# HashMap集合简介

HashMap基于**哈希表**的Map接口实现，是以key-value存储形式存在，即主要用来存放键值对。HashMap 的实现不是同步的，这意味着它不是线程安全的。它的key、value都可以为null。此外，HashMap中的映射不是有序的。

JDK1.8 之前 HashMap 由 数组+链表 组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突(两个对象调用的hashCode方法计算的哈希码值一致导致计算的数组索引值相同)。JDK8之后，当链表长度大于阈值（或者红黑树的边界值，默认为 8）**并且当前数组的长度大于64时**，此时此索引位置上的所有数据改为使用红黑树存储。

补充：将链表转换成红黑树前会判断，即使阈值大于8，但是数组长度小于64，此时并不会将链表变为红黑树。而是选择进行数组扩容。

当然虽然增了红黑树作为底层数据结构，结构变得复杂了，但是阈值大于8并且数组长度大于64时，链表转换为红黑树时，效率也变的相对更高效。

小结：

特点：

1.存取无序的

2.键和值位置都可以是null，但是键位置只能是一个null

3.键位置是唯一的，底层的数据结构控制键的位置

4.jdk1.8前数据结构是：链表 + 数组 jdk1.8之后是：链表 + 数组 + 红黑树

5.阈值(边界值) > 8 并且数组长度大于64，才将链表转换为红黑树，变为红黑树的目的是为了高效的查询。

# HashMap集合底层的数据结构

## 数据结构

数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。数据结构往往同高效的检索算法和索引技术有关。

说白了就是存储数据的一种方式。

在JDK1.8 之前 HashMap 由 数组+链表 数据结构组成的。

在JDK1.8 之后 HashMap 由 数组+链表 +红黑树数据结构组成的。

## HashMap底层的数据结构存储数据的过程

代码

|  |
| --- |
| public static void main(String[] args) {  HashMap<String, Integer> map = new HashMap<>();  map.put("稽哥", 18);  map.put("雷哥", 28);  map.put("吴彦祖", 18);  map.put("张学友", 40);  map.put("郭德纲", 50);  map.put("赵本山", 60);  map.put("肖战", 29);   System.*out*.println("Aa".hashCode());  System.*out*.println("BB".hashCode());  } |

图解添加过程



面试题：HashMap中hash函数是怎么实现的？还有哪些hash函数的实现方式？

对于key的hashCode做hash操作，无符号右移16位然后做异或运算。除此以外，还可以用取余数法、伪随机数法等，但是这些效率都比较低，而无符号右移16我异或运算效率是最高的。

面试题：当两个对象的hashCode相等时会怎么样？

会发生哈希碰撞。若key值内容相同，则替换旧的value，否则连接到链表后面。

链表长度超过8，数组长度超过64的时候，会转成红黑树

面试题：何时发生哈希碰撞和什么是哈希碰撞,如何解决哈希碰撞？

只要两个元素的key计算的哈希值相同就会发生哈希碰撞，jdk8之前使用链表解决哈希碰撞，jdk8之后使用链表+红黑树解决哈希碰撞

面试题：如果两个键的hashcode相同，如何存储键值对？

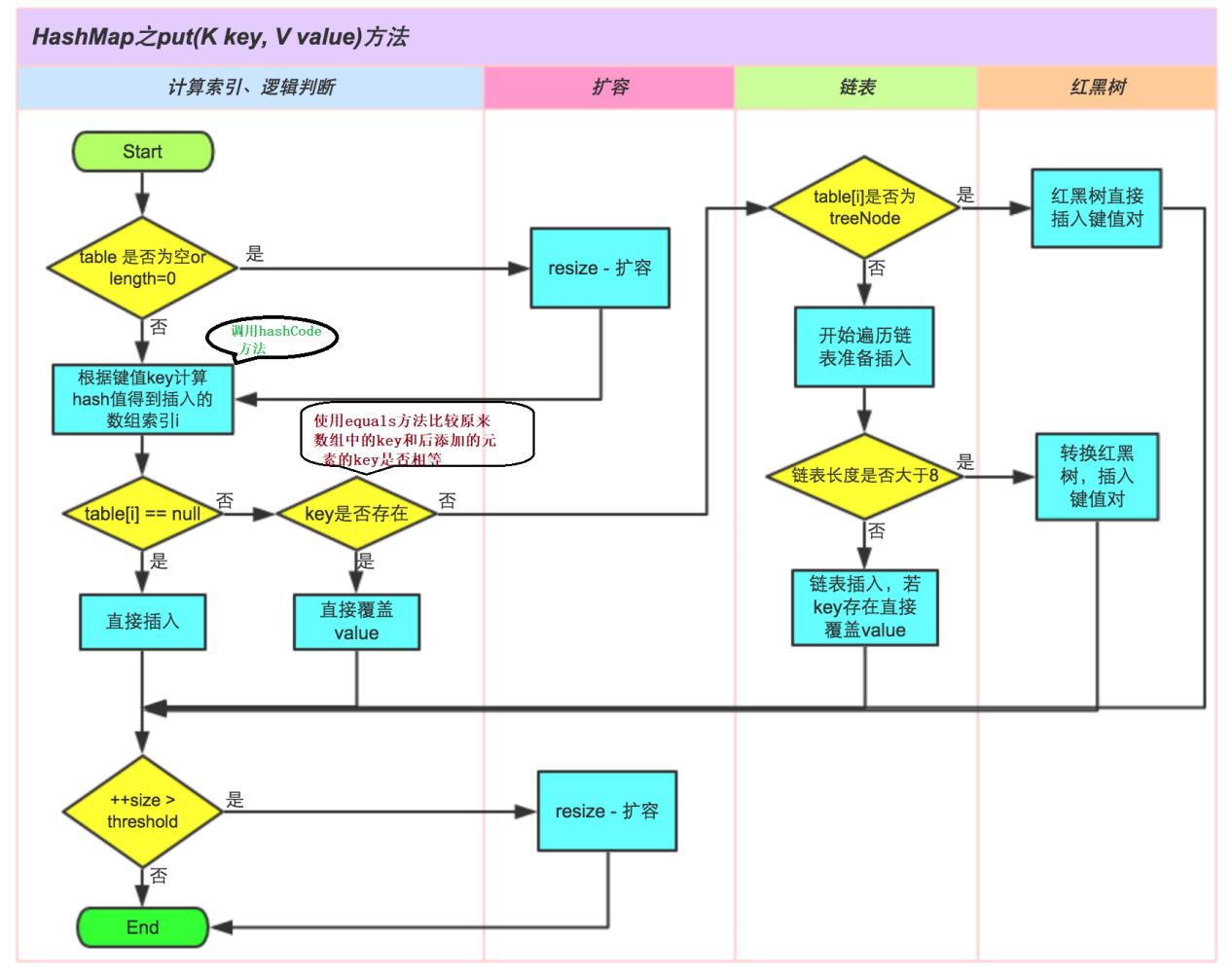
hashCode相同，通过equals方法比较内容是否相同

相同：新的value覆盖旧的value，返回旧的value

不相同：将新的键值对添加到链表中

总结：

上述我们大概阐述了HashMap底层存储数据的方式。为了方便大家更好的理解，我们结合一个存储流程图来进一步说明一下：(jdk8存储过程)



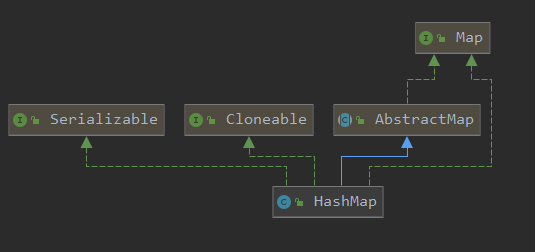
说明：

1.size表示 HashMap中K-V的实时数量， 注意这个不等于数组的长度 。

2.threshold( 临界值) =capacity(容量) \* loadFactor( 加载因子 0.75 )。这个值是当前已占用数组长度的最大值。size超过这个临界值就重新resize(扩容)，扩容后的 HashMap 容量是之前容量的两倍。在我们实际开发中，如果对效率要求很高，应当尽可能避免hashMap的扩容

