

 anonymous_1

面试常见手撕代码题

584 浏览 | 0 回复 | 2019-10-06



anonymous_1

[+关注](#)

常见排序算法的实现

堆排序

<https://www.jianshu.com/p/11655047ab58>

(升序排序) 整体思路是：

1. 初始数组表示一个未排序的原始堆
2. 构造一个最大堆
3. 依次将堆顶元素交换到堆尾，调整使其保持最大堆，堆尾指针前移，直到所有元素有序（堆中只剩下堆顶元素）

快速排序

快排的核心部分在于partition的实现。partition需要找到一个元素的正确位置，将其移动到这个位置，并返回这个位置。

归并排序

<https://www.cnblogs.com/chengxiao/p/6194356.html>

归并排序的核心在于分治。分就是把原数组一分为二，然后再分，直到每一部分只剩下最多一个元素；并就是把结果合并，当每一部分最多有一个元素的时候可以直接交换，而多于一个元素的时候可以将有序的两部分从头到尾比较排序。

在合并的时候会用到额外的空间，因此可以在一开始就定义一块和原数组相同大小的空间用于合并，避免递归过程中频繁开辟新的空间。

链表

反转链表（递归和非递归）

<https://leetcode-cn.com/problems/reverse-linked-list/>

 anonymous_1

```
1 public ListNode reverseList(ListNode head) {
2     ListNode newHead = null;
3     while(head!=null){
4         ListNode t = head.next;
5         head.next = newHead;
6         newHead = head;
7         head = t;
8     }
9     return newHead;
10 }
```

递归（头插法）

```
1 public ListNode reverseList(ListNode head) {
2     return reverse(null, head);
3 }
4
5 public ListNode reverse(ListNode newHead, ListNode head) {
6     if(head==null){
7         return newHead;
8     }
9     else{
10         ListNode t = head.next;
11         head.next = newHead;
12         newHead = head;
13         return reverse(newHead, t);
14     }
15 }
```

倒数第K个节点

使用双指针的时候要注意链表的长度，要考虑第二个指针在第一次前进k个节点时出现null的情况。

检测环

使用快慢指针。

引申问题1：为什么快指针的步长是2，而不是3、4、5...？

<https://blog.csdn.net/xgjonathan/article/details/18034825>

结论：

1. 无论步长为多少，快慢指针在环里都能相遇
2. 步长设置为2能最快确定有环

引申问题2：如何确定环的入口？

 anonymous_1

当慢指针刚好进入环中，也就是慢指针走了 s 步之后，快指针走了 $2s$ 步，所以快指针在环中走了 $2s - s = s$ 步；

由于存在 $s > cl$ 的情况，我们记快指针超出 ks (k 为自然数) 的距离是 $s \% cl$ ；

此时，快指针需要追及慢指针的距离是 $cl - s \% cl$ ；

因此，当慢指针在环中走了 $cl - s \% cl$ 步后，快指针追上了慢指针；

所以，相遇之后的慢指针距离 ks 的距离是 $cl - (cl - s \% cl) = s \% cl$ 。因为有环的存在，我们可以把这个距离看成 $s + M * cl$ ， M 是正整数。所以相遇时的慢指针距离环的起始结点 ks 是 s 。这时，我们再设置另一个指针从单向链表的头开始，以步长为 1 移动，移动 s 步后相遇，而这个相遇结点正好就是环的起始结点。

二叉树

二叉树的递归遍历以及（前中后序）非递归遍历

<https://www.cnblogs.com/dolphin0520/archive/2011/08/25/2153720.html>

二叉树的层序遍历（使用/不使用额外的数据结构）

https://blog.csdn.net/m0_37925202/article/details/80796010

图（矩阵）的深度优先和广度优先遍历

<https://blog.csdn.net/jeffleo/article/details/53309286>

动态规划

最长公共子串

<https://segmentfault.com/a/1190000002641054>

最长递增子序列

<https://blog.csdn.net/u013178472/article/details/54926531>

最长回文串

 anonymous_1

实现一个LRU的Cache

设计一个类LRUCache，满足以下条件：

1. 这个类的构造函数应该传入一个int类型的参数size，表示最多可容纳多少个元素
2. 包含方法put、get。put是放入新元素，get是获取某一个元素
3. 当LRUCache满了之后删除最近最久未使用的元素。

