虚拟机内存回收

* **垃圾回收GC（Garbage Collection）**

参考：<https://www.jianshu.com/p/214e42fc0d37>

当一些被占用的内存不再需要时，就应该予以释放，以让出空间，这种存储资源管理，称为垃圾回收。

* **回收哪部分内存**

根据虚拟机内存结构来看，虚拟机的各个内存区域中线程计数器、虚拟机栈、本地方法栈，其中程序计数器所占内存可以忽略不计，栈中的栈帧在线程执行的过程中进行着规范的入栈和出栈的操作并且栈帧的大小在类加载后就已经确定，最重要的是这三个区域随线程生而生，随线程灭而灭，所以这三块线程私有内存的回收可以不做过多考虑。

而Java堆和方法区不一样，它们只有在运行期间才知道需要加载哪些类并且创建哪些对象，对象的大小也只有在运行期间才能确定其具体大小，所以这两块区域的内存分配和回收都是动态的，垃圾回收器所关注的也是这部分内存。

* **什么时候进行GC操作**

1. 当应用程序空闲时进行回收。因为垃圾回收也是单独的在一个线程中进行的，叫GC线程，也是虚拟机中的守护线程，它的线程级别最低，所以当应用忙时，GC线程不会调用，除了Java堆内存不足的情况。
2. Java堆内存不足时，GC会被调用。当新建对象堆内存不足时，虚拟机会强制调用GC线程进行内存回收并用于新的分配。假如一次GC不能满足新对象的内存分配，则虚拟机会在调用2次GC操作，假如还不能满足，那么就会抛出 “OutOfMemoryError”异常，应用将停止。

