彻底理解引用在 Android 和 Java 中的工作原理

2017-11-20 Android开发中文站

摘要： 本文讲的是彻底理解引用在 Android 和 Java 中的工作原理，几周前，我很荣幸地参加了在波兰举行的 Mobiconf ，移动开发者参加的最好的研讨会之一。我的朋友兼同事 Jorge Barroso 做了个名为“最好（良好）的做法”的演说 ，这让我在听后很有感触：

本文讲的是彻底理解引用在 Android 和 Java 中的工作原理，
几周前，我很荣幸地参加了在波兰举行的 Mobiconf ，移动开发者参加的最好的研讨会之一。我的朋友兼同事 Jorge Barroso 做了个名为“最好（良好）的做法”的演说 ，这让我在听后很有感触：

> 对于一个 Android 开发者，如果你不使用 WeakReferences，这是有问题的。

举个恰当的例子，几个月前，我发布了我的最后一本书 “Android High Performance”, 联席作者是 Diego Grancini。最热门的章节之一就是讨论 Android 的内存管理。在本章中，我们介绍了移动设备中内存的工作原理，内存泄漏是如何发生的，为什么这个是重要的，以及我们可以应用哪些技术来避开它们。因为我从开发 Android 起，就常常看到这么种倾向：轻视甚至无视一切与内存泄漏和内存管理相关的问题。已经满足开发需求了，为何要庸人自扰呢？我们总是急于开发新的功能，我们宁愿在下一个 Sprint 演示中呈现一些可见的东西，也不会关心那些没有人一眼就能看到的东西。

这无疑是导致技术债务一个活生生的例子。 我甚至可以补充地说，技术债务在现实世界中也有一些影响，那是我们不能用单元测试衡量的：失望，开发者间的摩擦，低质量的软件和积极性的丧失。这种影响难以衡量的原因是在于它们常常发生在长远的将来的某个时间点。这有点像政客：如果我只当政 8 年，为何我要烦心 12 年后将要发生的事呢？除了在软件开发，一切都以更快的方式。

编写软件开发中应该采纳的设计思想可能需要一些大篇文章，而且已经有很多书和文章可供您参考。然而，简要地解释不同类型的内存引用，它们具体是什么，以及如何在 Android 中使用，这是个相对简短的任务，这也是我想在本文中做的。

首先：Java 中的引用是什么?

> 引用指向了一个对象，你能通过引用访问对象。

Java 默认有 4 种类型的引用：强引用（StrongReference）、软引用（SoftReference）、弱引

用（WeakReference） 和虚引用（PhantomReference）。部分人认为只有强引用和弱引用两种类型的引用，而弱引用有两个层次的弱化。我们习惯于将生活中的一切事物归类，那种毅力堪比植物学家对植物的分类的。不论你觉得哪种分类更好，首先你需要去理解这些引用。然后你可以找出自己的分类。

各种引用都是什么意思?

StrongReference： 强引用是 Java 中最为常见的引用类型。任何时候，当我们创建了一个对象，强引用也同时被创建了。比如，当我们这么做：

```
MyObject object = new MyObject();
```

一个新的 MyObject 对象被创建，指向它的强引用保存在 object 中。你还在看吧？ 嗯，更有意思的事情来了，这个 object 是可以强行到达的——意思就是，它可以通过一系列强引用找到，这将会阻止垃圾回收机制回收它，然而，这正是是我们最想要的。现在，我们来看个例子。

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        new MyAsyncTask().execute();
    }

