胡凯

- RSS

Search

» RSS

- 首页
- 存档
- Android官方培训课程
- 关于胡凯
- 友情链接

Android性能优化典范 – 第1季

Jan 17th, 2015 | Comments

Android Performance Patterns

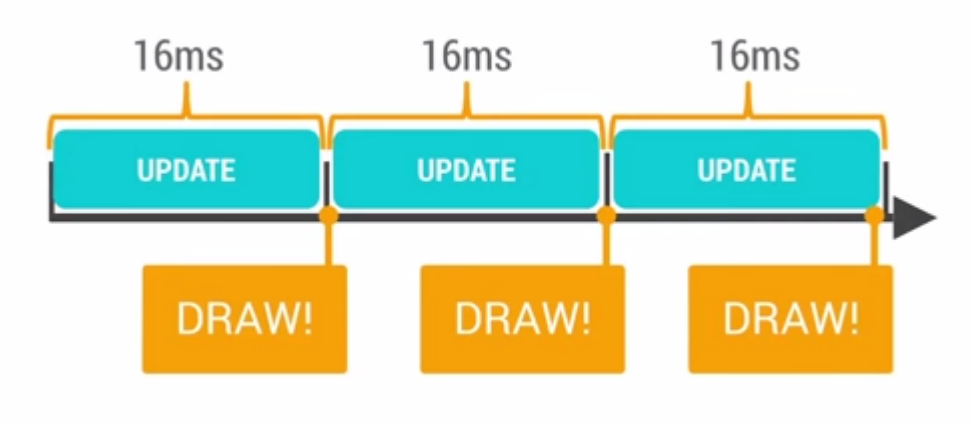
来自Android Developers • 16个视频 • 2,118次观看 • 1小时 2分钟

Android Performance Patterns is a collection of videos focused entirely on helping developers write faster, more performant Android Applications. On one side, it's about peeling back the layers of the Android System, and exposing how things are working under the hood. On the other side, it's about teaching you how the tools work, and what to look for in order to extract the right perf out of your app. But at the end of the day, Android Performance Patterns is all giving you the right resources, at the right time to help make the fastest, smoothest, most awesome experience for your users. And that's the whole point, right? 折叠

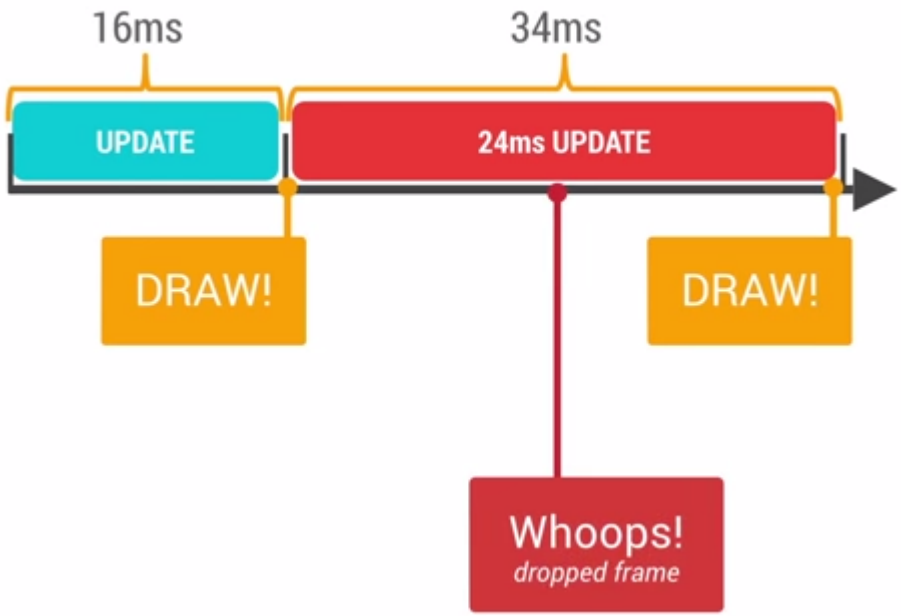
2015新年伊始，Google发布了关于Android性能优化典范的专题，一共16个短视频，每个3-5分钟，帮助开发者创建更快更优秀的Android App。课程专题不仅仅介绍了Android系统中有关性能问题的底层工作原理，同时也介绍了如何通过工具来找出性能问题以及提升性能的建议。主要从三个方面展开，Android的渲染机制，内存与GC，电量优化。下面是对这些问题和建议的总结梳理。

0)Render Performance

大多数用户感知到的卡顿等性能问题的最主要根源都是因为渲染性能。从设计师的角度，他们希望App能够有更多的动画，图片等时尚元素来实现流畅的用户体验。但是Android系统很有可能无法及时完成那些复杂的界面渲染操作。Android系统每隔16ms发出VSYNC信号，触发对UI进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需要的60fps，为了能够实现60fps，这意味着程序的大多数操作都必须在16ms内完成。



如果你的某个操作花费时间是24ms，系统在得到VSYNC信号的时候就无法进行正常渲染，这样就发生了丢帧现象。那么用户在32ms内看到的会是同一帧画面。

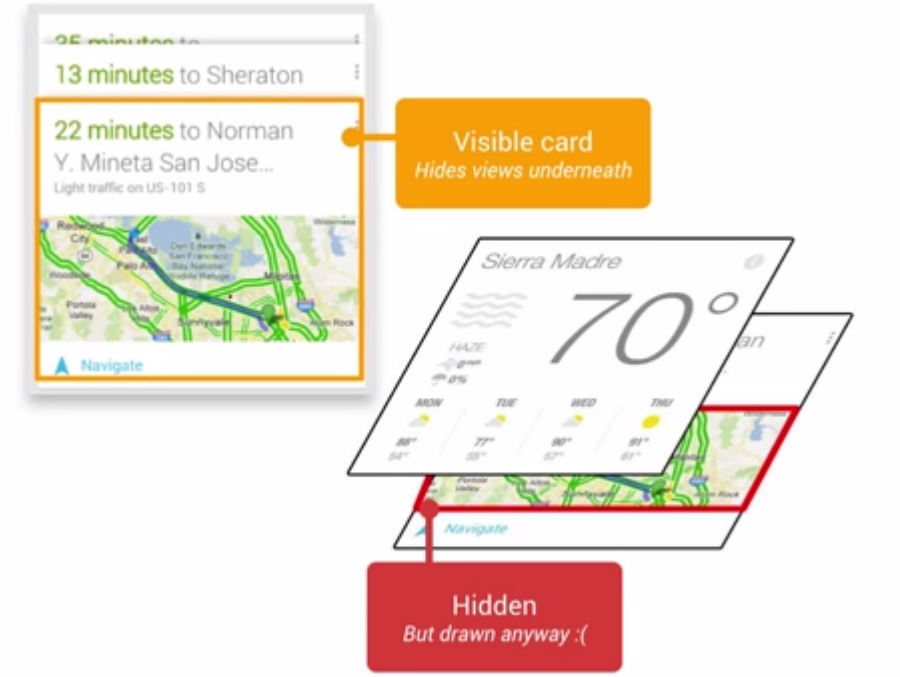


用户容易在UI执行动画或者滑动ListView的时候感知到卡顿不流畅，是因为这里的操作相对复杂，容易发生丢帧的现象，从而感觉卡顿。有很多原因可以导致丢帧，也许是因为你的layout太过复杂，无法在16ms内完成渲染，有可能是因为你的UI上有层叠太多的绘制单元，还有可能是因为动画执行的次数过多。这些都会导致CPU或者GPU负载过重。

我们可以通过一些工具来定位问题，比如可以使用HierarchyViewer来查找Activity中的布局是否过于复杂，也可以使用手机设置里面的开发者选项，打开Show GPU Overdraw等选项进行观察。你还可以使用TraceView来观察CPU的执行情况，更加快捷的找到性能瓶颈。

1)Understanding Overdraw

Overdraw(过度绘制)描述的是屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次的UI结构里面，如果不可见的UI也在做绘制的操作，这就会导致某些像素区域被绘制了多次。这就浪费大量的CPU以及GPU资源。



当设计上追求更华丽的视觉效果的时候，我们就容易陷入采用越来越多的层叠组件来实现这种视觉效果的怪圈。这很容易导致大量的性能问题，为了获得最佳的性能，我们必须尽量减少Overdraw的情况发生。

幸运的是，我们可以通过手机设置里面的开发者选项，打开Show GPU Overdraw的选项，可以观察UI上的Overdraw情况。

蓝色，淡绿，淡红，深红代表了4种不同程度的Overdraw情况，我们的目标就是尽量减少红色Overdraw，看到更多的蓝色区域。

Overdraw有时候是因为你的UI布局存在大量重叠的部分，还有的时候是因为非必须的重叠背景。例如某个Activity有一个背景，然后里面的Layout又有自己的背景，同时子View又分别有自己的背景。仅仅是通过移除非必须的背景图片，这就能够减少大量的红色Overdraw区域，增加蓝色区域的占比。这一措施能够显著提升程序性能。

