JDK1.8中HashMap的底层实现，底层数据结构 数组 + 链表（或红黑树） ，源码如下：

*/\*\**

*\* 数组*

*\*/*

transient Node<K,V>[] table;

*/\*\*  
 \* 链表结构*

*\*/*

static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 V value;  
 Node<K,V> next;  
  
 Node(int hash, K key, V value, Node<K,V> next) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = next;  
 }  
  
 public final K getKey() { return key; }  
 public final V getValue() { return value; }  
 public final String toString() { return key + "=" + value; }  
  
 public final int hashCode() {  
 return Objects.*hashCode*(key) ^ Objects.*hashCode*(value);  
 }  
  
 public final V setValue(V newValue) {  
 V oldValue = value;  
 value = newValue;  
 return oldValue;  
 }  
  
 public final boolean equals(Object o) {  
 if (o == this)  
 return true;  
 if (o instanceof Map.Entry) {  
 Map.Entry<?,?> e = (Map.Entry<?,?>)o;  
 if (Objects.*equals*(key, e.getKey()) &&  
 Objects.*equals*(value, e.getValue()))  
 return true;  
 }  
 return false;  
 }  
}

*/\*\*  
 \* 红黑树结构*

*\*/*

static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {  
 TreeNode<K,V> parent; // red-black tree links  
 TreeNode<K,V> left;  
 TreeNode<K,V> right;  
 TreeNode<K,V> prev; // needed to unlink next upon deletion  
 boolean red;

…



### 1、table的初始化

HashMap的构造方法有如下4种：

public HashMap(int initialCapacity, float loadFactor) {

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);  
 this.loadFactor = loadFactor;  
 this.threshold = *tableSizeFor*(initialCapacity);  
}  
  
*/\*\*  
 \**

*\*/*public HashMap(int initialCapacity) {  
 this(initialCapacity, *DEFAULT\_LOAD\_FACTOR*);  
}  
  
*/\*\*  
 \*   
 \*/*public HashMap() {  
 this.loadFactor = *DEFAULT\_LOAD\_FACTOR*; // all other fields defaulted  
}  
  
*/\*\*  
 \*   
 \*/*public HashMap(Map<? extends K, ? extends V> m) {  
 this.loadFactor = *DEFAULT\_LOAD\_FACTOR*;  
 putMapEntries(m, false);  
}

使用方式1实例化HashMap时，table并未进行初始化，那table是何时进行初始化的？

既然实例化的时候未进行table的初始化，那是不是在put的时候初始化的。

resize()初始化table或对table进行双倍扩容，源码如下：

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

Node<K,V>[] oldTab = table; // 第一次 put 的时候，table = null

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

int oldThr = threshold; // threshold=0, oldThr = 0

int newCap, newThr = 0;

if (oldCap > 0) { // 条件不满足，往下走

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

}

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

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

}

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

newCap = oldThr;

else { // zero initial threshold signifies using defaults 走到这里，进行默认初始化

newCap = DEFAULT\_INITIAL\_CAPACITY; // DEFAULT\_INITIAL\_CAPACITY = 1 << 4 = 16, newCap = 16;

newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY); // newThr = 0.75 \* 16 = 12;

}

if (newThr == 0) { // 条件不满足

float ft = (float)newCap \* loadFactor;

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

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

}

threshold = newThr; // threshold = 12; 重置阀值为12

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

Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; // 初始化 newTab, length = 16;

table = newTab; // table 初始化完成, length = 16;

if (oldTab != null) { // 此时条件不满足，后续扩容的时候，走此if分支 将数组元素复制到新数组

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

Node<K,V> e;

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

oldTab[j] = null;

if (e.next == null)

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

else if (e instanceof TreeNode)

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

else { // preserve order

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

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

Node<K,V> next;

do {

next = e.next;

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

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

}

else {

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

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

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead;

}

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;

}

}

}

}

}

return newTab; // 新数组

}

* 一般情况下，在第一次put的时候，调用resize方法进行table的初始化。
* 默认情况下，初始化的table.length = 16；
* 默认情况下，threshold = 12；
* 默认情况下，能存放12个元素，当存放第13个元素后进行扩容。

