— we won’t email you for any other reason, ever.

About the Author: Pierre-Yves Ricau

Pierre-Yves Ricau 是 Square 的一枚员工，享受编程和优质的代码

@piwai Website

Save the date for Droidcon SF in March — a conference with best-in-class presentations from leaders in all parts of the Android ecosystem.

(0:00)

大家好，我是 Pierre-Yvews Ricau (叫我 PY 就行)，现在在 Square 工作。

Square 出了一款名为：Square Register 的 App， 帮助你用移动设备完成支付。在用这个 App 的时候，用户先要登陆他的个人账号。

不幸的是，在签名页面有的时候会因为内存溢出而出现崩溃。老实说，这个崩溃来的太不是时候了 — 用户和商家都无法确认交易是否完成了，更何况是在和钱打交道的时候。我们也强烈的意识到，我们需要处理下内存溢出或者内存泄露这种事情了。

(1:40)

我想要聊的内存泄露解决方案是： LeakCanary。 LeakCanary 是一个可以帮助你发现和解决内存泄露的开源工具。但是到底什么是内存泄露呢？我们从一个非技术角度来开始，先来举个例子。

假设我的手代表着我们 App 能用的所有内存。我的手里能放很多东西。比如：钥匙，Android 玩偶等等。设想我的 Android 玩偶需要扬声器才能工作，而我的扬声器也需要依赖 Android 玩偶才能工作，因此他们持有彼此的引用。

我的手里可以持有如此多的东西。扬声器依附到 Android 玩偶上会增加总重量，就像引用会占用内存一样。一旦我放弃了我的玩偶，把他扔到地上，会有垃圾回收器来回收掉它。一旦所有的东西都进了垃圾桶，我的手又轻便了。

不幸的是，有的时候，一些不好的情况会发生。比如：我的钥匙没准和我的 Android 玩偶黏在了一起，阻止我把 Android 玩偶扔到地上。最终的结果就是 Android 玩偶无论如何都不会被回收掉。这就是内存泄露。

有外部的引用（钥匙，扬声器）指向了 本不应该再指向的对象（Andorid 玩偶）。类似这样的小规模的内存泄露堆积以后就会造成大麻烦。

(3:47)

这就是我们为什么要开发 LeakCanary。

我现在可能已经清楚了 可被回收的 Android 对象应该及时被销毁。

但是我还是没法清楚饿看到这些对象是否已经被回收掉。有了 LeakCanary 以后，我们给可被回收的 Android 对象上打了智能标记。智能标记能知道他们所指向的对象是否被成功释放掉。如果过一小段时间对象依然没有被释放，他就会给内存做个快照。

LeakCanary 随后会把结果发布出来，帮助我们看到内存到底怎么泄露了，清晰的展示无法被释放的对象的引用链。

举个具体的例子：在我们的 Square App 里的签名页面。用户准备签名的时候，App 因为内存溢出出错崩溃了。我们不能确认内存错误到底出在哪儿了。

签名页面持有了一个很大的有用户签名的 Bitmap 图片对象。图片的大小和用户手机屏幕大小一致 — 我们猜测这个有可能会造成内存泄露。首先，我们可以配置 Bitmap 为 alpha 8-bit 来节省内存。这是很常见的一种修复方案，而且效果也不错。但是并没有彻底解决问题，只是减少了泄露的内存总量。但是内存泄露依然在哪儿。

最主要的问题是我们 App 的堆满了，应该要留有足够的空间给我们的签名图片，但是由于很多处的内存泄露叠加在一起占用了很多内存。

(8:06)

假设，我有一个 App，这个 App 点一下就能买一个法棍面包（哈哈，可能只有法国人需要这样的App，没错，我就是法国人）。

```
privatestaticButtonbuyNowButton;
```

