Android (/tags/#Android) Performance (/tags/#Performance) Android Performance Patterns (/tags/#Android Performance Patterns)

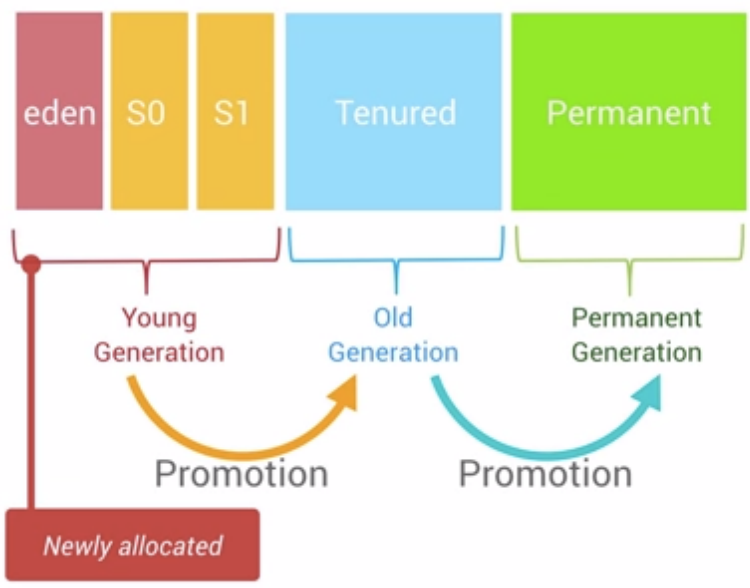
Android Performance Patterns——Memory Performance

Android Performance Patterns系列学习思考和实践笔记

Posted by Cheson on March 15, 2017

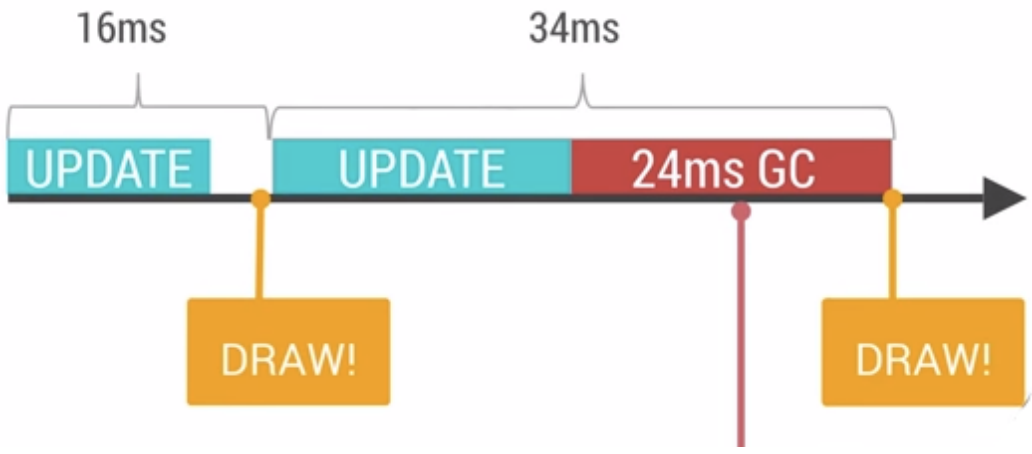
0. Garbage Collection in Android

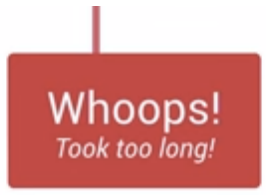
Android的内存管理是一个三级Generation的模型，最近分配的对象存放在Young Generation中，在该区域停留达到一定程度之后会被转移到Old Generation中，最后会被存放到Permanent Generation中。



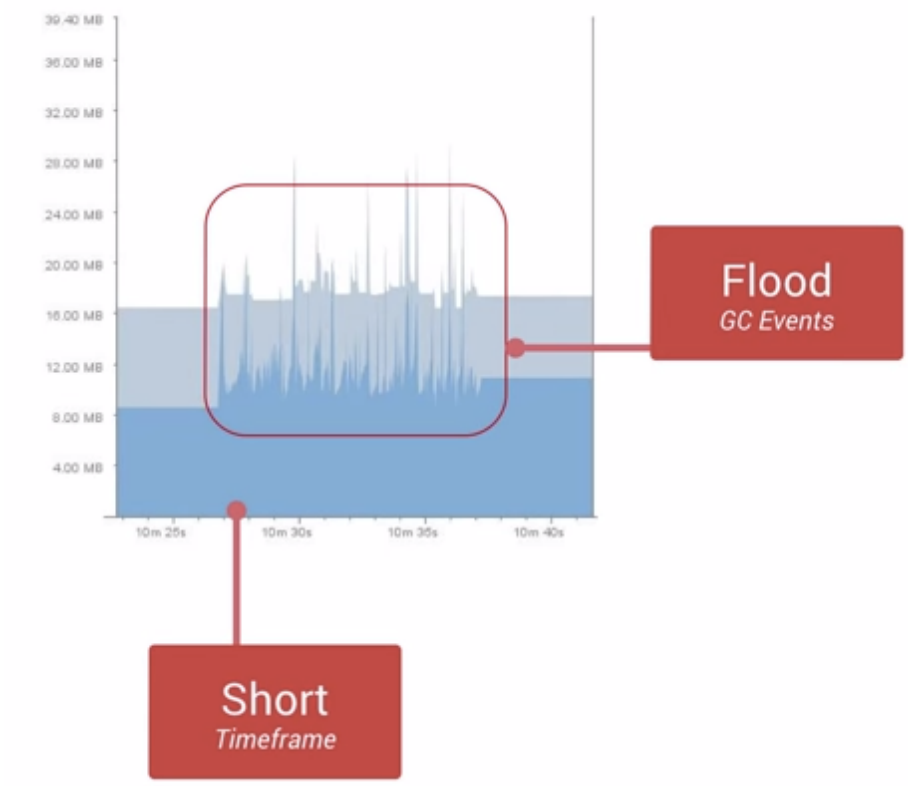
每个级别的区域中都有一定的空间限制，如果分配的对象空间达到阈值时就会触发GC操作。在不同的Generation中做GC操作的速度是不同的，Young Generation中每次GC操作时间最短，Old Generation其次，Permanent Generation最慢。当然GC操作的时间和遍历的对象数量也是相关的。GC是一个阻塞的操作，在进行GC时，其他的线程都是处于暂停状态的。这个知识点和异常分析有很大的关系，通常在分析ANR和SWT的问题时，会优先看callstack中suspend状态的thread，通常这些thread是比较可疑的，卡在native调用或者binder调用，又或者是发生了死锁卡住。然后还有写suspend的情况却不属于异常，其中一种就是恰好在做GC时，其他thread都会被suspend；还有种情况是arm在dumpstacktrace时也会把thread都suspend。

单个的GC操作并不耗时，但是频繁的GC就会对性能产生一定影响了。例如对UI Performance的影响，一帧的渲染需要在16ms内完成才能保证有流程的用户体验，如果频繁的GC占用了太多的CPU时间，导致无法及时完成这一帧的渲染，性能问题就随之而来了。这个就是GC这面双刃剑带来的危害，当然正常情况下剑总是冲敌人的，下面来看下什么情况下会对自己造成伤害。





导致GC频繁发生的原因可能有两个：1、内存抖动，内存抖动是在短时间内分配和回收大量对象时出现的现象，主要是在循环中创建对象的代码导致，频繁分配大量对象就可能会触发GC；2、瞬间产生大量的对象，会导致Young Generatation区域中空间达到阈值而触发GC。



分享一篇对GC讲解的比较通俗易懂的文章Android GC那些事 (https://zhuanlan.zhihu.com/p/20282779?columnSlug=magilu)

> *回收算法*
> *标记回收算法（**Mark and Sweep GC**）*
> *从"GC Roots"集合开始，将内存整个遍历一次，保留所有可以被GC Roots直接或间接引用到的对象，而剩下的对象都当作垃圾对待并回收，这个算法需要中断进程内其它组件的执行并且可能产生内存碎片。*
> *复制算法 **(Copying**）*
> *将现有的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。*
> *标记-压缩算法 **(Mark-Compact)***
> *先需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。*
> *分代*
> *将所有的新建对象都放入称为年轻代的内存区域，年轻代的特点是对象会很快回收，因此，在年轻代就选择效率较高的复制算法。当一个对象经过几次回收后依然存活，对象就会被放入称为老生代的内存空间。对于新生代适用于复制算法，而对于老年代则采取标记-压缩算法。*

看完这一段也就可以理解为何Young Generation中执行GC会比较快，因为其算法是用Copying，是一种以空间换时间的做法。

当然老罗的博客是不可缺少的一手资料Dalvik虚拟机垃圾收集（GC）过程分析 (http://blog.csdn.net/luoshengyang/article/details/41822747)

关于GC的算法公司同事做了更详细的分析，参考java虚拟机回收算法 (http://wiki.huaqin.com:8090/pages/viewpage.action?pageId=29409597)

1. Memory Churn & Performance

1.1 Typical Case

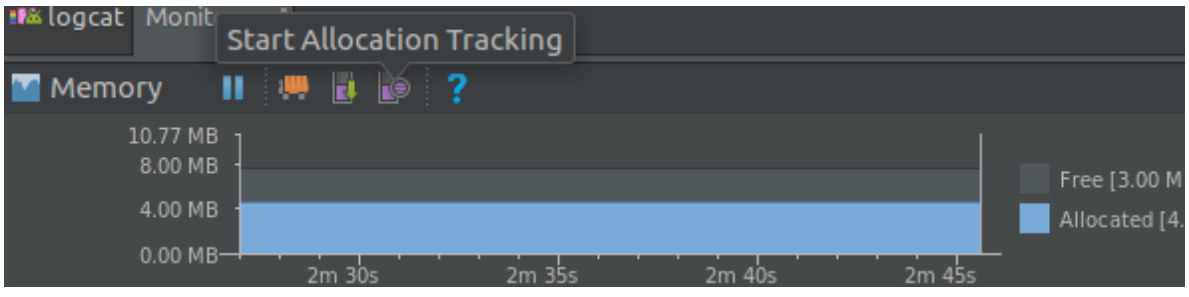
这一节中就来详细看下内存抖动的情况，以一个非常简单的案例开场，来看这样一段代码

```
public class MainActivity extends AppCompatActivity {

    private Context mContext = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mContext = getApplicationContext();

        testMemoryChurn();

    }

    private void testMemoryChurn() {
        for(int i = 0; i < 1000; i++) {
            createBitmap();
        }
    }

    private void createBitmap() {
        Bitmap bitmap = BitmapFactory.decodeResource(mContext.getResources(),R.drawable.an
    }
}
```