## HashMap继承关系

HashMap继承关系如下图所示：



说明：

Cloneable 空接口，表示可以克隆。 创建并返回HashMap对象的一个副本。

Serializable 序列化接口。属于标记性接口。HashMap对象可以被序列化和反序列化。

AbstractMap 父类提供了Map实现接口。以最大限度地减少实现此接口所需的工作。

补充：通过上述继承关系我们发现一个很奇怪的现象， 就是HashMap已经继承了AbstractMap而AbstractMap类实现了Map接口，那为什么HashMap还要在实现Map接口呢？同样在ArrayList中LinkedList中都是这种结构。

据 java 集合框架的创始人Josh Bloch描述，这样的写法是一个失误。在java集合框架中，类似这样的写法很多，最开始写java集合框架的时候，他认为这样写，在某些地方可能是有价值的，直到他意识到错了。显然的，JDK的维护者，后来不认为这个小小的失误值得去修改，所以就这样存在下来了。

# HashMap集合类的成员

## 成员变量

**序列化版本号**

|  |
| --- |
| private static final long *serialVersionUID* = 362498820763181265L; |

**集合的初始化容量**

|  |
| --- |
| */\*\*  \* 默认的初始容量 ，=1<<4=16。HashMap的容量必须是2的n次幂  \*/* static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; |

问题： **为什么必须是2的n次幂？如果输入值不是2的幂比如10会怎么样？**

HashMap构造方法还可以指定集合的初始化容量大小：

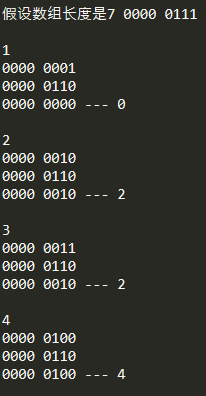
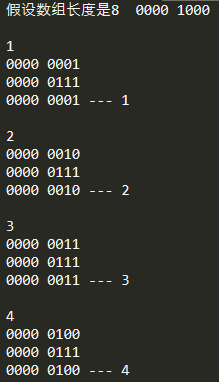
|  |
| --- |
| */\*\*  \* 指定容量去初始化一个HashMap  \*/* public HashMap(int initialCapacity) {  this(initialCapacity, *DEFAULT\_LOAD\_FACTOR*); } |

根据上述讲解我们已经知道，当向HashMap中添加一个元素的时候，需要根据key的hash值，去确定其在数组中的具体位置。 HashMap 为了存取高效，要尽量较少碰撞，就是要尽量把数据分配均匀，每个链表长度大致相同，这个实现就在把数据存到哪个链表中的算法。

这个算法实际就是取模，hash%length，计算机中直接求余效率不如位移运算(这点上述已经讲解)。所以源码中做了优化,使用 hash&(length-1)，**而实际上hash%length等于hash&(length-1)的前提是length是2的n次幂**。

为什么这样能均匀分布减少碰撞呢？2的n次方实际就是1后面n个0，2的n次方-1 实际就是n个1；

举例：



如果不考虑效率，直接求余数的话，就不需要要求长度是2的n次幂

小结：

1. 由上面可以看出，当我们根据key的hash确定其在数组的位置时，如果数组长度为2的n次幂，就可以保证数据的均匀插入 。如果不是2的n次幂，可能数组的一些位置永远不会插入数据，浪费数组的空间，加大了hash碰撞。
2. 另一方面，一般我们可能会想通过 % 求余数来确定位置，这样做其实也是可以的，但是性能不如 & 位与运算。而且当数组长度是2的n次幂时，hash & (length - 1)==hash % length
3. 因此，，HashMap容量是2的n次幂的原因，是为了数据的均匀分布，减少Hash冲突。Hash冲突越大，代表数组中一个链表就越长，这样会降低hashmap的性能。

如果在创建HashMap对象的时候，指定的容量不是2的n次幂，比如10，HashMap会通过一系列位移运算和或运算得到一个2的n次幂，这个数字是离我们指定容量最近的数字

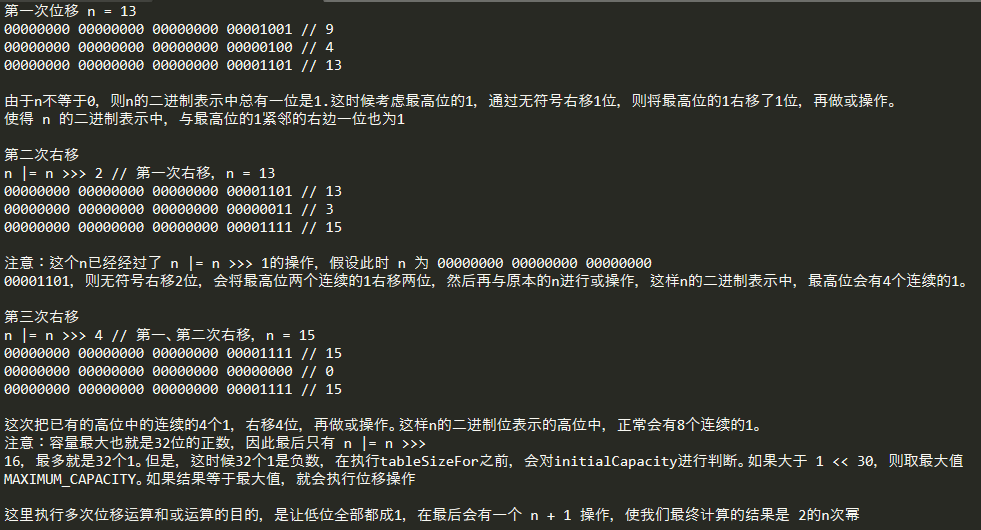
源代码

|  |
| --- |
| */\*\*  \* 用指定的容量和负载因子初始化一个HashMap  \*  \** ***@param*** *initialCapacity 初始值  \** ***@param*** *loadFactor 负载因子  \** ***@throws*** *IllegalArgumentException if the initial capacity is negative  \* or the load factor is nonpositive  \*/* public HashMap(int initialCapacity, float loadFactor) {  if (initialCapacity < 0) {  throw new IllegalArgumentException("Illegal initial capacity: " +  initialCapacity);  }  // 容量不能超过最大容量  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); }  */\*\*  \* 该方法作用是让HashMap的容量永远是2的n次幂  \*/* 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\_CAPACITY* : n + 1; } |

说明：由此可以看到，当在实例化HashMap的时候，如果给定了initialCapacity（假设是10），由于HashMap的initialCapacity必须都是2的n次幂，因此这个方法用于找到大于等于initialCapacity的最小的是2的n次幂。如果initialCapacity已经是2的n次幂，则直接返回这个数。

下面我们分析一下这个算法。

1. 首先，为什么要进行cap - 1 的操作。Int n = cap - 1.这是为了防止cap已经是2的n次幂。如果cap已经是2的n次幂，又没有执行减1的操作，则执行完后面的几条无符号右移操作之后，返回的capacity将是这个数的2倍。
2. 如果这时候n是0，最后返回的是capacity是1，因为最后有个n+1的操作。
3. 按位或运算符 | 运算规则：相同位置的二进制数位上，如果都是0，那么结果就是0，否则为1



**默认的负载因子，默认值是0.75**

|  |
| --- |
| */\*\*  \* 默认加载因子，=0.75  \*/* static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f; |

**使用TreeNode的临界值**

|  |
| --- |
| */\*\*  \* 使用TreeNode的临界值，默认=8  \* 经过大量的计算得知，链表中元素复合泊松分布  \* 0: 0.60653066  \* 1: 0.30326533  \* 2: 0.07581633  \* 3: 0.01263606  \* 4: 0.00157952  \* 5: 0.00015795  \* 6: 0.00001316  \* 7: 0.00000094  \* 8: 0.00000006  \* 上面是链表中，每个节点可能会存放元素的概率  \*   \* 网上还有另一种说法，红黑树的平均查找长度是log(n)，如果长度为8，平均查找长度是log(8) = 3  \* 链表平均查找长度是 n/2，如果长度是8的情况下，8/2=4，效率低于红黑树，所以需要转换为红黑树。  \* 如果链表长度小于等于6， 6/2=3.而log(6) ≈ 2.6，虽然比链表快，但是效率差距并不大  \* 而且，链表转换为红黑树也需要一定的时间，所以这时候并不会转换为红黑树  \*/* static final int *TREEIFY\_THRESHOLD* = 8; |