2)Understanding VSYNC

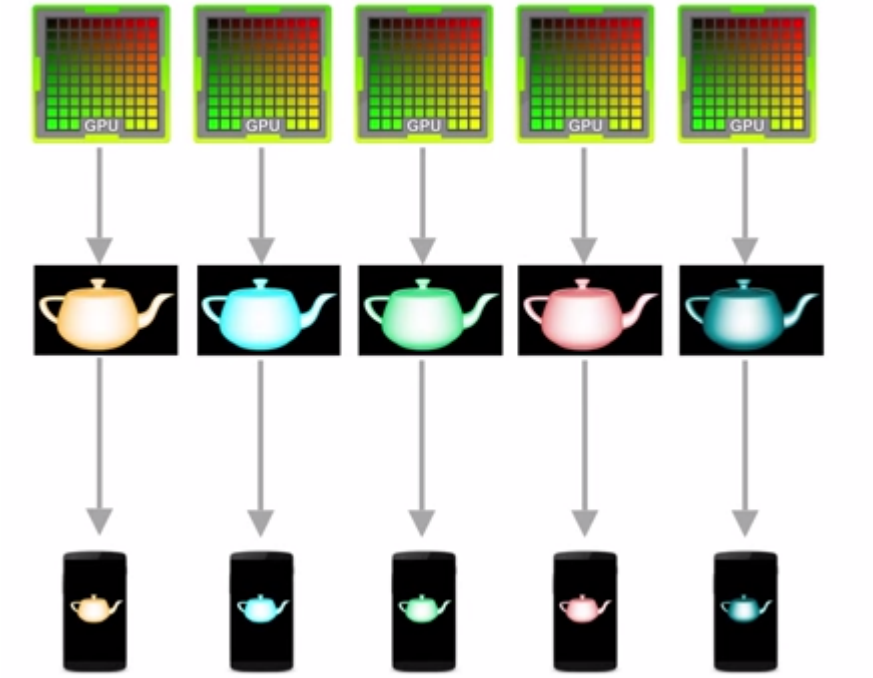
为了理解App是如何进行渲染的，我们必须了解手机硬件是如何工作，那么就必须理解什么是*VSYNC*。

在讲解VSYNC之前，我们需要了解两个相关的概念：

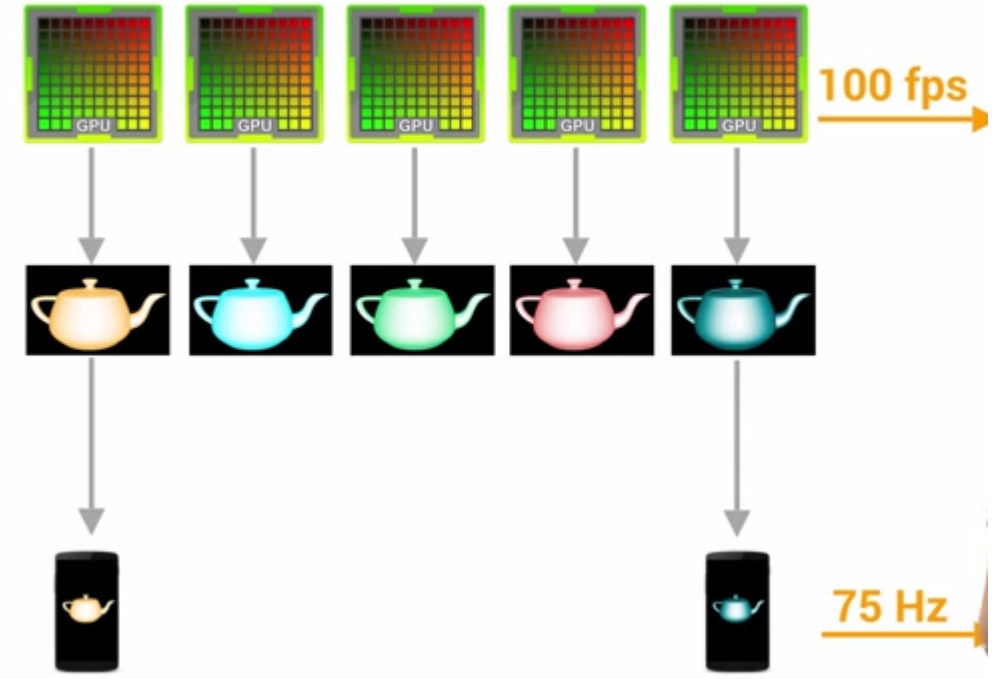
- Refresh Rate：代表了屏幕在一秒内刷新屏幕的次数，这取决于硬件的固定参数，例如60Hz。

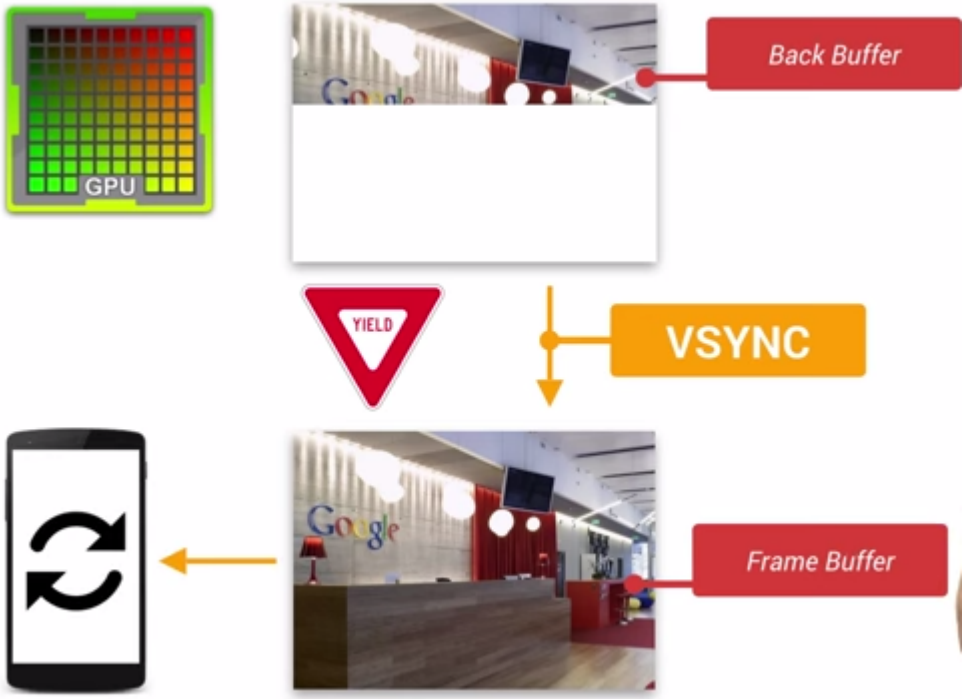
- Frame Rate：代表了GPU在一秒内绘制操作的帧数，例如30fps，60fps。

GPU会获取图形数据进行渲染，然后硬件负责把渲染后的内容呈现到屏幕上，他们两者不停的进行协作。



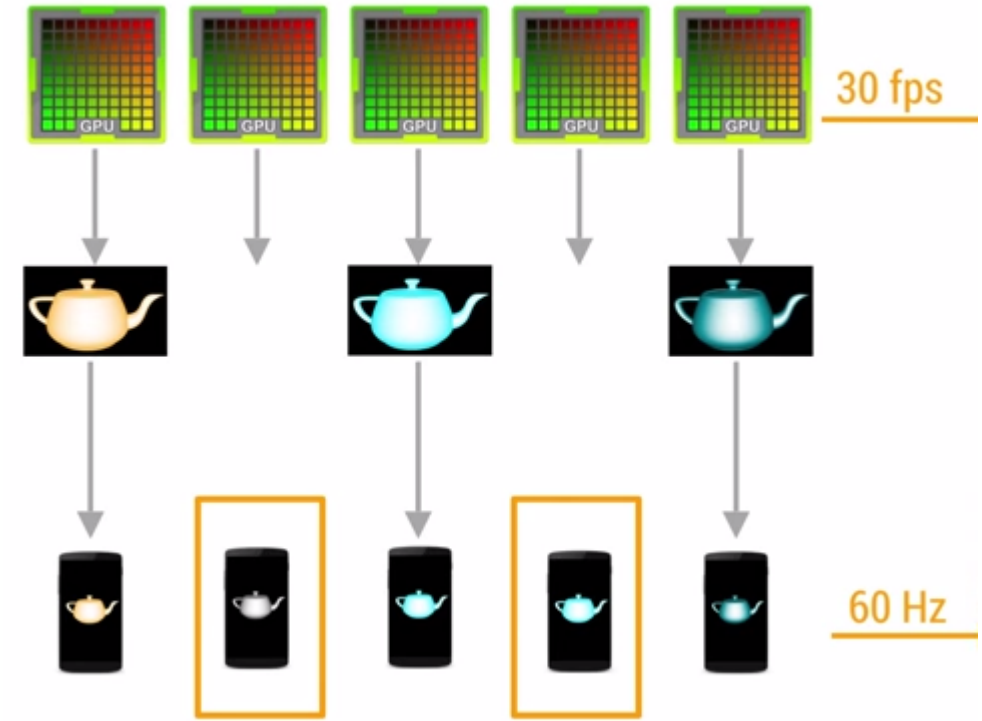
不幸的是，刷新频率和帧率并不是总能够保持相同的节奏。如果发生帧率与刷新频率不一致的情况，就会容易出现**Tearing**的现象(画面上下两部分显示内容发生断裂，来自不同的两帧数据发生重叠)。


GPU
GPU
GPU
GPU
GPU
100 fps
75 Hz



理解图像渲染里面的双重与三重缓存机制，这个概念比较复杂，请移步查看这里：http://source.android.com/devices/graphics/index.html，还有这里http://article.yeeyan.org/view/37503/304664。

