在优化MySQL时，通常需要对数据库进行分析，常见的分析手段有慢查询日志，EXPLAIN 分析查询，profiling分析以及show命令查询系统状态及系统变量，通过定位分析性能的瓶颈，才能更好的优化数据库系统的性能。

开启慢查询日志方法1

在my.cnf或者my.ini中添加log-queries-not-using-indexes参数，表示记录下没有使用索引的查询。

log-slow-queries=/data/mysqldata/slow-query.log

long\_query\_time=10

log-queries-not-using-indexes

方法2

set global slow\_query\_log = ON

mysql中有一个explain 命令可以用来分析select 语句的运行效果，例如explain可以获得select语句  
使用的索引情况、排序的情况等等。除此以外，explain 的extended 扩展能够在原本explain的基础  
上额外的提供一些查询优化的信息，这些信息可以通过mysql的show warnings命令得到。explain**extended** select \* from account\G;

mysql语句的执行时间，set profiling=1; 通过profiling来获取

mysql> show profiles\G; 可以得到被执行的SQL语句的时间和ID

mysql>show profile for query 1; 得到对应SQL语句执行的详细信息

二叉搜索树的插入操作非递归实现：

BinaryTreeNode<T> \* pointer=NULL;

if(NULL==root){

Initialize(newpointer);

return;

}

else pointer=root;

while(1){

if(newpointer->value()==pointer->value())

return;

else if(newpointer->value()<pointer->value())

{

if(pointer->leftchild()==NULL){

pointer->leftchild()=newpointer;

return;

}

else pointer=pointer->leftchild();

}

else{

if(pointer->rightchild()==NULL){

pointer->right=newpointer;

return;

}

else pointer=pointer->rightchild();

}

}

红黑树：

性质1. 节点是红色或黑色。  
性质2. 根节点是黑色。  
性质3. 每个叶子结点都带有两个空的黑色结点（被称为黑哨兵），如果一个结点n的只有一个左孩子，那么n的右孩子是一个黑哨兵；如果结点n只有一个右孩子，那么n的左孩子是一个黑哨兵。  
性质4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)  
性质5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

M阶B树：

每个节点至多有m个子节点

除根节点和叶节点外，其他每个节点至少有【m/2】个子节点

根节点至少有两个字节点

所有叶子节点在同一层

有k个字节点的非跟节点恰好包含k-1个关键码

节点的关键码是有序的，

插入节点的时候从根节点开始遍历，找到最底层插入，如果插入后节点的数量大于限制，那么需要分裂节点。中间关键码连同新指针插入父节点，分裂可能传递至根节点

删除节点时（如果不是叶子节点，那么需要交换到叶子节点）先找到该节点，然后删除，如果超过了下界，导致节点数量不足，然后就需要合并左右邻居的节点。

B+树：

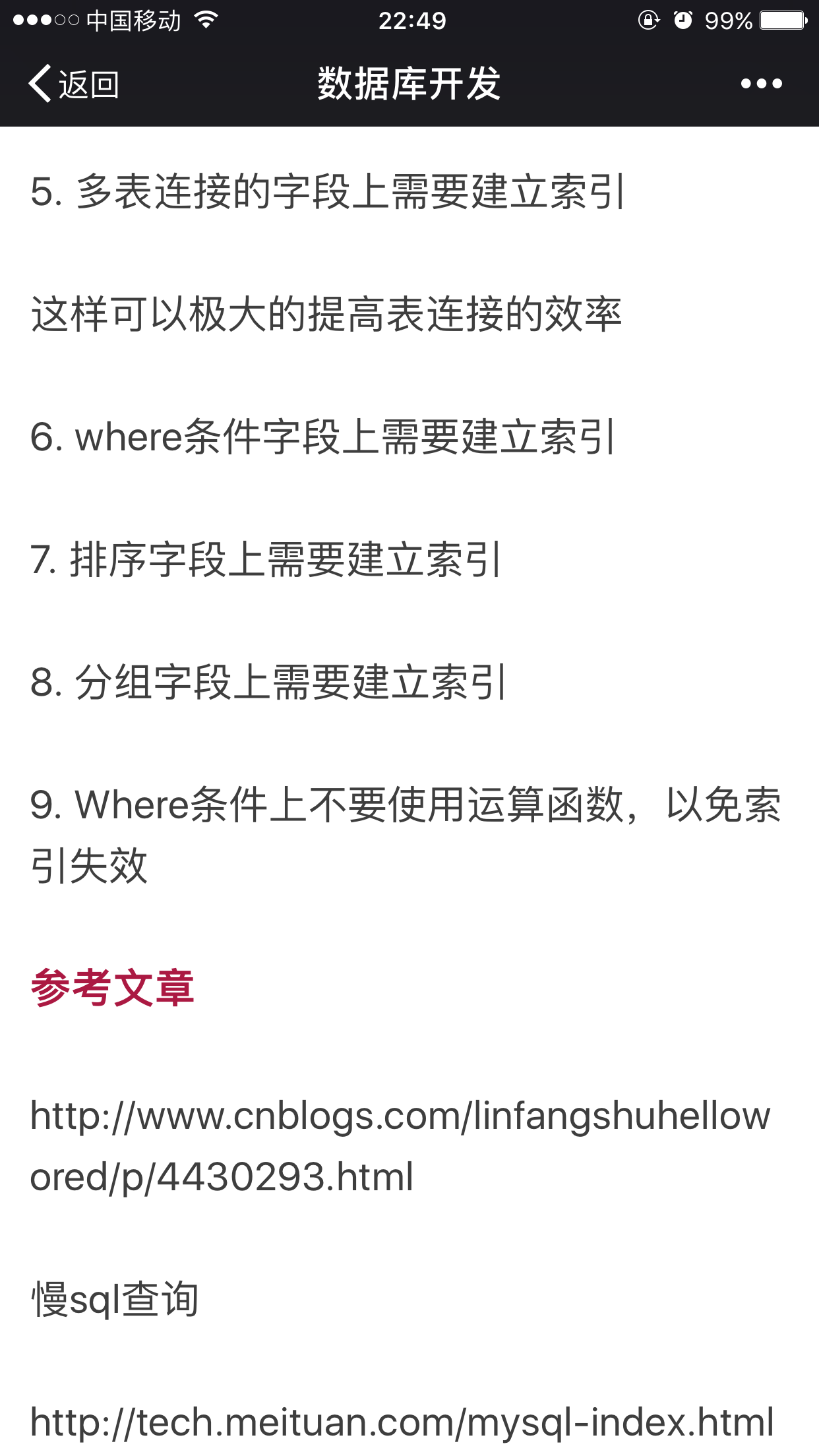
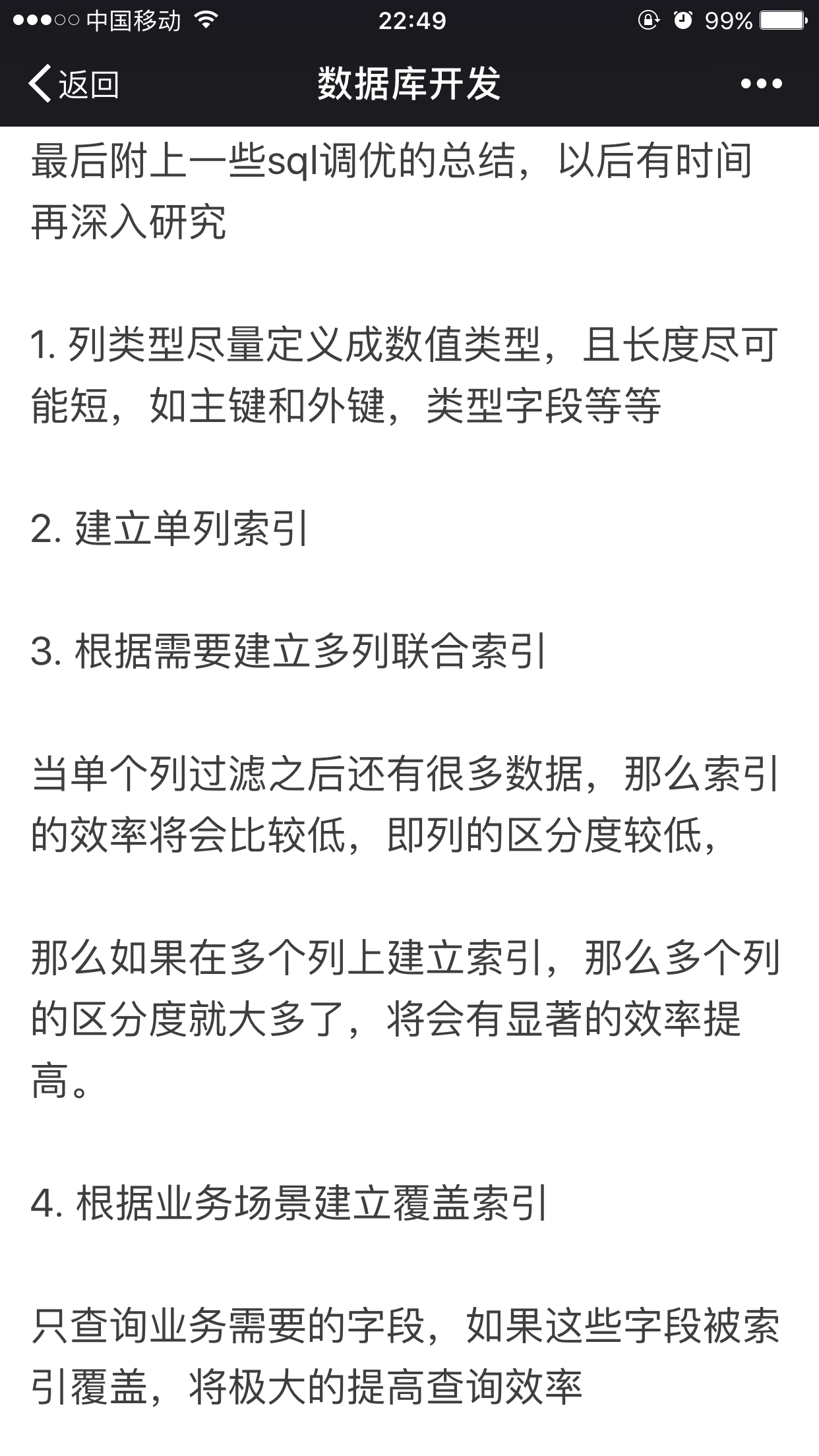
每个节点至多有m个子节点

