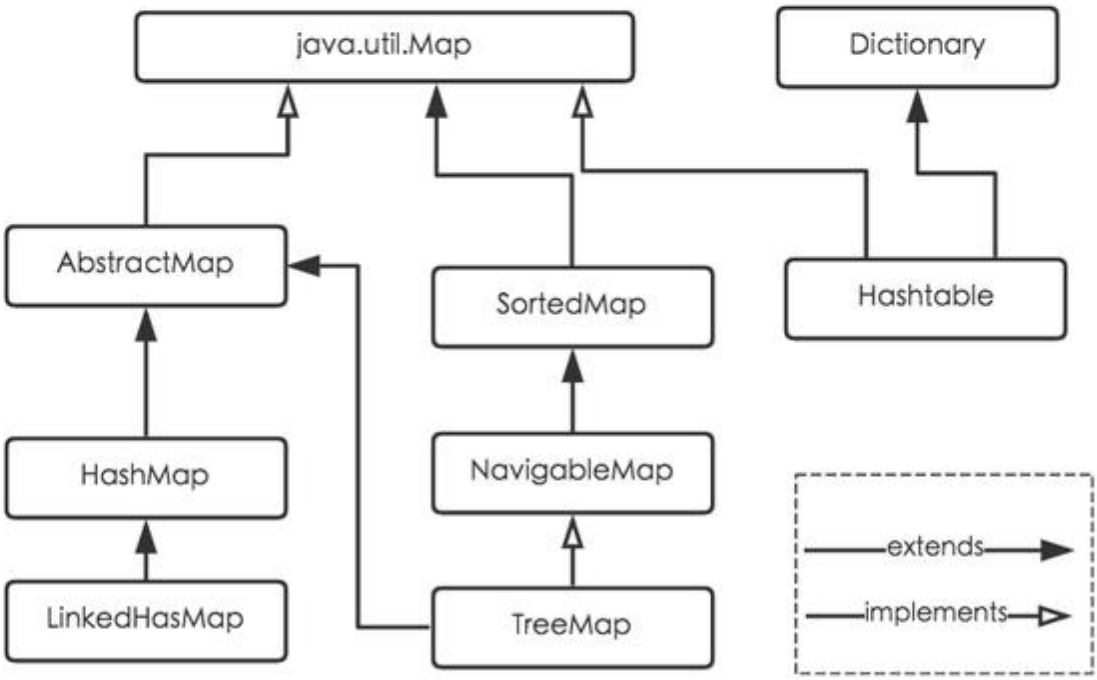
摘要

HashMap 是 Java 程序员使用频率最高的用于映射(键值对)处理的数据类型。

随着 JDK（Java Developmet Kit）版本的更新，JDK1.8 对 HashMap 底层的实现进行了优化，例如引入红黑树的数据结构和扩容的优化等。本文结合 JDK1.7 和 JDK1.8 的区别，深入探讨 HashMap 的结构实现和功能原理。

简介

Java 为数据结构中的映射定义了一个接口 java.util.Map，此接口主要有四个常用的实现类，分别是 HashMap、Hashtable、LinkedHashMap 和 TreeMap，类继承关系如下图所示：



下面针对各个实现类的特点做一些说明：

(1) HashMap：它根据键的 hashCode 值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。

HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。

HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。

(2) Hashtable：Hashtable 是遗留类，很多映射的常用功能与 HashMap 类似，不同的是它承自 Dictionary 类，并且是线程安全的，任一时间只有一个线程能写 Hashtable，并发性不如 ConcurrentHashMap，因为 ConcurrentHashMap 引入了分段锁。Hashtable 不建议在新代码中使用，不需要线程安全的场合可以用 HashMap 替换，需要线程安全的场合可以用 ConcurrentHashMap 替换。

(3) LinkedHashMap：LinkedHashMap 是 HashMap 的一个子类，保存了记录的插入顺序，在用 Iterator 遍历 LinkedHashMap 时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

(4) TreeMap：TreeMap 实现 SortedMap 接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 TreeMap 时，得到的记录是排过序的。如果使用排序的映射，建议使用 TreeMap。在使用 TreeMap 时，key 必须实现 Comparable 接口或者在构造

TreeMap 传入自定义的 Comparator，否则会在运行时抛出

java.lang.ClassCastException 类型的异常。

对于上述四种 Map 类型的类，要求映射中的 key 是不可变对象。不可变对象是

该对象在创建后它的哈希值不会被改变。如果对象的哈希值发生变化，Map 对象很可能就定位不到映射的位置了。

通过上面的比较，我们知道了 HashMap 是 Java 的 Map 家族中一个普通成员，鉴于它可以满足大多数场景的使用条件，所以是使用频度最高的一个。下文我们主要结合源码，从存储结构、常用方法分析、扩容以及安全性等方面深入讲解 HashMap 的工作原理。