**集合最大容量**

|  |
| --- |
| */\*\*  \* 最大容量，=1<<30  \*/* static final int *MAXIMUM\_CAPACITY* = 1 << 30; |

**当链表的值小于6则会从红黑树转回链表**

|  |
| --- |
| */\*\*  \* 链表值小于6会从红黑树转回链表  \*/* static final int *UNTREEIFY\_THRESHOLD* = 6; |

**当Map里面的数量超过这个值时，表中的桶才能进行树形化 ，否则桶内元素太多时会扩容，而不是树形化**

|  |
| --- |
| */\*\*  \* 当数组长度大于这个数时才会转红黑树，否则只是扩容  \*/* static final int *MIN\_TREEIFY\_CAPACITY* = 64; |

**table用来初始化(必须是二的n次幂)(重点)**

|  |
| --- |
| */\*\*  \* 实际存储的数组 Entry数组。jdk中称其为 hash桶  \*/* transient Node<K, V>[] table; |

Table在jdk8中我们了解到，是由数组+链表+红黑树来组成结构。Jdk8之前，这个类型叫Entry。实际上也只是改了个名字而已。二者都是实现了一样的接口，都是Map.Entry

**用来存放缓存**

|  |
| --- |
| */\*\*  \* 缓存  \*/* transient Set<Entry<K, V>> entrySet; |

**HashMap中存放元素的个数(重点)**

|  |
| --- |
| */\*\*  \* 实际存储的个数  \*/* transient int size; |

注意：这里的size是标识HashMap中key-value的数量，而不是数组的长度

**用来记录HashMap的修改次数**

|  |
| --- |
| */\*\*  \* map中数据改变次数的统计  \*/* transient int modCount; |

**用来调整大小下一个容量的值计算方式为(容量\*负载因子)**

|  |
| --- |
| */\*\*  \* 临界值，与HashMap扩容相关  \*  \** ***@serial*** *\*/* int threshold; |

**哈希表的加载因子(重点)**

|  |
| --- |
| */\*\*  \* 加载因子，初始值=0.75，与扩容有关  \*  \** ***@serial*** *\*/* final float loadFactor; |

说明

1. loadFactor负载因子，是用来衡量HashMap满的程度，标识HashMap的疏密程度，影响hash操作到同一个数组位置的概率，计算HashMap的实时加载因子的方法为：size / capacity
2. loadFactor太大会导致查找元素效率低，太小导致数组的利用率低，存放的数据会很分散。0.75是官方经过大量的数据测试，得出的最好的数字
3. 当HashMap中容纳的元素超过边界值时，认为HashMap太挤了，就需要扩容。这个扩容的过程涉及到rehash、复制等操作，非常的消耗性能，所以开发中尽量减少扩容的次数，可以通过创建HashMap时指定初始化容量来尽量的避免

比如：我们需要存放1000个元素到HashMap，那么我们可能需要new HashMap(1024)。但是1024\*0.75=768<1000，就会扩容，因此这个时候我们应该new HashMap(2048)

# 构造方法

1.空参构造，构造一个空的HashMap，默认负载因子是0.75，JDK7中，在我们new HashMap的时候，会立即创建Hash桶，而在jdk8中，在new HashMap时，并不会创建数组，而是在put方法中，先判断table是否为空

|  |
| --- |
| */\*\*  \* 用默认的容量去初始化一个HashMap（16）  \*/* public HashMap() {  this.loadFactor = *DEFAULT\_LOAD\_FACTOR*; } |

2.构造一个指定初始容量和默认负载因子的HashMap

|  |
| --- |
| */\*\*  \* 指定容量去初始化一个HashMap  \*/* public HashMap(int initialCapacity) {  this(initialCapacity, *DEFAULT\_LOAD\_FACTOR*); } |

3.构造一个具有指定容量和负载因子的HashMap

|  |
| --- |
| */\*\*  \* 用指定的容量和负载因子初始化一个HashMap  \*  \** ***@param*** *initialCapacity 初始值  \** ***@param*** *loadFactor 负载因子  \** ***@throws*** *IllegalArgumentException if the initial capacity is negative  \* or the load factor is nonpositive  \*/* public HashMap(int initialCapacity, float loadFactor) {  // 判断初始化容量 initialCapacity 是否小于0  if (initialCapacity < 0) {  // 如果小于 0，抛出非法的参数异常  throw new IllegalArgumentException("Illegal initial capacity: " +  initialCapacity);  }  // 判断初始化容量 initialCapacity 是否大于集合的最大容量 MAXIMUM\_CAPACITY  if (initialCapacity > *MAXIMUM\_CAPACITY*) {  // 如果超过最大容量，将最大容量赋值给 initialCapacity  initialCapacity = *MAXIMUM\_CAPACITY*;  }  // 判断加载因子 是否小于等于0，或者是否是一个非法数值  if (loadFactor <= 0 || Float.*isNaN*(loadFactor)) {  // 如果满足上面条件，抛出非法参数异常  throw new IllegalArgumentException("Illegal load factor: " +  loadFactor);  }  // 将指定的负载因子赋值给 loadFactor  this.loadFactor = loadFactor;  /\*  tableSizeFor 判断指定的初始化容量是否为 2 的n次幂，  如果不是，那就变为比指定容量大的最小的2的n次幂。  但是注意，这里计算出初始化容量之后，直接赋值给了threshold  有人认为这是个bug  事实上，在put方法中，会对threshold重新计算  \*/  this.threshold = *tableSizeFor*(initialCapacity); }  */\*\*  \* 该方法作用是让HashMap的容量永远是2的n次幂  \*/* static final int tableSizeFor(int cap) {  int n = cap - 1;  // | （位或运算符）运算规则：两个数都转为二进制，然后从高位开始比较，两个数只要有一个为1则为1，否则就为0。  /\*  >>> 表示符号位也会跟着移动，比如 -1 的最高位是1，表示是个负数，然后右移之后，最高位就是0表示当前是个正数。  所以 -1 >>>1 = 2147483647  >> 表示无符号右移，也就是符号位不变。那么-1 无论移动多少次都是-1  原理就是将最高位 1 右边的所有比特位全置为 1，然后再加 1，最高位进 1，右边的比特位全变成 0，从而得出一个 2 的次幂的值  \*/  n |= n >>> 1;  n |= n >>> 2;  n |= n >>> 4;  n |= n >>> 8;  n |= n >>> 16;  return (n < 0) ? 1 : (n >= *MAXIMUM\_CAPACITY*) ? *MAXIMUM\_CAPACITY* : n + 1; } |

1. 参数是Map的构造方法

|  |
| --- |
| public HashMap(Map<? extends K, ? extends V> m) {  this.loadFactor = *DEFAULT\_LOAD\_FACTOR*;  putMapEntries(m, false); }  final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {  // 获取map的元素个数  int s = m.size();  if (s > 0) {  // 判断 table是否已经初始化  if (table == null) {  // 未初始化， s是m的元素个数  float ft = ((float) s / loadFactor) + 1.0F;  int t = ((ft < (float) *MAXIMUM\_CAPACITY*) ?  (int) ft : *MAXIMUM\_CAPACITY*);  // 判断得到的值是否大于阈值，如果大于阈值，则初始化阈值  if (t > threshold) {  threshold = *tableSizeFor*(t);  }  } else if (s > threshold) {  // 已初始化，并且元素个数大于阈值，进行扩容  resize();  }  // 将m中所有的元素添加到HashMap中  for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {  K key = e.getKey();  V value = e.getValue();  putVal(*hash*(key), key, value, false, evict);  }  } } |