由于某种原因，我把这个 button 设置成了 static 的。问题随之而来，这个按钮除非你设置成了null，不然就内存泄露了。

你也许会说：“只是一个按钮而已，没啥大不了”。问题是这个按钮还有一个成员变量：叫 “mContext”，这个东西指向了一个 Acitvity，Acitivty 又指向了一个 Window，Window 有拥有整个 View 继承树。算下来，那可是一大段的内存空间。

静态的变量是 GC root 类型的一种。垃圾回收器会尝试回收所有非 GC root 的对象，或者某些被 GC root 持有的对象。所以如果你创建一个对象，并且移除了这个对象的所有指向，他就会被回收掉。但是一旦你将一个对象设置成 GC root，那他就不会被回收掉。

当你看到类似“法棍按钮”的时候，很显然这个按钮持有了一个 Activity 的引用，所以我们必须清理掉它。当你沉浸在你的代码的时候，你肯定很难发现这个问题。你可能只看到了引出的引用。你可以知道 Activity 引用了一个 Window，但是谁引用了 Activity?

你可以用像 IntelliJ 这样的工具做些分析，但是它并不会告诉你所有的东西。通常，你可以把这些 Object 的引用关系组织成图，但是是个单向图。

(10:16)

我们能做些什么呢？我们来做个快照。我们拿出所有的内存然后导出到文件里，这个文件会被用来分析和解析堆结构。其中一个工具叫做 Memory Analyzer，也叫 MAT。它会通过 dump 的内存，然后分析所有存活在内存中的对象和类。你可以用 SQL 对他做些查询，类似如下：

```
SELECT*FROMINSTANCEOFandroid.app.ActivityaWHEREa.mDestroyed=true
```

这条语句会返回所有的状态为 `destroyed` 的实例。一旦你发现了泄露的 Activity，你可以执行 `merge_shortest_paths` 的操作来计算出最短的 GC root 路径。从而找出阻止你 Acitivty 释放的那个对象。

之所以说要 “最短路径”，是因为通常从一个 GC root 到 Acitivty，有很多条路径可以到达。比如说：我的按钮的 parent view，同样也持有一个 `mContext` 对象。

当我们看到内存泄露的时候，我们通常不需要去查看所有的这些路径。我们只需要最短的一条。那样的话，我们就排除了噪音，很快的找到问题所在。

(12:04)

有 MAT 这样一个帮我们发现内存泄露的工具是个很棒的事情。但是在一个正在运行的 App 的上下文中，我们很难像我们的用户发现泄露那样发现问题所在。我们不能要求他们在做一遍相同操作，然后留言描述，再把 70多 MB 的文件发回给我们。我们可以在后台做这个，但是并不 Cool。我们期望的是，我们能够尽早的发现泄露。比如在我们开发的时候就发现这些问题。这也是 LeakCanary 诞生的意义。

一个 Activity 有自己生命周期。你了解它是如何被创建的，如何被销毁的，你期望他会在 `onDestroy()` 函数调用后，回收掉你所有的空闲内存。如果你有一个能够检测一个对象是否被正常的回收掉了的工具，那么你就会很惊讶的喊出：“这个可能造成内存泄露！仍然没有被垃圾回收掉，它本该被回收掉的！”

`Activity` 无处不在。很多人都把 Activity 当做神级 Object 一般的存在，因为它可以操作 Services，文件系统等等。经常会发生对象泄漏的情况，如果泄漏对象还持有 context 对象，那 context 也就跟着泄漏了。

```
publicclassMyActivityextendsActivity{@OverrideprotectedvoidonDestroy(){super.onDestroy();// Instance should be GCed soon.}}
Resourcesresources=context.getResources();LayoutInflaterinflater=LayoutInflater.from(context);FilefilesDir=context.getFilesDir();InputMethodManagerinpu
(InputMethodManager)context.getSystemService(Context.INPUT_METHOD_SERVICE);
```

(13:32)

