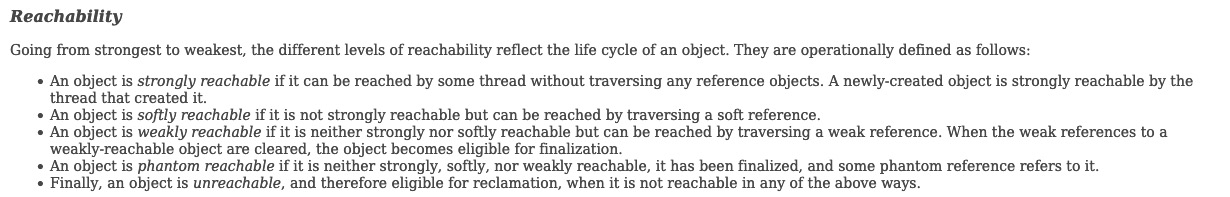
# Java引用类型

## Java引用可见性

先看官方文档[1]，截一下图。



Strongly reachable: 不解释了。

Softly reachable：通过遍历软引用可见（reachable）。

Weakly reachable：遍历弱引用可见，清理弱引用时，就可以执行finalization了。

Phantom reachable：已经执行了finalized。

## 引用队列

应用可以将引用注册到引用队列让*gc完成时*通知应用。参见Reference.class中next字段，以及入队列的逻辑。

## 引用类型

### 强引用

强引用即是普通的Java引用，例如：

*StringBuffer buffer = new StringBuffer();*

创建一个StringBuffer实例会将一个指向它的强引用存储到变量buffer。如果一个对象通过一串强引用可见（reachable），则成为强可见。强可见表示一个对象不应被垃圾收集器回收。

**当强引用过强时**

使用不能合理扩展的类很常见。如将类标识为final或者更复杂时通过很多未知具体实现类支撑的工厂返回的接口。设想使用一个Widget类，并且不可能（实际）扩展Widget增加新功能。

如果试图跟踪这样的对象会如何呢？比如我们想跟踪每个Widget对象的序列号，而其并不真的拥有序列号属性（没法扩展，不能加属性）。完全没问题，用HashMap：

*serialNumberMap.put(widget, widgetSerialNumber);*

看起来没问题，但指向widget实例的强引用引出了问题。我不得不知道（100%肯定）哪一个序列号不再使用了，以便我们可以将它从Map中清除。否则会发生内存泄漏（该删除而不删除，或者丢失了序列号）。这些问题很熟悉，当使用没有垃圾收集语言时即是如此，需要自己管理内存，在Java中不应该有如此担心。

另一个强引用的常见问题是缓存，尤其是大的数据结构如图像。设想你的应用不得不处理用户提供的图像。很自然地你会缓存这些图像，因为从磁盘加载图像耗费资源，并且要避免在内存中保存两份同样的图像。

图像缓存应避免反复加载不需要的图像，而且其只应保存指向内存中图像的引用。对普通强引用，引用本身强制图像保留在内存中，需要我们自己确定图像是否需要保存在内存中，并在不需要时从缓存中删除它以便让垃圾收集器清理内存空间。我们又一次回到了自己管理内存的问题上来。

### **弱引用**

简单来说，弱引用就是一个不能足够强到让对象保留在内存中的引用。弱引用让垃圾收集器确定是否其可见性（reachable），不需要我们自己来实现。创建弱引用的方法如下所示

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

在代码中，可以使用weakWidget.get()得到实际的Widget对象。当然弱引用还不能强到组织垃圾收集，有时（当然是对象没有强引用时）weakWidget.get()突然开始返回null值了。

为了解决上述widget序列号问题,最简单地方法就是使用内置的WeakHashMap类。WeakHashMap和HashMap以同样的方式工作, 除了其键值（不是值）是使用弱引用来引用的。如果WeakHashMap键值成为垃圾, 它的entry也会自动删除. 这避免了前面提到的陷阱。

### **引用队列**

一旦WeakReference开始返回null值，它指向的对象便以成为垃圾，WeakReference对象本身也便基本无用了。这常常意味着要做一些清理工作，如WeakHashMap, 将不得不删除无用的entry以避免保留不断增加的WeakReferences尸体。