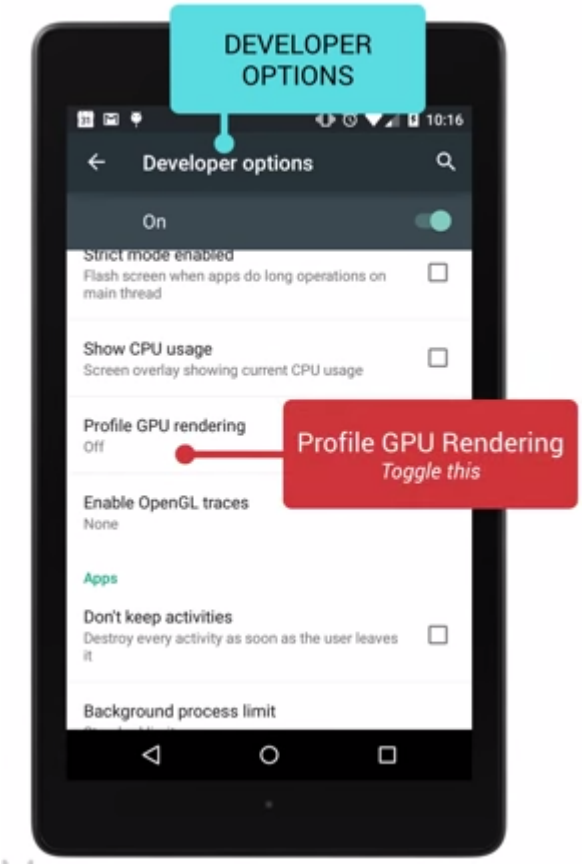
通常来说，帧率超过刷新频率只是一种理想的状况，在超过60fps的情况下，GPU所产生的帧数据会因为等待VSYNC的刷新信息而被Hold住，这样能够保持每次刷新都有实际的新的数据可以显示。但是我们遇到更多的情况是帧率小于刷新频率。



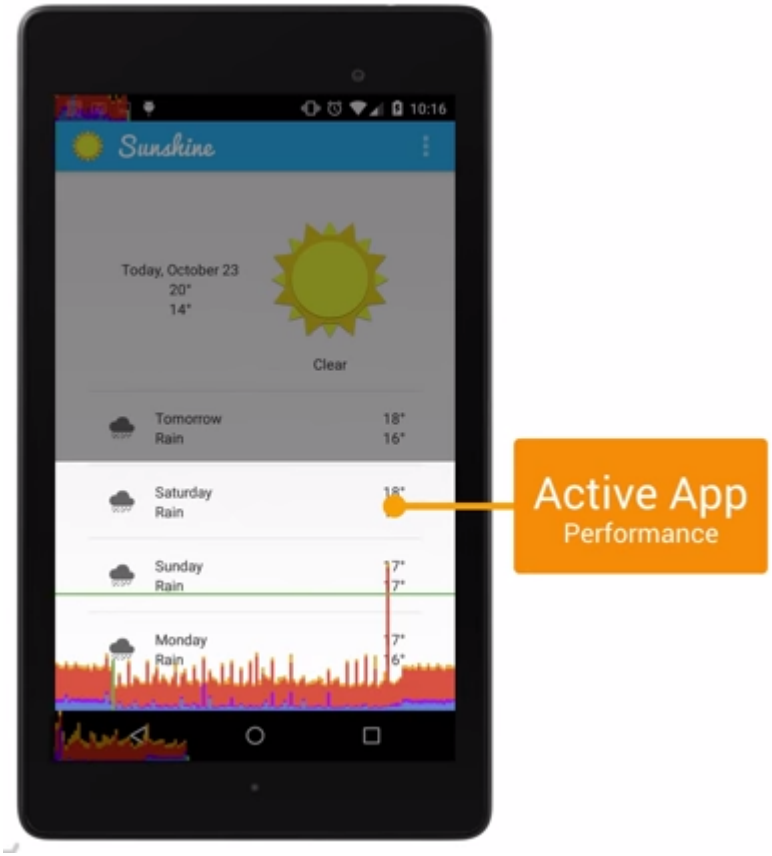
在这种情况下，某些帧显示的画面内容就会与上一帧的画面相同。糟糕的事情是，帧率从超过60fps突然掉到60fps以下，这样就会发生LAG，JANK，HITCHING等卡顿掉帧的不顺滑的情况。这也是用户感受不好的原因所在。

3)Tool:Profile GPU Rendering

性能问题如此的麻烦，幸好我们可以有工具来进行调试。打开手机里面的开发者选项，选择Profile GPU Rendering，选中On screen as bars的选项。

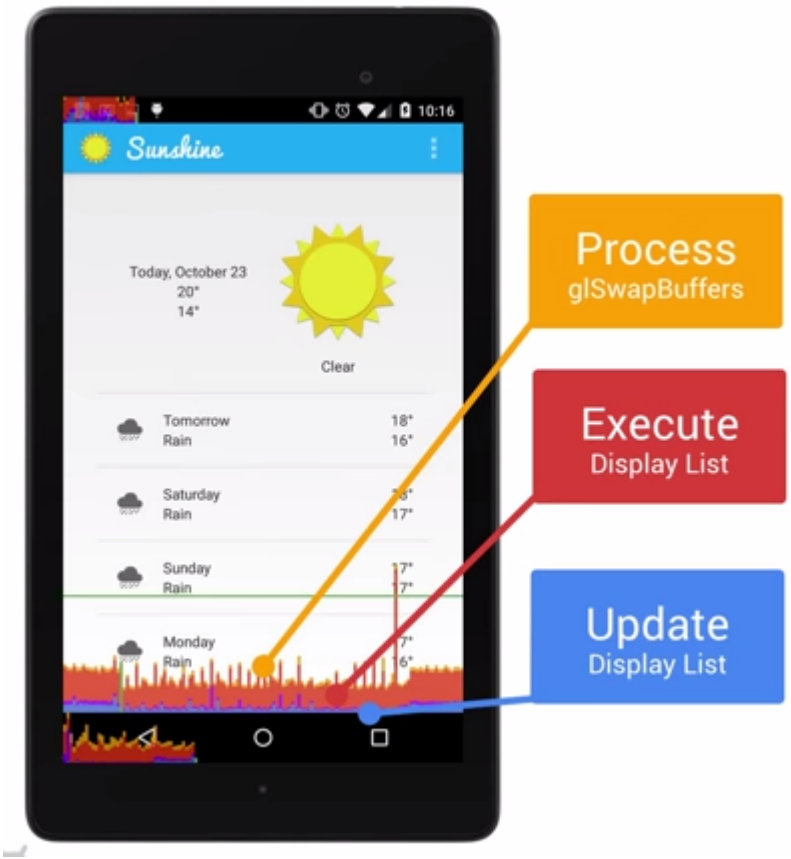


选择了这样以后，我们可以在手机画面上看到丰富的GPU绘制图形信息，分别关于StatusBar，NavBar，激活的程序Activity区域的GPU Rending信息。



随着界面的刷新，界面上会滚动显示垂直的柱状图来表示每帧画面所需要渲染的时间，柱状图越高表示花费的渲染时间越长。

中间有一根绿色的横线，代表16ms，我们需要确保每一帧花费的总时间都低于这条横线，这样才能够避免出现卡顿的问题。



每一条柱状线都包含三部分，蓝色代表测量绘制Display List的时间，红色代表OpenGL渲染Display List所需要的时间，黄色代表CPU等待GPU处理的时间。

4)Why 60fps?

我们通常都会提到60fps与16ms，可是知道为何会是以程序是否达到60fps来作为App性能的衡量标准吗？这是因为人眼与大脑之间的协作无法感知超过60fps的画面更新。

12fps大概类似手动快速翻动书籍的帧率，这明显是可以感知到不够顺滑的。24fps使得人眼感知的是连续线性的运动，这其实是归功于运动模糊的效果。24fps是电影胶圈通常使用的帧率，因为这个帧率已经足够支撑大部分电影画面需要表达的内容，同时能够最大的减少费用支出。但是低于30fps是无法顺畅表现绚丽的画面内容的，此时就需要用到60fps来达到想要的效果，当然超过60fps是没有必要的。

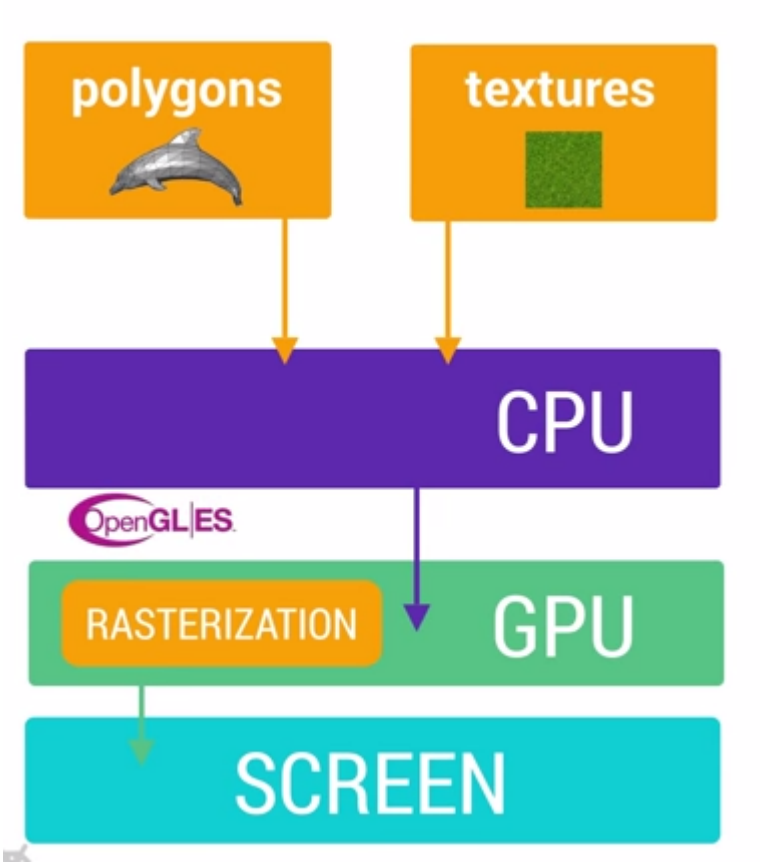
开发app的性能目标就是保持60fps，这意味着每一帧你只有16ms=1000/60的时间来处理所有的任务。

5)Android, UI and the GPU

了解Android是如何利用GPU进行画面渲染有助于我们更好的理解性能问题。那么一个最实际的问题是：activity的画面是如何绘制到屏幕上的？那些复杂的XML布局文件又是如何能够被识别并绘制出来的?

