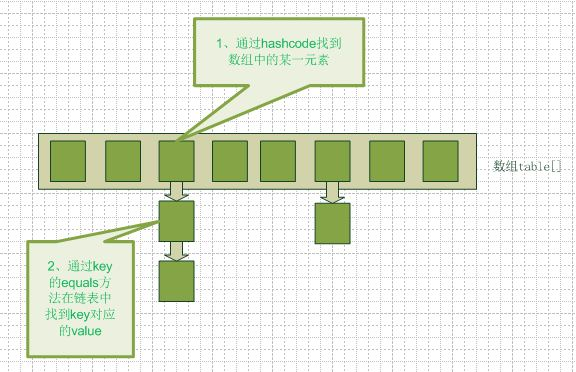
# [JDK1.7中HashMap底层实现原理](https://www.cnblogs.com/dijia478/p/8006713.html)

# **一、数据结构**

HashMap中的数据结构是数组+单链表的组合，以键值对(key-value)的形式存储元素的，通过put()和get()方法储存和获取对象。



（方块表示Entry对象，横排表示数组table[]，纵排表示哈希桶bucket【实际上是一个由Entry组成的链表，新加入的Entry放在链头，最先加入的放在链尾】，）

# **二、实现原理**

## **成员变量**

源码分析：

/\*\* 初始容量，默认16 \*/

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; //aka 16 /\*\* 最大初始容量，2^30 \*/

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

/\*\* 负载因子，默认0.75，负载因子越小，hash冲突机率越低 \*/

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

/\*\* 初始化一个Entry的空数组 \*/

static final Entry<?,?>[] EMPTY\_TABLE = {};

/\*\* 将初始化好的空数组赋值给table，table数组是HashMap实际存储数据的地方，并不在EMPTY\_TABLE数组中 \*/

transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY\_TABLE;

/\*\* HashMap实际存储的元素个数 \*/

transient int size;

/\*\* 临界值（HashMap 实际能存储的大小）,公式为(threshold = capacity \* loadFactor) \*/

int threshold;

/\*\* 负载因子 \*/

final float loadFactor;

/\*\* HashMap的结构被修改的次数，用于迭代器 \*/

transient int modCount;

**构造方法**

源码分析：

public HashMap(int initialCapacity, float loadFactor) {

// 判断设置的容量和负载因子合不合理

if (initialCapacity < 0)

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

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal load factor: " +loadFactor);

// 设置负载因子，临界值此时为容量大小，后面第一次put时由inflateTable(int toSize)方法计算设置

this.loadFactor = loadFactor;

threshold = initialCapacity;

init();

}

public HashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

public HashMap() {

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

}

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

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

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

inflateTable(threshold);

putAllForCreate(m);

}

## **put方法**

put()源码分析：

public V put(K key, V value) {

// 如果table引用指向成员变量EMPTY\_TABLE，那么初始化HashMap（设置容量、临界值，新的Entry数组引用）

if (table == EMPTY\_TABLE) {

inflateTable(threshold);

}

// 若“key为null”，则将该键值对添加到table[0]处，遍历该链表，如果有key为null，则将value替换。没有就创建新Entry对象放在链表表头

// 所以table[0]的位置上，永远最多存储1个Entry对象，形成不了链表。key为null的Entry存在这里

if (key == null)

return putForNullKey(value);

// 若“key不为null”，则计算该key的哈希值

int hash = hash(key);

// 搜索指定hash值在对应table中的索引

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

// 循环遍历table数组上的Entry对象，判断该位置上key是否已存在

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

Object k;

// 哈希值相同并且对象相同

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

// 如果这个key对应的键值对已经存在，就用新的value代替老的value，然后退出！

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

// 修改次数+1

modCount++;

// table数组中没有key对应的键值对，就将key-value添加到table[i]处 addEntry(hash, key, value, i);

return null;

}

可以看到，当我们给put()方法传递键和值时，HashMap会由key来调用hash()方法，返回键的hash值，计算Index后用于找到bucket（哈希桶）的位置来储存Entry对象。

如果两个对象key的hash值相同，那么它们的bucket位置也相同，但equals()不相同，添加元素时会发生hash碰撞，也叫hash冲突，HashMap使用链表来解决碰撞问题。

分析源码可知，put()时，HashMap会先遍历table数组，用hash值和equals()判断数组中是否存在完全相同的key对象， 如果这个key对象在table数组中已经存在，就用新的value代替老的value。如果不存在，就创建一个新的Entry对象添加到table[ i ]处。

如果该table[ i ]已经存在其他元素，那么新Entry对象将会储存在bucket链表的表头，通过next指向原有的Entry对象，形成链表结构（hash碰撞解决方案）。

Entry数据结构源码如下（HashMap内部类）：

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

final K key;

V value;

/\*\* 指向下一个元素的引用 \*/

Entry<K,V> next;

int hash;

/\*\*

\* 构造方法为Entry赋值

\*/

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

...

...