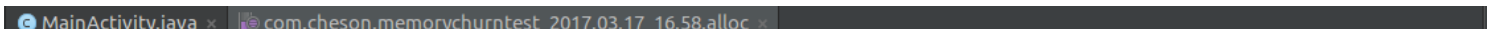
1.2 Memory Monitor & Allocation Tracker

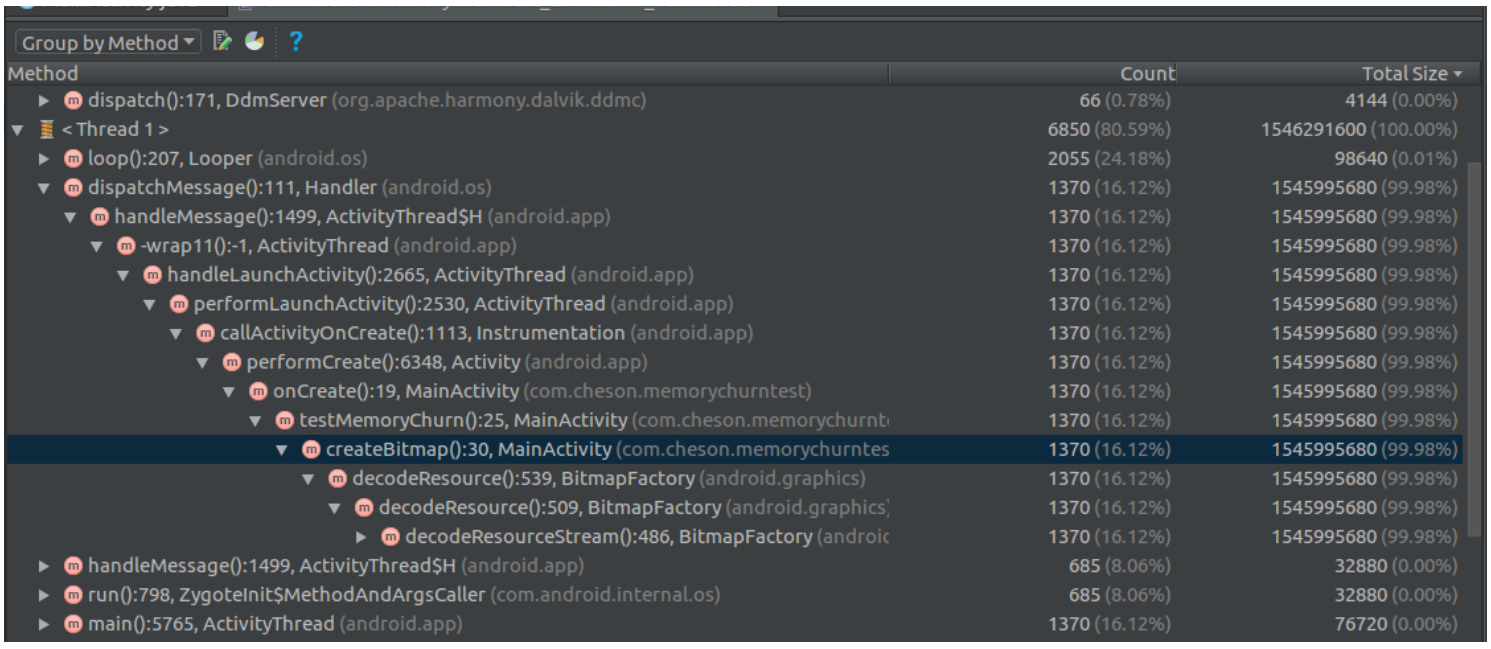
这段程序跑起来后在Android Studio中用Memory Monitor工具来查看内存的情况，出现了非常明显的内存抖动情况，其引发的原因就是for循环中的分配对象，每次循环结束后都会被回收，导致内存不停的变化。

在实际调试一个应用时，用Memory Monitor可以监视到内存的情况，但是无法定位到代码，如果看到有内存频繁或者长时间的内存抖动情况，如何去确定代码中的根因呢。这里需要用到Android Monitor中的另一个工具：Allocation Tracker。此工具的使用是需要手动抓取一段时间内的内存分配情况的，在操作之前点击下图中的Start Allocation Tracking，结束操作之后再点击stop。



例如抓取到的内存抖动的一段内存情况，可以看到thread-1占了几乎100%的内存，分配的size很大，顺藤摸瓜就能找到问题的代码在于decodeResource方法中。





这个案例非常简单，但也是此类问题非常典型的案例，通常在for循环中分配内存时就会出现这个问题，另一个可能导致内存抖动的罪魁祸首是在onDraw方法中分配内存，当view被频繁重绘的时候也会导致内存抖动。这里的内存抖动问题解决起来就很简单了，把对象分配的动作放在循环外就可以。

1.3 What is Memory Churn Exactly

直观了解了内存抖动之后来思考这样一个问题，内存抖动为什么会引起GC呢？要思考这个问题之前先要明白GC的一个原理，什么情况下会触发GC。在前一节介绍GC时的老罗博客的参考资料中给出了GC的4中触发条件

> ***GC_FOR_MALLOC:*** *表示是在堆上分配对象时内存不足触发的GC。*
> ***GC_CONCURRENT:*** *表示是在已分配内存达到一定量之后触发的GC。*
> ***GC_EXPLICIT:*** *表示是应用程序调用System.gc、VMRuntime.gc接口或者收到SIGUSR1信号时触发的GC。*
> ***GC_BEFORE_OOM:*** *表示是在准备抛OOM异常之前进行的最后努力而触发的GC。*
> *实际上，GC_FOR_MALLOC、GC_CONCURRENT和GC_BEFORE_OOM三种类型的GC都是在分配对象的过程触发的。*

在这个背景知识的基础上再来理解内存抖动的情况，以上面案例中的代码来思考，在循环内部分配内存后，当结束一次循环时，局部变量虽然已经失效了，但是其内存空间并没有立即被回收（虽然Java可以通过GC自动回收内存，但也不是即时的），即使在使用完对象之后立即将其置为null，也不会立即回收其所占的内存空间，还是需要等待系统的GC操作。所以我们看到的内存抖动现象其实不应该被称之为频繁引起GC的原因，而应该理解为频繁分配对象的操作，导致了Young Generation中的剩余空间频繁达到阈值（内存抖动截图中的波峰）而触发GC，然后可用内存又降下来（波谷），产生了内存抖动的现象。

1.4 What Can Help Us

对内存抖动和GC的含义做了了解之后就需要找寻解决方法了，非常幸运的时前人的智慧已经为我们铺好了道路。针对内存抖动的这种特性，我们需要找寻一种方案来解决这种频繁分配生命周期很短的对象的问题。于是乎“缓存池”的概念起了作用，这里来介绍以下符合此需求的对象池模式。

1.4.1 Object Pool

对象池模式是在Java 23种设计以外的一种模式，其设计的思想大致如下：对象池会预先申请一组对象，当客户端需要使用时向对象池申请，使用完之后可以手动或者自动归还给对象池，以此来减少频繁的动态分配和回收空间的动作。那么问题来了，为什么向对象池申请对象（赋值操作）会比系统动态分配具有更好的效果呢？这是有科学根据的，来看下面的表格

运算操作	示例	标准化时间

运算操作	示例	标准化时间
本地赋值	i = n	1.0
实例赋值	this.i = n	1.2
方法调用	func()	5.9
新建对象	new Object()	980
新建数组	new int[10]	3100

新建对象的耗时是980个标准时间，根据Java语言的特性，一个对象的生命周期包括了新建，使用和销毁，当新建对象时，其构造函数会被显式或者隐式的调用到，而且调用到的不仅仅是一个构造方法，而是一个构造方法链

> *在任何情况下，构造一个类的实例时，将会调用沿着继承链的所有父类的构造方法。通俗的说就是在构造一个子类的对象时，子类构造方法会在完成自己的任务之前先调用父类的构造方法，如果父类又继承自其他类，那么父类在完成自己的任务之前也会先调用他的父类的构造方法，一直持续，直到最后一个类的构造方法被调用，这就是构造方法链。*

所以新建对象的耗时是如此的惊人。而对象的销毁是通过我们反复提及的GC由系统自动来回收的，GC的弊端也是非常明显的，就是会阻塞到其他线程。因此，一个对象实际有用的耗时就在于被使用的这段时间，而引入对象池模式也就是最大程度的省去了新建和销毁的过程，充分利用了对象的有效生命周期。从另一个角度看也是避免了反复给相同的对象分配空间而导致的频繁GC问题。

当然如此优秀的一个技术也有其自身的局限，当使用对象池时，对象的生命周期变得更加复杂了，因为对象的产生和归还都不会涉及到真正的空间分配和回收，而是由对象池统一管理，增加了管理的复杂度。另一个方面，对象池其实也是一种以空间换时间的做法，当然牺牲的空间不能太大，还是要在一个应用的堆内存中保持可接受的程度。而且在使用对象池时，被优化的对象的需要有一定的“重要性”，极端点说为一个只会分配一次的对象来用对象池实现，那就得不偿失了。