注意：loat ft = ((float) s / loadFactor) + 1.0F; 为什么要 +1

S / loadFactor结果是小数，+1.0F作用是相当于给小数向上取整，尽可能保证更大容量。更大的容量能够减少resize次数。

# 成员方法

## Put方法

put方法是比较复杂的，实现步骤大致如下：

1）先通过hash值计算出key映射到哪个桶；

2）如果桶上没有碰撞冲突，则直接插入；

3）如果出现碰撞冲突了，则需要处理冲突：

​ a:如果该桶使用红黑树处理冲突，则调用红黑树的方法插入数据；

​ b:否则采用传统的链式方法插入。如果链的长度达到临界值，则把链转变为红黑树；

4）如果桶中存在重复的键，则为该键替换新值value；

5）如果size大于阈值threshold，则进行扩容；

具体的方法如下：

|  |
| --- |
| @Override public V put(K key, V value) {  return putVal(*hash*(key), key, value, false, true); } |

说明：

1）HashMap只提供了put用于添加元素，putVal方法只是给put方法调用的一个方法，并没有提供给用户使用。 所以我们重点看putVal方法。

2）我们可以看到在putVal()方法中key在这里执行了一下hash()方法,来看一下Hash方法是如何实现的。

## Hash方法

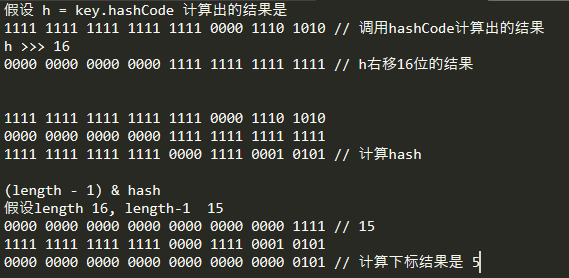
|  |
| --- |
| static final int hash(Object key) {  int h;  /\*  如果key为null  可以看到当key为null的时候也是有哈希值的，返回值是0  如果key不为null  首先计算出key的hashCode，然后赋值给h，接着，h进行无符号右移16位，再进行异或运算  \*/  return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16); } |

从上面我们可以得知，HashMap是支持null key的，而HashTable是直接使用key来获取hashCode，所以key为空会抛异常。

其实我们上面已经解释了为什么HashMap的长度要是2的n次幂，因为HashMap设计的非常巧妙，它通过 hash & (length - 1) 来得到该对象的保存位，等价于hash % length，但是 & 效率比 % 要高

探究为什么h 要右移16位

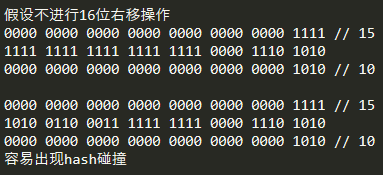
1. key.hashCode，返回散列值也就是hashCode，假设生成一个值
2. Length表示数组长度，16
3. ^ 一或运算符，运算规则：相同的二进制数位上，数字相同，结果为0，否则为1



说明：高16位不变，低16位和高16位进行了一个异或运算。

问题：为什么要怎么操作

如果当length的长度很小，假设是16，那么length-1 ---> 1111 ，这样的值和hashCode直接做按位与操作，实际上只使用了哈希值的后4位，高位将没有任何意义。如果当哈希值高位变化很大，低位变化很小，这样就非常容易造成哈希冲突，所以这里要把高位和低位都利用起来，从而解决这个问题。说白了，这个操作的作用，其实就是为了防止哈希冲突



## putVal

HashMap中put方法的核心方法。向hashMap中添加数据的主要逻辑在这个方法中。

|  |
| --- |
| final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  boolean evict) {  Node<K, V>[] tab;  Node<K, V> p;  // n存放数组长度。i存放key的hash计算后的值  int n, i;  /\*  判断table是否为空  1. table 表示存储在Map中的元素的数组  2. (tab = table) == null 表示将table赋值给tab，并且判断tab是否为null。  3. (n = tab.length) == 0 表示，将tab的长度赋值给n，并判断n 是否等于-  \*/  if ((tab = table) == null || (n = tab.length) == 0) {  // 如果为空就通过resize实例化一个数组  /\*  这里的代码等价于  tab = resize();  n = tab.length  \*/  n = (tab = resize()).length;  }  /\*  i = hash & (n - 1) 计算当前key所在下标，确定在哪个桶中，并将下标赋值给i  p = tab(i) 将该位置的元素赋值给p，并且判断是否为null  \*/  if ((p = tab[i = (n - 1) & hash]) == null) {  // 直接创建一个Node元素，赋值给当前下标位置  tab[i] = newNode(hash, key, value, null);  } else {  // 当前下标位置不为null  // 注意，我们在上面的if中，已经把当前下标位置的元素，赋值给了p  Node<K, V> e;  K k;  /\*  比较桶中第一个元素的hash值和key是否相等。  1. p.hash == hash ：判断第一个元素的hash与我们传进来的hash是否相等  2. ((k = p.key) == key || (key != null && key.equals(k)))  2.1 (k = p.key) == key 将第一个元素的key赋值给k，并且判断是否和我们传进来的key相等  2.2 判断我们传进来的key不等于null，并且key的值和k相等  上面如果都满足的情况下，说明第一个元素的key和我们传进来的key值是相等的  \*/  if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k)))) {  // 将该位置的节点赋值给e  e = p;  } else if (p instanceof TreeNode) {  // 判断当前下标位置的数据类型是否为红黑树  e = ((TreeNode<K, V>) p).putTreeVal(this, tab, hash, key, value);  } else {  // 说明当前元素是个链表  // 不是红黑树，当前下标位置的key也与要插入的key不相等  // 遍历链表  for (int binCount = 0; ; ++binCount) {  /\*  (e = p.next) == null 将p的下一个元素赋值给e，并判断e是否为null  如果等于null，说明当前元素是表尾，已经没有下一个元素了  如果不为null，说明下一个元素还存在，可以继续遍历  \*/  if ((e = p.next) == null) {  // 进入，说明e是表尾  // 直接将数据写到下一个节点  p.next = newNode(hash, key, value, null);  /\*  1. 节点添加完成之后判断此时节点个数是否大于临界值 8，如果大于则将链表转为红黑树。  2. int binCount = 0，表示for循环的初始化值，从0开始计算，记录遍历节点的个数  |- 0表示第一个节点  |- 1表示第二个节点  |- 。。。。  |- 7表示第八个节点  因此这里TREEIFY\_THRESHOLD需要-1  \*/  if (binCount >= *TREEIFY\_THRESHOLD* - 1) {  // 将链表转为红黑树  treeifyBin(tab, hash);  }  break;  }  // 如果当前位置的key与要存放位置的key相同，直接跳出  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k)))) {  /\*  要添加的元素和链表中存在的元素相等了，则跳出for循环，不需要再比较后面的元素了  直接进入下面的if语句去替换e的值  \*/  break;  }  // 说明新添加的元素和当前节点不相同，继续找下一个元素。  p = e;  }  }  // e不为空，说明上面找到了一个去存储Key-Value的Node  if (e != null) {  // 拿到旧Value  V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null) {  // 新的值赋值给节点  e.value = value;  }  afterNodeAccess(e);  // 返回旧value  return oldValue;  }  }  // 统计数据改变次数  ++modCount;  // 当最后一次调整之后的Size大于临界值，就需要调整数组容量  if (++size > threshold) {  resize();  }  afterNodeInsertion(evict);  return null; } |

# 数组扩容 resize

1. 什么时候需要扩容
2. HashMap中的扩容是什么

## 什么时候才需要扩容

当Hashmap中元素个数超过 **数组长度\*负载因子** 就会进行扩容。也就是说，当HashMap数组长度是16的时候，如果元素个数超过 16\*0.75=12 的时候，就会给数组扩容，扩容方式就是将原数组扩大2倍 16\*2=32。然后重新计算每个元素在数组中的位置。而这是个非常消耗性能的过程，所以当我们已经预知到HashMap中元素个数的时候，应当赋予一个基本不可能让这个HashMap扩容的长度。

