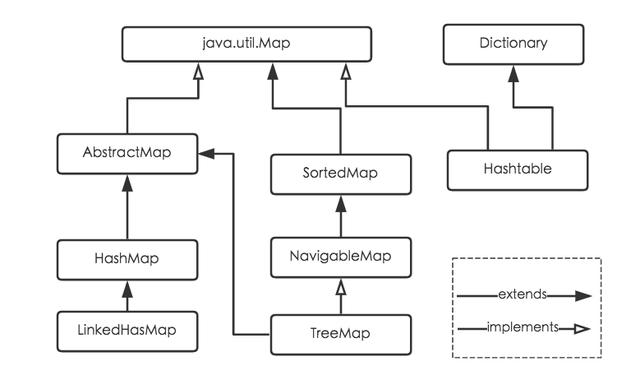
Java为数据结构中的映射定义了一个接口java.util.Map，此接口主要有四个常用的实现类，分别是HashMap、Hashtable、LinkedHashMap和TreeMap，类继承关系如下图所示：



(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。

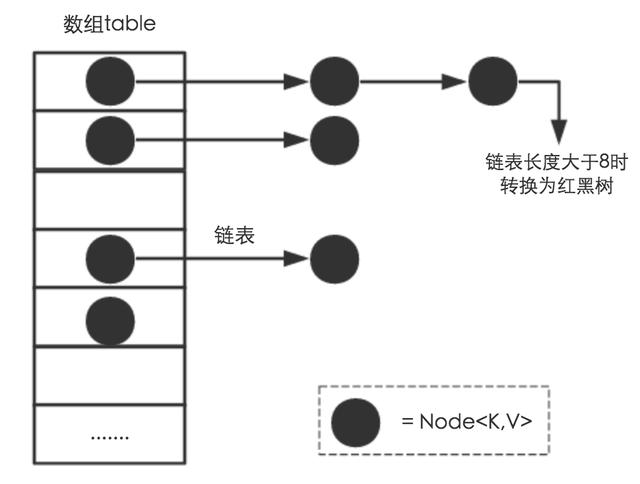
(2) Hashtable：Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。Hashtable不建议在新代码中使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

(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。

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。



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。

链表：

链表是一种数据结构，和数组同级。比如，Java中我们使用的ArrayList，其实现原理是数组。而LinkedList的实现原理就是链表了。链表在进行循环遍历时效率不高，但是插入和删除时优势明显。下面对单向链表做一个介绍。

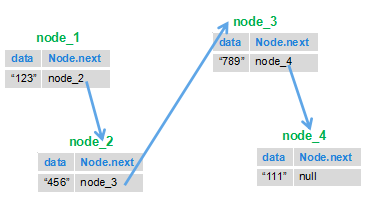


单向链表是一种线性表，实际上是由节点（Node）组成的，一个链表拥有不定数量的节点。其数据在内存中存储是不连续的，它存储的数据分散在内存中，每个结点只能也只有它能知道下一个结点的存储位置。由N各节点（Node）组成单向链表，每一个Node记录本Node的数据及下一个Node。向外暴露的只有一个头节点（Head），我们对链表的所有操作，都是直接或者间接地通过其头节点来进行的。

存储是分散的，每一个节点只要记录下一节点，就把所有数据串了起来，形成了一个单向链表。



节点（Node）是由一个需要储存的对象及对下一个节点的引用组成的。也就是说，节点拥有两个成员：储存的对象、对下一个节点的引用。下面图是具体的说明：



线性结构,顾名思义,就是像线一样的结构,数据元素在逻辑上一个连着一个,可以通过当前的位置找到下一个,甚至是之后的所有位置的元素。

线性结构分为数组和链表:

它不仅在逻辑上是连续的,在物理上也是连续的,一个接着一个(物理存储上).2)我定义一个数组,必须事先给他分配好几个长度的内存空间。(静态存储)

链表只是在逻辑上市连续的(满足线性结构),但是在物理上它并不是连续,它可以通过每个节点的指针找到下一个元素。2)定义链表的同时,并不需要事先分配好内存空间可以在程序运行的时候分配内存(动态存储)。

