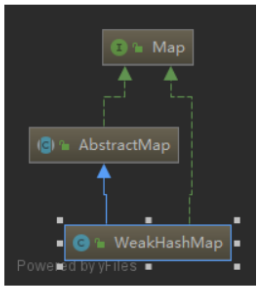
简介

WeakHashMap是一种弱引用map，内部的key会存储为弱引用，当jvm gc的时候，如果这些key没有强引用存在的话，会被gc回收掉，下一次当我们操作map的时候会把对应的Entry整个删除掉，基于这种特性，WeakHashMap特别适用于缓存处理。

继承体系



可见，WeakHashMap没有实现Clone和Serializable接口，所以不具有克隆和序列化的特性。

存储结构

WeakHashMap因为gc的时候会把没有强引用的key回收掉，所以注定了它里面的元素不会太多，因此也就不需要像HashMap那样元素多的时候转化为红黑树来处理了。

因此，WeakHashMap的存储结构只有（数组 + 链表）。

源码解析

属性

```
1.     /**
2.      * 默认初始容量为16
3.      */
4.     private static final int DEFAULT_INITIAL_CAPACITY = 16;
5.
6.     /**
7.      * 最大容量为2的30次方
8.      */
9.     private static final int MAXIMUM_CAPACITY = 1 << 30;
10.
11.    /**
12.     * 默认装载因子
13.     */
14.    private static final float DEFAULT_LOAD_FACTOR = 0.75f;
15.
16.    /**
17.     * 桶
18.     */
19.    Entry<K,V>[] table;
20.
```

```
    /**
     * 元素个数
     */
    private int size;

    /**
     * 扩容门槛，等于capacity * loadFactor
     */
    private int threshold;

    /**
     * 装载因子
     */
    private final float loadFactor;

    /**
     * 引用队列，当弱键失效的时候会把Entry添加到这个队列中
     */
    private final ReferenceQueue<Object> queue = new ReferenceQueue<>();

```

（1）容量

容量为数组的长度，亦即桶的个数，默认为16，最大为2的30次方，当容量达到64时才可以树化。

（2）装载因子

装载因子用来计算容量达到多少时才进行扩容，默认装载因子为0.75。

（3）引用队列

当弱键失效的时候会把Entry添加到这个队列中，当下次访问map的时候会把失效的Entry清除掉。

Entry内部类

WeakHashMap内部的存储节点，没有key属性。

```
    private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> {
        // 可以发现没有key，因为key是作为弱引用存到Referen类中
        V value;
        final int hash;
        Entry<K,V> next;

        Entry(Object key, V value,
              ReferenceQueue<Object> queue,
              int hash, Entry<K,V> next) {
            // 调用WeakReference的构造方法初始化key和引用队列
            super(key, queue);
            this.value = value;
            this.hash  = hash;
            this.next  = next;
        }
    }

    public class WeakReference<T> extends Reference<T> {
```

```
        public WeakReference(T referent, ReferenceQueue<? super T> q) {
            // 调用Reference的构造方法初始化key和引用队列
            super(referent, q);
        }
    }

    public abstract class Reference<T> {
        // 实际存储key的地方
        private T referent;         /* Treated specially by GC */
        // 引用队列
        volatile ReferenceQueue<? super T> queue;

        Reference(T referent, ReferenceQueue<? super T> queue) {
            this.referent = referent;
            this.queue = (queue == null) ? ReferenceQueue.NULL : queue;
        }
    }

```

从Entry的构造方法我们知道，key和queue最终会传到到Reference的构造方法中，这里的key就是Reference的referent属性，它会被gc特殊对待，即当没有强引用存在时，当下一次gc的时候会被清除。

构造方法

```
    public WeakHashMap(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);
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;
        table = newTable(capacity);
        this.loadFactor = loadFactor;
        threshold = (int)(capacity * loadFactor);
    }

    public WeakHashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }

    public WeakHashMap() {
        this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
    }

    public WeakHashMap(Map<? extends K, ? extends V> m) {
        this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1,
                DEFAULT_INITIAL_CAPACITY),
                DEFAULT_LOAD_FACTOR);
```

```
        putAll(m);
    }

```

构造方法与HashMap基本类似，初始容量为大于等于传入容量最近的2的n次方，扩容门槛threshold等于capacity * loadFactor。

put(K key, V value)方法

添加元素的方法。

```
    public V put(K key, V value) {
        // 如果key为空，用空对象代替
        Object k = maskNull(key);
        // 计算key的hash值
        int h = hash(k);
        // 获取桶
        Entry<K,V>[] tab = getTable();
        // 计算元素在哪个桶中，h & (length-1)
        int i = indexFor(h, tab.length);

        // 遍历桶对应的链表
        for (Entry<K,V> e = tab[i]; e != null; e = e.next) {
            if (h == e.hash && eq(k, e.get())) {
                // 如果找到了元素就使用新值替换旧值，并返回旧值
                V oldValue = e.value;
                if (value != oldValue)
                    e.value = value;
                return oldValue;
            }
        }

        modCount++;
        // 如果没找到就把新值插入到链表的头部
        Entry<K,V> e = tab[i];
        tab[i] = new Entry<>(k, value, queue, h, e);
        // 如果插入元素后数量达到了扩容门槛就把桶的数量扩容为2倍大小
        if (++size >= threshold)
            resize(tab.length * 2);
        return null;
    }