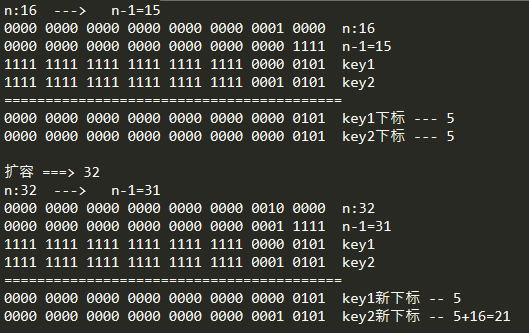
当HashMap中的其中一个链表的对象个数达到了8个，此时如果数组长度没有达到64，那么HashMap也会进行扩容。如果达到了64，那么这个链表会变成红黑树，节点类型由Node变成TreeNode。

## HashMap的扩容是什么

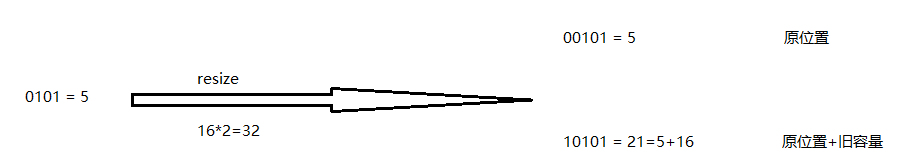
进行扩容，会伴随着一次重新的Hash分配，并且会遍历hash表中的所有元素，是非常耗时的，所以在编写代码的过程中，应当尽量的避免Hashmap的resize。

HashMap在进行扩容的时候，使用rehash非常的巧妙。因为，每次扩容都是翻倍，与原来的(n-1)&hash的结果相比，只是多了一个二进制位，所以节点要么在原来的位置，要么就被分配到 原位置+原容量 这个位置。

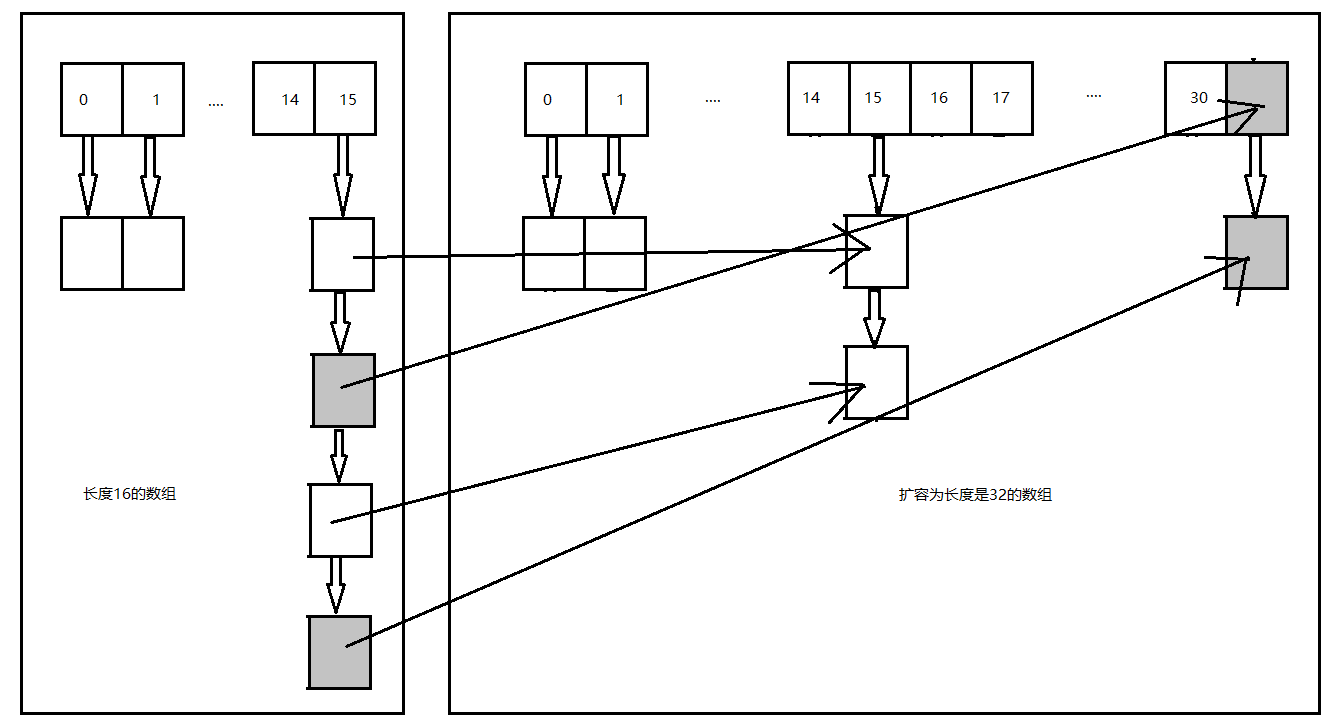
比如我们从16扩容到32



在元素重新计算hash之后，因为n变为2倍，那么n-1的标记范围在高位多1，因此我们的新index就会发生这样的变化。



因此，我们在扩容HashMap的时候，不需要重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就可以了。是0的话下标不变，是1的话下标变为 原位置+旧容量。可以看下图是16扩容到32的示意图。



正是因为这样巧妙地rehash方式，既省去了重新计算hash的时间，而且同时，因为新增的1bit是0还是1可以认为是随机的，在resize的过程中保证了rehash之后每一个桶上的节点数一定小于等于原来桶上的节点数，保证了rehash之后不会出现更严重的hash冲突，均匀的把之前的冲突的节点分散到新的桶中。

resize代码

|  |
| --- |
| */\*\*  \* 数组扩容  \*/* final Node<K, V>[] resize() {  // 先拿到旧的hash桶  Node<K, V>[] oldTab = table;  // 获取未扩容前的数组容量  int oldCap = (oldTab == null) ? 0 : oldTab.length;  // 旧的临界值  int oldThr = threshold;  // 定义新的容量和临界值  int newCap, newThr = 0;  // 旧容量大于0  if (oldCap > 0) {  // 旧的容量如果超过了最大容量  if (oldCap >= *MAXIMUM\_CAPACITY*) {  // 临界值就等于Integer类型最大值  threshold = Integer.*MAX\_VALUE*;  // 不扩容，直接返回就数组  return oldTab;  }  /\*  没超过最大值，数组扩容为原来的2倍  1.(newCap = oldCap << 1) < MAXIMUM\_CAPACITY 扩大到2倍之后赋值给newCap，判断newCap是否小于最大容量  2.oldCap >= DEFAULT\_INITIAL\_CAPACITY 原数组长度大于等于数组初始化长度  \*/  else if ((newCap = oldCap << 1) < *MAXIMUM\_CAPACITY* &&  oldCap >= *DEFAULT\_INITIAL\_CAPACITY*) {  // 当前容量在默认值和最大值的一半之间  // 新的临界值为当前临界值的2倍  newThr = oldThr << 1; // double threshold  }  } else if (oldThr > 0) // initial capacity was placed in threshold  {  // 旧容量为0，当前临界值不为0，让新的临界值等于当前临界值  newCap = oldThr;  } else {  // 当前容量和临界值都为0，让新的容量等于默认值，临界值=初始容量\*加载因子  newCap = *DEFAULT\_INITIAL\_CAPACITY*;  newThr = (int) (*DEFAULT\_LOAD\_FACTOR* \* *DEFAULT\_INITIAL\_CAPACITY*);  }  // 经过上面对新临界值的计算后如果还是0  if (newThr == 0) {  // 计算临界值为新容量 \* 加载因子  float ft = (float) newCap \* loadFactor;  // 判断新容量小于最大值，并且计算出的临界值也小于最大值  // 那么就把计算出的临界值赋值给新临界值。否则新临界值默认为Integer最大值  newThr = (newCap < *MAXIMUM\_CAPACITY* && ft < (float) *MAXIMUM\_CAPACITY* ?  (int) ft : Integer.*MAX\_VALUE*);  }  // 临界值赋值  threshold = newThr;  @SuppressWarnings({"rawtypes", "unchecked"})  // 使用新的容量创建新数组  Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap];  // 赋值给hash桶  table = newTab;  // 下面一堆是复制值  // 如果旧的桶不为空  if (oldTab != null) {  // 遍历旧桶，把旧桶中的元素重新计算下标位置，赋值给新桶  // j 表示数组下标位置  for (int j = 0; j < oldCap; ++j) {  Node<K, V> e;  /\*  (e = oldTab[j]) != null 将旧桶的当前下标位置元素赋值给e，并且e不为null  \*/  if ((e = oldTab[j]) != null) {  // 置空，置空之后原本的这个数据就可以被gc回收  oldTab[j] = null;  // 下一个节点如果为空  if (e.next == null) {  // 如果没有下一个节点，说明不是链表，当前桶上只有一个键值对，直接计算下标后插入  newTab[e.hash & (newCap - 1)] = e;  } else if (e instanceof TreeNode) {  // 节点是红黑树，进行切割操作  ((TreeNode<K, V>) e).split(this, newTab, j, oldCap);  } else { // preserve order  // 到这里说明该位置的元素是链表  /\*  loHead：链表头结点  loTail：数据链表  hiHead：新位置链表头结点  hiTail：新位置数据链表  \*/  Node<K, V> loHead = null, loTail = null;  Node<K, V> hiHead = null, hiTail = null;  Node<K, V> next;  // 循环链表，直到链表末再无节点  do {  // 获取下一个节点  next = e.next;  // 如果这里为true，说明e这个节点在resize之后不需要移动位置  if ((e.hash & oldCap) == 0) {  if (loTail != null) {  loTail.next = e;  } else {  loHead = 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; } |

