**[HashMap数据结构](http://www.cnblogs.com/rogerluo1986/p/5851300.html)**

**2.1 HashMap**

**2.1.1 HashMap介绍**

先看看HashMap类头部的源码：

public class HashMap<K,V>

    extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

HashMap基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

此实现假定哈希函数将元素适当地分布在各个桶（数组元素）之间，可为基本操作（get 和 put）提供稳定的性能。迭代 collection 视图所需的时间与 HashMap 实例的“容量”（桶的数量）及其大小（键-值映射关系数）成比例。所以，如果迭代性能很重要，则不要将初始容量设置得太高（或将负载因子设置得太低）。

HashMap 的实例有两个参数影响其性能：初始容量和负载因子。容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。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。当哈希表中的条目数超出了加载因子（加载因子是表示Hsah表中元素的填满的程度.若:加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.反之,加载因子越小,填满的元素越少,好处是:冲突的机会减小了,但:空间浪费多了.冲突的机会越大,则查找的成本越高.反之,查找的成本越小.因而,查找时间就越小. 在Java编程语言中，加载因子默认值为0.75，默认哈希表元为101

）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。

通常，默认加载因子  在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数HashMap 类的操作中，包括get 和put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生rehash 操作。

如果很多映射关系要存储在 HashMap 实例中，则相对于按需执行自动的 rehash 操作以增大表的容量来说，使用足够大的初始容量创建它将使得映射关系能更有效地存储。

注意，此实现不是同步的。如果多个线程同时访问一个哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。（结构上的修改是指添加或删除一个或多个映射关系的任何操作；仅改变与实例已经包含的键关联的值不是结构上的修改。）这一般通过对自然封装该映射的对象进行同步操作来完成。如果不存在这样的对象，则应该使用 [Collections.synchronizedMap](https://www.cnblogs.com/java/util/Collections.html#synchronizedMap(java.util.Map)) 方法来“包装”该映射。最好在创建时完成这一操作，以防止对映射进行意外的非同步访问，如下所示：

   Map m = Collections.synchronizedMap(new HashMap(...));

由所有此类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 [ConcurrentModificationException](https://www.cnblogs.com/java/util/ConcurrentModificationException.html)。因此，面对并发的修改，迭代器很快就会完全失败，而不会在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

**2.1.2 HashMap存储结构图**

这里先给出HashMap的存储结构，在后面的源码分析中，我们将更加详细的对此作介绍。HashMap采取数组加链表的存储方式来实现。亦即数组（散列桶）中的每一个元素都是链表，如下图：

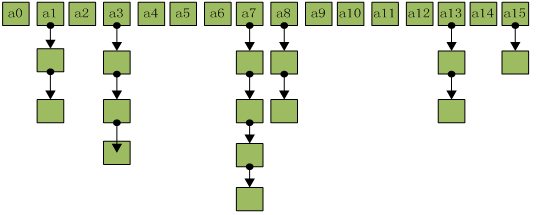


      图2-1

说明：下面针对HashMap的源码分析中，所有提到的桶或散列桶都表示存储结构中数组的元素，桶或散列桶的数量亦即表示数组的长度，哈希码亦即散列码。

**2.1.3 属性分析**

先来看看HashMap有哪些属性，HashMap没有从AbstractMap父亲中继承任何属性，下面这些都是HashMap的属性：

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

DEFAULT\_INITIAL\_CAPACITY是HashMap默认的初始化桶数量，如图2-1中所示。对于HashMap中桶数量的值必须是2的N次幂，而且这个是HashMap强制规定的。这样做的原因就是因为计算机进行2次幂的运算是非常高效的，仅通过位移操作就可以完成2的N次幂的运算。

static final int MAXIMUM\_CAPACITY = 1 << 30;

MAXIMUM\_CAPACITY是HashMap中散列桶数量的最大值，从上面的代码可知这个最大值为2的30次幂，即1073741824。

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

默认的负载因子，如果在在创建HashMap的构造函数中没有指定负载因子，则指定该HashMap的默认负载因子为0.75，这意味着当HashMap中条目的数量达到了条目数量75%时，HashMap将进行resize操作以增加桶的数量。对于桶的扩展，等分析到下面的具体时会作更详细的介绍。

transient Entry<K,V>[] table;

table就是HashMap的存储结构，显然这是一个数组，数组的每一个元素都是一个条目（Entry），Entry是HashMap中的一个内部类，它有如下4个属性：final K key;V value;Entry<K,V> next;int hash。分别为键、值、指向下一个链表结点的指针、散列（哈希）值。这就是图2.1中HashMap存储结构的代码实现。

transient int size;

size表示HashMap中条目（即键-值对）的数量。

int threshold;

threshold是HashMap的重构阈值，它的值为容量和负载因子的乘积。在HashMap中所有桶中条目的总数量达到了这个重构阈值之后，HashMap将进行resize操作以自动扩容。

final float loadFactor;

loadFactor表示HashMap的负载因子，它和容量一样都是HashMap扩容的决定性因素。

transient int modCount;

modCount表示HashMap被结构化更新的次数，比如插入、删除、清空等会更新HashMap结构的操作次数。

static final int ALTERNATIVE\_HASHING\_THRESHOLD\_DEFAULT

= Integer.MAX\_VALUE;