### 2、table扩容

putVal中调用resize()进行扩容，但与初始化时不同：

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

Node<K,V>[] oldTab = table; // 此时的 table != null，oldTab 指向旧的 table

int oldCap = (oldTab == null) ? 0 : oldTab.length; // oldCap = table.length; 第一次扩容时是 16

int oldThr = threshold; // threshold=12, oldThr = 12;

int newCap, newThr = 0;

if (oldCap > 0) { // 条件满足，走此分支

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

}

else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY && // oldCap左移一位; newCap = 16 << 1 = 32;

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold // newThr = 12 << 1 = 24;

}

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

newCap = oldThr;

else { // zero initial threshold signifies using defaults

newCap = DEFAULT\_INITIAL\_CAPACITY; // DEFAULT\_INITIAL\_CAPACITY = 1 << 4 = 16, newCap = 16;

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

}

if (newThr == 0) { // 条件不满足

float ft = (float)newCap \* loadFactor;

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

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

}

threshold = newThr; // threshold = newThr = 24; 重置阀值为 24

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

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

table = newTab; // table 指向 newTab, length = 32;

if (oldTab != null) { // 扩容后，将 oldTab(旧table) 中的元素移到 newTab（新table）中

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

Node<K,V> e;

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

oldTab[j] = null;

if (e.next == null)

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

else if (e instanceof TreeNode)

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

else { // preserve order

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

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

Node<K,V> next;

do {

next = e.next;

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

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

}

else {

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

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

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead;

}

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;

}

}

}

}

}

return newTab;

}

* 当 size > threshold 的时候进行扩容
* 扩容之后的 table.length = 旧 table.length \* 2,
* 扩容之后的 threshold = 旧 threshold \* 2

### 3、table的length为什么都是2的次幂

table是一个数组，那么如何最快的将元素e放入数组？当然是找到元素e在table中对应的位置index，然后table[index] = e;就好了。如何找到e在table中的位置？

我们知道只能通过数组下标（索引）操作数组，而数组的下标类型又是 int ，如果 e 是 int 类型，那好说，就直接用 e 来做数组下标（若 e > table.length，则可以 e % table.length 来获取下标），可 key - value 中的 key 类型不一定，所以我们需要一种统一的方式将 key 转换成 int ，最好是一个 key 对应一个唯一的 int (目前还不可能, int有范围限制，对转换方法要求也极高)，所以引入了 hash 方法

static final int hash(Object key) {

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

>>> 无符号右移，高位补0，这里的处理可以减少哈希碰撞。

实现key到int的转换（关于hash，暂不讨论）。拿到key对应的int h之后，最容易想到的对value的put和get操作也许如下：

// put

table[h % table.length] = value;

// get

e = table[h % table.length];

直接取模是我们最容易想到的获取下标的方法，但是是最高效的方法吗？计算机中的四则运算最终都会转换成二进制的位运算。



只有&树是1时，与运算的结果与被&数一致。

1&1=1;

0&1=0;

这同样适用于多位操作数。

1010&1111=1010; => 10&15=10;

1011&1111=1011; => 11&15=11;

01010&10000=00000; => 10&16=0;

01011&10000=00000; => 11&16=0;

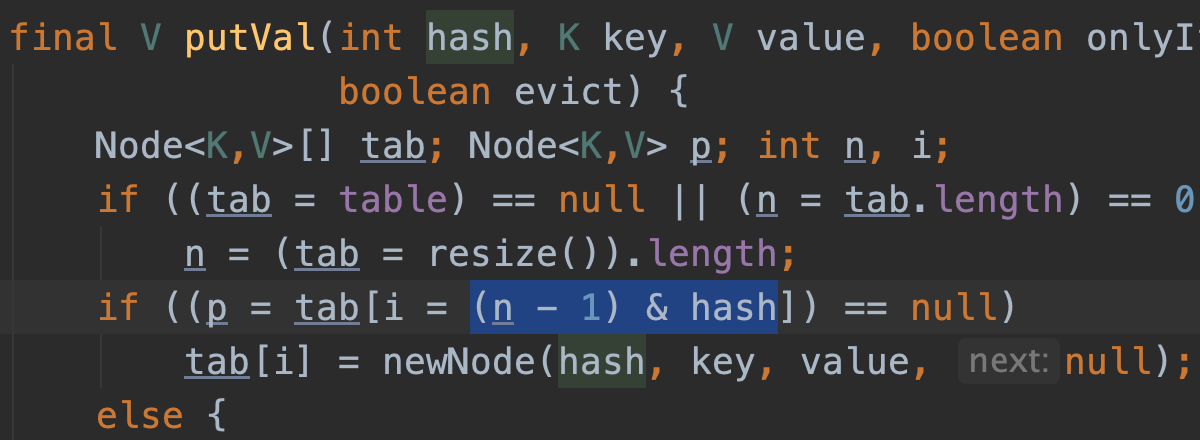
10 & 16 与 11 & 16 得到的结果一样，也就是冲突（碰撞）了，那么 10 和 11 对应的 value 会在同一个链表中，而 table 的有些位置则永远不会有元素，这就导致 table 的空间未得到充分利用，同时还降低了 put 和 get 的效率（对比数组和链表）；由于是 2 个数进行 & 运算，所以结果由这两个数决定，如果我们把这两个数都做下限制，那得到的结果是不是可控制在我们想要的范围内了？