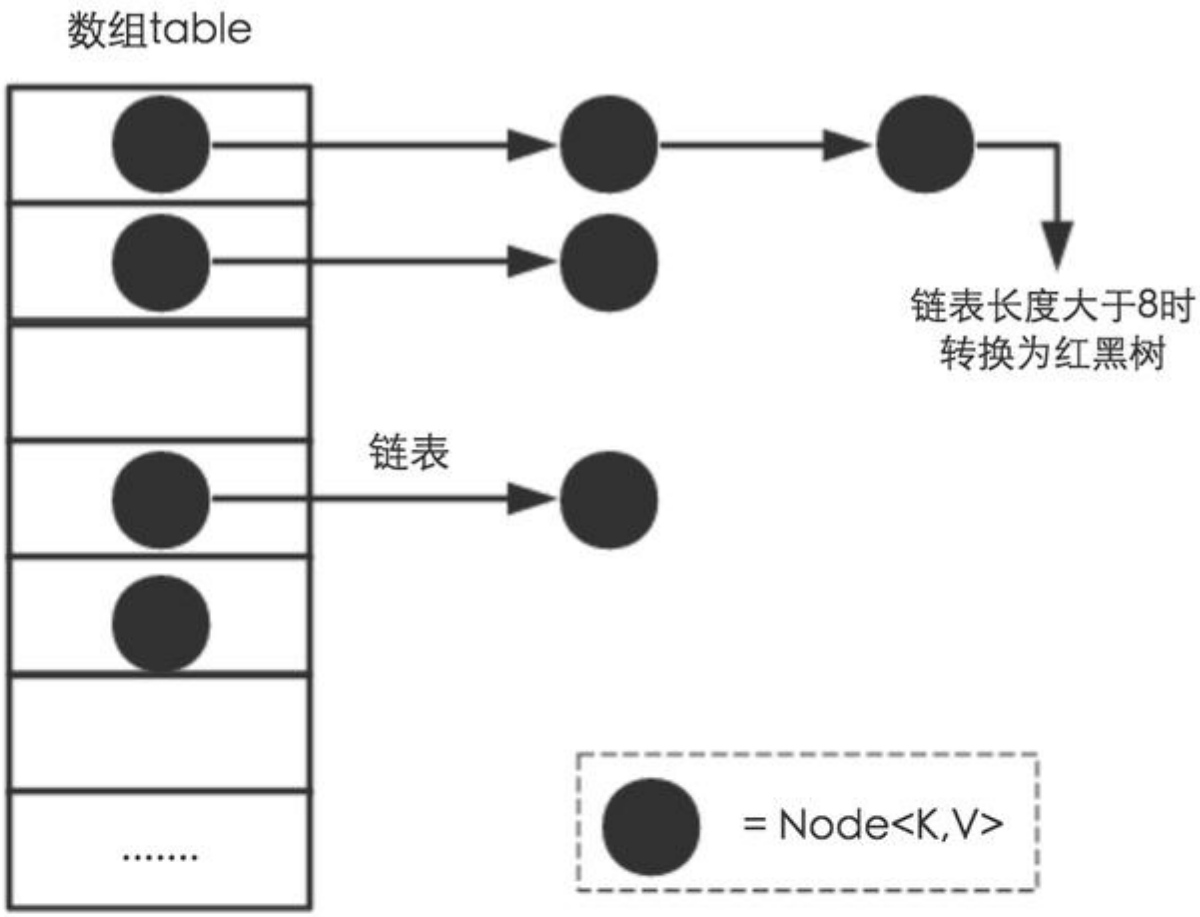
内部实现

搞清楚 HashMap，首先需要知道 HashMap 是什么，即它的存储结构-字段；

其次弄明白它能干什么，即它的功能实现-方法。下面我们针对这两个方面详细展开讲解。

存储结构-字段

从结构实现来讲，HashMap 是:**数组+链表+红黑树**（JDK1.8 增加了红黑树部分）实现的，如下如所示。



这里需要讲明白两个问题：数据底层具体存储的是什么？这样的存储方式有什么优点呢？

(1) 从源码可知，*HashMap* 类中有一个非常重要的字段，就是 *Node[] table*，即哈希桶数组，明显它是一个 *Node* 的数组。我们来看 *Node[JDK1.8]*是何物。

```
static class Node<K,V> implements Map.Entry<K,V> {
 final int hash; //用来定位数组索引位置
 final K key;
 V value;
 Node<K,V> next; //链表的下一个 node
 Node(int hash, K key, V value, Node<K,V> next) { ... }
 public final K getKey(){ ... }
```

```
public final V getValue() { ... }
public final String toString() { ... }
public final int hashCode() { ... }
public final V setValue(V newValue) { ... }
public final boolean equals(Object o) { ... }
}
```

Node 是 HashMap 的一个内部类，实现了 Map.Entry 接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个 Node 对象。

(2) HashMap 就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java 中 HashMap 采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被 Hash 后，得到数组下标，把数据放在对应下标元素的链表上。例如程序执行下面代码：

```
map.put("咕泡","IT 进阶站");
```