Resterization栅格化是绘制那些Button，Shape，Path，String，Bitmap等组件最基础的操作。它把那些组件拆分到不同的像素上进行显示。这是一个很费时的操作，GPU的引入就是为了加快栅格化的操作。

CPU负责把UI组件计算成Polygons，Texture纹理，然后交给GPU进行栅格化渲染。



然而每次从CPU转移到GPU是一件很麻烦的事情，所幸的是OpenGL ES可以把那些需要渲染的纹理Hold在GPU Memory里面，在下次需要渲染的时候直接进行操作。所以如果你更新了GPU所hold住的纹理内容，那么之前保存的状态就丢失了。

在Android里面那些由主题所提供的资源，例如Bitmaps，Drawables都是一起打包到统一的Texture纹理当中，然后再传递到GPU里面，这意味着每次你需要使用这些资源的时候，都是直接从纹理里面进行获取渲染的。当然随着UI组件的越来越丰富，有了更多演变的形态。例如显示图片的时候，需要先经过CPU的计算加载到内存中，然后传递给GPU进行渲染。文字的显示更加复杂，需要先经过CPU换算成纹理，然后再交给GPU进行渲染，回到CPU绘制单个字符的时候，再重新引用经过GPU渲染的内容。动画则是一个更加复杂的操作流程。

为了能够使得App流畅，我们需要在每一帧16ms以内处理完所有的CPU与GPU计算，绘制，渲染等等操作。

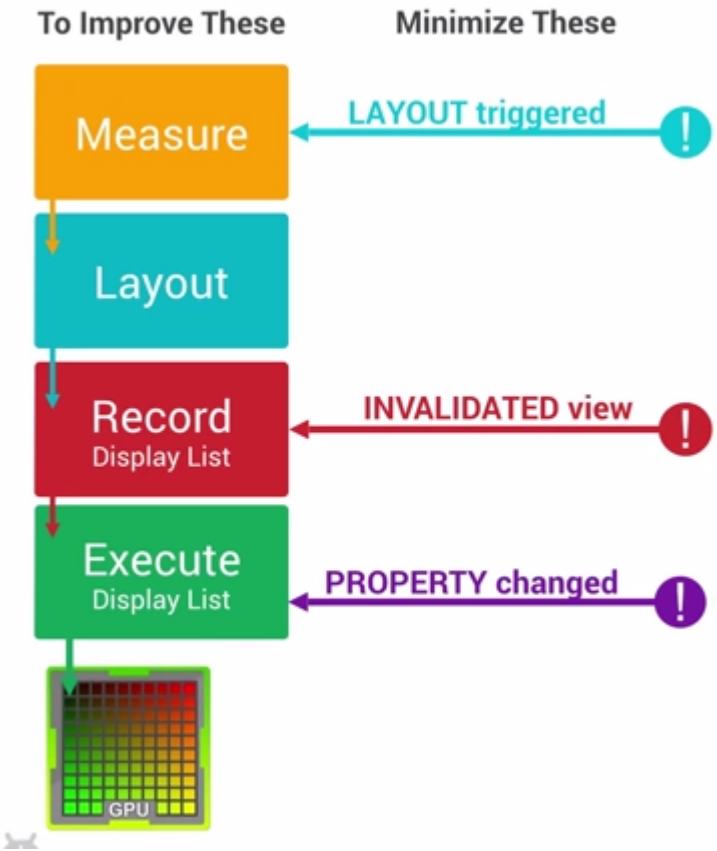
6)Invalidations, Layouts, and Performance

顺滑精妙的动画是app设计里面最重要的元素之一，这些动画能够显著提升用户体验。下面会讲解Android系统是如何处理UI组件的更新操作的。

通常来说，Android需要把XML布局文件转换成GPU能够识别并绘制的对象。这个操作是在**DisplayList**的帮助下完成的。DisplayList持有所有将要交给GPU绘制到屏幕上的数据信息。

在某个View第一次需要被渲染时，DisplayList会因此而被创建，当这个View要显示到屏幕上时，我们会执行GPU的绘制指令来进行渲染。如果你在后续有执行类似移动这个View的位置等操作而需要再次渲染这个View时，我们就仅仅需要额外操作一次渲染指令就够了。然而如果你修改了View中的某些可见组件，那么之前的DisplayList就无法继续使用了，我们需要回头重新创建一个DisplayList并且重新执行渲染指令并更新到屏幕上。

需要注意的是：任何时候View中的绘制内容发生变化时，都会重新执行创建DisplayList，渲染DisplayList，更新到屏幕上等一系列操作。这个流程的表现性能取决于你的View的复杂程度，View的状态变化以及渲染管道的执行性能。举个例子，假设某个Button的大小需要增大到目前的两倍，在增大Button大小之前，需要通过父View重新计算并摆放其他子View的位置。修改View的大小会触发整个HierarcyView的重新计算大小的操作。如果是修改View的位置则会触发HierarchView重新计算其他View的位置。如果布局很复杂，这就会很容易导致严重的性能问题。我们需要尽量减少Overdraw。



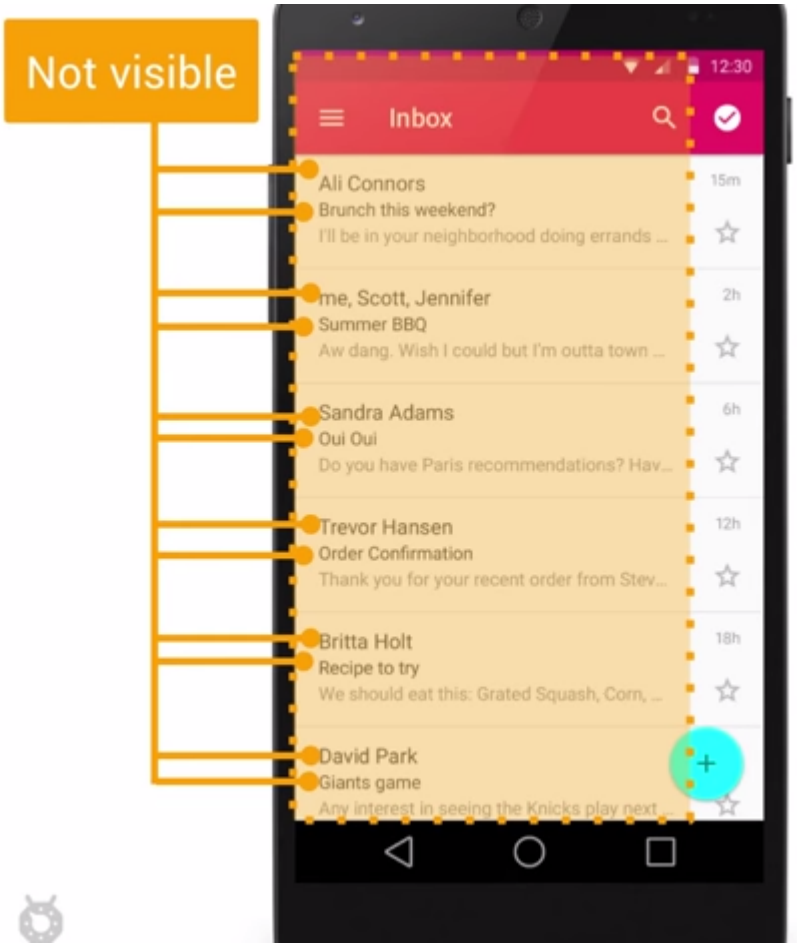
我们可以通过前面介绍的Monitor GPU Rendering来查看渲染的表现性能如何，另外也可以通过开发者选项里面的Show GPU view updates来查看视图更新的操作，最后我们还可以通过HierarchyViewer这个工具来查看布局，使得布局尽量扁平化，移除非必需的UI组件，这些操作能够减少Measure，Layout的计算时间。

