# **理解Reference**

[](https://www.jianshu.com/u/48a713ebca55)

[chandarlee](https://www.jianshu.com/u/48a713ebca55) 关注

2017.08.28 01:28\* 字数 3440 阅读 422评论 1喜欢 4

#### **java.lang.ref**

该包下提供了Reference相关的类，包括基类Reference，三个子类WeakReference、SoftReference和PhantomReference，以及一个能和它们配合使用的类ReferenceQueue。通过使用这些类，开发者可以通过包装目标对象，创建指向目标对象的不同的引用类型。使用这些引用类，并不会阻碍JVM对目标对象的回收。并且，如果和ReferenceQueue配合使用，在目标对象的可达性发生变化时，我们还能得到JVM的通知(确切来说是通过查询与之关联的引用队列感知到这种变化)，这可能对我们监控目标对象的生命周期很有帮助。通过使用这种方式，开发者和JVM的垃圾回收器能够有一定程度的交互。

#### **目标对象可达性的定义**

在JVM中，通过可达性可以判断一个目标对象是否存活从而进行垃圾回收。JVM会从GC Roots(如线程局部变量、类静态变量等)开始遍历，构建一颗引用树，如果不存在至目标对象的引用路径，目标对象将标记为不可达，并在未来进行回收。目标对象某些时刻可能同时存在多条引用路径。对象的可达性主要有：

* 强可达：目标对象至少存在一条引用路径，该引用路径中不包含(不经过)任何的Reference类。
* 软可达：目标对象非强可达，且至少存在这样一条引用路径，该路径中包含(经过)的第一个Reference类为SoftReference；(分为两种情况：1.目标对象为SoftReference中的根；2.目标对象在SoftReference中的根对象的某条引用路径上)
* 弱可达：目标对象非强可达和软可达，且至少存在这样一条引用路径，该路径中包含(经过)的第一个Reference类为WeakReference；
* 虚可达：目标对象非强可达、软可达和弱可达，且至少存在这样一条引用路径，该路径中包含(经过)的第一个Reference类为PhantomReference，且该对象已经执行过finalize方法；
* 不可达：不存在任何至目标对象的引用路径。

****对象的可达性是互斥的，从上至下可达性递减；对象如果同时存在多条引用路径，那么可达性由最强的路径决定；****

#### **finalize和对象的状态**

我们都知道，Object类中有个finalize方法；GC Collector中存在这样一个队列F-QUEUE，在GC首次标记一个对象为不可达时，如果目标对象重写了finalize方法，会将该对象添加至这个队列中，且状态变为finalizable。同时，也存在这样一个后台线程，姑且叫做finalizer-handler，它负责不断的从前面的队列中取出对象并执行finalize方法。一个对象如果执行过finalize方法，状态就是finalized；之后，如果对象的可达性不再发生变化，那么该对象就会被回收了。为什么这样说呢？因为在finalize方法中，我们可以改变该对象的可达性，比如重新引用该对象，通过这种方式，我们拯救了一个即将被回收的对象，这种情况也叫做对象重生。如下面的代码所示：

public class Reborn{

static Reborn sNewLife;

@Override

protected void finalize() throws Throwable {

super.finalize();

sNewLife = this;

}

}

那提供finalize方法的意义何在呢？因为GC只负责内存相关的回收工作，其他资源需要开发者自己释放，如数据库连接、文件句柄等。因此，我们可以根据需要重写该方法，在对象被回收之前，做一些最后的清理工作。但是在使用时需注意：

* 如果没有重写finalize方法，或者重写了但只采用默认实现，那么GC不会将目标对象加入F-QUEUE，而是直接回收对象。
* GC会记录相应的状态，对象的finalize方法在对象生命周期过程中只会被执行一次。因此如果对象在finalize方法中重生了，下一次再进入回收阶段时，不会再执行该方法。****应该尽量避免对象的再生，如果非要再生，请不要直接使用当前对象，而是基于当前对象重新构建一个新的对象。****

基于以上的原因，其实finalize并不是很可靠。我们不能过度依赖这个方法，其实使用PhantomReference和ReferenceQueue也能达到对应的效果且更稳定，这个后面再说。

介绍完finalize，我们再来说说对象的状态划分。在虚拟机中，对象的状态可以总结为以下几个阶段(细分的话还有其他状态，但跟Reference相关的主要是以下这几个)：