系统将调用"咕泡"这个 key 的 hashCode()方法得到其 hashCode 值（该方法适用于每个 Java 对象），然后再通过 Hash 算法的后两步运算（**高位运算和取模运算**，下文有介绍）**来定位该键值对的存储位置，有时两个 key 会定位到相同的位置，表示发生了 Hash 碰撞**。当然 Hash 算法计算结果越分散均匀，Hash 碰撞的概率就越小，map 的存取效率就会越高。

如果哈希桶数组很大，即使较差的 Hash 算法也会比较分散，如果哈希桶数组数组很小，即使好的 Hash 算法也会出现较多碰撞，所以就需要在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的 hash 算法减少 Hash 碰撞。那么通过什么方式来控制 map 使得 Hash

碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢？答案就是好的 Hash 算法和扩容机制。

在理解 Hash 和扩容流程之前，我们得先了解下 HashMap 的几个字段。从 HashMap 的默认构造函数源码可知，构造函数就是对下面几个字段进行初始化，源码如下：

```
int threshold; // 所能容纳的 key-value 对极限
final float loadFactor; // 负载因子
int modCount;
int size;
```

首先，Node[] table 的**初始化长度 length(默认值是 16)**，**Load factor 为负载因子(默认值是 0.75)**，threshold 是 HashMap 所能容纳的最大数据量的 Node(键值对)个数。threshold = length * Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

结合负载因子的定义公式可知，threshold 就是在此 Load factor 和 length(数组长度)对应下允许的最大元素数目，超过这个数目就重新 resize(扩容)，扩容后的 HashMap 容量是之前容量的两倍。默认的负载因子 0.75 是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子 Load factor 的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子 loadFactor 的值，这个值可以大于 1。

size 这个字段其实很好理解，就是 HashMap 中实际存在的键值对数量。注意和 table 的长度 length、容纳最大键值对数量 threshold 的区别。而 modCount 字段主要用来记录 HashMap 内部结构发生变化的次数，主要用于迭代的快速

失败。强调一点，内部结构发生变化指的是结构发生变化，例如 put 新键值对，但是某个 key 对应的 value 值被覆盖不属于结构变化。

在 HashMap 中，哈希桶数组 table 的长度 length 大小必须为 2 的 n 次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。

这里存在一个问题，即使负载因子和 Hash 算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响 HashMap 的性能。于是，在 JDK1.8 版本中，对数据结构做了进一步的优化，引入了红黑树。而当**链表长度太长（默认超过 8）时，链表就转换为红黑树**，利用红黑树快速增删改查的特点提高 HashMap 的性能，其中会用到红黑树的插入、删除、查找等算法。本文不再对红黑树展开讨论，想了解更多红黑树数据结构的工作原理。

功能实现-方法

HashMap 的内部功能实现很多，本文主要从：

1).根据 key 获取哈希桶数组索引位置

2).put 方法的详细执行

3).扩容过程三个具有代表性的点深入展开讲解。

1. 确定哈希桶数组索引位置

不管增加、删除、查找键值对，定位到哈希桶数组的位置都是很关键的第一步。前面说过 HashMap 的数据结构是数组和链表的结合，所以我们当然希望这个 HashMap 里面的元素位置尽量分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用 hash 算法求得这个位置的时候，马上就可以知道对应位

置的元素就是我们要的，不用遍历链表，大大优化了查询的效率。HashMap 定位数组索引位置，直接决定了 hash 方法的离散性能。先看看源码的实现(方法一+方法二):

```
方法一：
static final int hash(Object key) { //jdk1.8 & jdk1.7
 int h;
 // h = key.hashCode() 为第一步 取 hashCode 值
 // h ^ (h >>> 16) 为第二步 高位参与运算
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
方法二：
static int indexFor(int h, int length) { //jdk1.7 的源码，jdk1.8 没有这个方法，但是实现原理一样的
 return h & (length-1); //第三步 取模运算
}
```