注意：手动调用GC，只是告诉JVM尽快GC一次，并不保证GC一定会执行。

* **GC操作会造成什么影响**

当进行GC操作时，除了GC线程其余的所有线程都会被停止，假如GC时间过长会使程序出现明显的卡顿。所以我们要做的就是尽可能的缩短GC时间。

* **如何回收内存**

垃圾收集器回收内存之前先要判断内存中哪些对象有用，哪些对象没用（即不在被任何途径使用的对象）。然后回收这些没用的对象。

* **Java堆的回收**

垃圾回收的重点就是堆内的实例对象。

* **对象回收收判定**

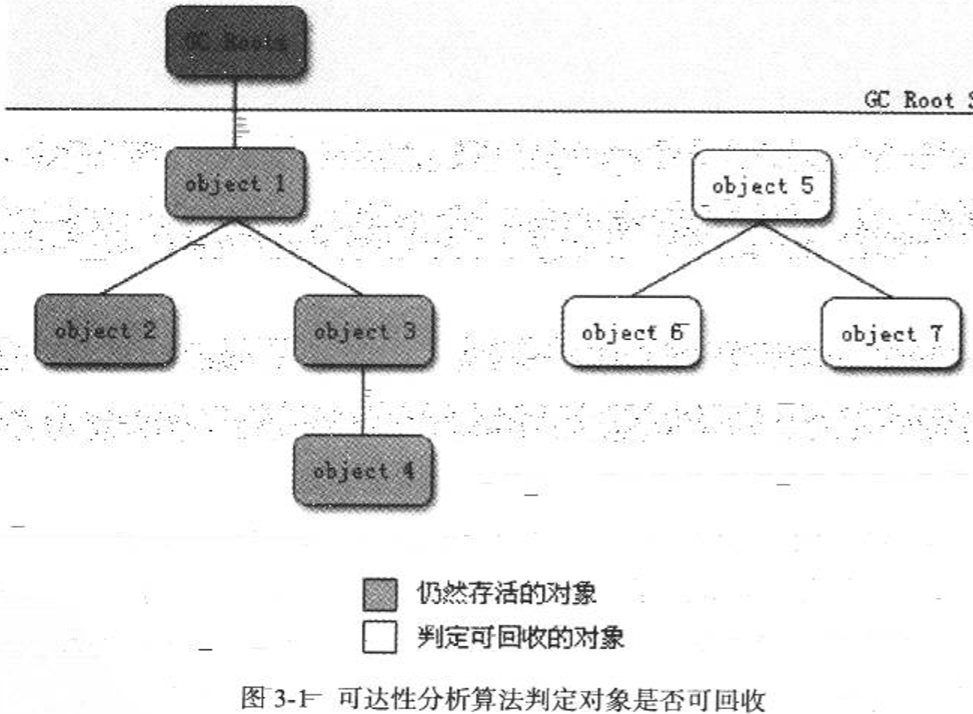
对象的回收判定有两种：引用计数算法、可达性分析算法。

**引用计数算法：**就是给对象中添加一个引用计数器，每当一个别的地方引用它时，计数器就加1；当引用失效时，计数器就减一；当计数器为0时，说明这个对象永远不可能再被使用，这时候就可以通知虚拟机回收这个对象。

优点：实现简单，效率高。

缺点：很难解决对象之间相互循环引用的问题。

**可达性分析算法：**就是通过一些列“GC Roots”对象为起始点，然后向下搜索，所经过的路径称为引用链，当对象没有任何路径可以到达“GC Roots”时，那么这个对象被称为不可用对象，所以可以被判定为可回收对象。



在Java语言中可作为“GC Roots”对象的包括以下几种：

1. 虚拟机栈（栈帧的本地变量表）中的引用的对象。
2. 方法区中静态属性引用的对象。
3. 方法区中常量引用的对象。
4. 本地方法栈中JNI（即一般说的Native方法）引用的对象。

* **引用分类**

在JDK 1.2之后，Java对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference），这四种引用强度一次递减：

**强引用：**是指在程序中普遍存在的引用，类似“Object o = new Object()”其中o就是强引用。只要引用存在虚拟机就不会回收被引用的对象。

**软引用：**是指那些还有用但是非必须存在的对象。软引用关联的对象在内存不足时，会被列进回收范围之内进行最后一次回收，如果这次回收还没有足够的内存，虚拟机会抛出OOM异常，如果内存充足永远不会回收软引用关联对象。JDK 1.2提供了SoftReference来实现软引用。

**弱引用：**也是描述那些非必须的对象，但是它比软引用更弱一些，他只能生存到下一次垃圾回收之前，并且不论当前内存是否充足，弱引用关联的对象都会被回收。JDK 1.2后使用WeakReference类来实现弱引用。

**虚引用：**它是最弱的一种引用关系。一个对象是否有虚引用存在完全不会对其生命周期有任何影响，也无法用虚引用获取一个对象的实例，虚引用关联的对象在任何时候都可能被垃圾回收器回收。虚引用可以用来查看对象是否被回收。

* **方法区的回收**

方法区的回收“性价比”很低，一般的Java堆的一次垃圾回收大约能回收70%—90%的空间，而方法区要远远低于这个范围。方法区的回收主要包括：废弃的常量和无用的类。

**废弃的常量的回收：**假如一个字符串“abc”已经进入到常量池中，但当前系统没有任何一个String对象引用常量池中的“abc”常量，也没有其他地方引用这个字面量，如果这时候发生内存回收，而且必要的话，这个“abc”常量就会被清理出常量池。常量池中其他的类（接口）、方法、字段的符号引用也与此类似。

**无用的类回收：**判断一个无用类的条件非常苛刻，

1. 该类所有的实例都已经被回收，也就是Java堆中不存在内核这个类的实例。
2. 加载该类的ClassLoader已经被回收。
3. 该类对应的java.lang.Class对象没有任何地方引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足以上三个条件的无用类进行回收，这里说的仅仅是“可以”，并不是一定回收。