每个节点除根节点外至少有【m/2】个子节点

根节点至少有两个子节点

有k个子节点的节点必有k个关键码

在叶子层是所有关键码的全集，上层是复写



索引覆盖：如果查询的列都建立了索引，这样获取结果集的时候不用再去磁盘获取其他列的数据了，直接返回索引值就可以。select age，sex from user where age = 10 and sex = 1; 在排序字段建立索引会提高效率

CREATE index user\_index\_sex on user(sex)

CREATE index user\_index\_age on user(age) 在user表中为sex和age建立两个索引

CREATE index user\_index\_sex\_age on user(sex,age)建立组合索引

那么为什么一般情况下join的效率要高于left join很多？很多人说不明白原因，只人云亦云，我今天下午感悟出来了一点。一般情况下参与联合查询的两张表都会一大一小，如果是join，在没有其他过滤条件的情况下MySQL会选择小表作为驱动表，但是left join一般用作大表去join小表，而left join本身的特性决定了MySQL会用大表去做驱动表，这样下来效率就差了不少，

•如果对读的响应度要求非常高，比如证券交易系统，那么适合用乐观锁，因为悲观锁会阻塞读

•如果读远多于写，那么也适合用乐观锁，因为用悲观锁会导致大量读被少量的写阻塞

•如果写操作频繁并且冲突比例很高，那么适合用悲观写独占锁

行级锁是MySQL中粒度最小的一种锁，他能大大减少数据库操作的冲突。但是粒度越小，实现的成本也越高。MYISAM引擎只支持表级锁，而INNODB引擎能够支持行级锁，下面的内容也是针对INNODB行级锁展开的。

INNODB的行级锁有共享锁（S LOCK）和排他锁（X LOCK）两种。共享锁允许事物读一行记录，不允许任何线程对该行记录进行修改。排他锁允许当前事物删除或更新一行记录，其他线程不能操作该记录。

共享锁：

用法： SELECT ... LOCK IN SHARE MODE;

MySQL会对查询结果集中每行都添加共享锁。

锁申请前提：当前没有线程对该结果集中的任何行使用排他锁，否则申请会阻塞。

排他锁：

用法： SELECT ... FOR UPDATE;

SELECT counter\_field FROM child\_codes FOR UPDATE;

UPDATE child\_codes SET counter\_field = counter\_field + 1;

MySQL会对查询结果集中每行都添加排他锁，在事物操作中，任何对记录的更新与删除操作会自动加上排他锁。

锁申请前提：当前没有线程对该结果集中的任何行使用排他锁或共享锁，否则申请会阻塞。

Session生成后，只要用户继续访问，服务器就会更新Session的最后访问时间，并维护该Session。用户每访问服务器一次，无论是否读写Session，服务器都认为该用户的Session“活跃（active）”了一次。

Cookie方式

用户发起登录请求，服务端根据传入的用户密码之类的身份信息，验证用户是否满足登录条件，如果满足，就根据用户信息创建一个登录凭证，这个登录凭证简单来说就是一个对象，最简单的形式可以只包含用户id，凭证创建时间和过期时间三个值。

服务端把上一步创建好的登录凭证，先对它做数字签名，然后再用对称加密算法做加密处理，将签名、加密后的字串，写入cookie。cookie的名字必须固定（如ticket），因为后面再获取的时候，还得根据这个名字来获取cookie值。这一步添加数字签名的目的是防止登录凭证里的信息被篡改，因为一旦信息被篡改，那么下一步做签名验证的时候肯定会失败。做加密的目的，是防止cookie被别人截取的时候，无法轻易读到其中的用户信息。

void Floyd()

{

v[1][1] = 0;

for (int k = 1; k <= MAX; k++)

for (int i = 1; i <= MAX; i++)

for (int j = 1; j <= MAX; j++)

if (v[i][k] + v[k][j]<v[i][j])

{

v[i][j] = v[i][k] + v[k][j];

p[i][j] = k;

}

}

template<class T>

class MinHeap

{

private:

T\* heapArray;//存放堆数据的数组

int CurrentSize;//当前堆中元素数目

int MaxSize;//堆所能容纳的最大元素数目

void BuildHeap();//建堆

public:

MinHeap(const int n);

MinHeap<T>::MinHeap(const int n)

{

if(n<=0)

return ;

CurrentSize=0;

MaxSize=n;//初始化堆容量为n

heapArray=new T[MaxSize];

BuildHeap();

}

};

建堆：从堆的第一个分支节点heapArray[CurrentSize/2-1]开始，自底向上逐步把以各分支节点为根的子树调整成堆。

template<class T>

void MinHeap<T>::BuildHeap()

{

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

SiftDown(i);

}

template<class T>

void MinHeap<T>::SiftDown(int position)

{

int i=position;//标识父节点

int j=2\*i+1;//标识关键值较小的子节点

T temp=heapArray[i];//保存父节点

while(j<CurrentSize){

if((j<CurrentSize-1)&&(heapArray[j]>heapArray[j+1]))

j++;

if(temp>heapArray[j]){

heapArray[i]=heapArray[j];

i=j;

j=2\*j+1;//向下继续

}

else break;

}

heapArray[i]=temp;

}

template<class T>

void MinHeap<T>::SiftUP(int position)

{//从position向上开始调整，使序列成为堆

int temppos=position;

T temp=heapArray[temppos];

while((temppos>0)&&(heapArray[parent(temppos)]>temp))

{//比较父节点直接swap的方法。

heapArray[temppos]=heapArray[parent(temppos)];

temppos=parent(temppos);

}

heapArray[temppos]=temp;

}

void MinHeap<T>:: SiftUP (int start) //自下往上调整

{

int j=start,i=(j-1)/2; //i指向j的双亲节点

T temp=heap[j];

while(j>0)

{

if(heap[i]<=temp)

break;

else{

heap[j]=heap[i];

j=i;

i=(i-1)/2;

}

}

heap[j]=temp;

}

template<class T>

int MinHeap<T>::parent(int pos) const

{

return (pos-1)/2;

}

堆中删除最小元素：将最后一个替换根节点，然后执行siftdown(0)，删除普通元素，用最后一个元素替换该元素，然后siftup();siftdown()。

template<class T>

bool MinHeap<T>::Remove(int pos,T& node)

{

if((pos<0)||(pos>=CurrentSize))

return false;

T temp=heapArray[pos];

heapArray[pos]=heapArray[--CurrentSize];

Siftup(pos);

Siftdown(pos);

node=temp;

return true;

}

bool MinHeap<T>::Insert(const T& newNode)

{

if(currentsize==maxsize)//堆空间已经满

return FALSE;

heapArray[currentsize]=newnode;

siftup(currentsize);//向上调整

currentsize++;

}

建堆的效率：时间复杂度是O(n)。由于堆有log n层深，插入节点，删除普通节点和删除最小元素的平均时间代价和最差时间代价都是O(log n)。

堆是优先队列的一种自然的实现方法。

散列基本思想：1确定一个函数关系。2以节点的关键码k为自变量。3函数值h(k)作为节点的存储地址。4检索时也是根据这个函数计算其存储位置。

通常散列表的存储空间是一个一维数组，散列地址是数组的下标。

冲突：某个散列函数对于不相等的关键码计算出了相同的散列地址。

负载因子：空间m,节点数n,q=n/m。

同义词：发生冲突的两个关键码。

散列函数的选取方法：除余法，平方取中法，折叠法。

除余法：用关键码x 除以M（往往取散列表长度，一般选择小于表长的最大质数），并取余数作为散列地址。

平方取中法：先通过求关键码的平方来扩大差别，再取其中的几位来作为散列地址。

变换基数：将十进制数看成13进制，然后转成10进制，然后取后几位存储。

折叠法：图书号是0-42-20586-4 用5864+4220+04 取最后几位得到0088作存储地址。或者分界叠加：5864+0224+04.