这里的 Hash 算法本质上就是三步：**取 key 的 hashCode 值、高位运算、取模运算**。

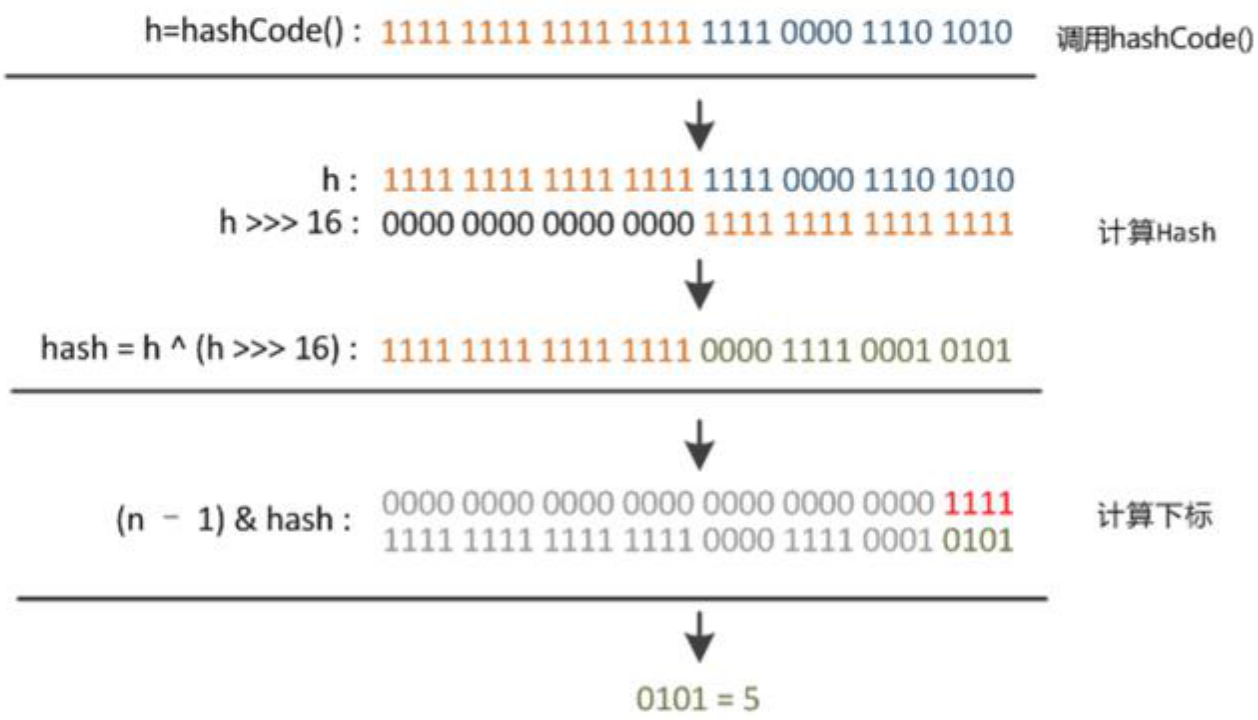
对于任意给定的对象，只要它的 hashCode()返回值相同，那么程序调用方法一所计算得到的 Hash 码值总是相同的。我们首先想到的就是把 hash 值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，模运算的消耗还是比较大的，在 HashMap 中是这样做的：调用方法二来计算该对象应该保存在 table 数组的哪个索引处。

这个方法非常巧妙，它通过 h & (table.length -1)来得到该对象的保存位，而 HashMap 底层数组的长度总是 2 的 n 次方，这是 HashMap 在速度上的优化。

当 length 总是 2 的 n 次方时，h& (length-1)运算等价于对 length 取模，也就是 h%length，但是&比%具有更高的效率。

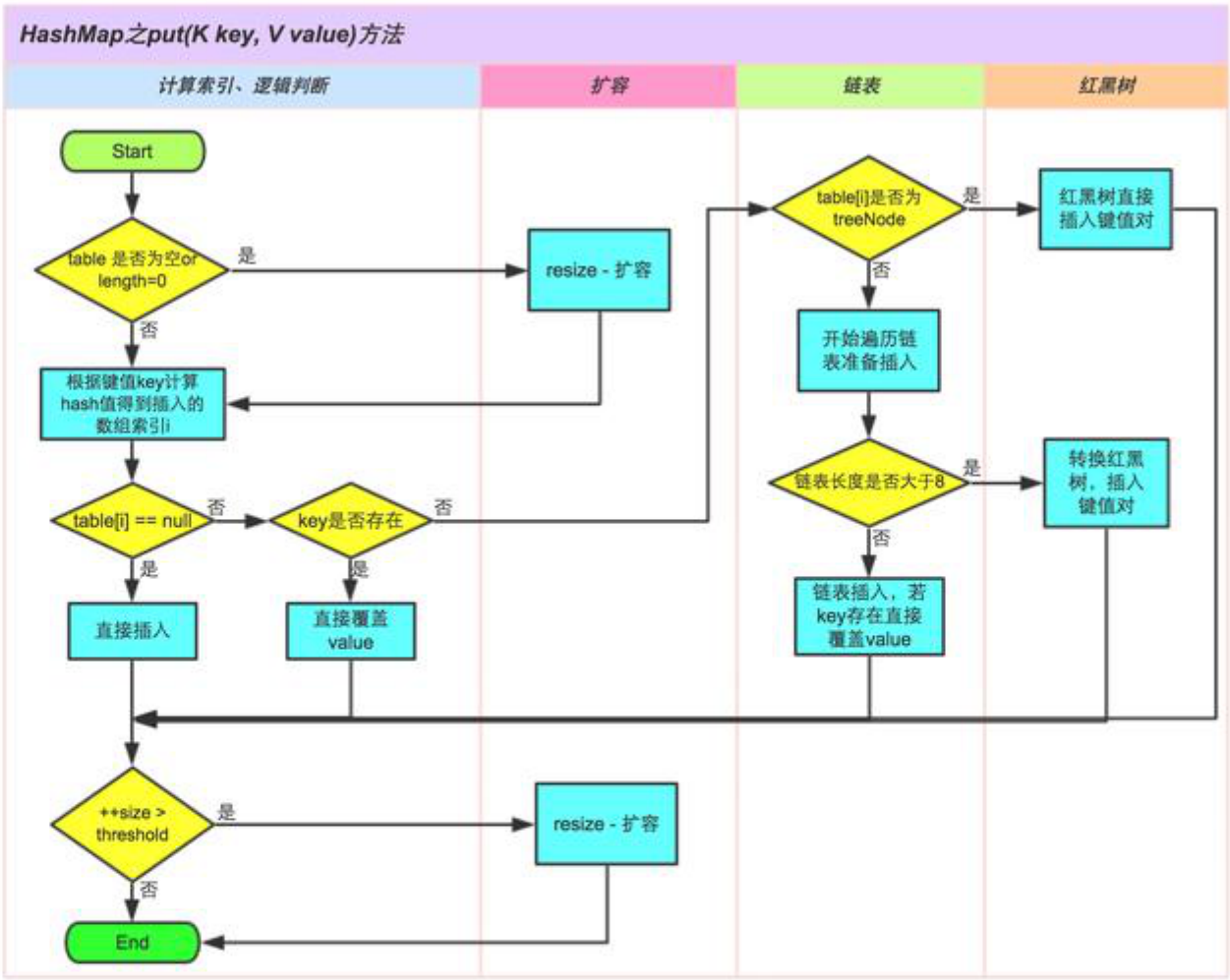
在 JDK1.8 的实现中，优化了高位运算的算法，通过 hashCode()的高 16 位异或低 16 位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组 table 的 length 比较小的时候，也能保证考虑到高低 Bit 都参与到 Hash 的计算中，同时不会有太大的开销。

