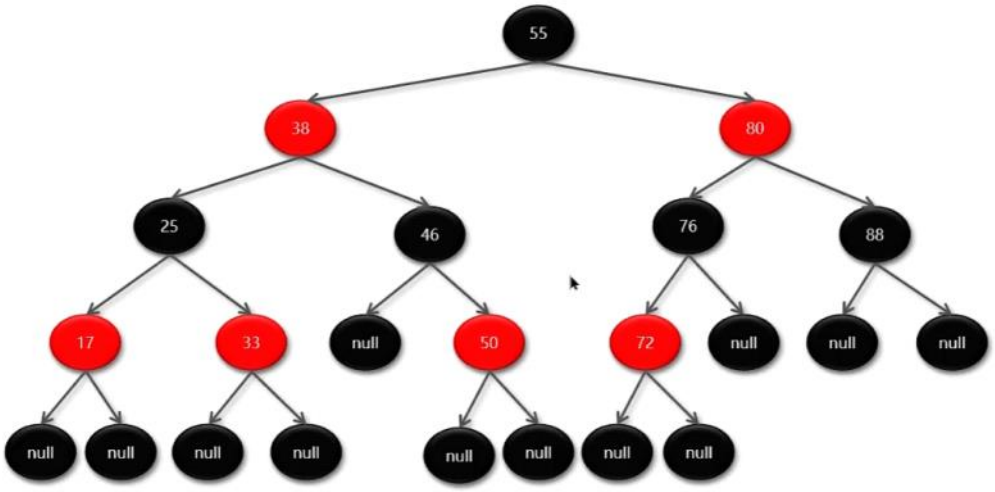
红黑树、散列表、HashMap

2024年4月2日 17:04

红黑树

红黑树（Red Black Tree）：也是一种自平衡的二叉搜索树(BST)，之前叫做平衡二叉B树（Symmetric Binary B-Tree）



红黑树的特质

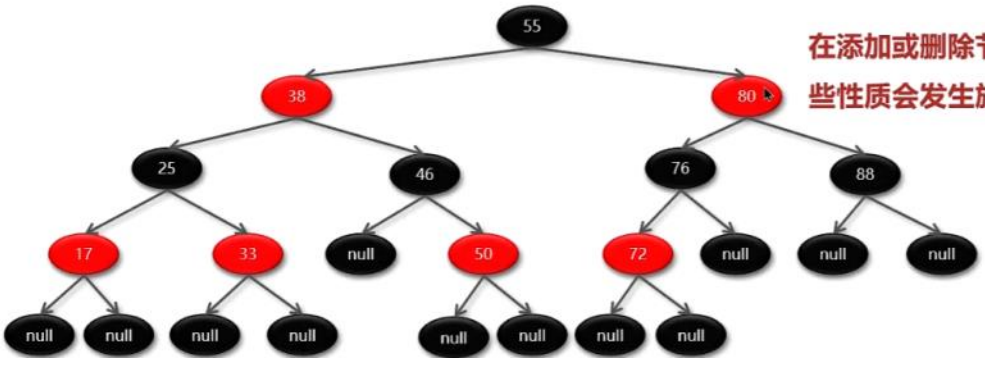
性质1：节点要么是**红色**,要么是**黑色**

性质2：根节点是**黑色**

性质3：叶子节点都是黑色的空节点

性质4：红黑树中红色节点的子节点都是黑色

性质5：从任一节点到叶子节点的所有路径都包含相同数目的黑色节点



在添加或删除节点的时候，如果不符合这些性质会发生旋转，以达到所有的性质

红黑树的复杂度

- 查找：

  红黑树也是一棵BST（二叉搜索树）树，查找操作的时间复杂度为：O(log n)

- 添加：

  添加先要从根节点开始找到元素添加的位置，时间复杂度O(log n)