# HashMap删除

|  |
| --- |
| */\*\*  \* 根据key删除元素  \* 删除是有返回值的  \* 并且返回值是被删除key所对应的value  \*/* @Override public V remove(Object key) {  Node<K, V> e;  return (e = removeNode(*hash*(key), key, null, false, true)) == null ?  null : e.value; }  */\*\*  \* 删除方法的核心逻辑  \*  \** ***@param*** *hash hash for key  \** ***@param*** *key the key  \** ***@param*** *value the value to match if matchValue, else ignored  \** ***@param*** *matchValue if true only remove if value is equal  \** ***@param*** *movable if false do not move other nodes while removing  \** ***@return*** *the node, or null if none  \*/* final Node<K, V> removeNode(int hash, Object key, Object value,  boolean matchValue, boolean movable) {  Node<K, V>[] tab;  Node<K, V> p;  int n, index;  /\*  1. (tab = table) != null 把hash桶赋值给tab，并且判断tab是否为nul  2. (n = tab.length) > 0 获取tab的长度，赋值给n，判断n是否大于0  3. (p = tab[index = (n - 1) & hash]) != null 根据hash计算索引位置，赋值给index  并从tab中取出该位置的元素，赋值给p，并判断，p不为null  \*/  if ((tab = table) != null && (n = tab.length) > 0 &&  (p = tab[index = (n - 1) & hash]) != null) {  // 进入这里面，说明hash桶不为空，并且当前key所在位置的元素不为空  Node<K, V> node = null, e;  K k;  V v;  if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k)))) {  // 当前第一个位置的元素就是我们要找的元素  node = p;  }   // 取出p的下一个节点赋值给e，并且e不为空  else if ((e = p.next) != null) {  if (p instanceof TreeNode) {  node = ((TreeNode<K, V>) p).getTreeNode(hash, key);  } else {  do {  if (e.hash == hash &&  ((k = e.key) == key ||  (key != null && key.equals(k)))) {  node = e;  break;  }  p = e;  } while ((e = e.next) != null);  }  }  // 判断node不为空，  if (node != null && (!matchValue || (v = node.value) == value ||  (value != null && value.equals(v)))) {  if (node instanceof TreeNode) {  ((TreeNode<K, V>) node).removeTreeNode(this, tab, movable);  } else if (node == p) {  // node==p，说明node是第一个节点，那么直接将下一个节点赋值给当前下标  tab[index] = node.next;  } else {  p.next = node.next;  }  ++modCount;  --size;  afterNodeRemoval(node);  return node;  }  }  return null; } |

# Get方法

|  |
| --- |
| */\*\*  \* Returns the value to which the specified key is mapped,  \* or {****@code*** *null} if this map contains no mapping for the key.  \*  \* <p>More formally, if this map contains a mapping from a key  \* {****@code*** *k} to a value {****@code*** *v} such that {****@code*** *(key==null ? k==null :  \* key.equals(k))}, then this method returns {****@code*** *v}; otherwise  \* it returns {****@code*** *null}. (There can be at most one such mapping.)  \*  \* <p>A return value of {****@code*** *null} does not <i>necessarily</i>  \* indicate that the map contains no mapping for the key; it's also  \* possible that the map explicitly maps the key to {****@code*** *null}.  \* The {****@link*** *#containsKey containsKey} operation may be used to  \* distinguish these two cases.  \*  \** ***@see*** *#put(Object, Object)  \*/* @Override public V get(Object key) {  Node<K, V> e;  return (e = getNode(*hash*(key), key)) == null ? null : e.value; }  */\*\*  \* Implements Map.get and related methods.  \*  \** ***@param*** *hash hash for key  \** ***@param*** *key the key  \** ***@return*** *the node, or null if none  \*/* final Node<K, V> getNode(int hash, Object key) {  Node<K, V>[] tab;  // first存放对应下标位置的第一个元素  Node<K, V> first, e;  int n;  K k;  /\*  1.(tab = table) != null 把table赋值给tab，并且判断tab不为空  2.(n = tab.length) > 0 把tab的长度赋值给n，并且判断n大于0  3.(first = tab[(n - 1) & hash]) != null 根据传进来的hash计算下标位置，取出该下标位置的元素赋值给first，并且判断first不为空  \*/  if ((tab = table) != null && (n = tab.length) > 0 &&  (first = tab[(n - 1) & hash]) != null) {  // 下标位置第一个元素的key就是我们要找的key  if (first.hash == hash && // always check first node  ((k = first.key) == key || (key != null && key.equals(k)))) {  return first;  }  // 获取下一个节点赋值给e，并且判断e不为空  if ((e = first.next) != null) {  if (first instanceof TreeNode) {  // 如果是红黑树，就用红黑树方式取值  return ((TreeNode<K, V>) first).getTreeNode(hash, key);  }  // 遍历链表直到下一个节点不存在为止  do {  // 找到对应的key的位置  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k)))) {  return e;  }  } while ((e = e.next) != null);  }  }  return null; } |

# HashMap遍历方式

1. 分别遍历key和value

|  |
| --- |
| */\*\*  \* 分别遍历key和value  \*/* @Test public void testMap1() {  HashMap<String, Integer> map = getMap();  for (String key : map.keySet()) {  System.*out*.println(key);  }  for (Integer value : map.values()) {  System.*out*.println(value);  } } |

1. 使用iterator迭代器迭代

|  |
| --- |
| @Test public void testIterator() {  HashMap<String, Integer> map = getMap();  Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();  while (iterator.hasNext()) {  Map.Entry<String, Integer> entry = iterator.next();  System.*out*.println(entry.getKey() + ":" + entry.getValue());  } } |

1. 通过get方式

|  |
| --- |
| @Test public void testGet() {  HashMap<String, Integer> map = getMap();  Set<String> keySet = map.keySet();  for (String key : keySet) {  System.*out*.println(key + ":" + map.get(key));  } } |

说明：根据阿里开发手册，不建议这种方式，因为要迭代多次。keySet一次，get一次。

1. Jdk8以后使用Map接口中的一个默认方法

|  |
| --- |
| @Test public void testForeach() {  HashMap<String, Integer> map = getMap();  map.forEach((key, value) -> {  System.*out*.println(key + ":" + value);  }); } |

