1. 基础概念
2. HashMap介绍

HashMap继承于AbstractMap抽象类，后者实现了Map接口。HashMap是基于Hash Table实现的。从Java2到Java8，HashMap的实现原理不停地发生变化，实现HashMap时一个重要的考量，就是如何尽可能地规避哈希碰撞。

1. 哈希分布和哈希碰撞

2.1 基本概念

1. 完美哈希函数

对于每个对象X和Y，如果当（且仅当，译者注）X.equals(Y)为false，使得X.hashCode() != Y.hashCode()为true，这样的函数叫做完美Hash函数。

1. 哈希碰撞

//如果两个输入串的hash函数的值是一样的，则称这两个串是一个哈希碰撞。

一个Boolean对象有true和false两个值，因此Boolean对象的Hash值可以通过一个二进制位 bit 表达，即0b0, 0b1。对于一些Number对象，比如Integer、Long、Double等，他们都可以使用自身原始的值作为Hash值。然而，想要构造这样的完美哈希函数，我们需要无限的内存大小，这种假设显然是不可能的。而且，即使我们能够为每个POJO（Plain Ordinary Java Object）或者String对象构造一个理论上不会有冲突的哈希函数，但是hashCode()函数的返回值是int型。**根据鸽笼理论，当我们的对象超过2^32个时，这些对象会发生哈希碰撞。**//鸽笼理论

1. 哈希实现

通过允许哈希碰撞来节省内存，可以用来提升总体性能。许多关联数组的实现，包括HashMap，使用了大小为M的桶来储存∣N∣个对象（M≤∣N∣）。在这种情况下，我们使用模值hashValue % M作为桶的索引，而不是hashValue本身。

代码实现：int index = X.hashCode() % M；

* 1. HashMap中解决哈希碰撞的方式

当一个对象的插入HashMap，发生哈希冲突的概率是1M，这与哈希函数的实现无关。根据我们的需要，即使是存在哈希冲突的环境中，数据的读取也应该能够被良好的执行。这里有两种著名的方式来解决这个问题，一种是开放寻址，一种是分离链接。

1. 开放寻址

开放寻址是一种解决哈希冲突的方式，当计算出的桶索引的位置被占据时，通过一定的探索方式，来寻找未被占据的哈希桶（适合数量确定，冲突较少的情况）。

1. 分离链接

分离链接则将每一个哈希桶作为一个链表的头结点，当哈希碰撞发生时，仅需在链表中进行储存、查找。

1. HashMap采用分离链接解决哈希碰撞的原因

这两种方法都有着同样的最坏时间复杂度O(M)，但是开放寻址使用连续的空间，因此有着缓存效率的提升。因此**当数据量较小时，能够放到系统缓存中时，开放寻址会表现出比分离链接更好的性能**。**但是当数据量增长时，它的性能就会越差，**因为我们无法期待一个大的数组能够得到缓存性能优化。这也是HashMap使用分离链表来解决哈希冲突的原因。此外，**开放寻址还有一个弱点。我们调用remove()方法会十分频繁，当我们删除数据时，一个特定的哈希冲突，可能会干扰总体的性能**，而分离链表则没有这样的缺点。

1. Java8中的分离链表
   1. 基本概念
2. 总体介绍

从Java 2到Java 1.7，HashMap在分离链表上的改变并不多，他们的算法基本上是相同的。如果我们假设对象的Hash值服从平均分布，那么获取一个对象需要的次数时间复杂度应该是O(N/M)（原为E(N/M)，但数学期望应改为E(N/2M)疑有误，译者注）。**Java 8 在没有降低哈希冲突的度的情况下，使用红黑树代替链表，将这个值降低到了O(log(N/M))**（与上同，疑有误，译者注）。数据越多，O(NM)和O(log(NM))的差别就会越明显。此外，**在实践中Hash值的分布并非均匀的**，正如“生日问题”所描述那样，哈希值有时也会集中在几个特定值上。因此**使用平衡树比如红黑树有着比使用链表更强的性能。**

1. 链表和树的切换标识

使用链表还是树，与一个哈希桶中的元素数目有关。定义了Java 8的HashMap在使用树和使用链表之间切换的阈值。当冲突的元素数增加到8时，链表变为树；当减少至6时，树切换为链表。中间有2个缓冲值的原因是避免频繁的切换浪费计算机资源。

