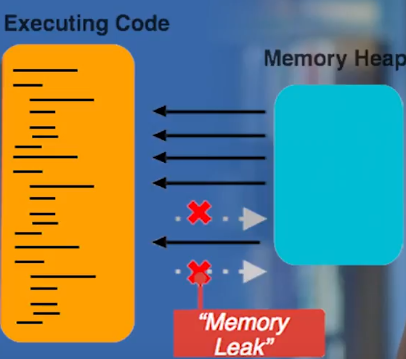
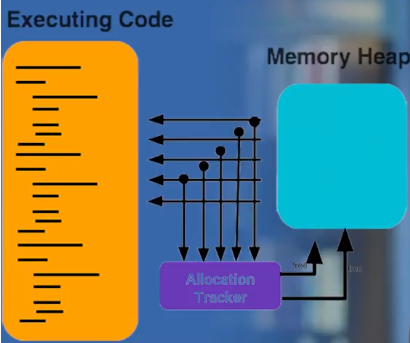
**内存与垃圾回收器**

不是所有指令都执行得又快又好，下面介绍内存及它如何影响系统运行。普遍认为，多数程序语言接近硬件或高性能，如C、C++和Fortran，通常程序员会自己管理内存，高手工程师对内存的分配，会慎重处理，并在未来结束使用时再次分配，一旦确认何时及怎样分配内存，内存管理的品质就依赖于工程师的技能跟效率。实际情况是工程师们，不都会去追踪那零碎的内存碎片。程序开发是个混乱又疯狂的过程，内存通常都没办法完全被释放，这些被囚禁的内存叫内存泄露。



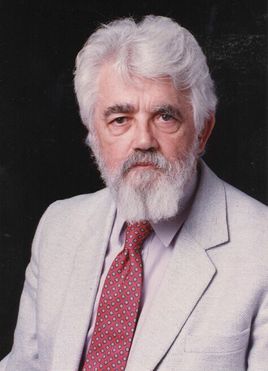
内存泄露

内存泄露占用了大量资源，这些资源其实可以更好地使用，为减少泄露引起的混乱、负担、甚至资金损失，便有了内存管理语言。



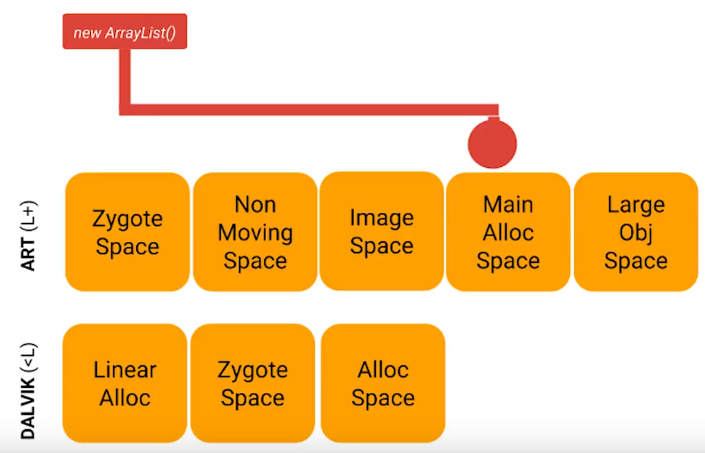
跟踪内存分配

这些语言在运行时跟踪内存分配，以便当程序不再需要时释放系统内存，完全不用工程师亲自操作，这些内存回收艺术或科学，在内存管理环节下叫垃圾清理。这个设计概念在1959年，当初为了解决lisp语言问题，由John McCarthy发明的。