非线性结构:和线性结构相反,用这种结构存储的数据元素在逻辑上是不连续的,也就是无法通过当前位置找到下一个元素。

**二叉查找树（BST）具备什么特性呢？**

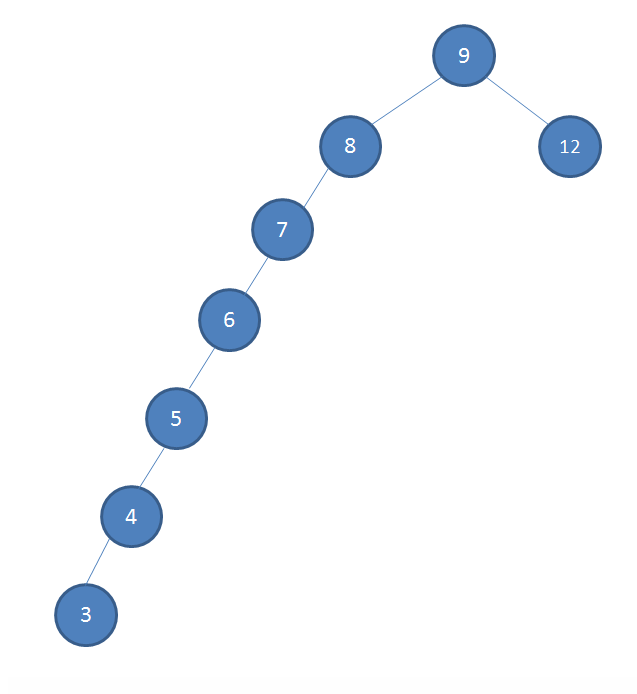
1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3.左、右子树也分别为二叉排序树。

这样的数据结构可以提高数据的查找效率。查找所需的最大次数，为树的深度。

也会存在问题，多次插入新比根节点小的数据，会导致一直往一边添加。



成为了线性。

红黑树就是为了解决这种问题的。

1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

红黑树的旋转

红黑树的应用

TreeMap TreeSet,JDK1.8HaseMap

# [Java中HashMap的实现原理](https://www.cnblogs.com/yuanblog/p/4441017.html)

<https://www.cnblogs.com/yuanblog/p/4441017.html>

### ****1、关于hashCode****

1. hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的
2. 如果两个对象相同，就是适用于equals(java.lang.Object) 方法，那么这两个对象的hashCode一定要相同
3. 如果对象的equals方法被重写，那么对象的hashCode也尽量重写，并且产生hashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点
4. 两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object) 方法，只能够说明这两个对象在散列存储结构中，如Hashtable，他们“存放在同一个篮子里“

**再归纳一下就是hashCode是用于查找存储位置使用的，而equals是用于比较两个对象的是否相等的。**

1. hashcode是用来查找的，如果你学过数据结构就应该知道，在查找和排序这一章有  
   例如内存中有这样的位置  
   0  1  2  3  4  5  6  7   
   而我有个类，这个类有个字段叫ID,我要把这个类存放在以上8个位置之一，如果不用hashcode而任意存放，那么当查找时就需要到这八个位置里挨个去找，或者用二分法一类的算法。  
   但如果用hashcode那就会使效率提高很多。  
   我们这个类中有个字段叫ID,那么我们就定义我们的hashcode为ID％8，然后把我们的类存放在取得得余数那个位置。比如我们的ID为9，9除8的余数为1，那么我们就把该类存在1这个位置，如果ID是13，求得的余数是5，那么我们就把该类放在5这个位置。这样，以后在查找该类时就可以通过ID除 8求余数直接找到存放的位置了。  
   2.但是如果两个类有相同的hashcode怎么办那（我们假设上面的类的ID不是唯一的），例如9除以8和17除以8的余数都是1，那么这是不是合法的，回答是：可以这样。那么如何判断呢？在这个时候就需要定义 equals了。  
   也就是说，我们先通过 hashcode来判断两个类是否存放某个桶里，但这个桶里可能有很多类，那么我们就需要再通过 equals 来在这个桶里找到我们要的类。  
   那么。重写了equals()，为什么还要重写hashCode()呢？  
   想想，你要在一个桶里找东西，你必须先要找到这个桶啊，你不通过重写hashcode()来找到桶，光重写equals()有什么用啊