代码： static final int TREEIFY\_THRESHOLD = 8; //切换为树

static final int UNTREEIFY\_THRESHOLD = 6; //切换为链表

1. HashMap使用的红黑树 //Java8引入的平衡树
2. Java 8 HashMap使用Node类替代了Entry类，它们的结构大体相同。一个显著地差别是，Node类具有导出类TreeNode，通过这种继承关系，一个链表很容易被转换成树。
3. Java 8 HashMap使用的树是红黑树，它的实现基本与JCF中的TreeMap相同。**通常，树的有序性通过两个或更多对象比较大小来保证。Java 8 HashMap中的树也通过对象的Hash值**（这个hash值与哈希桶索引值不同，索引值在这个hash值的基础上对桶大小M取模，译者注）**作为对象的排序键**。因为使用Hash值作为排序键打破了Total Ordering（可以理解为数学中的小于等于关系，译者注），因此这里有一个tieBreakOrder()方法来处理这个问题。//使用hash值来作为排序键而不是通过索引值
4. 哈希桶的动态扩容
5. 小数目的哈希桶可以有效的利用内存，但是会产生更高概率的哈希碰撞，最终损失性能。因此，**HashMap会在数据量达到一定大小时，将哈希桶的数量扩充到两倍。**当哈希桶的数量变为两倍后，NM会对应下降，Hash值重复的Key的数量也得以减少。
6. 哈希桶的默认数量是16，最大值是2^30。当哈希桶的数量成倍增长时，所有的数据需要重新插入。一种HashMap构造器包含初始桶数量这个参数。如果我们能够在使用这个构造器时指定桶的数量，这将使HashMap节约不必要的重新构造分离链表的时间。
7. 哈希桶的扩容

确定是否需要对桶进行扩展的临界值是loadFactor×currentBucketSize，其中loadFactor是负载因子，currentBucketSize是当前桶的数量。当数据量到达这个大小时，扩容就会发生，直到桶的数量达到2^30为止。默认的负载因子是0.75，它与默认桶大小16，一同作为构造默认的HashMap的参数。

1. 辅助哈希函数 //hash函数的实现

使用辅助哈希函数的目的是通过改变初始的哈希值，降低发生哈希冲突的概率。

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

1. String对象的哈希函数

String对象的Hash函数的时间开销与String值的长度成正比。

public int hashCode() { //hashCode的实现方式

int h = hash;

if (h == 0 && value.length > 0) {

char val[] = value;

for (int i = 0; i < value.length; i++) {

h = 31 \* h + val[i];

}

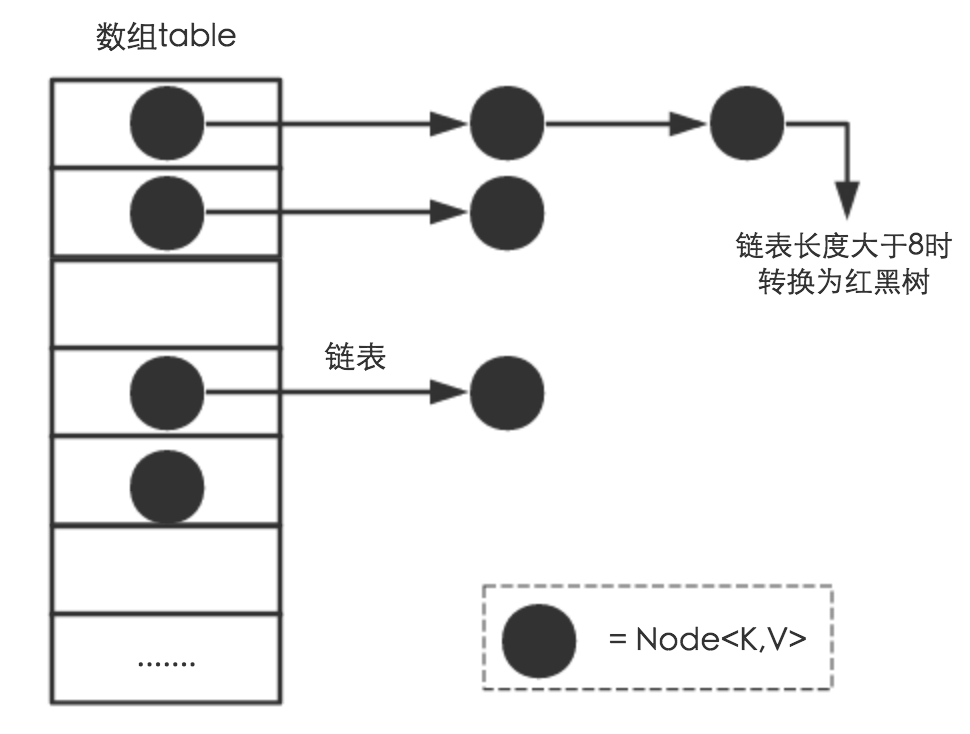
hash = h;