7)Overdraw, Cliprect, QuickReject

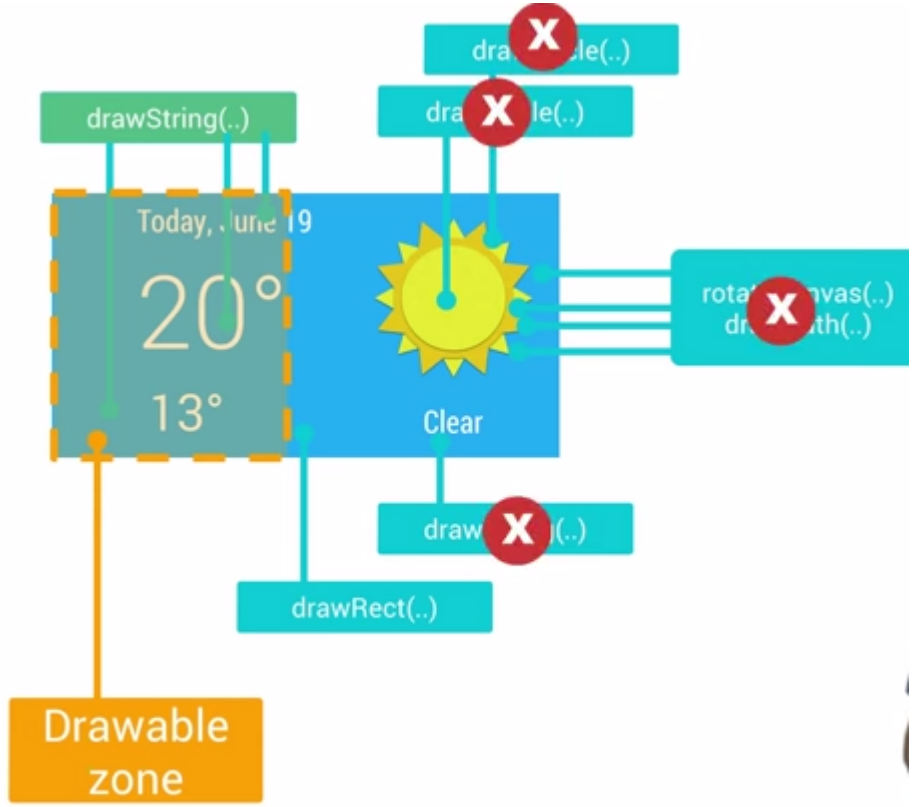
引起性能问题的一个很重要的方面是因为过多复杂的绘制操作。我们可以通过工具来检测并修复标准UI组件的Overdraw问题，但是针对高度自定义的UI组件则显得有些力不从心。

有一个窍门是我们可以通过执行几个APIs方法来显著提升绘制操作的性能。前面有提到过，非可见的UI组件进行绘制更新会导致Overdraw。例如Nav Drawer从前置可见的Activity滑出之后，如果还继续绘制那些在Nav Drawer里面不可见的UI组件，这就导致了Overdraw。为了解决这个问

题，Android系统会通过避免绘制那些完全不可见的组件来尽量减少Overdraw。那些Nav Drawer里面不可见的View就不会被执行浪费资源。



但是不幸的是，对于那些过于复杂的自定义的View(重写了onDraw方法)，Android系统无法检测具体在onDraw里面会执行什么操作，系统无法监控并自动优化，也就无法避免Overdraw了。但是我们可以通过canvas.clipRect()来帮助系统识别那些可见的区域。这个方法可以指定一块矩形区域，只有在这个区域内才会被绘制，其他的区域会被忽视。这个API可以很好的帮助那些有多组重叠组件的自定义View来控制显示的区域。同时clipRect方法还可以帮助节约CPU与GPU资源，在clipRect区域之外的绘制指令都不会被执行，那些部分内容在矩形区域内的组件，仍然会得到绘制。

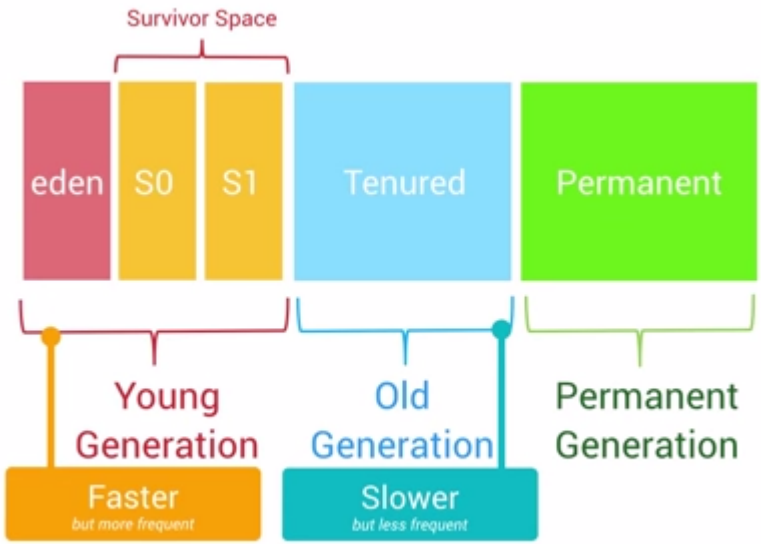


除了clipRect方法之外，我们还可以使用canvas.quickreject()来判断是否没和某个矩形相交，从而跳过那些非矩形区域内的绘制操作。做了那些优化之后，我们可以通过上面介绍的Show GPU Overdraw来查看效果。

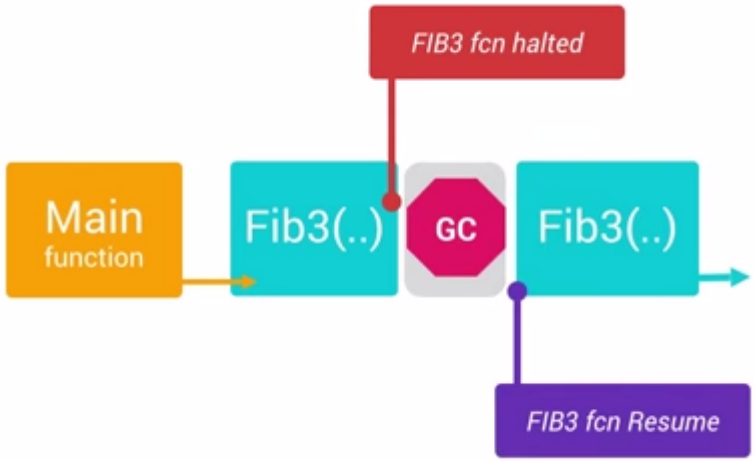
8)Memory Churn and performance

虽然Android有自动管理内存的机制，但是对内存的不恰当使用仍然容易引起严重的性能问题。在同一帧里面创建过多的对象是件需要特别引起注意的事情。

Android系统里面有一个**Generational Heap Memory**的模型，系统会根据内存中不同的内存数据类型分别执行不同的GC操作。例如，最近刚分配的对象会放在Young Generation区域，这个区域的对象通常都是会快速被创建并且很快被销毁回收的，同时这个区域的GC操作速度也是比Old Generation区域的GC操作速度更快的。



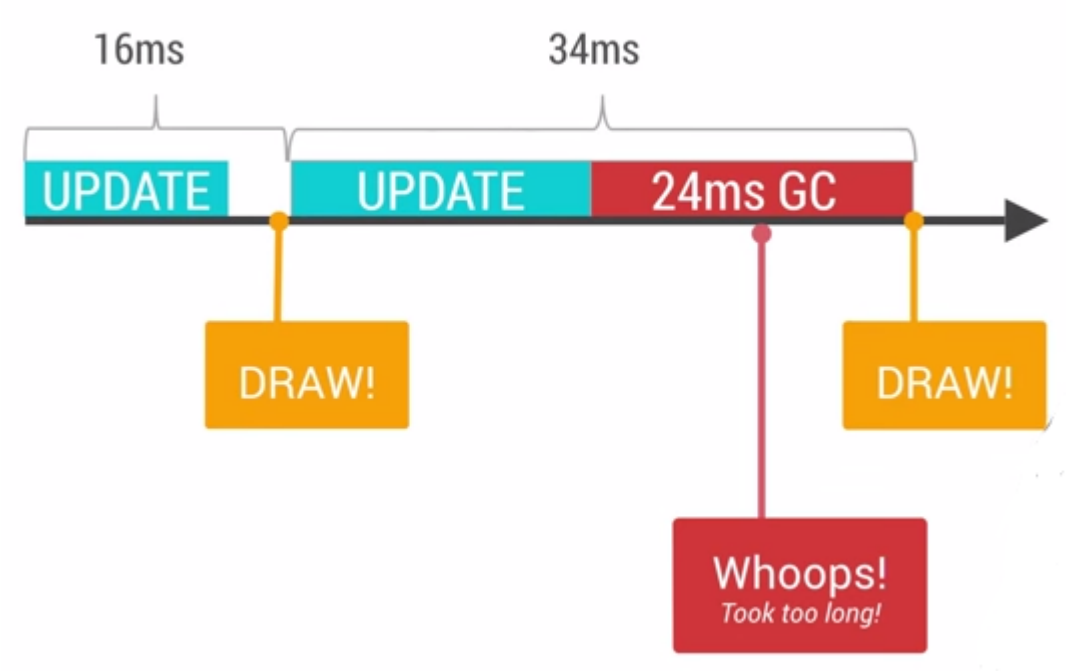
除了速度差异之外，执行GC操作的时候，所有线程的任何操作都会需要暂停，等待GC操作完成之后，其他操作才能够继续运行。



