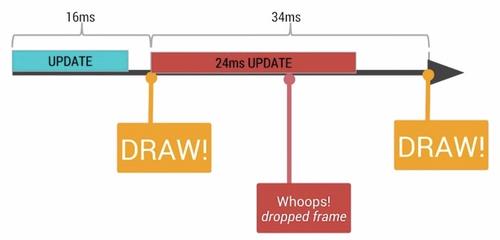
**卡顿现象**

渲染功能是应用程序最普遍的功能，开发任何应用程序都是这样，一方面，设计师要求为用户展现可用性最高的超然体验，另一方面，那些华丽的图片和动画，并不是在所有的设备上都能刘畅地运行。我们来了解一下什么是渲染性能。

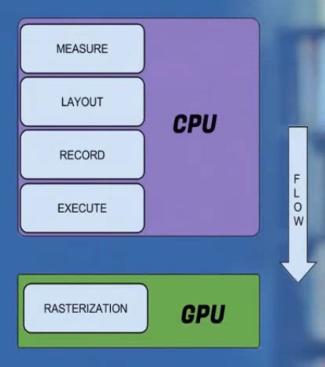
首先，我们要知道Android系统每隔16ms就重新绘制一次Activity，也就是说，我们的应用必须在16ms内完成屏幕刷新的全部逻辑操作，这样才能达到每秒60帧，然而这个每秒帧数的参数由手机硬件所决定，现在大多数手机屏幕刷新率是60赫兹（赫兹是国际单位制中频率的单位，它是每秒中的周期性变动重复次数的计量），也就是说我们有16ms（1000ms/60次=16.66ms）的时间去完成每帧的绘制逻辑操作，如果错过了，比如说我们花费34ms才完成计算，那么就会出现我们称之为丢帧的情况。



安卓系统尝试在屏幕上绘制新的一帧，但是这一帧还没准备好，所以画面就不会刷新。如果用户盯着同一张图看了32ms而不是16ms，用户会很容易察觉出卡顿感，哪怕仅仅出现一次掉帧，用户都会发现动画不是很顺畅，如果出现多次掉帧，用户就会开始抱怨卡顿，如果此时用户正在和系统进行交互操作，例如滑动列表或者输入数据，那么卡顿感就会更加明显，用户会毫不留情地对我们的应用进行吐槽，现在我们对绘制每帧花费的时间有了更清晰的了解，再来看看是什么原因导致了卡顿，如何去解决应用中的这些问题

**渲染管线**

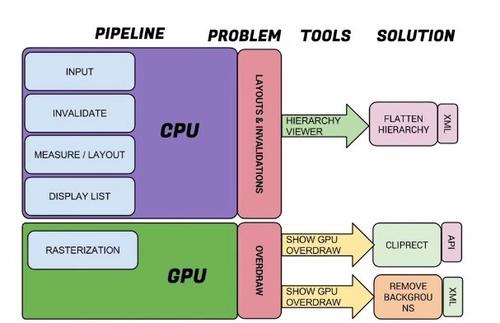
Android系统的渲染管线分为两个关键组件：CPU和GPU，它们共同工作，在屏幕上绘制图片，每个组件都有自身定义的特定流程。我们必须遵守这些特定的操作规则才能达到效果。



在CPU方面，最常见的性能问题是不必要的布局和失效，这些内容必须在视图层次结构中进行测量、清除并重新创建，引发这种问题通常有两个原因：一是重建显示列表的次数太多，二是花费太多时间作废视图层次并进行不必要的重绘，这两个原因在更新显示列表或者其他缓存GPU资源时导致CPU工作过度。

在GPU方面，最常见的问题是我们所说的过度绘制（overdraw），通常是在像素着色过程中，通过其他工具进行后期着色时浪费了GPU处理时间。

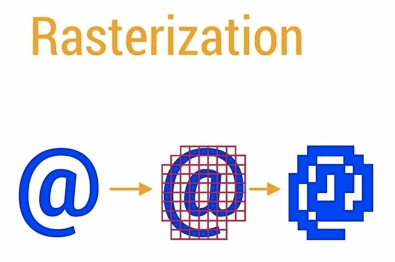
接下来我们将讲解更多关于失效布局和重绘的内容，以及如何使用SDK中的工具找出拖累应用性能的原因