}

return h;}

1. HashMap的内部底层实现

4.1 存储结构 - 字段

(1) 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。 

(2) 哈希桶数组 Node[] table

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。

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() { return key; }

public final V getValue() { return value; }

public final String toString() { return key + "=" + value; }

public final int hashCode() {……}

public final V setValue(V newValue) {……}

public final boolean equals(Object o) {……}

(3) 哈希表实现

实现代码：map.put("美团","小美");

1. 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。
2. 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。
3. 哈希桶数组较大，较差的hash算法也会比较分散，但是所占的空间就比较大。哈希桶数组较小，要求的hash算法也比较高，否则哈希碰撞会变多，花费的时间会变长。
4. 重要属性
5. transient Node<K,V>[] table; //HashMap底层的实际数组实际占内存总大小，哈希桶数量。

transient int size; // HashMap中实际存在的键值对数量

int threshold; // HashMap所能容纳的最大数据量的Node(键值对)个数

final float loadFactor; // 负载因子

**负载因子：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。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。

阈值（threshold）：threshold就是在此**Loadfactor \* length**(数组长度)对应下允许的最大元素数目，size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。

4.2 功能实现 - 方法

4.2.1确定哈希桶数组索引位置

(1) 哈希算法

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

1. 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);

}

1. static int indexFor(int h, int length) { //jdk1.8没有这个方法，但是实现原理一样的

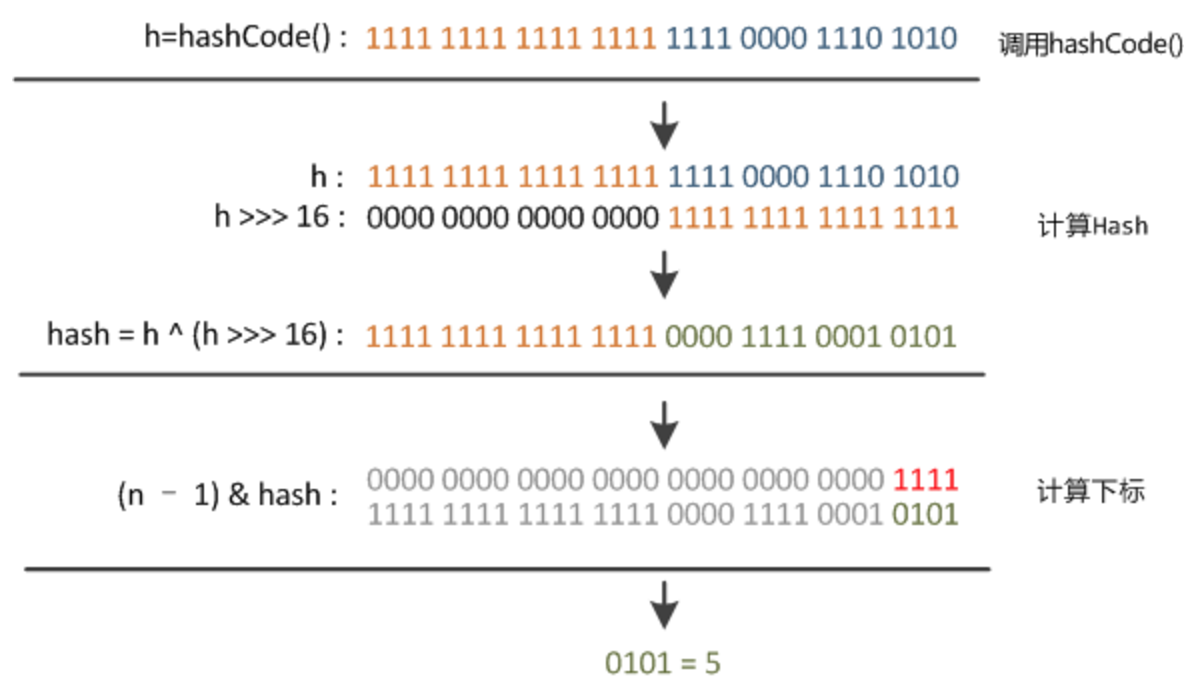
return h & (length-1); //第三步 取模运算

}

(2) 高位参与运算

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。



(3) 取模运算

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。

4.2.2 HashMap的put方法

(1) 源码

// HashMap-put()方法

public V put(K key, V value) {

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

}

// 底层核心方法putVal()方法

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,boolean evict) {

// 哈希桶数组tab，数组元素p

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

// 判断底层数组是否为空或者null，是就执行resize()进行扩容

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

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

//1.查找索引

// 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