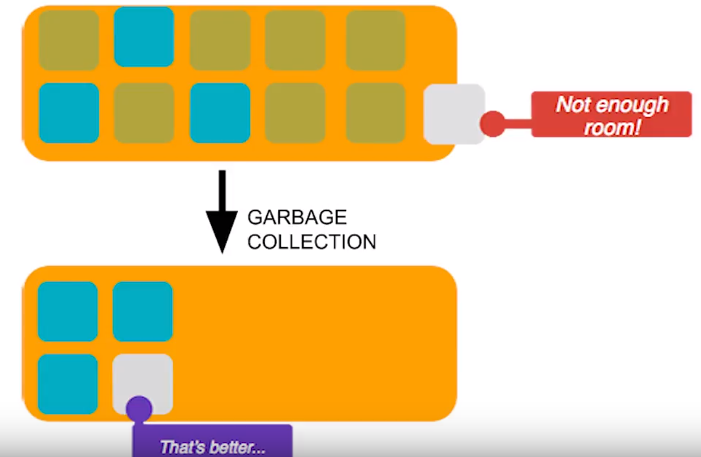
约翰麦卡锡-人工智能之父

垃圾清理的基本概念有：第一，找到未来无法存取的数据，例如所有不受指令操控的内存。第二，回收被利用过的资源。原理简单，但是两百万行编码，跟4gigs的分配，在实际操作时却非常困难。如果在程序中有20000个对象分配，垃圾清理会让人困惑，哪一个是没用的？或者，何时启动垃圾清理释放内存？这些问题其实很复杂。好在50年来，我们找到了解决问题的方法，就是Android Runtime中的垃圾清理。比McCarthy最初的方法更高级，速度快且是非侵入性的。经由分配类型，及系统如何有效地组织分配以利GC的运行，并作为新的配置。所有影响android runtime的内存堆都被分割到空间中，根据这些特点，哪些数据适合放到什么空间，取决于哪个Android版本。



根据不同类型进行运行时内存的分配

最重要的一点是，每个空间都有预设的大小，在分配目标时要跟踪综合大小，且空间不断地扩大，系统需要执行垃圾清理，以确保内存分配的正常运行，值得一提的是使用不同的Android runtime，GC的运行方式就会不同。例如在Dalvik中很多GC是停止事件，意思是很多指令的运行直到操作完成才会停止。

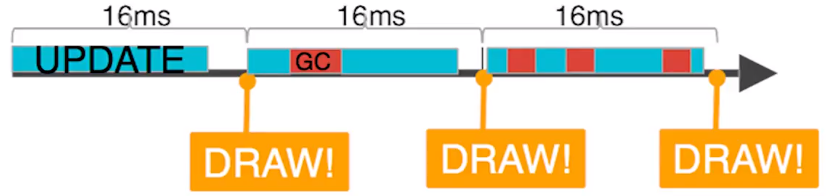


内存不足时GC处理

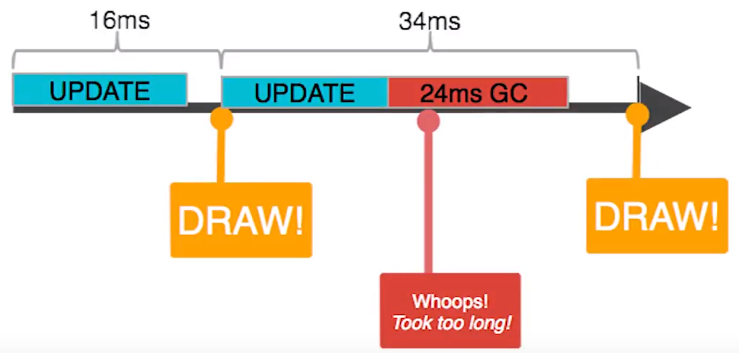
当这些GCs所用时间超过一般值，或者一大堆一起执行会耗费庞大的帧象时间，这是很麻烦的事情。



绘图过程中GC回收



GC回收时间过长导致卡顿



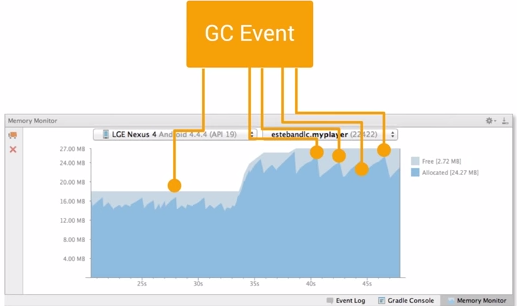
GC回收时间过长导致卡顿

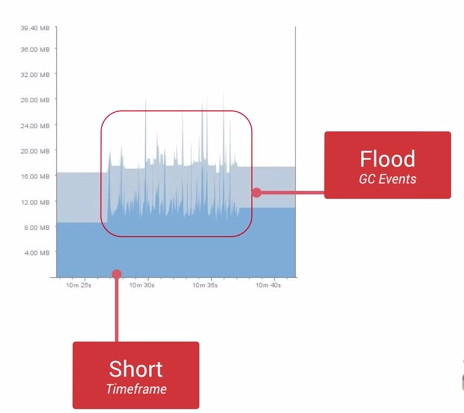
Android工程师花费大量时间降低干扰，确保这些程序以最快的速度运行，话虽如此，在指令中影响程序执行的问题仍然存在，首先程序在任意帧内执行GCs所用的时间越多，消除少于16毫秒的呈像障碍，所必需的时间就会变少，如果有许多GCs或一大串指令一个接一个地操作，帧象时间很可能会超过16毫秒的呈像障碍，这会导致隐形的碰撞或闪躲。其次，指令流程可能造成GCs强制执行的次数增多，或者，执行时间超过正常值。例如，在一个长期运行的循环最内侧分配囤积对象，很多数据就会污染内存堆，马上就会有许多GCs启动，由于这一额外的内存压力，虽然内存环境管理良好，计算比其他语言复杂，内存泄露仍会产生，这些漏洞在GCs启动时，通过无法被释放的数据污染内存堆，严重降低可用空间的总量，并以常规方式强制GC的执行。就是这样，如果要减少任意帧内启动GC的次数，需要着重优化程序的内存使用量，从指令的角度看，或许很难追踪这些问题的起因，但是，多亏Android SDK拥有一组不错的工具。