1. 使用Java自带的LinkedHashMap

这种方法比较简单，就是维护一个LinkedHashMap，重写它的removeEldestEntry方法。

LinkedHashMap的三个构造参数分别表示初始大小、扩张因子、使用accessOrder排序。如果不指定第三个参数，则会按照添加的顺序排序。

```
1 public class LRUCache {
2     private LinkedHashMap<String,String> cache;
3     private int maxsize;
4     public LRUCache(int maxsize){
5         this.maxsize = maxsize;
6         this.cache = new LinkedHashMap<String,String>(16,0.75f,true){
7             <a href="/profile/992988" data-card-uid="992988" class="js-nc-card" target=
8                 protected boolean removeEldestEntry(Map.Entry<String,String> eldest) {
9                     return this.size()>maxsize;
10                }
11        };
12    }
13    public void put(String key,String value){
14        cache.put(key,value);
15    }
16    public String get(String key){
17        return cache.get(key);
18    }
19 }</a>
```

2. 仅使用Java中的HashMap和Deque

本质上LRUCache就是一个HashMap，只不过需要使元素之间维持一定的顺序。LinkedHashMap实现了这个功能。如果不使用LinkedHashMap的话，则需要使用与其类似的思路，使用HashMap存储键值对，然后使用一个双端队列来记录访问顺序。这样效率很低，因为查询的时候需要先从队列中找到对应的key，完成一系列出队、入队操作最后再将这个key入队。

```
1 public class LRUCache {
2     private HashMap<String,String> cache;
3     private Deque<String> orderQueue;
4     private int maxsize;
```

anonymous_1

```
8         this.orderQueue = new ArrayDeque<>();
9         this.cache = new HashMap<>();
10    }
11    public void put(String key,String value){
12        if(cache.size()>=maxsize){
13            String eldestKey = orderQueue.poll();
14            cache.remove(eldestKey);
15        }
16        cache.put(key,value);
17        orderQueue.offer(key);
18    }
19    public String get(String key){
20        if(cache.size()<1||!cache.containsKey(key)){
21            return null;
22        }
23        Deque<String> stack = new ArrayDeque<String>();
24        String e = null;
25        while(!(e=orderQueue.poll()).equals(key)){
26            stack.push(e);
27        }
28        while(!stack.isEmpty()){
29            orderQueue.offerFirst(stack.pop());
30        }
31        orderQueue.offer(e);
32        return cache.get(key);
33    }
34 }
```

实现一个阻塞队列

阻塞队列的实现需要保证多线程围绕队满/队空这两个条件来协作。当队满时，添加元素的线程应当阻塞；当队空时，获取元素的线程应当阻塞。

```
1 public class MyBlockingQueue<T>{
2     private volatile List<T> queue;
3     private int size;
4
5     public MyBlockingQueue(int size){
6         queue = new ArrayList<>();
7         this.size = size;
8     }
9
10    public synchronized void put(T element) throws InterruptedException {
11        while(queue.size()>=size){
12            wait();
13        }
14        if(queue.size()==0){
15            // 如果为0，则可能有其他线程在阻塞get，因此调用notifyAll
16            notifyAll();
17        }
18        queue.add(element);
19    }
```

 anonymous_1

```
23     }
24     if(queue.size()>=size){
25         // 如果为size, 则可能有其他线程在阻塞put, 因此调用notifyAll
26         notifyAll();
27     }
28     return queue.remove(0);
29 }
30 }
```