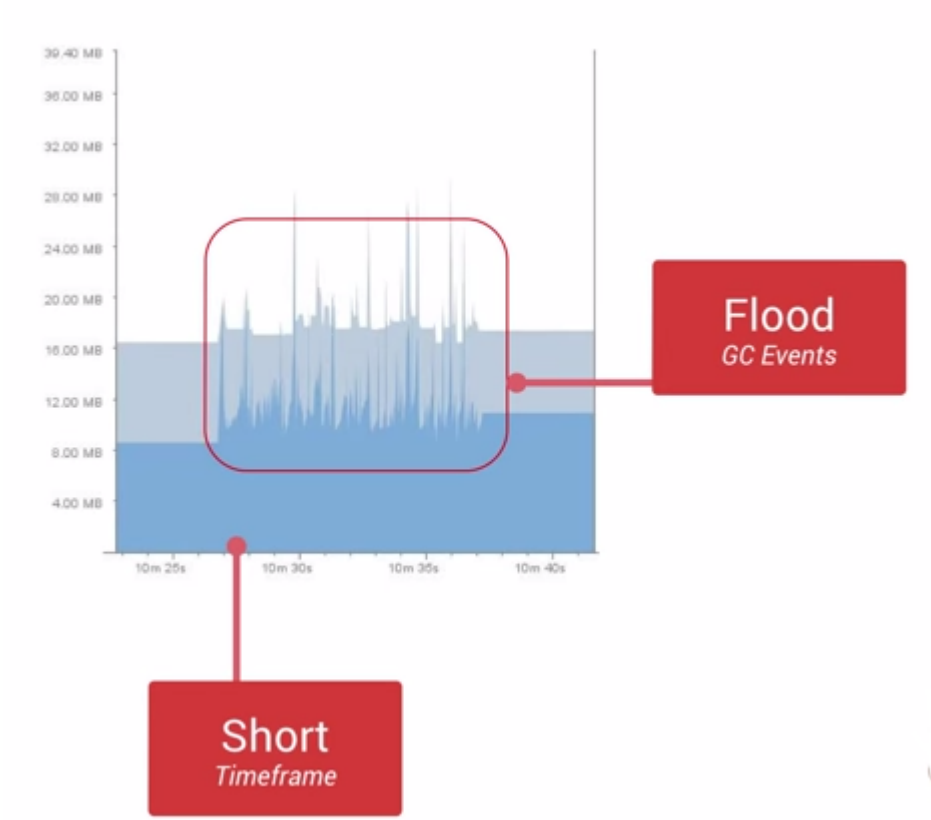
通常来说，单个的GC并不会占用太多时间，但是大量不停的GC操作则会显著占用帧间隔时间(16ms)。如果在帧间隔时间里面做了过多的GC操作，那么自然其他类似计算，渲染等操作的可用时间就变得少了。

导致GC频繁执行有两个原因：

- **Memory Churn内存抖动**，内存抖动是因为大量的对象被创建又在短时间内马上被释放。
- 瞬间产生大量的对象会严重占用Young Generation的内存区域，当达到阀值，剩余空间不够的时候，也会触发GC。即使每次分配的对象占用了很少的内存，但是他们叠加在一起会增加Heap的压力，从而触发更多其他类型的GC。这个操作有可能会影响到帧率，并使得用户感知到性能问题。



解决上面的问题有简洁直观方法，如果你在**Memory Monitor**里面查看到短时间发生了多次内存的涨跌，这意味着很有可能发生了内存抖动。



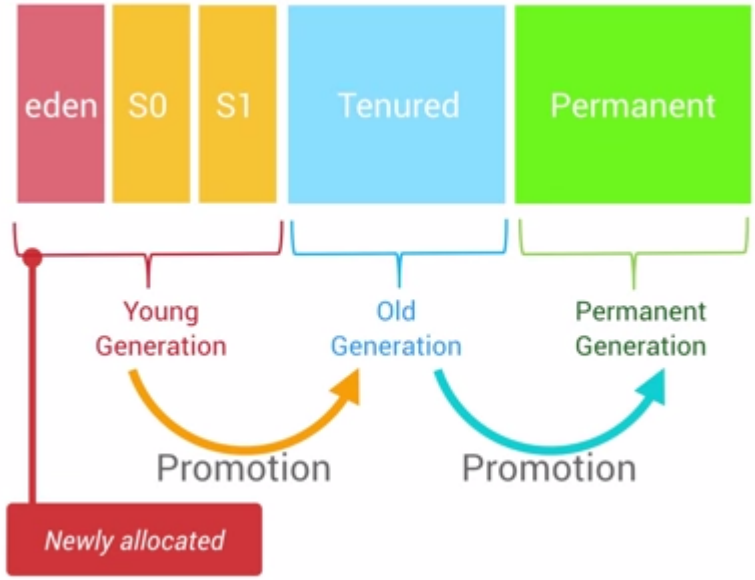
同时我们还可以通过**Allocation Tracker**来查看在短时间内，同一个栈中不断进出的相同对象。这是内存抖动的典型信号之一。

当你大致定位问题之后，接下去的问题修复也就显得相对直接简单了。例如，你需要避免在for循环里面分配对象占用内存，需要尝试把对象的创建移到循环体之外，自定义View中的onDraw方法也需要引起注意，每次屏幕发生绘制以及动画执行过程中，onDraw方法都会被调用到，避免在onDraw方法里面执行复杂的操作，避免创建对象。对于那些无法避免需要创建对象的情况，我们可以考虑对象池模型，通过对象池来解决频繁创建与销毁的问题，但是这里需要注意结束使用之后，需要手动释放对象池中的对象。

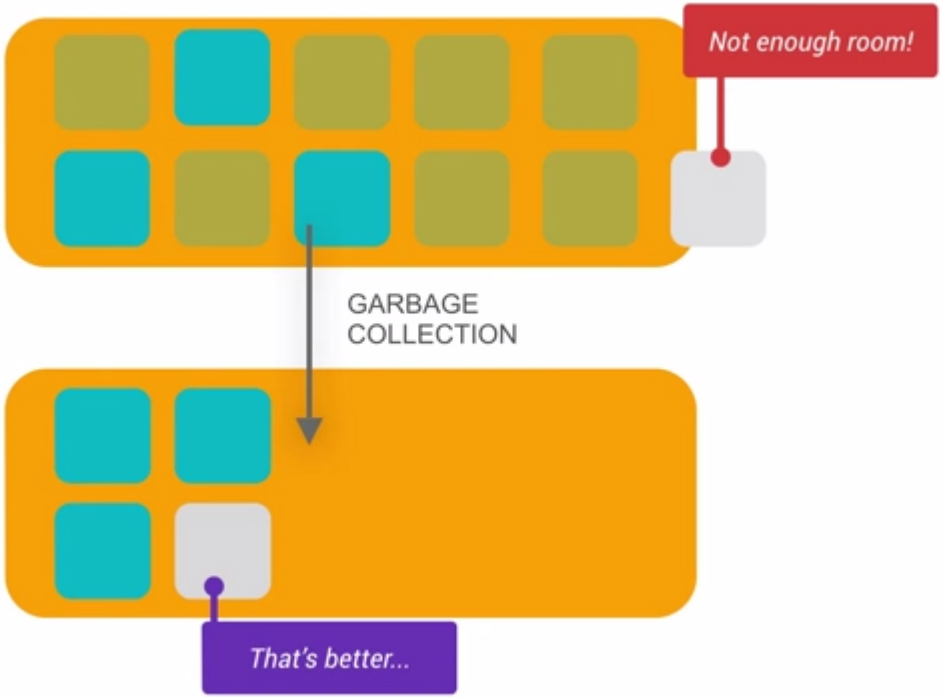
9)Garbage Collection in Android

JVM的回收机制给开发人员带来很大的好处，不用时刻处理对象的分配与回收，可以更加专注于更加高级的代码实现。相比起Java，C与C++等语言具备更高的执行效率，他们需要开发人员自己关注对象的分配与回收，但是在一个庞大的系统当中，还是免不了经常发生部分对象忘记回收的情况，这就是内存泄漏。

原始JVM中的GC机制在Android中得到了很大程度上的优化。Android里面是一个三级Generation的内存模型，最近分配的对象会存放在Young Generation区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到Old Generation，最后到Permanent Generation区域。



每一个级别的内存区域都有固定的大小，此后不断有新的对象被分配到此区域，当这些对象总的大小快达到这一级别内存区域的阀值时，会触发GC的操作，以便腾出空间来存放其他新的对象。



前面提到过每次GC发生的时候，所有的线程都是暂停状态的。GC所占用的时间和它是哪一个Generation也有关系，Young Generation的每次GC操作时间是最短的，Old Generation其次，Permanent Generation最长。执行时间的长短也和当前Generation中的对象数量有关，遍历查找20000个对象比起遍历50个对象自然是要慢很多的。