  添加完成后涉及到复杂度为O(1)的旋转调整操作

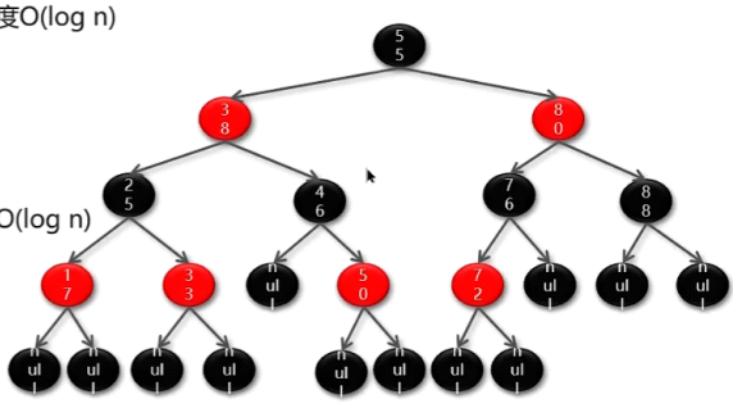
  故整体复杂度为：O(log n)

- 删除：

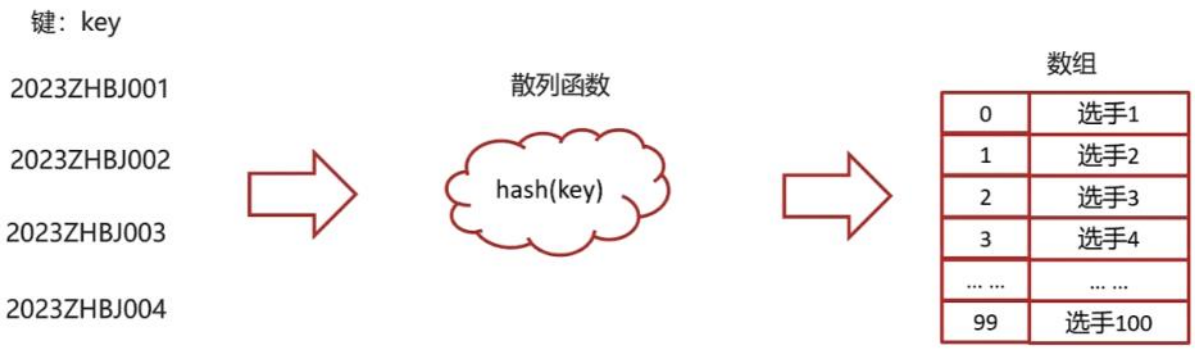
  首先从根节点开始找到被删除元素的位置，时间复杂度O(log n)

  删除完成后涉及到复杂度为O(1)的旋转调整操作

  故整体复杂度为：O(log n)



散列表（Hash Table）



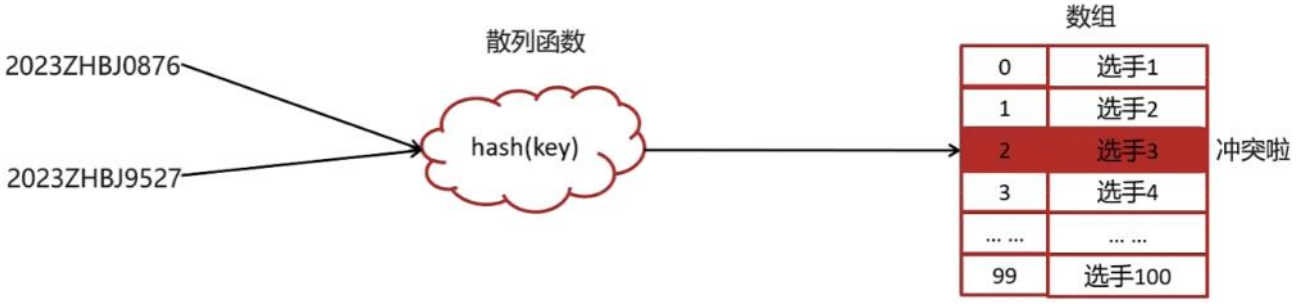
将键(key)映射为数组下标的函数叫做散列函数。可以表示为：hashValue = hash(key)

散列函数的基本要求：

- 散列函数计算得到的散列值必须是大于等于0的正整数，因为hashValue需要作为数组的下标。
- 如果key1==key2，那么经过hash后得到的哈希值也必相同即：hash(key1) == hash(key2)
- **如果key1 != key2，那么经过hash后得到的哈希值也必不相同即：hash(key1) != hash(key2)**