碰撞的处理：

1.开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）

2.再哈希法

3.链地址法(Java hashmap就是这么做的)

4.建立一个公共溢出区

开散列方法，也称拉链法，把发生冲突的关键码存储在散列主表之外。

闭散列法：也成开地址法。

散列的搜索时间一般是<2的。

如果整个散列表存储在内存中，开散列方法比较容易实现。

如果散列表存储在磁盘中，用开散列不合适，因为同义词元素可能会存储在不同的磁盘页中，会引起多次的磁盘访问，从而增加检索时间，可用桶式散列方法（把一个文件的记录分为若干个存储桶，每个存储桶包含一个或多个页块，一个存储桶内的各页块用指针连接起来，每个页块包含若干记录。散列函数h(k)表示具有关键码值k的记录所在的存储桶号）。

闭散列方法：当冲突发生时，使用某种方法为关键码值K生成一个散列地址序列d0,d1,d2,…dm-1。其中d0=h(k)称为K的基地址，所有di(0<i<m)是后继散列地址。形成探查的方法不同，所得到的解决冲突的方法也不同。

线性探查：依次探查下述地址单元：d+1,d+2,d+3…0,1,…d-1。

改进线性探测法：每次跳过c个槽。

二次探测：探测序列依次为1^2,-1^2,2^2,-2^2,

伪随机数探测：产生一个伪随机序列，每次都是用同样的序列。

void permute(int\* array,int n)

{

for(int i=1;i<=n;i++)

swap(array[i-1],array[Random(i)]);

}

双散列探查法：使用两个散列函数h1 和h2。若在地址h1(key)=d发生冲突，则再计算h2(key)得到的探查序列为：(d+h2(key))%M, (d+2h2(key))%M,(d+3h2(key))%M。

散列表中删除元素：开散列可以真正删除，闭散列方法要做标记，不能真正删除，删除后会使探查序列断掉。

红黑树：

性质1. 节点是红色或黑色。  
性质2. 根节点是黑色。  
性质3. 每个叶子结点都带有两个空的黑色结点（被称为黑哨兵），如果一个结点n的只有一个左孩子，那么n的右孩子是一个黑哨兵；如果结点n只有一个右孩子，那么n的左孩子是一个黑哨兵。  
性质4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)  
性质5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

http优缺点

1.支持客户/服务器模式。2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径，通信速度很快。3.灵活：HTTP允许传输任意类型的数据对象。4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。5.无状态：HTTP协议是无状态协议。无状态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，导致每次连接传送的数据量增大。缺点就是不够安全，可以使用https完成使用

traceroute www.baidu.com

通过traceroute我们可以知道信息从你的计算机到互联网另一端的主机是走的什么路径

TTL是Time To Live，每经过一个路由就会被减去一，如果它变成0，包会被丢掉。它的主要目的是防止包在有回路的网络上死转，浪费网络资源。ping和traceroute用到它。

7.epoll和select的区别？

1）select在一个进程中打开的最大fd是有限制的，由FD\_SETSIZE设置，默认值是2048。不过 epoll则没有这个限制，内存越大，fd上限越大，1G内存都能达到大约10w左右。

2）select的轮询机制是系统会去查找每个fd是否数据已准备好，当fd很多的时候，效率当然就直线下降了，epoll采用基于事件的通知方式，一旦某个fd数据就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，高效。

3）select还是epoll都需要内核把FD消息通知给用户空间，epoll是通过内核于用户空间mmap同一块内存实现的，而select则做了不必要的拷贝

4. tcp与udp的区别（必问）为什么TCP要叫做数据流？

1）．基于连接与无连接

2）．对系统资源的要求（TCP较多，UDP少）

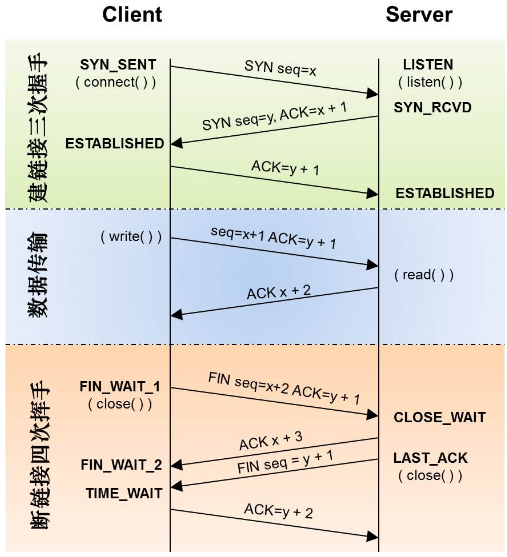
3）．UDP程序结构较简单

4）．流模式与数据报模式

5）．TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证

6）.TCP有拥塞控制和流量控制，UDP没有

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



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

TCP通过滑动窗口实现流量控制，控制发送窗口的大小。

拥塞控制：慢开始 拥塞避免 快重传 快恢复

慢开始是逐渐扩大拥塞窗口，每收到一个新的报文段的确认后 就把拥塞窗口增加一倍。拥塞避免是让拥塞窗口缓慢变大，每经过一个往返时间就加1，快重传就是受到一个失序的报文就立即发出重复确认。

Tcp的头



ip头



Linux进程调度

对于实时进程，Linux采用了两种调度策略，即FIFO(先来先服务调度)和RR（时间片轮转调度，对于普通进程，Linux采用动态优先级调度，优先级权值取决于（20-nice）和时间片剩余的counter之和

内核给进程发送信号，是在进程所在的进程表项的信号域设置对应的信号的位。进程处理信号的时机就是从内核态即将返回用户态度的时候。执行用户自定义的信号处理函数的方法很巧妙。把该函数的地址放在用户栈栈顶，进程从内核返回到用户态的时候，先弹出信号处理函数地址，于是就去执行信号处理函数了，然后再弹出，才是返回进入内核时的状态。

13、如何实现守护进程？

1）创建子进程，父进程退出

2）在子进程中创建新会话

3）改变当前目录为根目

4）重设文件权限掩码

5) 关闭文件描述符

6) 守护进程退出处理

当用户需要外部停止守护进程运行时，往往会使用 kill命令停止该守护进程。所以，守护进程中需要编码来实现kill发出的signal信号处理，达到进程的正常退出。

\_exit 函数的作用是：直接使进程停止运行，清除其使用的内存空间，并清除其在内核的各种数据结构；exit 函数则在这些基础上做了一些小动作，在执行退出之前还加了若干道工序。exit() 函数与 \_exit() 函数的最大区别在于exit()函数在调用exit 系统调用前要检查文件的打开情况，把文件缓冲区中的内容写回文件。也就是图中的“清理I/O缓冲”。

死锁的四个条件

互斥 持有并等待 不可剥夺 环形等待

避免死锁：如果能确保所有的线程都是按照相同的顺序获得锁，

设置获取锁的超时时间，

Windows检测内存泄漏

#define \_CRTDBG\_MAP\_ALLOC

#include <stdlib.h>

#include <crtdbg.h>

#include <iostream>

using namespace std;

void GetMemory(char \*p, int num)

{

    p = (char\*)malloc(sizeof(char) \* num);

}

int main(int argc,char\*\* argv)

{

    char \*str = NULL;

    GetMemory(str, 100);

    cout<<"Memory leak test!"<<endl;

    \_CrtDumpMemoryLeaks();

    return 0;

}

Linux下 valgrind 命令：valgrind --leak-check=full ./a.out

子线程循环 10 次，接着主线程循环 100 次，接着又回到子线程循环 10 次，接着再回到主线程又循环 100 次，如此循环50次，试写出代码。

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <string.h>

#define LOOP 5

int num = 0;

int parm\_0 = 0;

int parm\_1 = 1;

pthread\_mutex\_t mylock = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t qready=PTHREAD\_COND\_INITIALIZER;

void\* thread\_func(void \*arg)

{

int i, j;

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

{

pthread\_mutex\_lock(&mylock);

while(parm\_0 != num)

pthread\_cond\_wait(&qready, &mylock);

printf("thread: \n");

for(j = 0; j < 10; j++)

printf(" %d ", j);

printf("\n");

num = (num + 1) % 2;

pthread\_mutex\_unlock(&mylock);

pthread\_cond\_signal(&qready);

}

return (void\*) 0;

}

int main()

{

int i, k;

pthread\_t tid;

void \*tret;

pthread\_create(&tid, NULL, thread\_func, NULL);

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

{

pthread\_mutex\_lock(&mylock);

while(parm\_1 != num)

pthread\_cond\_wait(&qready, &mylock);

printf("main: \n");

for(k = 0; k < 10; k++)

printf(" %d", k + 100);

printf("\n");

num = (num + 1) % 2;

pthread\_mutex\_unlock(&mylock);

pthread\_cond\_signal(&qready);

}

pthread\_join(tid, &tret);

}

编写一个程序，开启3个线程，这3个线程的ID分别为A、B、C，每个线程将自己的ID在屏幕上打印10遍，要求输出结果必须按ABC的顺序显示；如：ABCABC….依次递推。

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <string.h>

#define DEBUG 1

int num = 0;

pthread\_mutex\_t mylock = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t qready=PTHREAD\_COND\_INITIALIZER;

void\* thread\_func(void \*arg)

{

int param = (int) arg;

int i;

for(i = 0; i < 10; i++)

{

pthread\_mutex\_lock(&mylock);

while(param != num)

pthread\_cond\_wait(&qready, &mylock);

printf("%c", param + 'A');

num = (num + 1) % 3;

pthread\_mutex\_unlock(&mylock);

pthread\_cond\_broadcast(&qready);

}

return (void \*)0;

}

int main()

{

int i;

pthread\_t tid[3];

void \*tret;

for(i = 0; i < 3; i++)

pthread\_create(&tid[i], NULL, thread\_func, (void \*) i);

for(i = 0; i < 3; i++)

pthread\_join(tid[i], &tret);

}

/\*

\* 有四个线程1、2、3、4。线程1的功能就是输出A，线程2的功能就是输出B，以此类推.........

\* 现在有四个文件file1, file2,file3, file4。初始都为空。现要让四个文件呈如下格式：

\* file1：A B C D A B....