自己写了一个对象池的demo，请参考github上的工程(ObjecytPoolTest (https://github.com/chendongqi/ObjectPoolTest))。当然，原始的对象池模式的源码和demo，apache已经帮我们实现封装了，可以直接下载jar包导入到工程中使用，也可以去下载apache的源码来研究其实现原理。apache对对象池的实现也在迭代更新中，区别在于使用了不同的对象管理的数据结构。这里概括的介绍下一个对象池模式实现的基本思想：1、定义一种大对象（可以是现有的，也可以自定义封装，对象池主要是针对重量级对象）；2、定义一个对象生产的接口并实现对对象生命周期进行操作的方法；3、定义一个对象池操作的接口；4、继承3中的接口，以一种特定的数据结构实现对象池管理。5、需要使用对象时，从对象池中借，用完之后归还到对象池中。

2. Performance Cost of Memory Leaks

这一章中介绍Android中的内存泄露，包括了内存泄露的产生，分析和优化以及实际案例分析和工具。要理解为什么会产生内存泄露，就必须先知道在Android中内存是如何管理的。

2.1 Stack&Heap

在前面介绍了Android中将内存分为了3个Generations，这是超越于Java的一种创新。而最基本对内存的分配策略还是沿用了Java的套路，可以细分为5种：Register（这里不做讨论）、Heap、Stack、静态变量区、常量区。下面一张表格将清楚的展现不同区域的分工和作用。

	Heap	**Stack**	静态变量区	常量区

	Heap	Stack	静态变量区	常量区
JVM中的作用	内存数据区	内存指令区	存放静态变量	存放常量
存放数据	对象	基本数据类型、指令代码、对象引用	static变量（全局变量）	字符串常量、基本类型常量

Heap（堆）：堆中存放的是动态分配的对象，以new系列方式产生的对象都会放在这里，也是灵活管理代码中动态创建出来的对象的区域。也正因为这里的对象都是动态创建的，所以其生命周期都是不确定的，因此前面聊了很多的GC就是回收这里不再使用的对象。

Stack（栈）：栈中主要存放的是基本数据类型（int、char、long、byte、float、boolean等）、指令代码（函数方法）和对象的引用。这里说明下，当一个对象A被动态创建时，会分配两处内存，在堆中开辟一片空间存放A的成员变量，在堆中会创建一个A的引用来指向堆中的地址。另外，栈中存放的对象是方法中的局部变量，其生命周期是确定的。所以栈里的数据的两个特点就呼之欲出了，数据形式确定，数据生命周期确定。由于其数据的格式确定、大小确定，为栈里的数据分配空间的算法内置于处理器的指令集中，所以栈操作的速度非常快，仅次于寄存器。由于其生命周期确定，所以在一个方法结束时，其中的变量也就被销毁，空间被释放掉了，这个对内存利用的效率也要比堆高。而栈的局限在于空间较小。栈还有另外一个特性是可共享，如何理解共享呢？

```
private int test() {
  int a = 1;
  int b = 1;
  return a+b;
}
```

以上面这段代码，在编译阶段会在栈中分配内存，编译器在处理 `a = 1` 时会给a分配一个引用对象，然后在栈中查找是否有1这个常量，没有找到则分配一块常量空间来放1，让a引用指向1这个常量；在处理 `b = 1` 时，先给b分配一个引用对象，然后找到已经有1这个常量了，则让b也指向这个常量，这里就是共享。那么另一个问题随之而来，a和b引用都指向了同一片内存空间，如果在后面执行 `a = 2` 的赋值操作，那么从b中读到的值是否也会变成2呢？现象大家当然知道是不会产生b=2这个结果的，具体原因是：1享用的是一片常量空间，当操作 `a = 2` 时，会先去栈中寻找是否有2这个常量，如果已经给2这个常量分配给内存了，则直接让a引用转而指向2，如果不存在则新分配一片内存给2这个常量，然后让a指向它，此时a和b指向的内存就不同了。

静态变量区：该区域是用来分配给静态变量的，也就是static修饰的成员变量。

常量区：该区域用来分配给常量，如上面例子中提到的整形常量1。在网上看到一个挺有意思的题目：`String str = new String("xyz");`，问这段代码分配了几个对象。答案是一个或者两个，当常量区存在”xyz”时是一个，不存在时则是两个。

静态变量区和常量区通常也合称为静态存储区，其所属的大区域应该也是在栈中（从其共享、数据特性、分配原理推测，暂未得到考究）。下面一段程序来总结各种数据对应的分配规则。

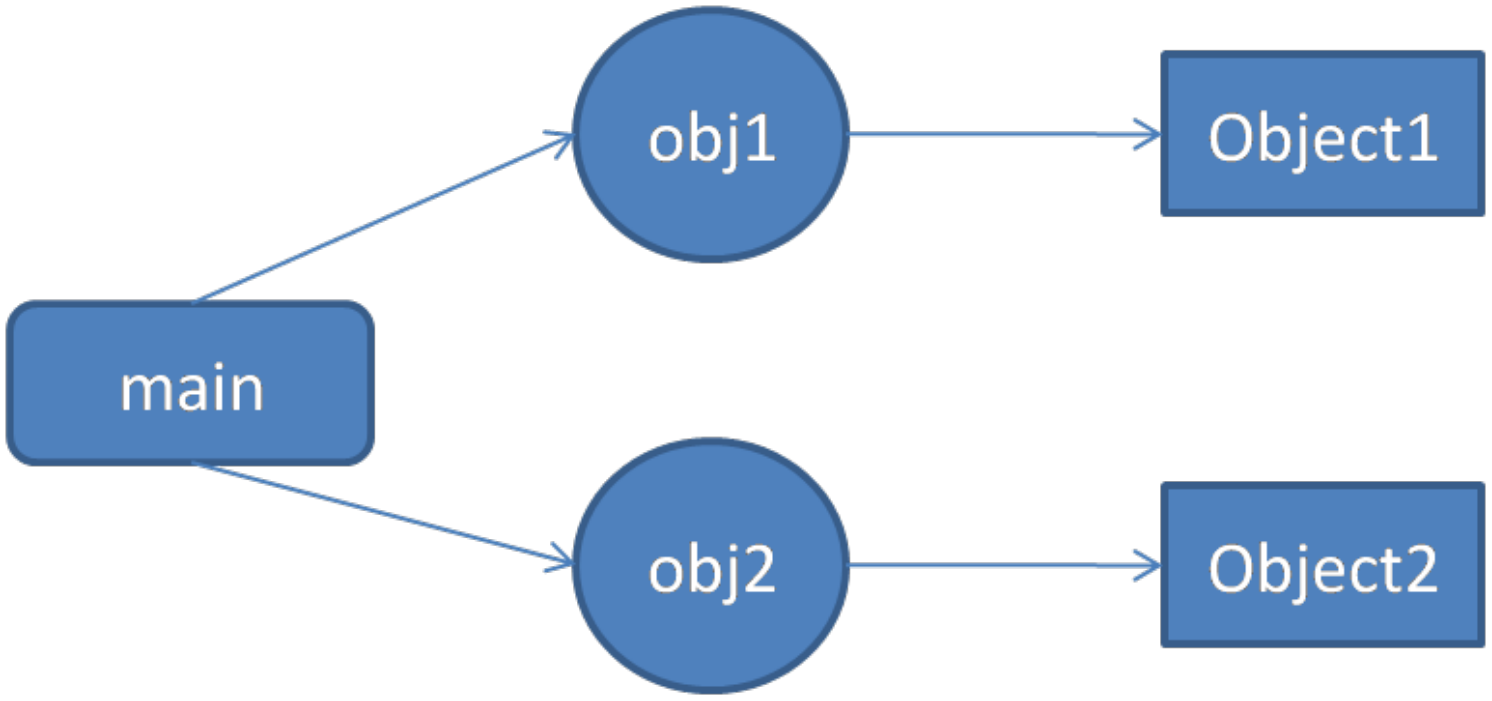
```
public class Demo {
  private float a;// 成员变量跟随对象new出来，存放在堆中
  private static int b;// 静态变量，编译时分配，在静态变量区中
  private static final String str = "DEMO";// 静态常量，编译时分配，DEMO在常量区中，str在栈中

  private void test() {
    int c;// 局部变量，分配在栈中
    Demo demo1 = new Demo();// demo1在栈中，其指向的对象位于堆中
  }
}

private Demo demo2 = new Demo();// demo2位于堆中，其执行的对象的成员变量也都在堆中
```

2.2 How To Manage Memory

上一节把Android中内存分配的机制介绍了一下，这是了解内存泄露的一个基础，而另一个基础则是内

存管理。这里无须非常深奥的内核MM的知识，无须了解如何分页，分块，只需探索下和内存泄露紧密相关的内存回收。内存回收和分配机制也是紧密相连的，对照前一节中介绍的分配机制，挨个来聊下它们的回收机制。首先是常量区和静态变量区，这两块区域中的内存分配出去之后，其作用域是全局的，所以在程序运行期间都不用考虑其回收，直到程序结束之后会自动销毁掉；然后是栈中的变量，因为都是局部变量，跟随这方法的生命周期结束被销毁，所以也是比较安全的；最后是堆中的空间，这部分是动态分配的，在Java中通常程序员也无须关注内存的回收动作，交由GC来执行回收动作。GC回收的算法有三种，在”Garbage Collection in Android”一章中有做介绍，这里我们以典型的标记回收算法（Mark and Sweep GC）来展开。标记回收算法的原理这里不再重复了，以一个例子再来更形象的加深理解

```
class test {
  public static void main(String[] argc) {
        Object obj1 = new Object();
        Object obj2 = new Object();
        obj2 = obj1;
  }
}
```



上图描述了这个例子中的内存管理走向，当main方法结束后，obj2引用和obj1引用都指向了Object1（第一个分配的Object），而Object2变成了无引用指向的对象，它就符合被Mark and Sweep GC算法回收的对象的条件。了解这个基础之后就可以解释内存泄露的产生了。

