**http://blog.51cto.com/image/skin/artType01.jpg 深入理解WeakHashmap**

2012-05-28 14:29:22

标签：[SoftReference](http://blog.51cto.com/tag-SoftReference.html) [WeakReference](http://blog.51cto.com/tag-WeakReference.html) [WeakHashmap](http://blog.51cto.com/tag-WeakHashmap.html) [finilize](http://blog.51cto.com/tag-finilize.html)

原创作品，允许转载，转载时请务必以超链接形式标明文章 [原始出处](http://mikewang.blog.51cto.com/3826268/880775) 、作者信息和本声明。否则将追究法律责任。<http://mikewang.blog.51cto.com/3826268/880775>

WeakHashmap

**(一) 查看API文档，WeakHashmap要点如下：**

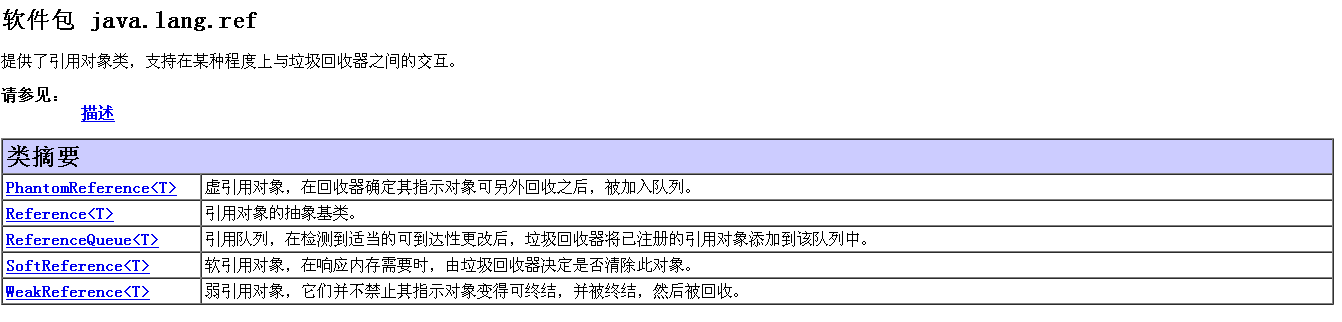
1. 以弱键 实现的基于哈希表的 Map。在 WeakHashMap 中，当某个键**不再正常使用**时，将自动移除其条目。更精确地说，对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的丢弃，这就使该键成为可终止的，被终止，然后被回收。丢弃某个键时，其条目从映射中有效地移除

2. WeakHashMap 类的行为部分取决于垃圾回收器的动作。因为垃圾回收器在任何时候都可能丢弃键，WeakHashMap 就像是一个被悄悄移除条目的未知线程。特别地，即使对 WeakHashMap 实例进行同步，并且没有调用任何赋值方法，在一段时间后 size 方法也可能返回较小的值，对于 isEmpty 方法，返回 false，然后返回true，对于给定的键，containsKey 方法返回 true 然后返回 false，对于给定的键，get 方法返回一个值，但接着返回 null，对于以前出现在映射中的键，put 方法返回 null，而 remove 方法返回 false，对于键 set、值 collection 和条目 set 进行的检查，生成的元素数量越来越少。

3. WeakHashMap 中的每个键对象间接地存储为一个弱引用的指示对象。因此，不管是在映射内还是在映射之外，只有在垃圾回收器清除某个键的弱引用之后，该键才会自动移除。

**(二) 引用对象的四种分类：**

Api图片如下：

[](http://img1.51cto.com/attachment/201205/143200794.png)

1**. 强引用（Strong Reference）**

2**. 弱引用（Weak Reference）**

3.**软引用（Soft Reference）**

4**. 幻象引用（Phantom Reference）**

**(三) 深入理解**

至此，对于WeakHashMap有了一个基本概念，但是还是比较模糊。找到一篇英文文档，将要点总结如下（如有翻译不对，请指出，谢谢）：

**1. 强引用**

强引用就是普通的Java引用，代码：

1. StringBuffer buffer = **new** StringBuffer();

这句代码创建了一个StringBuffer对象，在变量buffer中存储了这个对象的一个强引用。强引用之所以称之为“强”（Strong），是因为他们与垃圾回收器（garbage collector）交互的方式。特别是（specifically），如果一个对象通过强引用连接（strongly reachable-强引用可到达），那么它就不在垃圾回收期处理之列。当你正在使用某个对象而不希望垃圾回收期销毁这个对象时，强引用通常正好能满足你所要的。