\* file2：B C D A B C....

\* file3：C D A B C D....

\* file4：D A B C D A....

\*/

#include <stdio.h>

#include <pthread.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

pthread\_t threads[4];

char writer\_char[4] = {'A','B', 'C', 'D'};

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

struct file\_res{

pthread\_t \*writer; /\*当前文件可以被那个线程写入\*/

int fd; /\*文件描述符\*/

}file\_res[4] =

{

{.writer=&threads[0]},/\*file1初始化可以被线程1写入'A'\*/

{.writer=&threads[1]},/\*file2初始化可以被线程2写入'B'\*/

{.writer=&threads[2]},/\*file3初始化可以被线程3写入'C'\*/

{.writer=&threads[3]},/\*file4初始化可以被线程4写入'D'\*/

};

void \*writer\_routine(void \*arg)

{

int index = (int)arg;

int file\_index[4] ;

int can\_write\_file\_cnt = 0;

int i = 0;

int next\_index=0;

printf("thread %d is running, and will write '%c' to files\n", index, writer\_char[index]);

while(1)

{

if (0!=pthread\_mutex\_lock(&mutex))

exit(-1);

for(;;) {

/\*下面的for语句查询本线程可以写的文件id\*/

for(i=0; i<(sizeof(file\_res)/sizeof(file\_res[0])); i++) {

if (&threads[index]==file\_res[i].writer) {

file\_index[can\_write\_file\_cnt++] = i;

next\_index = index+1;

if (next\_index>3)next\_index=0;

file\_res[i].writer = &threads[next\_index];

}

}

/\*找到本线程可以写的文件id,则退出for(;;), 执行写操作\*/

if (can\_write\_file\_cnt != 0)

break;

/\*等待其他线程唤醒\*/

pthread\_cond\_wait(&cond,&mutex);

}

/\*本线程可以写的文件,执行写操作\*/

for(i=0; i<can\_write\_file\_cnt; i++) {

write(file\_res[file\_index[i]].fd, &writer\_char[index], sizeof(writer\_char[index]));

}

can\_write\_file\_cnt = 0;

/\*唤醒下一批线程\*/

pthread\_cond\_broadcast(&cond);

if (0!=pthread\_mutex\_unlock(&mutex))

exit(-1);

}

}

int main(int argc, char\* argv[])

{

int i;

for(i=0; i<4; i++)

{

char file\_name[] = "filex";

sprintf(file\_name, "file%d", i+1);

if ((file\_res[i].fd = open(file\_name, O\_RDWR|O\_CREAT|O\_TRUNC, 0666))<0)

{

printf("open %s error.\n", file\_name);

exit(-1);

}

}

for (i=0; i<(sizeof(threads)/sizeof(pthread\_t)); i++)

{

if(pthread\_create(&threads[i], NULL, writer\_routine, (void \*)i))

{

printf("create writer thread error\n");

exit(-1);

}

}

pthread\_exit(NULL);

return 0;

}

三个线程顺序打印ABC

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<string.h>

/\*

\*声明3个线程

\*/

void \*thread1\_func(void \*arg);

void \*thread2\_func(void \*arg);

void \*thread3\_func(void \*arg);

sem\_t sem1 ;

sem\_t sem2 ;

sem\_t sem3 ;

pthread\_t thread1 = 0;

pthread\_t thread2 = 0;

pthread\_t thread3 = 0;

int main(void)

{

int res = 0;

void \*thread\_return = (void \*)0;

/\*

\*创建3个信号量

\*/

res = sem\_init(&sem1, 0, 0);

if(res != 0)

{

perror("fail to create sem1\n");

exit(EXIT\_FAILURE);

}

res = sem\_init(&sem2, 0, 0);

if(res != 0)

{

perror("fail to create sem2\n");

exit(EXIT\_FAILURE);

}

res = sem\_init(&sem3, 0, 0);

if(res != 0)

{

perror("fail to create sem3\n");

exit(EXIT\_FAILURE);

}

/\*

\*创建3个线程

\*/

res = pthread\_create(&thread1, NULL, thread1\_func, (void\*)"A");

if(res != 0)

{

perror("fail to create thread1\n");

exit(EXIT\_FAILURE);

}

res = pthread\_create(&thread2, NULL, thread2\_func, (void\*)"B");

if(res != 0)

{

perror("fail to create thread2\n");

exit(EXIT\_FAILURE);

}

res = pthread\_create(&thread3, NULL, thread3\_func, (void\*)"C");

if(res != 0)

{

perror("fail to create thread3\n");

exit(EXIT\_FAILURE);

}

res = sem\_post(&sem1);

if(res != 0)

{

perror("fail to post sem1\n");

exit(EXIT\_FAILURE);

}

/\*

\*等待线程执行

\*/

printf("wait for thread to finish\n");

res = pthread\_join(thread1, &thread\_return);

if(res != 0)

{

perror("fail to jion thread1\n");

exit(EXIT\_FAILURE);

}

res = pthread\_join(thread2, &thread\_return);

if(res != 0)

{

perror("fail to jion thread2\n");

exit(EXIT\_FAILURE);

}

res = pthread\_join(thread3, &thread\_return);

if(res != 0)

{

perror("fail to jion thread3\n");

exit(EXIT\_FAILURE);

}

return 0;

}

void \*thread1\_func(void \*arg)

{

int reg = 0;

sem\_wait(&sem1);

if(reg != 0)

{

perror("fail to wait sem1\n");

exit(EXIT\_FAILURE);

}

printf("%s\n", (char \*)arg);

flush(stdout);

sem\_post(&sem2);

sem\_destroy(&sem1);

pthread\_exit(0);

}

void \*thread2\_func(void \*arg)

{

int reg = 0;

sem\_wait(&sem2);

if(reg != 0)

{

perror("fail to wait sem2\n");

exit(EXIT\_FAILURE);

}

printf("%s", (char\*)arg);

flush(stdout);

sem\_post(&sem3);

sem\_destroy(&sem2);

pthread\_exit(0);

}

void \*thread3\_func(void \*arg)

{

int reg = 0;

sem\_wait(&sem3);

if(reg != 0)

{

perror("fail to wait sem1\n");

exit(EXIT\_FAILURE);

}

printf("%s", (char\*)arg);

flush(stdout);

sem\_destroy(&sem3);

pthread\_exit(0);

}

1.多线程和多进程的区别（重点 必须从cpu调度，上下文切换，数据共享，多核cup利用率，资源占用，等等各方面回答，然后有一个问题必须会被问到：哪些东西是一个线程私有的？答案中必须包含寄存器，否则悲催）！

1）进程数据是分开的:共享复杂，需要用IPC，同步简单；多线程共享进程数据：共享简单，同步复杂

2）进程创建销毁、切换复杂，速度慢 ；线程创建销毁、切换简单，速度快

3）进程占用内存多， CPU利用率低；线程占用内存少， CPU利用率高

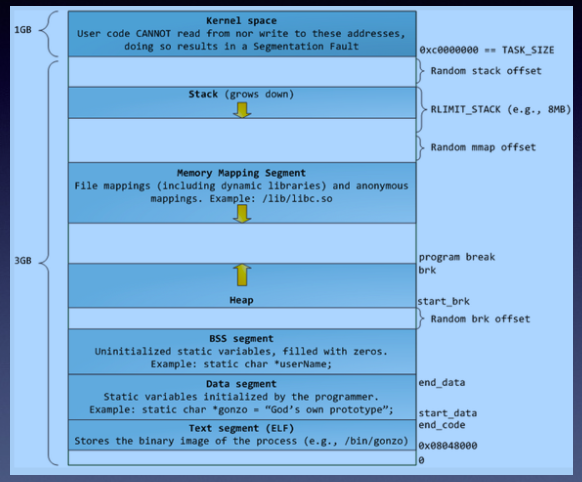
4）进程编程简单，调试简单；线程 编程复杂，调试复杂

5）进程间不会相互影响 ；线程一个线程挂掉将导致整个进程挂掉

6）进程适应于多核、多机分布；线程适用于多核

线程所私有的：

线程id、寄存器的值、栈、线程的优先级和调度策略、线程的私有数据、信号屏蔽字、errno变量、



makefile 文件 述依赖关系和编译命令(一定要以 tab 键开头)。

a.out: daemon.o make\_daemon.o work.o//左边是目的文件，右边是依赖文件。

g++ daemon.o make\_daemon.o work.o//一定要以 tab 键开头。

daemon.o: daemon.cc daemon.h

g++ -c daemon.cc

make\_daemon.o:make\_daemon.cc daemon.h

g++ -c make\_daemon.cc

work.o: work.cc

g++ -c work.cc

out : daemon.o make\_daemon.o 其中:左边的叫目的文件，:右边的叫依赖文件。

信号的实现机制：

进程的task\_struct结构中有关于本进程中未决信号的数据成员： struct sigpending pending：

struct sigpending{

struct sigqueue \*head, \*tail;

sigset\_t signal;

};

第三个成员是进程中所有未决信号集，第一、第二个成员分别指向一个sigqueue类型的结构链（称之为"未决信号信息链"）的首尾，信息链中的每个sigqueue结构刻画一个特定信号所携带的信息，并指向下一个sigqueue结构:

struct sigqueue{

struct sigqueue \*next;

siginfo\_t info;

