拨开云雾看世界

摘要视图 订阅

Map(一)之HashMap (java8)

标签： java java8 hashmap

2016-07-13 16:26 3353人阅读 评论(2) 收藏 举报

分类： java (9) java集合类 (4) java8 (1) hashmap (1)

目录(?) [+]

概述【本文基于jdk1.8.0_60】

在我们日常开发中，HashMap被使用到的概率非常高。它是一种非常典型的**数据结构**。我们应该都知道Map是存储key-value键值对的集合类，也就是说元素是成对出现的。并且key可以为null但必须是唯一的。

定义

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

HashMap实现了Map接口，继承了AbstractMap。

基本属性

收藏到代码笔记

关闭

```
    /**
     * 默认初始容量(桶的数量)，16
     */
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;

    /**
     * 最大容量1*2^30
     */
    static final int MAXIMUM_CAPACITY = 1 << 30;

    /**
     * 默认装载因子，此值表示当前容量的HashMap装entry满的程度，当entry数量大于当前容量与装载因子的
     * 乘积时，HashMap就会进行rehash操作。也就是HashMap会扩充容量，扩充容量之后整个HashMap就会
     * 重建
     */
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    /**
     *当“桶”中的元素大于此阀值时使用树代替单向链表，这里的树实际上是红黑树
     */
    static final int TREEIFY_THRESHOLD = 8;

    /**
     * 当“桶”中的元素小于此阀值时，树转成单向链表
     */
    static final int UNTREEIFY_THRESHOLD = 6;

    /**
     *HashMap中所有元素总数小于此值时，即使“桶”中元素超过TREEIFY_THRESHOLD也不转成树
     */
    static final int MIN_TREEIFY_CAPACITY = 64;
```

构造

```
    //默认构造方法，装载因子设置为默认值0.75
    public HashMap() {
```

个人资料

chengli_007

关注 发私信

访问： 28718次
积分： 487
等级： BLOG 2
排名： 千里之外

原创： 18篇 转载： 1篇
译文： 0篇 评论： 7条

文章搜索

文章存档

2017年01月 (2)
2016年12月 (1)
2016年11月 (3)
2016年08月 (4)
2016年07月 (9)

文章分类

spring (2)
java (10)
mysql (0)
redis (0)
nginx (2)
数据结构与算法 (1)
java集合类 (5)
HashSet (1)
java8 (2)
hashmap (2)
request (1)
单点登录 (2)
session共享 (1)
其他 (0)
springmvc (1)
jvm (2)

阅读排行

Spring mvc请求处理流程详解... (8666)
Map(一)之HashMap (java8) (3343)
jvm - 类的初始化过程 (2745)
jvm - ClassLoader (2486)
spring-session导致request.g... (2351)
电梯算法题 (1590)

```
        this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
    }
```

```
    //指定初始容量和加载因子
    public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }
```

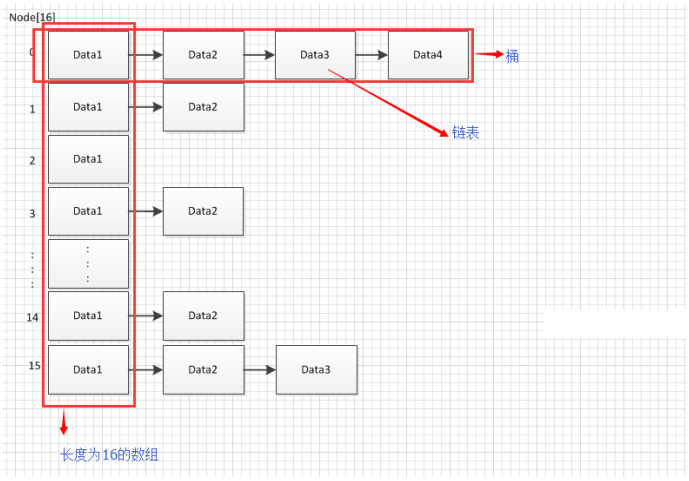
```
    //指定初始容量
    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }
```

```
    //使用一个Map来初始化一个HashMap
    public HashMap(Map<? extends K, ? extends V> m) {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        putMapEntries(m, false);
    }
```

底层存储

```
    /**
     *HashMap底层实际上是一个Node<K,V>类型的数组
     */
    transient Node<K,V>[] table;
```

