HashMap源码分析

笔记本: DSA
创建时间: 2020/9/6 22:10 **更新时间:** 2020/9/6 23:42
作者: Jie Zhong
URL: http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classe...

源码:
http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java
文
档: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html

java.util.HashMap 基于散列表, 实现了接口 java.util.Map, 它存储是元素是键值对<kay, value>映射 .

- key和value都可以是null
- 不可排序
- 非同步
- 默认负载因子为0.75

常用方法

返回值	方法	实现
void	clear()	
boolean	containsKey()	
boolean	containsValue()	
Set<Map.Entry<K, V>>	entrySet()	

V	get(Object key)	public V get(Object key) {
		Node<K,V> e;
		return (e = getNode(hash(key), key)) == null ? null : e.value;
		}

V	put(K key, V value)	public V put(K key, V value) {
		return putVal(hash(key), key, value, false, true);
		}

boolean	isEmpty()	
int	size()	
Set<K>	keySet()	
Collection<V>	values()	
V	remove(Object key)	

V	getOrDefault(Object key, V defaultValue)	

散列表中的元素有两种Node, TreeNode

重要方法分析

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

```
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    // table已经初始化，长度大于0，根据hash寻找table中的项也不为空
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        // 桶中第一项(数组元素)相等
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        // 桶中不止一个结点
        if ((e = first.next) != null) {
            // 为红黑树结点
            if (first instanceof TreeNode)
                // 在红黑树中查找
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            // 否则，在链表中查找
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

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

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // table未初始化或者长度为0，进行扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // (n - 1) & hash 确定元素存放在哪个桶中，桶为空，新生成结点放入桶中(此时，这个
结点是放在数组中)
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // 桶中已经存在元素
    else {
        Node<K,V> e; K k;
        // 比较桶中第一个元素(数组中的结点)的hash值相等，key相等
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
                // 将第一个元素赋值给e，用e来记录
                e = p;
        // hash值不相等，即key不相等；为红黑树结点
        else if (p instanceof TreeNode)
            // 放入树中
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 为链表结点
        else {
            // 在链表最末插入结点
            for (int binCount = 0; ; ++binCount) {
                // 到达链表的尾部
                if ((e = p.next) == null) {
                    // 在尾部插入新结点
                    p.next = newNode(hash, key, value, null);
                    // 结点数量达到阈值，转化为红黑树
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    // 跳出循环
                    break;
```

```
                }
                // 判断链表中结点的key值与插入的元素的key值是否相等
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    // 相等，跳出循环
                    break;
                // 用于遍历桶中的链表，与前面的e = p.next组合，可以遍历链表
                p = e;
            }
        }
        // 表示在桶中找到key值、hash值与插入元素相等的结点
        if (e != null) {
            // 记录e的value
            V oldValue = e.value;
            // onlyIfAbsent为false或者旧值为null
            if (!onlyIfAbsent || oldValue == null)
                //用新值替换旧值
                e.value = value;
            // 访问后回调
            afterNodeAccess(e);
            // 返回旧值
            return oldValue;
        }
    }
    // 结构性修改
    ++modCount;
    // 实际大小大于阈值则扩容
    if (++size > threshold)
        resize();
    // 插入后回调
    afterNodeInsertion(evict);
    return null;
}
```