散列冲突

实际的情况下想找一个散列函数能够做到对于不同的key计算得到的散列值都不同几乎是不可能的，即便像著名的MD5,SHA等哈希算法也无法避免这一情况，这就是散列冲突(或者哈希冲突，哈希碰撞，就是指多个key映射到同一个数组下标位置)



散列冲突-链表法（拉链）

在散列表中，数组的每个下标位置我们可以称之为桶（bucket）或者槽（slot），每个桶(槽)会对应一条链表，所有散列值相同的元素我们都放到相同槽位对应的链表中。

散列冲突-链表法（拉链）- 时间复杂度

（1）插入操作，通过散列函数计算出对应的散列槽位，将其插入到对应链表中即可，插入的时间复杂度是 O(1)

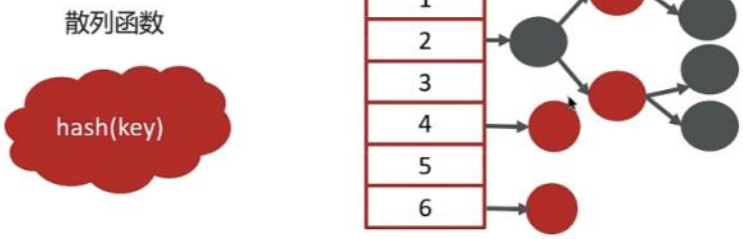
散列冲突-链表法（拉链）

(2) 当查找、删除一个元素时，我们同样通过散列函数计算出对应的槽，然后遍历链表查找或者删除

- 平均情况下基于链表法解决冲突时查询的时间复杂度是O(1)
- 散列表可能会退化为链表,查询的时间复杂度就从 O(1) 退化为 O(n)
- 将链表法中的链表改造为其他高效的动态数据结构，比如红黑树，查询的时间复杂度是 O(logn)

散列冲突-链表法（拉链）

2023ZHBJ0875
2023ZHBJ0876
2023ZHBJ0877
2023ZHBJ0878
2023ZHBJ0879
2023ZHBJ0880
2023ZHBJ0881
2023ZHBJ0882
2023ZHBJ0883
.....



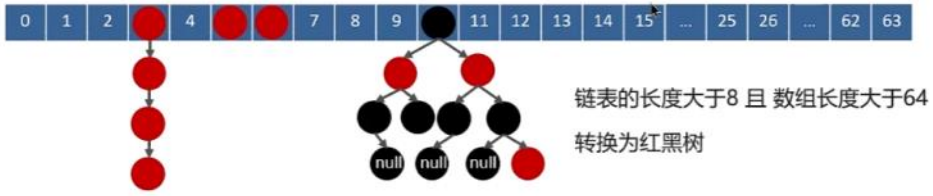
将链表法中的链表改造红黑树还有一个非常重要的原因，可以防止DDos攻击

DDos 攻击:
分布式拒绝服务攻击(英文意思是Distributed Denial of Service，简称DDoS)
指处于不同位置的多个攻击者同时向一个或数个目标发动攻击，或者一个攻击者控制了位于不同位置的多台机器并利用这些机器对受害者同时实施攻击。由于攻击的发出点是分布在不同地方的，这类攻击称为分布式拒绝服务攻击，其中的攻击者可以有多个

HashMap实现原理

HashMap的数据结构： 底层使用hash表数据结构，即数组和链表或红黑树

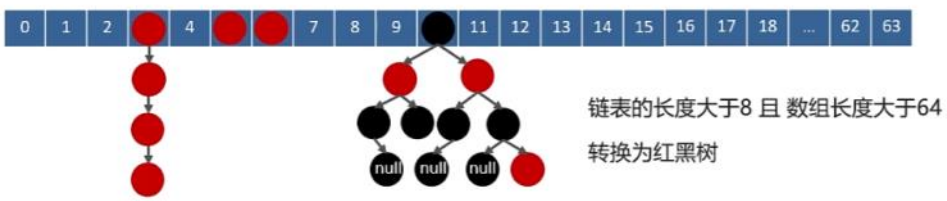
1. 当我们往HashMap中put元素时，利用key的hashCode重新hash计算出当前对象的元素在数组中的下标
2. 存储时，如果出现hash值相同的key，此时有两种情况。
   a. 如果key相同，则覆盖原始值；
   b. 如果key不同（出现冲突），则将当前的key-value放入链表或红黑树中
3. 获取时，直接找到hash值对应的下标，在进一步判断key是否相同，从而找到对应值。



面试官追问：HashMap的jdk1.7和jdk1.8有什么区别

HashMap的jdk1.7和jdk1.8有什么区别

- JDK1.8之前采用的是拉链法。拉链法：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。
- jdk1.8在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时并且数组长度达到64时，将链表转化为红黑树，以减少搜索时间。扩容 resize( ) 时，红黑树拆分成的树的结点数小于等于临界值6个，则退化成链表



1. 说一下HashMap的实现原理?