简单画一下HashMap底层存储结构图：



关闭

上图描述的是一个容量为**16**(即16个桶)的的HashMap结构图。可以理解为是一个长度为16的Node类型的数组，每个数组元素都是一个链表的head结点。于是乎就构成了上面的结构。我们用默认构造方法创建的HashMap在没有resize之前容量就是16。

resize与树化

随着结点的增多，单个桶中的元素越来越大，即链表的长度越来越长。这样会导致get，remove（大家可以想一下链表的相关操作）等的性能越来越差，怎么办呢？HashMap中有对应的解决方案。

resize

HashMap中有一个叫做threshold的参数：

```
    /**
     * The next size value at which to resize (capacity * load factor).
     */
    int threshold;
```

这个参数表示当HashMap的size达到此值时就要进行resize操作。它是怎么计算来的呢？计算公式如下：

```
threshold = capacity * load factor
```

拿上图举个例子，假如上图的HashMap是通过默认构造方法创建的。其中HashMap的容量为16，load_factor为默认值0.75。所以threshold = 16 * 0.75 = 12。也就是说当元素个数达到12时，就会发生resize操作。结构就会发生改变。看一下resize的源码（我用注释在代码上描述下关键步骤的意思）：

```
    final Node<K, V>[] resize() {
      Node<K, V>[] oldTab = table;
      int oldCap = (oldTab == null) ? 0 : oldTab.length;
      int oldThr = threshold;
      int newCap, newThr = 0;
      //扩充容量会进入这个if语句块
      if (oldCap > 0) {
          //容量不能超过MAXIMUM_CAPACITY上限
          if (oldCap >= MAXIMUM_CAPACITY) {
              threshold = Integer.MAX_VALUE;
              return oldTab;
          //新容量为旧容量的2倍，新阀值为旧阀值的两倍。（<<1表示2进制左移一位，也就是乘以2）
          } else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                  oldCap >= DEFAULT_INITIAL_CAPACITY)
              newThr = oldThr << 1; // double threshold
      //这个分支是应对使用不能构造方法创建HashMap的情况
      } else if (oldThr > 0) // initial capacity was placed in threshold
          newCap = oldThr;
      else {               // zero initial threshold signifies using defaults
          newCap = DEFAULT_INITIAL_CAPACITY;
          newThr = (int) (DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
      }
      if (newThr == 0) {
          float ft = (float) newCap * loadFactor;
          newThr = (newCap < MAXIMUM_CAPACITY && ft < (float) MAXIMUM_CAPACITY ?
                  (int) ft : Integer.MAX_VALUE);
      }
      //阀值赋值为新的值
      threshold = newThr;
      //创建一个更大容量的Node数组
      Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap];
      //把新创建的数组赋值给table属性
      table = newTab;
      if (oldTab != null) {
          //遍历旧数组中每个桶
          for (int j = 0; j < oldCap; ++j) {
              Node<K, V> e;
              if ((e = oldTab[j]) != null) {
                  oldTab[j] = null;
                  if (e.next == null)
                      newTab[e.hash & (newCap - 1)] = e;
                  else if (e instanceof TreeNode)
                      ((TreeNode<K, V>) e).split(this, newTab, j, oldCap);
                  else { // preserve order
                      Node<K, V> loHead = null, loTail = null;
                      Node<K, V> hiHead = null, hiTail = null;
                      Node<K, V> next;
                      do {
                          next = e.next;
                          /**
                          *下面的是把旧HashMap中的元素放到新HashMap。映射算法后面再详细讲解
                          */
                          if ((e.hash & oldCap) == 0) {
                              if (loTail == null)
                                  loHead = e;
                              else
                                  loTail.next = e;
                              loTail = e;
                          } else {
                              if (hiTail == null)
                                  hiHead = e;
                              else
                                  hiTail.next = e;
                              hiTail = e;
                          }
                      } while ((e = next) != null);
                      if (loTail != null) {
                          loTail.next = null;
                          newTab[j] = loHead;
                      }
                      if (hiTail != null) {
                          hiTail.next = null;
                          newTab[j + oldCap] = hiHead;
                      }
                  }
              }
          }
      }
      return newTab;
  }
```

关闭

树化

我们观察会发现HashMap有两个属性：

```
    static final int TREEIFY_THRESHOLD = 8;
    static final int MIN_TREEIFY_CAPACITY = 64;
```