### ****2、关于equals****

1.equals和==  
==用于比较引用和比较基本数据类型时具有不同的功能：  
比较基本数据类型，如果两个值相同，则结果为true  
而在比较引用时，如果引用指向内存中的同一对象，结果为true;

equals()作为方法，实现对象的比较。由于==运算符不允许我们进行覆盖，也就是说它限制了我们的表达。因此我们复写equals()方法，达到比较对象内容是否相同的目的。而这些通过==运算符是做不到的。

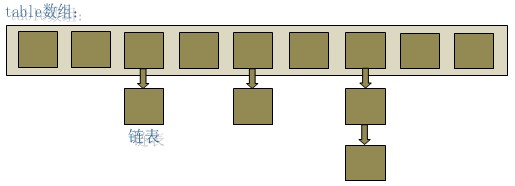
2.object类的equals()方法的比较规则为：如果两个对象的类型一致，并且内容一致，则返回true,这些类有：  
java.io.file,java.util.Date,java.lang.string,包装类（Integer,Double等）  
String s1=new String("abc");  
String s2=new String("abc");  
System.out.println(s1==s2);  
System.out.println(s1.equals(s2));  
运行结果为false true

## 二、HashMap的实现原理

### 1.    HashMap概述

    HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

    在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。



从上图中可以看出，HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个数组。

其中Java源码如下：

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry[] table;

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

final int hash;

……

}

可以看出，Entry就是数组中的元素，每个 Map.Entry 其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。

### 2、HashMap实现存储和读取

public V put(K key, V value) {

// HashMap允许存放null键和null值。

// 当key为null时，调用putForNullKey方法，将value放置在数组第一个位置。

if (key == null)

return putForNullKey(value);

// 根据key的keyCode重新计算hash值。

int hash = hash(key.hashCode());

// 搜索指定hash值在对应table中的索引。

int i = indexFor(hash, table.length);

// 如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素。

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

// 如果发现已有该键值，则存储新的值，并返回原始值

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

// 如果i索引处的Entry为null，表明此处还没有Entry。

modCount++;

// 将key、value添加到i索引处。

addEntry(hash, key, value, i);

return null;

}根据hash值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

hash(int h)方法根据key的hashCode重新计算一次散列。此算法加入了高位计算，防止低位不变，高位变化时，造成的hash冲突。

根据上面 put 方法的源代码可以看出，当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部——具体说明继续看 addEntry() 方法的说明。

2）读取

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

3）归纳起来简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

### 3、HashMap的resize

       当hashmap中的元素越来越多的时候，碰撞的几率也就越来越高（因为数组的长度是固定的），所以为了提高查询的效率，就要对hashmap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，所以这是一个通用的操作，很多人对它的性能表示过怀疑，不过想想我们的“均摊”原理，就释然了，而在hashmap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

       那么hashmap什么时候进行扩容呢？当hashmap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，也就是说，默认情况下，数组大小为16，那么当hashmap中元素个数超过16\*0.75=12的时候，就把数组的大小扩展为2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知hashmap中元素的个数，那么预设元素的个数能够有效的提高hashmap的性能。比如说，我们有1000个元素new HashMap(1000), 但是理论上来讲new HashMap(1024)更合适，不过上面annegu已经说过，即使是1000，hashmap也自动会将其设置为1024。 但是new HashMap(1024)还不是更合适的，因为0.75\*1000 < 1000, 也就是说为了让0.75 \* size > 1000, 我们必须这样new HashMap(2048)才最合适，既考虑了&的问题，也避免了resize的问题。

**总结：HashMap的实现原理：**

1. **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。(1)如果key相同，则覆盖原始值；(2)如果key不同（出现冲突），则将当前的key-value放入链表中**
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。**
4. **理解了以上过程就不难明白HashMap是如何解决hash冲突的问题，核心就是使用了数组的存储方式，然后将冲突的key的对象放入链表中，一旦发现冲突就在链表中做进一步的对比。**