**2. 当强引用太强的时候**

假定你要使用一个final类Widget，但是基于某种原因，你不能继承（extend）这个类或者通过其他方法为这个类增加一些新的功能。如果你需要跟踪（track）这个类的不同对象的序列号。那么可以将这些对象放入HashMap中，获得不同的value值，这样就可以做到通过不同的value值跟踪这些对象了。代码：

1. serialNumberMap.put(widget, widgetSerialNumber);

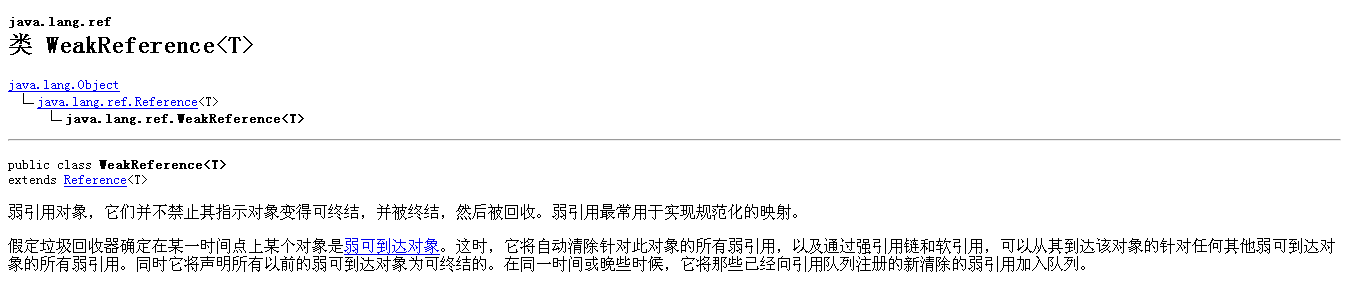
但是widget的强引用会产生一些问题。在我们不需要一个Widget的序列号时，我们需要将这个Widget对应的Entry从HashMap中移除。否则我们可能面临内存泄露（memory leak）的问题（如果我们没有在应当移除Widgt的时候移除它），或者我们将莫名其妙的丢失序列号（如果我们正在使用Widget时却移除了它）。如果这些问题类似，那么这些问题是无垃圾回收机制（non-garbage-collected language）的编程语言的开发者所面临的问题。Java的开发者不需要担心这种问题。

**W：如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。**

强引用的另外一个常见问题就是图片缓存（cache）。普通的强引用将使得Image继续保存在内存中。在一些情况下，我们不需要有些Image继续留在内存中，我们需要将这些图片从内存中移出，这时，我们将扮演垃圾回收期的角色来决定哪些照片被移除。使这些被移出的图片被垃圾回收器销毁。下一次，你被迫再次扮演垃圾回收期的角色，手动决定哪些Image被回收。

Note：我觉得也可以用对象的hashCode来跟踪对象。作者在此所举的例子，只在说明Strong Reference。

**3. 弱引用**

[](http://img1.51cto.com/attachment/201205/143311578.png)

简单说，就是弱引用不足以将其连接的对象强制保存在内存中。弱引用能够影响（leverage）垃圾回收器的某个对象的可到达级别。代码：

1. WeakReference<Widget> weakWidget = **new** WeakReference<Widget>(widget);

你可以使用weakWidget.get()方法老获取实际的强引用对象。但是在之后，有可能突然返回null值（如果没有其他的强引用在Widget之上），因为这个弱引用被回收。其中包装的Widget也被回收。

解决Widget序列号的问题，最简单的方法就是使用WeakHashmap。其key值为弱引用。如果一个WeakHashmap的key变成垃圾，那么它对应用的value也自动的被移除。