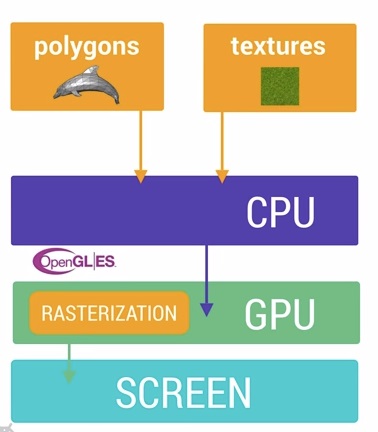
**CPU和 GPU**

想要开发一款性能优越的应用，我们必须了解底层是如何运行的。有一个主要问题就是，Activity是如何绘制到屏幕上的？那些复杂的XML布局文件和标记语言，是如何转化成用户能看懂的图像的？

实际上，这是由格栅化操作来完成的，格栅化就是将例如字符串、按钮、路径或者形状的一些高级对象，拆分到不同的像素上在屏幕上进行显示，格栅化是一个非常费时的操作。我们所有人的手机里面都有一块特殊硬件，它就是图像处理器（GPU显卡的处理器），目的就是加快格栅化的操作，GPU在上个世纪90年代被引入用来帮助加快格栅化操作。

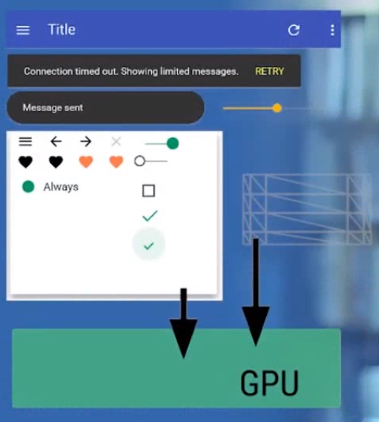


GPU使用一些指定的基础指令集，主要是多边形和纹理，也就是图片，CPU在屏幕上绘制图像前会向GPU输入这些指令，这一过程通常使用的API就是Android的OpenGL ES，这就是说，在屏幕上绘制UI对象时无论是按钮、路径或者复选框，都需要在CPU中首先转换为多边形或者纹理，然后再传递给GPU进行格栅化。



我们要知道，一个UI对象转换为一系列多边形和纹理的过程肯定相当耗时，从CPU上传处理数据到GPU同样也很耗时。所以很明显，我们需要尽量减少对象转换的次数，以及上传数据的次数，幸亏，OpenGL ES API允许数据上传到GPU后可以对数据进行保存，当我们下次绘制一个按钮时，只需要在GPU存储器里引用它，然后告诉OpenGL如何绘制就可以了，一条经验之谈：渲染性能的优化就是尽可能地上传数据到GPU，然后尽可能长地在不修改的情况下保存数据，因为每次上传资源到GPU时，我们都会浪费宝贵的处理时间，Android系统的Honeycomb版本发布之后，整个UI渲染系统就在GPU中运行，之后各个版本都在渲染系统性能方面有更多改进。