}

形成单链表的核心代码如下：

/\*\*

\* 将Entry添加到数组bucketIndex位置对应的哈希桶中，并判断数组是否需要扩容

\*/

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

// 如果数组长度大于等于容量×负载因子，并且要添加的位置为null

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

// 长度扩大为原数组的两倍，代码分析见下面扩容机制

resize(2 \* table.length);

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

bucketIndex = indexFor(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

/\*\*

\* 在链表中添加一个新的Entry对象在链表的表头

\*/

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

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

table[bucketIndex] = new Entry<>(hash, key, value, e);

size++;

}

## **IMG_267**

（put方法执行过程）

## **get方法**

如果两个不同的key的hashcode相同，两个值对象储存在同一个bucket位置，要获取value，我们调用get()方法，HashMap会使用key的hashcode找到bucket位置，因为HashMap在链表中存储的是Entry键值对，所以找到bucket位置之后，会调用key的equals()方法，按顺序遍历链表的每个 Entry，直到找到想获取的 Entry 为止——如果恰好要搜索的 Entry 位于该 Entry 链的最末端（该 Entry 是最早放入该 bucket 中），那HashMap必须循环到最后才能找到该元素。

get()方法源码如下：

[IMG_268](https://www.cnblogs.com/dijia478/p/javascript:void(0);)

public V get(Object key) {

// 若key为null，遍历table[0]处的链表（实际上要么没有元素，要么只有一个Entry对象），取出key为null的value

if (key == null)

return getForNullKey();

// 若key不为null，用key获取Entry对象

Entry<K,V> entry = getEntry(key);

// 若链表中找到的Entry不为null，返回该Entry中的value

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

}

final Entry<K,V> getEntry(Object key) {

if (size == 0) {

return null;

}

// 计算key的hash值

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

// 计算key在数组中对应位置，遍历该位置的链表

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

e != null;

e = e.next) {

Object k;

// 若key完全相同，返回链表中对应的Entry对象

if (e.hash == hash &&

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

return e;

}

// 链表中没找到对应的key，返回null

return null;

}

# **三、hash算法**

我们可以看到在HashMap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。如何计算这个位置就是hash算法。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的分布均匀些，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，而不用再去遍历链表。

源码分析：

/\*\*

\* Returns index for hash code h.

\*/

static int indexFor(int h, int length) {

// assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";

return h & (length-1);

}

# **四、性能问题**

HashMap有两个参数影响其性能：初始容量和负载因子。均可以通过构造方法指定大小。

容量capacity是HashMap中bucket哈希桶(Entry的链表)的数量，初始容量只是HashMap在创建时的容量，最大设置初始容量是2^30，默认初始容量是16（必须为2的幂），解释一下，当数组长度为2的n次幂的时候，不同的key通过indexFor()方法算得的数组位置相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，get()的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

负载因子loadFactor是HashMap在其容量自动增加之前可以达到多满的一种尺度，默认值是0.75。

## **扩容机制：**

当HashMapde的长度超出了加载因子与当前容量的乘积（默认16\*0.75=12）时，通过调用resize方法重新创建一个原来HashMap大小的两倍的newTable数组，最大扩容到2^30+1，并将原先table的元素全部移到newTable里面，重新计算hash，然后再重新根据hash分配位置。这个过程叫作rehash，因为它调用hash方法找到新的bucket位置。

扩容机制源码分析：

[IMG_272](https://www.cnblogs.com/dijia478/p/javascript:void(0);)

void resize(int newCapacity) {

Entry[] oldTable = table;

int oldCapacity = oldTable.length;

// 如果之前的HashMap已经扩充打最大了，那么就将临界值threshold设置为最大的int值

if (oldCapacity == MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return;

}

// 根据新传入的newCapacity创建新Entry数组

Entry[] newTable = new Entry[newCapacity];

// 用来将原先table的元素全部移到newTable里面，重新计算hash，然后再重新根据hash分配位置 transfer(newTable, initHashSeedAsNeeded(newCapacity));

// 再将newTable赋值给table

table = newTable;

// 重新计算临界值，扩容公式在这儿（newCapacity \* loadFactor）

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

}

void transfer(Entry[] newTable, boolean rehash) {

int newCapacity = newTable.length;

for (Entry<K,V> e : table) {

while(null != e) {

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

if (rehash) {

e.hash = null == e.key ? 0 : hash(e.key);

}

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

e.next = newTable[i];

newTable[i] = e;

e = next;

}

}

}

## **扩容问题：**

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，这个操作是极其消耗性能的。所以如果我们已经预知HashMap中元素的个数，那么预设初始容量能够有效的提高HashMap的性能。

重新调整HashMap大小，当多线程的情况下可能产生条件竞争。因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。

# **五、线程安全**

HashMap是线程不安全的，在多线程情况下直接使用HashMap会出现一些莫名其妙不可预知的问题。在多线程下使用HashMap，有几种方案：

1. 在外部包装HashMap，实现同步机制
2. B.使用Map m = Collections.synchronizedMap(new HashMap(...));实现同步（官方参考方案，但不建议使用，使用迭代器遍历的时候修改映射结构容易出错）

D.使用java.util.HashTable，效率最低（几乎被淘汰了）

E.使用java.util.concurrent.ConcurrentHashMap，相对安全，效率高（建议使用）

注意一个小问题，HashMap所有集合类视图所返回迭代器都是快速失败的(fail-fast)，在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器自身的 remove 或 add 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。。因此，面对并发的修改，迭代器很快就会完全失败。

# **六、关于JDK1.8的问题**

JDK1.8的HashMap源码实现和1.7是不一样的，有很大不同，其底层数据结构也不一样，引入了红黑树结构。有网友测试过，JDK1.8HashMap的性能要高于JDK1.7 15%以上，在某些size的区域上，甚至高于100%。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。当一个链表长度大于8的时候，HashMap会动态的将它替换成一个红黑树（JDK1.8引入红黑树大程度优化了HashMap的性能），这会将时间复杂度从O(n)降为O(logn)。