下面举例说明下，n 为 table 的长度。



2. 分析 HashMap 的 put 方法

HashMap 的 put 方法执行过程可以通过下图来理解，自己有兴趣可以去对比源码更清楚地研究学习。



①.判断键值对数组 table[i]是否为空或为 null，否则执行 resize()进行扩容；

②.根据键值 key 计算 hash 值得到插入的数组索引 i，如果 table[i]==null，直接新建节点添加，转向⑥，如果 table[i]不为空，转向③；

③.判断 table[i]的首个元素是否和 key 一样，如果相同直接覆盖 value，否则转向④，这里的相同指的是 hashCode 以及 equals；

④.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，否则转向⑤；

⑤.遍历 table[i]，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，否则进行链表的插入操作；遍历过程中若发现 key 已经存在直接覆盖 value 即可；

⑥.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，如果超过，进行扩容。

3. 扩容机制

扩容(resize)就是重新计算容量，向 HashMap 对象里不停的添加元素，而 HashMap 对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然 Java 里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

我们分析下 resize 的源码，鉴于 JDK1.8 融入了红黑树，较复杂，为了便于理解我们仍然使用 JDK1.7 的代码，好理解一些，本质上区别不大。

```
1 void resize(int newCapacity) { //传入新的容量
2 Entry[] oldTable = table; //引用扩容前的 Entry 数组
3 int oldCapacity = oldTable.length;
4 if (oldCapacity == MAXIMUM_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)了
```

```
 5 threshold = Integer.MAX_VALUE; //修改阈值为 int 的最大值(2^31-1)，这样以后就不会扩容了
 6 return;
 7 }
 8
 9 Entry[] newTable = new Entry[newCapacity]; //初始化一个新的 Entry 数组
10 transfer(newTable); //！！将数据转移到新的 Entry 数组里
11 table = newTable; //HashMap 的 table 属性引用新的 Entry 数组
12 threshold = (int)(newCapacity * loadFactor);//修改阈值
13 }
```

这里就是使用一个容量更大的数组来代替已有的容量小的数组，*transfer()*方法将原有 *Entry* 数组的元素拷贝到新的 *Entry* 数组里。

```
 1 void transfer(Entry[] newTable) {
 2 Entry[] src = table; //src 引用了旧的 Entry 数组
 3 int newCapacity = newTable.length;
 4 for (int j = 0; j < src.length; j++) { //遍历旧的 Entry 数组
 5 Entry<K,V> e = src[j]; //取得旧 Entry 数组的每个元素
 6 if (e != null) {
 7 src[j] = null;//释放旧 Entry 数组的对象引用（for 循环后，旧的 Entry 数组不再引用任何对象）
 8 do {
 9 Entry<K,V> next = e.next;
10 int i = indexFor(e.hash, newCapacity); //！！重新计算每个元素在数组中的位置
11 e.next = newTable[i]; //标记[1]
12 newTable[i] = e; //将元素放在数组上
13 e = next; //访问下一个 Entry 链上的元素
14 } while (e != null);
15 }
```

```
16 }
17 }
```

newTable[i]的引用赋给了 e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到 Entry 链的尾部(如果发生了 hash 冲突的话），这一点和 Jdk1.8 有区别。

