Jdk7 中hashmap计算hash值的方法如下

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

JDK 的 HashMap 使用了一个 hash 方法对hash值使用位的操作，使hash值的计算效率很高。为什么这样做？主要是因为如果直接使用hashcode值，那么这是一个int值（8个16进制数，共32位），int值的范围正负21亿多，但是hash表没有那么长，一般比如初始16，自然散列地址需要对hash表长度取模运算，得到的余数才是地址下标。假设某个key的hashcode是0AAA0000，hash数组长默认16，如果不经过hash函数处理，该键值对会被存放在hash数组中下标为0处，因为0AAA0000 & (16-1) = 0。过了一会儿又存储另外一个键值对，其key的hashcode是0BBB0000，得到数组下标依然是0，这就说明这是个实现得很差的hash算法，因为hashcode的1位全集中在前16位了，导致算出来的数组下标一直是0。于是明明key相差很大的键值对，却存放在了同一个链表里，导致以后查询起来比较慢（蜕化为了顺序查找）。故JDK的设计者使用hash函数的若干次的移位、异或操作，把hashcode的“1位”变得“松散”，非常巧妙。

一下基于jdk1.7，有点过时，看看就好，对照源码看更好

# ****下面是几个常见的面试题****

## **说下hashmap的 扩容机制？**

前面说了，hashmap的构造器里指明了两个对于理解HashMap比较重要的两个参数 int initialCapacity, float loadFactor,这两个参数会影响HashMap效率，HashMap底层采用的散列数组实现，利用initialCapacity这个参数我们可以设置这个数组的大小，也就是散列桶的数量，但是如果需要Map的数据过多，在不断的add之后，这些桶可能都会被占满，这是有两种策略，一种是不改变Capacity，因为即使桶占满了，我们还是可以利用每个桶附带的链表增加元素。但是这有个缺点，此时HaspMap就退化成为了LinkedList，使get和put方法的时间开销上升，这是就要采用另一种方法：增加Hash桶的数量，这样get和put的时间开销又回退到近于常数复杂度上。Hashmap就是采用的该方法。

### **关于扩容。看hashmap的扩容方法，resize方法，它的源码如下**

IMG_256 扩容的resize方法

很明显，是从新建了一个HashMap的底层数组，长度为原来的两倍，而后调用transfer方法，将旧HashMap的全部元素添加到新的HashMap中（要重新计算元素在新的数组中的索引位置）。transfer方法的源码如下：

IMG_257 transfer方法源码

很明显，扩容是一个相当耗时的操作，因为它需要重新计算这些元素在新的数组中的位置并进行复制处理。因此，我们在用HashMap时，最好能提前预估下HashMap中元素的个数，这样有助于提高HashMap的性能。

## **hashmap什么时候需要增加容量呢？**

因为效率问题，JDK采用预处理法，这时前面说的loadFactor就派上了用场，当size > initialCapacity \* loadFactor，hashmap内部resize方法就被调用，使得重新扩充hash桶的数量，在目前的实现中，是增加一倍，这样就保证当你真正想put新的元素时效率不会明显下降。所以一般情况下HashMap并不存在键值放满的情况。当然并不排除极端情况，比如设置的JVM内存用完了，或者这个HashMap的Capacity已经达到了MAXIMUM\_CAPACITY（目前的实现是2^30）。

## **initialCapacity和loadFactor参数设什么样的值好呢？**

initialCapacity的默认值是16，有些人可能会想如果内存足够，是不是可以将initialCapacity设大一些，即使用不了这么大，就可避免扩容导致的效率的下降，反正无论initialCapacity大小，我们使用的get和put方法都是常数复杂度的。这么说没什么不对，但是可能会忽略一点，实际的程序可能不仅仅使用get和put方法，也有可能使用迭代器，如initialCapacity容量较大，那么会使迭代器效率降低。所以理想的情况还是在使用HashMap前估计一下数据量。

加载因子默认值是0.75，是JDK权衡时间和空间效率之后得到的一个相对优良的数值。如果这个值过大，虽然空间利用率是高了，但是对于HashMap中的一些方法的效率就下降了，包括get和put方法，会导致每个hash桶所附加的链表增长，影响存取效率。如果比较小，除了导致空间利用率较低外没有什么坏处，只要有的是内存，毕竟现在大多数人把时间看的比空间重要。但是实际中还是很少有人会将这个值设置的低于0.5。

## **HashMap的key和value都能为null么？如果k能为null，那么它是怎么样查找值的？**

如果key为null，则直接从哈希表的第一个位置table[0]对应的链表上查找。记住，key为null的键值对永远都放在以table[0]为头结点的链表中。

## **HashMap中put值的时候如果发生了冲突，是怎么处理的？**

JDK7使用了链地址法，hash表的每个元素又分别链接着一个单链表，元素为头结点，如果不同的key映射到了相同的下标，那么就使用头插法，插入到该元素对应的链表。JDK8中不是应用头插法，而是尾插法

## **HashMap的key是如何散列到hash表的？相比较HashTable有什么改进？**

我们一般对哈希表的散列很自然地会想到用hash值对length取模（即除留余数法），HashTable就是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，且hashtable直接使用了hashcode值，没有重新计算。

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

HashMap中则通过 h&(length-1) 的方法来代替取模，其中h是key的hash值，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。

首先，length为2的整数次幂的话，h&(length-1) 在数学上就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；

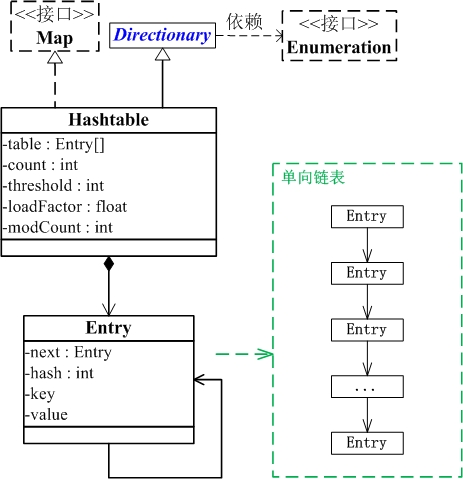
其次，length为2的整数次幂的话，则一定为偶数，那么 length-1 一定为奇数，奇数的二进制的最后一位是1，这样便保证了 h&(length-1) 的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀，而如果length为奇数的话，很明显 length-1 为偶数，它的最后一位是0，这样 h&(length-1) 的最后一位肯定为0，即只能为偶数，这样导致了任何hash值都只会被散列到数组的偶数下标位置上，浪费了一半的空间，因此length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

# **作为对比，在讨论一下Hashtable**

HashTable同样是基于哈希表实现的，其实类似HashMap，只不过有些区别，HashTable同样每个元素是一个key-value对，其内部也是通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

HashTable比较古老， 是JDK1.0就引入的类，而HashMap 是 1.2 引进的 Map 的一个实现。

HashTable 是线程安全的，能用于多线程环境中。Hashtable同样也实现了Serializable接口，支持序列化，也实现了Cloneable接口，能被克隆。



Hashtable继承于Dictionary类，实现了Map接口。Dictionary是声明了操作"键值对"函数接口的抽象类。 有一点注意，HashTable除了线程安全之外（其实是直接在方法上增加了synchronized关键字，比较古老，落后，低效的同步方式），还有就是它的key、value都不为null。另外Hashtable 也有 **初始容量** 和 **加载因子**。

public Hashtable() {

this(11, 0.75f);

}

**默认加载因子也是 0.75，HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。因为HashTable是直接使用除留余数法定位地址。且Hashtable计算hash值，直接用key的hashCode()。**

还要注意：前面说了Hashtable中key和value都不允许为null，而HashMap中key和value都允许为null（key只能有一个为null，而value则可以有多个为null）。但如在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。

最后针对扩容：**Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。**

# ****下面是几个常见的笔试，面试题****

## **HashTable和HashMap的区别有哪些？**

HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。

理解HashMap是Hashtable的轻量级实现（非线程安全的实现，hashtable是非轻量级，线程安全的），都实现Map接口，主要区别在于：

1、由于HashMap非线程安全，在只有一个线程访问的情况下，效率要高于HashTable

2、HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

3、HashMap把Hashtable的contains方法去掉了，改成containsValue和containsKey。因为contains方法容易让人引起误解。