* **内存泄漏**

Android进程中堆内存里的对象回收策略是通过可达性分析算法来进行的。上面已经讲解了，当GCRoot引用链上的某些对象已经无用，此时这些无用对象是不能通过虚拟机GC（垃圾回收）回收掉的，但是它们还占用着堆内存，这种无用对象占用着内存，但是无用对象还不能被GC回收的状况叫做内存泄露。

GC Root

A

B

如上图，当B对象无用时，但是B对象依然存在在引用链上，此时GC是无法回收B对象的，此时B对象就属于内存泄漏的对象。

注意，GCRoot引用链的路径是单向的，内存泄露的对象指的是GCRoot引用链单向路径上的对象，并不包括，指向引用链上对象的对象。如下：

GC Root

C

A

B

上面图中，A、B对象在一条GC引用链上，不可能会被GC回收的，这里注意虽然C对象指向了GCRoot引用链上的对象，但是C对象并不属于GCRoot引用链上的对象，当虚拟机进行GC的时候，依然会回收C对象，所以C对象并不会早场内存泄漏。

内存泄露的主要影响就是会使堆内存的可用空间越来越来小，最终会引起内存溢出。

* **内存分析工具**

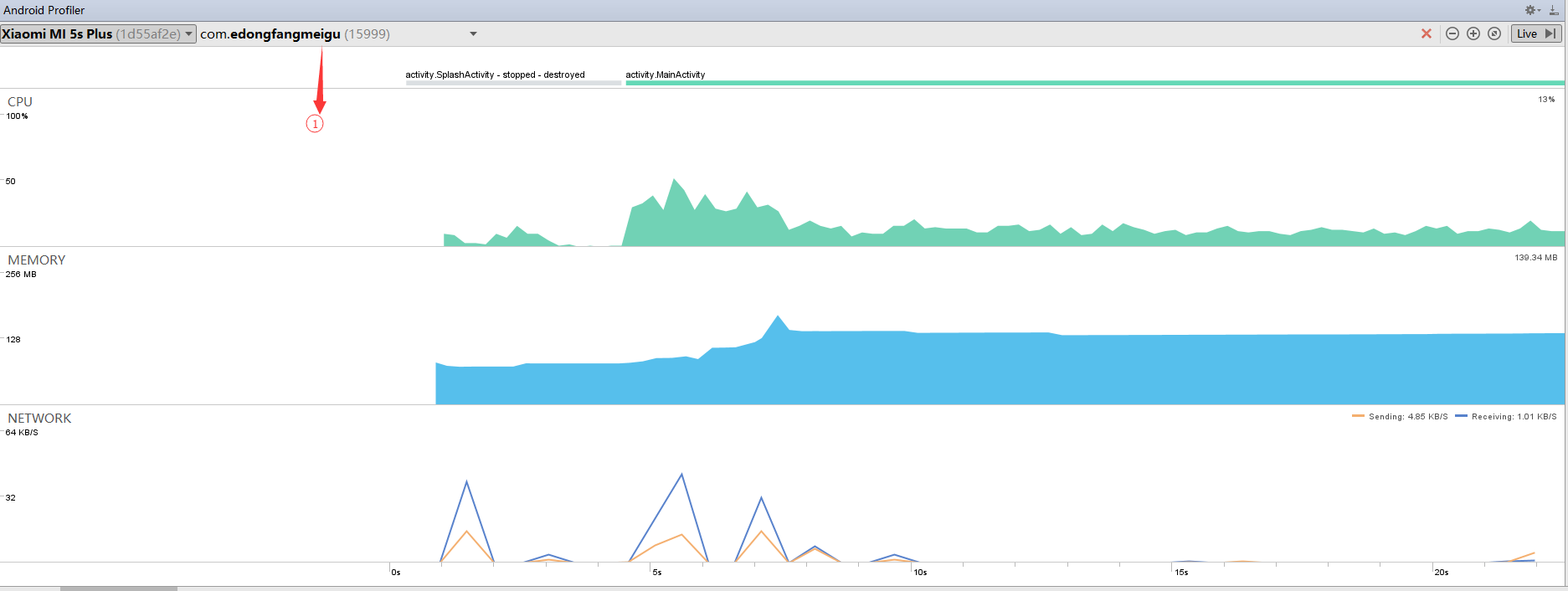
分析内存主要是分析堆内存，通过内存分析工具，我们可以知道，堆内存的使用情况,包括分配了什么类型的对象，以及这些对象的个数，所占内存大小，引用它们的对象以及它们引用的对象等等，最重要的是分析内存泄漏情况。