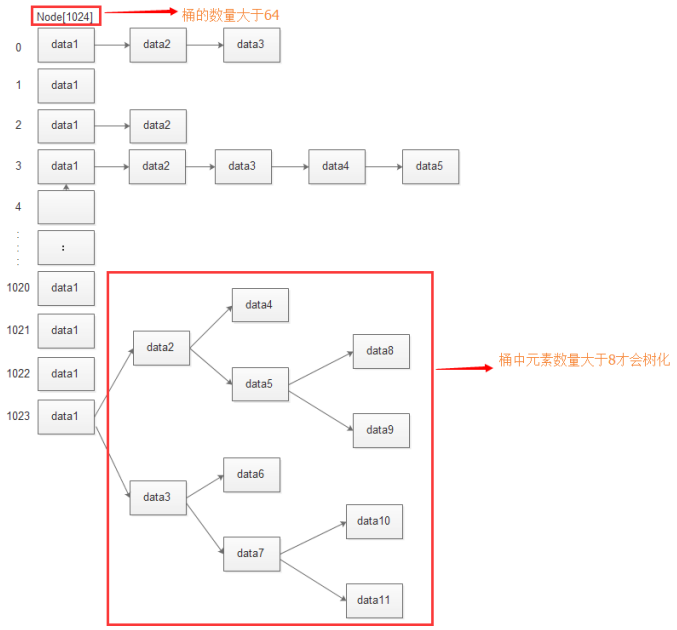
当HashMap的容量大于MIN_TREEIFY_CAPACITY 并且桶中元素数量大于等于TREEIFY_THRESHOLD 时，该桶中的元素结构就会由**链表**结构转成**树**结构

以提高性能。看下源码：

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    //这里是保证桶中元素和HashMap容量同时满足条件才对相应桶进行树化。否则只进行resize操作
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        TreeNode<K,V> hd = null, tl = null;
        /**
         *这里可以理解为把链表中每个结点都替换成树结点，世界上是创建一个新链表，结点类型
         *由Node变为TreeNode
         */
        do {
            //创建树结点
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            //具体树化算法
            hd.treeify(tab);
    }
}
```

关闭

看一下树化之后的HashMap结构长什么样：



上图编号为**1023**的桶由于数量超过了**阀值**，所以有**链表**转成了**树**形结构。关于链表转成树的具体**算法**这里就不详细讲解了。

有树化过程当然就有**“非树化”**过程，我这里树的**非树化**就是树转链表。当然如果我们频繁做**remove**操作，链表里面的元素越来越少就会进行逆向操作。相应的**“非树化”**阀值参数定义如下：

```
/**
 * The bin count threshold for untreeifying a (split) bin during a
 * resize operation. Should be less than TREEIFY_THRESHOLD, and at
 * most 6 to mesh with shrinkage detection under removal.
 */
static final int UNTREEIFY_THRESHOLD = 6;
```

意思是当桶中元素个数小于**UNTREEIFY_THRESHOLD** 时，就会由**树**转成**链表**，当然，前提是你之前已经进行了**树化**操作。

其他创建流程

HashMap提供了4个构造方法，这使我们可以在创建**HashMap**的时候指定初始容量和加载因子。当我们使用这两个构造方法时有一点可能和你想的有点不一样，下面我们探讨一下：

```
//指定初始容量和加载因子
```

```
    public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapac
    }
```

关闭

```
    //指定初始容量
    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }
```

假如我们想创建一个初始容量为30的HashMap，也许你会这样写：

```
Map<String, Object> map = new HashMap<>(30);
```

我们这样写实际上调用的是 **public HashMap(int initialCapacity, float loadFactor)** ，这里**loadFactor**是默认值，我们不管它，看最后一句。

```
this.threshold = tableSizeFor(initialCapacity);
```

啊？**initialCapacity**和**threshold**什么关系？是不是感觉有点混乱？我们刚刚不是想创建容量为**30**的**HashMap**吗？为何这里把**initialCapacity**通过计算赋值给了**threshold**？别急！我们一步步来，先来看下这个**tableSizeFor**方法时干嘛的，上源码：