**Memory Monitor工具**

我们来介绍一个叫作Memory Monitor的工具，Memory Monitor用于测试程序在一段时间后占用了多少内存，下面来操作一下。点击打开，然后会在Android Studio右下边的视窗里，开启一个制表键，一旦发现在运行的程序，就会马上开始记录内存使用量，正如这里所示，在Memory Monitor视窗的左上端，可以切换当前连接的装置，右边这里可以选择要监测的程序。几乎占用全部视窗的叠层图，表示还有多少内存可用。深蓝色的区域，表示当前正在使用中的内存总量，浅蓝色或者浅灰色区域，表示空闲内存或者叫作未分配内存。图表会在内存使用量变化时不断更新，随着时间推移，它也会不断显示可用内存量。随着时间推移，它也会不断显示可用内存量，总之，如果程序都没有在运行，图表就完全是平坦的。







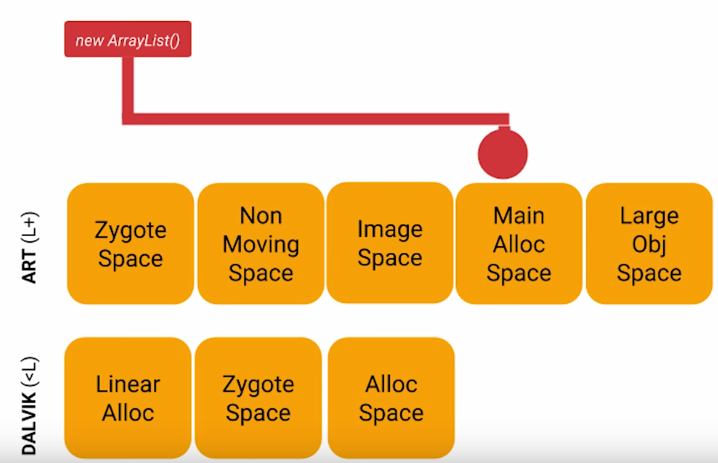
大量的垃圾回收事件

光从性能角度看，这是相当理想的状态，但随着程序分配跟内存释放，图表的分配总量也在跟着变化。如果要装的程序急需大量内存，内存分配也急剧增加，显示在空格里，不然的话，装置内存不足会导致死机。所以对于内存分配，不管什么时候都要特别小心，当垃圾清理开启时就要特别留意内存量，在这个范例中垃圾清理运作良好。另外，如图所示这里也可能有问题，这里有个程序占用了大量内存，然后又一下子释放了刚被占用的内存。生成这些又细又窄的锋线，不断重复，这就是程序在花大量时间运行垃圾清理，运行垃圾清理所用的时间越多，其他可用时间就越少，像播放和发送录音。我们来看下实际情况。

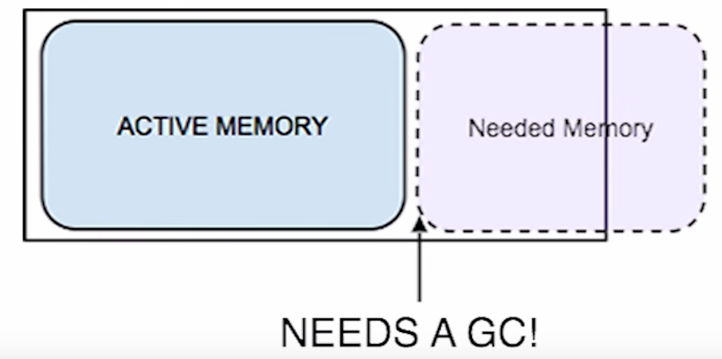
momory monitor已经在监测Sunshine情况了，点击一个日期，看下具体内容，点击返回键，重复这个动作，内存就会持续被占用，如这里所显示的。如果想要新的数据，只要改变几次坐标就行了，看下所得的天气预报，不错，星期三天气明朗。内存被慢慢的占用，最终，内存会被全部占用，这种情况如果持续下去，垃圾清理就会启动，释放大块的内存，这里可以看到变化。要记得，因为Android内存管理系统是固有的，所以垃圾清理不会释放所有的内存。我们的利器，可以强制执行单项的垃圾清理，在Memory Monitor的左上方有个garbage truck工具，单击一下，就会开启单项的垃圾清理，注意图表右边的变化。现在可以多点击几次，再继续点击，所有可被释放的内存都会被释放，装置会恢复到初始状态。接下来我们将了解内存泄露和heap viewer工具。