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

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

else {

Node<K,V> e; K k;

// tab[i]的hash等于key的hash && tab[i]的key == key

// 或者 key不等于空 && tab[i] 的key值等于key值

// 将tab[i] 赋值给e，用 新value值 覆盖 旧value值

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

e = p;

// 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

else if (p instanceof TreeNode)

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

else {

// 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

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

// e赋值为tab[i]的next，如果为空，新建该key的Node对象

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

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

//链表长度大于等于8转换为红黑树进行处理

if (binCount >= TREEIFY\_THRESHOLD - 1)

treeifyBin(tab, hash);

break;

}

//当e等于该hash &&

//e的key==key || key不等于空 同时 e的key值等key值

if (e.hash == hash &&

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

break;

p = e;

}

}

//当e（p.next）不等于空，新的value值覆盖旧的value值

if (e != null) {

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null)

e.value = value;

afterNodeAccess(e);

return oldValue;

}

}

++modCount;

//当添加元素后size值大于threshold，执行resize()扩容

//如果e=null，则说明新添加了元素，所以要将size++

if (++size > threshold)

resize();

afterNodeInsertion(evict);

return null;

}

(2) 方法逻辑：

① 传入一个key，利用hash算法锁定索引的那个哈希桶。

② 如果哈希桶为null，则新建一个节点当做链表的头结点。

③ 如果哈希桶不为null，判断头结点的hashcode是否等于插入的key，是就更新value值。

④ 如果头结点不等于key，判断该头结点是否为红黑树，如果不是，进入链表循环筛选。

⑤ 判断p.next是否为空，如果为空则新建一个该插入值的节点，并且判断链表长度是否 大于8，如果大于8则转为红黑树处理。

⑥ 如果p.next不为空，则判断这个节点是否等于key，如果不是则继续下一个节点。

⑦ 如果e值为空，则说明新建了一个节点，所以将size的长度增加1；如果e值不为空，则说明用新value值覆盖了旧value值，返回旧value值，不用增加size的长度。

4.3 扩容机制，核心方法resize()

4.3.1 JDK1.7的扩容方法

(1) 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**.

void resize(int newCapacity) { //传入新的容量

Entry[] oldTable = table; //引用扩容前的Entry数组

int oldCapacity = oldTable.length;

//扩容前的数组大小如果已经达到最大(2^30)了

if (oldCapacity == MAXIMUM\_CAPACITY) {

//修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了

threshold = Integer.MAX\_VALUE;

return;