实现生产者/消费者模型

使用上面的阻塞队列实现:

```
1  public class ProducerConsumer{
2      private static MyBlockingQueue<Integer> queue;
3      static class Producer extends Thread{
4          private MyBlockingQueue<Integer> queue;
5
6          public Producer(MyBlockingQueue<Integer> queue){
7              super();
8              this.queue = queue;
9          }
10
11          <a href="/profile/992988" data-card-uid="992988" class="js-nc-card" target="_
12          public void run(){
13              Random rnd = new Random();
14              for(int i=0;i<100;i++) {
15                  try {
16                      queue.put(i);
17                      System.out.println("Producing "+i);
18                  } catch (InterruptedException e) {
19                      e.printStackTrace();
20                  }
21              }
22          }
23      }
24      static class Consumer extends Thread{
25          private MyBlockingQueue queue;
26
27          public Consumer(MyBlockingQueue<Integer> queue){
28              super();
29              this.queue = queue;
30          }
31
32          </a><a href="/profile/992988" data-card-uid="992988" class="js-nc-card" targe
33          public void run(){
34              while(true){
35                  try {
36                      System.out.println("Consuming "+queue.get());
37                  } catch (InterruptedException e) {
38                      e.printStackTrace();
39                  }

```

☰ anonymous_1

```

43     public static void main(String[] args){
44         MyBlockingQueue<Integer> queue = new MyBlockingQueue<>(10);
45         Producer p = new Producer(queue);
46         Consumer c = new Consumer(queue);
47         p.start();
48         c.start();
49     }
50 }
51 </a>

```

实现两个线程交替打印

两个线程围绕一个合作变量flag进行合作。

需要将flag声明为volatile，否则程序会卡住，这跟JMM有关系。理论上说，每次工作线程修改了flag都迟早会同步到主内存，但是如果while循环体为空的话，**访问flag会太频繁导致JVM来不及将修改后的值同步到主内存**，这样一来程序就会一直卡在一个位置。

如果不使用volatile也是可以的，将循环体里面加上一句 `System.out.print("")` 就行了，这样可以降低访问flag的频率，从而使JVM有空将工作内存中的flag和主内存中的flag进行同步。

```

1  public class PrintAlternately {
2      static volatile int flag = 0;
3      static class EvenThread extends Thread{
4          <a href="/profile/992988" data-card-uid="992988" class="js-nc-card" target="_
5              public void run(){
6                  for(int i=0;i<101;i+=2){
7                      while(flag!=0){}
8                      System.out.println(i);
9                      flag = 1;
10                 }
11             }
12         }
13         static class OddThread extends Thread{
14             </a><a href="/profile/992988" data-card-uid="992988" class="js-nc-card" targe
15                 public void run(){
16                     for(int i=1;i<100;i+=2){
17                         while(flag!=1){}
18                         System.out.println(i);
19                         flag = 0;
20                     }
21                 }
22             }
23         public static void main(String[] args){
24             EvenThread e = new EvenThread();
25             OddThread o = new OddThread();
26             e.start();
27             o.start();
28         }
29     }</a>

```

 anonymous_1

文件的输入和输出

最长见的方法是使用BufferedReader逐行（字符流）读取文件，使用FileWriter或者BufferedWriter（字符流）写文件，区别是后者提供了方法newLine来写换行符，而前者需要手动写入 `\n` 或者 `\r\n` 来换行。

如果要求写文件的时候是在文件末尾添加，而不是整体覆盖，则FileWriter构造函数需要传入第二个参数true（表示使用append模式）。

```
1 public class WriterReader {
2     public static void main(String[] args){
3         try(BufferedReader reader = new BufferedReader(new FileReader("from.txt"));
4             BufferedWriter writer = new BufferedWriter(new FileWriter("to.txt"))){
5             String line = null;
6             while((line=reader.readLine())!=null){
7                 writer.write(line);
8                 writer.newLine();
9             }
10        } catch (Exception e) {
11            e.printStackTrace();
12        }
13    }
14 }
```

字符串转整型

思路：应该尽快调试出来一个转化的程序，然后再考虑各种情况。面试时手撕代码时间非常有限，所以优先使代码能工作，而不是所有情况都考虑全了，发现时间不够或者疯狂调bug。

先是一个合法字符串转正整数的方法：

```
1 public int transform(String str){
2     int res = 0;
3     int base = 1;
4     for(int i=str.length()-1;i>-1;i--){
5         char ch = str.charAt(i);
6         res = res + base*(ch-'0');
7         base *= 10;
8     }
9     return res;
10 }
```