    private class MyAsyncTask extends AsyncTask {
        @Override
        protected Object doInBackground(Object[] params) {
            return doSomeStuff();
        }
        private Object doSomeStuff() {
            //do something to get result
            return new MyObject();
        }
    }
}
```

花几分钟，尝试去找可能出现问题的点。

不用担心，如果一时找不到，那再花点时间看看。

现在呢?

AsyncTask 对象会在 Activity onCreate() 方法中创建并运行。但这里有个问题：内部类在它的整个生命周期中是会访问外部类。

如果 Activity 被 destroy 掉时，会发生什么？ AsyncTask 仍然持有 Activity 的引用，所以 Activity 是不能被 GC 回收的。这就是我们所说的内存泄漏。

> 旁注： 以前，我曾经对合适的人进行访谈，我问他们如何创建内存泄漏，而不是询问内存泄漏的理论方面。这总是更有趣!

内存泄漏实际上不仅发生在 Activity 自身销毁的时候，配置的改变（译者注：比如横屏切换成竖屏）或系统需要更多的内存时，也可能系统强行销毁。如果 AsyncTask 复杂点（比如，持有 Activity 上的 View 的引用），它甚至会导致崩溃，因为 view 的引用是 null。（译者注：这个是直译过来的，可能对部分同学来说，不太好理解。我举个例子吧，比如 AsyncTask 中引用了 ProgressDialog，AsyncTask 运行时会显示 ProgressDialog，当横屏切成竖屏时，这时会出现崩溃。 (╯ ^ ╰〉)

那么，要如何防止这种问题再次发生呢？我们接下来介绍另一种类型的引用：

WeakReference：弱引用是引用强度不足以将对象保持在内存中的引用。如果垃圾回收机制试图确定对象的引用强度，如果恰好是通过 WeakReferences 引用，那么该对象将被垃圾回收。为了便于理解，最好是先抛开理论，用上个例子来说明如何使用 WeakReference 来避免内存泄漏：

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        new MyAsyncTask(this).execute();
     }
    private static class MyAsyncTask extends AsyncTask {
        private WeakReference mainActivity;

        public MyAsyncTask(MainActivity mainActivity) {
            this.mainActivity = new WeakReference<>(mainActivity)
;
        }
```

```
        @Override
        protected Object doInBackground(Object[] params) {
            return doSomeStuff();
        }
        private Object doSomeStuff() {
            //do something to get result
            return new Object();
        }
        @Override
        protected void onPostExecute(Object object) {
            super.onPostExecute(object);
            if (mainActivity.get() != null){
                //adapt contents
            }
        }
    }
    }
```

现在注意一个主要区别：Activity 是这样被内部类引用的：

```
  private WeakReference mainActivity;
```

这样做有什么差别呢？ 当 Activity 不存在时，由于它是被 WeakReference 持有的，可以被收集。 因此，不会发生内存泄漏。

> 旁注: 如果你现在对 WeakReferences 有预期的更好的了解，你会发现类 WeakHashMap 是很有用的。 它完全就是一个 HashMap，除了使用 WeakReferences 引用键（键，而不是值）。 这使得它们对于实现诸如缓存之类的实体非常有用。

我们提到过更多的引用类型。 让我们看看它们在什么地方有用，以及我们如何能从中受益：

SoftReference: 软引用可以作为一个引用强度更强的弱引用。在弱引用将被立即回收的情形下，软引用会向 GC 请求留在内存中，除非没有其他选项（否则是不会回收软引用持有的对象）。垃圾回收算法真的很有意思，你可以几个小时内沉醉于研究它，而不会感到疲惫。但大体上，垃圾回收会这么解说“我会永远收回弱引用。 如果对象是 软引用，我将基于具体条件决定是否回收。” 这使得软引用对于实现缓存非常有用：只要内存足够，我们就不必担心手动删除对象。 如果你想看实际中的例子，你可以查看这个例子，用软引用实现的缓存。

PhantomReference：额，虚引用! 在实际产品开发中，我见过的，使用的次数不会超过 5 次。

垃圾收集器能随时回收虚引用 持有的对象，只要它乐意。没有进一步的解释，没有回调，这使得它难以描述。为什么我们要使用这样的东西？ 其他几个的问题还不够吗？ 为什么我会选择成为程序员？ 虚引用可以精确地用于检测对象是否已从内存中删除。 说实话，在我的整个职业生涯中不得不用虚引用的场景只有两次。 所以，即便你现在不是很难理解，也不要感到有压力。

希望这有稍微消除点之前你对引用的疑虑。作为学习的东西，或许你现在想要来点练习，玩你自己的代码，看看能怎么改进它。第一步应该是看看有没有内存泄漏，然后看看能否通过这里所学的知识去改掉那些令人讨厌的内存泄漏。如果你喜欢这篇文章，或者它确实帮到了你，请随意分享或者留下你的评论。这也是我这位业余写手的动力。

感谢我的同事 Sebastian 对这篇文章的投入！

阅读原文