- 底层使用hash表数据结构，即数组+（链表 | 红黑树）
- 添加数据时，计算key的值确定元素在数组中的下标
  - key相同则替换
  - 不同则存入链表或红黑树中

  获取数据通过key的hash计算数组下标获取元素

2. HashMap的jdk1.7和jdk1.8有什么区别

- JDK1.8之前采用的拉链法，数组+链表
- JDK1.8之后采用数组+链表+红黑树，链表长度大于8且数组长度大于64则会从链表转化为红黑树

HashMap源码分析 – 常见属性

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
static final float DEFAULT_LOAD_FACTOR = 0.75f;
transient HashMap.Node<K,V>[] table;
transient int size;
```

```
static class Node<K, V> implements Map.Entry<K, V> {
    final int hash;
    final K key;
    V value;
    HashMap.Node<K, V> next;

    Node(int hash, K key, V value, HashMap.Node<K, V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }
}
```

- *DEFAULT_INITIAL_CAPACITY 默认的初始容量*
- *DEFAULT_LOAD_FACTOR 默认的加载因子*

扩容阈值 == 数组容量 * 加载因子

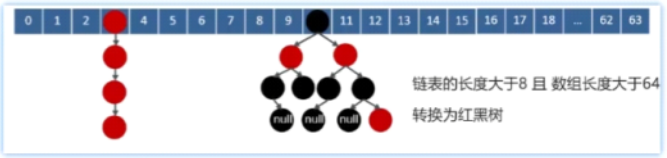
HashMap源码分析