2.3 Why Memory Leak

GC回收的对象需要符合无用且不可达的条件，那么当对象无用且可达的时候就会出现GC无法回收的情况，内存泄露也就因此而产生。那么什么情况下对象会出现无用但可达的情况呢？在网上参考了一个最典型的例子

```
Vector vector = new Vector(10);
for(int i = 0; i < 10; i++) {
  Object o = new Object();
  vector.add(o);
  o = null;
}
```

其思路就是把对象放到容器中去（可达的状态），然后将对象的引用置为空（无用的状态）。以对象的几种状态来作为内存泄露原理的总结：可用可达（正常）、无用不可达（GC）、无用可达（内存泄露）和有用不可达（存在吗？）。

内存泄露的产生的原理到这里就理清楚了，那么内存泄露是如何对系统造成危害的呢？这是个非常显而易见的问题，但也许个人的理解也有不同把，简单的说下我个人的理解。内存泄露的直接影响是造成了部分内存区域的不可达，也就是说系统可用内存变少，从另一个更量化的角度讲就是变相造成了Generation中的阈值降低，所以更容易触发GC甚至是LMK。这就又回到了之前讨论过很多的GC引发的性

能问题了。 而另一个方面，如果内存泄露严重的话就可能引发内存溢出（OOM， Out Of Memory），也就是说这个程序需要的内存系统给不了了，这种情况就可能导致比较不友好的用户体验了，应用弹框或者闪退。

2.4 Memory Leak Case & Analysis

这一节中以一个案例来介绍内存泄露的几种排查方法。在网上也看了很多别人总结的分析方法以及什么情况下会出现内存泄露，躬身去做之后才能体会到其中的众多内涵所在，在开始时先抒发下个人感悟。首先要说的是关于内存泄露的案例总结，讲了很多各种情况（后面会稍作介绍），但是当我亲自去写案例时发现这些程序都没有出现内存泄露的情况，当时分析的方法是参考他人介绍的反复启动退出Activity来分析这个Activity中的对象是否都被释放了。结合前面的内存分配和管理的背景知识中的介绍，我们需要关注的内存泄露是在堆中动态分配并且GC无法收回的对象的内存。但是忽略了一个大前提，Java层所能够操作到的堆内存是在其进程的JVM层面上，当程序退出时，这部分内存不管GC是否能够回收到，都会被系统给收回了，所以当程序退出时，也就不存在内存泄露了。这是内存泄露这本秘籍的修炼总纲，如没有想通这一点，无论是在看别人各处转载的文章或是亲自做案例抑或后续的coding，每每遇到内存泄露就会觉得不是那么通透，仍有雾里看花似是而非的感觉。

2.4.1 Case

这里要介绍的案例还是网上介绍内存泄露被说的最多的一个例子，使用集合导致内存泄露，先贴下案例的代码

```
public class MainActivity extends AppCompatActivity {

    private static Vector vector = new Vector(10);// new一个初始大小为10的数组
    private Button button = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new Button.OnClickListener(){
            @Override
            public void onClick(View view) {
                Intent intent = new Intent(MainActivity.this, Main2Activity.class);
                startActivity(intent);// 跳转到一个空白的Activity
            }
        });

    }

    @Override
    protected void onResume() {
        super.onResume();
        // 内存泄露原因1：集合类泄露
        for(int i = 0; i < 100; i++) {
            Object o = new Object();
            vector.add(o);
            o = null;
        }
        Log.d("chendongqi", vector.get(0).toString());// 验证Object对象可达
    }
}
```

此案例的设计是这样的，new出Object之后将其引用放入到了Vector数组中，然后将引用置为了空。对象的引用是存放在栈中的，跳出for循环之后就被收回了，而对象的内存是在堆中，所以需要等待GC来做回收。而当GC的时机到来时，如果没有 `vector.add(o)` 这一步的话，那么对象就是不可达的，它的内存空间就会被回收。但是这个案例中将其做成了依旧可达，所以无法被GC回收。但是其实对程序员来说或者是从后续代码的角度，这个对象是无用的（之后再也没有被用到过了），然而又无法被及时回收，所以出现了所谓的内存泄露。ps. 在论坛中有看到有人说将o置为空之后，后面引用Object对象时报空指针了，特意打了log验证了下，出了循环之后对象确实是存在的，而且是通过Vector可达。

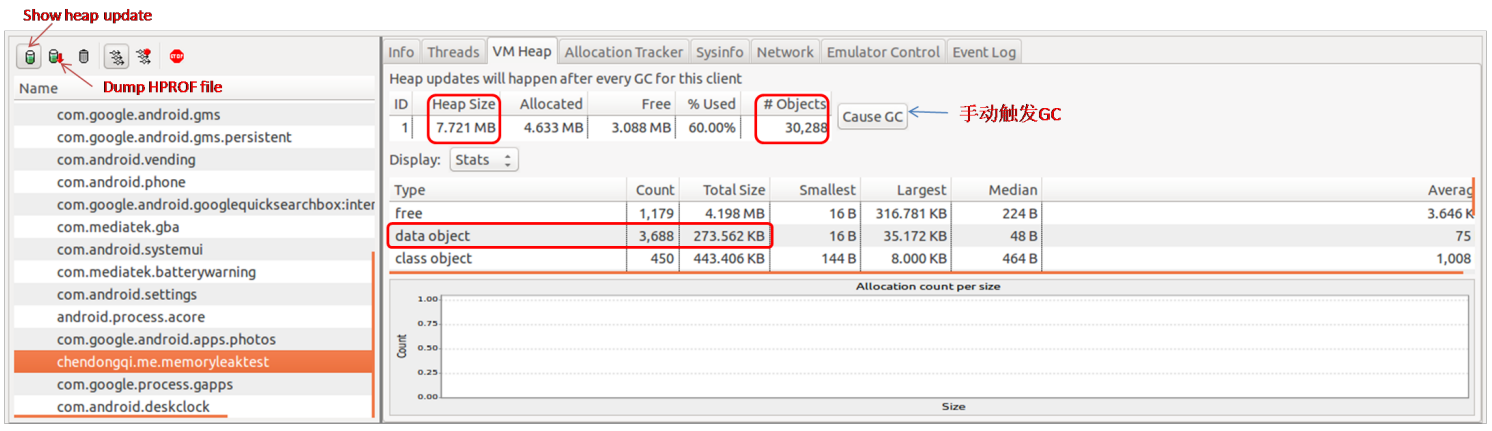
通过对此案例的研究也想通了内存泄露的另一个本质，其实内存泄露也并非那么严重和可怕，根本没有到一种程序员谈虎色变的地步。个中本质和被泄露的变量的作用域是紧密相连的，其含义也可以从上一段中加黑的“及时回收”中细细体悟。在我写的这个案例中，将vector数组用static做了修饰 `private static Vector vector = new Vector(10);` ，因为static的静态变量生命周期是和此程序的生命周期是一致的，所以就算跳转到此程序的另一空白Activity后，vector的对象也是不会被收回，那么可以表述为在此程序内部出现了内存泄露；那么如果去掉static修饰，Vector对象就成了第一个Activity类的成员变量，而当跳转到第二个空白Activity之后，第一个Activity变成不可达对象，其中所有的成员变量所占内存都被回收，那么也就是说内存泄露的作用域缩小了，只是变成了第一个Activity内部，更准确的来说，是在for循环结束一直到该Activity生命周期结束的时间内是存在内存泄露的；那么如果我们更加缩小Vector对象的作用域呢，比如说就放在onResume方法中，那么在onResume结束之后存放在栈中的Vector的引用被销毁，其对象也变得不可达了，内存泄露也就不存在了。

前面讨论了从作用域的角度来看内存泄露，作用域可以理解为对象的作用域和内存泄露起作用的作用域。从这个角度来归结，内存泄露的产生即是赋予了一个对象不合适的作用域。资治通鉴中讲楚汉相争篇，刘邦在一统天下之后就开始大肆迫害萧何，张良等一伙开国功臣，其中原因之一在于刘邦多疑，害怕功高盖主。历史上功高盖主的例子也是时常有之，追随李世民打江山的一帮瓦岗英雄，得善终的也是寥寥无几。其实功高盖主的事和这里的内存泄露在一定层面上也是相通的，当一个变量的作用域过大（战功赫赫，统领三军），当不需要此变量的时候内存空间无法被收回（平定天下之后还不交出兵权），那么内存泄露就产生了（皇帝就沦为傀儡了，例如少年康熙和鳌拜）。所以历史上被广为批判的迫害开国元勋的罪行，在这里如果用纯理性的角度去辩证，何错之有？