**Android Profiler工具**

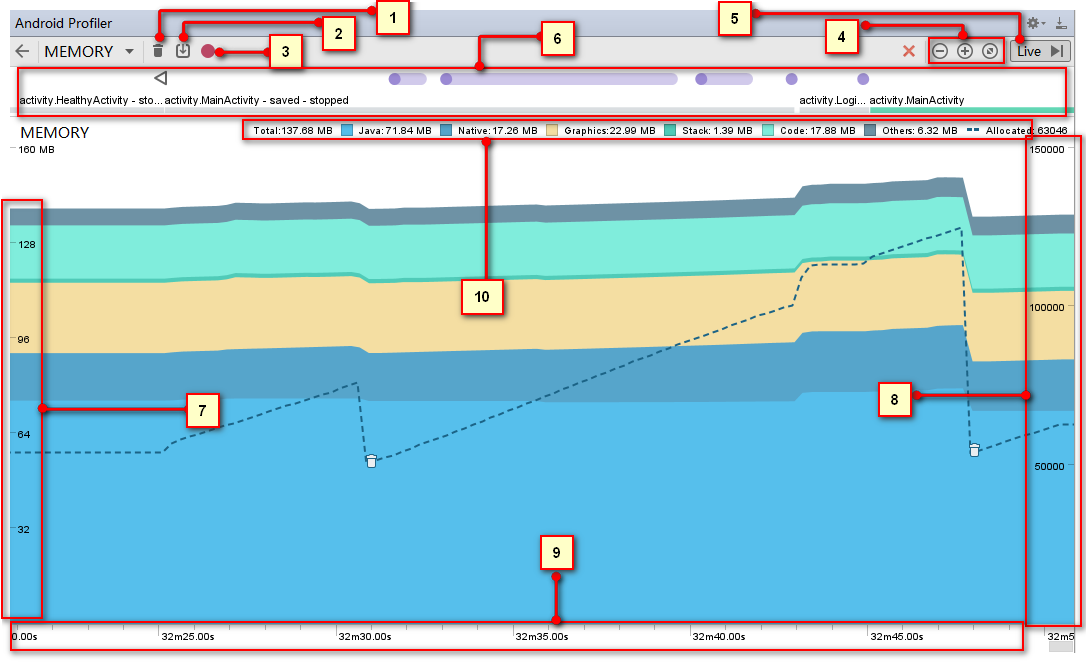
参考：<https://developer.android.google.cn/studio/profile/memory-profiler>

https://www.jianshu.com/p/e75680772375

Android Profiler可以监控内存，CPU，网络。这里值介绍检测内存的使用。我们可以通过View-->Tool Windows-->Android Profiler打开。打开后，如下图：