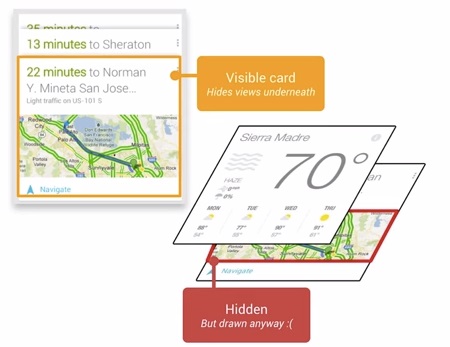
Android系统在降低、重新利用GPU资源方面做了很多工作，这方面完全不用担心，举例说，任何我们的主题所提供的资源，例如Bitmaps、Drawables等都是一起打包到统一的纹理当中，然后使用网格工具上传到GPU，例如Nine Patches等，这样每次我需要绘制这些资源时，我们就不用做任何转换，他们已经存储在GPU中了，大大加快了这些视图类型的显示。然而随着UI对象的不断升级，渲染流程也变得越来越复杂，例如说绘制图像，就是把图片上传到CPU存储器，然后传递到GPU中进行渲染。路径使用时完全另外一码事，我们需要在CPU中创建一系列的多边形，甚至在GPU中创建掩蔽纹理来定义路径。绘制字符就更加复杂一些，首先我们需要在CPU中把字符绘制制成图像，然后把图像上传到GPU进行渲染再返回到CPU，在屏幕上为字符串的每个字符绘制一个正方形。



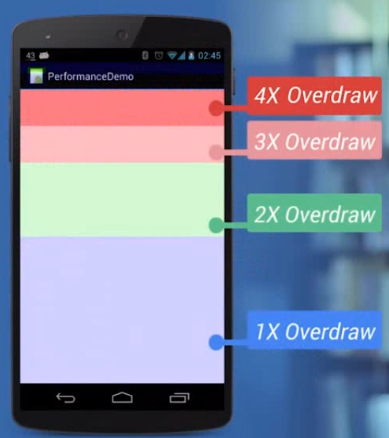
现在Android系统已经解决了大多数性能问题，除非我们还有更高要求，我们基本不会发现与GPU相关的问题，然后还有一个GPU性能问题瓶颈，这个问题困扰着每个程序员，这就是过度绘制。

**GPU的主要问题 -过度绘制（overdraw）**

如果我们曾经粉刷过房子，我们应该知道，给墙壁粉刷工作量非常大，如果我们需要重新粉刷，第一次的粉刷就白干了。同样的道理，我们的应用程序会因为过度绘制，从而导致性能问题，如果我们想兼顾高性能和完美的设计，往往会碰到一种性能问题，即过度绘制。过度绘制是一个术语，指的是屏幕上的某个像素点在同一帧的时间内被绘制了多次。假如我们有一堆重叠的UI卡片，最接近用户的卡片在最上面，其余卡片都藏在下面，也就是说我们花大力气绘制的那些下面的卡片基本都是不可见的。



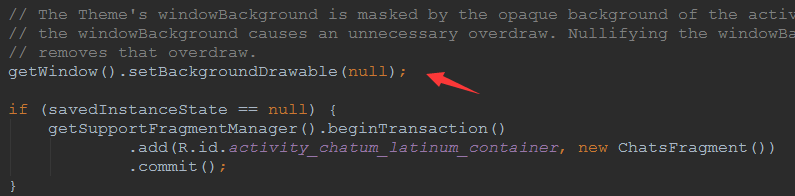
问题就在于此，因为每次像素经过渲染后，并不是用户最后看到的部分，这就是在浪费GPU的时间。目前流行的一些布局是一把双刃剑，带给我们漂亮视觉感受的同时，也造成过度绘制的问题，为了最大限度地提高应用程序的性能，我们必须尽量减少过度绘制。幸运的是，Android手机提供了查看过度绘制情况的工具，在开发者选项中打开“Show GPU overdraw”选项，手机屏幕显示会出现一些异常不用过于惊慌，Android在屏幕上使用不同颜色，标记过度绘制的区域，如果某个像素点只渲染了一次，我们看到的是它原来的颜色，随着过度绘制的增多，标记颜色也会逐渐加深，例如1倍过度绘制会被标记为蓝色，2倍、3倍、4倍过度绘制遵循同样的模式。所以当我们调试应用程序的用户界面时，目标就是尽可能的减少过度绘制，将红色区块转变成蓝色区块，为了完成目标有两种清除过度绘制的方法，首先要从视图中清除那些不必要的背景和图片，他们不会在最终渲染图像中显示，记住，这些都会影响性能。其次，对视图中重叠的屏幕区域进行定义，从而降低CPU和GPU的消耗，接下来我们深入了解过度绘制