# HashMap使用细节

1. hashmap初始化问题描述

如果我们确切的知道我们有多少键值对需要存储，那么我们在初始化hashmap的时候就应该指定它的容量，以防止hashmap自动扩容，影响使用效率。

默认情况下HashMap容量是16，如果用户通过构造方法指定了一个数字作为容量，那么hashmap会选择大于等于该数字的第一个2的n次幂作为容量。。

《阿里巴巴Java开发手册》中建议我们使用hashmap的初始化容量。



1. Hashmap中容量的初始化

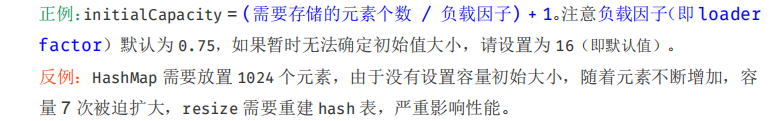
当我们使用HashMap(int initialCapacity)来初始化容量的时候，jdk会默认给我吗计算一个相对合理的值来当做出事容量。那么，我们是不是直接把元素个数作为initialCapacity就可以了呢？

答案是否定的。因为在我们使用HashMap的过程中，随着元素的数量不断增大，hashmap会不断地进行扩容。并且扩容条件是元素个数=数组长度\*0.75。比如我们需要存放1000个元素，那么我们设置1000会有两个不合理之处。

（1）1000不是2的n次幂，设置为1000hashmap会给我们计算成1024

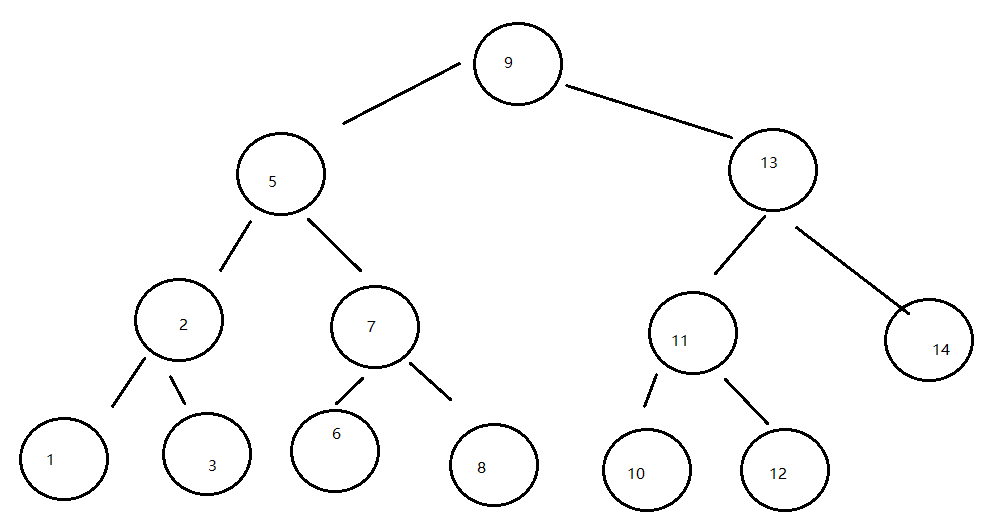
（2）1024虽然是2的n次幂，但是1024\*0.75 < 1000，因此当我们使用的过程中，肯定会出现扩容，造成性能上的浪费

因此我们需要设置为2048.



# 红黑树

## 二叉查找树



上面这张图就是个二叉查找树。二叉查找树满足下面几条特征。

1. 左子树上所有节点的值均小于或者等于它的根节点的值
2. 右子树上所有节点的值均大于或者等于它的根节点的值
3. 左右子树也分别为二叉查找树

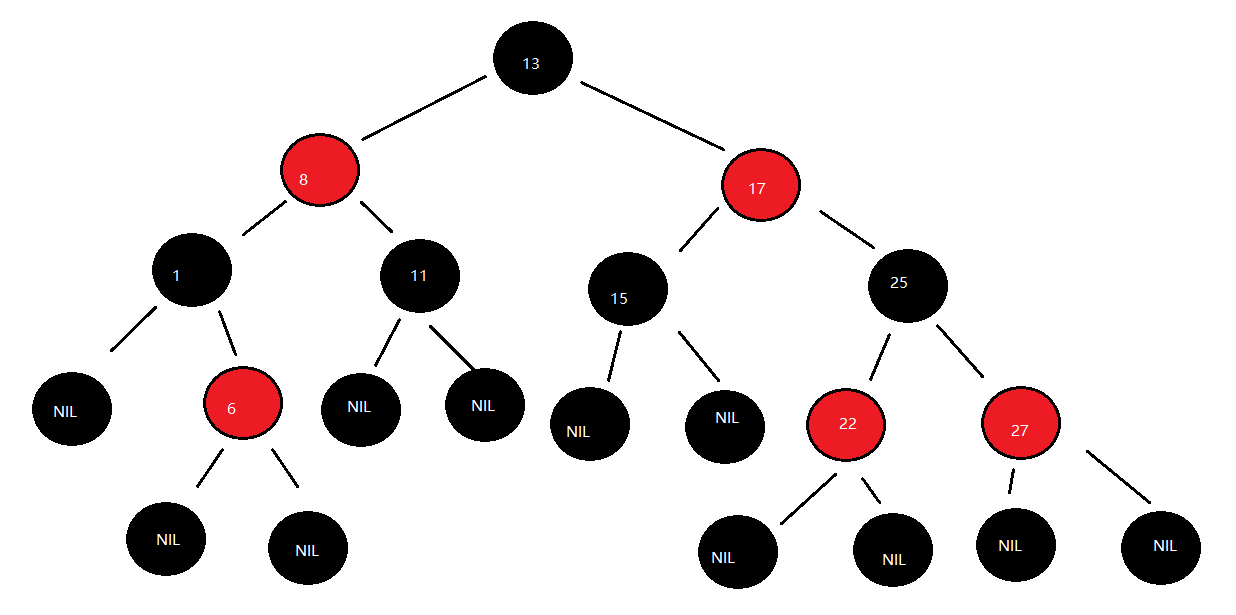
既然名字中带有“查找”，那么它是怎么查找的呢？比如我们要查找10这个元素，首先找到根节点，然后根据1、2特性，10>9，那么继续从右边节点查找，10<13，那么继续从左边节点查找，10<11，继续查左边节点，找到了10这个节点。

## 红黑树

上面我们说到了二叉查找树的思想，那么我们思考一个问题，如果我们要在9这个节点插入7、6、5、4、3，一个比一个小，就会成一条直线，也就是成了线性的查询。为了解决这个情况，就需要使用红黑树了。

红黑树是一种自平衡的二叉查找树，每个节点都带有颜色属性，颜色是红色或者黑色。在二叉查找树的特征以外，任何一条红黑树都有以下额外的特性：

1. 节点是红色或者黑色
2. 根节点一定是黑色
3. 每个叶子结点（NIL节点）是黑色的
4. 每个红色节点的两个子节点都是黑色的（从每个叶子到根的所有路径上不可能有两个连续的红色节点）
5. 从任意一个节点到其每个叶子节点的所有路径包含相同数目的黑色节点



### 红黑树查找

因为红黑树是一个自平衡的二叉查找树，查询操作不会破坏红黑树的平衡，所以查找和二叉查找树的查询方式没有区别。

1. 从根节点开始，把根节点设置为当前节点。
2. 若当前节点为空，则返回null。
3. 若当前节点不为空，用当前节点的key和查找key做比较。
4. 若当前节点的key等于要查找的key，那么该key就是查找目标，返回当前节点。
5. 若当前节点key大于查找的key，把当前节点的左子节点设置为当前节点，重复2.
6. 若当前节点key小于查找的key，把当前节点的右子节点设置为当前节点，重复2

|  |
| --- |
| final TreeNode<K, V> find(int h, Object k, Class<?> kc) {  TreeNode<K, V> p = this;  do {  int ph, dir;  K pk;  TreeNode<K, V> pl = p.left, pr = p.right, q;  if ((ph = p.hash) > h) {  p = pl;  } else if (ph < h) {  p = pr;  } else if ((pk = p.key) == k || (k != null && k.equals(pk))) {  return p;  } else if (pl == null) {  p = pr;  } else if (pr == null) {  p = pl;  } else if ((kc != null ||  (kc = *comparableClassFor*(k)) != null) &&  (dir = *compareComparables*(kc, k, pk)) != 0) {  p = (dir < 0) ? pl : pr;  } else if ((q = pr.find(h, k, kc)) != null) {  return q;  } else {  p = pl;  }  } while (p != null);  return null; } |

