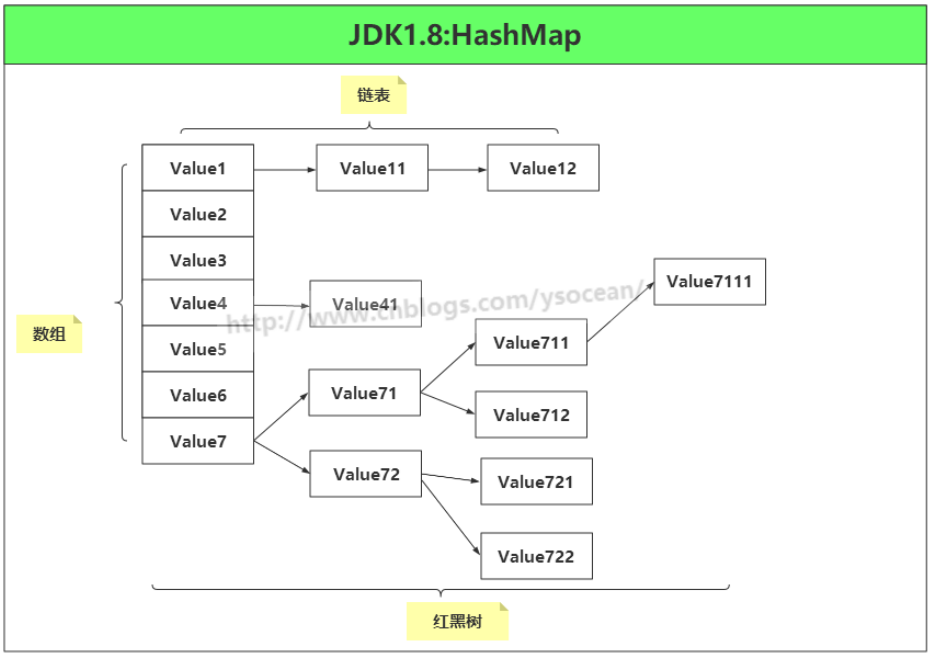
简述

HashMap 是一个利用哈希表原理来存储元素的集合。遇到冲突时，HashMap 是采用的链地址法来解决，在 JDK1.7 中，HashMap 是由 中，HashMap 是由 数组+链表+红黑树构成，新增了红黑树作为底层数据结构，结构变得复杂了，但是效率也变的更高效



HashMap定义

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

小插曲

HashMap继承了 AbstractMap，并且实现了Map接口。 AbstractMap也实现了Map接口

```
据 java 集合框架的创始人Josh Bloch描述，这样的写法是一个失误。在java集合框架中，类似这样的写法很多，最开始写java集合框架的时候，他认为这样写，在某些地方可能是有价值的，直到他意识到错了。显然的，JDK的维护者，后来不认为这个小小的失误值得去修改，所以就这样存在下来了
```

字段属性

```
//默认 HashMap 集合初始容量为16（必须是 2 的倍数）
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

//集合的最大容量，如果通过带参构造指定的最大容量超过此数，默认还是使用此数
static final int MAXIMUM_CAPACITY = 1 << 30;

//默认的填充因子（负载因子、加载因子） = 元素个数 / 散列表大小（数组的长度）
//会根据这个值来动态扩容
static final float DEFAULT_LOAD_FACTOR = 0.75f;

//当桶(bucket)上的结点数大于这个值时会转成红黑树(JDK1.8新增)
static final int TREEIFY_THRESHOLD = 8;

//当桶(bucket)上的节点数小于这个值时会转成链表(JDK1.8新增)
static final int UNTREEIFY_THRESHOLD = 6;

/**(JDK1.8新增)
 * 当集合中的容量大于这个值时，表中的桶才能进行树形化 ，否则桶内元素太多时会扩容，
 * 而不是树形化 为了避免进行扩容、树形化选择的冲突，这个值不能小于 4 * TREEIFY_THRESHOLD
 */
static final int MIN_TREEIFY_CAPACITY = 64;
```

```
/**
* 初始化使用，长度总是 2的幂（在确定哈希数组索引位置的时候通过这种方式做了优化，计算hash值更快）

* HashMap 是由数组+链表+红黑树组成，这里的数组就是 table 字段。后面对其进行初始化长度默认是
* DEFAULT_INITIAL_CAPACITY= 16。而且 JDK 声明数组的长度总是 2的n次方
*/
transient Node<K,V>[] table;

/**
 * 保存缓存的entrySet ()
 */
transient Set<Map.Entry<K,V>> entrySet;

/**
* 此映射中包含的键值映射的数量。（集合存储键值对的数量）
*/
transient int size;

/**
* 记录集合被修改的次数,文章末尾扩展详解
*/
transient int modCount;
```

```
23
24 /**
25 * 调整大小的下一个大小值（容量*加载因子）。capacity * loadfactor
26 */
27 int threshold;
28
29 /**
30 * 装载因子，是用来衡量 HashMap 满的程度，计算HashMap的实时装载因子的方法为：size/capacity
31 */
32 final float loadFactor;
```