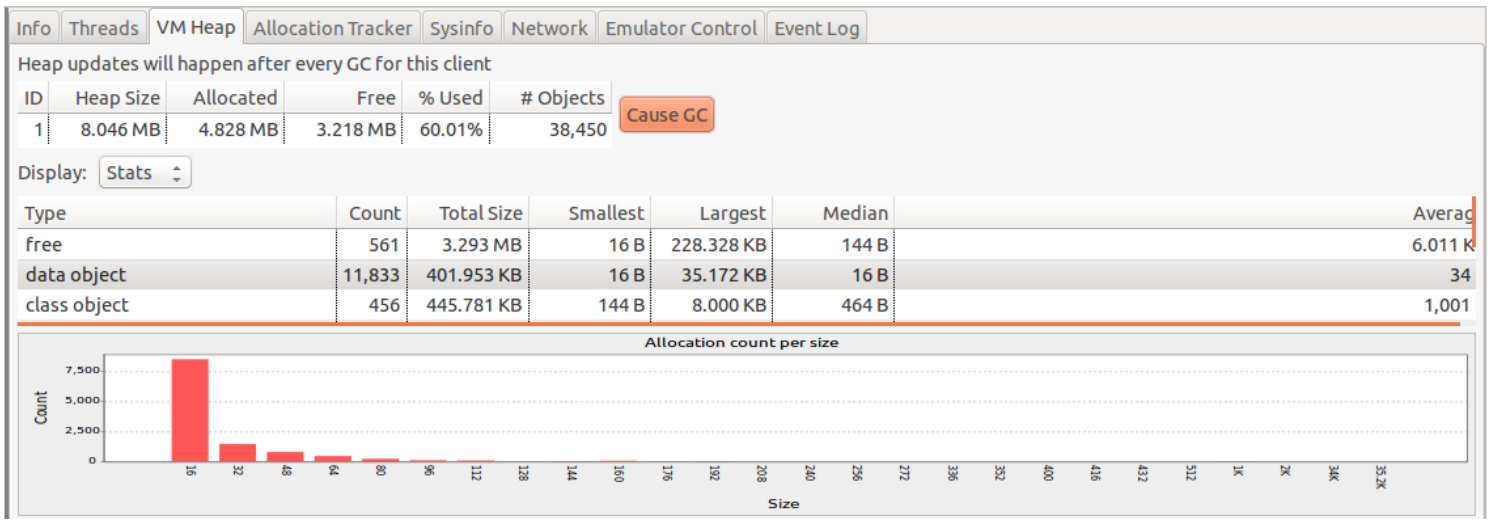
2.4.2 Use Heap Tool To Check Memory Leak

讨论了这么多内存泄露的原理和自己的思考，下面来看下当一个应用是一个黑盒时如何去判断是否存在内存泄露的情况，还是通过上面的案例来介绍一种非常典型的方法。用到的工具是Heap Tool来查看待观察进程的堆内存的情况，分析的思路如下：1、启动待分析Activity界面，手动触发几次GC，记录下堆中内存情况；2、跳转到其他Activity（最好是一个空白的Activity或者是其他确保没有内存泄露的Activity）；3、返回到待观察Activity界面；4、重复步骤2和3多次，最终退回到待观察Activity界面，记录下Heap内存情况；5、对比两次记录的Heap分析待观察界面是否存在内存泄露。下面来看下实际操作过程。

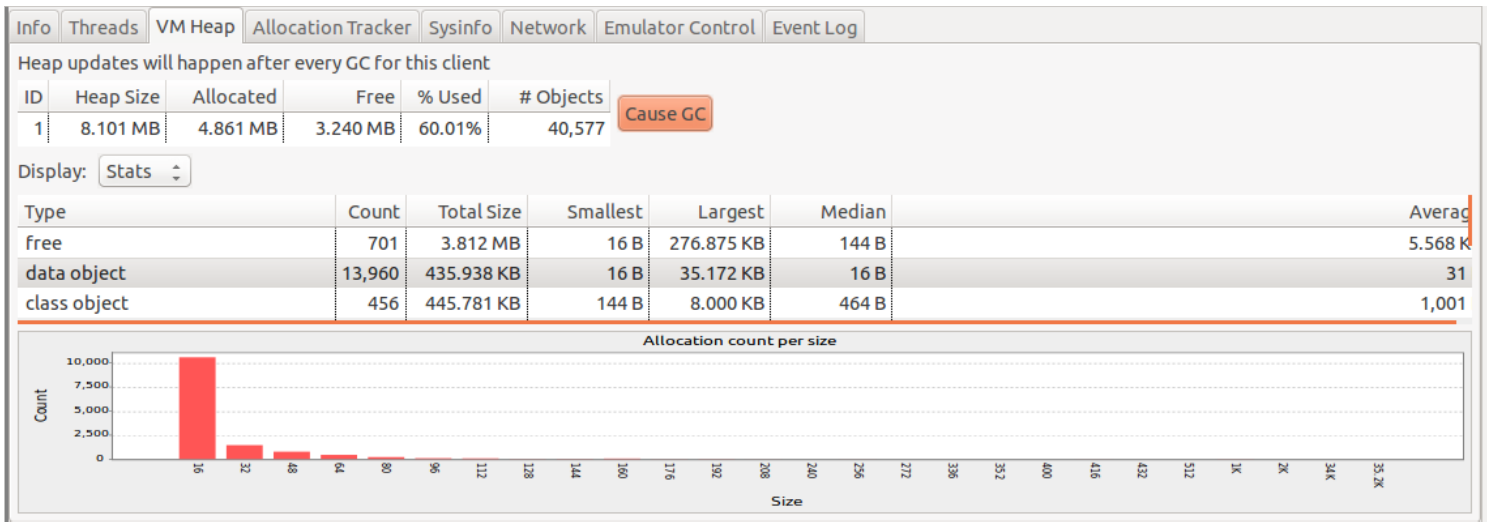
启动测试应用之后，启动DDMS，然后选中测试进程



进入到该案例中的MainActivity中之后点击Heap Tool的“Show heap updates”，然后手动触发几次GC之后抓下heap中的内存情况，如下图。



然后启动到Main2Activity（一个空白的Activity），再返回到MainActivity，多次重复该操作后，停留在MainActivity，然后再手动触发几次GC，当Heap内存情况不再改变之后再记录下heap中的内存情况，如下下图

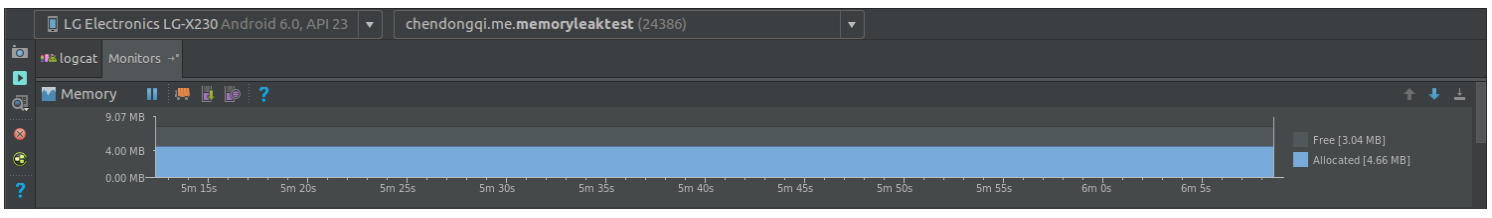


可以对比两组数据来判断MainActivity中是否有出现内存泄露，第一组是看Heap Size和Allocated，可以发现第二次的截图中明显比第一次截图中都有增长，这里需要说明的最直接的应该是看free的空间在出现不停减小时就可以推断出有内存泄露，其实不然，因为在需要给对象分配内存时，堆的大小是会增长的，所以free的值可能反而会增长一些或者保持不变，当heap size达到上限时看free才有意义；第二组数据可以看data object的count和total size。如果没有内存泄露的情况，当进入到Main2Activity时，MainActivity中的成员变量都被GC回收掉了，反复操作之后count和total size的值也应是不变的，这个例子中可以明显看到这组数据也是在不停增长。说明MainActivity的对象并没有被释放掉。

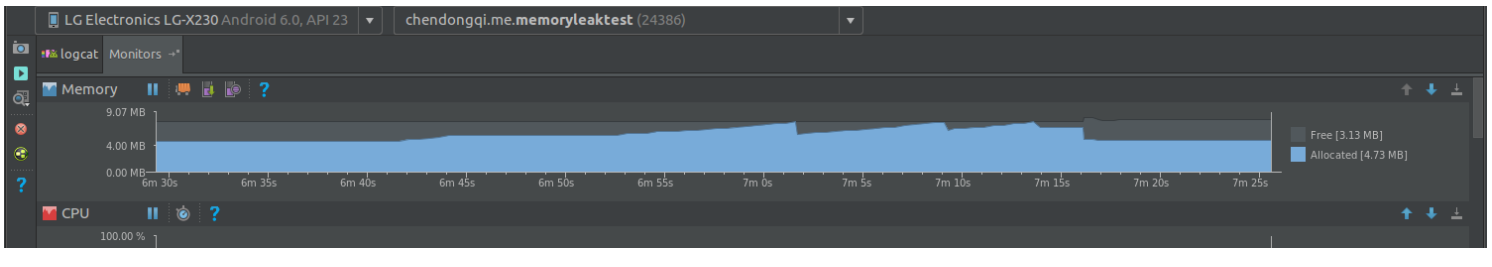
罪魁祸首就是 `vector.add(o);` 当注释掉这一句之后，同样的操作就不会看到heap size或者是data object有持续增长的情况出现了。另外在这个案例中出现了另一个没有预料到的结果，`private static Vector vector = new Vector(10);` 将这里的static去掉之后再测，依照之前理论的推测，当退出MainActivity对象被销毁，成员变量所占的内存也就被GC回收了，但是结果是依旧出现了内存泄露的情况。如何解释这个情况？思考了下这个现象，还是需要从Activity的生命周期着手，当跳转到Main2Activity中时，MainActivity处于onStop的状态，此时Activity是不会被销毁的（根源在此），只有走到onDestroy时才会被彻底销毁掉，而触发onDestroy是有条件的，要不就是系统内存不足触发lmk或者是该应用整个退出了。所以在这个案例中，Activity的成员变量的生命周期可能也会很长，内存泄露的危害程度比之前理论推测的要高了一分。

2.4.3 Use Memory Monitor To Check Memory Leak

除了DDMS中的Heap Tool，AS中的Memory Monitor也可以做到同样效果的分析效果，我自身更倾向于Memory Monitor。首次进入MainActivity后记录下heap中的memory信息