### treeifyBin方法

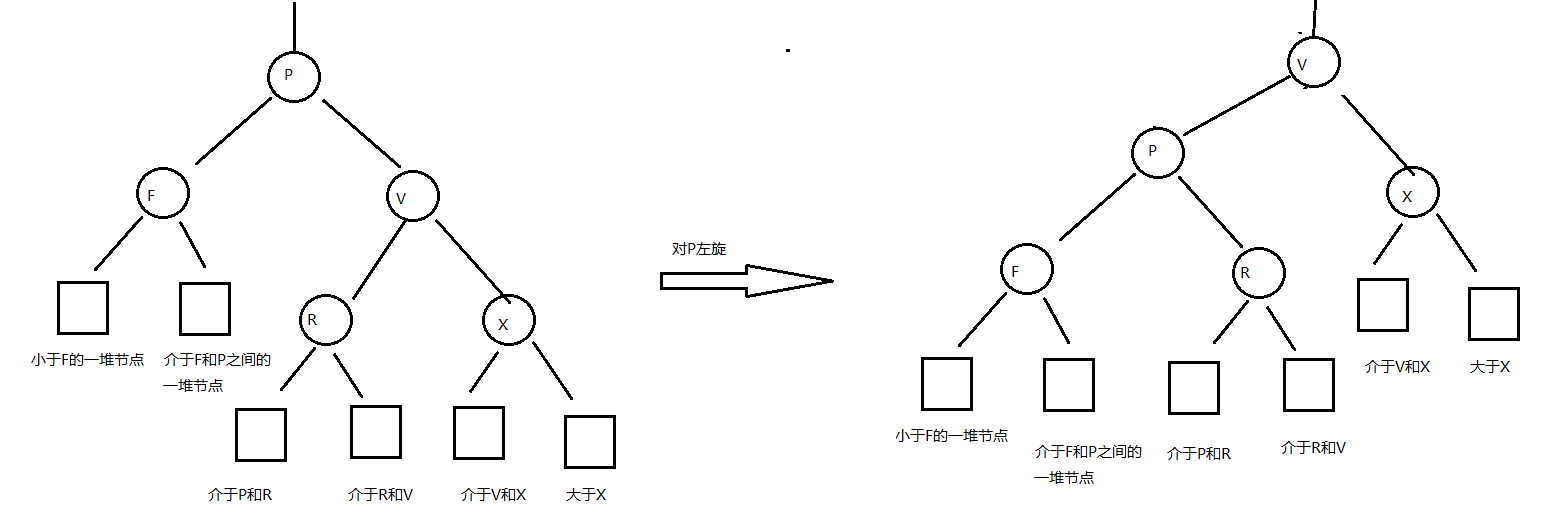
在前面分析put方法的时候，节点添加完成之后就会判断此时节点个数是否大于8，如果大于则将链表转换为红黑树。

|  |
| --- |
| */\*\*  \* 替换指定哈希表的所引出桶中的所有节点，除非表太小，否则将修改大小，  \*/* final void treeifyBin(Node<K, V>[] tab, int hash) {  int n, index;  Node<K, V> e;  /\*  如果当前数组为空，或者数组长度小于进行树形化的阈值（64）就去扩容，而不是转换为红黑树。  目的：如果数组很小，那么转换为红黑树然后遍历效率要低一些，这时候进行扩容，那么重新计算哈希值  链表的长度就有可能变短了，数据会放到数组中，这样相对来说效率高一些  \*/  if (tab == null || (n = tab.length) < *MIN\_TREEIFY\_CAPACITY*) {  resize();  } else if ((e = tab[index = (n - 1) & hash]) != null) {  /\*  1.执行到这里说明哈希表中数组长度大于阈值64，开始进行树形化。  2.(e = tab[index = (n - 1) & hash]) != null 通过当前key的hash计算当前key所在的下标位置，取出来赋值给e，判断e不为空  \*/  // hd：红黑树的头结点。tl：红黑树的尾结点  TreeNode<K, V> hd = null, tl = null;  do {  // 重新创建一个树节点，内容和当前链表节点e一致  TreeNode<K, V> p = replacementTreeNode(e, null);  if (tl == null) {  // 将新创建的p节点赋值给红黑树的头结点  hd = p;  } else {  /\*  p.prev = tl 将上一个节点p赋值给现在的p的前一个节点  tl.next = p 将现在的节点p作为树的为节点的下一个节点  \*/  p.prev = tl;  tl.next = p;  }  tl = p;  } while ((e = e.next) != null);  /\*  让桶中第一个元素即数组中的元素指向新建的红黑树的节点，以后这个桶里的元素就是红黑树，而不是链表  \*/  if ((tab[index] = hd) != null) {  hd.treeify(tab);  }  } } |

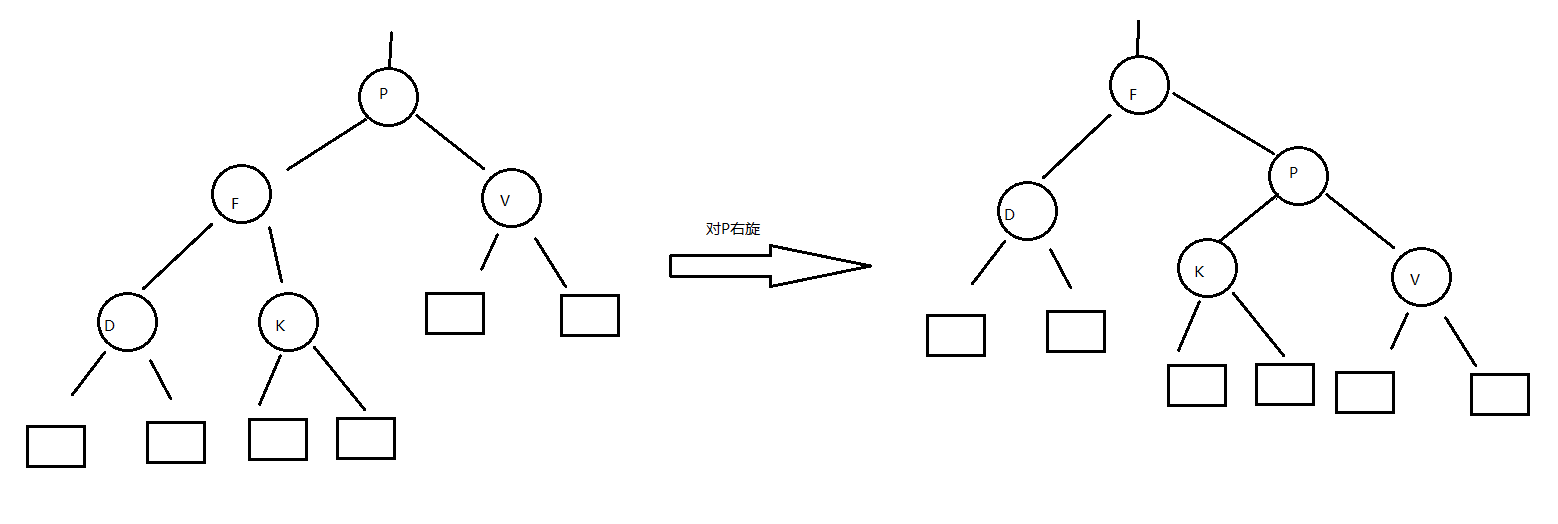
### 左旋、右旋、变色

前面我们讲到红黑树能自平衡，它考的就是左旋、右旋、变色三个操作。

左旋：以某个节点作为支点，，其右节点变为旋转节点的父节点，右节点的左节点变为旋转节点的右节点，其余不变。



右旋：以某个节点作为支点，其左节点变为旋转节点的父节点，左节点的右节点变为旋转节点的左节点，其余不变。



变色：节点的颜色由红变黑或者由黑变红的过程