**可视化方式解决过度绘制**

现在我们看到了示例代码的应用程序，现在就想象我们自己开发了一款聊天应用，我们想了解应用程序在过度绘制性能上的表现如何，首先要做的就是搜集信息，在这一步我们需要打开手机上的GPU过度绘制调试，看看这些过度绘制的地方，我们需要减少这些过度绘制，尤其是红色区域，这里说明一下各个颜色代表的意思。

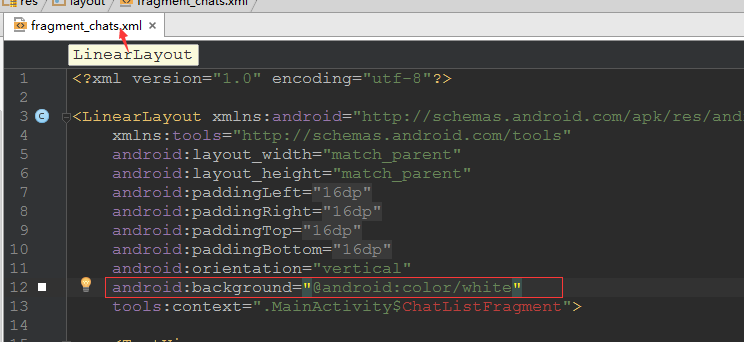
现在深入了解一下UI是如何创建的，看看能否做一些清理，减少过度绘制，办法一就是清除不必要的背景和图片。例如我们想把Chatum背景中的这块区域变成绿色或者2倍过度绘制区域，为什么能实现这个效果呢？这是由于Chatum的BaseActivity采用了不透明白色背景的布局填充整个屏幕，我们喜欢这样，但是却与Android的材料主题默认设置相冲突，特别是窗口背景图片，这些都导致了不必要的过度绘制，作为一个开发者我们必须做一个决定，我们希望保留白色背景，材料主题其实没有任何意义，我们能做的一个优化就是把Activity的背景图片设置为null，我向我们们展示一下在代码中如何实现，打开Chatum的BaseActivity我们看一下onCreate方法，使用下列声明取消原来的背景，就是这样，通过取消背景我们将过度绘制区域的颜色，由绿色变成了蓝色，变成了1倍过度绘制。



优化后

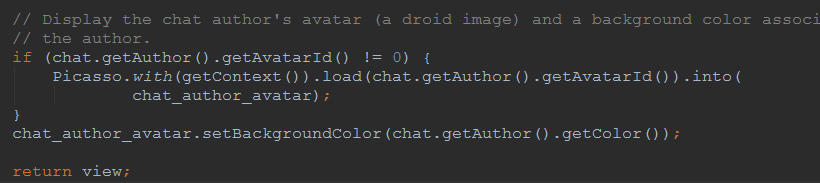
我们看一下XML标记文件看看能否再做一些调整，我们可能已经注意到有三个XML文件，指定了Chatum的用户界面，有Chatum Latinum的BaseActivity，XML聊天片段还有聊天记录的单个XML，前面已经提过，我们想在这里保留白色背景，在这里我们什么都不做，但是其他两个XML文件能否做一些调整，现在我需要我们的帮助，请我们帮我梳理一下其他XML文件，看看能否清除一些不必要的背景，完成之后，把清除的背景数目填到这个方框里，如果碰到什么麻烦也用不着担心。

在剩下的文件中应该可以找到4个不必要的背景，我们来查看一下，在BaseActivity中记住我们希望保留白色背景，现在我们来看聊天片段XML文件我们在这里声明了一个不必要的白色背景，我们并不需要这个声明因为可以使用MainActivity的白色背景（fragment\_chats.xml）。