在旧数组中同一条 Entry 链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

线程安全性

在多线程使用场景中，应该尽量避免使用线程不安全的 HashMap，而使用线程安全的 ConcurrentHashMap。

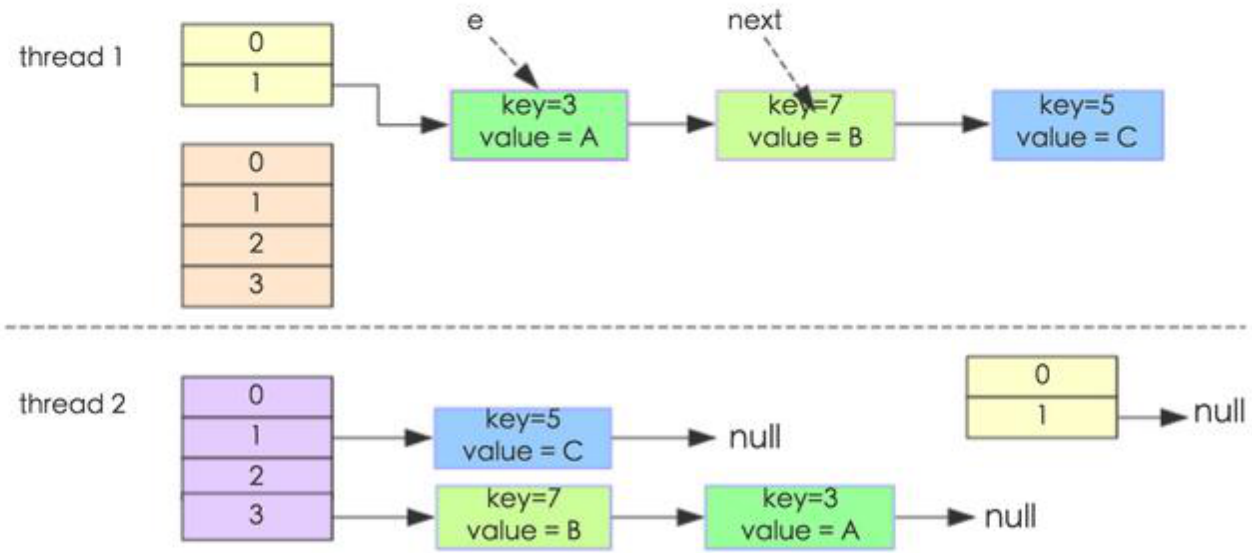
那么为什么说 HashMap 是线程不安全的，下面举例子说明在并发的多线程使用场景中使用 HashMap 可能造成死循环。代码例子如下(便于理解，仍然使用 JDK1.7 的环境)：

```
public class HashMapInfiniteLoop {
 private static HashMap<Integer,String> map = new HashMap<Integer,Strin
g>(2, 0.75f);
 public static void main(String[] args) {
 map.put(5,  "C");
 new Thread("Thread1") {
 public void run() {
 map.put(7, "B");
 System.out.println(map);
 };
 }.start();
 new Thread("Thread2") {
 public void run() {
```

```
map.put(3, "A");
System.out.println(map);
};
}.start();
}
}
```