ALTERNATIVE\_HASHING\_THRESHOLD\_DEFAULT表示在对字符串键（即key为String类型）的HashMap应用备选哈希函数时HashMap的条目数量的默认阈值。备选哈希函数的使用可以减少由于对字符串键进行弱哈希码计算时的碰撞概率。

transient boolean useAltHashing;

useAltHashing表示是否要对字符串键的HashMap使用备选哈希函数。

transient final int hashSeed = sun.misc.Hashing.randomHashSeed(this);

hashSeed表示一个与当前实例关联并且可以减少哈希碰撞概率应用于键的哈希码计算的随机种子。

**2.1.4 构造分析**

HashMap提供了4个构造方法，按照它们在源码中的位置顺序从上至下列出：

HashMap(int initialCapacity, float loadFactor)

HashMap(int initialCapacity)

HashMap()

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

(1) 我们先来分析第一个同时传递初始化容量参数和负载因子参数的源码，因为其它的3个构造方法都会调用这个构造方法，下面给出这个方法的代码及分析：

public HashMap(int initialCapacity, float loadFactor) {

//部分构造参数容错处理的源码已省略...

    /\*\*

     \* 根据传入的初始化容量计算该HashMap的容量（即桶的数量）

     \* 算法为：将capacity进行不断的左移，直至capacity大于或等于初始化容量

    \*/

    int capacity = 1;

    while (capacity < initialCapacity)

        capacity <<= 1;

    //负载因子初始化

    this.loadFactor = loadFactor;

    /\*\*

     \* 条目阈值的计算

     \* 算法：超出条目最大容量前取容量与负载因子的乘积作为条目阈值

     \*/

    threshold = (int)Math.min(capacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

    //创建数组（散列桶）

    table = new Entry[capacity];

    //计算是否对字符串键的HashMap使用备选哈希函数

    useAltHashing = sun.misc.VM.isBooted() &&

        (capacity >= Holder.ALTERNATIVE\_HASHING\_THRESHOLD);

init();//调用初始化方法，默认情况下什么也没做

}

(2) 下面是只传初始化容量参数的构造方法：

public HashMap(int initialCapacity) {

    //初始化容量传入，加载因子为默认值0.75f

    this(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

(3) 下面是无参构造方法：

public HashMap() {

    //初始化容量为默认值16，加载因子也为默认值0.75f

    this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_LOAD\_FACTOR);

}

(4) 下面是根据已有Map构造新HashMap的构造方法：

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

    /\*\*

     \* 取下面两个值的较大的值作为当前要构造的HashMap的初始容量

     \* 第1个值：用传入的Map的条目数量除以默认加载因子再加上1

     \* 第2个值：默认的初始化容量

     \*/

     this(Math.max((int) (m.size() / DEFAULT\_LOAD\_FACTOR) + 1,

          DEFAULT\_INITIAL\_CAPACITY), DEFAULT\_LOAD\_FACTOR);

     /\*\*

      \* 把传入的map里的所有条目放入当前已构造的HashMap中

      \* 关于putAllForCreate方法后面会作分析

     \*/

     putAllForCreate(m);

}

**2.1.5 hash方法**

hash方法的源码及分析如下：

final int hash(Object k) {

int h = 0;

/\*\*

   \* 如果useAltHashing的值为true

     \*      并且键的类型为String，则对字符串键使用备选哈希函数

     \*     否则，返回用于对键进行哈希码计算的随机种子hashSeed

     \* 关于hashSeed在2.1.3.1小节中已介绍过，这里不再赘述

     \*/

if (useAltHashing) {

if (k instanceof String) {

return sun.misc.Hashing.stringHash32((String) k);

        }

        h = hashSeed;

    }

    /\*\*

     \* 对h和键的哈希码进行抑或并赋值运算

     \* 等价于h = h ^ k.hashCode();

     \*/

h ^= k.hashCode();

    //下面两步的运算过程如图2-2所示

    h ^= (h >>> 20) ^ (h >>> 12);

    return h ^ (h >>> 7) ^ (h >>> 4);