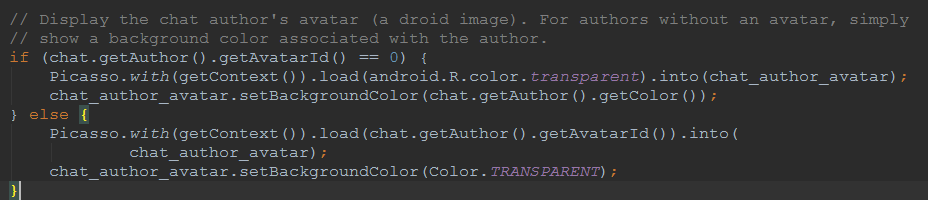


现在来看聊天记录单个XML文件（chat\_item.xml），这里有三个不必要的背景，我们来删除它们。

好了，现在我们来看看，过度绘制的情况有没有改善，我们的屏幕就应该现在这样，恰当的删除这些背景，干净多了，对不对？很快就快大功告成了，但是我们还可以再做一个优化，注意头像区域仍存在过度绘制，因为我们绘制了一个方框然后再绘制头像图片，在没有获取到头像时我们才设置一个背景，我们可以使用一些条件码来实现，我们打开ChatAdapter.java，这部分代码负责在个人聊天记录上传后进行填写，我们找到了getview方法，在底部这里我们找到一个逻辑操作，用来显示头像的同时设置背景颜色，我们来看看能否变得更智能一些，我们来写一段代码，在未获取到头像时仅用来设置背景颜色，然后我们将把背景颜色设置为透明，然后上传头像，好了，这就是我们更新后的代码。



修改前

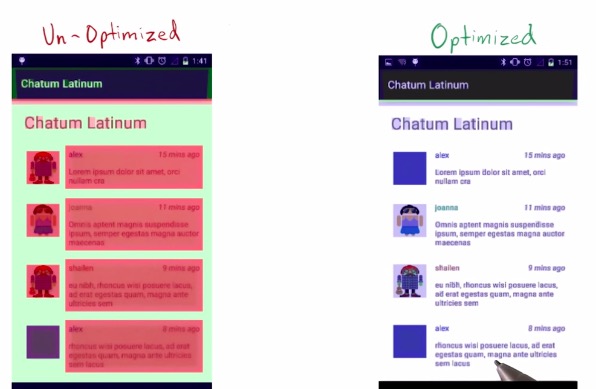


修改后

注意，当为获取到头像时我们要做的是，在头像通常的位置加载透明色，然后，为头像设置真的背景色，剩下的就是获取到头像时的操作，我们恰当的加载头像，然后，我们将背景色设置为透明，这样我们就能将过度绘制最小化。好了，我们来看看情况有没有改善。很好，我们可以看到头像区域，在更新代码后过度绘制减少了，好了，这就是我们最后的优化。

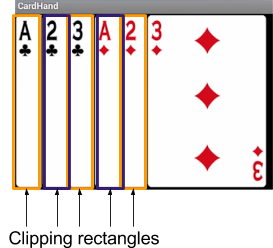
我们总结下，优化前过度绘制非常严重，首先要做的，就是将背景图片设置为null，其次，就是清除XML文件中，不必要的背景声明，最后，我们只在为获取到头像时，显示背景颜色。经过这些优化，我们再来看看，过度绘制的情况相比开始有了很大改善。

注意：有些过度绘制对于运行性能，可能是必要的也是可以接受的，比如说Android的ActionBar，但是，如果我们希望应用体验更进一步，我们可以考虑尽可能地减少过度绘制。