```
Map<String, String> map = new HashMap<>();
map.put("name", "itheima");
```

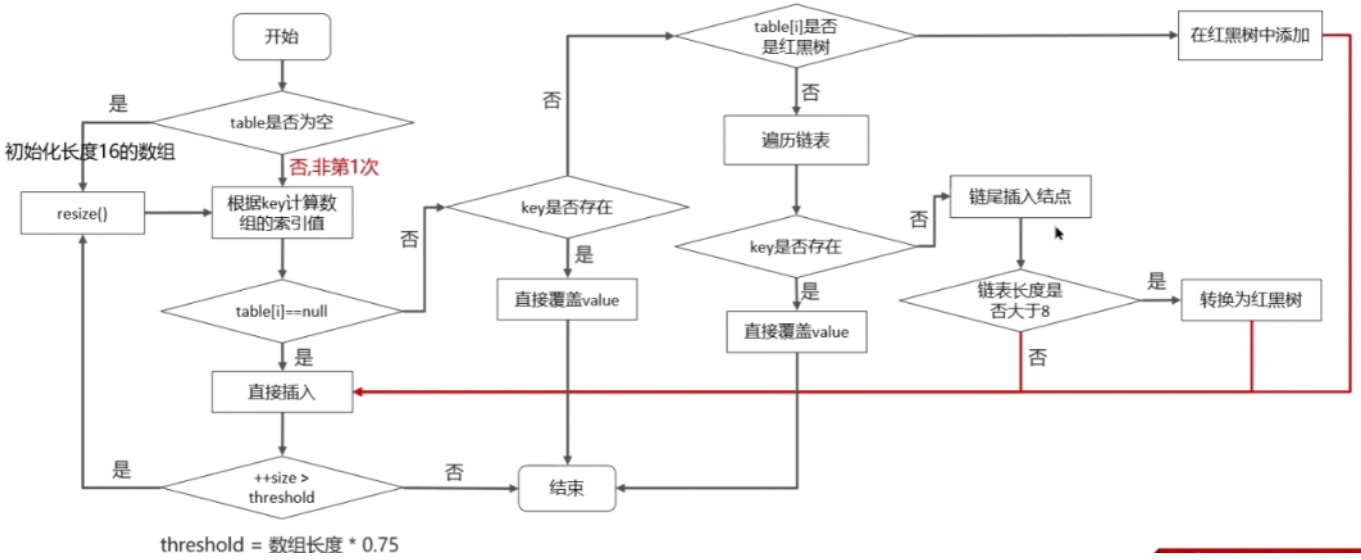
```
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
```

- HashMap是懒惰加载，在创建对象时并没有初始化数组
- 在无参的构造函数中，设置了默认的加载因子是0.75

HashMap源码分析-添加数据



添加数据流程图

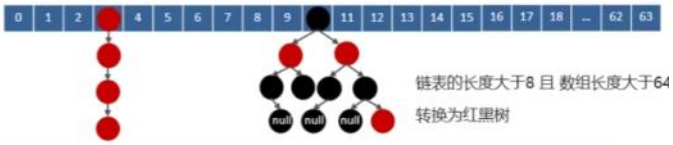


threshold = 数组长度 * 0.75

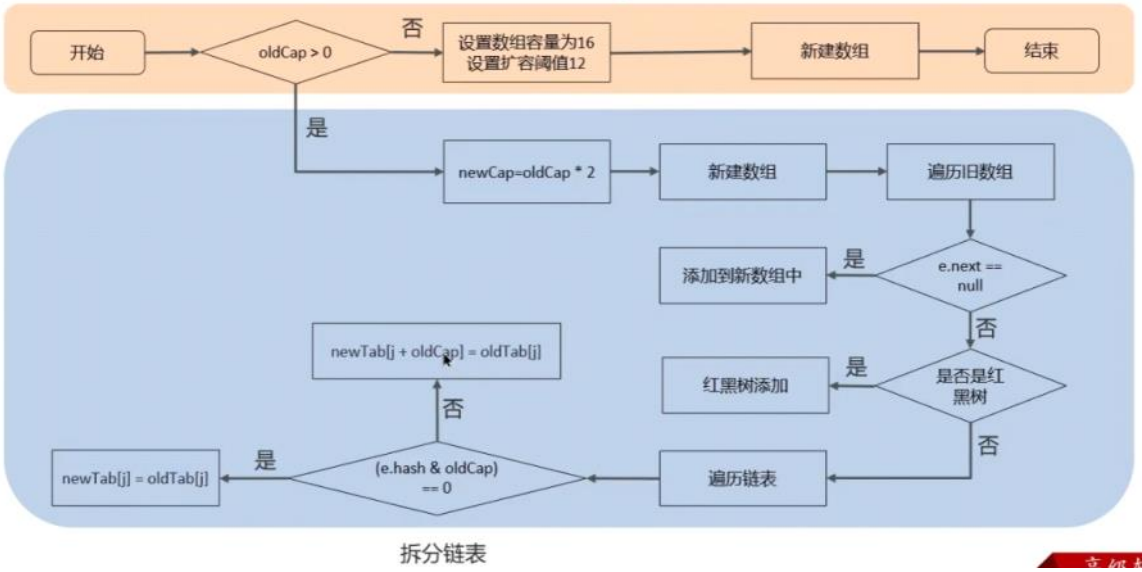
HashMap的put方法的具体流程

1. 判断键值对数组table是否为空或为null，否则执行resize()进行扩容（初始化）
2. 根据键值key计算hash值得到数组索引
3. 判断table[i]==null，条件成立，直接新建节点添加
4. 如果table[i]==null ,不成立
   - 4.1 判断table[i]的首个元素是否和key一样，如果相同直接覆盖value
   - 4.2 判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对
   - 4.3 遍历table[i]，链表的尾部插入数据，然后判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操 作，遍历过程中若发现key已经存在直接覆盖value
5. 插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold（数组长度*0.75），如果超过，进行扩容。

HashMap源码分析-扩容



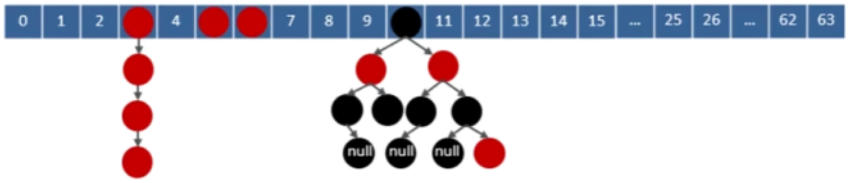
扩容的流程



拆分链表

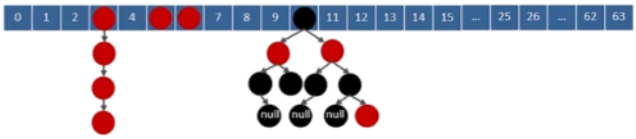
讲一讲HashMap的扩容机制

- 在添加元素或初始化的时候需要调用resize方法进行扩容，第一次添加数据初始化数组长度为16，以后每次每次扩容都是达到了扩容阈值（数组长度 * 0.75）
- 每次扩容的时候，都是扩容之前容量的2倍；
- 扩容之后，会新创建一个数组，需要把老数组中的数据挪动到新的数组中
  - 没有hash冲突的节点，则直接使用 e.hash & (newCap - 1) 计算新数组的索引位置
  - 如果是红黑树，走红黑树的添加
  - 如果是链表，则需要遍历链表，可能需要拆分链表，判断(e.hash & oldCap)是否为0，该元素的位置要么停留在原始位置，要么移动到原始位置+增加的数组大小这个位置上



hashMap的寻址算法

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```