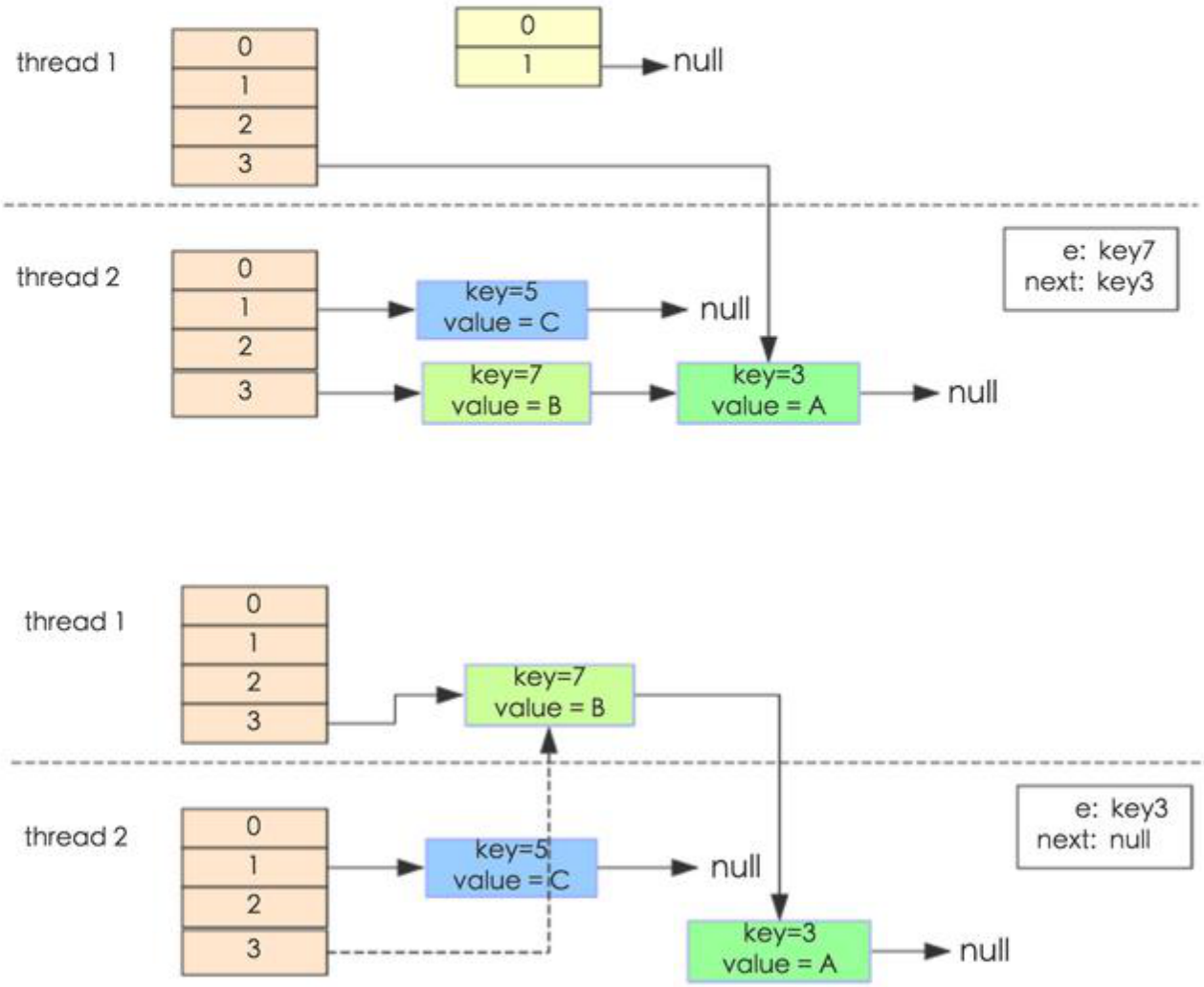
其中，map 初始化为一个长度为 2 的数组，loadFactor=0.75，threshold=2*0.75=1，也就是说当 put 第二个 key 的时候，map 就需要进行 resize。

通过设置断点让线程 1 和线程 2 同时 debug 到 transfer 方法(3.3 小节代码块)的首行。注意此时两个线程已经成功添加数据。放开 thread1 的断点至 transfer 方法的“Entry next = e.next;” 这一行；然后放开线程 2 的的断点，让线程 2 进行 resize。结果如下图。

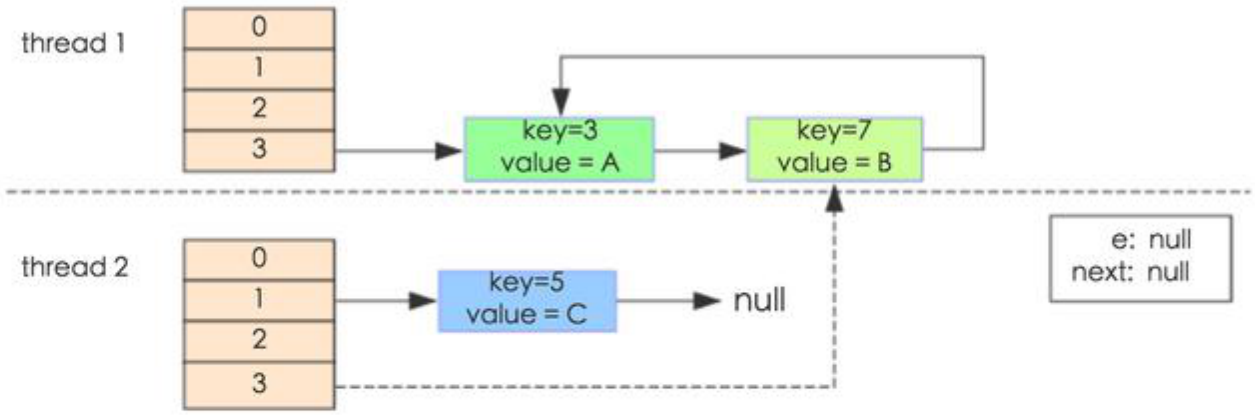


注意，Thread1 的 e 指向了 key(3)，而 next 指向了 key(7)，其在线程二 rehash 后，指向了线程二重组后的链表。

线程一被调度回来执行，先是执行 newTalbe[i] = e， 然后是 e = next，导致了 e 指向了 key(7)，而下一次循环的 next = e.next 导致了 next 指向了 key(3)。



e.next = newTable[i] 导致 key(3).next 指向了 key(7)。注意：此时的 key(7).next 已经指向了 key(3)， 环形链表就这样出现了。