```
    /**
     * Returns a power of two size for the given target capacity.
     */
    static final int tableSizeFor(int cap) {
        int n = cap - 1;
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
        return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACIT
    }
```


它的作用是根据传入的容量，返回**2**的**幂次方**大小的容量。你刚刚传的是**30**，这里会返回**2^5=32**。但是这个值也没有赋值给**initialCapacity**参数呀？继续看：

我们知道我们虽然**new HashMap(30)**，但此时**HashMap**里面的真正用来存储数组的**Node**数组table还是**null**呢：

```
    transient Node<K,V>[] table;
```

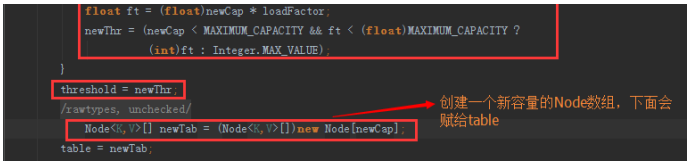
此时当我们往**HashMap**中**put**元素时，如果**table==null**，就需要进行第一次**resize**操作：

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
```

我们继续看**resize**方法：

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
```


关闭



这里会把threshold 的值赋给**newCap**，然后根据这个**newCap**创建一个**Node**数组赋给**table**。然后根据**newCap**计算一个**threshold**赋值给**threshold**。此时的**newCap**的值为**32**，threshold的值为**24** 。所以我们刚开始使用 `Map<String, Object> map = new HashMap<>(30);` 创建的**map**的容量不是**30**，而是**32**。

HashMap中映射算法介绍

- 1 key映射到桶

  我们拿**get(object key)**方法来分析，key是如何映射到桶的，也就是HashMap如何知道key对应的元素存在哪个链表的？我们看下源码：


```
1   public V get(Object key) {
2         Node<K,V> e;
3         return (e = getNode(hash(key), key)) == null ? null : e.value;
4       }
5
6    final Node<K,V> getNode(int hash, Object key) {
7         Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
8         if ((tab = table) != null && (n = tab.length) > 0 &&
9             (first = tab[(n - 1) & hash]) != null) {
10            if (first.hash == hash && // always check first node
11                ((k = first.key) == key || (key != null && key.equals(k))))
12                return first;
13            if ((e = first.next) != null) {
14                if (first instanceof TreeNode)
15                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
16                do {
17                    if (e.hash == hash &&
18                        ((k = e.key) == key || (key != null && key.equals(k))))
19                        return e;
20                } while ((e = e.next) != null);
21            }
22        }
23        return null;
24    }
```

仔细观察有这么一行代码： `tab[(n - 1) & hash]` 其中hash的值就是key的哈希值，从中我们可以大致知道hash值是怎么映射到桶的。接下来我们通过例子来更深层次的探讨下：

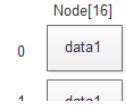
假如我们通过构造方法创建了一个HashMap，那么它的初始容量是16。现在我们往HashMap中添加4个元素。假如这4个元素key的hash值分别是 6、22、38、54 （具体key的值是多少我们这里就不列出了）。现在我们看下这4个元素会映射到哪个桶？由于容量是16，所以 `n=16` ，那么 `n - 1) & hash` 代入n就变成了 `15 & hash` 。我们把 6、22、38、54 分别于 `15` 进行按位与操作，换算成二进制：

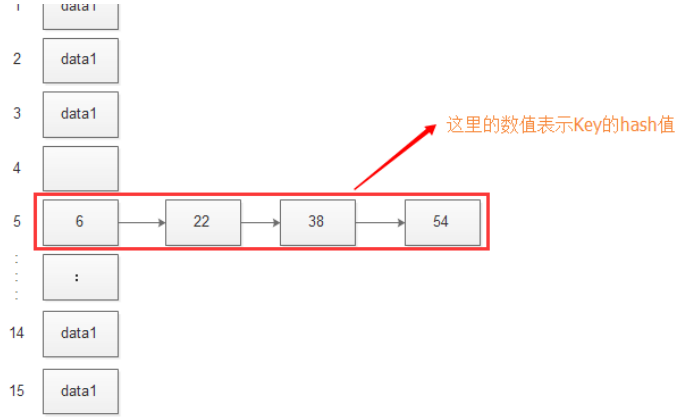
```
 6  &  15          22  &  15          38  &  15          54  &  15