```

（1）计算hash；

这里与HashMap有所不同，HashMap中如果key为空直接返回0，这里是用空对象来计算的。

另外打散方式也不同，HashMap只用了一次异或，这里用了四次，HashMap给出的解释是一次够了，而且就算冲突了也会转换成红黑树，对效率没什么影响。

（2）计算在哪个桶中；

（3）遍历桶对应的链表；

（4）如果找到元素就用新值替换旧值，并返回旧值；

（5）如果没找到就在链表头部插入新元素；

HashMap就插入到链表尾部。

（6）如果元素数量达到了扩容门槛，就把容量扩大到2倍大小；

HashMap中是大于threshold才扩容，这里等于threshold就开始扩容了。

resize(int newCapacity)方法

扩容方法。

```
    void resize(int newCapacity) {
        // 获取旧桶，getTable()的时候会剔除失效的Entry
        Entry<K,V>[] oldTable = getTable();
        // 旧容量
        int oldCapacity = oldTable.length;
        if (oldCapacity == MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return;
        }

        // 新桶
        Entry<K,V>[] newTable = newTable(newCapacity);
        // 把元素从旧桶转移到新桶
        transfer(oldTable, newTable);
        // 把新桶赋值桶变量
        table = newTable;

        /*
         * If ignoring null elements and processing ref queue caused massive
         * shrinkage, then restore old table.  This should be rare, but avoids
         * unbounded expansion of garbage-filled tables.
         */
        // 如果元素个数大于扩容门槛的一半，则使用新桶和新容量，并计算新的扩容门槛
        if (size >= threshold / 2) {
            threshold = (int)(newCapacity * loadFactor);
        } else {
            // 否则把元素再转移回旧桶，还是使用旧桶
            // 因为在transfer的时候会清除失效的Entry，所以元素个数可能没有那么大了，就不需要扩容了
            expungeStaleEntries();
            transfer(newTable, oldTable);
            table = oldTable;
        }
    }

    private void transfer(Entry<K,V>[] src, Entry<K,V>[] dest) {
        // 遍历旧桶
        for (int j = 0; j < src.length; ++j) {
            Entry<K,V> e = src[j];
            src[j] = null;
            while (e != null) {
                Entry<K,V> next = e.next;
                Object key = e.get();
```

```
                // 如果key等于了null就清除，说明key被gc清理掉了，则把整个Entry清除
                if (key == null) {
                    e.next = null;  // Help GC
                    e.value = null; //  "    "
                    size--;
                } else {
                    // 否则就计算在新桶中的位置并把这个元素放在新桶对应链表的头部
                    int i = indexFor(e.hash, dest.length);
                    e.next = dest[i];
                    dest[i] = e;
                }
                e = next;
            }
        }
    }

```

（1）判断旧容量是否达到最大容量；

（2）新建新桶并把元素全部转移到新桶中；

（3）如果转移后元素个数不到扩容门槛的一半，则把元素再转移回旧桶，继续使用旧桶，说明不需要扩容；

（4）否则使用新桶，并计算新的扩容门槛；

（5）转移元素的过程中会把key为null的元素清除掉，所以size会变小；

get(Object key)方法

获取元素。

```
    public V get(Object key) {
        Object k = maskNull(key);
        // 计算hash
        int h = hash(k);
        Entry<K,V>[] tab = getTable();
        int index = indexFor(h, tab.length);
        Entry<K,V> e = tab[index];
        // 遍历链表，找到了就返回
        while (e != null) {
            if (e.hash == h && eq(k, e.get()))
                return e.value;
            e = e.next;
        }
        return null;
    }

```

（1）计算hash值；

（2）遍历所在桶对应的链表；

（3）如果找到了就返回元素的value值；

（4）如果没找到就返回空；

remove(Object key)方法

移除元素。

```
    public V remove(Object key) {
        Object k = maskNull(key);
        // 计算hash
        int h = hash(k);
        Entry<K,V>[] tab = getTable();
        int i = indexFor(h, tab.length);
        // 元素所在的桶的第一个元素
        Entry<K,V> prev = tab[i];
        Entry<K,V> e = prev;

        // 遍历链表
        while (e != null) {
            Entry<K,V> next = e.next;
            if (h == e.hash && eq(k, e.get())) {
                // 如果找到了就删除元素
                modCount++;
                size--;

                if (prev == e)
                    // 如果是头节点，就把头节点指向下一个节点
                    tab[i] = next;
                else
                    // 如果不是头节点，删除该节点
                    prev.next = next;
                return e.value;
            }
            prev = e;
            e = next;
        }

        return null;
    }