}

假设h=0x7FFFFFFF，则上面最后两步对h的运算过程如下图：

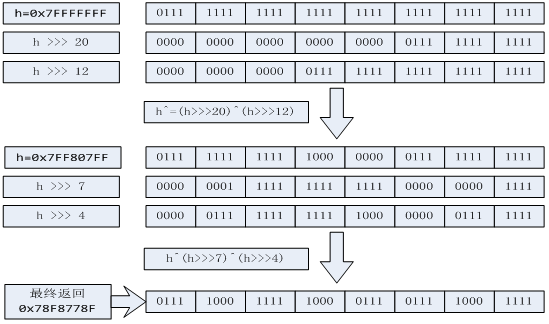


图2-2

**2.1.6 indexFor方法**

/\*\*

 \* h表示通过hash(Object k)方法计算得来的哈希码

 \* length表示桶的数量（即数组的长度）

 \*/

static int indexFor(int h, int length) {

/\*\*

 \* 将哈希码和length进行按位与运算

 \* 所有的h值都会在映射在闭区间[0,length-1]内

 \* 不同的h值可能映射到闭区间[0,length-1]内同一个值上

 \*/

    return h & (length-1);

}

**2.1.7 put方法**

/\*\*

\* 在HashMap中存储一个键值对，若指定的键已经存在于HashMap中

\* 则将新的值替换掉旧值，否则新添加一个条目来存储这个键值对

\* @param key 指定的键

\* @param value 指定的值

\* @return 若该键已经存在则返回该键对应的旧值，否则返回null

\*/

public V put(K key, V value) {

    if (key == null)

     /\*\*

      \* 若键为null，则调用putForNullKey方法进行插入

      \* putForNullKey的源码这里不再分析，读者有兴趣可以自行分析它的源码

      \*/

        return putForNullKey(value);

    //下面这两个方法在前面两小节中已经分析过

    int hash = hash(key);//计算键对应的哈希码

    int i = indexFor(hash, table.length);//计算桶的索引

    //遍历桶中所有的元素（即链表的结点）

    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))) {

         //桶中已经存在指定的键，替换指定键对应的旧值并返回该旧值

            V oldValue = e.value;

            e.value = value;

            e.recordAccess(this);

            return oldValue;

        }

    }

modCount++;

    /\*\*

     \* 桶中不存在指定键，则调用addEntry方法添加向桶中添加新结点

     \* addEntry方法下一小节将会详细介绍

     \*/

    addEntry(hash, key, value, i);

    return null;

}

**2.1.8 addEntry方法**

/\*\*

 \* 向HashMap的指定桶中添加一个新的键对值

 \* 若要对HashMap扩容（即增加桶的数量），则下面的方法可能会修改传入的桶索引

 \* @param hash 指定键对应的哈希码

 \* @param key 指定键

 \* @param value 指定值

 \* @param bucketIndex 桶索引

 \*/

void addEntry(int hash, K key, V value, int bucketIndex) {

    if ((size >= threshold) && (null != table[bucketIndex])) {

     //如果HashMap中条目的数量达到了重构阈值且指定的桶不为null，则对HashMap进行扩容（即增加桶的数量）

     /\*\*

      \* 调用resize方法对HashMap进行扩容

      \* 对于resize方法，下面会有专门的一小节来作介绍，这里先不介绍

      \*/

        resize(2 \* table.length);

        //扩容后，桶的数量增加了，故需要重新对键进行哈希码的计算

        hash = (null != key) ? hash(key) : 0;

        //根据新的键哈希码和新的桶数量重新计算桶索引值

        bucketIndex = indexFor(hash, table.length);

    }

    /\*\*

     \* 在指定的桶中创建一个新的条目以存储我们传入的键值对

     \* 对于createEntry方法，读者若有兴趣可以自行阅读其源码

     \*/

    createEntry(hash, key, value, bucketIndex);

}

**2.1.9 resize方法**

/\*\*

 \* 重新调整HashMap中桶的数量

 \* @param newCapacity 新的桶数量

 \*/

void resize(int newCapacity) {

/\*\*

 \* 下面的这段代码对新值进行判断

 \* 如果新值超过了条目(Entry)数量的最大值

 \* 则新int最大值赋值给重构阈值然后，然后直接返回而不会进行扩容

 \*/

    Entry[] oldTable = table;

    int oldCapacity = oldTable.length;

    if (oldCapacity == MAXIMUM\_CAPACITY) {

        threshold = Integer.MAX\_VALUE;

        return;

    }

    //若newCapacity合法，则新建一个桶数组。

    Entry[] newTable = new Entry[newCapacity];

    //计算是否需要对键重新进行哈希码的计算

    boolean oldAltHashing = useAltHashing;

    useAltHashing |= sun.misc.VM.isBooted() &&

            (newCapacity >= Holder.ALTERNATIVE\_HASHING\_THRESHOLD);

    boolean rehash = oldAltHashing ^ useAltHashing;