**clipRect和quickReject**

值得指出的是，Android系统知道过度绘制是个麻烦，Android会设法避免绘制，那些在最终图片中不显示的UI组件，这种优化类型，称作剪辑，它对UI性能非常重要。如果我们能确定某个对象会被完全阻挡，那就完全没有必要绘制它，事实上，这是最重要的性能优化方法之一，而且是有Android系统执行的，但是不幸的是，这一技术无法应对复杂的自定义的View，系统无法检测onDraw具体会执行什么操作。这些情况下，底层系统无法识别如何去绘制对象，系统很难将覆盖的View，从渲染管道中清除。例如，这叠牌只有最上面的牌是完全可见的，其他牌都被挡住了，这就意味着绘制那些重叠的像素就是浪费时间。



为了解决这个问题，我们可以使用Canvas类的一些特别方法去帮助Android系统识别被遮挡的不需要绘制的部分，最有用的办法是Canvas.clipRect，它可以帮助我们识别给定View的图片边界，边界之外区域的任何绘制操作会被忽视，如果碰到此类重叠的View，这个方法特别好用，就像例子中的纸牌。如果我们知道自定义View可见部分的范围，或者知道遮挡部分的范围，我们就可以定义ClipRect边界，可以避免遮挡区域的任何绘制操作，ClipRect API帮助系统识别出无需绘制的区域，对自定义View进行剪辑时，这个方法也很有用处。比如说，如果我们知道绘制对象在剪辑矩形之外，这个方法就非常好用，幸运的是，我们不必亲自搞清楚重叠逻辑，我们可以使用Canvas.quickReject方法，判定给定区域是否完全在剪辑矩形之外，这种情况下可以忽略全部绘制工作。现在我们来看一个相关案例，我们对它做一些改进。

**布局优化**

是时候来了解一下渲染管道中的CPU部分，为了在屏幕上绘制某个东西，Android通常将高级XML文件转换为GPU能够识别的对象，然后显示在屏幕上，这个操作是在DisplayList的帮助下完成的，DisplayList持有所有要交给GPU绘制到屏幕上的数据信息，包含GPU要绘制的全部对象的信息列表，还有执行绘制操作的OpenGL命令列表，在某个View第一次需要被渲染时，DisplayList会因此被创建，当这个View要显示到屏幕上时，我们将绘制指令提交给GPU来执行DisplayList，我们下次渲染这个View时，比如说位置发生了变化，我们仅仅需要执行DisplayList就够了，但是如果我们修改了View的某些可见组件的内容，那么之前的DisplayList就无法继续使用了，这时我们要重新创建一个DisplayList，重新执行渲染指令并更新到屏幕上，请注意，任何时候View的绘制内容发生变化，都需要重新创建DisplayList并重新执行指令更新到屏幕，这个流程的表现性能，取决于我们的View的复杂程度，取决于视觉变化的类型，同时对渲染管道也会产生一些影响。举例说，假如某个文本框尺寸突然变成当前的两倍，在改变尺寸前，需要通过父View重新计算，并摆放其他子View的位置，在这种情况下我们改变了某个View，后面就会有很多工作要做，这些类型的视觉变化需要渲染管道的额外工作，当我们的View的尺寸变化时，触发了测量操作，会经过整个View Hierarchy，询问各个View的新尺寸，我们一旦改变了View的大小就会触发上述过程，无论是填充或者图片尺寸、设置文本大小、宽度、高度等等，如果我们是改变对象位置或者询问布局，或者某个View重新摆放子View都会触发布局操作，会触发整个Hierarchy重新计算对象在屏幕上的新位置，现在Android运行系统已经非常善于处理记录并执行渲染管道，除非我们要处理自定义View或者同时需要绘制太多View，其他情况下一般不会耗费太多时间，测量和布局操作性能也很好，但是当我们的View Hierarchy失控时也更容易出现问题，执行这些功能的时间是和我们的View Hierarchy中需要处理的节点数成正比的，系统需要处理的View越多处理时间就越长。某些View可能比其他View要耗费更多时间，造成这种浪费的首要原因是，View Hierarchy中包含太多的无用View，这些View根本不会显示在屏幕上，一旦触发测量操作和布局操作只会拖累应用程序的性能表现，幸好有一款叫做Hierarchy Viewer工具，它可以帮助我们查找并修复这些流氓View，我们来看看。