* Reachable：可达的，这里的可达指强可达；一般而言，新创建的对象都处于这个状态；
* Finalizable：即将执行对象的finalize方法，F-QUEUE中的对象都是这个状态；
* Finalized：已经执行过对象的finalize方法，此时对象可能是可达的或者是等待回收的状态的，因为对象可能重生；
* Reclaimable：可回收的；处于该状态的对象是Finalized的且没有其他强引用的。
* Reclaimed：完成内存回收。

这些状态之间有些是互斥的，有些是能够并存的！比如一个再生的对象应该是Reachable且Finalized的。而一个虚可达的对象是Finalized且Reclaimable的，只要清空引用就能真正被回收。

#### **Reference类如何工作**

Reference类的三个子类可以单独使用，也可以和ReferenceQueue配合使用，在目标对象的可达性发生变化时，如果提供有ReferenceQueue，那么会将该Reference对象加入到队列中。开发者通过ReferenceQueue#poll或是ReferenceQueue#remove方法查看队列是否包含对应的Reference对象，从而可以判断目标对象的可达性是否发生了变化，这方便了监控或是进行其他与对象生命周期相关的处理逻辑。先来看看WeakReference的工作过程：

* 直接将对应的Reference对象设置为null，不会触发下面的处理过程
* 未被处理时，WeakReference#get方法可以返回目标对象的引用
* GC时，一旦检测到目标对象仅为弱可达，无论当时的内存情况如何，会进一步处理WeakReference
* 具体的，GC会清除掉****所有****WeakReference中对目标对象的引用，即将referent字段置为null，这样会导致WeakReference#get方法将返回null
* 如果目标对象需要执行finalize方法(有实现且未执行过），则加入F-QUEUE，目标对象转到finalizable状态
* 与此同时或之后某个时间，将WeakReference对象添加到对应的ReferenceQueue队列中(如果存在）
* 当我们从ReferenceQueue中查询到对应的WeakReference对象时，并不知道目标对象的命运到底是如何或会如何！这个时候有可能并没有执行finalize方法，也可能执行过了！我们只知道目标对象曾经是finalizable的，可能执行完finalize方法之后，目标对象又重生了\*\*
* 处理目标对象时，会级联处理通过目标对象到达的其他弱可达的对象

@Test(timeout = 10000)public void weak\_reference() throws InterruptedException {

A a = new A();

ReferenceQueue<A> queue = new ReferenceQueue<>();//关联的队列

WeakReference<A> weakReferenceA = new WeakReference<>(a, queue);

a = null;//目标对象a只存在弱引用，为弱可达

Runtime.getRuntime().gc(); //更容易触发gc

Thread.sleep(2000);

assertTrue(A.sA != null);//对象重生了

assertTrue(weakReferenceA.get() == null);//引用被GC Clear掉了

//check queue

while (true){

Reference<A> item = (Reference<A>) queue.poll();

if (item != null){

assertTrue(weakReferenceA == item);//被添加到队列中了

break;

}

}

}

GC时，目标对象仅存在弱引用，接着弱引用被清除，并被添加到引用队列中。虽然对象a通过finalize方法完成了再生，但不妨碍它被清除且添加至引用队列中。

对于SoftReference来说，基本类似于WeakReference的表现。只有一点需要注意，****GC在内存不足时才会处理软引用可达的对象，而WeakReference弱可达是一旦GC触发就会处理****。  
而对于PhantomReference就跟前两者不太一样，具体说明如下：

* 直接将对应的Reference对象设置为null，不会触发下面的处理过程
* 无论是否已经被处理，PhantomReference#get方法始终返回null
* GC时，一旦检测到目标对象仅为虚可达，无论当时的内存情况如何，会进一步处理PhantomReference
* GC****不会清除****掉PhantomReference中对目标对象的引用，即不会将对象中的referent字段置为null，需要我们手动调用clear方法进行清除
* 如果目标对象需要执行finalize方法(有实现且未执行过），则加入F-QUEUE，目标对象转到finalizable状态
* PhantomReference对象不会立刻被添加至对应的ReferenceQueue队列中，需要确保目标对象****执行完成****finalize方法，且不会重生；即：****当我们在ReferenceQueue中检测到PhantomReference对象时，它所包装的目标对象肯定是Finalized，且仅仅存在虚引用的，也就是说此时目标对象的状态为Reclaimable****
* ****因为虚引用的referent不会被gc主动clear，因此需要我们手动调用clear方法，或者将对应PhantomReference变为不可达，否则目标对象也不会被执行到最后的内存回收阶段，而仅仅是保持在可回收状态****
* 处理目标对象时，会级联处理通过目标对象到达的其他虚可达的对象