    /\*\*

     \* 将原有所有的桶迁移至新的桶数组中

     \* 在迁移时，桶在桶数组中的绝对位置可能会发生变化

     \* 这就是为什么HashMap不能保证存储条目的顺序不能恒久不变的原因

     \* 读者若有兴趣，可以自行阅读transfer方法的源码

     \*/

    transfer(newTable, rehash);

    //将新的桶数组的引用赋值给旧数组

    table = newTable;

    //像构造方法中一样来重新计算重构阈值

    threshold = (int)Math.min(newCapacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

}

**2.1.10 get方法**

/\*\*

 \* 根据指定键获取该键对应的值

 \* @param key 指定键

 \* @return 若该键存在于HashMap中，则返回该键对应的值，否则返回null

 \*/

public V get(Object key) {

    if (key == null)

     //若键为null,则返回null键对应的值

        return getForNullKey();

    //根据键获取条目，下一小节会单独介绍getEntry方法

    Entry<K,V> entry = getEntry(key);

    //返回条目的值，若条目为null，则返回null

    return null == entry ? null : entry.getValue();

}

**数据结构**

[复制代码](javascript:void(0);)

## HashMap的数据结构

## 数据结构中有数组和链表来实现对数据的存储，但这两者基本上是两个极端。

## 数组：数组存储区间是连续的，占用内存严重，故空间复杂的很大。但数组的二分查找时间复杂度小，为O(1)；数组的特点是：寻址容易，插入和删除困难；

## 链表：链表存储区间离散，占用内存比较宽松，故空间复杂度很小，但时间复杂度很大，达O（N）。链表的特点是：寻址困难，插入和删除容易。

## 哈希表

## 那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？

## 答案是肯定的，这就是我们要提起的哈希表。

## 哈希表（(Hash table）既满足了数据的查找方便，同时不占用太多的内容空间，使用也十分方便。

## 哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法—— 拉链法，我们可以理解为“链表的数组” ，如图：

## https://images2015.cnblogs.com/blog/795730/201607/795730-20160702194158093-77003893.jpg https://images2015.cnblogs.com/blog/795730/201607/795730-20160702194159202-333261561.jpg

从上图我们可以发现哈希表是由【数组+链表】组成的，一个长度为16的数组中，每个元素存储的是一个链表的头结点。

那么这些元素是按照什么样的规则存储到数组中呢？

一般情况是通过【hash(key)%len】获得，也就是元素的key的哈希值对数组长度取模得到。

比如上述哈希表中，12%16=12,28%16=12,108%16=12,140%16=12。所以12、28、108以及140都存储在数组下标为12的位置。

HashMap也可以理解为其存储数据的容器就是一个【线性数组】。

这可能让我们很不解，一个线性的数组怎么实现按键值对来存取数据呢？

这里HashMap有做一些处理。首先HashMap里面实现一个静态内部类Entry，其重要的属性有 key , value, next。从属性key,value我们就能很明显的看出来Entry就是HashMap键值对实现的一个基础bean，我们上面说到HashMap的基础就是一个线性数组，这个数组就是Entry[]，Map里面的内容都保存在Entry[]里面。

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

**transient** Entry[] table;

[复制代码](javascript:void(0);)

**存数据的逻辑**

[复制代码](javascript:void(0);)

既然是线性数组，为什么能随机存取？这里HashMap用了一个小算法，大致是这样实现：

//存储时:

**int** hash = key.hashCode(); // 每个key的hash是一个固定的int值

**int** index = hash % Entry[].length;// 去模运算，运算后的值肯定在0-length之间

Entry[index] = value;// 以去模后的值为索引，把value存进去

疑问：如果两个key通过hash%Entry[].length得到的index相同，会不会有覆盖的危险？

这里HashMap里面用到链式数据结构的一个概念。

上面我们提到过Entry类里面有一个next属性，作用是指向下一个Entry。

打个比方， 第一个键值对A进来，通过计算其key的hash得到的index=0，记做:Entry[0] = A。

一会后又进来一个键值对B，通过计算其index也等于0，现在怎么办？

HashMap会这样做:B.next = A,Entry[0] = B。

如果又进来C,index也等于0,那么C.next = B,Entry[0] = C；

这样我们发现index=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起。

**public** V put(K key, V value) {

**if** (key == **null**) **return** putForNullKey(value); //null总是放在数组的第一个链表中

**int** hash = hash(key.hashCode());

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

        //遍历链表

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

            Object k;

            //如果key在链表中已存在，则替换为新value（不要误解为是用新的值把旧的值覆盖了！）

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

                V oldValue = e.value;

                e.value = value;

                e.recordAccess(**this**);