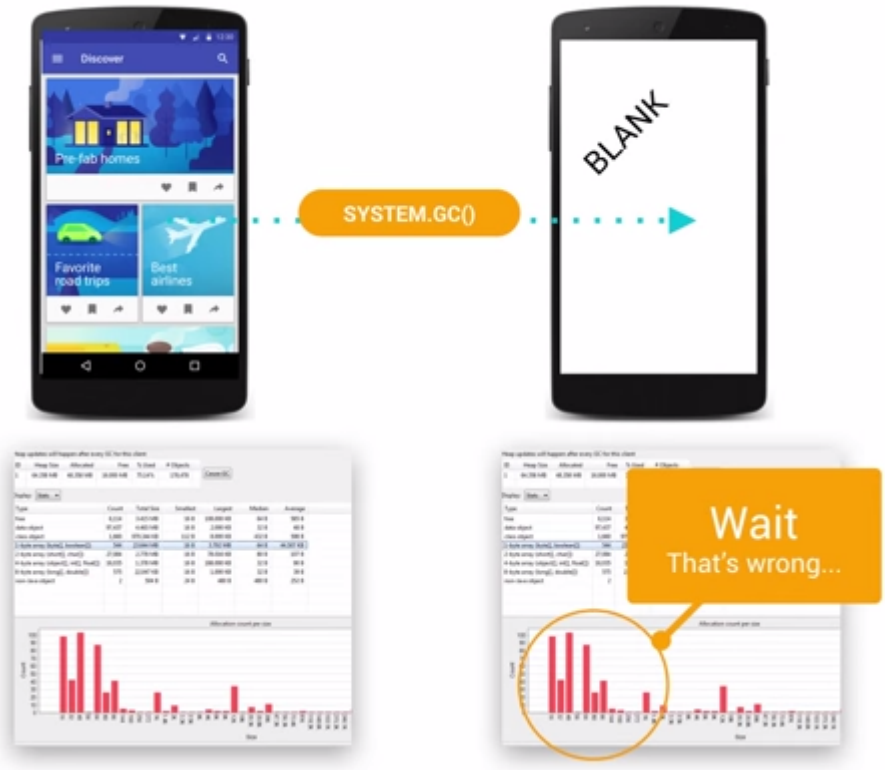
虽然Google的工程师在尽量缩短每次GC所花费的时间，但是特别注意GC引起的性能问题还是很有必要。如果不小心在最小的for循环单元里面执行了创建对象的操作，这将很容易引起GC并导致性能问题。通过Memory Monitor我们可以查看到内存的占用情况，每一次瞬间的内存降低都是因为此时发生了GC操作，如果在短时间内发生大量的内存上涨与降低的事件，这说明很有可能这里有性能问题。我们还可以通过**Heap and Allocation Tracker**工具来查看此时内存中分配的到底有哪些对象。

10)Performance Cost of Memory Leaks

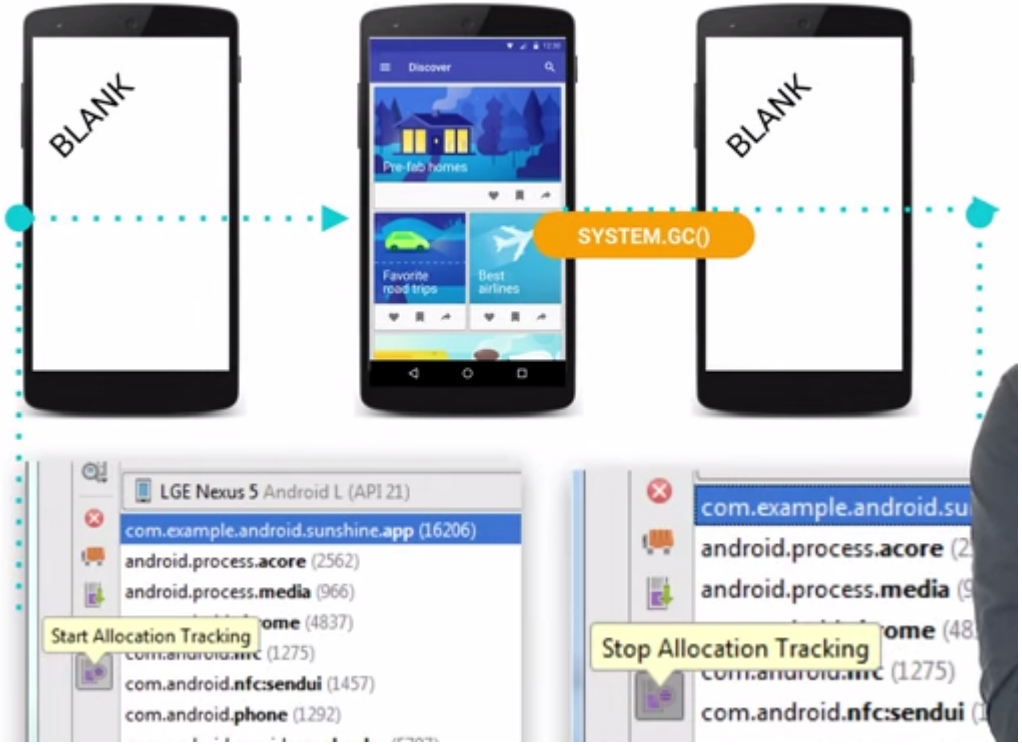
虽然Java有自动回收的机制，可是这不意味着Java中不存在内存泄漏的问题，而内存泄漏会很容易导致严重的性能问题。

内存泄漏指的是那些程序不再使用的对象无法被GC识别，这样就导致这个对象一直留在内存当中，占用了宝贵的内存空间。显然，这还使得每级Generation的内存区域可用空间变小，GC就会更容易被触发，从而引起性能问题。

寻找内存泄漏并修复这个漏洞是件很棘手的事情，你需要对执行的代码很熟悉，清楚的知道在特定环境下是如何运行的，然后仔细排查。例如，你想知道程序中的某个activity退出的时候，它之前所占用的内存是否有完整的释放干净了？首先你需要在activity处于前台的时候使用Heap Tool获取一份当前状态的内存快照，然后你需要创建一个几乎不这么占用内存的空白activity用来给前一个Activity进行跳转，其次在跳转到这个空白的activity的时候主动调用System.gc()方法来确保触发一个GC操作。最后，如果前面这个activity的内存都有全部正确释放，那么在空白activity被启动之后的内存快照中应该不会有前面那个activity中的任何对象了。



如果你发现在空白activity的内存快照中有一些可疑的没有被释放的对象存在，那么接下去就应该使用**Alocation Track Tool**来仔细查找具体的可疑对象。我们可以从空白activity开始监听，启动到观察activity，然后再回到空白activity结束监听。这样操作以后，我们可以仔细观察那些对象，找出内存泄漏的真凶。



11)Memory Performance

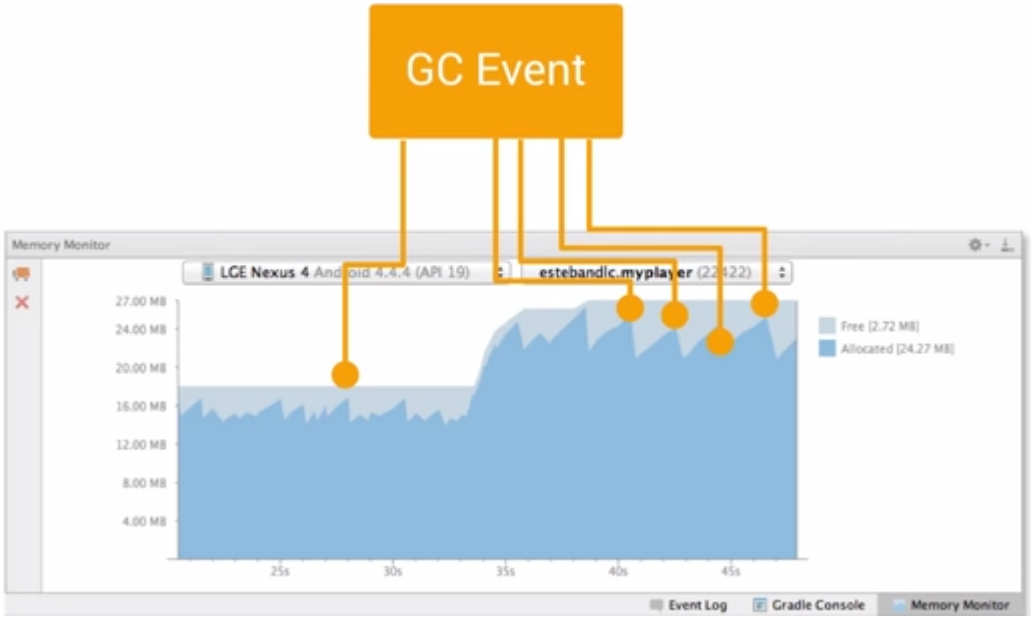
通常来说，Android对GC做了大量的优化操作，虽然执行GC操作的时候会暂停其他任务，可是大多数情况下，GC操作还是相对很安静并且高效的。但是如果我们对内存的使用不恰当，导致GC频繁执行，这样就会引起不小的性能问题。

为了寻找内存的性能问题，Android Studio提供了工具来帮助开发者。

- **Memory Monitor**：查看整个app所占用的内存，以及发生GC的时刻，短时间内发生大量的GC操作是一个危险的信号。
- **Allocation Tracker**：使用此工具来追踪内存的分配，前面有提到过。
- **Heap Tool**：查看当前内存快照，便于对比分析哪些对象有可能是泄漏了的，请参考前面的Case。

12)Tool – Memory Monitor

Android Studio中的Memory Monitor可以很好的帮助我们查看程序的内存使用情况。



13)Battery Performance

电量其实是目前手持设备最宝贵的资源之一，大多数设备都需要不断的充电来维持继续使用。不幸的是，对于开发者来说，电量优化是他们最后才会考虑的的事情。但是可以确定的是，千万不能让你的应用成为消耗电量的大户。

Purdue University研究了最受欢迎的一些应用的电量消耗，平均只有30%左右的电量是被程序最核心的方法例如绘制图片，摆放布局等等所使用掉的，剩下的70%左右的电量是被上报数据，检查位置信息，定时检索后台广告信息所使用掉的。如何平衡这两者的电量消耗，就显得非常重要了。

有下面一些措施能够显著减少电量的消耗：

- 我们应该尽量减少唤醒屏幕的次数与持续的时间，使用WakeLock来处理唤醒的问题，能够正确执行唤醒操作并根据设定及时关闭操作进入睡眠状态。
- 某些非必须马上执行的操作，例如上传歌曲，图片处理等，可以等到设备处于充电状态或者电量充足的时候才进行。
- 触发网络请求的操作，每次都会保持无线信号持续一段时间，我们可以把零散的网络请求打包进行一次操作，避免过多的无线信号引起的电量消耗。关于网络请求引起无线信号的电量消耗，还可以参考这里http://hukai.me/android-training-course-in-chinese/connectivity/efficient-downloads/efficient-network-access.html