单击位置①可以选择项目，然后单击下面MEMORY这一行进入内存监测时间表页面。如下如：



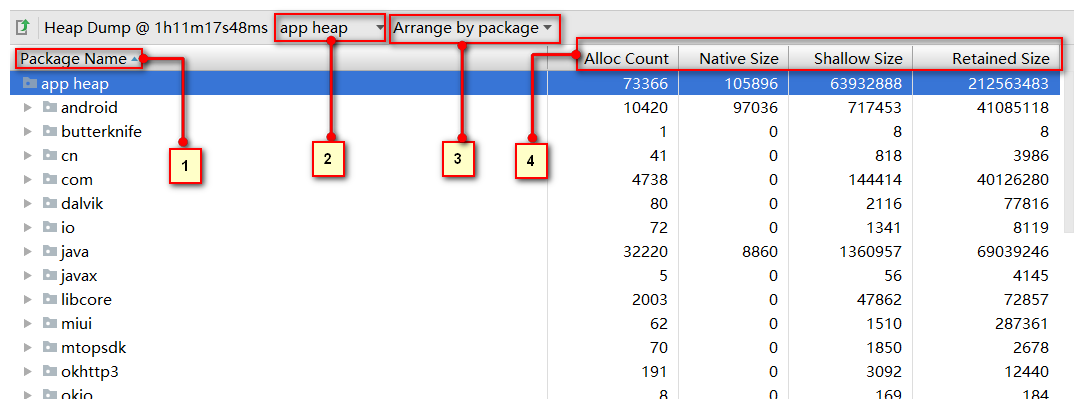
1. 强制进程GC按钮。点击依次在图中的虚线上就会出现一个垃圾桶的图标。
2. 生成当前时间点的内存快照。
3. 选取某一时间段内存分配情况，选取的开始和停止按钮。
4. 依次是缩小、方法、恢复底部X轴的时间线。
5. 实时分配内存的暂停和开始按钮。
6. App中产生事件的记录（跳转页面、滑动列表等）。
7. 每个类别内存使用量，以及总使用量坐标。
8. App产生对象个数，即时间表中的虚线代表数量变化曲线。
9. 时间线坐标。
10. 表示内存类别，以及每个类别的使用量：

**Java :** java代码分配的内存  
 **Native:**c/c++代码分配的内存。  
 **Graphics:**图像缓存等，包括GL surfaces, GL textures等.  
 **Stack:**栈内存（包括java和c/c++）  
 **Code:**代码的内存分配（例如代码，资源，libs等等）  
 **Other:**这个是连系统都不知道是什么类型的内存，就是other。  
 **Allocated:** jave分配的对象个数 (在Android7.1和以下的设备，这个计数是在设备连接后开始，所以并不是app启动时候的计数。Android8.0或更高，在系统里面内置的Profiler工具，所以无论什么时候连接，都是app启动时候的计数)。

上面的介绍只是这个工具的初始化界面做了简单的介绍，通过这个界面我们可以看出通过操作app时内存使用的大小以及内存中生成对象的个数。想要更详细的分析内存信息就需要通过按钮②和按钮③来生成某一个时间点或者某一段时间的内存快照。