**return** oldValue;

            }

        }

        modCount++;

        addEntry(hash, key, value, i);

**return** **null**;

    }

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

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

        table[bucketIndex] = **new** Entry<K, V>(hash, key, value, e); //参数e, 是Entry.next

        //如果size超过threshold，则扩充table大小。再散列

**if** (size++ >= threshold) resize(2 \* table.length);

    }

当然HashMap里面也包含一些优化方面的实现，比如：Entry[]的长度一定后，随着map里面数据的越来越长，这样同一个index的链就会很长，会不会影响性能？

HashMap里面设置一个因子，随着map的size越来越大，Entry[]会以一定的规则加长长度。

[复制代码](javascript:void(0);)

**取数据的逻辑**

[复制代码](javascript:void(0);)

//取值时:

**int** hash = key.hashCode();

**int** index = hash % Entry[].length;

**return** Entry[index];

**public** V get(Object key) {

**if** (key == **null**) **return** getForNullKey();

**int** hash = hash(key.hashCode());

        //先定位到数组元素，再遍历该元素处的链表

**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**;

    }

[复制代码](javascript:void(0);)

**其他逻辑**

[复制代码](javascript:void(0);)

null key的存取

null key总是存放在Entry[]数组的第一个元素。

**private** V putForNullKey(V value) {

**for** (Entry<K, V> e = table[0]; e != **null**; e = e.next) {

**if** (e.key == **null**) {

                V oldValue = e.value;

                e.value = value;

                e.recordAccess(**this**);

**return** oldValue;

            }

        }

        modCount++;

        addEntry(0, **null**, value, 0);

**return** **null**;

    }

**private** V getForNullKey() {

**for** (Entry<K, V> e = table[0]; e != **null**; e = e.next) {

**if** (e.key == **null**) **return** e.value;

        }

**return** **null**;

    }

确定数组index：hashcode % table.length取模

HashMap存取时，都需要计算当前key应该对应Entry[]数组哪个元素，即计算数组下标；算法如下：

    /\*\* Returns index for hash code h. \*/

**static** **int** indexFor(**int** h, **int** length) {

**return** h & (length - 1);

    }

按位取并，作用上相当于取模mod或者取余%。

注意：不过的hashCode进行运算后的值可能相等，这意味着数组下标相同；但是，不要错误的理解为数组下标相同表示hashCode相同。

初始大小

**public** HashMap(**int** initialCapacity, **float** loadFactor) {

    .....

    // Find a power of 2 >= initialCapacity

**int** capacity = 1;

**while** (capacity < initialCapacity)

        capacity <<= 1;

**this**.loadFactor = loadFactor;

    threshold = (**int**)(capacity \* loadFactor);

    table = **new** Entry[capacity];

    init();

}

注意初始大小并不是构造函数中的initialCapacity！而是 >= initialCapacity的2的n次幂！！！！！

[复制代码](javascript:void(0);)

**解决hash冲突的方法**

开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）

再哈希法

链地址法

建立一个公共溢出区

Java中HashMap的解决办法是采用的链地址法。

**再散列过程**

[复制代码](javascript:void(0);)

当哈希表的容量超过默认容量时，必须调整table的大小。

当容量已经达到最大可能值时，那么该方法就将容量调整到Integer.MAX\_VALUE返回，这时，需要创建一张新表，将原表映射到新表中。

    /\*\*

    \* Rehashes the contents of this map into a new array with a

    \* larger capacity.  This method is called automatically when the

    \* number of keys in this map reaches its threshold.

    \*

    \* If current capacity is MAXIMUM\_CAPACITY, this method does not

    \* resize the map, but sets threshold to Integer.MAX\_VALUE.

    \* This has the effect of preventing future calls.

    \*

    \* **@param** newCapacity the new capacity, MUST be a power of two;

    \*        must be greater than current capacity unless current

    \*        capacity is MAXIMUM\_CAPACITY (in which case value

    \*        is irrelevant).

    \*/

**void** resize(**int** newCapacity) {

        Entry[] oldTable = table;

**int** oldCapacity = oldTable.length;

**if** (oldCapacity == MAXIMUM\_CAPACITY) {

            threshold = Integer.*MAX\_VALUE*;

**return**;

        }

        Entry[] newTable = **new** Entry[newCapacity];

        transfer(newTable);

        table = newTable;

        threshold = (**int**) (newCapacity \* loadFactor);

    }

    /\*\*

     \* Transfers all entries from current table to newTable.