同样做内存泄露的操作之后截图如下

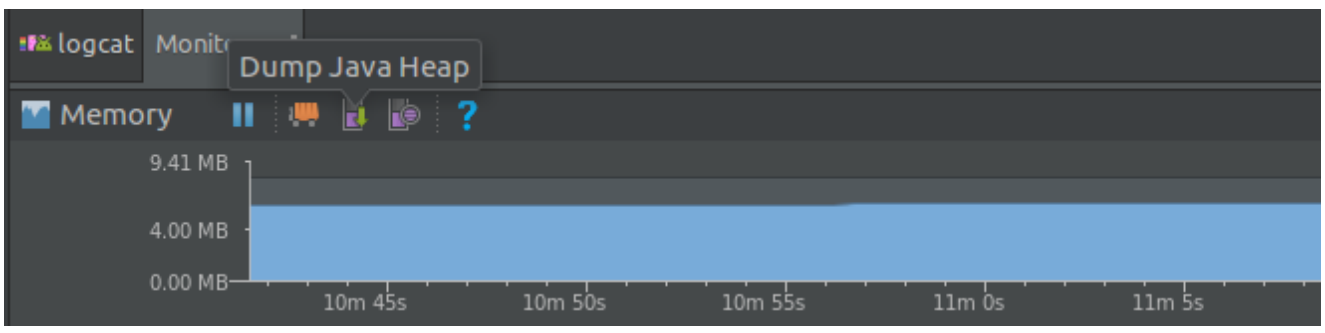


对比查看Allocated的值就可以发现一致在增长，从而推断出内存泄露的情况。

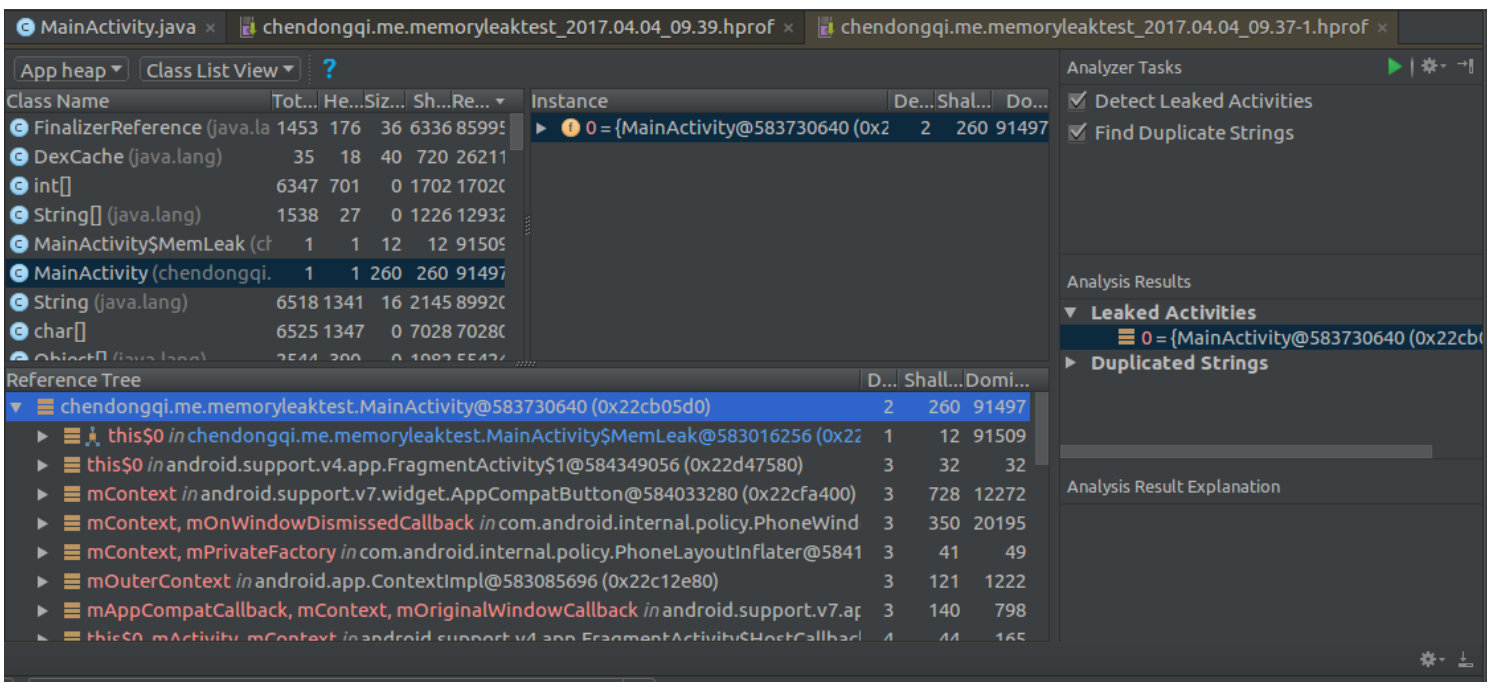
2.4.4 Who Is The Murderer

确定存在内存泄露之后，紧接而来的问题是谁导致了内存泄露。漫谈个观点，理工科问题的解决都是一脉相承，程序员分析bug和医生看病，警察破案都是同样的原理，找线索，用工具，推断，验证等等手段和思路都是可以相互借鉴的，区别只在于对象不同导致的某些特性差异。继续回到主题，如何去查找内存泄露的罪魁祸首，可用的典型工具有Android Studio的Memory monitor和MAT。另外还有一个插件工具LeakCanary (https://github.com/square/leakcanary)，可以直接作为第三方插件在手机端来分析应用的内存泄露，我还未尝试，有兴趣的同学可以亲测下。这边我主要来介绍Memory monitor和MAT的分析方法，其实对于这两款工具的使用网上也有了很多说明性的文档，但是所站的角度都很高，以一种你照着文档操作就万无一失的架势在折磨这读者，我想无论是那款工具的使用都会有一定的困难，尤其面对内存泄露这么复杂的问题，没有丰富的经验，手握利器也是无从下手的。那么就丑话说在前面，分析内存泄露的问题，不要那么乐观，并非都是工具在手，天下我有的局面。在下面的介绍中我将一边讲解工具的使用，一边谈及Memory Monitor和MAT定位内存泄露的的元凶的基本思路、可能出现的难点和困惑（主要是个人遭遇到的经验）以及分析中的技巧等。

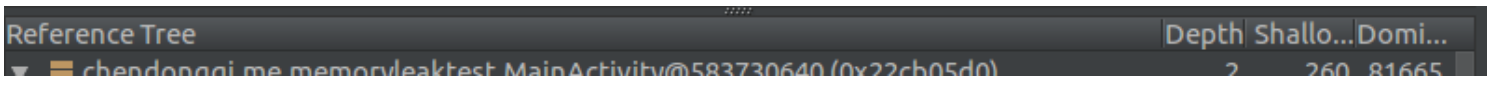
上一小节中已经介绍了用Memory Monitor来观察确定是否有出现内存泄露的情况，那么此刻就先来看Memory Monitor如何进一步的查出Memory Leak的元凶。Memory Monitor分析的过程比较简单，首先是抓Java Heap，然后自动化分析，最后根据分析结果定位问题代码。首先需要在进行内存泄露的操作之后将堆内存给dump出来，按下图中显示的按钮操作

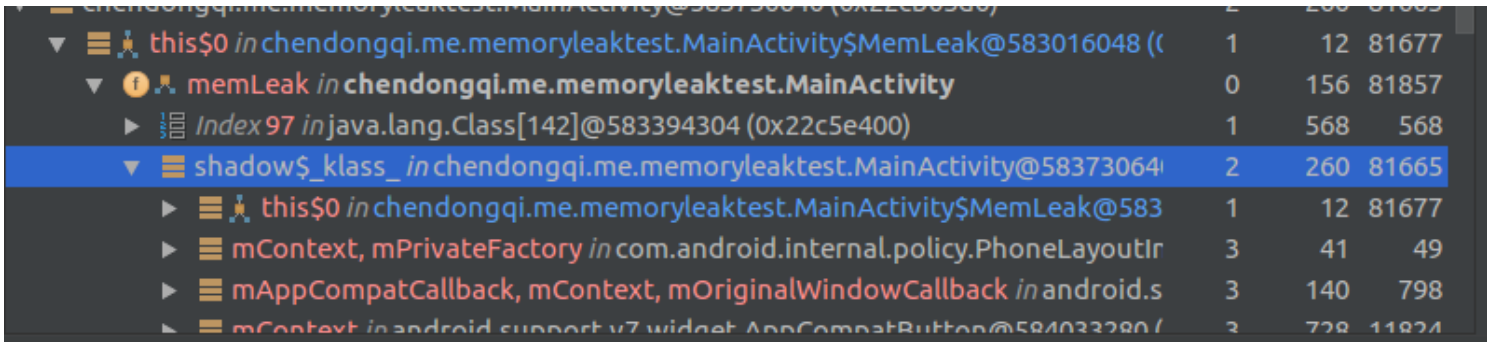


需要说明的，从上面一直作为案例的Vector集合导致泄露的问题，在Memory Monitor中却真真切切的无法被分析出来，所以下图开始使用了另一个典型的问题来做为范例。秉承寻找Murderer的原则，代码就在后面再公开，先来看查找的过程。点击“Dump Java Heap“按钮之后会在Android Studio中抓到并自动打开hprof文件



然后点开面板右侧的Analyser Tasks（原先应该是收起来的），点击上面的绿色小三角（Perform Analysis）进行内存泄露的分析。在上面的截图中就能看到有Analysis Results出现了Leaked Activities和Duplicated Strings两个结果，而Leaked Activities就是我们在寻找的导致内存泄露的重要线索。点开Leaked Activities之后可以发现内存泄露的是MainActivity。查看其详细的引用关系可以看到如下图





看这些信息之前需要先了解下图中每一列的含义，可以参见下表

名称	描述
Class name	类名
Total Count	该类的实例总数
Heap Count	所选择的堆中该类的实例的数量
Sizeof	单个实例所占空间大小（如果每个实例所占空间大小不一样则显示0）
Shallow Size	堆里所有实例大小总和（Heap Count * Sizeof）
Retained Size	该类所有实例所支配的内存大小
Instance	具体的实例
Reference Tree	所选实例的引用，以及指向该引用的引用
Depth	GC根节点到所选实例的最短路径的深度
Shallow Size	所选实例的大小
Dominating Size	所选实例所支配的内存大小