我们回过头来再看看智能标记（smart pin），我们希望知道的是当生命后期结束后，发生了什么。幸运的时，LearkCanary有一个很简单的 API。

第一步：创建 RefWatcher。给 RefWatcher 传入一个对象的实例，它会检测这个对象是否被成功释放掉。

```
publicclassExampleApplicationextendsApplication{publicstaticRefWatchergetRefWatcher(Contextcontext){ExampleApplicationapplication=
(exampleApplication)context.getApplicationContest();returnapplication.refWatcher;}privateRefWatcherrefWatcher;@OverridepublicvoidonCreate()
{super.onCreate();// Using LeakCanaryrefWatcher=LeakCanary.install(this);}}
```

第二步：监听 Activity 生命周期。然后，当 onDestroy 被调用的时候，我们传入 Activity。

```
publicActivityRefWatcher(Applicationapplication,finalRefWatcherrefWatcher)
{this.application=checkNotNull(application,"application");checkNotNull(refWatcher,"androidLeakWatcher");lifecycleCallbacks=newActivityLifecycleCallbacl
{@OverridepublicvoidonActivityDestroyed(Activityactivity){refWatcher.watch(activity);}};}publicvoidwatchActivities(){// Make sure you don't get
installed twice.stopWatchingActivities();application.registerActivityLifecycleCallbacks(lifecycleCallbacks);}
```

(14:17)

想要了解这个是怎么工作的，我得先跟大家聊聊弱引用（weak reference）。我刚才提到过静态域的变量会持有Activity 的引用。所以刚才说的“下单”按钮就会持有 `mContext` 对象，导致 Activity 无法被释放掉。这个被称作强引用（strong reference）。在垃圾回收过程中，你可以对一个对象有很多的强引用。当这些强引用的个数总和为零的时候，垃圾回收器就会释放掉它。

弱引用，就是一种不增加引用总数的持有引用方式。垃圾回收期是否决定要回收一个对象，只取决于它是否还存在强引用。所以说，如果我们将我们的 Activity 持有为弱引用，一旦我们发现弱引用持有的对象已经被销毁了，那么这个 Activity 就已经被垃圾回收器回收了。否则，那可以大概确定这个 Activity 已经被泄露了。

```
privatestaticButtonbuyNowButton;ContextmContext;
WeakReference<T>/** Treated specially by GC. */Treferent;
publicclassBaguetteActivityextendsActivity{@Overrideprotectedvoidoncreate(Bundlestate)
{super.onCreate(state);setContentView(R.layout.activity_main);}}
```

弱引用的主要目的是为了做 Cache，而且非常有用。主要就是告诉 GC，尽管我持有了这个对象，但是如果一旦没有对象在用这个对象的时候，GC 就可以在需要的时候销毁掉。

在下面的例子中，我们继承了 WeakReference：

```
finalclassKeyedWeakReferenceextendsWeakReference<Object>{publicfinalStringkey;// (1) Unique
identifierpublicfinalStringname;KeyedWeakReference(Objectreferent,Stringkey,Stringname,ReferenceQueue<Object>referenceQueue)
{super(checkNotNull(referent,"referent"),checkNotNull(referenceQueue,"referenceQueue"));this.key=checkNotNull(key,"key");this.name=checkNotNull(name,"
```

你可以看到，我们给弱引用添加了一个 Key，这个 Key 是一个唯一字符串。想法是这样的：当我们解析一个 heap dump 文件的时候，我们可以询问所有的 KeyedWeakReference 实例，然后找到对应的 Key。

首先，我们创建一个 weakReference，然后我们写入『一会儿，我需要检查弱引用』。（尽管一会儿可能就是几秒后）。

当我们调用 watch 函数的时候，其实就是发生了这些事情。

```
publicvoidwatch(ObjectwatchedReference,StringreferenceName)
{checkNotNull(watchedReference,"watchedReference");checkNotNull(referenceName,"referenceName");if(debuggerControl.isDebuggerAttached())
{return;}finallongwatchStartNanoTime=System.nanoTime();Stringkey=UUID.randomUUID().toString();retainedKeys.add(key);finalKeyedWeakReferencereference=r
```