     \*/

**void** transfer(Entry[] newTable) {

        Entry[] src = table;

**int** newCapacity = newTable.length;

**for** (**int** j = 0; j < src.length; j++) {

            Entry<K, V> e = src[j];

**if** (e != **null**) {

                src[j] = **null**;

**do** {

                    Entry<K, V> next = e.next;

                    //重新计算index

**int** i = indexFor(e.hash, newCapacity);

                    e.next = newTable[i];

                    newTable[i] = e;

                    e = next;

                } **while** (e != **null**);

            }

        }

    }

[复制代码](javascript:void(0);)

# [Java集合——HashMap、HashTable以及ConCurrentHashMap异同比较](http://blog.csdn.net/seu_calvin/article/details/52653711)

2016-09-25 15:56 13599人阅读 [评论](http://blog.csdn.net/seu_calvin/article/details/52653711#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/seu_calvin/article/details/52653711#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Java（30） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**0. 前言**

HashMap和HashTable的区别一种比较简单的回答是：

（1）HashMap是非线程安全的，HashTable是线程安全的。

（2）HashMap的键和值都允许有null存在，而HashTable则都不行。

（3）因为线程安全、哈希效率的问题，HashMap效率比HashTable的要高。

但是如果继续追问：Java中的另一个线程安全的与HashMap功能极其类似的类是什么？

同样是线程安全，它与HashTable在线程同步上有什么不同？带着这些问题，开始今天的文章。

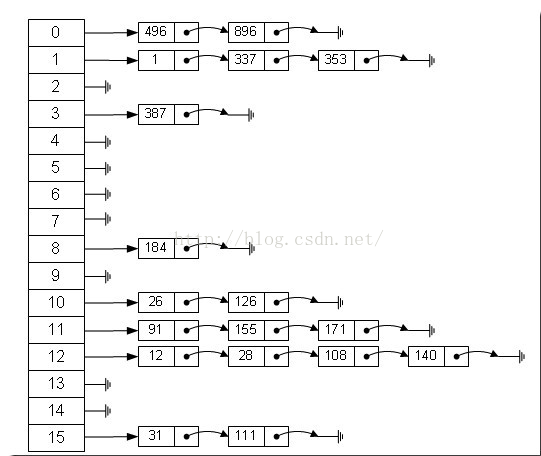
本文为原创，相关内容会持续维护，转载请标明出处：<http://blog.csdn.net/seu_calvin/article/details/52653711>。

**1．  HashMap概述**

Java中的数据存储方式有两种结构，一种是数组，另一种就是链表，前者的特点是连续空间，寻址迅速，但是在增删元素的时候会有较大幅度的移动，所以数组的特点是查询速度快，增删较慢。

而链表由于空间不连续，寻址困难，增删元素只需修改指针，所以链表的特点是查询速度慢、增删快。

那么有没有一种数据结构来综合一下数组和链表以便发挥他们各自的优势？答案就是哈希表。哈希表的存储结构如下图所示：



从上图中，我们可以发现哈希表是由数组+链表组成的，一个长度为16的数组中，每个元素存储的是一个链表的头结点，通过功能类似于hash(key.hashCode())%len的操作，获得要添加的元素所要存放的的数组位置。

HashMap的哈希算法实际操作是通过位运算，比取模运算效率更高，同样能达到使其分布均匀的目的，后面会介绍。

键值对所存放的数据结构其实是HashMap中定义的一个Entity内部类，数组来实现的，属性有key、value和指向下一个Entity的next。

**2．  HashMap初始化**

HashMap有两种常用的构造方法：

第一种是不需要参数的构造方法：

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **static** **final** **int** DEFAULT\_INITIAL\_CAPACITY = 16; //初始数组长度为16
2. **static** **final** **int** MAXIMUM\_CAPACITY = 1 << 30; //最大容量为2的30次方
3. //装载因子用来衡量HashMap满的程度
4. //计算HashMap的实时装载因子的方法为：size/capacity
5. **static** **final** **float** DEFAULT\_LOAD\_FACTOR = 0.75f; //装载因子
7. **public** HashMap() {
8. **this**.loadFactor = DEFAULT\_LOAD\_FACTOR;
9. threshold = (**int**)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);
10. //默认数组长度为16
11. table = **new** Entry[DEFAULT\_INITIAL\_CAPACITY];
12. init();
13. }

第二种是需要参数的构造方法：

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **public** HashMap(**int** initialCapacity, **float** loadFactor) {
2. **if** (initialCapacity < 0)
3. **throw** **new** IllegalArgumentException("Illegal initial capacity: " + initialCapacity);
4. **if** (initialCapacity > MAXIMUM\_CAPACITY)
5. initialCapacity = MAXIMUM\_CAPACITY;
6. **if** (loadFactor <= 0 || Float.isNaN(loadFactor))
7. **throw** **new** IllegalArgumentException("Illegal load factor: " + loadFactor);
9. // Find a power of 2 >= initialCapacity
10. **int** capacity = 1;
11. **while** (capacity < initialCapacity)
12. capacity <<= 1;
14. **this**.loadFactor = loadFactor;
15. threshold = (**int**)(capacity \* loadFactor);
16. table = **new** Entry[capacity];
17. init();
18. }