构造函数

默认无参构造

```
1 public HashMap() {
2         this.loadFactor = DEFAULT_LOAD_FACTOR;
3 }
```

指定初始容量的构造函数

```
1 public HashMap(int initialCapacity, float loadFactor) {
2     .......
3 }
```

确定哈希索引位置

```
1 static final int hash(Object key) {
2     int h;
3     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
4 }
5
6 // i = (table.length - 1) & hash;//这一步是在后面添加元素putVal()方法中进行位置的确定
```

主要分为三步：

①、取 hashCode 值： key.hashCode()

②、高位参与运算：h>>>16

③、取模运算：(n-1) & hash

这里获取 hashCode() 方法的值是变量，但是我们知道，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 hash 总是相同的。

为了让数组元素分布均匀，我们首先想到的是把获得的 hash码对数组长度取模运算(hash%length)，但是计算机都是二进制进行操作，如何优化呢?

HashMap 使用的方法很巧妙，它通过 hash & (table.length -1)来得到该对象的保存位，前面说过 HashMap 底层数组的长度总是2的n次当 length 总是2的n次方时，hash & (length-1)运算等价于对 length 取模，也就是 hash%length，但是&比%具有更高的效率。比如 n % 32

这也解释了为什么要保证数组的长度总是2的n次方

添加元素

①、判断键值对数组 table 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
value即可；

⑥、插入成功后，判断实际存在的键值对数量size是否超过了最大容量threshold，如果超过，进行扩容。

⑦、如果新插入的key不存在，则返回null，如果新插入的key存在，则返回原key对应的value值（注意新插入的value会覆盖原value值）

扩容机制

扩容（resize），我们知道集合是由数组+链表+红黑树构成，向 HashMap 中插入元素时，如果HashMap 集合的元素已经大于了最大承
loadFactor），这里的threshold不是数组的最大长度。那么必须扩大数组的长度，Java中数组是无法自动扩容的，我们采用的方法是用一个
以前是用小桶装水，现在小桶装不下了，我们使用一个更大的桶

相比于JDK1.7，1.8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
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

扩展

modCount详解

在ArrayList,LinkedList,HashMap等等的内部实现增，删，改中总能看到modCount的身影，modCount字面意思就是修改次数，但为什

有一个公共特点，所有使用modCount属性的全是线程不安全的，这是为什么呢?

以hashMap为例

```
final Node<K,V> nextNode() {
    Node<K,V>[] t;
    Node<K,V> e = next;
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
    if (e == null)
        throw new NoSuchElementException();
    if ((next = (current = e).next) == null && (t = table) != null) {
        do {} while (index < t.length && (next = t[index++]) == null);
     }
     return e;
}
```

迭代遍历的时候，会初始化expectedModCount=modCount，这时候对HashMap进行修改操作，modCount会+1，继续遍历的时候e
继而抛出java.util.ConcurrentModificationException异常

总结

无序
key和value都可以为null
value可以重复
非线程安全