4、Hashtable继承自陈旧的Dictionary类，而HashMap是Java1.2引进的Map 的一个实现。

5、Hashtable和HashMap扩容的方法不一样，HashTable中hash数组默认大小11，扩容方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数，增加为原来的2倍，没有加1。

6、两者通过hash值散列到hash表的算法不一样，HashTbale是古老的除留余数法，直接使用hashcode，而后者是强制容量为2的幂，重新根据hashcode计算hash值，在使用hash  位与  （hash表长度 – 1），也等价取膜，但更加高效，取得的位置更加分散，偶数，奇数保证了都会分散到。前者就不能保证。

7、另一个区别是HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。

* fail-fast和iterator迭代器相关。如果某个集合对象创建了Iterator或者ListIterator，然后其它的线程试图“结构上”更改集合对象，将会抛出ConcurrentModificationException异常。但其它线程可以通过set()方法更改集合对象是允许的，因为这并没有从“结构上”更改集合。但是假如已经从结构上进行了更改，再调用set()方法，将会抛出IllegalArgumentException异常。
* 结构上的更改指的是删除或者插入一个元素，这样会影响到map的结构。
* 该条说白了就是在使用迭代器的过程中有其他线程在结构上修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

## **为什么HashMap是线程不安全的，实际会如何体现？**

第一，如果多个线程同时使用put方法添加元素

假设正好存在两个put的key发生了碰撞(hash值一样)，那么根据HashMap的实现，这两个key会添加到数组的同一个位置，这样最终就会发生其中一个线程的put的数据被覆盖。

第二，如果多个线程同时检测到元素个数超过数组大小\*loadFactor

这样会发生多个线程同时对hash数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给table，也就是说其他线程的都会丢失，并且各自线程put的数据也丢失。**且会引起死循环的错误。**

具体细节上的原因，可以参考：[不正当使用HashMap导致cpu 100%的问题追究](http://ifeve.com/hashmap-infinite-loop/" \t "http://www.cnblogs.com/kubixuesheng/p/_blank)

## **能否让HashMap实现线程安全，如何做？**

1、直接使用Hashtable，但是当一个线程访问HashTable的同步方法时，其他线程如果也要访问同步方法，会被阻塞住。举个例子，当一个线程使用put方法时，另一个线程不但不可以使用put方法，连get方法都不可以，效率很低，现在基本不会选择它了。

2、HashMap可以通过下面的语句进行同步：

Collections.synchronizeMap(hashMap);

3、直接使用JDK 5 之后的 ConcurrentHashMap，如果使用Java 5或以上的话，请使用ConcurrentHashMap。

## **Collections.synchronizeMap(hashMap);又是如何保证了HashMap线程安全？**

直接分析源码吧

IMG_259 View Code

// synchronizedMap方法

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {

return new SynchronizedMap<>(m);

}

// SynchronizedMap类

private static class SynchronizedMap<K,V>

implements Map<K,V>, Serializable {

private static final long serialVersionUID = 1978198479659022715L;

private final Map<K,V> m; // Backing Map

final Object mutex; // Object on which to synchronize

SynchronizedMap(Map<K,V> m) {

this.m = Objects.requireNonNull(m);

mutex = this;

}

SynchronizedMap(Map<K,V> m, Object mutex) {

this.m = m;

this.mutex = mutex;

}

public int size() {

synchronized (mutex) {return m.size();}

}

public boolean isEmpty() {

synchronized (mutex) {return m.isEmpty();}

}

public boolean containsKey(Object key) {

synchronized (mutex) {return m.containsKey(key);}

}

public boolean containsValue(Object value) {

synchronized (mutex) {return m.containsValue(value);}

}

public V get(Object key) {

synchronized (mutex) {return m.get(key);}

}

public V put(K key, V value) {

synchronized (mutex) {return m.put(key, value);}

}

public V remove(Object key) {

synchronized (mutex) {return m.remove(key);}

}

// 省略其他方法

}

从源码中看出 synchronizedMap()方法返回一个SynchronizedMap类的对象，而在SynchronizedMap类中使用了synchronized来保证对Map的操作是线程安全的，故效率其实也不高。

## **为什么HashTable的默认大小和HashMap不一样？**

前面分析了，Hashtable 的扩容方法是乘2再+1，不是简单的乘2，故hashtable保证了容量永远是奇数，结合之前分析hashmap的重算hash值的逻辑，就明白了，因为在数据分布在等差数据集合(如偶数)上时，如果公差与桶容量有公约数 n，则至少有(n-1)/n 数量的桶是利用不到的，故之前的hashmap 会在取模（使用位与运算代替）哈希前先做一次哈希运算，调整hash值。这里hashtable比较古老，直接使用了除留余数法，那么就需要设置容量起码不是偶数（除（近似）质数求余的分散效果好）。而JDK开发者选了11。

Hashtable默认大小是11是因为除（近似）质数求余的分散效果好：

Hashtable的扩容是这样做的：

int oldCapacity = table.length;

int newCapacity = oldCapacity \* 2 + 1;

虽然不保证capacity是一个质数，但至少保证它是一个奇数。

Hashtable的寻址是这样做的：

Entry tab[] = table;

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

直接用key的hashCode()，不像HashMap里为了增强hash的分散效果而要做二次hash

## **JDK 8对HashMap有了什么改进？说说你对红黑树的理解？**

参考更新的jdk 8对hashmap的的改进部分整理，并且还能引申出高级数据结构——红黑树，这又能引出很多问题……学无止境啊！

1. **HashMap** 的默认大小为16，有什么特别的理由吗？

已解决。

HashMap 的容量大小需要保证为 2^n。

HashMap 定位桶时是对桶容量( HashMap 容量)取余，（无论因果），该取余函数是用 h & (length - 1) 实现。@RednaxelaFX

2. **HashTable** 的默认大小为11，有什么特别的理由吗？

已解决。

HashTable 的容量增加逻辑是乘2+1，保证奇数。

在应用数据随机分布时，容量越大越好。Hash时取模一定要模质数吗？ - 编程

在应用数据分布在等差数据集合(如偶数)上时，如果公差与桶容量有公约数n，则至少有(n-1)/n数量的桶是利用不到的。

实际上 HashMap 也会有此问题，并且不能指定桶容量。所以 HashMap 会在取模哈希前先做一次哈希，@RednaxelaFX。

h ^= (h >>> 20) ^ (h >>> 12);

h ^ (h >>> 7) ^ (h >>> 4);

3. **ArrayList** 默认大小为10，有什么特别的理由吗？

ArrayList 的容量增长逻辑是乘 1.5 + 1，逻辑比较随意，看不出有什么特别含义。

4. **ArrayDeque** 默认大小为8，注释明确说明容量需要取 2^n。但是 ArrayDeque 内部存储结构是数组(没有哈希操作)，并没有与问题1类似的需求。

已解决

ArrayDeque 的容量大小需要保证为 2^n。

ArrayDeque 使用 (tail - head) & (elements.length - 1) 计算 size()

5. **PriorityQueue** 默认大小为11，内部存储结构是数组(最小堆)，却没有使用 2^n 而是11。

PriorityQueue 的容量增长逻辑是 乘 2 或 1.5，逻辑比较随意，看不出有什么特别含义。

HashMap也是我们使用非常多的Collection，它是基于哈希表的 Map 接口的实现，以key-value的形式存在。在HashMap中，key-value总是会当做一个整体来处理，系统会根据hash算法来来计算key-value的存储位置，我们总是可以通过key快速地存、取value。下面就来分析HashMap的存取。

## 一、定义

      HashMap实现了Map接口，继承AbstractMap。其中Map接口定义了键映射到值的规则，而AbstractMap类提供 Map 接口的骨干实现，以最大限度地减少实现此接口所需的工作，其实AbstractMap类已经实现了Map，这里标注Map LZ觉得应该是更加清晰吧！

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

## 二、构造函数

      HashMap提供了三个构造函数：

      HashMap()：构造一个具有默认初始容量 (16) 和默认加载因子 (0.75) 的空 HashMap。

      HashMap(int initialCapacity)：构造一个带指定初始容量和默认加载因子 (0.75) 的空 HashMap。

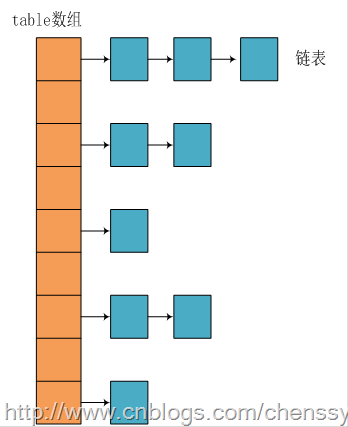
      HashMap(int initialCapacity, float loadFactor)：构造一个带指定初始容量和加载因子的空 HashMap。