**内存泄露**

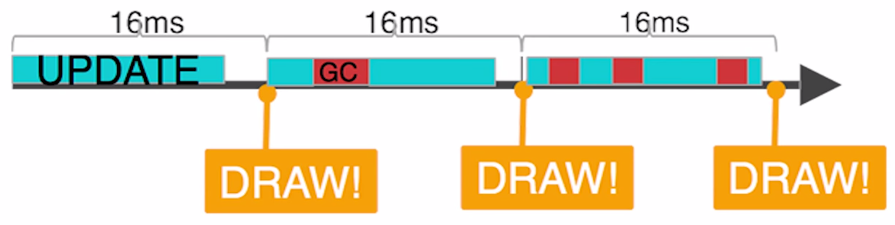
Android的Java语言有个最大的优点，是托管内存环境，对象在创建或消除时不用特别小心。这点尽管不错，但也有些潜在的问题不易被发现。划分到Android运行时的内存堆，是根据声明类型和利于垃圾清理操作的角度来分配的，每一区域都有其预设的内存空间。



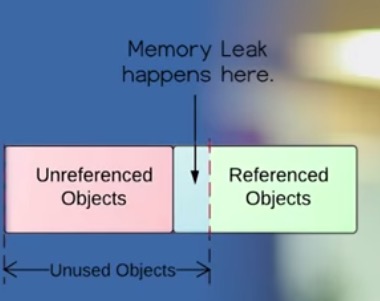
当一个程序所需的总存储空间接近上限，垃圾清理就会启动，删除掉没用的数据，一般情况下不用特别注意垃圾清理的执行。



但是大量的清理动作不断地重复，很快地消耗掉帧像周期，花费在垃圾清理上的时间越多，播放或发送录音等事情的时间就越少。

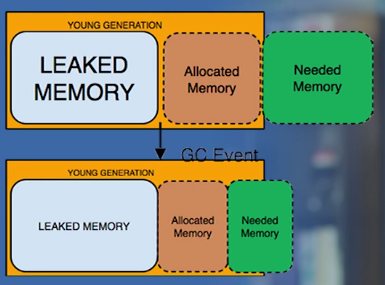


工程师们制造的内存泄露，是垃圾清理运行的常见因素，内存泄露是不能被继续使用的空间，但是垃圾收集器却无法辨别出来，结果他们就一直存在于堆中，占用有效空间，永远无法被删除，随着内存不断泄露，堆中的可用空间就不断变小，这意味着为了执行常用的程序，垃圾清理需要启动的次数越来越多。



内存泄漏表示的是不再用到的对象因为被错误引用而无法进行回收

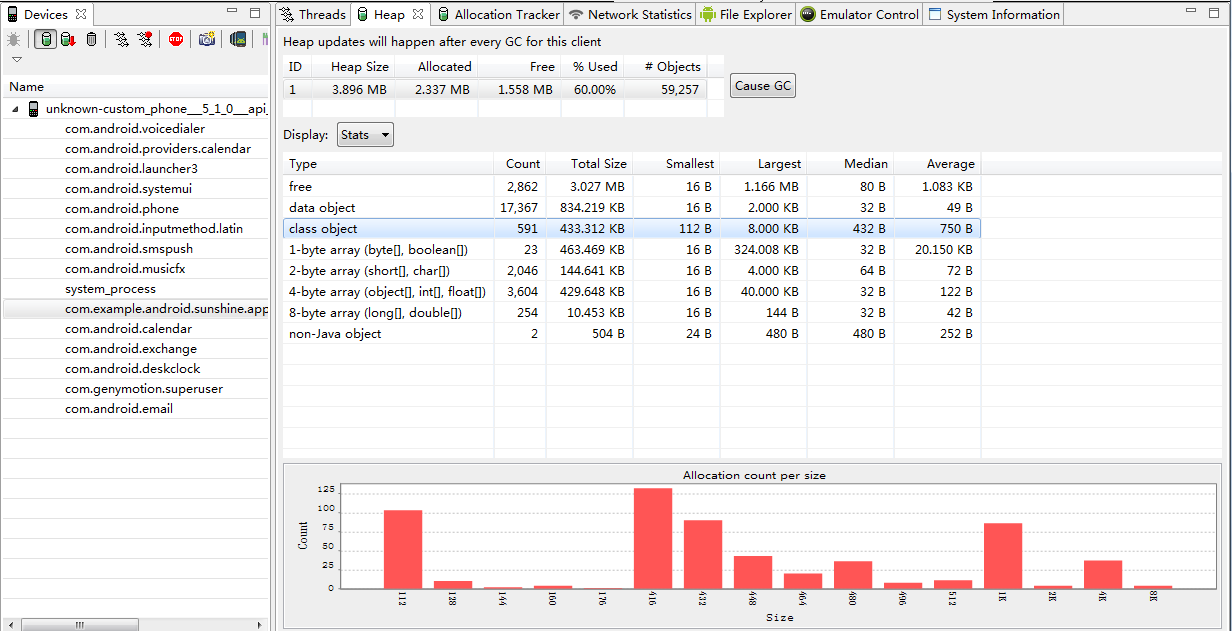
搜索跟修复泄露是个很棘手的问题，有些泄露很容易就会产生，例如对没有使用的对象的循环引用。不过有些也很复杂，例如，在类别载入器安装未完成就强制执行，不管怎样，一个程序想要运行得又快又好，就需留意可能存在的内存泄露。你的代码将允许在各种各样的设备上，又互相结合，不是所有的数据都占用同样的内存，不过，还在有一个简单的工具，可以查看Android SDK中潜在的漏洞。