}

内核处理一个进程收到的软中断信号是在该进程的上下文中，因此，进程必须处于运行状态。当其由于被信号唤醒或者正常调度重新获得CPU时，在其从内核空间返回到用户空间时会检测是否有信号等待处理。如果存在未决信号等待处理且该信号没有被进程阻塞，则在运行相应的信号处理函数前，进程会把信号在未决信号链中占有的结构卸掉。

信号在进程中注册指的就是信号值加入到进程的未决信号集sigset\_t signal（每个信号占用一位）中，并且信号所携带的信息被保留到未决信号信息链的某个sigqueue结构中。只要信号在进程的未决信号集中，表明进程已经知道这些信号的存在，但还没来得及处理，或者该信号被进程阻塞。

重要信号（整型）头文件<signal.h>：掌握红色信号

SIGABRT 调用abort函数产生此信号。

SIGALRM 超时信号（由alarm设置时钟）

SIGCHLD 当子进程终止时调用此信号（向父进程发送，回收子进程资源）。

SIGINFO （ctrl+T）状态键产生此信号（用此信号终止失控程序）

SIGINT （按中断键Delete or Ctrl+C时产生）（用此信号终止失控程序）

SIGIO 此信号用于表示一个异步IO事件

SIGKILL 强制杀死进程信号（无法捕获）

SIGPIPE 当读进程终止，写入管道时会产生此信号

SIGQUIT 当用户在终端退出时，产生此信号

SIGSTOP 作业控制信号，停止一个进程，不能被捕捉或忽略

SIGSTP 挂起后台进程（Ctrl+Z），终端产生此进程

SIGUSR1 用户自定义

SIGUSR2 用户自定义

SIGTERM kill进程时会收到信号。

用 旧函数名signal(信号（int），函数名（函数指针）)登记信号处理函数，

kill -USR1 2074向2074进程发送信号SIGUSR1。

在有些系统中signal()对某个符号的处理一次有效。

1 #include<iostream>

2 #include<signal.h>

3 using namespace std;

4 void func(int sig)

5 {

6 signal(sig,func);// 在有些系统中signal()对某个符号的处理一次有效,在第一次调用时再登记一次。

7 if(sig==SIGINT)

8 cout<<"Ctrl+C"<<endl;

9 else if(sig==SIGUSR1)

10 cout<<"user signal1"<<endl;

11 else if(sig==SIGUSR2)

12 cout<<"user signal2"<<endl;

13 else

14 cout<<"unknown??"<<endl;

15 }

16

17 int main()

18 {

19 cout<<"==========1============"<<endl;

20 signal(SIGINT,func);

21 signal(SIGUSR1,func);

22 signal(SIGUSR2,func);

23 cout<<"======2========="<<endl;

24 for(int i=0;i<200;i++){

25 sleep(10);

26 cerr<<'.';

27 }

28 cout<<"==================3============"<<endl;

29 }// 用kill -USR1 2074向2074进程发送信号SIGUSR1。

signal(信号，SIG\_IGN)忽略信号。signal(信号，SIG\_DFL)对信号采取缺省处理。

signal()返回SIG\_ERR表示登记失败。

测试大端还是小端

#include<stdio.h>

int main()

{

short int x;

char x0,x1;

x=0x1122;

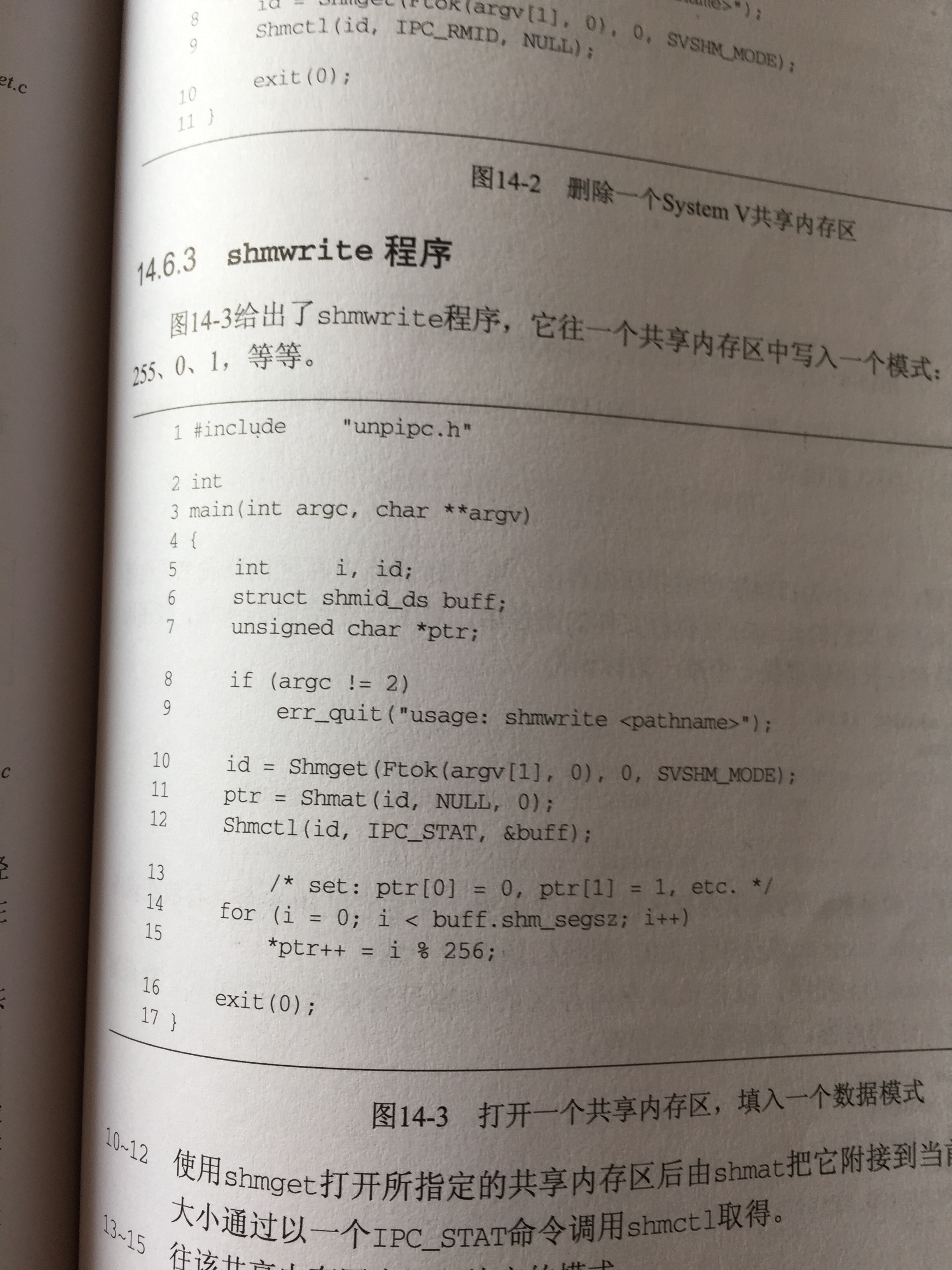
x0=((char \*)&x)[0]; //低地址单元

x1=((char \*)&x)[1]; //高地址单元

printf("x0=0x%x,x1=0x%x",x0,x1);// 若x0=0x11,则是大端; 若x0=0x22,则是小端......

return 0;

}



查看共享内存大小

cat /proc/sys/kernel/shmmax

通过直接更改 /proc 文件系统，你不需重新启动机器就可以改变 SHMMAX 的默认设置。我使用的方法是将以下命令放入 />etc/rc.local 启动文件中：

1. # echo "2147483648" **&gt;** /proc/sys/kernel/shmmax
2. # sysctl -w kernel.shmmax=2147483648

Netstat 命令用于显示各种网络相关信息，如网络连接，路由表，接口状态 (Interface Statistics)，masquerade 连接，多播成员 (Multicast Memberships) 等等。

**列出所有端口 netstat -a**

**列出所有 tcp 端口 netstat -at**

**列出所有 udp 端口 netstat -au**

**只显示监听端口 netstat -l**

**只列出所有监听 tcp 端口 netstat -lt**

tcpdump tcp port 23 and host 210.27.48.1

tcpdump udp port 123

**ipcs用法**   
ipcs -a  是默认的输出信息 打印出当前系统中所有的进程间通信方式的信息  
ipcs -m  打印出使用共享内存进行进程间通信的信息  
ipcs -q   打印出使用消息队列进行进程间通信的信息  
ipcs -s  打印出使用信号进行进程间通信的信息

ipcs -t   输出信息的详细变化时间

ipcs -p  输出ipc方式的进程ID  
ipcs -c  输出ipc方式的创建者/拥有者

**11.keepalive 是什么东西？如何使用？**

keepalive，是在TCP中一个可以检测死连接的机制。

1）.如果主机可达，对方就会响应ACK应答，就认为是存活的。

2）.如果可达，但应用程序退出，对方就发RST应答，发送TCP撤消连接。

3）.如果可达，但应用程序崩溃，对方就发FIN消息。

4）.如果对方主机不响应ack, rst，继续发送直到超时，就撤消连接。默认二个小时。

**12.socket什么情况下可读？**

 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(准备好读入的数据的字节数).