# [深入探讨HashMap的底层结构、原理、扩容机制](http://youzhixueyuan.com/the-underlying-structure-and-principle-of-hashmap.html)

## 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。

**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("优知","IT进阶站");

系统将调用”优知”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

ashMap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

}

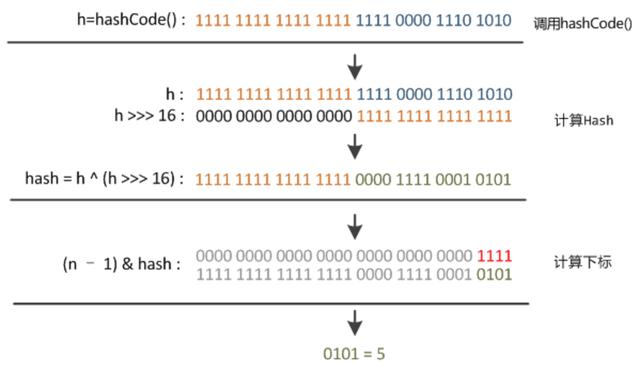
这里的Hash算法本质上就是三步：**取key的hashCode值、高位运算、取模运算**。

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。

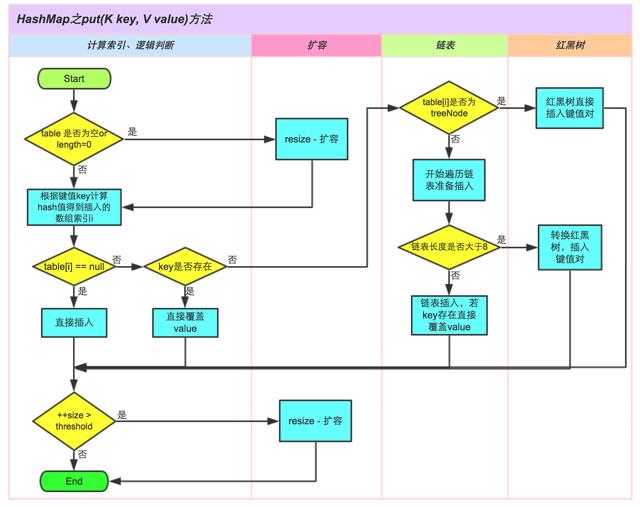
在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

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。

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)：

ublic class HashMapInfiniteLoop {

private static HashMap<Integer,String> map = new HashMap<Integer,String>(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);

};

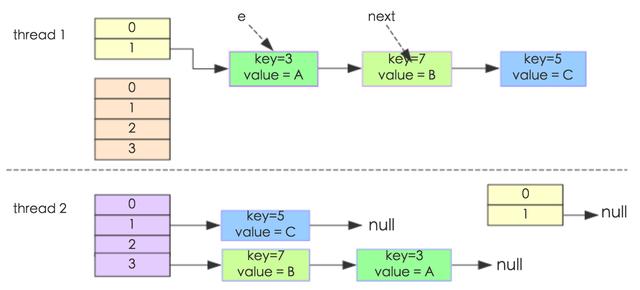
}.start();

}

}

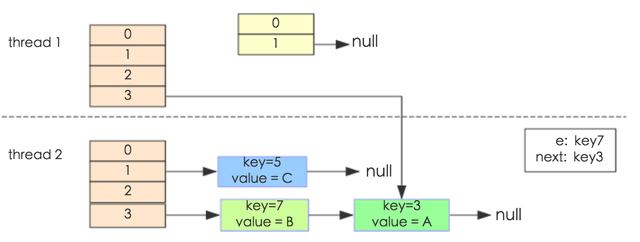
其中，map初始化为一个长度为2的数组，loadFactor=0.75，threshold=2\*0.75=1，也就是说当put第二个key的时候，map就需要进行resize。

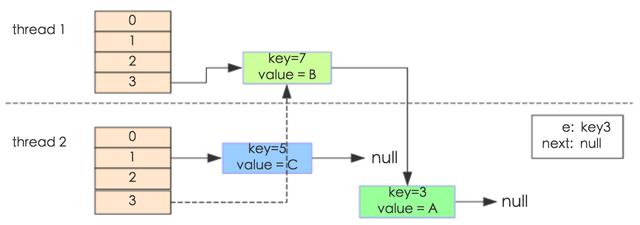
通过设置断点让线程1和线程2同时debug到transfer方法(3.3小节代码块)的首行。注意此时两个线程已经成功添加数据。放开thread1的断点至transfer方法的“Entry next = e.next;” 这一行；然后放开线程2的的断点，让线程2进行resize。结果如下图。