```

（1）计算hash；

（2）找到所在的桶；

（3）遍历桶对应的链表；

（4）如果找到了就删除该节点，并返回该节点的value值；

（5）如果没找到就返回null；

expungeStaleEntries()方法

剔除失效的Entry。

```
    private void expungeStaleEntries() {
        // 遍历引用队列
        for (Object x; (x = queue.poll()) != null; ) {
            synchronized (queue) {
                @SuppressWarnings("unchecked")
                Entry<K,V> e = (Entry<K,V>) x;
                int i = indexFor(e.hash, table.length);
                // 找到所在的桶
                Entry<K,V> prev = table[i];
                Entry<K,V> p = prev;
                // 遍历链表
                while (p != null) {
                    Entry<K,V> next = p.next;
                    // 找到该元素
                    if (p == e) {
                        // 删除该元素
                        if (prev == e)
                            table[i] = next;
                        else
                            prev.next = next;
                        // Must not null out e.next;
                        // stale entries may be in use by a HashIterator
                        e.value = null; // Help GC
                        size--;
                        break;
                    }
                    prev = p;
                    p = next;
                }
            }
        }
    }

```

（1）当key失效的时候gc会自动把对应的Entry添加到这个引用队列中；

（2）所有对map的操作都会直接或间接地调用到这个方法先移除失效的Entry，比如getTable()、size()、resize()；

（3）这个方法的目的就是遍历引用队列，并把其中保存的Entry从map中移除掉，具体的过程请看类注释；

（4）从这里可以看到移除Entry的同时把value也一并置为null帮助gc清理元素，防御性编程。

使用案例

说了这么多，不举个使用的例子怎么过得去。

```
    package com.coolcoding.code;

    import java.util.Map;
    import java.util.WeakHashMap;

    public class WeakHashMapTest {

    public static void main(String[] args) {
        Map<String, Integer> map = new WeakHashMap<>(3);
```

```
10.
11.         // 放入3个new String()声明的字符串
12.         map.put(new String("1"), 1);
13.         map.put(new String("2"), 2);
14.         map.put(new String("3"), 3);
15.
16.         // 放入不用new String()声明的字符串
17.         map.put("6", 6);
18.
19.         // 使用key强引用"3"这个字符串
20.         String key = null;
21.         for (String s : map.keySet()) {
22.             // 这个"3"和new String("3")不是一个引用
23.             if (s.equals("3")) {
24.                 key = s;
25.             }
26.         }
27.
28.         // 输出{6=6, 1=1, 2=2, 3=3}，未gc所有key都可以打印出来
29.         System.out.println(map);
30.
31.         // gc一下
32.         System.gc();
33.
34.         // 放一个new String()声明的字符串
35.         map.put(new String("4"), 4);
36.
37.         // 输出{4=4, 6=6, 3=3}，gc后放入的值和强引用的key可以打印出来
38.         System.out.println(map);
39.
40.         // key与"3"的引用断裂
41.         key = null;
42.
43.         // gc一下
44.         System.gc();
45.
46.         // 输出{6=6}，gc后强引用的key可以打印出来
47.         System.out.println(map);
48.     }
49.     }
50.
51.
```

在这里通过new String()声明的变量才是弱引用，使用"6"这种声明方式会一直存在于常量池中，不会被清理，所以"6"这个元素会一直在map里面，其它的元素随着gc都会被清理掉。

总结

（1）WeakHashMap使用（数组 + 链表）存储结构；

（2）WeakHashMap中的key是弱引用，gc的时候会被清除；

（3）每次对map的操作都会剔除失效key对应的Entry；

（4）使用String作为key时，一定要使用new String()这样的方式声明key，才会失效，其它的基本类型的包装类型是一样的；

（5）WeakHashMap常用来作为缓存使用；

带详细注释的源码地址

WeakHashMap.java

彩蛋

强、软、弱、虚引用知多少？

（1）强引用

使用最普遍的引用。如果一个对象具有强引用，它绝对不会被gc回收。如果内存空间不足了，gc宁愿抛出OutOfMemoryError，也不是会回收具有强引用的对象。

（2）软引用

如果一个对象只具有软引用，则内存空间足够时不会回收它，但内存空间不够时就会回收这部分对象。只要这个具有软引用对象没有被回收，程序就可以正常使用。

（3）弱引用

如果一个对象只具有弱引用，则不管内存空间够不够，当gc扫描到它时就会回收它。

（4）虚引用

如果一个对象只具有虚引用，那么它就和没有任何引用一样，任何时候都可能被gc回收。

软（弱、虚）引用必须和一个引用队列（ReferenceQueue）一起使用，当gc回收这个软（弱、虚）引用引用的对象时，会把这个软（弱、虚）引用放到这个引用队列中。

比如，上述的Entry是一个弱引用，它引用的对象是key，当key被回收时，Entry会被放到queue中。