 2.连接的读一半关闭(即:接收到对方发过来的FIN的TCP连接),并且返回0;   
 3.socket收到了对方的connect请求已经完成的连接数为非0.这样的soocket处于可读状态；   
 4.异常的情况下socket的读操作将不会阻塞,并且返回一个错误(-1)。

**13.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).可以通过在已建立连接的UDP套接字上，调用connect()实现指定新的IP地址和端口号以及断开连接。

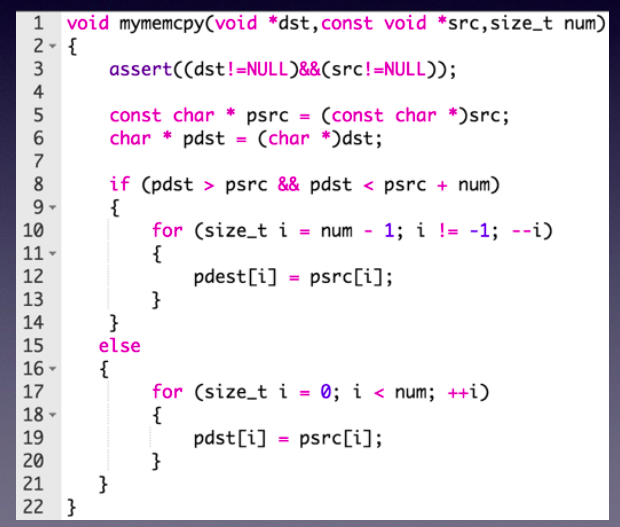
**14. socket编程，如果client断电了，服务器如何快速知道？**

使用定时器（适合有数据流动的情况）；

使用socket选项SO\_KEEPALIVE（适合没有数据流动的情况）;

1）、自己编写心跳包程序,简单的说就是自己的程序加入一条线程,定时向对端发送数据包,查看是否有ACK,根据ACK的返回情况来管理连接。此方法比较通用,一般使用业务层心跳处理,灵活可控,但改变了现有的协议;  
2）、使用TCP的keepalive机制,UNIX网络编程不推荐使用SO\_KEEPALIVE来做心）跳检测。  
keepalive原理:TCP内嵌有心跳包,以服务端为例,当server检测到超过一定时间(/proc/sys/net/ipv4/tcp\_keepalive\_time 7200 即2小时)没有数据传输,那么会向client端发送一个keepalive packet。

Memcpy



char \*strcpy(char \*dst,const char \*src) {

      assert(dst != NULL && src != NULL);

      char \*ret = dst;

      while((\* dst++ = \* src++) != '\0') ;

      return ret;

 }

char \*strcat(char \*strDes, const char \*strSrc){

assert((strDes != NULL) && (strSrc != NULL));

char \*address = strDes;

while (\*strDes != ‘\0′)

++ strDes;

while ((\*strDes ++ = \*strSrc ++) != ‘\0′)

return address;

}

char \*strncat(char \*strDes, const char \*strSrc, int count){

assert((strDes != NULL) && (strSrc != NULL));

char \*address = strDes;

while (\*strDes != ‘\0′)

++ strDes;

while (count — && \*strSrc != ‘\0′ )

\*strDes ++ = \*strSrc ++;

\*strDes = ‘\0′;

return address;

}

int strcmp(const char \*str1,const char \*str2){

    /\*不可用while(\*str1++==\*str2++)来比较，当不相等时仍会执行一次++，

    return返回的比较值实际上是下一个字符。应将++放到循环体中进行。\*/

    while(\*str1 == \*str2){

        if(\*str1 == '\0')

            return0;

        ++str1;

        ++str2;

    }

    return \*str1 - \*str2;

}

int strncmp(const char \*s, const char \*t, int count){

    assert((s != NULL) && (t != NULL));

    while (\*s && \*t && \*s == \*t && count –) {

        ++ s;

        ++ t;

    }

    return (\*s – \*t);

}

int strlen(const char \*str){

    assert(str != NULL);

    int len = 0;

    while (\*str ++ != ‘\0′)

        ++ len;

    return len;

}

strstr

class Solution { public:

char \*strStr(const char \*haystack, const char \*needle) {

// if needle is empty return the full string

if (!\*needle) return (char\*) haystack;

const char \*p1;

const char \*p2;

const char \*p1\_advance = haystack;

for (p2 = &needle[1]; \*p2; ++p2) {

p1\_advance++; // advance p1\_advance M-1 times

}

for (p1 = haystack; \*p1\_advance; p1\_advance++) {

char \*p1\_old = (char\*) p1;

p2 = needle;

while (\*p1 && \*p2 && \*p1 == \*p2) {

p1++;

p2++; }

if (!\*p2) return p1\_old;

p1 = p1\_old + 1;

}

return nullptr;

}

};

class Solution {

public:

int strStr(string haystack, string needle) {

int m = haystack.length(), n = needle.length();

if (!n) return 0;

for (int i = 0; i < m - n + 1; i++) {

int j = 0;

for (; j < n; j++)

if (haystack[i + j] != needle[j])

break;

if (j == n) return i;

}

return -1;

}

};

**31. stl各容器的实现原理（必考）**

1) Vector顺序容器，是一个动态数组，支持随机插入、删除、查找等操作，在内存中是一块连续的空间。在原有空间不够情况下自动分配空间，增加为原来的两倍。vector随机存取效率高，但是在vector插入元素，需要移动的数目多，效率低下。

注：vector动态增加大小时是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。因此，对vector空间重新配置，指向原vector的所有迭代器就都失效了。

2) Map关联容器，以键值对的形式进行存储，方便进行查找。关键词起到索引的作用，值则表示与索引相关联的数据。红黑树的结构实现，插入删除等操作都在O(logn)时间内完成。

3) Set是关联容器，set每个元素只包含一个关键字。set支持高效的关键字检查是否在set中。set也是以红黑树的结构实现，支持高效插入、删除等操作。

c++堆

#include <queue>

#include <iostream>

# include <algorithm>

using namespace std;

struct cmp{//自定义类型时的比较函数写法

bool operator() ( Node a, Node b ){

if( a.x== b.x ) return a.x> b.x;

return a.x> b.x; }

};

struct cmp

{

bool operator()(const int &a,const int &b)

{

if( a == b ) return a > b;

return a > b;//小顶堆

}

};

int main(){

int ia[9] = {0,1,2,3,4,8,9,3,5};

priority\_queue<int> ipq(ia,ia+9); //默认是从大到小排列

cout<<"size = "<<ipq.size()<<endl;

priority\_queue< int, vector<int>, cmp > qu;

qu.push(0);

qu.push(1);

qu.push(2);

qu.push(8);

qu.push(6);

qu.push(4);

while(!qu.empty()){

cout<<qu.top()<<' ';

qu.pop();

}

cout<<endl;

for(int i = 0; i < ipq.size(); i++)

cout<<ipq.top()<<' ';

cout<<endl;

while(!ipq.empty()){

cout<<ipq.top()<<' ';

ipq.pop();

}

cout<<endl;

return 0;

}

c++哈希

#include <iostream>

#include <string>

#include <unordered\_map>

#include <algorithm>

using namespace std;

int main()

{

unordered\_map<string, int> months;

months["january"] = 31;

months["february"] = 28;

months["march"] = 31;

months["april"] = 30;

cout << "april -> " << months["april"] <<endl;

unordered\_map<string,int>::const\_iterator got = months.find ("march");

if ( got == months.end() )

cout << "not found";

else

cout << got->first << " is " << got->second<<endl;

return 0;

}

int singleNumber(int A[], int n) {

unordered\_map<int,int> map;

unordered\_map<int,int> ::const\_iterator it;

for (int i = 0; i < n; i++){

it = map.find(A[i]);

if (it != map.end()){

map.erase(it);

}else{

map[A[i]] = i;

}

}

return A[map.begin()->second];

}

二叉树的非递归后序遍历

//非递归后序遍历二叉树或其子树。

{

using std::stack;//使用STL栈部分

StackElement<T> element;

stack<StackElement<T>>aStack;//栈声明

BinaryTreeNode<T>\* pointer;

if(root==NULL)

return;//空树即返回。

pointer=root;

while(true){//进入左子树

while(pointer!=NULL){

element.pointer=pointer;

element.tag=Left;

// visit(pointer->value());//前序周游时用。

aStack.push(element);

//沿左子树方向向下周游

pointer=pointer->leftchild();

}

//托出栈顶元素

element=aStack.top();

aStack.pop();

pointer=element.pointer;

//从右子树回来

while(element.tag==Right){

visit(pointer->value());//访问当前结点

if(aStack.empty())

return;

else{

element=aStack.top();

aStack.pop();//弹栈

pointer=element.pointer;

}//end else

}//end while

//从左子树回来

// visit(pointer->value());//中序周游时用。

element.tag=Right;

aStack.push(element);//如果要访问右子树的左子树则将出栈的父节点重新入栈。

//转向访问右子树

pointer=pointer->rightchild();