从源码可以看出，初始化的数组长度为capacity，capacity的值总是2的N次方，大小比第一个参数稍大或相等。

**3.  HashMap的put操作**

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **public** V put(K key, V value) {
2. **if** (key == **null**)
3. **return** putForNullKey(value);
4. **int** hash = hash(key.hashCode());
5. **int** i = indexFor(hash, table.length);
6. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
7. Object k;
8. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
9. V oldValue = e.value;
10. e.value = value;
11. e.recordAccess(**this**);
12. **return** oldValue;
13. }
14. }
15. modCount++;
16. addEntry(hash, key, value, i);
17. **return** **null**;
18. }

**3.1  put进的key为null**

从源码中可以看出，HashMap是允许key为null的，会调用putForNullKey()方法：

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **private** V putForNullKey(V value) {
2. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
3. **if** (e.key == **null**) {
4. V oldValue = e.value;
5. e.value = value;
6. e.recordAccess(**this**);
7. **return** oldValue;
8. }
9. }
10. modCount++;
11. addEntry(0, **null**, value, 0);
12. **return** **null**;
13. }
15. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
16. Entry<K,V> e = table[bucketIndex];
17. table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e);
18. **if** (size++ >= threshold)
19. resize(2 \* table.length);
20. }

putForNullKey方法会遍历以table[0]为链表头的链表，如果存在key为null的KV，那么替换其value值并返回旧值。否则调用addEntry方法，这个方法也很简单，将[null,value]放在table[0]的位置，并将新加入的键值对封装成一个Entity对象，将其next指向原table[0]处的Entity实例。

size表示HashMap中存放的所有键值对的数量。

threshold = capacity\*loadFactor，最后几行代码表示当HashMap的size大于threshold时会执行resize操作，将HashMap扩容为原来的2倍。扩容需要重新计算每个元素在数组中的位置，indexFor()方法中的table.length参数也证明了这一点。

但是扩容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，那么我们就该声明new HashMap(2048)，因为需要考虑默认的0.75的扩容因子和数组数必须是2的N次方。若使用声明new HashMap(1024)那么put过程中会进行扩容。

**3.2  put进的key不为null**

将上述put方法中的相关代码复制一下方便查看：

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **int** hash = hash(key.hashCode());
2. **int** i = indexFor(hash, table.length);
3. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
4. Object k;
5. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
6. V oldValue = e.value;
7. e.value = value;
8. e.recordAccess(**this**);
9. **return** oldValue;
10. }
11. }
12. modCount++;
13. addEntry(hash, key, value, i);
14. **return** **null**;
15. }

从源码可以看出，第1、2行计算将要put进的键值对的数组的位置i。第4行判断加入的key是否和以table[i]为链表头的链表中所有的键值对有重复，若重复则替换value并返回旧值，若没有重复则调用addEntry方法，上面对这个方法的逻辑已经介绍过了。

至此HashMap的put操作已经介绍完毕了。

**4.  HashMap的get操作**

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **public** V get(Object key) {
2. **if** (key == **null**)
3. **return** getForNullKey();
4. **int** hash = hash(key.hashCode());
5. **for** (Entry<K,V> e = table[indexFor(hash, table.length)]; e != **null**; e = e.next) {
6. Object k;
7. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))
8. **return** e.value;
9. }
10. **return** **null**;
11. }
13. **private** V getForNullKey() {
14. **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {
15. **if** (e.key == **null**)
16. **return** e.value;
17. }
18. **return** **null**;
19. }

如果了解了前面的put操作，那么这里的get操作逻辑就很容易理解了，源码中的逻辑已经非常非常清晰了。

需要注意的只有当找不到对应value时，返回的是null。或者value本身就是null。这是可以通过containsKey()来具体判断。

了解了上面HashMap的put和get操作原理，可以通过下面这个小例题进行知识巩固，题目是打印在数组中出现n/2以上的元素，我们便可以使用HashMap的特性来解决。

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **public** **class** HashMapTest {
2. **public** **static** **void** main(String[] args) {
3. **int** [] a = {2,1,3,2,0,4,2,1,2,3,1,5,6,2,2,3};
4. Map<Integer, Integer> map = **new** HashMap<Integer,Integer>();
5. **for**(**int** i=0; i<a.length; i++){
6. **if**(map.containsKey(a[i])){
7. **int** tmp = map.get(a[i]);
8. tmp+=1;
9. map.put(a[i], tmp);
10. }**else**{
11. map.put(a[i], 1);
12. }
13. }
14. Set<Integer> set = map.keySet();
15. **for** (Integer s : set) {
16. **if**(map.get(s)>=a.length/2){
17. System.out.println(s);
18. }
19. }
20. }
21. }

