100 where name='a';

commit

# 大数据

## 1. 分布式系统怎么根据日志实现故障报警（检查日志文件单位时间正确的量和错误的量的变化，动态采样。）如何统计各分布式机器上的日志（根据日志，将某个模块的日志数量求和上报，发现故障直接连入该故障机器检查日志，有点类似namenode和datanode的关系）

## 2. 让自己设计一个类似map/reduce的分布式计算方法，能够实时计算大量数据

## 3. 聚类算法、分类算法

# 设计模式

单例、观察者、责任链

工厂、装饰器、适配器等

## 1、用到的设计模式、再说一个设计模式/2

## 2、画设计模式的类图(观察者模式)

## 3、单例模式

# 建模

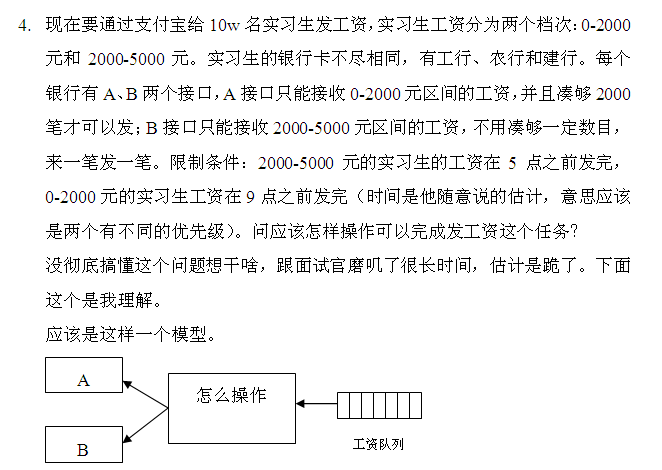
## 1、一个水果经营商每天去市场进水果和干果，拿到自己商店卖，每天盘 点库存，期望得到收支平衡的数据。抽象建模，抽象出各个类的层次 ，说明其协作关系。 说明组合方式和依赖方式的区别

# 开放性问题

## 1、两个文本文件，一个是过年发红包的1亿个qq用户的id信息，另外一个是5千万个会员的id信息，求出过年发红包的会员，也即两个的交集

## 2、你家有头牛，你如何说服别人把你家牛卖个好价钱

## 3、在服务器和客户端上有两个文件，均为10G，其中只有一个字节出错，如何用最快的方式同步这两个文件



# 项目:

## 1、项目选题

## 2、项目采用的技术

## 3、新颖的地方

## 4、最终达到的效果

我**从项目的选题（为啥要做这个方向）、项目采用的技术、新颖的地方、最终达到的效果（由于我做的是某个算法的性能提高，那么我就会讲速度提高了多少？空间压缩了多少？）这几个方面详细介绍了自己的项目**。

# 高并发

1.

# 程序

## 实现memcpy函数

函数原型：void \*memcpy(void\*dest, const void \*src, size\_t n);

功能：由src指向地址为起始地址的连续n个字节的数据复制到以destin指向地址为起始地址的空间内。

头文件： #include<string.h>

返回值：函数返回一个指向dest的指针。

说明：

　　1.source和destin所指内存区域不能重叠，函数返回指向destin的指针。对于地址重叠的情况，该函数的行为是未定义的。

　　2.与strcpy相比，memcpy并不是遇到'\0'就结束，而是一定会拷贝完n个字节。

memcpy用来做内存拷贝，你可以拿它拷贝任何数据类型的对象，可以指定拷贝的数据长度；

　　3.如果目标数组destin本身已有数据，执行memcpy（）后，将覆盖原有数据（最多覆盖n）。如果要追加数据，则每次执行memcpy后，要将目标数组地址增加到你要追加数据的地址。

　　//注意，source和destin都不一定是数组，任意的可读写的空间均可。

本题的陷阱：自己动手实现memcpy()时就需要考虑地址重叠的情况。

**[cpp]** [view plain](https://blog.csdn.net/laoyang360/article/details/8020409) [copy](https://blog.csdn.net/laoyang360/article/details/8020409)

1. **int** main(intargc, **char** \*argv[])  {
2. **char** buf[100] = "abcdefghijk";
3. memcpy(buf+2, buf, 5);
4. //Memcpy(buf+2, buf, 5);
5. printf("%s\n", buf+2);
6. **return** 0;
7. }
9. **void**\* Memcpy(**void** \*dst, **const** **void** \*src, **size\_t** size)  {
10. **char** \*psrc, \*pdst;
11. **if**(NULL == dst || NULL == src)
12. **return** NULL;
13. **if**((src < dst) && (**char** \*)src +size > (**char** \*)dst) { //有覆盖，自后向前拷贝
14. psrc = (**char** \*)src + size - 1;
15. pdst = (**char** \*)dst + size - 1;
16. **while**(size--)
17. \*pdst-- = \*psrc--;
18. }
19. **else**  {
20. psrc = (**char** \*)src;
21. pdst = (**char** \*)dst;
22. **while**(size--)
23. \*pdst++ = \*psrc++;
24. }
25. **return** dst;
26. }

## 给定 N 张扑克牌和一个随机函数，设计一个洗牌算法。

## 3.机器人移动问题

两个机器人，初始时位于数轴上的不同位置。给这两个机器人输入一段相同的程序，使得这两个机器人保证可以相遇。程序只能包含“左移n个单位”、“右移n个单位”，条件判断语句If，循环语句while，以及两个返回Boolean值的函数“在自己的起点处”和“在对方的起点处”。你不能使用其它的变量和计数器。

答案：两个机器人同时开始以单位速度右移，直到一个机器人走到另外一个机器人的起点处。然后，该机器人以双倍速度追赶对方。

while(!at\_other\_robots\_start) {

move\_right 1

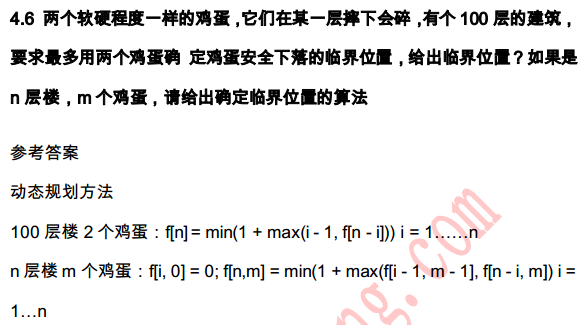
}

while(true) {

move\_right 2

}

## 4. 两个软硬程度一样的鸡蛋，它们在某一层摔下会碎，有个 100 层的建筑，要求最多用两个鸡蛋确 定鸡蛋安全下落的临界位置，给出临界位置？如果是n 层楼，m 个鸡蛋，请给出确定临界位置的算法



5.蓄水池采样法