**分析某一个时间点的内存快照**

通过点击按钮②即可获取当前时间点的内存使用情况，如下如：

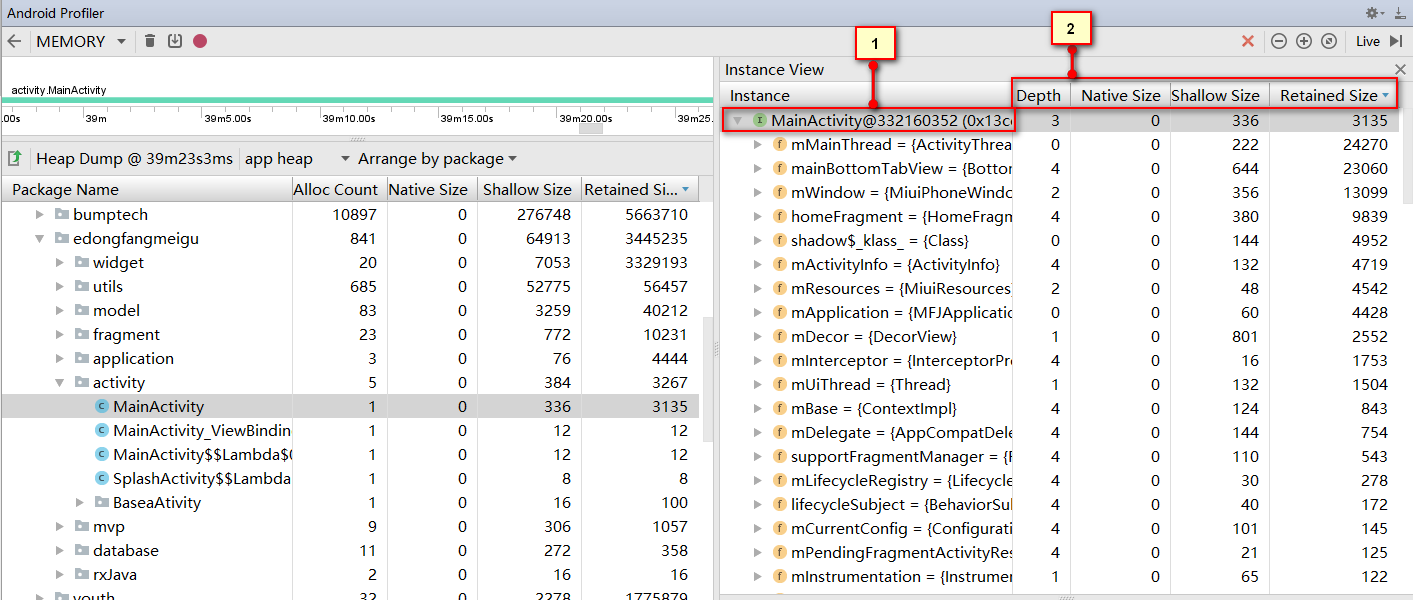


1. 通过name字段排序按钮，按a-z正序或者倒序。
2. 有4个选项，其中app heap是查看当前堆内存分配情况。
3. 将产生的对象的所属类型（即类名）进行分组，这里选择按包名分组比较直观。
4. 包含四个值：  
    **Alloc Count :** 相应类在内存中一共存在的对象个数。  
    **Native Size :** 相应类的所有对象在c/c++层内存大小(bytes)  
    **Shallow Size :** 相应类的所有对象在java层占用内存大小(bytes)。

Shallow Size = 对象个数 \* 单个对象所占内存大小

**Retained Size :** 回收这个类型的所有对象能回收的内存大小。这个值与Shallow Size 不 一致的原因是相应对象里还引用着其它对象，所以回收一个对象不单单回收这 个对象所占内存，还需要回收这个对象引用的其它对象所占内存。

点击其中包下的一个类名，即可打开Instance View，这个界面中展示了被点击的类的所有实例：



1. 表示左侧点击的MainActivity类的实例。点击这个实例展开可以查看它内部的变量以及父类的变量（不包括static修饰的变量，因为static修饰的变量不属于类实例），以及变量指向的对象，然后可以继续展开这些对象，查看这些对象内部的变量，依次类推，可以查看很详细的内容。如果这个MainActivity类的实例作为一个GCRoot根，然后展开这个实例查看它内部变量的引用的对象，然后继续查看它引用对象的对象，这样依次循环展开的路径就形成了一条引用链。
2. 包含四个值：

