HashTable 源码解析 | MrBird

> HashTable 是 Map 接口线程安全实现版本，数据结构和方法实现与 HashMap 类似，本文记录 HashTable 源码解析，基于 JDK1.8。

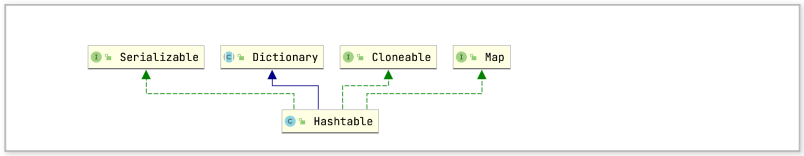
HashTable 源码解析

2020-08-27 | Visit count 1058057

HashTable 是 Map 接口线程安全实现版本，数据结构和方法实现与 HashMap 类似，本文记录 HashTable 源码解析，基于 JDK1.8。

类结构

HashTable 类层级关系图：



主要成员变量：

```
// 内部采用Entry数组存储键值对数据，Entry实际为单向链表的表
private transient Entry<?,?>[] table;
// HashTable里键值对个数
private transient int count;
// 扩容阈值，当超过这个值时，进行扩容操作，计算方式为：数组
private int threshold;
// 加载因子
private float loadFactor;
// 用于快速失败
private transient int modCount = 0;
```

table 属性通过 transient 修饰，原因在介绍 HashMap 源码 的时候分析过。

Entry 代码如下：

```
private static class Entry<K,V> implements Map.Entry<K,
    final int hash;
    final K key;
    V value;
    Entry<K,V> next;

    protected Entry(int hash, K key, V value, Entry<K,V
        this.hash = hash;
        this.key =  key;
        this.value = value;
        this.next = next;
    }

    ......
}
```

Entry 为单向链表节点，HashTable 采用数组加链表的方式存储数据，不过没有类似于 HashMap 中当链表过长时转换为红黑树的操作。

方法解析

构造函数

```
// 设置指定容量和加载因子，初始化HashTable
public Hashtable(int initialCapacity, float loadFactor)
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Cap
                                           initialCapac
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal Loa

    if (initialCapacity==0)
        // 容量最小为1
        initialCapacity = 1;
    this.loadFactor = loadFactor;
    table = new Entry<?,?>[initialCapacity];
    // 初始扩容阈值
    threshold = (int)Math.min(initialCapacity * loadFac
}

// 设置指定容量初始HashTable，加载因子为0.75
public Hashtable(int initialCapacity) {
    this(initialCapacity, 0.75f);
}
```

```
// 手动指定数组初始容量为11，加载因子为0.75
public Hashtable() {
    this(11, 0.75f);
}
```

put(K key, V value)

put(K key, V value) 添加指定键值对，键和值都不能为 null：

```
// 方法synchronized修饰，线程安全
public synchronized V put(K key, V value) {
    // Make sure the value is not null
    if (value == null) {
        throw new NullPointerException();
    }

    // Makes sure the key is not already in the hashtal
    Entry<?,?> tab[] = table;
    // 得到key的哈希值
    int hash = key.hashCode();
    // 得到该key存在到数组中的下标
    int index = (hash & 0x7FFFFFFF) % tab.length;
    @SuppressWarnings("unchecked")
    // 得到该下标对应的Entry
    Entry<K,V> entry = (Entry<K,V>)tab[index];
    // 如果该下标的Entry不为null，则进行链表遍历
    for(; entry != null ; entry = entry.next) {
        // 遍历链表，如果存在key相等的节点，则替换这个节点
        if ((entry.hash == hash) && entry.key.equals(k
            V old = entry.value;
            entry.value = value;
            return old;
        }
    }
    // 如果数组下标对应的节点为空，或者遍历链表后发现没有和
    addEntry(hash, key, value, index);
    return null;
}

private void addEntry(int hash, K key, V value, int in
    // 模数+1
    modCount++;

    Entry<?,?> tab[] = table;
    // 判断是否需要扩容
    if (count >= threshold) {
        // 如果count大于等于扩容阈值，则进行扩容
        rehash();

        tab = table;
        // 扩容后，重新计算该key在扩容后table里的下标
        hash = key.hashCode();
        index = (hash & 0x7FFFFFFF) % tab.length;
    }
```

```
    // Creates the new entry.
    @SuppressWarnings("unchecked")
    // 采用头插的方式插入，index位置的节点为新节点的next节
    // 新节点取代inde位置节点
    Entry<K,V> e = (Entry<K,V>) tab[index];
    tab[index] = new Entry<>(hash, key, value, e);
    // count+1
    count++;
}
```