}

Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组

transfer(newTable); //！！将数据转移到新的Entry数组里

table = newTable; //HashMap的table属性引用新的Entry数组

threshold = (int)(newCapacity \* loadFactor); //修改阈值

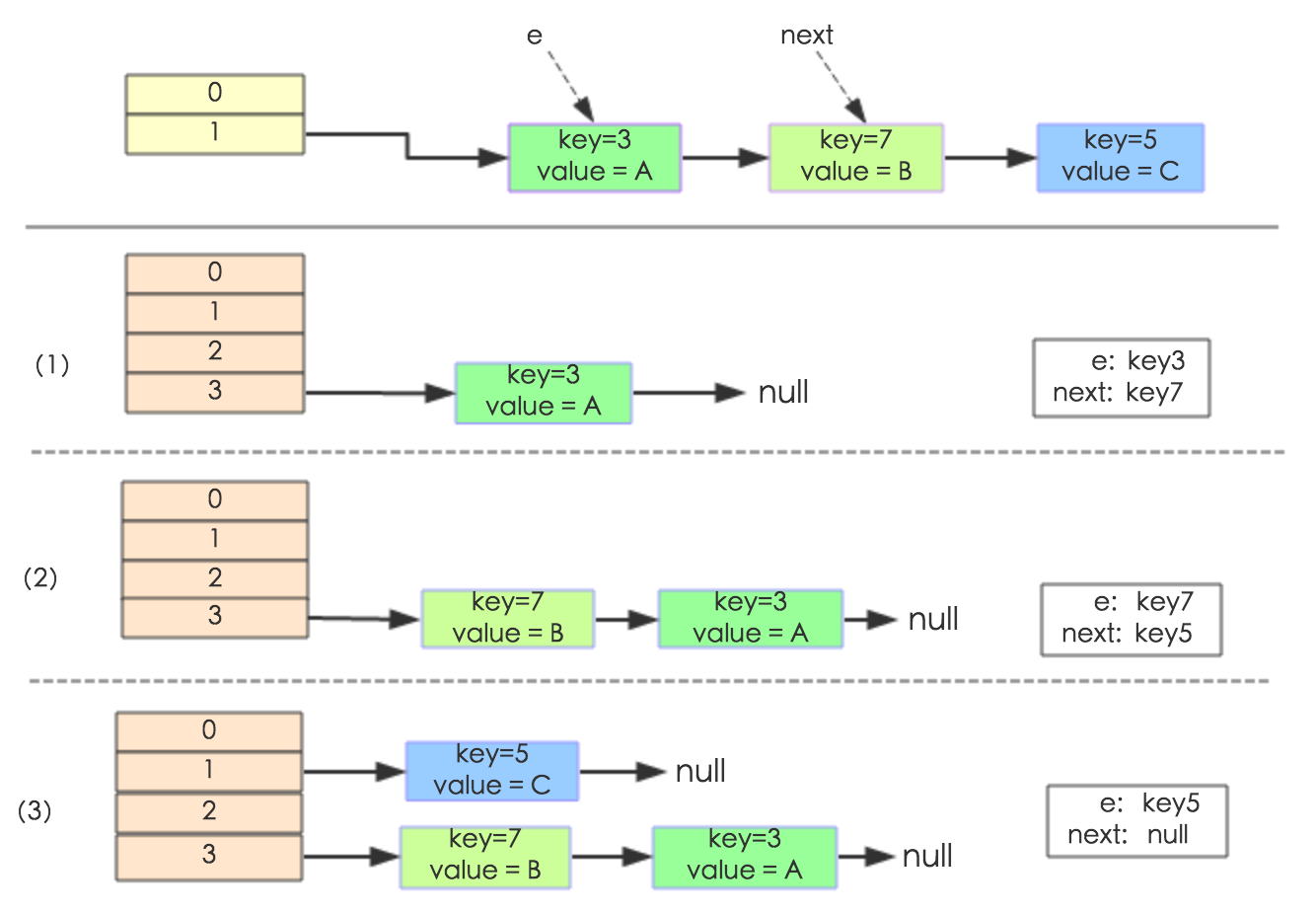
}

(2) 重新计算索引的过程

① 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。

② 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。

③ 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。