ReferenceQueue类使得跟踪引用尸体很容易。如果在创建弱引用时指定了一个 ReferenceQueue那么在引用对象指向的对象本身成为垃圾时，引用本身会被自动添加到引用队列。 我们可以定期处理ReferenceQueue对引用尸体执行必要的清理工作。

### **“弱”级别**

实际上有强到弱有四个引用强度：strong, soft, weak, and phantom. 前面讨论了强引用和弱引用，下面继续看其他两个。

### **软引用**

软引用和弱引用很像，除了它不是很期盼丢弃它引用的对象。A soft reference is exactly like a weak reference, except that it is less eager to throw away the object to which it refers. 弱可见的对象（最强引用时弱引用）将会在下一个垃圾收集周期被丢弃（回收），而软可见对象通常会多存在一段时间。

SoftReferences不需要和WeakReferences有什么不同, 但实际上在内存足够多时弱可见对象通常会尽可能保留在内存中。这使得SoftReferences更适合构建缓存，我们只需要让垃圾收集器考虑对象的可见性（强可见对象不会被垃圾收集器从内存中删除）。

### **幻引用**

幻引用和SoftReference或者WeakReference有很大区别. 我们不能从幻引用获取其引用的对象，它的get()方法始终返回null值。幻引用的唯一用处是跟踪何时其被放入引用队列ReferenceQueue, 这时我们知道其指向的对象已经死了。那么其和弱引用有什么不同吗？

不同之处发生在进入ReferenceQueue时。只要发现WeakReferences引用的对象弱可见时， 弱引用就会被添加到引用队列中。这**发生在对象finalization或者垃圾收集之前**。 理论上，对象仍旧可能在finalize()中复活。但WeakReference仍会保留死亡状态。PhantomReferences只会在**其指向对象物理上从内存中抹除时才会放入到引用队列**。其get()方法返回null值可以阻止其复活僵死的对象。

那PhantomReferences有什么用处吗？有两个场景:

第一，他允许我们确定对象何时从内存在抹除了。而且这是唯一的方法。看起来似乎用处不大，但我们确实了解了发生在内存中的时间，我们可以采取应对措施尽可能避免OutOfMemoryError。

第二，PhantomReferences避免了finalization可能引起的问题: finalize() 方法可能通过创建强引用让将死之人死而复生。那又怎么样？问题是，现在确定回收一个实现了finalize方法的对象需要至少两个垃圾收集周期。第一个周期确定它是垃圾，准备执行finalization。由于在执行finalization过程中，将死的对象可能重新复活，垃圾收集器必须在另一个收集周期确定其是否真正可以回收。而且finalization也不一定可以及时进行，那么就有可能多个垃圾收集周期后，内存还是没有被释放。这意味着，清除内存需要更长的时间，发生OutOfMemoryErrors 的概率更大（这时有些该清除的内存无法清理掉）.

使用PhantomReference, 这种情况就不会发生了。当PhantomReference进入引用队列时，已经无法得到僵尸对象指针（通过get）. 幻引用不可能让对象重生，其指向的对象可以在发现其为幻可见时即刻清除. 可以释放任何需要处理的资源。

Arguably, the finalize() method should never have been provided in the first place. PhantomReferences are definitely safer and more efficient to use, and eliminating finalize() would have made parts of the VM considerably simpler. But, they're also more work to implement, so I confess to still using finalize() most of the time. The good news is that at least you have a choice. （finalize仍旧为我们提供了选项）

以上引用类型章节来自于[4]的简单翻译。[3]中有对幻引用入队列循序的不同解释，可以对比看看。[1][5]说所有引用在入引用队列之前都会被清理（cleared）。

1. <https://docs.oracle.com/javase/8/docs/api/java/lang/ref/package-summary.html>
2. [https://web.archive.org/web/20110303225354/http://weblogs.java.net/blog/2006/05/04/understanding-weak-references](https://web.archive.org/web/20110303225354/http:/weblogs.java.net/blog/2006/05/04/understanding-weak-references)
3. <http://www.pawlan.com/monica/articles/refobjs/>
4. <https://plumbr.io/blog/garbage-collection/weak-soft-and-phantom-references-impact-on-gc>
5. <https://www.ibm.com/developerworks/library/j-jtp01246/index.html>