@Test(timeout = 20000)public void phantom\_reference2() throws InterruptedException, NoSuchFieldException, IllegalAccessException {

A a = new A();

ReferenceQueue<A> queue = new ReferenceQueue<>();

PhantomReference<A> phantomReferenceA = new PhantomReference<>(a, queue);

assertTrue(phantomReferenceA.get() == null); //get方法始终返回null

a = null; //对象仅虚可达

System.gc();

Thread.sleep(2000);

assertTrue(A.sA != null);//对象在finalize中重生，因此不会加入引用队列中

//如果下面的代码注释掉了，测试用例会因为timeout而执行失败

//A.sA = null;

//System.gc();

//Thread.sleep(2000);

//check queue

while (true){

Reference<A> item = (Reference<A>) queue.poll();

if (item != null){

Field field = Reference.class.getDeclaredField("referent");

field.setAccessible(true);

Object object = field.get(item);

//区别于WeakReference和SoftReference，GC不会帮PhantomReference自动清理

assertTrue(object != null);

//需要手动clear掉

item.clear();

break;

}

}

}

当我们将中间的一段代码注释掉运行时，测试用例会因为超时运行失败，因为在引用队列中无法获取对应的PhantomReference导致死循环，因为Reference对象不满足加入到队列中的条件****finalized且仅虚可达****，而当我们不注释这段代码时，运行正常。需要注意的是，在代码中，我们还通过反射方式去获取对象中的referent字段，发现是存在值得且可用的，说明虚拟机并没有自动替我们清理掉这个字段，这一点不同于上面的两个Reference类型。

#### **Reference和ReferenceQueue的应用案例**

通过上面的说明，我们已经了解了Reference和ReferenceQueue的大概，下面来看它们配合使用的一个例子；****LeakCanary****，想必大家都很熟悉了，它是开发阶段用来检测内存泄漏的一个库，这其中的原理就是使用****WeakReference****和****ReferenceQueue****完成的。这里只描述一下，就不贴代码了。

* 在应用的Application类中注册一个ActivityLifecycleCallbacks回调，重写onActivityDestroyed(Activity activity)方法
* 在退出Activity的时候，该回调中的destroy方法触发，创建一个WeakReference对象包装这个销毁的activity目标对象，并指定一个ReferenceQueue
* 触发GC并监控ReferenceQueue的变化。因为activity对象即将被销毁，因此未来某个时刻应该仅仅存在该activity的弱引用并在GC时得到处理，对应的WeakReference对象被添加至ReferenceQueue中。如果一直检测不到该WeakReference对象被添加至队列中，说明肯定存在其他的引用路径，也就代表了可能存在内存泄漏问题
* 通过android.os.Debug#dumpHprofData方法dump此时的java heap至一个文件中，在后台通过工具分析该heap profile，找出目标activity的引用路径，发送状态栏通知告知开发者

大致的流程就是这样，除了最后一步的dump，前面的都比较简单。说到这里，又不得不提以下Android SDK中的StrictMode类。该类可以帮助开发者在开发阶段发现一些问题。通过该类也能检测到Activity的泄露，但相比leakcanary，它仅能通过打印日志或是抛出异常通知开发者可能发生leak，但不能给出引用路径，还是需要开发者自己去dump heap，自己去分析。另外还有一点，它检测leak的方式是区别于leakcanary的。当开启leak检测的时候，StrictMode类中会记录所有activity类的instance数量，通过一个静态的hashmap字段保存。而在创建和销毁activity的地方会更新activity类对应的instance数量。如下：

public class ActivityThread{

...

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {

...

java.lang.ClassLoader cl = r.packageInfo.getClassLoader();

activity = mInstrumentation.newActivity(

cl, component.getClassName(), r.intent);

StrictMode.incrementExpectedActivityCount(activity.getClass());

...

}

private ActivityClientRecord performDestroyActivity(IBinder token, boolean finishing,

int configChanges, boolean getNonConfigInstance) {

...

mActivities.remove(token);

StrictMode.decrementExpectedActivityCount(activityClass);

...

}

}

在StrictMode.decrementExpectedActivityCount方法中，会触发GC，然后检测预期的activity的实例数量和实际的实例数量是否一致来判断是否发生leak，而实际的实例数量通过android.os.Debug#countInstancesOfClass方法可以获取。除了检测activity泄露，StrictMode在开发阶段还能做更多事情，如检测类的实例数量是否超出限制、SqliteObjectLeaks、RegistrationLeaks等，当然这些和Reference扯不上关系，就不谈了，有兴趣可以自己去看代码。