然后再考虑下面这些情况：

- 正负号
- 小数点
- 溢出

 anonymous_1

```
1 public class Solution {
2     public static int transform(String str) throws Exception{
3         String input = str;
4
5         // 先判断是否合法
6         if(!valid(str)){
7             throw new Exception("Invalid input string!");
8         }
9
10        // 判断正负
11        boolean isNegative = false;
12        if(str.charAt(0)=='-'){
13            isNegative = true;
14            input = input.substring(1,input.length());
15        }
16
17        // 去掉前面的0
18        int t = 0;
19        while(input.charAt(t)=='0'){
20            t += 1;
21            if(t>input.length()-1){
22                return 0;
23            }
24        }
25        input = input.substring(t, input.length());
26
27        // 去掉小数点
28        int pointIndex = getPointIndex(input);
29        int extra = 0;
30        if(pointIndex>-1){
31            if(pointIndex==input.length()-1){
32                input = input.substring(0, input.length()-1);
33            }else if(input.charAt(pointIndex+1)>'4'){
34                extra = 1;
35                input = input.substring(0, pointIndex);
36            }else{
37                input = input.substring(0, pointIndex);
38            }
39        }
40
41        // 第一个字符是小数点，则输出只能是0或1
42        if(input.length()<1){
43            return extra;
44        }
45
46        // 溢出判断
47        String MAXPOS = Integer.MAX_VALUE+"";
48        String MAXNEG = Integer.MIN_VALUE+"";
49        if(isNegative){
50            String temp = '-' + input;
51            if(temp.length()>MAXNEG.length()){
52                throw new Exception("Overflow error!");
53            }else if(temp.length()==MAXNEG.length()){
```

≡ anonymous_1

```

57         throw new Exception("Overflow error!");
58     }
59 }
60 }else{
61     if(input.length()>MAXPOS.length()){
62         throw new Exception("Overflow error!");
63     }else if(input.length()==MAXPOS.length()){
64         if(input.compareTo(MAXPOS)>0){
65             throw new Exception("Overflow error!");
66         }else if(input.compareTo(MAXPOS)==0&&extra>0){
67             throw new Exception("Overflow error!");
68         }
69     }
70 }
71
72 //当前的字符串为合法的，无符号的，只包含数字的字符串，可以直接转化为数字（按需要添加
73 int res = 0;
74 int base = 1;
75 for(int i=input.length()-1;i>-1;i--){
76     char ch = input.charAt(i);
77     res = res + base*(ch-'0');
78     base *= 10;
79 }
80 return res*(isNegative?-1:1);
81 }
82
83 public static int getPointIndex(String str){
84     for(int i=0;i<str.length();i++){
85         if(str.charAt(i)=='.'){
86             return i;
87         }
88     }
89     return -1;
90 }
91
92 public static boolean valid(String str){
93     if(str.charAt(0)=='-'&&str.length()>1){
94         return valid(str.substring(1, str.length()));
95     }
96     int nPoint = 0;
97     for(char ch:str.toCharArray()){
98         if(ch=='.'){
99             nPoint += 1;
100             if(nPoint>1){
101                 return false;
102             }
103         }else if(ch>'9' || ch<'0'){
104             return false;
105         }
106     }
107     return true;
108 }
109
110 public static void main(String args[]){
111     //-2147483648 ~ 2147483647

```


☰ anonymous_1

```
115         "0.67", //1
116         "adfasdf", //Invalid input string!
117         "2147483648", //Overflow error!
118         "1.1.5", //Invalid input string!
119         "123.", //123
120         ".67" //1
121     };
122     for(String s:strs){
123         try{
124             System.out.println(transform(s));
125         }catch(Exception e){
126             System.out.println(e.getMessage());
127         }
128     }
129 }
130 }
131 }
```

面试手撕代码

举报

收藏 2

赞 1

0条评论

🔄 默认排序 ▾



没有回复

请留下你的观点吧~

发布