      在这里提到了两个参数：初始容量，加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。系统默认负载因子为0.75，一般情况下我们是无需修改的。

      HashMap是一种支持快速存取的数据结构，要了解它的性能必须要了解它的数据结构。

## 三、数据结构

      我们知道在Java中最常用的两种结构是数组和模拟指针(引用)，几乎所有的数据结构都可以利用这两种来组合实现，HashMap也是如此。实际上HashMap是一个“链表散列”，如下是它数据结构：



      从上图我们可以看出HashMap底层实现还是数组，只是数组的每一项都是一条链。其中参数initialCapacity就代表了该数组的长度。下面为HashMap构造函数的源码：

public HashMap(int initialCapacity, float loadFactor) {

//初始容量不能<0  
        if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal initial capacity: "+ initialCapacity);

//初始容量不能 > 最大容量值，HashMap的最大容量值为2^30  
        if (initialCapacity > MAXIMUM\_CAPACITY)  
            initialCapacity = MAXIMUM\_CAPACITY;

//负载因子不能 < 0  
        if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal load factor: "+ loadFactor);        // 计算出大于 initialCapacity 的最小的 2 的 n 次方值。  
        int capacity = 1;

while (capacity < initialCapacity)  
            capacity <<= 1;          
        this.loadFactor = loadFactor;

//设置HashMap的容量极限，当HashMap的容量达到该极限时就会进行扩容操作  
        threshold = (int) (capacity \* loadFactor);

//初始化table数组  
        table = new Entry[capacity];  
        init();  
    }

      从源码中可以看出，每次新建一个HashMap时，都会初始化一个table数组。table数组的元素为Entry节点。

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

final K key;  
        V value;  
        Entry<K,V> next;

final int hash;

/\*\*  
         \* Creates new entry.

\*/  
        Entry(int h, K k, V v, Entry<K,V> n) {  
            value = v;  
            next = n;  
            key = k;  
            hash = h;  
        }  
        .......  
    }

      其中Entry为HashMap的内部类，它包含了键key、值value、下一个节点next，以及hash值，这是非常重要的，正是由于Entry才构成了table数组的项为链表。

      上面简单分析了HashMap的数据结构，下面将探讨HashMap是如何实现快速存取的。

## 四、存储实现：put(key,vlaue)

      首先我们先看源码public V put(K key, V value) {

//当key为null，调用putForNullKey方法，保存null与table第一个位置中，这是HashMap允许为null的原因

        if (key == null)

return putForNullKey(value);

//计算key的hash值  
        int hash = hash(key.hashCode());                  ------(1)        //计算key hash 值在 table 数组中的位置  
        int i = indexFor(hash, table.length);             ------(2)        //从i出开始迭代 e,找到 key 保存的位置  
        for (Entry<K, V> e = table[i]; e != null; e = e.next) {  
             Object k;

//判断该条链上是否有hash值相同的(key相同)

//若存在相同，则直接覆盖value，返回旧value  
            if (e.hash == hash && ((k = e.key) == key || key.equals(k)))  {  
                V oldValue = e.value;    //旧值 = 新值  
                e.value = value;  
                e.recordAccess(this);

return oldValue;     //返回旧值

}  
        }

//修改次数增加1  
        modCount++;

//将key、value添加至i位置处        addEntry(hash, key, value, i);

return null;  
    }

      通过源码我们可以清晰看到HashMap保存数据的过程为：首先判断key是否为null，若为null，则直接调用putForNullKey方法。若不为空则先计算key的hash值，然后根据hash值搜索在table数组中的索引位置，如果table数组在该位置处有元素，则通过比较是否存在相同的key，若存在则覆盖原来key的value，否则将该元素保存在链头（最先保存的元素放在链尾）。若table在该处没有元素，则直接保存。这个过程看似比较简单，其实深有内幕。有如下几点：

      1、 先看迭代处。此处迭代原因就是为了防止存在相同的key值，若发现两个hash值（key）相同时，HashMap的处理方式是用新value替换旧value，这里并没有处理key，这就解释了HashMap中没有两个相同的key。

      2、 在看（1）、（2）处。这里是HashMap的精华所在。首先是hash方法，该方法为一个纯粹的数学计算，就是计算h的hash值。

static int hash(int h) {  
        h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);  
    }

      我们知道对于HashMap的table而言，数据分布需要均匀（最好每项都只有一个元素，这样就可以直接找到），不能太紧也不能太松，太紧会导致查询速度慢，太松则浪费空间。计算hash值后，怎么才能保证table元素分布均与呢？我们会想到取模，但是由于取模的消耗较大，HashMap是这样处理的：调用indexFor方法。

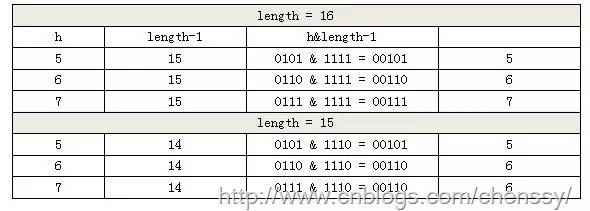
static int indexFor(int h, int length) {

return h & (length-1);  
    }

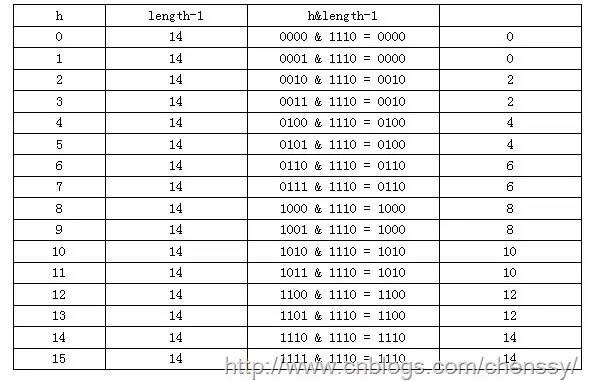
      HashMap的底层数组长度总是2的n次方，在构造函数中存在：capacity <<= 1;这样做总是能够保证HashMap的底层数组长度为2的n次方。当length为2的n次方时，h&(length - 1)就相当于对length取模，而且速度比直接取模快得多，这是HashMap在速度上的一个优化。至于为什么是2的n次方下面解释。

      我们回到indexFor方法，该方法仅有一条语句：h&(length - 1)，这句话除了上面的取模运算外还有一个非常重要的责任：均匀分布table数据和充分利用空间。

      这里我们假设length为16(2^n)和15，h为5、6、7。



      当n=15时，6和7的结果一样，这样表示他们在table存储的位置是相同的，也就是产生了碰撞，6、7就会在一个位置形成链表，这样就会导致查询速度降低。诚然这里只分析三个数字不是很多，那么我们就看0-15。



      从上面的图表中我们看到总共发生了8此碰撞，同时发现浪费的空间非常大，有1、3、5、7、9、11、13、15处没有记录，也就是没有存放数据。这是因为他们在与14进行&运算时，得到的结果最后一位永远都是0，即0001、0011、0101、0111、1001、1011、1101、1111位置处是不可能存储数据的，空间减少，进一步增加碰撞几率，这样就会导致查询速度慢。而当length = 16时，length – 1 = 15 即1111，那么进行低位&运算时，值总是与原来hash值相同，而进行高位运算时，其值等于其低位值。所以说当length = 2^n时，不同的hash值发生碰撞的概率比较小，这样就会使得数据在table数组中分布较均匀，查询速度也较快。

      这里我们再来复习put的流程：当我们想一个HashMap中添加一对key-value时，系统首先会计算key的hash值，然后根据hash值确认在table中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依此比较其key的hash值。如果两个hash值相等且key值相等(e.hash == hash && ((k = e.key) == key || key.equals(k))),则用新的Entry的value覆盖原来节点的value。如果两个hash值相等但key值不等 ，则将该节点插入该链表的链头。具体的实现过程见addEntry方法，如下：void addEntry(int hash, K key, V value, int bucketIndex) {

//获取bucketIndex处的Entry

        Entry<K, V> e = table[bucketIndex];