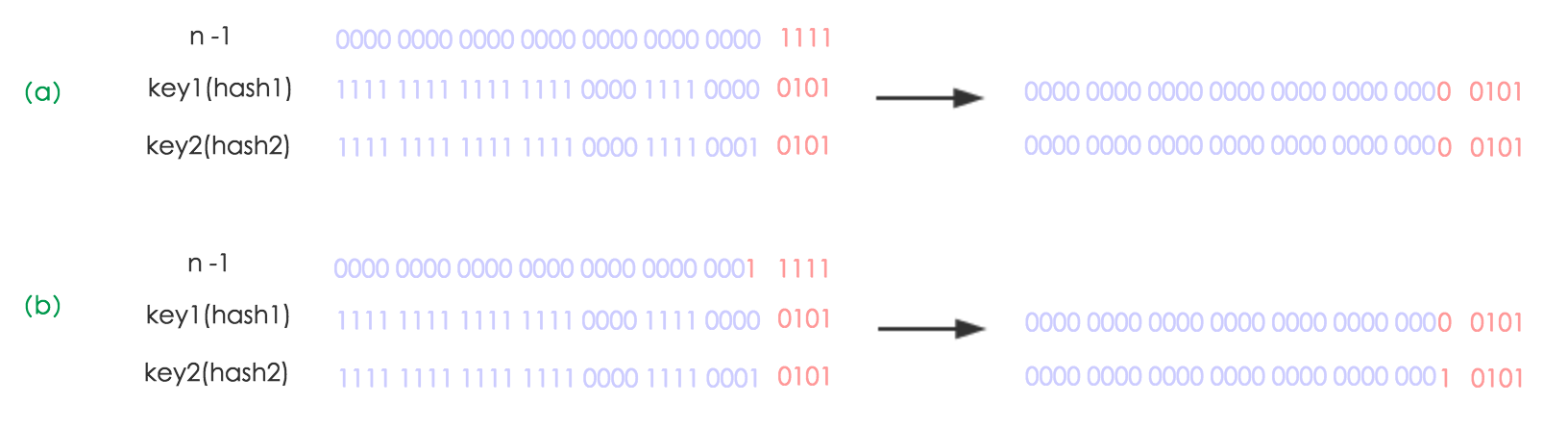
说明：

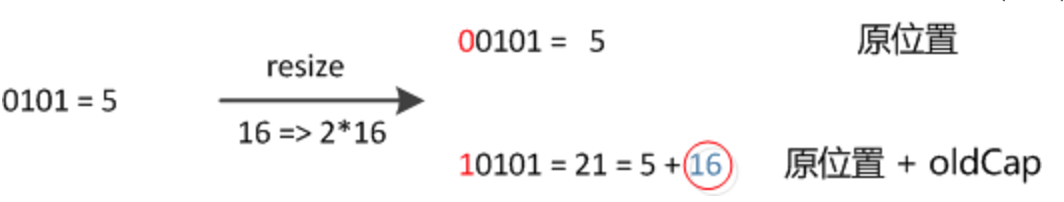
JDK1.7扩容过程就是①确定新数组的length，创建一个新数组。②将旧数组的值依次取出，重新计算索引后放入新数组中，所以在数组顺序是先取出的会被放到最后面。

4.3.2 JDK1.8对扩容机制的优化

(1) 扩容后的元素

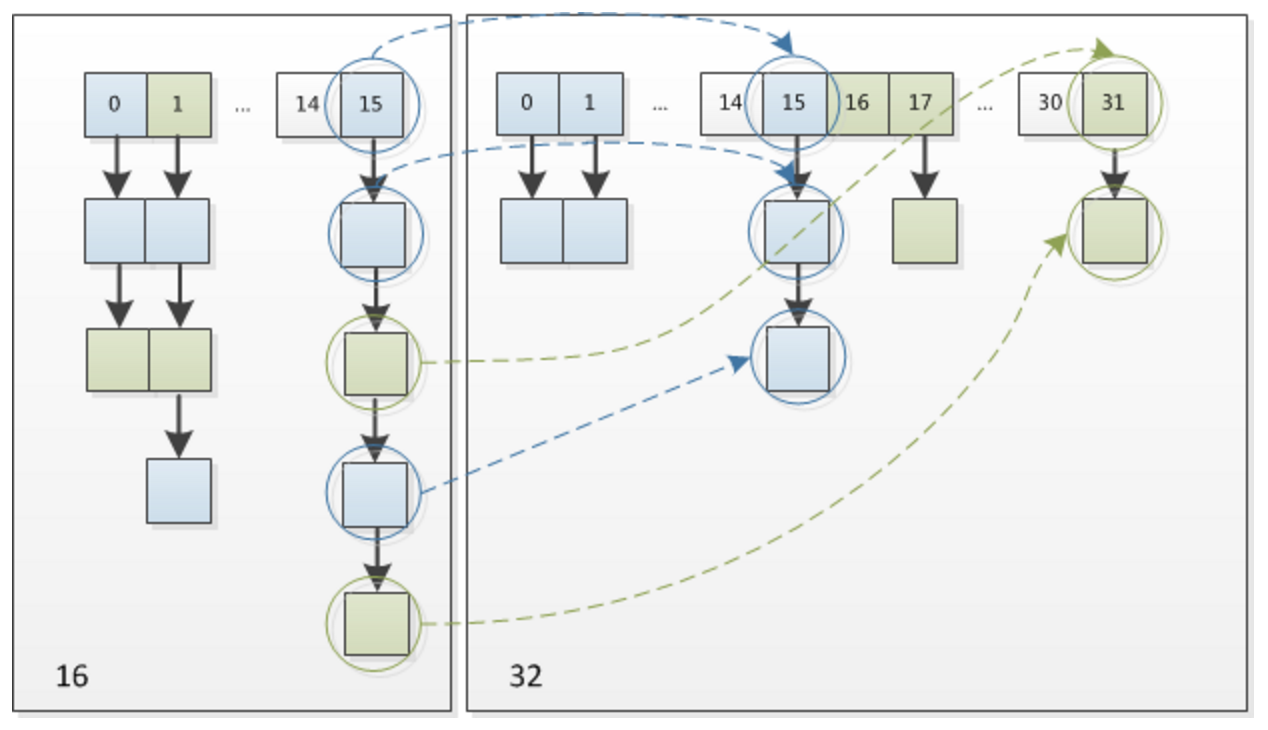
经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，**元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置**。看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。（取模运算）