**Heap Viewer工具**

Heap Viewer是个很简单的工具，利用它可以查看内存状态，以及空间占用率的情况。通过Heap Viewer可知程序在特定时间内的内存使用量，跟原来一样，先在装置上打开Android Studio里的sunshine，在执行start Heap Viewer前，先打开Android Device Monitor。

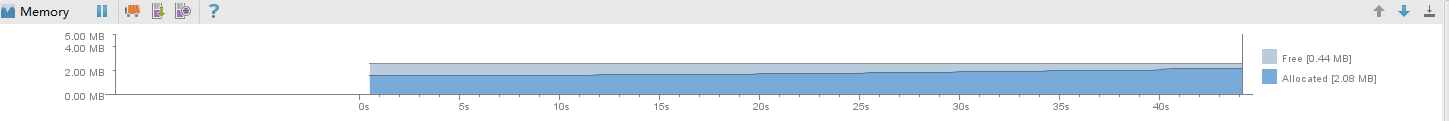
我们看到，每次垃圾清理后，Heap都会更新，点击Cause GC，发现所有的数据都更新了，更新后的表格显示，在Heap上哪些数据是可用的，选中其中任一行数据，就可以看到详细数据，点击class object，屏幕上马上出现大量更新的数据，矩形图列出这一数据内存分配的数量，跟确切的容量。我们这里讨论的是class object，heap viewer可以有效地分析程序在堆中所分配的数据类型，以及数量和大小。这里列出在堆中各别类型程序的总容量，例如，这两个在堆里超过1400的数据组，用掉约1200个千字节，而这个只有27的数据组，却占用了约2个兆字节。heap viewer能够准确地，辨别出程序分配的类型和数量，以及各自在堆中的容量。比方说，这个27的数据组占用了近2兆的字节，可这4个2000的数据组，目前占用了228个千字节。在搜索内存漏洞时，这是个相当不错的工具。



Heap Viewer可以看到当前进程中的Heap Size的情况，分别有哪些类型的数据、占比

**使用Memory Monitor观察内存泄露**

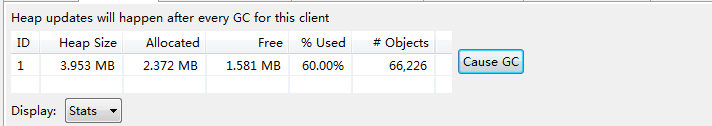
讨论下内存泄露的问题，内存泄露的行踪，常常神出鬼没，常慢慢不动声色的出现，有时要几天或几个星期后，才会被发现。实际上，可能到程序莫名其妙地操作缓慢时，才会发现内存不足的问题。只要用对工具，耐心分析，解决内存泄露不是难事。首先用Memory Monitor，观察漏洞是怎样生成的，在下一个影片中，再利用Heap Viewer做初步确认。举例说明漏洞的生成，以及SDK工具，如何侦测这样微小的漏洞，先把手机旋转几下，然后打开Memory Monitor，这样做的目的是要说明，一个简单的动作就会产生漏洞。像这样不断改变手机方向，就会有漏洞产生，听起来很奇怪，但是借由这一动作，可知漏洞是怎么缓慢且隐秘地产生的。首先，漏洞慢慢吞噬程序内的可用内存，直到GC的启动，再来，值得注意的是由于程序上有漏洞，导致GC无法回收全部垃圾。结果大约30秒后，就会启动第二次GC，当漏洞吞噬所有的可用内存时，Android调整并分配给程序更高的内存上限。这样做的同时，如果漏洞没有修复，内存会不断地被吞噬，结果导致系统无法再配置，手机也就没办法再用了，最后死机。稍等下，第三次的GC就会启动，第四次跟前两次类似，现在这组指令在持续运行，系统分配更多的内存量，可以用同样的方法操作Heap Viewer。