在这一切的背后，我们调用了 System.GC — 免责声明— 我们本不应该去做这件事情。然而，这是一种告诉垃圾回收器：『Hey，垃圾回收器，现在是一个不错的清理垃圾的时机。』，然后我们再检查一遍，如果发现有些对象依然存活着，那么可能就有问题了。我们就要触发 heap dump 操作了。

(16:55)

亲手做 heap dump 是件超酷的事情。当我亲手做这些的时候，花了很多时间和功夫。我每次都是做相同的操作：下载 heap dump 文件，在内存分析工具里打开它，找到实例，然后计算最短路径。但是我很懒，我根本不想一次次的做这个。（我们都很懒对吧，因为我们是开发者啊！）

我本可以为内存分析器写一个 Eclipse 插件，但是 Eclipse 插件机制太糟糕了。 后来我灵机一动，我其实可以把某个 Eclipse 的插件，移除 UI，利用它的代码。

HAHA 是一个无UI Android 内存分析器。基本上就是把另一个人写的代码重新打包。开始的时候，我就是 fork 了一份别的代码然后移除了UI部分。两年前，有人重新fork了我的代码，然后添加了 Android 支持。又过了两年，我才发现这个人的仓储，然后我又重新打包上传到了 maven center。

我最近根据 Android Studio 修改了代码实现。代码还说的过去，还会继续维护。

(19:19)

我们有自己的库去解析 heap dump 文件，而且实现的很容易。我们打开 heap dump，加载进来，然后解析。然后我们根据 key 找到我们的引用。然后我们根据已有的 Key 去查看拥有的引用。我们拿到实例，然后得到对象图，再反向推导发现泄漏的引用。

所有的工作实际上都发生在 Android 设备上。当 LeakCanary 探测到一个 Activity 已经被销毁掉，而没有被垃圾回收器回收掉的时候，它就会强制导出一份 heap dump 文件存在磁盘上。然后开启另外一个进程去分析这个文件得到内存泄漏的结果。如果在同一进程做这件事的话，可能会在尝试分析堆内存结构的时候而发生内存不足的问题。

最后，你会得到一个通知，点击一下就会展示出详细的内存泄漏链。而且还会展示出内存泄漏的大小，你也会很明确自己解决掉这个内存泄漏后到底能够解救多少内存出来。

LeakCanary 也是支持 API 的，这样你就可以挂载内存泄漏的回调，比方说可以把内存泄漏问题传到服务器上。在 Square 我们用了 Slack 的 API，在测试阶段出现内存泄漏的时候，它就会通知我们。

```
@OverrideprotectedvoidonLeakDetected(HeapDumpheapDump,AnalysisResultresult){Stringname=classSimpleName(result.className);Stringtitle=name+" has
leaked";slackUploader.uploadHeapDumpBlocking(heapDump.headDumpFile,title,result.leakTrace.toString(),MEMORY_LEAK_CHANNEL);}
```

用上 API 以后，我们的程序崩溃率降低了 94%！简直棒呆！

(22:12)

这里有个例子，是来自 AOSB 的一个内存泄漏的代码。假设我们一个 App，包含了一个 undobar，你在点击某些按钮的时候 undobar 会消失掉。

```
publicclassMyActivityextendsActivity{@OverrideprotectedvoidonCreate(BundlesavedInstanceState)
{super.onCreate(savedInstatnceState);setContentView(R.layout.activity_main);findViewbyID(R.id.button).setOnClidkListener(newView.OnClickListener()
{@OverridepublicvoidonClick(Viewv){removeUndoBar();}});}privatevoidremoveUndoBar(){...}privatevoidcheckUndoBarGCed(ViewGroupundoBar){...}}
```