//将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry   
        table[bucketIndex] = new Entry<K, V>(hash, key, value, e);         //若HashMap中元素的个数超过极限了，则容量扩大两倍  
        if (size++ >= threshold)  
            resize(2 \* table.length);  
    }

      这个方法中有两点需要注意：

      一是链的产生。这是一个非常优雅的设计。系统总是将新的Entry对象添加到bucketIndex处。如果bucketIndex处已经有了对象，那么新添加的Entry对象将指向原有的Entry对象，形成一条Entry链，但是若bucketIndex处没有Entry对象，也就是e==null,那么新添加的Entry对象指向null，也就不会产生Entry链了。

      二、扩容问题。

      随着HashMap中元素的数量越来越多，发生碰撞的概率就越来越大，所产生的链表长度就会越来越长，这样势必会影响HashMap的速度，为了保证HashMap的效率，系统必须要在某个临界点进行扩容处理。该临界点在当HashMap中元素的数量等于table数组长度\*加载因子。但是扩容是一个非常耗时的过程，因为它需要重新计算这些数据在新table数组中的位置并进行复制处理。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。

## 五、读取实现：get(key)

      相对于HashMap的存而言，取就显得比较简单了。通过key的hash值找到在table数组中的索引处的Entry，然后返回该key对应的value即可。

public V get(Object key) {

// 若为null，调用getForNullKey方法返回相对应的value  
        if (key == null)

return getForNullKey();

// 根据该 key 的 hashCode 值计算它的 hash 码    
        int hash = hash(key.hashCode());

// 取出 table 数组中指定索引处的值  
        for (Entry<K, V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {  
            Object k;

//若搜索的key与查找的key相同，则返回相对应的value  
            if (e.hash == hash && ((k = e.key) == key || key.equals(k)))                return e.value;  
        }

return null;  
    }

      在这里能够根据key快速的取到value除了和HashMap的数据结构密不可分外，还和Entry有莫大的关系，在前面就提到过，HashMap在存储过程中并没有将key，value分开来存储，而是当做一个整体key-value来处理的，这个整体就是Entry对象。同时value也只相当于key的附属而已。在存储的过程中，系统根据key的hashcode来决定Entry在table数组中的存储位置，在取的过程中同样根据key的hashcode取出相对应的Entry对象。

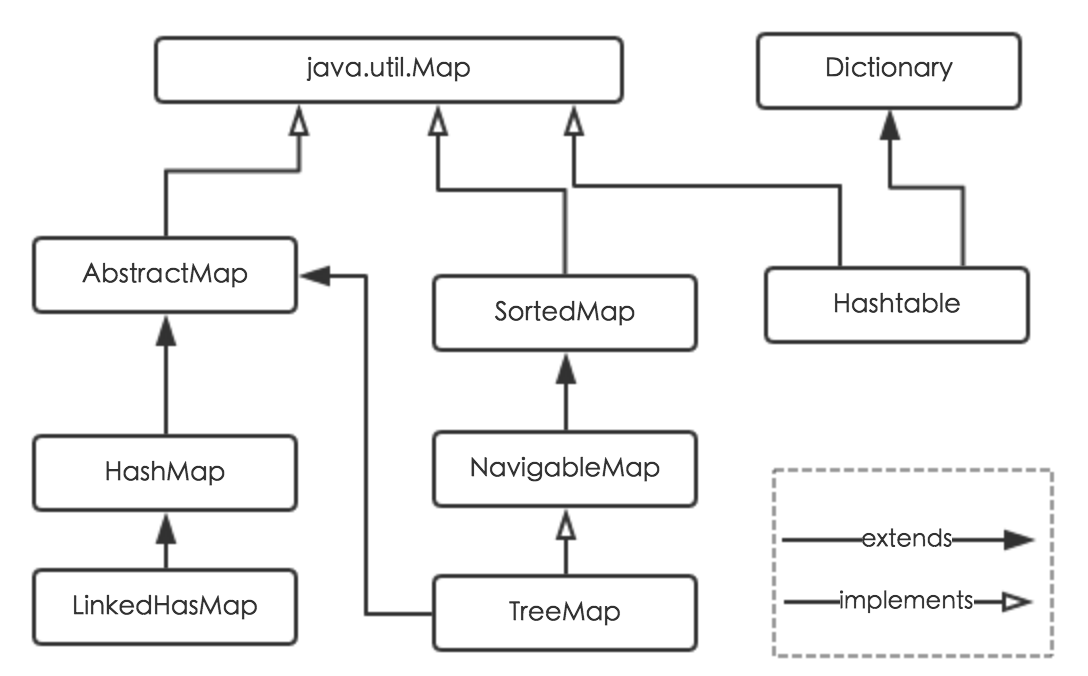
摘要

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：

java.util.map类图



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。

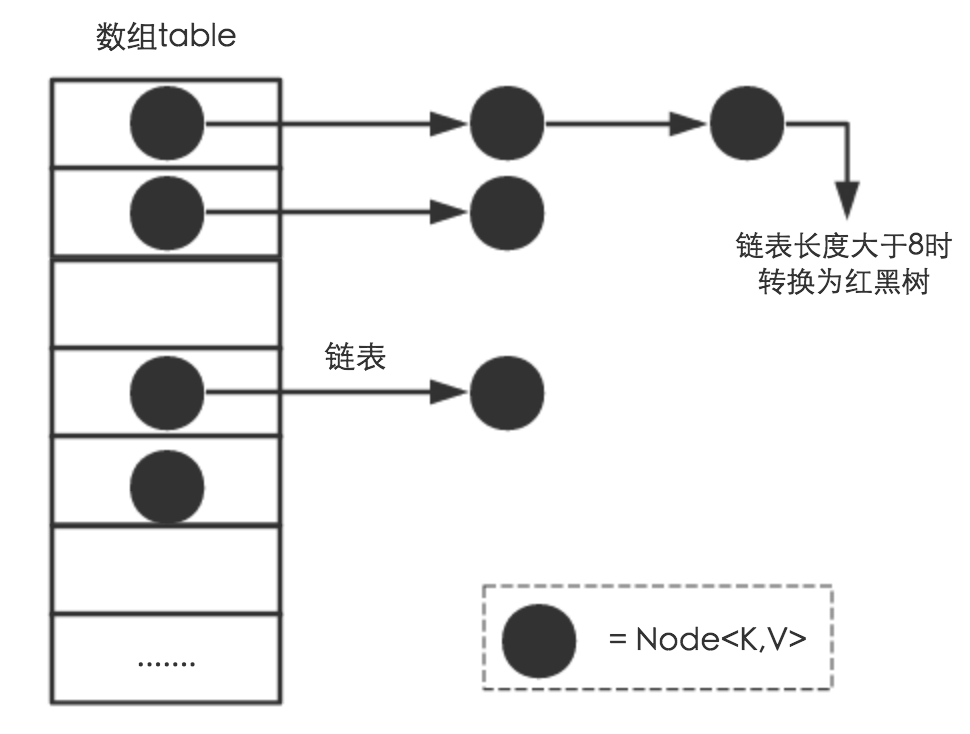
通过上面的比较，我们知道了HashMap是Java的Map家族中一个普通成员，鉴于它可以满足大多数场景的使用条件，所以是使用频度最高的一个。下文我们主要结合源码，从存储结构、常用方法分析、扩容以及安全性等方面深入讲解HashMap的工作原理。

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。



hashMap内存结构图

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("美团","小美");

系统将调用"美团"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。相对来说素数导致冲突的概率要小于合数，具体证明可以参考http://blog.csdn.net/liuqiyao\_01/article/details/14475159，Hashtable初始化桶大小为11，就是桶大小设计为素数的应用（Hashtable扩容后不能保证还是素数）。HashMap采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

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可以参考http://blog.csdn.net/v\_july\_v/article/details/6105630。

功能实现-方法

HashMap的内部功能实现很多，本文主要从根据key获取哈希桶数组索引位置、put方法的详细执行、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