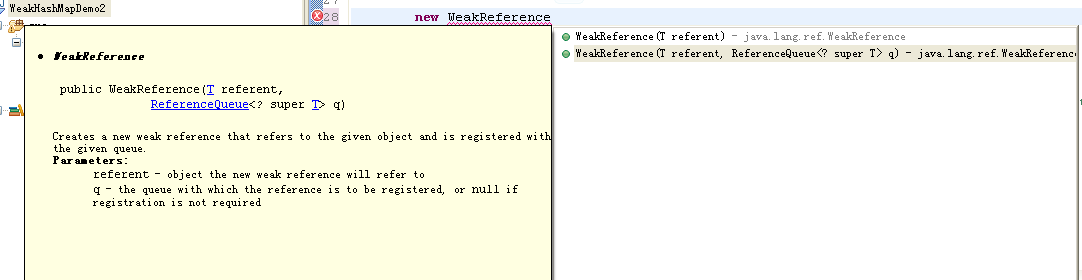
**W：垃圾回收期并不会总在第一次就找到弱引用，而是会找几次才能找到。**

**4. 引用队列（Reference Quene）**

一旦弱引用返回null值，那么其指向的对象（即Widget）就变成了垃圾，这个弱引用对象(即weakWidget)也就没有用了。这通常意味着要进行一定方式的清理（cleanup）。例如，WeakHashmap将会移除一些死的（dread）的entry，避免持有过多死的弱引用。

ReferenceQuene能够轻易的追踪这些死掉的弱引用。可以讲ReferenceQuene传入WeakHashmap的构造方法（constructor）中，这样，一旦这个弱引用指向的对象成为垃圾，这个弱引用将加入ReferenceQuene中。

如下图所示：

[](http://img1.51cto.com/attachment/201205/143814985.png)

**5. 软引用**

除了在抛出自己所指向的对象的迫切程度方面不一样之外，软引用和弱引用基本一样。一个对象为弱可到达（或者指向这个对象的强引用是一个弱引用对象-即强引用的弱引用封装），这个对象将在一个垃圾回收循环内被丢弃。但是，弱引用对象会保留一段时间之后才会被丢弃。

软引用的执行和弱引用并没有任何不同。但是，在供应充足（in plentiful supply）的情况下，软可到达对象将在内存中保存尽可能长的时间。这使得他们在内存中有绝佳的存在基础（即有尽可能长存在的基础）。因为你让垃圾回收器去担心两件事情，一件是这个对象的可到达性，一件是垃圾回收期多么想要这些对象正在消耗的内存。

**6. 幻象引用（phantom reference）**

幻象引用于弱引用和软引用均不同。它控制其指向的对象非常弱（tenuous），以至于它不能获得这个对象。get()方法通常情况下返回的是null值。它唯一的作用就是跟踪列队在ReferenceQuene中的已经死去的对象。

幻象引用和弱引用的不同在于其入队（enquene）进入ReferenceQuene的方式。当弱引用的对象成为若可到达时，弱引用即列队进入ReferenceQuene。这个入队发生在终结（finialize）和垃圾回收实际发生之前。理论上，通过不正规（unorthodox）的finilize（）方法，成为垃圾的对象能重新复活（resurrected），但是弱引用仍然是死的。幻象引用只有当对象在物理上从内存中移出时，才会入队。这就阻止我们重新恢复将死的对象。

W：终结(Finalization)指比拉圾回收更一般的概念,可以回收对象所占有的任意资源,比如文件描述符和图形上下文等。

幻象引用由两个好处：

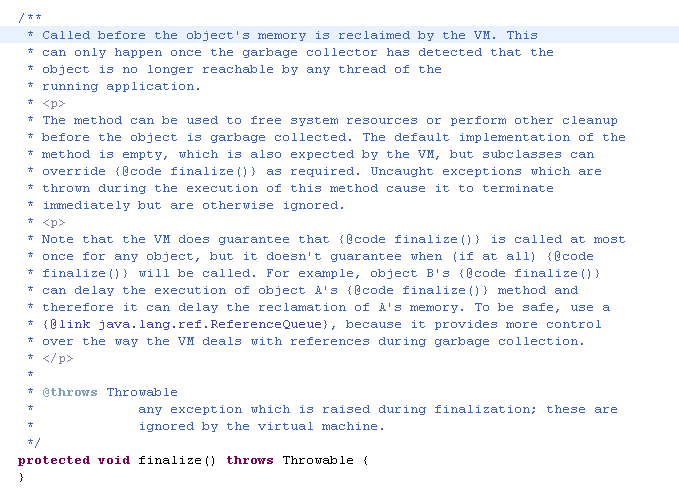
A:它能确定某一个对象从内存中移除的时间，这也是唯一的方式。通常情况下，这不是非常有用。但是迟早（come in handy）会用到手动处理大图片的情况：如果你确定一张图片需要被垃圾回收，那么在你尝试加载下一张照片前，你应该等待这张照片被回收完成。这样就使得令人恐惧的（dreaded）内存溢出不太可能发生。