 000110            010110            100110            110110
 001111            001111            001111            001111
 ------            ------            ------            ------
 000110            000110            000110            000110
```

计算之后的结果全是6，于是这四个元素全部放到了**table[6]**中，即第**6**个桶。另外由于**15**二进制**001111**，除了低四位为**1**，其他高位全是**0** 。所以与**15**进行**按位与**结果永远不会大于**15**，所以不会超出**table**大小范围。这四个元素放到**HashMap**中的结构如图：

关闭





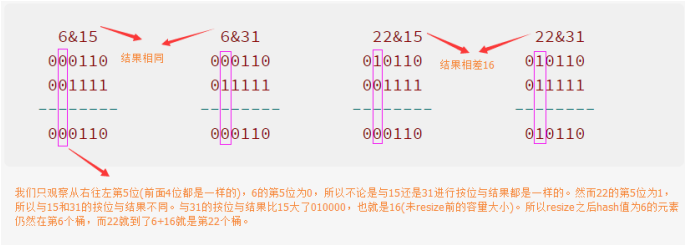
- 2 resize时key的重新映射

接着上面的例子讲，随着**HashMap**中元素的增多，我们建立的**HashMap**容量是**16**，加载因子**loader_factor**为**0.75** 。所以 `threshold = 16*0.75 = 12` 所以当元素数量超过12时，HashMap就会resize。假如现在上例中元素刚好达到12，第6个桶中的元素还是上述那四个元素。我们看下这四个元素分别会重新映射到新table中的哪个桶。在开始讨论之前，先看下重新映射的代码：

```
if (oldTab != null) {
    for (int j = 0; j < oldCap; ++j) {
        Node<K,V> e;
        if ((e = oldTab[j]) != null) {
            oldTab[j] = null;
            if (e.next == null)
                newTab[e.hash & (newCap - 1)] = e;
            else if (e instanceof TreeNode)
                ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
            else { // preserve order
                Node<K,V> loHead = null, loTail = null;
                Node<K,V> hiHead = null, hiTail = null;
                Node<K,V> next;
                do {
                    next = e.next;
                    if ((e.hash & oldCap) == 0) {
                        if (loTail == null)
                            loHead = e;
                        else
                            loTail.next = e;
                        loTail = e;
                    }
                    else {
                        if (hiTail == null)
                            hiHead = e;
                        else
                            hiTail.next = e;