了解每个数据的含义之后可以看到图中引用树里占用Dominating Size最大的一条线索是MainActivity->this->memLeak->Shadow$_klass_->this。插入介绍下这条线索中看起来比较陌生的一个变量Shadow$_klass_，这是在Android5.0之后加入到Object.java对象中的一个变量，也可以理解为该对象的引用

```
private transient Class<?> shadow$_klass_;
private transient int shadow$_monitor_;
...
/**
     * Returns the unique instance of {@link Class} that represents this
     * object's class. Note that {@code getClass()} is a special case in that it
     * actually returns {@code Class<? extends Foo>} where {@code Foo} is the
     * erasure of the type of the expression {@code getClass()} was called upon.
     * <p>
     * As an example, the following code actually compiles, although one might
     * think it shouldn't:
     * <p>
     * <pre>{@code
     *   List<Integer> l = new ArrayList<Integer>();
     *   Class<? extends List> c = l.getClass();}</pre>
     *
     * @return this object's {@code Class} instance.
     */
public final Class<?> getClass() {
      return shadow$_klass_;
    }
```

这条线中间出现了一个可疑的引用memLeak，查看代码中可以发现该变量为MainActivity的内部类的一个静态引用，如果在此方面经验丰富的工程师看到这句应该就恍然大悟了，下面就来公布下问题的真相

吧。这个案例的设计也是非常典型的一个内存泄露的示例，原理就如前面说的，内部类的静态引用引发的一场血案。具体代码设计如下

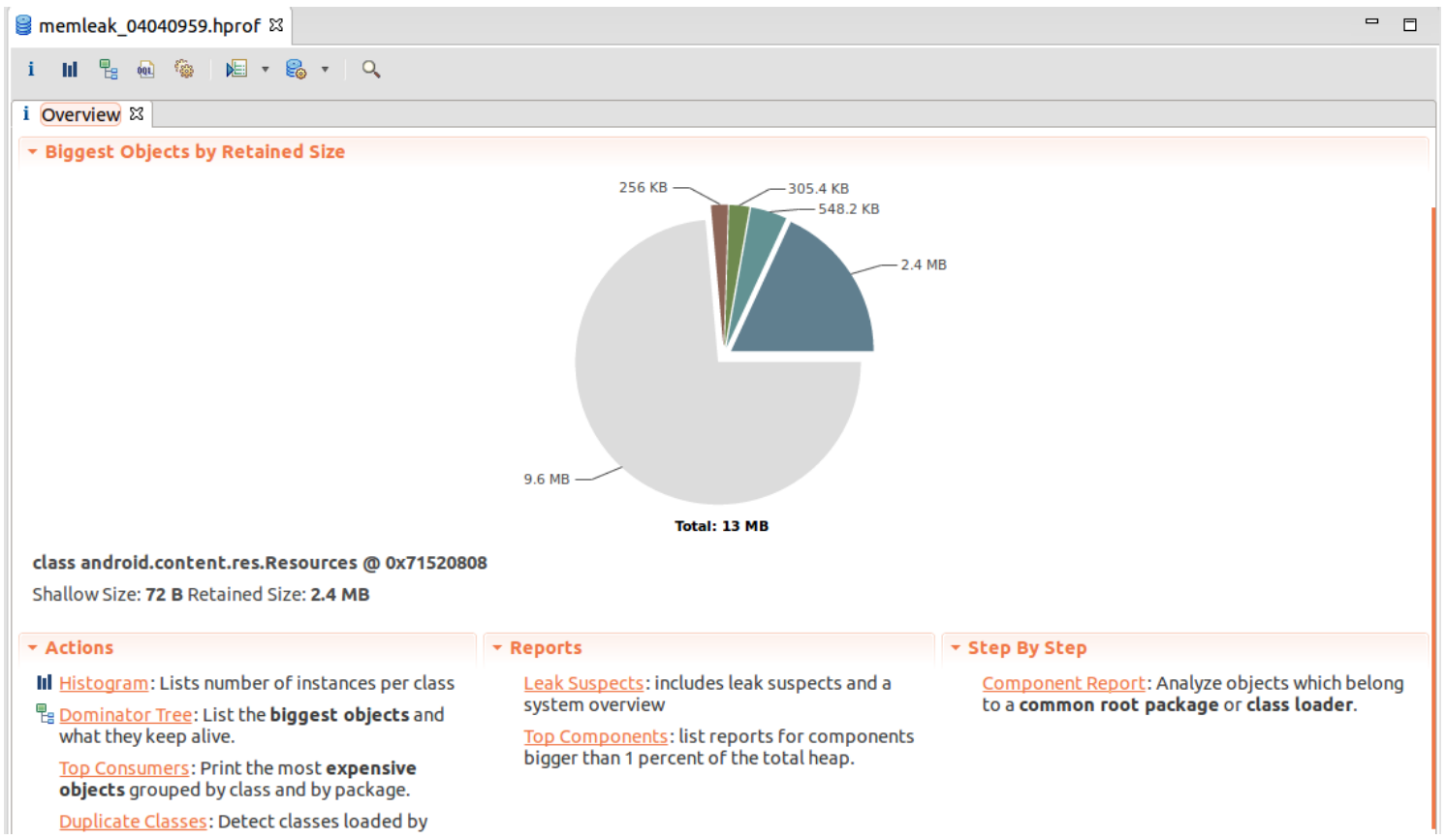
```
public class MainActivity extends AppCompatActivity {
  ...
  private static MemLeak memLeak = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
      ...
      memLeak = new MemLeak();
      ...
    }

  class MemLeak {

  }
  ...
}
```

为何内部类的静态引用会持有外部类的引用导致内存泄露呢？熟悉内部类的同学一定知道在内部类中可以访问所有外部类的成员变量和方法，因为Java编译器在创建内部类对象时，隐式的把其外部类对象的引用也传了进去并一直保存着。所以内部类的对象也持有了外部类的引用，而当我们将内部类的对象又定义为static时，其引用就放在了静态变量区，不会被GC回收掉了。

这个案子也就水落石出，再回过头来评价下Memory Monitor在分析此类问题的作用。优点显而易见，非常自动化，Analyser Tasks可以为我们在众多杂乱的数据中理出一条线索来进一步追踪；而不足之处也是深有体会，第一个Vector导致的内存泄露问题自动化分析就江郎才尽了，原因是Analyser Tasks还只能分析到所有已经被销毁但是无法回收的Activity，所以只能分析到Activity层次的内存泄露问题。

下面再来看下内存分析的又一利器MAT (http://eclipse.org/mat/downloads.php)， MAT也是通过对Java Heap的分析来确认是谁导致了内存泄露。hprof的来源可以有两种渠道，前面也有零散的提到了，这里再重复下，第一种就是在DDMS中通过dump HPROF file来抓取，另一种就是在Android Studio的Memory Monitor中通过dump Java heap来抓取，这里抓取到的文件会存放在工程的captures目录下。MAT原先是为分析Java程序所设计，而我们抓到的hprof文件是基于Android的，首先需要使用Android SDK下的platform-tools/hprof-conv工具来转换成Java的格式，否则MAT无法打开。转换的最简单命令为hprof-conv source target。当然如此强大的Android Studio也直接为我们提供了图形界面的转换方式，在Android Studio中找到captures目录（最左侧和Project同级），然后右键选择需要转换的文件，执行”Export to standard .hprof“，不过好像我的Android Studio不怎么听话一直失败，具体原因也是没有去追查了，改用SDK的工具做的。

hprof文件拿到之后，在MAT中选择file->Open Heap Dump来打开该文件，内存泄露分析的奇妙之旅就此开启，或将翱翔在数据海洋之后不可自拔，甚至可能迷失其中而不能自已。幸而有前人铺设的台阶和岔路口的火把以供我们走向光明的出口。先来看下MAT打开hprof文件之后呈现的一些信息



multiple class loaders

占据中央位置的是一个饼图，呈现了堆中内存的分布情况，列举了主要的几块内存占据比例较大的对象。通常来说导致较大程度的内存泄露问题才会更有关注的价值，而这种类型的内存泄露就会有一类对象占据大量的内存空间，所以在饼图中有可能会被找到。当然本文中用到的内部类静态引用的case是很难出现马上导致大量的内存泄露。

下面的Actions一列中，都是检测和排序class的信息，Histogram（列举出所有对象信息，数量，引用所占的内存和实例所占的堆内存）、Dominator_Tree（支配树，除了内存情况还列出了所有引用的指向关系）、Top Consumers（也是饼图的形式，列出了消耗内存排名最多的一些对象信息，比Overview中更详细）、Dumplicate_classes（列出被Class Loader多次装在的类，这里没有发现）。

然后来看Reports一栏，这一栏都是报告形式的数据。首先是非常常用的Leak Suspects，是一份MAT帮我们自动分析之后给出的一份内存泄露嫌疑人的报告，其中包含了几个可疑的问题点，并给出了问题的说明，占用的内存，进一步分析的关键字等信息；第二个Top Components展示了占用内存超过堆内存百分之一的组件的信息（非常多的信息，感觉很难从这儿入手分析内存泄露问题）。

大体了解了MAT能够给我们提供的一些信息，下面入手来定位内存泄露的问题。先用第一种方式，我个人习惯是用Histogram或者Dominator_Tree来入手。以Histogram为例，打开之后也是一堆的对象无从下手。然后加入一条重要线索，在前面用DDMS或者Memory Monitor检测时我们已经可以定位到内存泄露是在MainActivity中，所以我们在Class Name下面搜索MainActivity来过滤出相关的信息（搜索支持正则表达式）。

Class Name	Objects	Shallow Heap	Retained Heap
.*MainActivity.*	<Numeric>	<Numeric>	<Numeric>
chendongqi.me.memoryleaktest.MainActivity	1	272	>= 8,968
chendongqi.me.memoryleaktest.MainActivity$1	1	16	>= 64
chendongqi.me.memoryleaktest.MainActivity$MemLeak	1	16	>= 64
Σ Total: 3 entries (4,552 filtered)	3	304	