B:虚幻引用能够避终结（finilize）的基本问题。finilize（）方法能够通过给一个垃圾对象关联一个强引用使之复活。问题是覆写了finilize()方法的对象在成为垃圾之前，为了回收，垃圾回收期需要执行两次单独的循环。第一轮循环确定某个对象是垃圾，那么它就符合终结finilize的条件。在finilize的过程中，这个对象可能被“复活”。在这个对象被实际移除之前，垃圾回收期不得不重新运行一遍。因为finilization并不是实时调用的，所以在终止进行的过程中，可能发生了gc的多次循环。在实际清理垃圾对象时，这导致了一些延时滞后。这将导致Heap中有大量的垃圾导致内存溢出。

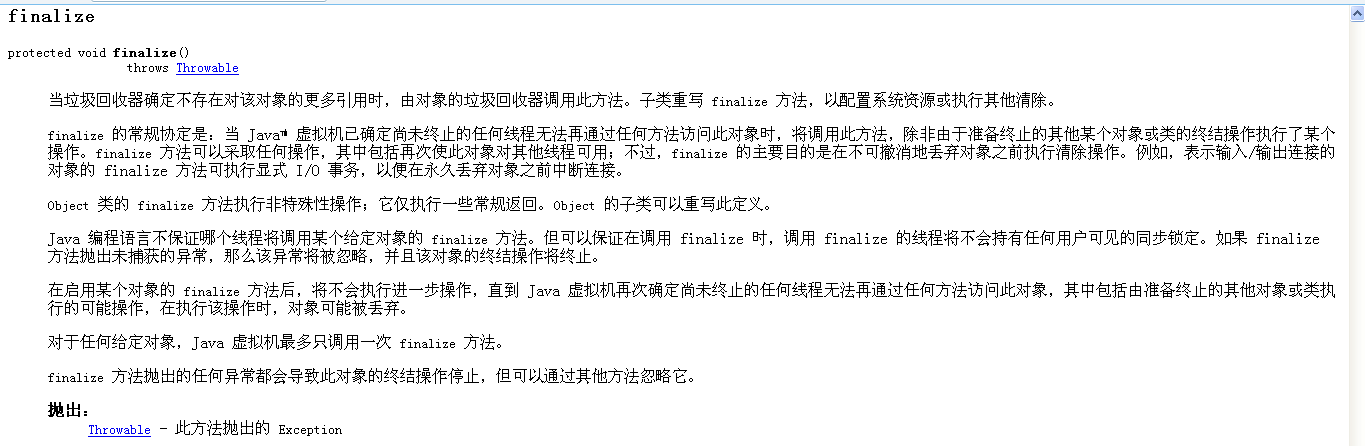
幻象引用不可能发生以上的情况，当幻象引用入队时，它实际上已经被移除了内存。幻象内存无法“复活”对象。这发现这个对象时虚幻可到达时，在第一轮循环中，它就被回收。

可以证明，finilize()方法从不在第一种情况下使用，但是虚幻引用提供了一种更安全和有效的使用和被排除掉的finilize方法的机制，使得垃圾回收更加简单。但是因为有太多的东西需要实现，我通常不适用finilize。

**W:Object类中相关内容如下：**

[](http://img1.51cto.com/attachment/201205/143413177.png)

**文档中相关内容如下：**

[](http://img1.51cto.com/attachment/201205/143500346.png)

**(四) 原文地址及其他翻译**

原文地址：<http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html>

其他翻译地址：<http://blog.csdn.net/fancyerii/article/details/5610360>

本文出自 “[小新专栏](http://mikewang.blog.51cto.com/)” 博客，请务必保留此出处<http://mikewang.blog.51cto.com/3826268/880775>