我们把所有的 View 都存起来，然后设置了一个 Layout 的 Transition 动画（1）。我们还给 undobar 增加了一个 View 进去（2）。然后我们从 parent layout 移除了 undobar（3）。

```
publicclassMyActivityextendsActivity{@OverrideprotectedvoidonCreate(BundlesavedInstanceState){...}privatevoidremoveUndoBar(){ViewGrouprootLayout=
(ViewGroup)findViewById(R.id.root);ViewGroupundoBar=(ViewGroup)findViewById(R.id.undo_bar);undoBar.setLayoutTransition(newLayoutTransition());//
(1)// (2)ViewsomeView=newView(this);undoBar.addView(someView);rootLayout.removeView(undoBar);//
(3)checkUndoBarGCed(undoBar);}privatevoidcheckUndoBarGCed(ViewGroupundoBar){...}}
```

现在看来，没有任何对象指向 undobar 了，所以说它应该被回收掉。否则的话...我们可能就遇到麻烦了。

我们用 LeakCanary 的 API 告诉系统：『Hey，现在该回收掉 undobar 了，几秒后检查下 undobar 是否被回收掉了』：

```
publicclassMyActivityextendsActivity{@OverrideprotectedvoidonCreate(BundlesavedInstanceState){...}privatevoidremoveUndoBar()
{...}privatevoidcheckUndoBarGCed(ViewGroupundoBar){RefWatcherwatcher=MyApplication.from(this).getWatcher();watcher.watch(undoBar);// (1)}}
```

我们删除了 undobar，但是... LeakCanary 好像不高兴了，它发现了内存泄露，返回的报告如下：

```
staticInputMethodManager.sInstancereferencesInputMethodManager.mCurRootViewreferencesPhoneWindow$DecorView.mAttachInforeferencesView$AttachInfo.mTreeOb
```

不难发现，静态的InputMethodManager有一个引用指向了当前的 root view：mCurRootView。mCurRootView 是当前窗口所有 View 的父容器。Root view 有一个叫做 TreeObserver 的对象。TreeObserver 是用在布局改变时候的做回调，通知监听布局改变的 listener 的，一个 View 关系树上只有一个 TreeObserver，所以你会看到它有很多的 PreDrawListener。这就意味着你每添加一个 PreDrawLisener，当布局改变的时候就会发起一次回调。

到现在为止看起来没有什么异常。但是不难发现某个 PreDrawListener持有一个 LayoutTransition$1，这种写法是 Java 表示匿名类的一种写法，意味着这是第一个在 LayoutTransition 里定义的匿名类。然后你还会看到有一个叫做 val$parent 的变量指向了我们泄露的 undobar。val$parent 意思就是这是在匿名类外声明的 final 类型的临时变量，变量名为 parent。

我们继续来看：

```
android.animation.LayoutTransition#runChangeTransition// This is the cleanup step. When we get this rendering event, we know that all of// the
appropriate animations have been set up and run. Now we can clear out the// layout
listeners.observer.addOnPreDrawListener(newViewTreeObserver.OnPreDrawListener(){publicbooleanonPreDraw()
{parent.getViewTreeObserver().removeOnPreDrawListener(this);// ... More codereturntrue;}});
```

这是 Android 源码的一部分 — LayoutTransition 是一个 Android 的类。这里的 Observer 是 ViewTreeObserver。一切都看起来没啥问题，注册了一个 ViewTreeObserver，然后马上在第一个回调里解除掉注册。这个按理说不应该出问题，但是到底发生了什么？我们来看看 getViewTreeObserver。

```
publicViewTreeObservergetViewTreeObserver(){if(mAttachInfo!=null){returnmAttachInfo.mTreeObserver;// (1)}if(mFloatingTreeObserver==null)
{mFloatingTreeObserver=newViewTreeObserver();}returnmFloatingTreeObserver;// (2)}
undoBar.setLayoutTransition(newLayoutTransition());// (3)ViewsomeView=newView(this);undoBar.addView(someView);//
(4)rootLayout.removeView(undoBar);// (5)
```

getViewTreeObserver 函数首先判断 attached状态，如果是，则返回一个 view 树，即 ViewTreeObserver。否则，返回一个临时的 ViewTreeObserver。