}

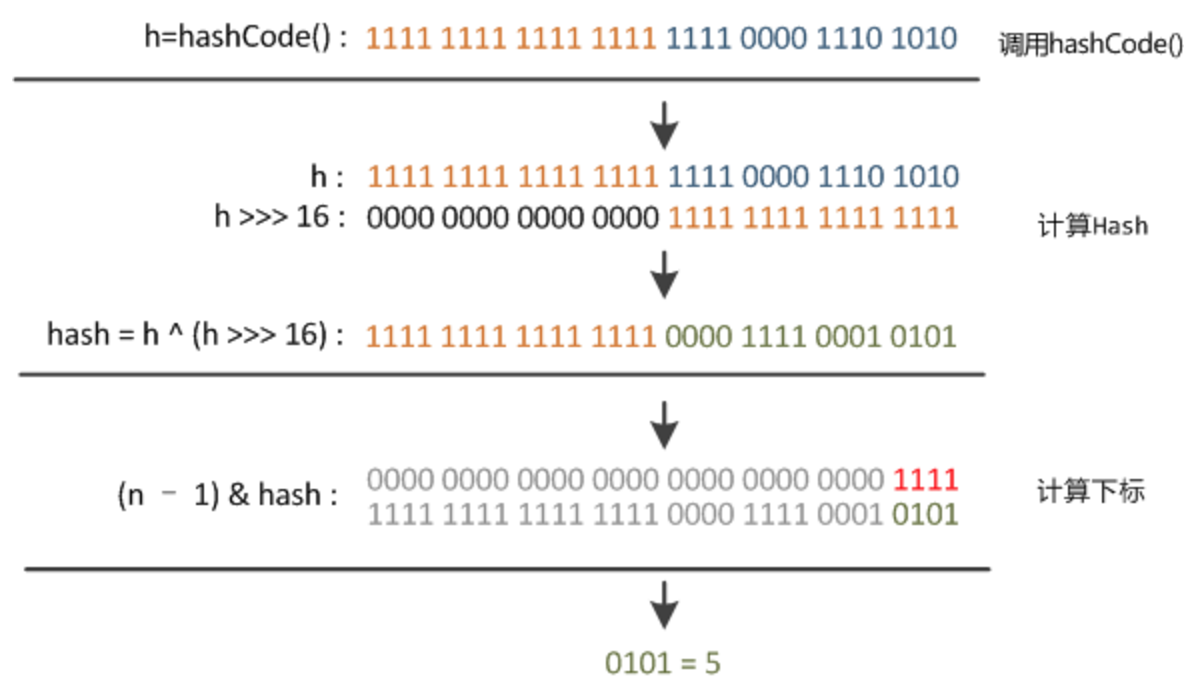
这里的Hash算法本质上就是三步：取key的hashCode值、高位运算、取模运算。

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。

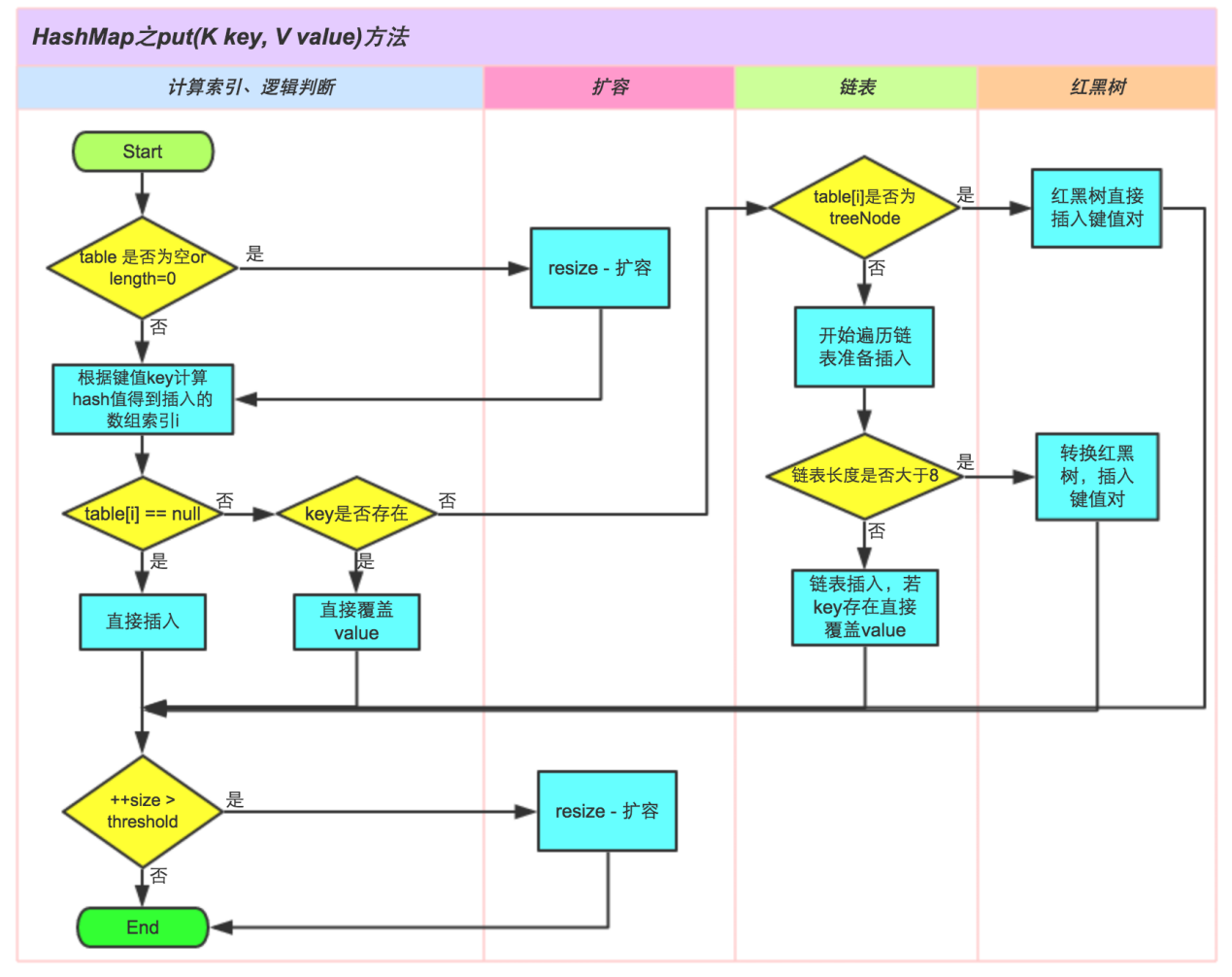
下面举例说明下，n为table的长度。



hashMap哈希算法例图

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。



hashMap put方法执行流程图

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。

JDK1.8HashMap的put方法源码如下:

1 public V put(K key, V value) {

2 // 对key的hashCode()做hash

3 return putVal(hash(key), key, value, false, true);

4 }

5

6 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

7 boolean evict) {

8 Node<K,V>[] tab; Node<K,V> p; int n, i;

9 // 步骤①：tab为空则创建

10 if ((tab = table) == null || (n = tab.length) == 0)

11 n = (tab = resize()).length;

12 // 步骤②：计算index，并对null做处理

13 if ((p = tab[i = (n - 1) & hash]) == null)

14 tab[i] = newNode(hash, key, value, null);

15 else {

16 Node<K,V> e; K k;

17 // 步骤③：节点key存在，直接覆盖value

18 if (p.hash == hash &&

19 ((k = p.key) == key || (key != null && key.equals(k))))

20 e = p;

21 // 步骤④：判断该链为红黑树

22 else if (p instanceof TreeNode)

23 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

24 // 步骤⑤：该链为链表

25 else {

26 for (int binCount = 0; ; ++binCount) {

27 if ((e = p.next) == null) {

28 p.next = newNode(hash, key,value,null);

//链表长度大于8转换为红黑树进行处理

29 if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

30 treeifyBin(tab, hash);

31 break;

32 }

// key已经存在直接覆盖value

33 if (e.hash == hash &&

34 ((k = e.key) == key || (key != null && key.equals(k))))

35 break;

36 p = e;

37 }

38 }

39

40 if (e != null) { // existing mapping for key

41 V oldValue = e.value;

42 if (!onlyIfAbsent || oldValue == null)

43 e.value = value;

44 afterNodeAccess(e);

45 return oldValue;

46 }

47 }

48 ++modCount;

49 // 步骤⑥：超过最大容量 就扩容

50 if (++size > threshold)

51 resize();

52 afterNodeInsertion(evict);

53 return null;

54 }

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，具体区别后文再说。

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);