rehash()

rehash 扩容操作：

```
protected void rehash() {
    // 暂存旧的table和容量
    int oldCapacity = table.length;
    Entry<?,?>[] oldMap = table;

    // 新容量为旧容量的2n+1倍
    int newCapacity = (oldCapacity << 1) + 1;
    // 判断新容量是否超过最大容量
    if (newCapacity - MAX_ARRAY_SIZE > 0) {
        // 如果旧容量已经是最大容量大话，就不扩容了
        if (oldCapacity == MAX_ARRAY_SIZE)
            // Keep running with MAX_ARRAY_SIZE buckets
            return;
        // 新容量最大值只能是MAX_ARRAY_SIZE
        newCapacity = MAX_ARRAY_SIZE;
    }
    // 用新容量创建一个新Entry数组
    Entry<?,?>[] newMap = new Entry<?,?>[newCapacity];
    // 模数+1
    modCount++;
    // 重新计算下次扩容阈值
    threshold = (int)Math.min(newCapacity * loadFactor
    // 将新Entry数组赋值给table
    table = newMap;
    // 遍历数组和链表，进行新table赋值操作
    for (int i = oldCapacity ; i-- > 0 ;) {
        for (Entry<K,V> old = (Entry<K,V>)oldMap[i] ; 
            Entry<K,V> e = old;
            old = old.next;

            int index = (e.hash & 0x7FFFFFFF) % newCap
            e.next = (Entry<K,V>)newMap[index];
            newMap[index] = e;
        }
    }
}
```

get(Object key)

get(Object key) 获取指定 key 对应的 value：

```
public synchronized V get(Object key) {
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
    // 根据key哈希得到index，遍历链表取值
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<?,?> e = tab[index] ; e != null ; e = e
        if ((e.hash == hash) && e.key.equals(key)) {
            return (V)e.value;
        }
    }
    return null;
}
```

synchronized 修饰，线程安全。

remove(Object key)

remove(Object key) 删除指定 key，返回对应的 value：

```
public synchronized V remove(Object key) {
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
    // 获取key对应的index
    int index = (hash & 0x7FFFFFFF) % tab.length;
    @SuppressWarnings("unchecked")
    // 遍历链表，如果找到key相等的节点，则改变前继和后继节点
    Entry<K,V> e = (Entry<K,V>)tab[index];
    for(Entry<K,V> prev = null ; e != null ; prev = e,
        if ((e.hash == hash) && e.key.equals(key)) {
            modCount++;
            if (prev != null) {
                prev.next = e.next;
            } else {
                tab[index] = e.next;
            }
            count--;
            V oldValue = e.value;
            e.value = null;
            return oldValue;
        }
    }
    return null;
}
```

synchronized 修饰，线程安全。

> 剩下方法有兴趣自己阅读源码，public 方法都用 synchronized 修饰，确保线程安全，并发环境下，多线

> 程竞争对象锁，效率低，不推荐使用。线程安全的 Map 推荐使用 ConcurrentHashMap。

和 HashMap 对比

1. 线程是否安全：HashMap 是线程不安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 synchronized 修饰；

2. 对 Null key 和 Null value 的支持：HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null；HashTable 中 key 和 value 都不能为 null，否则抛出空指针异常；

3. 初始容量大小和每次扩充容量大小的不同：
   3.1. 创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩容，容量变为原来的 2n+1。HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍；
   3.2. 创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充 为 2 的幂次方大小。

4. 底层数据结构：JDK1.8 及以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间，Hashtable 没有这样的机制。

全文完