                        hiTail = e;
                    }
                } while ((e = next) != null);
                if (loTail != null) {
                    loTail.next = null;
                    newTab[j] = loHead;
                }
                if (hiTail != null) {
                    hiTail.next = null;
                    newTab[j + oldCap] = hiHead;
                }
            }
```


关闭

我们知道原来HashMap的容量是16，resize之后容量是32 。旧数组中的元素映射到新数组中的算法就如上图所示。是不是有点晕？为什么要这样算呢？我们先看一下resize之前和之后往HashMap中添加元素是如何映射的，以key的hash值是6、22为例，映射算法分别是 `6 & 15` 与 `6 & 31` 和 `22 & 15` 与 `22 & 31` ，换算成二进制计算如下图：



如上图所示，我们只观察从右往左第**5**位(前面4位都是一样的)，**6**的第**5**位为**0**，所以不论是与15还是31进行按位与操作结果都是一样的。然而**22**的第

5位为1，所以与15和31的按位与结果不同。与31的按位与结果比15大了010000，也就是16(未resize前的容量大小)。可以理解为resize之后，元素在新HashMap中桶的位置是否改变，取决于原第5(*在这个例子中是第五位*)位是**0**还是**1**。那么代码中的算法就是按照第5位是0，还是1进行了分组。代码中的 `hash&oldCap` ，刚好可以区分第5位是0还是1 。看例子：

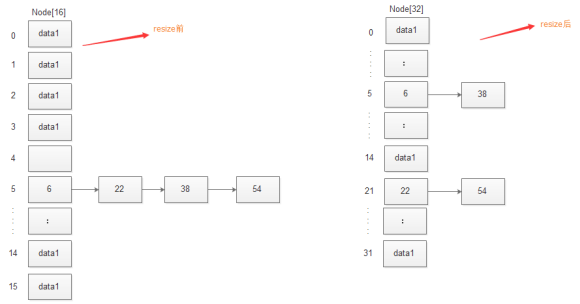
```
 6&16          22&16
000110        010110
010000        010000
------        ------
000000        010000
```

判断6和22的第5位是否为1

结果为0的第5位是0，否则第5位不为0 。

说明一下：上述算法只有当resize之后是之前的2倍的基础上才成立。换算成2进制就是左移了一位： `newCap = oldCap <<1` ，这个是必要条件。否则上述算法就不对了。不知道我有没有讲清楚。

下面看下resize前后结构图：



关闭

HashMap的bug

我们知道，旧版本**jdk**中**HashMap**在多线程条件下**rehash**操作有可能会产生**无限循环**的问题。不了解此问题的麻烦搜索下相关问题，这里不做描述。在jdk1.8中(我研究的这个版本)，此问题已经得到解决。(**我看其他人写的博客中还提到说java8中hashMap死循环的问题，他们应该是搞错了**)。产生**无限循环**的本质就是链表中产生了**环**，而java8中不具有产生环的条件，所以不会产生无限循环。但是产生了其他bug。

1. 为何不会产生环

java8中resize时对每个桶进行的操作都是先将链表拆分成两个链表，然后分别将两个链表放到新table中对应的桶当中去。这个过程在多线程条件下不会产生环。

2. 其他bug（多线程条件下）

- 1 HashMap中有值，但是get出来为null。

  来看下resize时有这么一行代码：

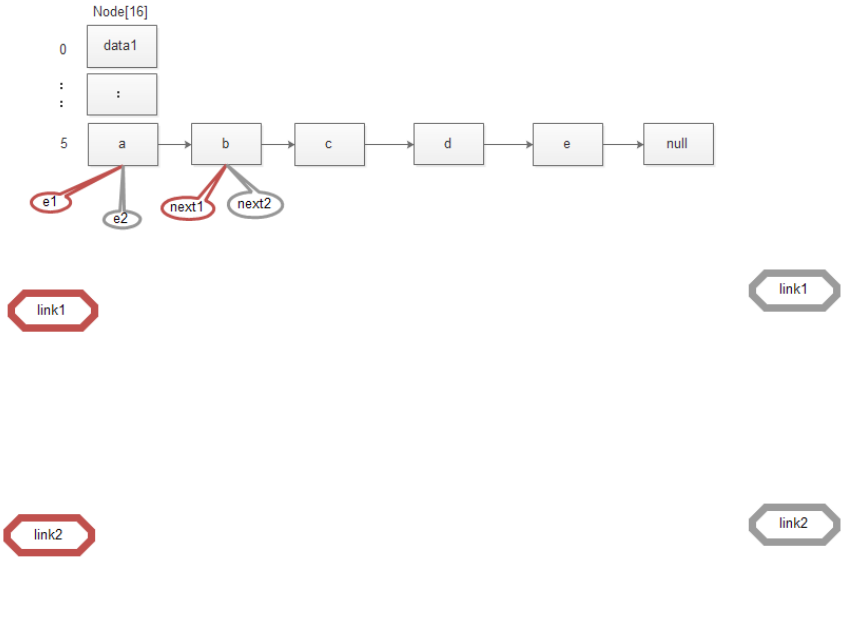
```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
```