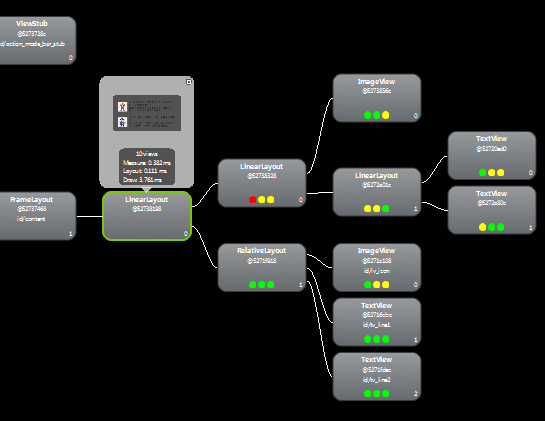
**Hierarchy Viewer工具**

让我们来看第一个工具，Hierarchy Viewer将帮助我们快速可视化整个UI结构，另外，它还提供一个更好的方法，让我们理解这个结构内的独特视图的相对渲染性能。它看起来是这样的，让我们来进行设置。我们需要设置一些代码来使用这个工具，我们将使用Android基础类中的Sunshine，在下面的教师笔记中，我们可以找到详细的信息介绍如何抓取这些代码，完成之后，请点击Continue继续。对了，还有一件事情，如果我们的手机已经root，而且运行Android Jellybean或更新版本的系统，将不会有问题。我需要在计算机上设置一个环境变量，使Hierarchy Viewer能够与我们的设备通信。我们也可以在教师笔记中找到这方面的详细信息，好了，让我们继续。我们启动Hierarchy Viewer，我们首先要做的事情是进入Android Studio，然后，为了使用Hierarchy Viewer，我们需要启动Android Device Monitor。

在这个例子中，这是sunshine应用程序的主活动，现在如果我们把鼠标悬停在这里，这是树状概览，将概览显示整个UI结构。现在，我们还可以看到这个视图端口，允许我们移动和更改左侧详情视图中显示的内容。这是详情视图，我们可以看到，当我移动视图端口时，详情视图中的详细信息也发生变化，这可以放大层级结构中的部分内容。现在如果我们跳转到树状视图，还会看到另外一个缩放控制功能，允许放大和缩小我们的结构，像这样。我还可以拖动布局以移动位置，放大和缩小以检视更多的内容。当我们点击一个视图时，可以在属性窗格中，看到它的所有属性。这个属性窗格在左侧的这个位置打开，现在我要展开它。这里有一些可缩回式菜单，显示视图的各种详细信息。例如，我可以查看事件，和屏幕绘图相关的其他详细信息，比如透明度、缩放、旋转等。现在，双击一个视图，可以预览我们所选择的这个视图在屏幕的实际显示效果，然后，我们移到右下角的这个位置，这个Layout视图，当我点击层级上的一个视图时，Layout视图突出显示Layout视图中的相同视图。换言之，我们可以将这个Layout视图看做是一个框架，用来标记我们的设备屏幕，因此，当我们可以选择详情视图中的项目时，并且看到他们将会布局我们的设备屏幕。我们还可以进入反方向，点击这里的项目。可以看到，在树状视图的顶部，这些项目变亮，它对应于这里的这个视图。如果我移动视图页面，并且放大一点，我可以看到这个视图的详情，不幸的是，我们不能从Hierarchy Viewer转到我们的源代码，因为它链接到正在运行的程序，而不是源代码本身。但是，每个节点显示类型和ID，这样我们可以在以后参考它们，或者以后在我们的源代码中找到他们。我们应该知道，Hierarchy Viewer很有用，首先，它帮助我们从Android的角度，理解我们的用户界面的结构，另外，这可以帮助我们确定哪些视图是多余的，让我们可以简化我们的视图层级以节省内存，并提高渲染性能。但是真实的性能信息来自于，每个节点的概况显示功能和Hierarchy Viewer。这个工具建模显示整个用户界面的渲染过程，并提供一个特定节点，相对特定树状图中的其他节点的渲染数据，让我们回到详情树状视图，我点击一个子数级中的一个根节点，我们想要显示它的结构情况，从测试的角度出发，我将点击这个操作工具条容器。现在为了调用Hierarchy Viewer的概要显示功能，我们需要点击这里的这个Profile Node图标，我们可能注意到每个视图有三个独特的圆点，它们可能具有不同的颜色，绿色、黄色或红色。但是，这些圆点的顺序也具有特定的含义。最左边的圆点表示渲染管道的测量阶段，中间的圆点表示布局阶段，最右边的圆点表示渲染管道的绘制阶段。现在，我来介绍这些颜色的含义，这些圆点的颜色表示这个节点相对于所有其他已经概要显示的节点的性能。那么，相对性能是什么意思呢？让我们来看这个绿色表示这个管道阶段，这个视图的渲染速度快于至少一半以上的其他视图。这个黄色表示它的渲染速度属于比较慢的50%，如果我们看到红色，意味着这是视图层级中最慢的节点，我们还应该知道红色节点可能是存在问题。我们可能不希望它的渲染速度如此之慢。例如，在一个叶片节点或仅有少数几个子元素的视图中，应该是一个红旗。另外还应该记住，当处理大型层级时，还有些节点应该是速度最慢的节点，那么我们应该问，这个节点是我们所期望的节点吗？另外请记住，由于它的性能相对较慢，但是它的绝对性能可能并不慢，因此使用实际数字将会起到帮助作用。现在，我们已经熟悉Hierarchy Viewer，让我们继续使用本课的一些示例代码进行练习。