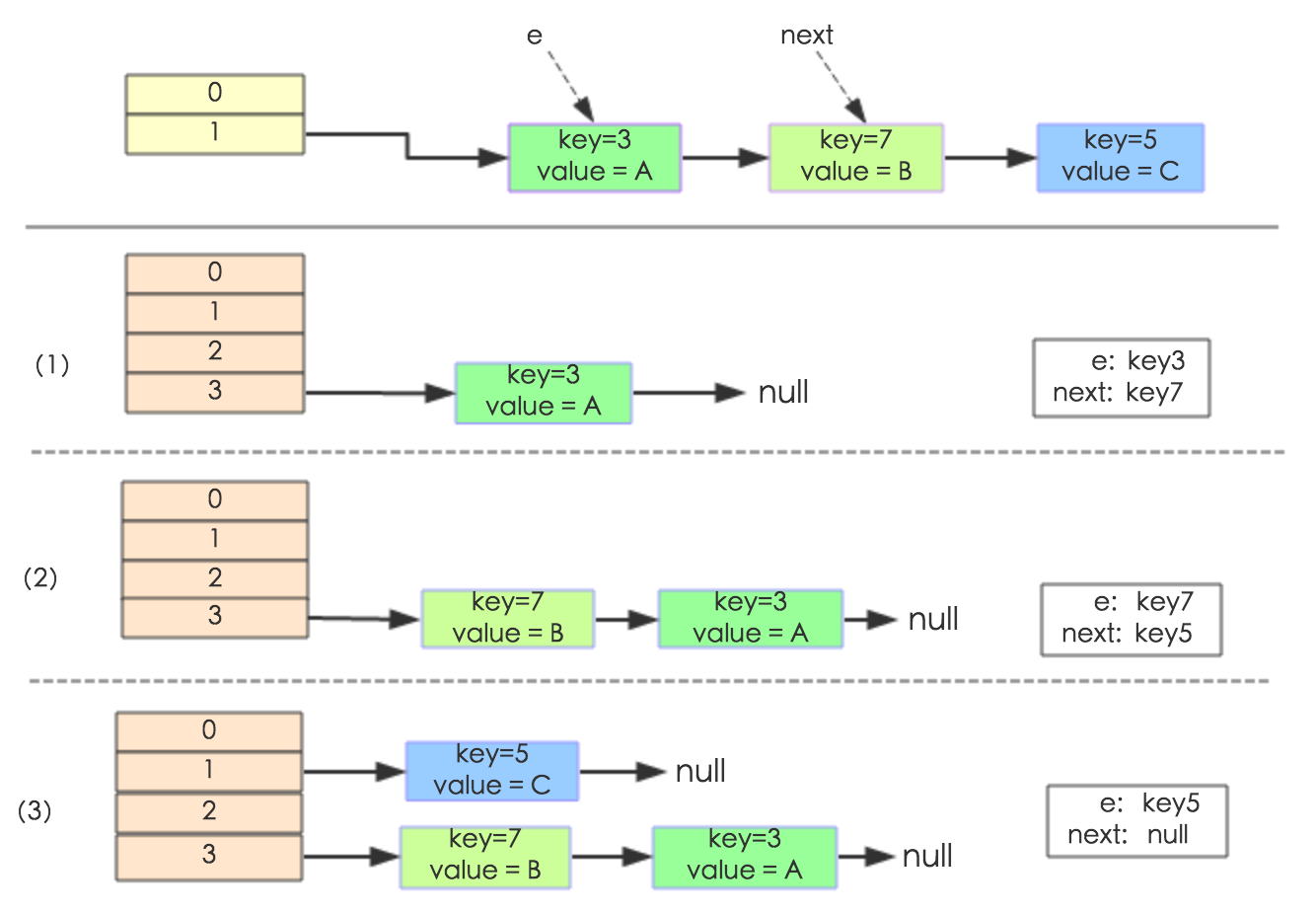
15 }

16 }

17 }

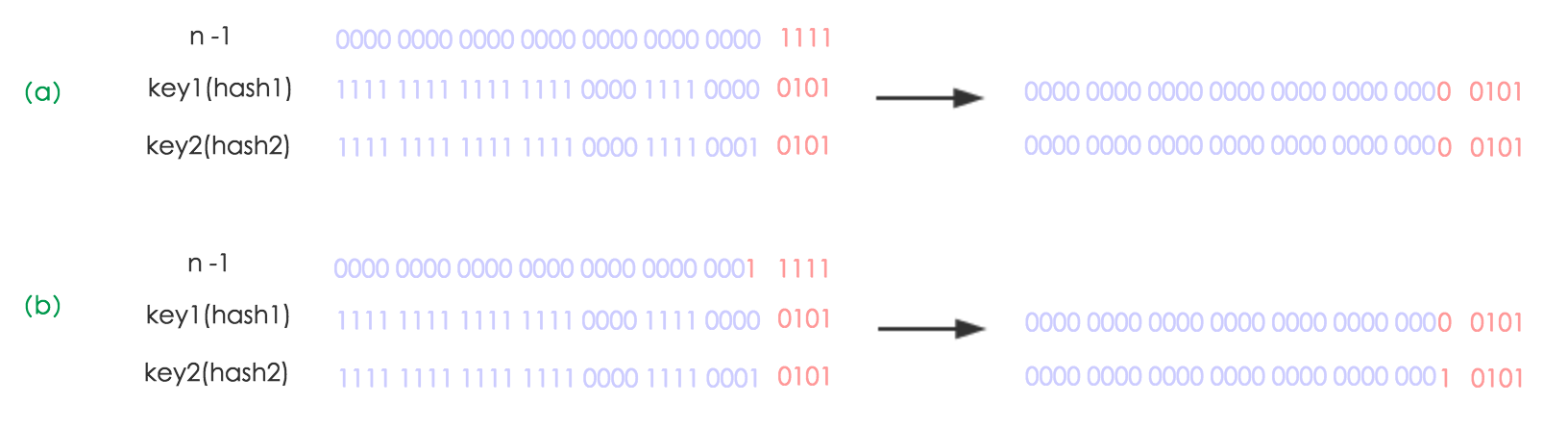
newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话），这一点和Jdk1.8有区别，下文详解。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

下面举个例子说明下扩容过程。假设了我们的hash算法就是简单的用key mod 一下表的大小（也就是数组的长度）。其中的哈希桶数组table的size=2， 所以key = 3、7、5，put顺序依次为 5、7、3。在mod 2以后都冲突在table[1]这里了。这里假设负载因子 loadFactor=1，即当键值对的实际大小size 大于 table的实际大小时进行扩容。接下来的三个步骤是哈希桶数组 resize成4，然后所有的Node重新rehash的过程。