不断旋转屏幕导致内存慢慢被吞噬

**使用Heap Viewer观察内存泄露**

通过Heap Viewer，可知第一次GC仅释放了1.39兆内存，这种结果显示，因为漏洞的存在，垃圾清理无法回收全部垃圾。Heap viewer显示第二次GC后，系统必须经由配置更多的内存，来调整内存量。堆从第一次GC的20兆，增加到32兆，此次Java堆释放了12.9兆，这是，系统不断地为程序配置更多的内存。以上动作如果一再重复，系统终会无法配置内存，程序也就挂了。切记，内存漏洞非常缓慢又不易被发现，需要时间，跟适当的环境来确认，有时，这样的数据，也表示内存的正当存取。比如，处理图片跟照片的程序，表面看似内存在泄露，实际上它针对核心功能的存储器，不停地进行数据评估。因此，要明白内存泄露如何显示在SD上，也要清楚，内存泄露如何显示在拥有SDK的工具上，如Memory Monitor和Heap Viewer。但是，各位可能不知道他们源于何地，以下这些方法可以防止漏洞的出现。利用编码查看程序的寿命，清理不用的文件，接下来，辨别漏洞产生的原因。



每旋转一次，Allocated值不断上升，堆内存值（Heap Size）不断上升

**追踪内存泄露的代码**

查看自定义控件init方法中如下代码：

private void init() {

ListenerCollector collector = new ListenerCollector();

collector.setListener(this, mListener);

}

存储一个Activity中所有视图监听器，这个想法看似无害，但如果你忘了清理它们，你可能会不经意地造成一个缓慢的泄漏。相关代码：

collector.setListener(this, mListener);

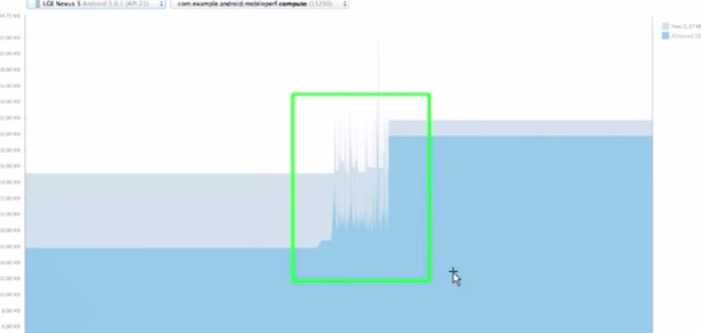
当Activity被销毁和创建时，这一问题被复杂化。在示例中，由于设备的方向变化使一个新的Activity创建，相关联的监听被创建，但是当Activity被销毁时，该监听永远不会被释放。这意味着，监听无法被GC回收，这里导致了内存泄露。当设备旋转并调用当前Activity的onStop方法时，一定要清理所有视图的监听。

**使用Allocation Tracker观察内存泄露**

另外，分配追踪器，可以辨别额外的内存膨胀，这是由于内存的历史浏览记录不断扩充产生的。选择一组仍在堆中的数据或者程序，这组数据堆中，在这个操作里，堆中数据叫作onCreate。这样一来，手机每旋转一次就有新的动作，类似的数据组，基本上就会在堆中膨胀。所以，如果在漏洞存在时旋转手机，垃圾清理无法清除这些数据，就会在堆中产生大量的垃圾。借由分配追踪器，可以弄清这一问题。