(2) 扩容后元素不需要再重新计算索引位置，只需要判断是否移动位置。

---16扩充为32的例子



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。

5. 线程安全的HashMap

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。

(1) HashMap是线程不安全的例子。

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();

}

} -- 会造成环形链表

6. JDK1.8和JDK1.7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)。

(1) 当hashcode均匀的情况下（hashCode较少重复）

这时候没有用到红黑树，所以jdk8性能提升不明显。

(2) 当hashcode不均匀的情况下（hashCode较多重复）

当链表长度大于8时，jdk1.8引入红黑树，抛弃了链表，这时候jdk1.8性能提升明显。

7. 底层原理总结

(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。

二、HashMap 具体方法

1. 主要属性

tatic final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; //默认的初始容量2^4

static final int MAXIMUM\_CAPACITY = 1 << 30; //最大的容量2^30

static final float DEFAULT\_LOAD\_FACTOR = 0.75f; //默认的加载因子

transient Node<K,V>[] table; //实际内存存储的数组表示哈希桶数量，Entry类型数组

transient int size; //实际存储键值对的大小

int threshold; //中能够存储键值对的数量，由实际容量length\*loadFactor相乘得到

final float loadFactor; //加载因子，默认为0.75

2. 构造方法

1. public HashMap()

说明：构造一个默认加载因子 (0.75) 的空 HashMap 。

public HashMap() {

this.loadFactor = DEFAULT\_LOAD\_FACTOR; // 默认加载因子

} //当执行put()方法时，会自动扩容为最小容量16

1. public HashMap(int initialCapacity)

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

public HashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_LOAD\_FACTOR);// 调用另一个构造方法

}

1. public HashMap(int initialCapacity, float loadFactor)

说明：构造一个带指定初始容量和加载因子的空 HashMap 。

public HashMap(int initialCapacity, float loadFactor) {

if (initialCapacity < 0) //当指定容量小于0抛出异常

throw new IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

if (initialCapacity > MAXIMUM\_CAPACITY) //当指定容量大于最大容量

initialCapacity = MAXIMUM\_CAPACITY;//使容量等于最大容量

if (loadFactor <= 0 || Float.isNaN(loadFactor)) //当加载因子小于0或者不是数字

throw new IllegalArgumentException("Illegal load factor: " +

loadFactor);

this.loadFactor = loadFactor;

//当构造方法传入初始容量时，调用该方法将容量扩充为离初始容量最近的那个

// 2的幂次方，注：该方法不是扩容方法，而是规范容量大小的方法

this.threshold = tableSizeFor(initialCapacity);

}

1. public HashMap(Map< ? extends K, ? extends V> m)

构造一个映射关系与指定 Map 相同的新 HashMap。

3. 常用方法

3.1 put添加键值对方法

★4.3.2put方法解析，主要是引入了红黑树

3.2 get()方法

(1) 根据key值获取value值

public V get(Object key) {

Node<K,V> e;

//获取e对象是否为空，不为空则返回e的value值

return (e = getNode(hash(key), key)) == null ? null : e.value;

}

//根据hash值和key值获取对应的node对象

final Node<K,V> getNode(int hash, Object key) {

Node<K,V>[] tab; Node<K,V> first, e; int n; K k;