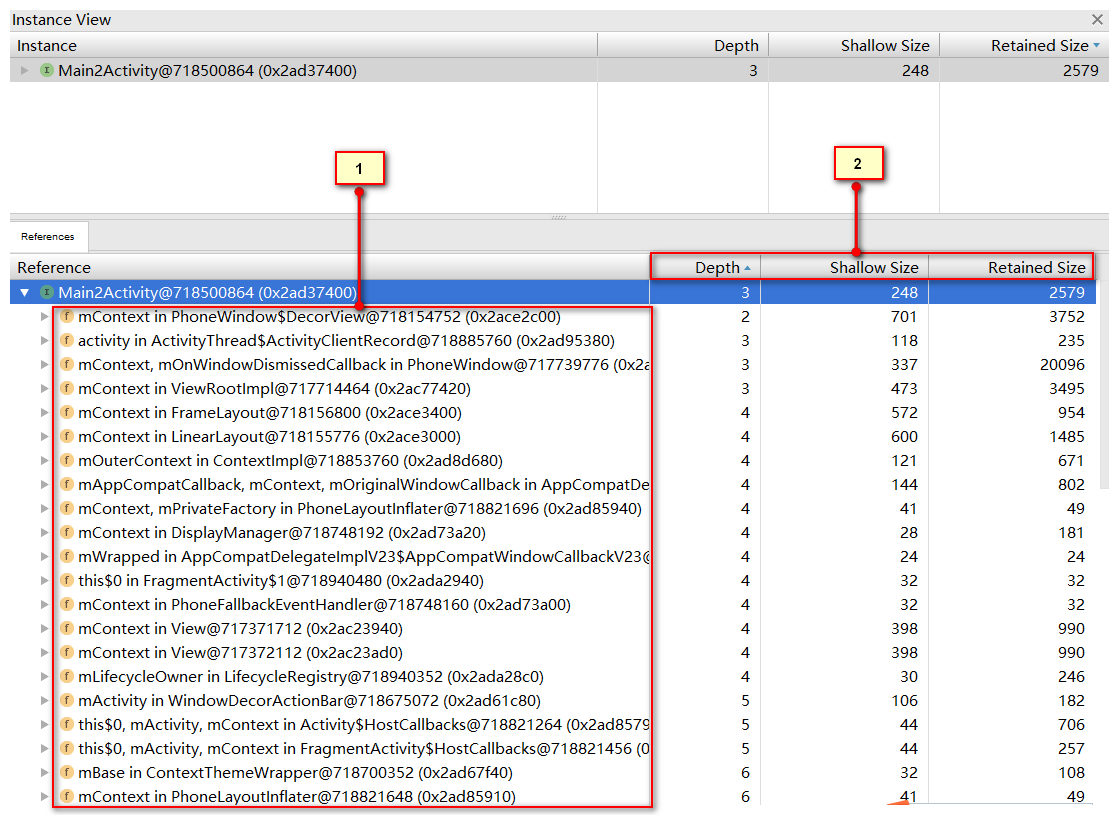
**Depth :** GC Root 到该对象的引用链路的最短步数

**Native Size :** 这个实例所占native层内存大小(bytes)。

**Shallow Size :** 这个实例在java堆中占用内存大小(bytes)。

**Retained Size :** 回收这个实例能回收的内存大小。

点击Instance界面下的实例，可以打开Reference界面，这个界面展示了所有指向Instance界面中被选中的实例的引用：



1. 这里列列出了指向上面Main2Activity类实例的所有引用，它们的格式是：

引用名称 + in + 引用所在类

in前面是指向这个实例的引用名称，in后面是这个引用所在的类。

这里的引用是可以展开的，展开后可以看到这个引用所在类的实例的引用列表。例如第一行指向Main2Activity类实例的mContext引用，这个引用是存在PhoneWindow类中，当点击小三角展开这个mContext引用后，展示的是指向PhoneWindow类实例的所有引用，当然这些引用还可以展开，以此类推，不断展开，会不断的显示指向当前引用所在实例的所有引用。

1. 同Instance View界面。

**分析某一个时间段的内存快照**

Android Profiler可以分析一段时间内的内存分配情况，注意生成的这一段时间内存快照展示的是**从起始点到结束点这段时间内，内存中新增加的对象的信息**。分析一段时间内的内存快照需要用到下图中红色按钮：

IMG_256

点击图中红色按钮，设置时间起始点，此时红色按钮会变灰，当再次点击会设置结束结束时间点，并且生成这一段时间内的内存分配情况。例如，我们在app首页，然后点击红色按钮，接着在app首页，点击按钮跳转到另一个页面，然后再次点击这个按钮，就会生成从首页跳转到另一个页面后，新生成的对象信息，并不会展示目前内存中之前已经存在的使用信息，接下来的界面和操作和上面分析的生成某一个时间点内存分配情况是一样的。按照这个思路就可以分析内存分配情况了。