查看MainActivity一项，左边的信息栏中可以看有个memLeak的引用持有了它，这里就值得引起怀疑了。然后我们再来进一步确诊，右键点击MainActivity，执行“Merge Shortest Paths to GC Roots->exclude all phantom/weak/soft/etc. references”，然后查看结果。

<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
class chendongqi.me.memoryleaktest.MainActivity @ 0x22cac000 Sy	1	24	272	80
memLeak chendongqi.me.memoryleaktest.MainActivity$MemLeak	1	16	272	16
this$0 chendongqi.me.memoryleaktest.MainActivity @ 0x22cb05d	1	272	272	8,888

要理解生成的结果需要解释下上面执行的这个命令，是查看一个对象从GC Root开始是否有引用链相连接，不包含所有虚引用、弱引用和软引用。也就是说过滤出来的就是无法被GC所回收的对象了，那么也就是问题的元凶。这里可以明确的定位到是存在这样一条引用链的，而其根源就是memLeak这个引用，所以再回头查代码就能很快定位到问题了。Dominator_Tree可以用来和Histogram做类似的分析。

除了使用Merge Shortest Paths to GC Roots 我们还可以使用

- List object - With outgoing References 显示选中对象持有那些对象
- List object - With incoming References 显示选中对象被那些外部对象所持有
- Show object by class - With outgoing References 显示选中对象持有哪些对象, 这些对象按类合并在一起排序
- Show object by class - With incoming References 显示选中对象被哪些外部对象持有, 这些对象按类合并在一起排序

上面这种方法是在能确定一条关键线索（内存泄露和MainActivity相关）的时候比较适用和快速的，而当拿到一份hprof文件缺少这类线索时，那么更多的需要首先借助MAT的自动化分析来给我们提供一些可疑线索了。来看下上面提到过的Leak Suspects。

Overview

Problem Suspect 1

The class **"android.content.res.Resources"**, loaded by **"<system class loader>"**, occupies **2,471,968 (18.10%)** bytes. The memory is accumulated in one instance of **"android.util.LongSparseArray[]"** loaded by **"<system class loader>"**.

Keywords
android.util.LongSparseArray[]
android.content.res.Resources

Details »

Problem Suspect 2

60,659 instances of **"java.lang.String"**, loaded by **"<system class loader>"** occupy **5,367,112 (39.29%)** bytes.

Keywords
java.lang.String

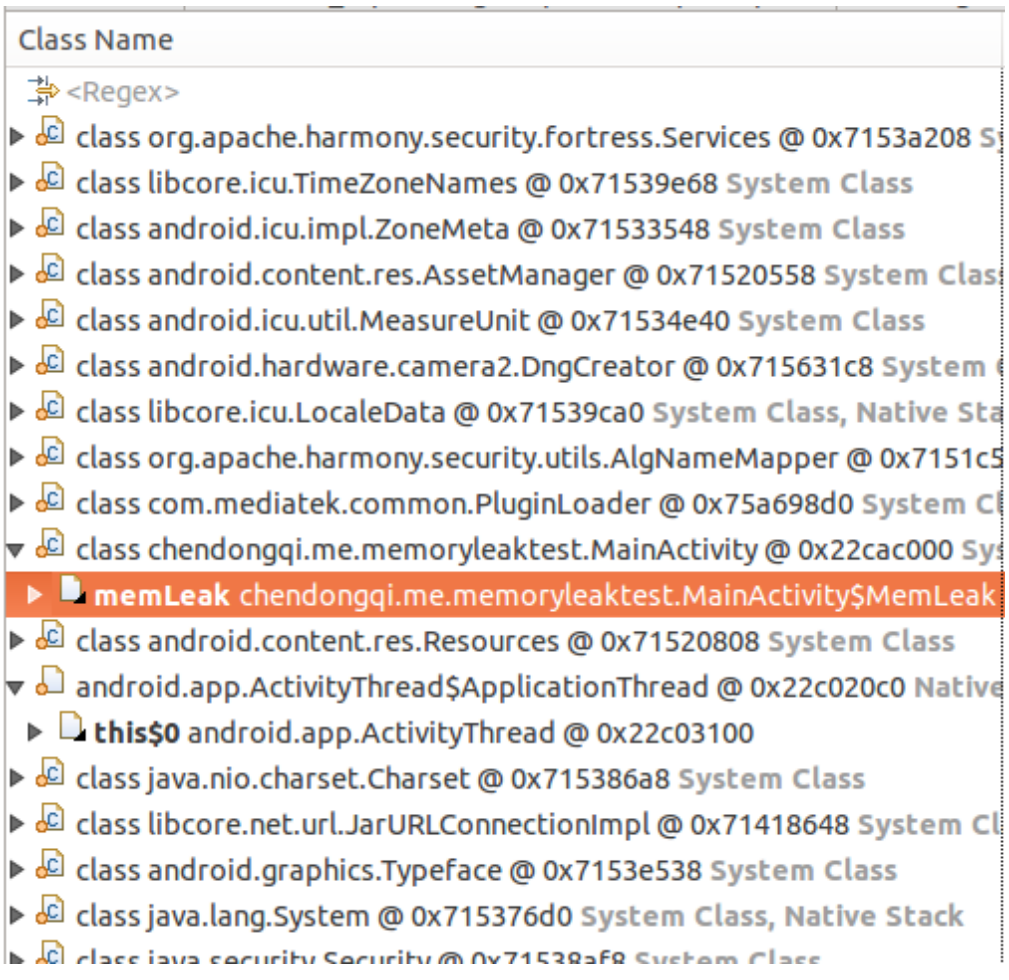
Details »

截图中将overview的饼图给收起来了，可以看到后面会有对怀疑点的解释，没有头绪的时候就挨个查看下。这个案例中一眼看去会比较迷茫，占内存多的部分是资源、字串等，挨个check下。以“Problem Suspect 2”为例来示范下，60659个实例占据了5367112个字节的内存，提示的关键字为java.lang.String。然后就去Histogram中去搜索下该关键字

Overview | default_report org.eclipse.mat.api:suspects | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
.*java.lang.String.*	<Numeric>	<Numeric>	<Numeric>
java.lang.String	65,155	1,563,720	>= 5,837,856
java.lang.String[]	1,536	1,346,536	>= 1,418,584
java.lang.String[][]	7	5,008	>= 63,320
java.lang.StringBuilder	3	72	>= 256
java.lang.String$CaseInsensitiveCompa	1	8	>= 48
java.lang.StringBuffer	0	0	>= 192
java.lang.StringFactory	0	0	>= 32
java.lang.StringIndexOutOfBoundsExce	0	0	>= 40
java.lang.StringToReal$StringExponent	0	0	
java.lang.StringToReal	0	0	
Σ Total: 10 entries (4,545 filtered)	66,702	2,915,344	

以之前的分析方法挨个检查下是否有可达的GC路径



也找到了memLeak这个引用导致的无法回收的原因，这一步的分析需要过滤掉很多干扰信息，或者说需要更多的经验来找出可疑的那个引用，很多时候也许也会是一无所获。此种方法个人的感悟并非很深，就介绍这么多吧，后续做到更有代表的案例再来补充。

2.5 Memory Summary

这一章中介绍了内存泄露的原理和分析，为了理解内存泄露的产生，首先介绍了堆和栈中内存的分配规则：栈中存放的局部变量（基础变量，对象的引用），堆中分配的是new出来的对象，另外还有静态变量区和常量区中分配的静态变量和常量。然后又介绍了内存管理的模型，主要是GC的标记回收算法将无法回收那些可达无用的对象，内存泄露由此而来。之后很大的篇幅介绍了内存泄露case的分析方法，通过DDMS或者Memory Monitor观察Java Heap的动态变化可以确认某个Activity或者是操作是否存在内存泄露，然后是定位具体的问题原因。可以通过Memory Monitor中的Analyser Tasks来自动检测泄露的Activity或者是更专业的MAT来分析。

3. Tools

内存性能分析相关的工具其实在上面的内容中已经都有讲到过，也有案例来支撑具体的使用方法。这一章中会将这些工具再择出来汇总，以作为单独的一个工具集合以供方便查阅。

	Memory Churn	**Memory Leak**
Memory Monitor	通过波形图观察内存抖动的发生	观察内存泄露的现象和抓取hprof文件并分析
Allocation Tracker	抓取内存抖动过程中内存的分配以供分析	NA
Heap Tool	NA	观察内存泄露的现象和抓取hprof文件
MAT	NA	分析hprof来定位内存泄露的原因

PREVIOUS
ANDROID PERFORMANCE PATTERNS——UI PERFORMANCE **(/2017/03/08/ANDROID_PERF_PATTERNS_UI/)**

NEXT
ANDROID电源管理之DOZE模式专题系列（六）**(/2017/03/15/PM_DOZE_SENSING_TO_LOCATION/)**

FEATURED TAGS (/tags/)

前端 (/tags/#前端) Android (/tags/#Android) frameworks (/tags/#frameworks) AlarmManager (/tags/#AlarmManager) Performance (/tags/#Performance) systrace (/tags/#systrace) PowerManager (/tags/#PowerManager) Wakelock (/tags/#Wakelock) Guitar (/tags/#Guitar) 民谣 (/tags/#民谣) 赵雷 (/tags/#赵雷) Doze (/tags/#Doze) Android Performance Patterns (/tags/#Android Performance Patterns)

FRIENDS

待遇见志同道合的你 (https://github.com)　小明 (http://www.betterming.cn)

(https://twitter.com/chendongqi)

(https://www.zhihu.com/people/chendongqi)


(http://weibo.com/chendongqi)

(https://www.facebook.com/chendongqi)


(https://github.com/chendongqi)


(https://www.linkedin.com/in/firstname-lastname-idxxxx)

Theme by Cheson (https://github.com/chendongqi/blog) | Star 1