//当内存数组table不等于null且长度大于0，且hash值取模得到的头结点不为空

if ((tab = table) != null && (n = tab.length) > 0 &&(first = tab[(n - 1) & hash]) != null) {

//头结点的hash值等于传入的hash值

//且first.key等于key值 || key值不等于空且equals first.key

//返回索引数组的头结点

if (first.hash == hash && ((k = first.key) == key || (key != null && key.equals(k))))

return first;

//将索引数组的下一个对象赋值给e对象，继续进行判断

if ((e = first.next) != null) {

//当索引数组头结点为红黑树对象，利用红黑树进行查找对象

if (first instanceof TreeNode)

return ((TreeNode<K,V>)first).getTreeNode(hash, key);

do {

//继续进行hash值和key值相等判断，如果有则返回响应的对象

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

return e;

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

}

}//如果查找不到响应的对象，则返回null值

return null;

}

3.3 remove删除键值对方法

(1) 同时传入key值和value值才能删除一个node对象

public boolean remove(Object key, Object value) {

//调用removeNode方法

return removeNode(hash(key), key, value, true, true) != null;

}

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;

//当内存数组table不为null且长度大于0，且索引数组的头结点不为null

if ((tab = table) != null && (n = tab.length) > 0 &&(p = tab[index = (n - 1) & hash]) != null) {

Node<K,V> node = null, e; K k; V v;

//当头结点等于传入的key时，赋值给node

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

node = p;

//当头结点下一个元素不为空时

else if ((e = p.next) != null) {

//判断是否为红黑树节点，如果是则利用红黑树获取该node对象

if (p instanceof TreeNode)

node = ((TreeNode<K,V>)p).getTreeNode(hash, key);

else {//如果不是红黑树节点，则循环该链表继续判断

do {

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

(key != null && key.equals(k)))) {

node = e;

break;

}

p = e;

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

}

}

//当根据key获取的node对象不为空，且value值相等

if (node != null && (!matchValue || (v = node.value) == value ||

(value != null && value.equals(v)))) {

//判断该node对象是否为红黑树对象，如果是调用树的删除方法

if (node instanceof TreeNode)

((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);

//如果node刚好等于头结点，则将头结点移动到next节点

else if (node == p)

tab[index] = node.next;

//如果node为链表中的元素，则将p的next指向下一个元素

else

p.next = node.next;

++modCount;

--size;

afterNodeRemoval(node);

return node;

}

}

return null;}

3.4 replace替换value值方法

(1) 传入key值和new value值

public V replace(K key, V value) {

Node<K,V> e;

//根据key获取的node对象不为null时

//将旧value值替换成新value值，返回旧value值

if ((e = getNode(hash(key), key)) != null) {

V oldValue = e.value;

e.value = value;

afterNodeAccess(e);

return oldValue;

}

return null;

}

(2) 传入key值和old value值和new value值

public boolean replace(K key, V oldValue, V newValue) {

Node<K,V> e; V v;

//key值的判断是否为null，和value值的判断是否等于旧value值

//替换成功则返回true，替换失败则返回false

if ((e = getNode(hash(key), key)) != null &&

((v = e.value) == oldValue || (v != null && v.equals(oldValue)))) {

e.value = newValue;

afterNodeAccess(e);

return true;

}

return false;

}

4. 特殊方法应用

4.1 cotain是否包含相对应的键值对

(1) 传入key值

public boolean containsKey(Object key) {

return getNode(hash(key), key) != null;

}

(2) 传入value值

public boolean containsValue(Object value) {

Node<K,V>[] tab; V v;

if ((tab = table) != null && size > 0) {

//循环内存数组进行判断

for (int i = 0; i < tab.length; ++i) {

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

if ((v = e.value) == value ||

(value != null && value.equals(v)))

return true;

}

}

}

return false;

}

说明：查找key值利用hash算法查找，查找value值则是循环整个内存数组

4.2 获取视图

public Set<Map.Entry<K,V>> entrySet() {

Set<Map.Entry<K,V>> es;

return (es = entrySet) == null ? (entrySet = new EntrySet()) : es;

}

说明：返回此Map中包含的映射的Set视图。 对视图的更改将反映在集合中，反之亦然。

三、总结

1. 根据键保存和获取值的效率都很高，为O(1)，每个单向链表往往只有一个或少数几个节点，根据hash值就可以直接快速定位。

2. HashMap中的键值对没有顺序，因为hash值是随机的。