我们可以通过手机设置选项找到对应App的电量消耗统计数据。我们还可以通过**Battery Historian Tool**来查看详细的电量消耗。

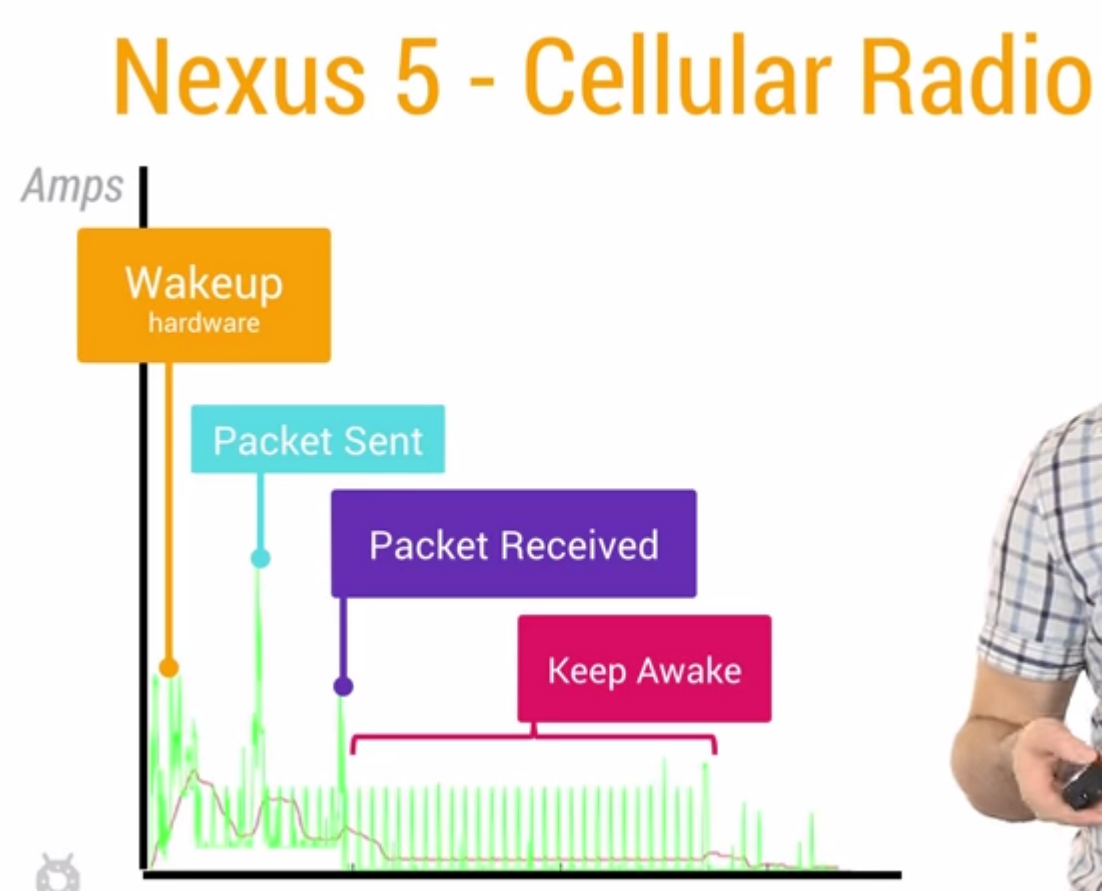
如果发现我们的App有电量消耗过多的问题，我们可以使用JobScheduler API来对一些任务进行定时处理，例如我们可以把那些任务重的操作等到手机处于充电状态，或者是连接到WiFi的时候来处理。 关于JobScheduler的更多知识可以参考http://hukai.me/android-training-course-in-chinese/background-jobs/scheduling/index.html

14)Understanding Battery Drain on Android

电量消耗的计算与统计是一件麻烦而且矛盾的事情，记录电量消耗本身也是一个费电量的事情。唯一可行的方案是使用第三方监测电量的设备，这样才能够获取到真实的电量消耗。

当设备处于待机状态时消耗的电量是极少的，以N5为例，打开飞行模式，可以待机接近1个月。可是点亮屏幕，硬件各个模块就需要开始工作，这会需要消耗很多电量。

使用WakeLock或者JobScheduler唤醒设备处理定时的任务之后，一定要及时让设备回到初始状态。每次唤醒无线信号进行数据传递，都会消耗很多电量，它比WiFi等操作更加的耗电，详情请关注http://hukai.me/android-training-course-in-chinese/connectivity/efficient-downloads/efficient-network-access.html



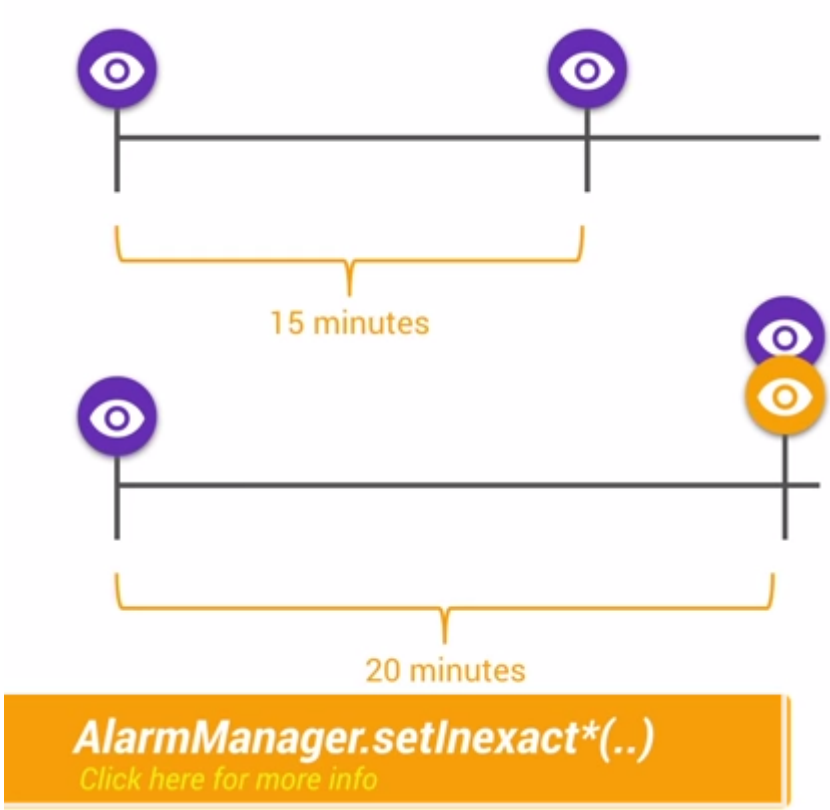
修复电量的消耗是另外一个很大的课题，这里就不展开继续了。

15)Battery Drain and WakeLocks

高效的保留更多的电量与不断促使用户使用你的App会消耗电量，这是矛盾的选择题。不过我们可以使用一些更好的办法来平衡两者。

假设你的手机里面装了大量的社交类应用，即使手机处于待机状态，也会经常被这些应用唤醒用来检查同步新的数据信息。Android会不断关闭各种硬件来延长手机的待机时间，首先屏幕会逐渐变暗直至关闭，然后CPU进入睡眠，这一切操作都是为了节约宝贵的电量资源。但是即使在这种睡眠状态下，大多数应用还是会尝试进行工作，他们将不断的唤醒手机。一个最简单的唤醒手机的方法是使用PowerManager.WakeLock的API来保持CPU工作并防止屏幕变暗关闭。这使得手机可以被唤醒，执行工作，然后回到睡眠状态。知道如何获取WakeLock是简单的，可是及时释放WakeLock也是非常重要的，不恰当的使用WakeLock会导致严重错误。例如网络请求的数据返回时间不确定，导致本来只需要10s的事情一直等待了1个小时，这样会使得电量白白浪费了。这也是为何使用带超时参数的wakelock.acquice()方法是很关键的。但是仅仅设置超时并不足够解决问题，例如设置多长的超时比较合适？什么时候进行重试等等？

解决上面的问题，正确的方式可能是使用非精准定时器。通常情况下，我们会设定一个时间进行某个操作，但是动态修改这个时间也许会更好。例如，如果有另外一个程序需要比你设定的时间晚5分钟唤醒，最好能够等到那个时候，两个任务捆绑一起同时进行，这就是非精确定时器的核心工作原理。我们可以定制计划的任务，可是系统如果检测到一个更好的时间，它可以推迟你的任务，以节省电量消耗。



这正是JobScheduler API所做的事情。它会根据当前的情况与任务，组合出理想的唤醒时间，例如等到正在充电或者连接到WiFi的时候，或者集中任务一起执行。我们可以通过这个API实现很多免费的调度算法。

从Android 5.0开始发布了Battery History Tool，它可以查看程序被唤醒的频率，又谁唤醒的，持续了多长的时间，这些信息都可以获取到。

请关注程序的电量消耗，用户可以通过手机的设置选项观察到那些耗电量大户，并可能决定卸载他们。所以尽量减少程序的电量消耗是非常有必要的。

Posted by HuKai Jan 17th, 2015 Android, Android:Performance

« Android APK安装包瘦身 Android性能优化之渲染篇 »

Comments