```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

扰动算法，是hash值更加均匀，减少hash冲突

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    ......
    if ((p = tab[i = (n - 1) & hash]) == null)
    ......
}
```

(n-1)&hash : 得到数组中的索引，代替取模，性能更好

数组长度必须是2的n次幂

为何HashMap的数组长度一定是2的次幂?

1. 计算索引时效率更高：如果是 2 的 n 次幂可以使用位与运算代替取模
2. 扩容时重新计算索引效率更高： hash & oldCap == 0 的元素留在原来位置 ，否则新位置 = 旧位置 + oldCap

1. hashMap的寻址算法

- 计算对象的 hashCode()
- 再进行调用 hash() 方法进行二次哈希， hashcode值右移16位再异或运算，让哈希分布更为均匀
- 最后 (capacity – 1) & hash 得到索引

2. 为何HashMap的数组长度一定是2的次幂?

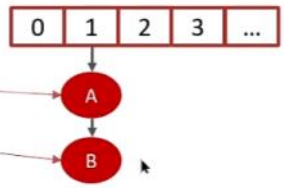
- 计算索引时效率更高：如果是 2 的 n 次幂可以使用位与运算代替模
- 扩容时重新计算索引效率更高： hash & oldCap == 0 的元素留在原来位置，否则新位置 = 旧位置 + oldCap

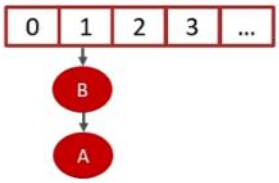
hashmap在1.7情况下的多线程死循环问题

jdk7的的数据结构是：数组+链表

在数组进行扩容的时候，因为链表是头插法，在进行数据迁移的过程中，有可能导致死循环

```
void transfer(Entry[] newTable, boolean rehash) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            if (rehash) {
                e.hash = null == e.key ? 0 : hash(e.key);
            }
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```





- 变量e指向的是需要迁移的对象
- 变量next指向的是下一个需要迁移的对象
- Jdk1.7中的链表采用的头插法
- 在数据迁移的过程中并没有新的对象产生，只是改变了对象的引用

参考回答:

在jdk1.7的hashmap中在数组进行扩容的时候，因为链表是头插法，在进行数据迁移的过程中，有可能导致死循环

比如说，现在有两个线程

线程一：读取到当前的hashmap数据，数据中一个链表，在准备扩容时，线程二介入

线程二：也读取hashmap，直接进行扩容。因为是头插法，链表的顺序会进行颠倒过来。比如原来的顺序是AB，扩容后的顺序是BA，线程二执行结束。

线程一：继续执行的时候就会出现死循环的问题。

线程一先将A移入新的链表，再将B插入到链头，由于另外一个线程的原因，B的next指向了A，所以B->A->B,形成循环。

当然，JDK 8 将扩容算法做了调整，不再将元素加入链表头（而是保持与扩容前一样的顺序），**尾插法**，就避免了jdk7中死循环的问题。