**什么是内存抖动？**

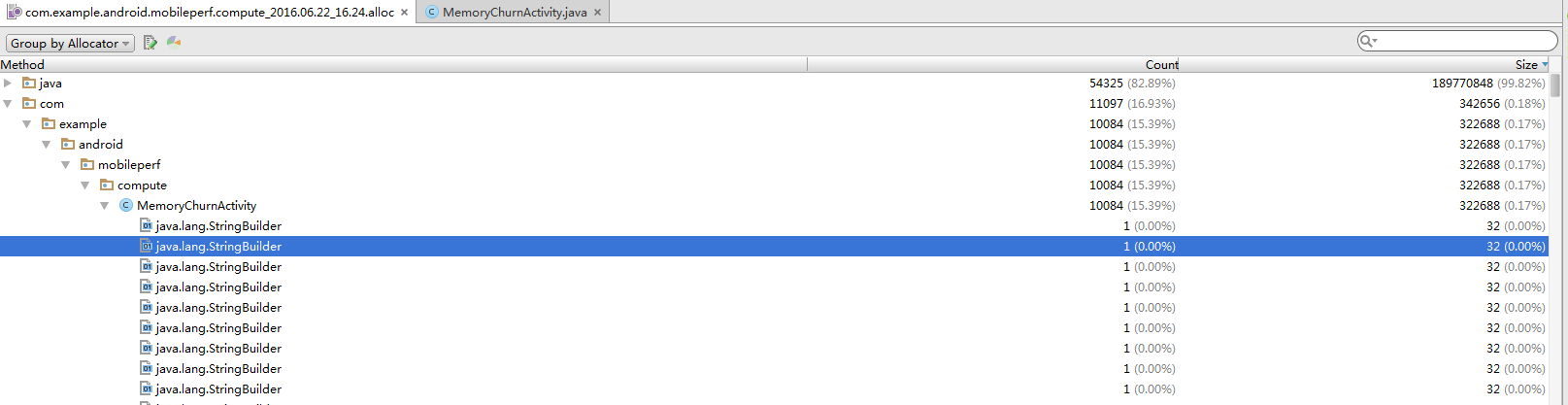
我们解决了哪些讨厌的泄露，现在遇到了更大的问题，内存抖动。要知道，堆内存都有一定的大小，能容纳的数据是有限制的，当Java堆的大小太大时，垃圾收集会启动停止堆中不再应用的对象，来释放内存。现在，内存抖动这个术语可用于描述在极短时间内分配给对象的过程。例如，当你在循环语句中配置一系列临时对象，或者在绘图功能中配置大量对象时，这相当于内循环，当屏幕需要重新绘制或出现动画时，你需要一帧帧使用这些功能，不过它会迅速增加你的堆的压力。这两种情况下，我们都制定了解决方案，可在短时间内创造大量的对象。根据创造的对象的量，或者每个对象的大小，你可能很快就消耗掉所有剩余内存，导致垃圾收集强行开启。随着它们的开启运行，会消耗更多宝贵的帧时间，所以，高性能的应用很有必要，你需要鉴别并从内循环里，取消会被重复执行的代码配置。为了更好的寻找到这些代码配置，Android Studio为此特别打造了一个方便的工具。



内存抖动

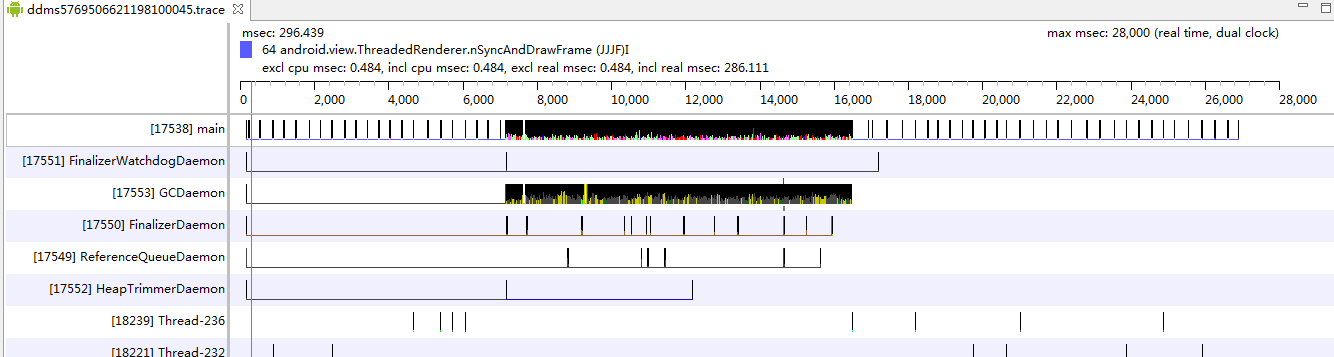
**使用Allocation Tracker**

现在看一下你的应用内存分配图，这能有效的获悉大部分数据到底用在哪里，以及正在分配哪种类型的数据，这能帮你找到现有的不必要分配的数据。可惜Heap Viewer不能显示你的数据具体分配在代码的何处，为此，我们需要一个叫做分配追踪器的工具。和以前一样，我们打开Android Studio Device Monitor，在前台载入Sunshine，打开DDMS视图点击start allocation tracking按钮，然后使用应用，隔一段时间在点击stop allocation tracking按钮。停止之后在DDMS出现了一个列表，这个列表显示了你在使用应用期间，所有的分配情况，这里的每一行都代表不同的分配，allocation order这一栏会提示你，分配进行的具体时间，分配类别这一栏显示了分配数据的类型，以及大小，还有其他信息来告诉你哪个线程具体决定了这一分配。最后，分配站这一栏告诉你代码的哪一个功能实际分配了内存。比如，我们选择整型，测试的值决定了这个整型的分配，如果你点击一个分配，你可以看见完整的调用堆栈。这个表格包含大量信息！



**通过Trace View找出内存抖动**

本次练习，我们来运行内存抖动活动。下面点这个按钮，对数组来点有意思的事情，你会发现跳着舞的海盗会暂停，但最后都会接着跳舞。这就是讨厌的卡顿，让我们解决它吧。通过跟踪显示来剖析这个活动，打开trace view的面板，注意短时间内发生的频繁的垃圾收集活动，可能会伤害到应用的性能。记住，我们还可以采集这个内存监控器图像，这个截屏展示了内存抖动是怎样通过Memory Monitor清晰显示的。