我们需要利用好 & 运算的特点，当右边的数的低位二进制是连续的 1 ，且左边是一个均匀的数（需要 hash 方法实现，尽量保证 key 的 h 唯一），那么得到的结果就比较完美了。低位二进制连续的 1，我们很容易想到 2^n - 1; 而关于左边均匀的数，则通过 hash 方法来实现。

自此，2 的 n 次幂的相关问题就清楚了

table的length是2的次幂是为了利用位运算&求key的下标。

求索引的时候为什么是：h & (length-1)，而不是h & length，更不是h % length。



* h % length效率不如位运算快
* h & length会提高碰撞几率，导致table的空间得不到更充分的利用、降低table的操作效率。

那么为什么不直接用 2^n-1 作为 table.length ？

### 4、指定initialCapacity

调用了HashMap(int initialCapacity, float loadFactor)构造器，其中调用 tableSizeFor 进行 threshold 的初始化。

*/\*\**

*\* Returns a power of two size for the given target capacity.*

*\* 返回 >= cap 最小的 2^n*

*\* cap = 10, 则返回 2^4 = 16;*

*\* cap = 5, 则返回 2^3 = 8;  
 \*/*static final int tableSizeFor(int cap) {  
 int n = cap - 1;  
 n |= n >>> 1;  
 n |= n >>> 2;  
 n |= n >>> 4;  
 n |= n >>> 8;  
 n |= n >>> 16;  
 return (n < 0) ? 1 : (n >= *MAXIMUM\_CAPACITY*) ? *MAXIMUM\_CAPACITY* : n + 1;  
}

虽然此处初始化的是 threshold，但后面初始化 table 的时候，会将其用于 table 的 length，同时会重置 threshold 为 table.length \* loadFactor

自此，问题 5 也就清楚了

Map map = new HashMap(1000); 当我们存入多少个元素时会触发map的扩容：

此时的 table.length = 2^10 = 1024; threshold = 1024 \* 0.75 = 768; 所以存入第 769 个元素时进行扩容

Map map1 = new HashMap(10000); 我们存入第 10001个元素时会触发 map1 扩容吗：

此时的 table.length = 2^14 = 16384; threshold = 16384 \* 0.75 = 12288; 所以存入第 10001 个元素时不会进行扩容

### 5、加载因子

为什么加载因子的默认值是 0.75，并且不推荐我们修改

* 如果loadFactor太小，那么map中的table需要不断的扩容，扩容是个耗时的过程；
* 如果loadFactor太大，那么map中table放满了也不不会扩容，导致冲突越来越多，解决冲突而起的链表越来越长，效率越来越低；
* 而 0.75 这是一个折中的值，是一个比较理想的值

### 6、总结

1. table.length = 2^n，是为了能利用位运算（&）来求 key 的下标，而 h&(length-1) 是为了充分利用 table 的空间，并减少 key 的碰撞
2. 加载因子太小， table 需要不断的扩容，影响 put 效率；太大会导致碰撞越来越多，链表越来越长（转红黑树），影响效率；0.75 是一个比较理想的中间值
3. table.length = 2^n、hash 方法获取 key 的 h、加载因子 0.75、数组 + 链表（或红黑树），一环扣一环，保证了 key 在 table 中的均匀分配，充分利用了空间，也保证了操作效率，环环相扣的，而不是心血来潮的随意处理；缺了一环，其他的环就无意义了！
4. put方法的流程图如下：

