WeakHashMap

总体介绍

在Java集合框架系列文章的最后，笔者打算介绍一个特殊的成员：*WeakHashMap*，从名字可以看出它是某种 *Map*。它的特殊之处在于 *WeakHashMap* 里的`entry`可能会被GC自动删除，即使程序员没有调用`remove()`或者`clear()`方法。

更直观的说，当使用 *WeakHashMap* 时，即使没有显示的添加或删除任何元素，也可能发生如下情况：

> - 调用两次`size()`方法返回不同的值；
> - 两次调用`isEmpty()`方法，第一次返回`false`，第二次返回`true`；
> - 两次调用`containsKey()`方法，第一次返回`true`，第二次返回`false`，尽管两次使用的是同一个`key`；
> - 两次调用`get()`方法，第一次返回一个`value`，第二次返回`null`，尽管两次使用的是同一个对象。

遇到这么奇葩的现象，你是不是觉得使用者一定会疯掉？其实不然，***WeekHashMap* 的这个特点特别适用于需要缓存的场景**。在缓存场景下，由于内存是有限的，不能缓存所有对象；对象缓存命中可以提高系统效率，但缓存MISS也不会造成错误，因为可以通过计算重新得到。

要明白 *WeekHashMap* 的工作原理，还需要引入一个概念：**弱引用（WeakReference）**。我们都知道Java中内存是通过GC自动管理的，GC会在程序运行过程中自动判断哪些对象是可以被回收的，并在合适的时机进行内存释放。GC判断某个对象是否可被回收的依据是，**是否有有效的引用指向该对象**。如果没有有效引用指向该对象（基本意味着不存在访问该对象的方式），那么该对象就是可回收的。这里的**“有效引用”**并不包括**弱引用**。也就是说，**虽然弱引用可以用来访问对象，但进行垃圾回收时弱引用并不会被考虑在内，仅有弱引用指向的对象仍然会被GC回收**。

WeakHashMap 内部是通过弱引用来管理`entry`的，弱引用的特性对应到 *WeakHashMap* 上意味着什么呢？**将一对`key, value`放入到 *WeakHashMap* 里并不能避免该`key`值被GC回收，除非在 *WeakHashMap* 之外还有对该`key`的强引用**。

关于强引用，弱引用等概念以后再具体讲解，这里只需要知道Java中引用也是分种类的，并且不同种类的引用对GC的影响不同就够了。

具体实现

WeakHashMap的存储结构类似于HashMap，读者可自行参考前文，这里不再赘述。

关于强弱引用的管理方式，博主将会另开专题单独讲解。

Weak HashSet?

如果你看过前几篇关于 *Map* 和 *Set* 的讲解，一定会问：既然有 *WeekHashMap*，是否有 *WeekHashSet* 呢？答案是没有:(。不过Java *Collections*工具类给出了解决方案，`Collections.newSetFromMap(Map<E,Boolean> map)`方法可以将任何 *Map*包装成一个*Set*。通过如下方式可以快速得到一个 *Weak HashSet*：

```
// 将WeakHashMap包装成一个Set
Set<Object> weakHashSet = Collections.newSetFromMap(
        new WeakHashMap<Object, Boolean>());
```

不出你所料，`newSetFromMap()`方法只是对传入的 *Map*做了简单包装：

```
// Collections.newSetFromMap()用于将任何Map包装成一个Set
public static <E> Set<E> newSetFromMap(Map<E, Boolean> map) {
    return new SetFromMap<>(map);
}

private static class SetFromMap<E> extends AbstractSet<E>
    implements Set<E>, Serializable
{
    private final Map<E, Boolean> m;  // The backing map
    private transient Set<E> s;       // Its keySet
    SetFromMap(Map<E, Boolean> map) {
        if (!map.isEmpty())
            throw new IllegalArgumentException("Map is non-empty");
        m = map;
        s = map.keySet();
    }
    public void clear()               {        m.clear(); }
    public int size()                 { return m.size(); }
    public boolean isEmpty()          { return m.isEmpty(); }
    public boolean contains(Object o) { return m.containsKey(o); }
    public boolean remove(Object o)   { return m.remove(o) != null; }
    public boolean add(E e) { return m.put(e, Boolean.TRUE) == null; }
    public Iterator<E> iterator()     { return s.iterator(); }
    public Object[] toArray()         { return s.toArray(); }
    public <T> T[] toArray(T[] a)     { return s.toArray(a); }
    public String toString()          { return s.toString(); }
    public int hashCode()             { return s.hashCode(); }
    public boolean equals(Object o)   { return o == this || s.equals(o); }
    public boolean containsAll(Collection<?> c) {return s.containsAll(c);}
    public boolean removeAll(Collection<?> c)   {return s.removeAll(c);}
    public boolean retainAll(Collection<?> c)   {return s.retainAll(c);}
    // addAll is the only inherited implementation
    ......
}
```

结语

至此*Java Collections Framework Internals*系列已经全部讲解完毕，希望这几篇简短的博文能够帮助各位读者对Java容器框架建立基本的理解。通过这里可以返回本系列文章目录

如果对各位有哪怕些微的帮助，博主将感到非常高兴！如果博文中有任何的纰漏和谬误，欢迎各位博友指正。