}//end while

}

用后续的框架可以实现前序和中序，只是Visit()函数的位置不同。

void postOrder2(BinTree \*root) //非递归后序遍历  
{  
 stack<BTNode\*> s;  
 BinTree \*p=root;  
 BTNode \*temp;  
 while(p!=NULL||!s.empty())  
 {  
 while(p!=NULL) //沿左子树一直往下搜索，直至出现没有左子树的结点   
 {  
 BTNode \*btn=(BTNode \*)malloc(sizeof(BTNode));  
 btn->btnode=p;  
 btn->isFirst=true;  
 s.push(btn);  
 p=p->lchild;  
 }  
 if(!s.empty())  
 {  
 temp=s.top();  
 s.pop();  
 if(temp->isFirst==true) //表示是第一次出现在栈顶   
 {  
 temp->isFirst=false;  
 s.push(temp);  
 p=temp->btnode->rchild;   
 }  
 else //第二次出现在栈顶   
 {  
 cout<<temp->btnode->data<<" ";  
 p=NULL;  
 }  
 }  
 }   
}

void postOrder3(BinTree \*root) //非递归后序遍历  
{  
 stack<BinTree\*> s;  
 BinTree \*cur; //当前结点   
 BinTree \*pre=NULL; //前一次访问的结点   
 s.push(root);  
 while(!s.empty())  
 {  
 cur=s.top();  
 if((cur->lchild==NULL&&cur->rchild==NULL)||  
 (pre!=NULL&&(pre==cur->lchild||pre==cur->rchild)))  
 {  
 cout<<cur->data<<" "; //如果当前结点没有孩子结点或者孩子节点都已被访问过   
 s.pop();  
 pre=cur;   
 }  
 else  
 {  
 if(cur->rchild!=NULL)  
 s.push(cur->rchild);  
 if(cur->lchild!=NULL)   
 s.push(cur->lchild);  
 }  
 }   
}

#include<iostream>

using namespace std;

void max\_min(int\* a,int i,int j,int &max,int &min)

{

if(i==j){

max=a[i];

min=a[i];

}

else if(i==j-1){

max=(a[i]>a[j]?a[i]:a[j]);

min=(a[i]<a[j]?a[i]:a[j]);

}

else{

int mid =i+(j-i)/2;

int max1,min1;

max\_min(a,i,mid,max,min);

max\_min(a,mid+1,j,max1,min1);

if(max<max1) max=max1;

if(min>min1)min=min1;

}

}

int main()

{

int a[20]={1,2,3,4,5,6,7,8,34,23,56,78,32,12,13,15,36,45,65,89};

int max,min;

max\_min(a,0,19,max,min);

cout<<"max is:"<<max<<" min is:"<<min<<endl;

}

int **Partition** (Type a[], int m, int r)

{

int i = m, j = r+1; //考虑这里为什么+1？

Type x=a[m];

// 将< x的元素交换到左边区域

// 将> x的元素交换到右边区域

while (true)

{

while (a[++i] <x);

while (a[- -j] >x);

if (i >= j) break; //退出循环

Swap(a[i], a[j]);//交换两个元素

}

a[m] = a[j];

a[j] = x;

return j;

}

void **QuickSort** (Type a[], int p, int r)

{

if (p<r) {

int q=Partition(a,p,r);

QuickSort (a,p,q-1); //对左半段排序

QuickSort (a,q+1,r); //对右半段排序

}

}

删除字符串s中出现的所有字符c。

/\* squeeze: delete all c from s \*/

void squeeze(char s[], int c)

{

int i, j;

for (i = j = 0; s[i] != '\0'; i++)

if (s[i] != c)

s[j++] = s[i];

s[j] = '\0';

}

删除连续的重复字符：

#include<iostream>

using namespace std;

void remove\_elem(char\* s)

{

int i=0,j=0;

while(s[i]!='\0'){

if(s[i]==s[j])

s[j]=s[i++];

else{

++j;

s[j]=s[i++];

}

}

s[++j]='\0';

}

int main()

{

char a[17]="kooooooosddddedd";

remove\_elem(a);

int i=0;

while(a[i]!='\0')

cout<<a[i++]<<' ';

cout<<endl;

}

速度快的memcpy：

void \*my\_memcpy(void \*dst, void\* src, int size)  
{  
    int len = sizeof(long);  
    long \*pdst = (long\*)dst;  
    long \*psrc = (long\*)src;  
    for (; size >= len; ++pdst, ++psrc, size -= len)  
    {     
        \*pdst = \*psrc;  
    }     
    for (len = size, size = 0; size < len; ++size)  
    {     
        \*((char\*)pdst + size) = \*((char\*)psrc + size);  
    }     
    return dst;  
}  
int main()  
{  
    char src[] = "hello", dst[10] = {'\0'};  
    my\_memcpy(dst, src, 5);   
    printf("dst = %s\n", dst);  
    return 0;  
}

将整型数组的元素左移m位，不能申请数组空间，不允许采用每次左移：

#include<iostream>

using namespace std;

void swap(int\* a,int\* b)

{

int x;

x=\*a;

\*a=\*b;

\*b=x;

}

void move\_array(int data[],int n,int m)

{

int i,j;

m=m%n;

if(m==0) return;

for(i=0;i<n-m;i+=m)

{

for(j=i;j<i+m&&j<n-m;j++)

{

swap(&data[j],&data[j+m]);

}

}

move\_array(data+n-m,m,m-n%m);

}

[#面试题#](http://huati.weibo.com/k/é¢è¯é¢?from=501)给定一个无序的整数数组，怎么找到第一个大于0，并且不在此数组的最小整数。比如[1,2,0] 返回 3, [3,4,-1,1] 返回 2。最好能O(1)空间和O(n)时间。

来源[http://weibo.com/lirenchen](http://weibo.com/lirenchen?from=otherprofile&wvr=4&loc=infweihao)。

解法：

直接考虑这个问题是比较困难的。不如先换个简单的问题。

**问题1**

给定一个数组，长度为n，除a[0]以外，其他元素都是a[i] == i。那么请找出第一个大于0，且不在此数组中的最小整数。

答案就是:  n + (a[0] == n);

**问题2**

给定一个数组，长度为n，某几个位置的元素满足 a[x] < 0 或者 a[x] > n，余下的元素都满足a[i] == i。那么请找出第一个大于0，且不在此数组中的最小整数。

答：

很明显：当x的值限定为0时。就变成了问题1。

除此之外，这个问题的解也容易求得。

情况1： 首先从1~n开始扫描，当发现i != a[i]时，直接返回i。此时的i必定是最小的未出现的整数。

情况2： 当扫描完1~n之后，那么就回归至问题1了。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ju136/article/details/8153274)

1. for (i = 1; i < n; ++i) {
2. if (i != a[i]) return i;
3. }
4. return (n + (n == a[0]));

for (i = 1; i < n; ++i) {

if (i != a[i]) return i;

}

return (n + (n == a[0]));

**问题3**

当给出问题2之后，就需要考虑如何把原题目转换成为问题2了。

实际上要完成的任务就变成了，如何把一个数组中的元素**元神归位**。也就是让a[i] == i。如果能让这些元素元神归位，那么就转换成为问题2了。就很容易求解了。

算法如下：

       我们从后往前扫描，i = n - 1 to 0;

                  step1如果发现a[i] < 0 || a[i] > n; 则 continue;

                  step2 如果发现a[i] == a[a[i]]; 则continue;

                  step3 如果发现0 < a[i] < n

                          说明需要将a[i]元神归位。也就是放到a[a[i]]上去。

                           swap(a[i], a[a[i]]).

                           再跳转至step2。

Note: 需要注意一种有重复数的情况，比如a[11] = 2, a[2] = 2。这时候，就不用进行交换了。直接处理下一个元素。

可以给出代码：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ju136/article/details/8153274)

1. **int** i = n, t, temp;
2. if (!a || n <= 0) return -1;
4. while ((--i) >= 0) {
5. while (0 < a[i] && a[i] < n && i != a[i]) {
6. t = a[i];
7. if (a[i] == a[t]) break;
8. temp = a[i];
9. a[i] = a[t];
10. a[t] = temp;
11. }
12. }

int i = n, t, temp;

if (!a || n <= 0) return -1;

while ((--i) >= 0) {

while (0 < a[i] && a[i] < n && i != a[i]) {

t = a[i];

if (a[i] == a[t]) break;

temp = a[i];

a[i] = a[t];

a[t] = temp;

}

}

好吧，到现在为止，原题已经变得很简单了。可以直接给出代码了。

**原题解答**

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ju136/article/details/8153274)