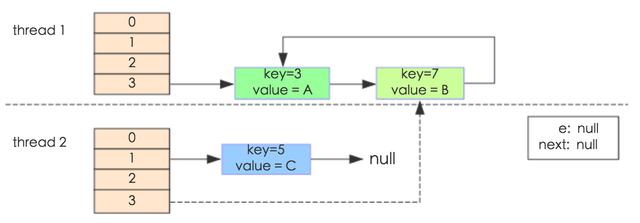
注意，Thread1的 e 指向了key(3)，而next指向了key(7)，其在线程二rehash后，指向了线程二重组后的链表。

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) {

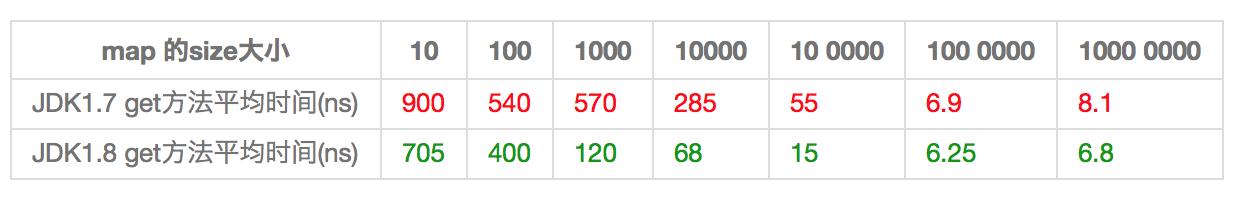
for(int i=10;i<= 1000 0000;i\*= 10){

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：



## 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

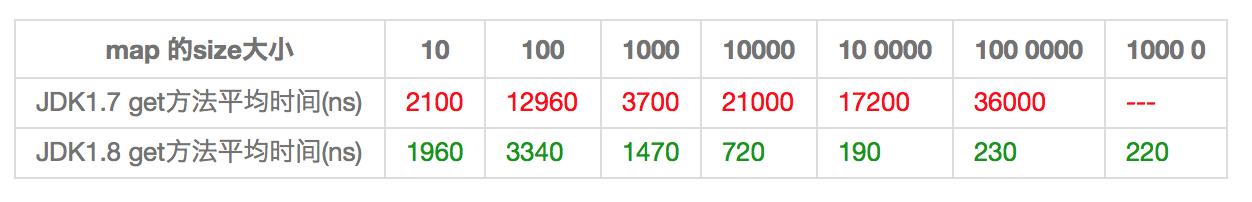
public int hashCode() {

return 1;

}

}

仍然执行main方法，得出的结果如下表所示：



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。

<http://youzhixueyuan.com/the-underlying-structure-and-principle-of-hashmap.html>

<https://www.cnblogs.com/yuanblog/p/4441017.html>

（1）线程不安全的HashMap

在多线程环境下，使用HashMap进行put操作会引起死循环，导致CPU利用率接近100%，所

以在并发情况下不能使用HashMap。例如，执行以下代码会引起死循环。

final HashMap<String, String> map = new HashMap<String, String>(2);

Thread t = new Thread(new Runnable() {

@Override

public void run() {

for (int i = 0; i < 10000; i++) {

new Thread(new Runnable() {

@Override

public void run() {

map.put(UUID.randomUUID().toString(), "");

}

}, "ftf" + i).start();

}

}

}, "ftf");

t.start();

t.join();

HashMap在并发执行put操作时会引起死循环，是因为多线程会导致HashMap的Entry链表

形成环形数据结构，一旦形成环形数据结构，Entry的next节点永远不为空，就会产生死循环获

取Entry。

（2）效率低下的HashTable

HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable

的效率非常低下。因为当一个线程访问HashTable的同步方法，其他线程也访问HashTable的同

步方法时，会进入阻塞或轮询状态。如线程1使用put进行元素添加，线程2不但不能使用put方

法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

（3）ConcurrentHashMap的锁分段技术可有效提升并发访问率

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的

线程都必须竞争同一把锁，假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么

当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效提高并

发访问效率，这就是ConcurrentHashMap所使用的锁分段技术。首先将数据分成一段一段地存

储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数

据也能被其他线程访问。

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重

入锁（ReentrantLock），在ConcurrentHashMap里扮演锁的角色；HashEntry则用于存储键值对数

据。一个ConcurrentHashMap里包含一个Segment数组。Segment的结构和HashMap类似，是一种

数组和链表结构。一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元

素，每个Segment守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，

必须首先获得与它对应的Segment锁