于是，当我们用线程一调用 map.get(11)时，悲剧就出现了——Infinite Loop。

JDK1.8 与 JDK1.7 的性能对比

HashMap 中，如果 key 经过 hash 算法得出的数组索引位置全部不相同，即 Hash 算法非常好，那样的话，getKey 方法的时间复杂度就是 O(1)，如果 Hash 算法技术的结果碰撞非常多，假如 Hash 算极其差，所有的 Hash 算法结果得出的索引位置一样，那样所有的键值对都集中到一个桶中，或者在一个链表中，或者在一个红黑树中，时间复杂度分别为 O(n)和 O(lgn)。 鉴于 JDK1.8 做了多方面的优化，总体性能优于 JDK1.7，下面我们从两个方面用例子证明这一点。

Hash 较均匀的情况

为了便于测试，我们先写一个类 Key，如下：

```
class Key implements Comparable<Key> {
 private final int value;
 Key(int value) {
 this.value = value;
 }
 @Override
 public int compareTo(Key o) {
 return Integer.compare(this.value, o.value);
 }
 @Override
 public boolean equals(Object o) {
 if (this == o) return true;
 if (o == null || getClass() != o.getClass())
 return false;
 Key key = (Key) o;
 return value == key.value;
 }
 @Override
 public int hashCode() {
 return value;
 }
}
```

这个类复写了 *equals* 方法，并且提供了相当好的 *hashCode* 函数，任何一个值的 *hashCode* 都不会相同，因为直接使用 *value* 当做 *hashcode*。为了避免频繁的 *GC*，我将不变的 *Key* 实例缓存了起来，而不是一遍一遍的创建它们。代码如下：

```
public class Keys {
 public static final int MAX_KEY = 10_000_000;
```

```
private static final Key[] KEYS_CACHE = new Key[MAX_KEY];
static {
for (int i = 0; i < MAX_KEY; ++i) {
KEYS_CACHE[i] = new Key(i);
}
}
public static Key of(int value) {
return KEYS_CACHE[value];
}
}
```

现在开始我们的试验，测试需要做的仅仅是，创建不同 *size* 的 *HashMap*（*1*、*10*、*100*、*……10000000*），屏蔽了扩容的情况，代码如下：

```
static void test(int mapSize) {
HashMap<Key, Integer> map = new HashMap<Key,Integer>(mapSize);
for (int i = 0; i < mapSize; ++i) {
map.put(Keys.of(i), i);
}
long beginTime = System.nanoTime(); //获取纳秒
for (int i = 0; i < mapSize; i++) {
map.get(Keys.of(i));
}
long endTime = System.nanoTime();
System.out.println(endTime - beginTime);
}
public static void main(String[] args) {
for(int i=10;i<= 1000 0000;i*= 10){
test(i);
}
```

```
}
```

在测试中会查找不同的值，然后度量花费的时间，为了计算 getKey 的平均时间，我们遍历所有的 get 方法，计算总的时间，除以 key 的数量，计算一个平均值，主要用来比较，绝对值可能会受很多环境因素的影响。结果如下：

map 的size大小	10	100	1000	10000	10 0000	100 0000	1000 0000
JDK1.7 get方法平均时间(ns)	900	540	570	285	55	6.9	8.1
JDK1.8 get方法平均时间(ns)	705	400	120	68	15	6.25	6.8

通过观测测试结果可知，JDK1.8 的性能要高于 JDK1.7 15%以上，在某些 size 的区域上，甚至高于 100%。由于 Hash 算法较均匀，JDK1.8 引入的红黑树效果不明显，下面我们看看 Hash 不均匀的的情况。

Hash 极不均匀的情况

假设我们又一个非常差的 Key，它们所有的实例都返回相同的 hashCode 值。这是使用 HashMap 最坏的情况。代码修改如下：

```
class Key implements Comparable<Key> {
//...
@Override
public int hashCode() {
return 1;
}
```

```
}
```

仍然执行 main 方法，得出的结果如下表所示：

map 的size大小	10	100	1000	10000	10 0000	100 0000	1000 0
JDK1.7 get方法平均时间(ns)	2100	12960	3700	21000	17200	36000	---
JDK1.8 get方法平均时间(ns)	1960	3340	1470	720	190	230	220

从表中结果中可知，**随着 size 的变大，JDK1.7 的花费时间是增长的趋势，而 JDK1.8 是明显的降低趋势，并且呈现对数增长稳定。当一个链表太长的时候，HashMap 会动态的将它替换成一个红黑树**，这话的话会将时间复杂度从 O(n) 降为 O(logn)。hash 算法均匀和不均匀所花费的时间明显也不相同，这两种情况的相对比较，可以说明一个好的 hash 算法的重要性。

小结

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用 HashMap 的时候，估算 map 的大小，初始化的时候给一个大致的数值，避免 map 进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于 1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap 是线程不安全的，不要在并发的环境中同时操作 HashMap，建议使用 ConcurrentHashMap。

(4) JDK1.8 引入红黑树大程度优化了 HashMap 的性能。

(5) HashMap 的性能提升仅仅是 JDK1.8 的冰山一角。