1. **int** find(**int** \*a, **int** n) {
2. **int** i = n, t, temp;
3. if (!a || n <= 0) return -1;
5. while ((--i) >= 0) {
6. while (0 < a[i] && a[i] < n && i != a[i]) {
7. t = a[i];
8. if (a[i] == a[t]) break;
9. temp = a[i];
10. a[i] = a[t];
11. a[t] = temp;
12. }
13. }
15. for (i = 1; i < n; ++i)
16. if (a[i] != i) return i;
18. return (n + (a[0] == n));
19. }
20. 散列基本思想：1确定一个函数关系。2以节点的关键码k为自变量。3函数值h(k)作为节点的存储地址。4检索时也是根据这个函数计算其存储位置。
21. 通常散列表的存储空间是一个一维数组，散列地址是数组的下标。
22. 冲突：某个散列函数对于不相等的关键码计算出了相同的散列地址。
23. 负载因子：空间m,节点数n,q=n/m。
24. 同义词：发生冲突的两个关键码。
25. 散列函数的选取方法：除余法，平方取中法，折叠法。
26. 除余法：用关键码x 除以M（往往取散列表长度，一般选择小于表长的最大质数），并取余数作为散列地址。
27. 平方取中法：先通过求关键码的平方来扩大差别，再取其中的几位来作为散列地址。
28. 变换基数：将十进制数看成13进制，然后转成10进制，然后取后几位存储。
29. 折叠法：图书号是0-42-20586-4 用5864+4220+04 取最后几位得到0088作存储地址。或者分界叠加：5864+0224+04.
30. ELFhash字符串散列函数。可用于搜索引擎URL的散列。
31. int ELFhash(char\* key)
32. {
33. unsigned long h=0;
34. while(\*key){
35. h=(h<<4)+\*key++;
36. unsigned long g=h&0xF000000L;
37. if(g) h^=g>>24;
38. h&=~g;
39. }
40. return h%M;
41. }
42. 碰撞的处理：
43. 开散列方法，也称拉链法，把发生冲突的关键码存储在散列主表之外。
44. 闭散列法：也成开地址法。
45. 散列的搜索时间一般是<2的。
46. 如果整个散列表存储在内存中，开散列方法比较容易实现。
47. 如果散列表存储在磁盘中，用开散列不合适，因为同义词元素可能会存储在不同的磁盘页中，会引起多次的磁盘访问，从而增加检索时间，可用桶式散列方法（把一个文件的记录分为若干个存储桶，每个存储桶包含一个或多个页块，一个存储桶内的各页块用指针连接起来，每个页块包含若干记录。散列函数h(k)表示具有关键码值k的记录所在的存储桶号）。
48. 闭散列方法：当冲突发生时，使用某种方法为关键码值K生成一个散列地址序列d0,d1,d2,…dm-1。其中d0=h(k)称为K的基地址，所有di(0<i<m)是后继散列地址。形成探查的方法不同，所得到的解决冲突的方法也不同。
49. 线性探查：依次探查下述地址单元：d+1,d+2,d+3…0,1,…d-1。
50. 改进线性探测法：每次跳过c个槽。
51. 二次探测：探测序列依次为1^2,-1^2,2^2,-2^2,
52. 伪随机数探测：产生一个伪随机序列，每次都是用同样的序列。
53. void permute(int\* array,int n)
54. {
55. for(int i=1;i<=n;i++)
56. swap(array[i-1],array[Random(i)]);
57. }
58. 双散列探查法：使用两个散列函数h1 和h2。若在地址h1(key)=d发生冲突，则再计算h2(key)得到的探查序列为：(d+h2(key))%M, (d+2h2(key))%M,(d+3h2(key))%M。
59. 散列表中删除元素：开散列可以真正删除，闭散列方法要做标记，不能真正删除，删除后会使探查序列断掉。
60. 索引是把一个关键码与它对应的数据记录的位置相关联的过程。索引文件用于记录这种联系的文件组织结构。索引文件的记录：关键码和指针对。指针指向主要数据库文件中的完整记录。二级索引是一级索引的索引。
61. B树是一种平衡的多分树，有5个特点：
62. 1每个节点至多有m个子节点。
63. 2除根节点和叶节点外，其他每个节点至少有m/2取上整个子节点。
64. 3根节点至少有两个子节点。唯一例外的是根节点就是叶节点时，没有子节点，此时B树只包含一个节点。
65. 4所有的叶节点在同一层。
66. 5有K个子节点的非根节点恰好包含k-1个关键码。
67. B树的性质：1树高平衡，所有节点都在同一层。2关键码没有重复，父节点中的关键码是其子节点的分界。3B树把（值接近）相关记录放在同一个磁盘页中，从而利用了访问局部性原理。4B树保证树中至少有一定比例的节点是满的，这样能够改进空间的利用率，能减少检索和更新操作的磁盘读取数目。
68. B树的查找：在两个关键码之间有一个指针，指向关键码值介于这两个关键码之间的关键码值。每一个关键码都包含一个指针，这个指针指向关键码在主文件中的位置。
69. B树的插入：注意保持性质，特别是等高和阶的限制，1找到最底层，插入。2若溢出，则节点分裂，中间关键码连同新指针插入父节点。3若父节点也溢出，则继续分裂。4分裂过程可能传达到根节点（则树升高一层）。
70. B树的删除:删除的关键码不在叶节点层，先把此关键码与它在B树里的后继兑换位置(找左边子树的最大值或右边子树的最小值交换)，然后再删除该关键码。
71. 如果删除的关键码在叶节点层：
72. 删除后关键码个数不小于m/2-1，则直接删除。
73. 如果关键码值小于m/2-1，则：
74. 如果兄弟节点关键码值个数不等于m/2-1,则从兄弟节点中移若干个关键码到该节点中来，父节点中的一个关键码要做相应的变化。
75. 如果兄弟节点关键码个数等于m/2-1，则合并。同时将父节点中的关键码拉下来。
76. B+树： 1每个节点至多有m个子节点，2每个节点（除根外）至少有m/2个子节点。3根节点至少有两个子节点。4有k个子节点的节点必有k个关键码。在叶节点层是所有关键码的全集。节点中的关键码是子节点中最大(最小)关键码的复写。
77. B+树的查找：应该到叶节点层，在上层已找到待查的关键码，并不停止，而是继续沿指针向下一直查到叶节点层的这个关键码。B+树的叶节点一般链接起来，形成一个双链表。B+树的应用更广泛。B+树的插入：插入到底层，如果溢出，就分裂，然后将分裂后节点的最大元素写入父节点，如果父节点也溢出，则继续分裂。B+树的删除：从父节点找到大于要删除节点的最小关键码，然后一直查找到叶节点。然后删除，如果要删除的节点在父节点中的有副本，则不必删除副本。动态索引使用B树和B+树。
78. B树的性能分析：包含N个关键码的B树有N+1个外部空指针。
79. 中科院出的切词包，能将一段话分成词语，ICTCLAS。将源代码中的六个文件夹中的内容添加到自己的工程中，运行后将错误提示的头文件包含进来。直到没有错误为止。
80. Trie结构主要应用于信息检索，用来存储英文字符串，尤其是大规模的英文字典。
81. Trie结构是非平衡的，PATRICIA结构是平衡的，每个内部节点都代表一个位的比较，必然产生两个子节点，所以它是满二叉树。进行一次检索，最多只需要关键码位数次的比较即可。
82. AVL树：如果T是一棵AVL树，则它的左右子树也是AVL树，并且|hl-hr|<=1。左右子树的高度之差是平衡因子，平衡因子是0,1，-1。

int find(int \*a, int n) {

int i = n, t, temp;

if (!a || n <= 0) return -1;

while ((--i) >= 0) {

while (0 < a[i] && a[i] < n && i != a[i]) {

t = a[i];

if (a[i] == a[t]) break;

temp = a[i];

a[i] = a[t];

a[t] = temp;

}

}

for (i = 1; i < n; ++i)

if (a[i] != i) return i;

return (n + (a[0] == n));

}

折半查找的迭代版

int bsearch(int \*a,int x,int y,int t){

int m;

while(x<y){

m = x+(y-x)/2;

if(a[m] == t)

return m;

else if(a[m] > t)

y=m;

else

x=m+1;

}

return -1;

}

链表检测是否有环

bool hasCycle(ListNode \*head) {

if(head == NULL)

return false;

ListNode \*pre = head;

ListNode \*cur = head;

while(cur != NULL){

pre = pre->next;

cur = cur->next;

if(cur != NULL)

cur = cur->next;

else

return false;

if(cur == pre)

return true;

}

return false;

}

检测链表环的起始位置

ListNode \*detectCycle(ListNode \*head) {

if(head == NULL)

return NULL;

ListNode \*pre = head;

ListNode \*cur = head;

while(cur != NULL){

pre = pre->next;

cur = cur->next;

if(cur != NULL)

cur = cur->next;

else

return NULL;

if(cur == pre){

cur = head;

while(pre != cur){

pre = pre->next;

cur = cur->next;

}

return cur;

}

}

return NULL;

}

合并两个有序的链表

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode \*p;

ListNode \*s;

ListNode \*pre;

ListNode \*beh;

ListNode \*result;

if(l1 == NULL)

return l2;

else if(l2 == NULL)

return l1;

else if(l1->val <= l2->val){

pre = l1;

p=l1;

s=l2;

result = l1;

beh = s->next;

}else if(l1->val > l2->val){

pre = l2;

p=l2;

s=l1;

result = l2;

beh = s->next;

}

p = p->next;

while(p !=NULL && s !=NULL){

if(p->val < s->val){

p = p->next;

pre = pre->next;

}else{

pre->next = s;

s->next = p;

pre = pre->next;

if(beh != NULL){

s = beh;

beh = beh->next;

}else

break;

}

}

if(p == NULL){

pre->next = s;

}

return result;

}

链表反转 pre是反转起始节点的前一个节点。

void reverse(ListNode \*pre,ListNode \*begin,ListNode \*end){

ListNode \*end\_next = end->next;

ListNode \*cur = begin;

ListNode \*beh = begin->next;

while(beh != end\_next ){

ListNode \*next;

if(beh->next != NULL)

next = beh->next;

else

next = NULL;

beh->next = cur;

cur = beh;

beh = next;

}

pre->next = end;

begin->next = end\_next;

}

单链表归并排序

ListNode \*sortList(ListNode \*head) {

if (head == NULL || head->next == NULL)return head;

ListNode \*fast = head, \*slow = head;

while (fast->next != NULL && fast->next->next != NULL) {

fast = fast->next->next;

slow = slow->next;

}

fast = slow;

slow = slow->next;

fast->next = NULL;

ListNode \*l1 = sortList(head);

ListNode \*l2 = sortList(slow);

return mergeTwoLists(l1, l2);

}

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode dummy(-1);

for (ListNode\* p = &dummy; l1 != nullptr || l2 != nullptr; p = p->next) {

int val1 = l1 == nullptr ? INT\_MAX : l1->val;

int val2 = l2 == nullptr ? INT\_MAX : l2->val;

if (val1 <= val2) {

p->next = l1;

l1 = l1->next;

} else {

p->next = l2;

l2 = l2->next;

}

}

return dummy.next;

}

http://www.zhihu.com/question/28431865