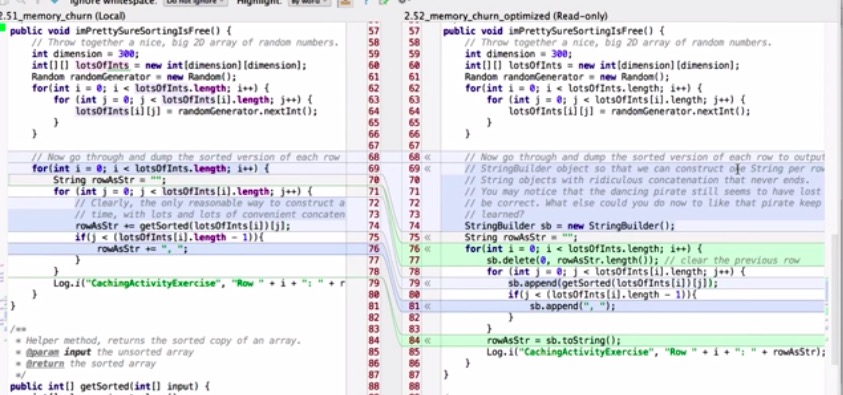


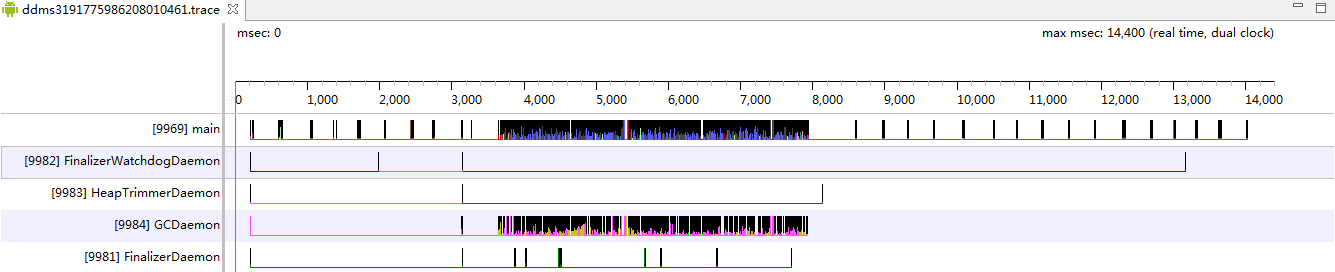
**什么导致了内存抖动**

我们已经使用SDK工具采集足够多的数据，能知道内存抖动情况出现的时间，现在来揪出导致这种情况的代码吧。Trace View给我们提供了一个方法，让我们仔细看一下在主线程里，选择方式时的数据配置文件，当你选择主线程方式时，你会发现反复出现的Java字符串赋值操作，比如这个。再看调用堆栈，我们会更加确定数据队列副本被运用于扩大字符串缓冲。来看MemoryChurnActivity的源代码，正如OnClickListener所显示，我们称此功能为imPrettySureSortingIsFree，让我们来看这个代码。此处的方法叫作imPrettySureSortingIsFree，这个代码产生了新的字符串，通过字符串连接每次都有一个单元值，看一下我说的这个代码的指导提示，但是，出现连接的地方比较特别。这个初看起来似乎没什么问题，为什么这个代码会导致内存抖动？ 频繁使用垃圾清理会造成两种后果，一是，每个单元值的连结都会生成新的字符数组，这是因为，在循环之内骤然接到重复指令组合而成，二是，通过定位追踪器，确认字符数组的膨胀，更新一下数据，在下一节中，向大家介绍所得的结果。

**修改代码减少内存抖动**

我们可以在我们的代码进行小的调整，以防止**内存抖动**。让我们来看看对比图，而不是在一个时间串联一个单元格值打造每一行，让我们使用一个StringBuilder实例，并用一个字符串构造每一行，需要注意的是StringBuilder中的实例化的循环外。因此它的内存分配一次，然后，我们只是作为一个缓冲，在每次循环我们先清除它，然后我们追加，整数的一个字符串来表示对于循环迭代的行。更多细节见导师的笔记到这个代码段，运行memory\_churn\_optimized，确认我们减少的GC在短期时间窗中发生的量，您也可以使用allocation tracker验证。现在对于我们来说，即使修改了代码，海盗动画仍然会出现卡顿的现象，这意味着该处理放到后台处理可能更加合适。





时间变短，且有时间间隔

**工具的特色**

1）Memory Monitor获得内存的动态视图

2）Heap Viewer显示堆内存中存储了什么

3）Allocation Tracker 具体是哪些代码使用了内存