```
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        if (oldTab != null) {
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
```

如上图所示，在进行真正的元素移动前，先将newTab赋值给了table，这时table是空的，什么都没有，假如这个时候进行get操作，什么都取不到。

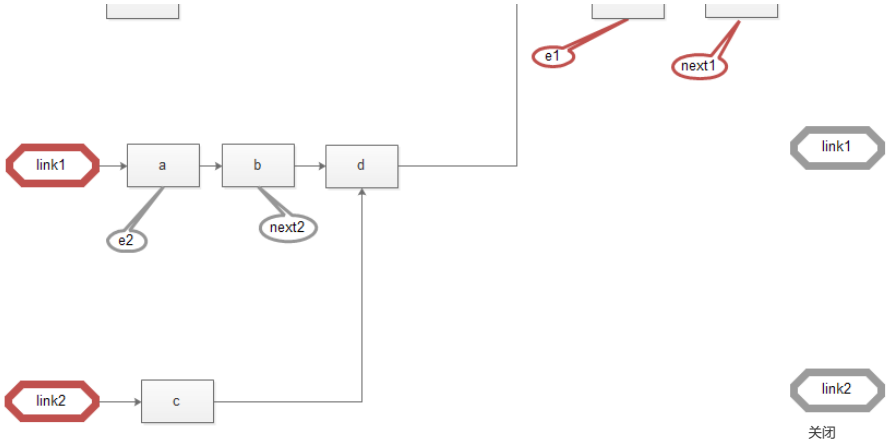
- 2 resize时元素丢失问题。

  假设同时又两个线程thread1、thread2同时进行resize操作，初始状态图如下：

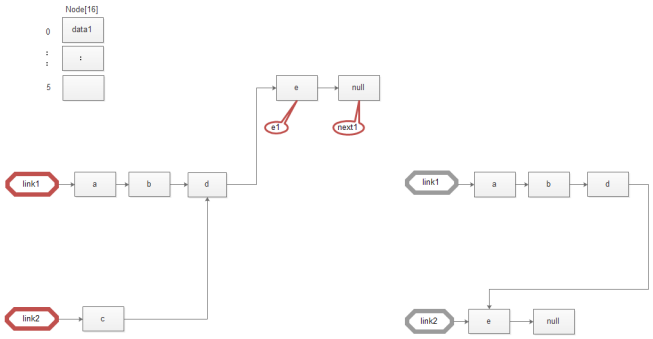


红色为thread1，银灰色为thread2

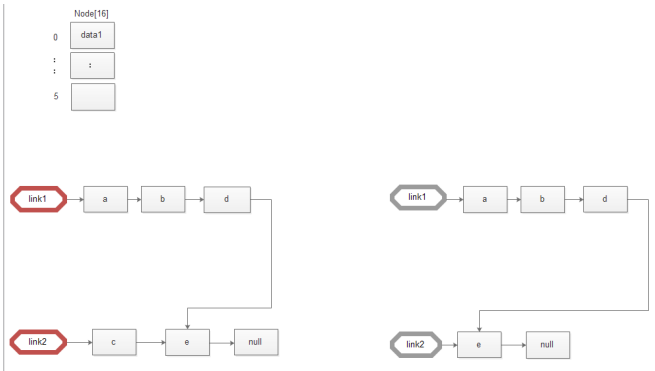
图中红色代表thread1，银灰色代表thread2，此时thread2被挂起，thread1执行到如下图：



此时仔细观察会发现thread2指向的链表少了一个节点c。倘若此时thread1被挂起，thread2执行，直到遍历完整个链表：



此时thread1执行到遍历完整个链表，结果如下图：



此时thread2中少了一个元素，现在如果thread1先将结果赋值给新table，thread2再进行赋值操作，那么thread2就会覆盖thread1的结果。这样就导致resize之后少了一个元素。

如有发现文章中有任何错误，麻烦留言指出。谢谢~

顶 2　　踩 0

- 上一篇　java集合类（七）Set之LinkedHashSet&TreeSet
- 下一篇　Map(二)之LinkedHashMap（java8）

我的同类文章

java (9) java集合类 (4) java8 (1) hashmap (1)

- jvm - ClassLoader 2017-01-02 阅读 2486
- Map(二)之LinkedHashMap (java8) 2016-07-21 阅读 832
- java集合类 (六) Set之HashSet 2016-07-13 阅读 526
- java集合类 (三) List之LinkedList 2016-07-12 阅读 705
- java集合类 (一) 综述 2016-07-11 阅读 341
- 关闭 阅读 2351
- java集合类 (七) Set之LinkedHashSet... 2016-07-13 阅读 677
- java集合类 (五) Vector与ArrayList比较 2016-07-12 阅读 796
- java集合类 (二) List之ArrayList 2016-07-11 阅读 577

参考知识库

算法与数据结构知识库

15802 关注 | 2320 收录

猜你在找

ArcGIS for JavaScript
ArcGIS for javascript 项目实战··
JavaSE高级篇---（IO流+多线程+···
深入Javascript数组视频课程
Java Swing、JDBC开发桌面级应用
深入理解HashMap
HashMap与HashTable一-源代码
java API源码初体验3---collect···
阿里巴巴编程规范-javaMySQL工程
HIBERNATE - 符合Java习惯的关系··

查看评论

SugaryoTT 1楼 2016-09-30 18:00发表

看不懂啊。。。。但还是要给楼主一个赞~~~

chengli_007 Re: 2016-10-18 17:05发表

回复SugaryoTT：哪里不懂，可以探讨哦

发表评论

用户名： jinxin70

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP
IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API
HTML SDK IIS Fedora XML LBS Unity Splashtop 关闭ils QEMU KDE
Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular
Cloud Foundry Redis Scala Django Bootstrap

关闭