**5.  HashMap和HashTable的对比**

HashTable和HashMap采用相同的存储机制，二者的实现基本一致，不同的是：

（1）HashMap是非线程安全的，HashTable是线程安全的，内部的方法基本都经过synchronized修饰。

（2）因为同步、哈希性能等原因，性能肯定是HashMap更佳，因此HashTable已被淘汰。

（3） HashMap允许有null值的存在，而在HashTable中put进的键值只要有一个null，直接抛出NullPointerException。

（4）HashMap默认初始化数组的大小为16，HashTable为11。前者扩容时乘2，使用位运算取得哈希，效率高于取模。而后者为乘2加1，都是素数和奇数，这样取模哈希结果更均匀。

这里本来我没有仔细看两者的具体哈希算法过程，打算粗略比较一下区别就过的，但是最近师姐面试美团移动开发时被问到了稍微具体一些的算法过程，我也是醉了…不过还是恭喜师姐面试成功，起薪20W，真是羡慕，希望自己一年后找工作也能顺顺利利的。

言归正传，看下两种集合的hash算法。看源码也不难理解。

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. //HashMap的散列函数，这里传入参数为键值对的key
2. **static** **final** **int** hash(Object key) {
3. **int** h;
4. **return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
5. }
6. //返回hash值的索引，h & (length-1)操作等价于 hash % length操作， 但&操作性能更优
7. **static** **int** indexFor(**int** h, **int** length) {
8. // length must be a non-zero power of 2
9. **return** h & (length-1);
10. }
12. //HashTable的散列函数直接在put方法里实现了
13. **int** hash = key.hashCode();
14. **int** index = (hash & 0x7FFFFFFF) % tab.length;

**6.  HashTable和ConCurrentHashMap的对比**

先对ConcurrentHashMap进行一些介绍吧，它是线程安全的HashMap的实现。

HashTable里使用的是synchronized关键字，这其实是对对象加锁，锁住的都是对象整体，当Hashtable的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。

ConcurrentHashMap算是对上述问题的优化，其构造函数如下，默认传入的是16，0.75，16。

**[java]** [view plain](http://blog.csdn.net/seu_calvin/article/details/52653711) [copy](http://blog.csdn.net/seu_calvin/article/details/52653711)

1. **public** ConcurrentHashMap(**int** paramInt1, **float** paramFloat, **int** paramInt2)  {
2. //…
3. **int** i = 0;
4. **int** j = 1;
5. **while** (j < paramInt2) {
6. ++i;
7. j <<= 1;
8. }
9. **this**.segmentShift = (32 - i);
10. **this**.segmentMask = (j - 1);
11. **this**.segments = Segment.newArray(j);
12. //…
13. **int** k = paramInt1 / j;
14. **if** (k \* j < paramInt1)
15. ++k;
16. **int** l = 1;
17. **while** (l < k)
18. l <<= 1;
20. **for** (**int** i1 = 0; i1 < **this**.segments.length; ++i1)
21. **this**.segments[i1] = **new** Segment(l, paramFloat);
22. }
24. **public** V put(K paramK, V paramV)  {
25. **if** (paramV == **null**)
26. **throw** **new** NullPointerException();
27. **int** i = hash(paramK.hashCode()); //这里的hash函数和HashMap中的不一样
28. **return** **this**.segments[(i >>> **this**.segmentShift & **this**.segmentMask)].put(paramK, i, paramV, **false**);
29. }

ConcurrentHashMap引入了分割(Segment)，上面代码中的最后一行其实就可以理解为把一个大的Map拆分成N个小的HashTable，在put方法中，会根据hash(paramK.hashCode())来决定具体存放进哪个Segment，如果查看Segment的put操作，我们会发现内部使用的同步机制是基于lock操作的，这样就可以对Map的一部分（Segment）进行上锁，这样影响的只是将要放入同一个Segment的元素的put操作，保证同步的时候，锁住的不是整个Map（HashTable就是这么做的），相对于HashTable提高了多线程环境下的性能，因此HashTable已经被淘汰了。

**7.  HashMap和ConCurrentHashMap的对比**

最后对这俩兄弟做个区别总结吧：

（1）经过4.2的分析，我们知道ConcurrentHashMap对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用lock锁进行保护，相对于HashTable的syn关键字锁的粒度更精细了一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。

（2）HashMap的键值对允许有null，但是ConCurrentHashMap都不允许。

至此对HashMap、HashTable以及ConCurrentHashMap异同比较总结完毕。