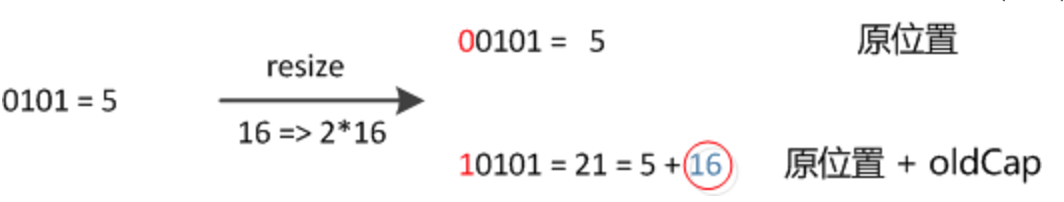
jdk1.7扩容例图

下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



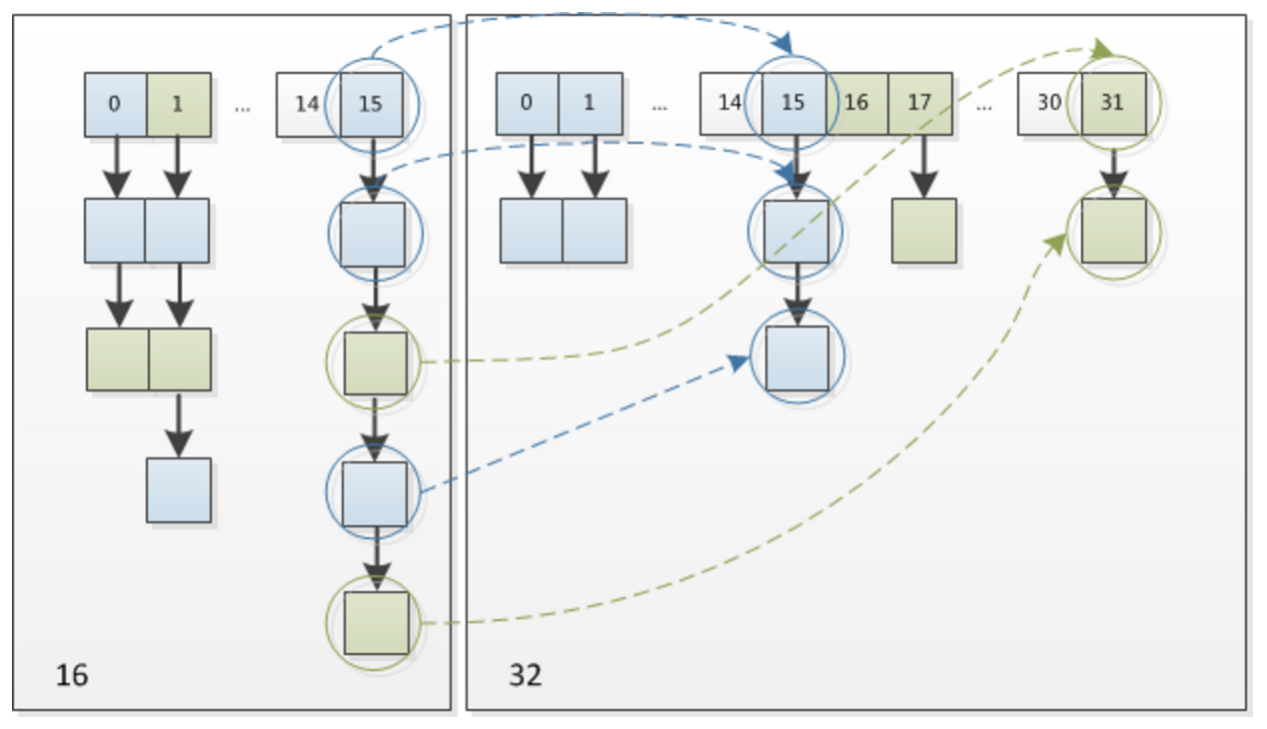
hashMap 1.8 哈希算法例图1

元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



hashMap 1.8 哈希算法例图2

因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



jdk1.8 hashMap扩容例图

这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。有兴趣的同学可以研究下JDK1.8的resize源码，写的很赞，如下:

1 final Node<K,V>[] resize() {

2 Node<K,V>[] oldTab = table;

3 int oldCap = (oldTab == null) ? 0 : oldTab.length;

4 int oldThr = threshold;

5 int newCap, newThr = 0;

6 if (oldCap > 0) {

7 // 超过最大值就不再扩充了，就只好随你碰撞去吧

8 if (oldCap >= MAXIMUM\_CAPACITY) {

9 threshold = Integer.MAX\_VALUE;

10 return oldTab;

11 }

12 // 没超过最大值，就扩充为原来的2倍

13 else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

14 oldCap >= DEFAULT\_INITIAL\_CAPACITY)

15 newThr = oldThr << 1; // double threshold

16 }

17 else if (oldThr > 0) // initial capacity was placed in threshold

18 newCap = oldThr;

19 else { // zero initial threshold signifies using defaults

20 newCap = DEFAULT\_INITIAL\_CAPACITY;

21 newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

22 }

23 // 计算新的resize上限

24 if (newThr == 0) {

25

26 float ft = (float)newCap \* loadFactor;

27 newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?

28 (int)ft : Integer.MAX\_VALUE);

29 }

30 threshold = newThr;

31 @SuppressWarnings({"rawtypes"，"unchecked"})

32 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];

33 table = newTab;

34 if (oldTab != null) {

35 // 把每个bucket都移动到新的buckets中

36 for (int j = 0; j < oldCap; ++j) {

37 Node<K,V> e;

38 if ((e = oldTab[j]) != null) {

39 oldTab[j] = null;

40 if (e.next == null)

41 newTab[e.hash & (newCap - 1)] = e;

42 else if (e instanceof TreeNode)

43 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

44 else { // 链表优化重hash的代码块

45 Node<K,V> loHead = null, loTail = null;

46 Node<K,V> hiHead = null, hiTail = null;

47 Node<K,V> next;

48 do {

49 next = e.next;

50 // 原索引

51 if ((e.hash & oldCap) == 0) {

52 if (loTail == null)

53 loHead = e;

54 else

55 loTail.next = e;

56 loTail = e;

57 }

58 // 原索引+oldCap

59 else {

60 if (hiTail == null)

61 hiHead = e;

62 else

63 hiTail.next = e;

64 hiTail = e;

65 }

66 } while ((e = next) != null);

67 // 原索引放到bucket里

68 if (loTail != null) {

69 loTail.next = null;

70 newTab[j] = loHead;

71 }

72 // 原索引+oldCap放到bucket里

73 if (hiTail != null) {

74 hiTail.next = null;

75 newTab[j + oldCap] = hiHead;

76 }

77 }

78 }

79 }

80 }

81 return newTab;

82 }

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,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);

};

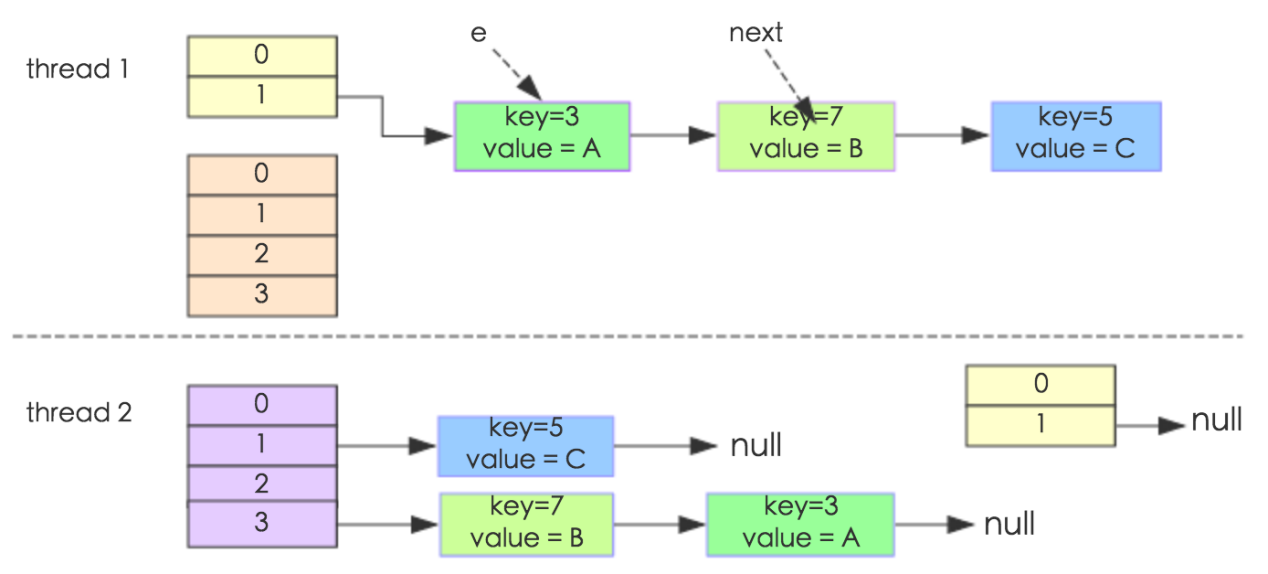
}.start();