问题是如果我的 View 被 detach 掉了，我将会得到一个假的 ViewTreeObserver，而非一个真实的 ViewTreeObserver。你会发现，我们设置了 LayoutTransition（3），然后增加了一个 view （4）。这个触发了 addOnPreDrawListener 监听器。然后我们移除掉了 undo-bar，这就意味着 undobar 无法再访问 ViewTreeObserver了。

```
finalViewTreeObserverobserver=parent.getViewTreeObserver();// used for later cleanupif(!observer.isAlive()){// If the observer's not in a good
state, skip the transitionreturn;}publicvoidonAnimationEnd(Animatoranimator){...// layout
listeners.observer.addOnPreDrawListener(newViewTreeObserver.OnPreDrawListener(){publicbooleanonPreDraw()
{observer.removeOnPreDrawListener(this);parent.getViewTreeObserver().removeOnPreDrawListener(this);
```

它得到了一个假的 ViewTreeObserver，并且无法移除自己，因为他并不在这个假的 ViewTreeObserver 里。

这个是 Android 4年前的一次代码修改留下的问题，当时是为了修复另一个 bug，然而带来了无法避免的内存泄漏。我们也不知道何时能被修复。

(28:10)

通常来说，总是有些内存泄漏是你无法修复的。我们某些时候需要忽略掉这些无法修复的内存泄漏提醒。在 LeakCanary 里，有内置的方法去忽略无法修复的问题。

```
LAYOUT_TRANSITION(SDK_INT>=ICE_CREAM_SANDWICH&&SDK_INT<=LOLLIPOP_MR1){@Overridevoidadd(ExcludedRefs.Builderexcluded){// LayoutTransition leaks
parent ViewGroup through ViewTreeObserver.OnPreDrawListener// When triggered, this leak stays until the window is destroyed.// Tracked here:
https://code.google.com/p/android/issues/detail?id=171830excluded.instanceField("android.animation.LayoutTransition$1","val$parent");}}
```

我想要重申一下，LeakCanary 只是一个开发工具。不要将它用到生产环境中。一旦有内存泄漏，就会展示一个通知给用户，这一定不是用户想看到的。

我们即便用上了 LeakCanary 依然有内存溢出的错误出现。我们的内存泄露依然有多个。有没有办法改变这些呢?

(29:14)

```
publicclassOomExceptionHandlerimplementsThread.UncaughtExceptionHandler{privatefinalThread.UncaughtExceptionHandlerdefaultHandler;privatefinalContext
{...}@OverridepublicvoidUncaughtException(Threadthread,Throwableex){if(containsOom(ex)){FileheapDumpFile=newFile(context.getFilesDir(),"out-of-
memory.hprof");try{Debug.dumpHprofData(heapDumpFile.getAbsolutePath());}catch(Throwableignored)
{}}defaultHandler.uncaughtException(thread,ex);}privatebooleancontainsOom(Throwableex){...}}
```

这是一个 Thread.UncaughtExceptionHandler，你可以将线程崩溃委托给它，它会导出 heap dump 文件，并且在另一个进程里分析内存泄漏情况。

有了这个以后，我们就能做一些好玩儿的事情了，比如：列出所有的应该被销毁却依然在内存里存活的 Activity，然后列出所有的 Detached View。我们可以依此来为泄漏的内存按重要性排序。

我实际上已经有一个很简单的 Demo 了，是我在飞机上写的。还没有发布，因为还有些问题，最严重的问题是没有足够的内存去解析 heap dump 文件。想要修复这个问题，得想想别的办法。比如采用 stream 的方法去加载文件等等。

(31:50)

Q: LeakCanary 能用于 Kotlin 开发的 App?

PY: 我不知道，但是应该是可以的，毕竟到最后他们都是字节码，而且 Kotlin 也有引用。

Q:你们是在 Debug 版本一直开启 LeakCanary 么？还是只在最后的某些版本开启做做测试

PY： 不同的人有不同的方法，我们通常是一直都开着的。