}

}

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。

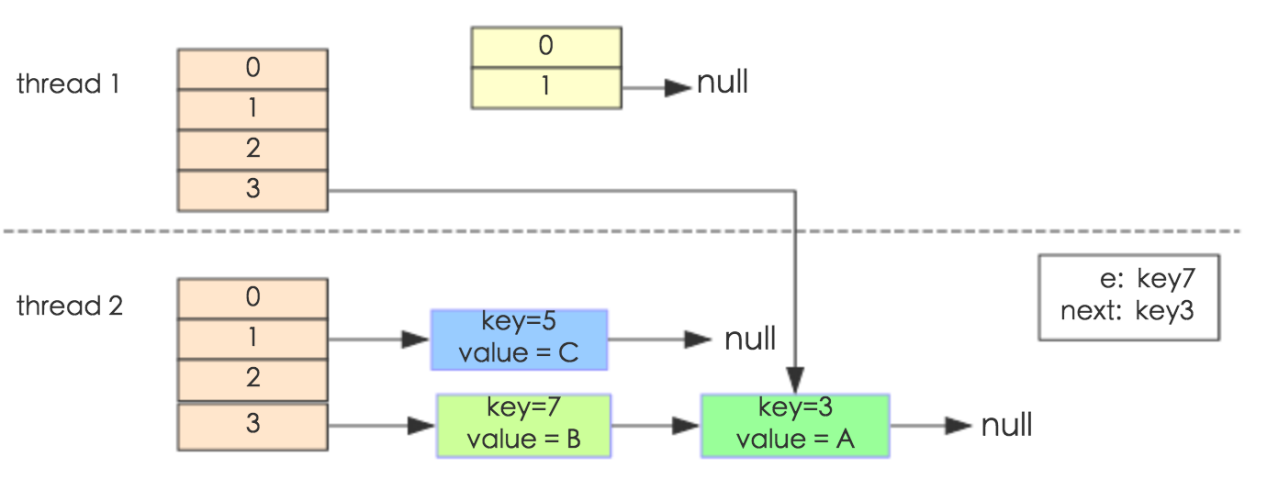
通过设置断点让线程1和线程2同时debug到transfer方法(3.3小节代码块)的首行。注意此时两个线程已经成功添加数据。放开thread1的断点至transfer方法的“Entry next = e.next;” 这一行；然后放开线程2的的断点，让线程2进行resize。结果如下图。

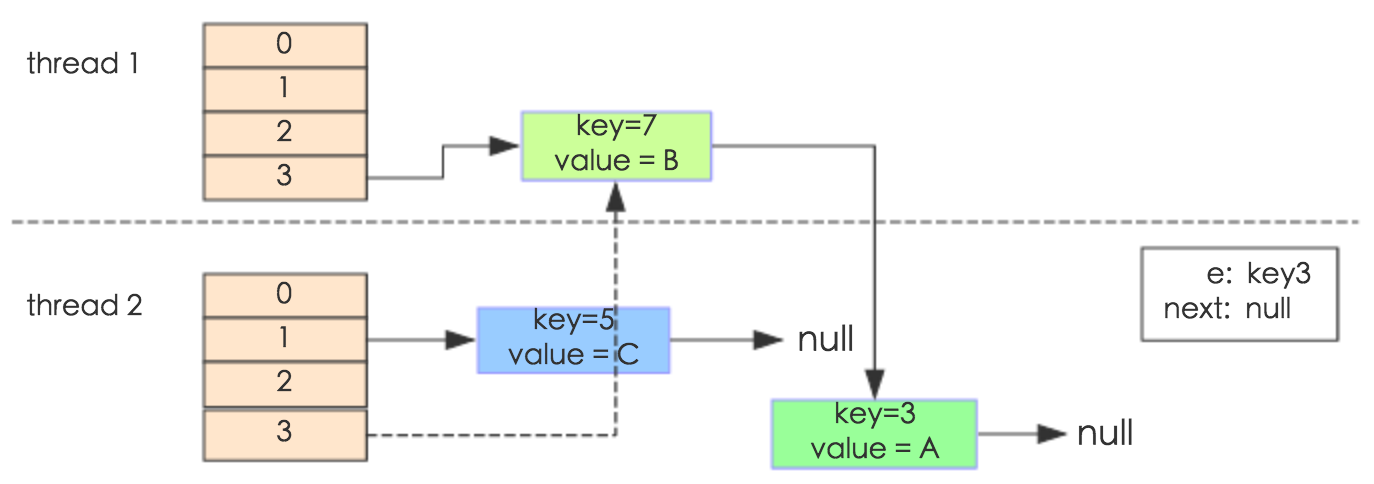


jdk1.7 hashMap死循环例图1

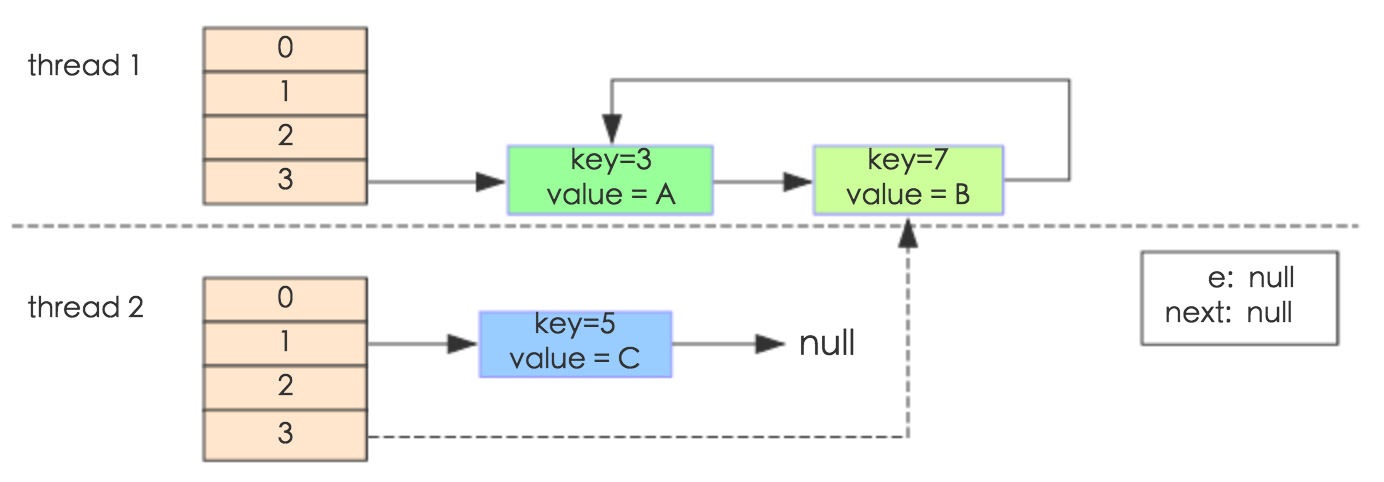
注意，Thread1的 e 指向了key(3)，而next指向了key(7)，其在线程二rehash后，指向了线程二重组后的链表。

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。



jdk1.7 hashMap死循环例图4

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) {

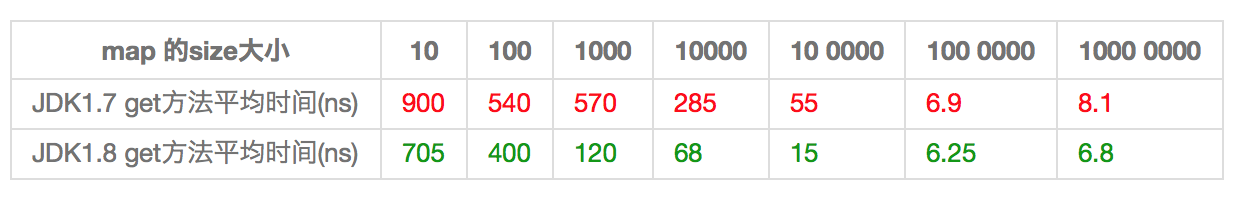
for(int i=10;i<= 1000 0000;i\*= 10){

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

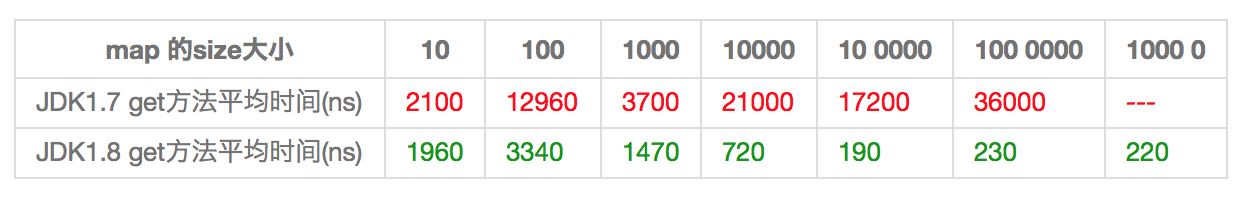
public int hashCode() {

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。

测试环境：处理器为2.2 GHz Intel Core i7，内存为16 GB 1600 MHz DDR3，SSD硬盘，使用默认的JVM参数，运行在64位的OS X 10.10.1上。

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) 还没升级JDK1.8的，现在开始升级吧。HashMap的性能提升仅仅是JDK1.8的冰山一角。