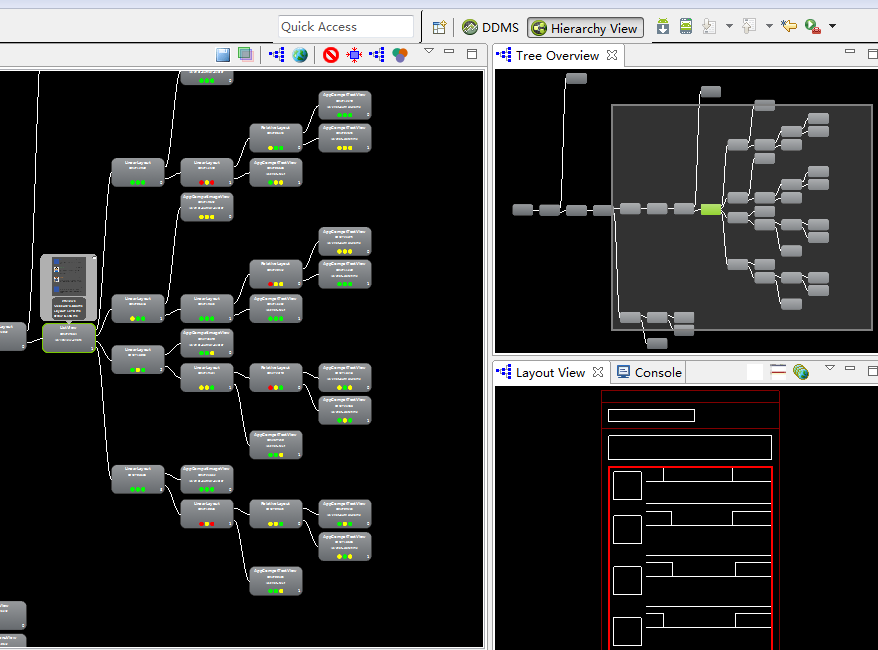
**嵌套结构的性能评测**

我们可能已经知道，在我们创建Android用户界面时，应该让我们的布局尽可能简单和扁平化，我有一些很好的建议，请记住庞大的布局十分浪费资源，每个附加嵌套布局和内置视图都会直接影响我们的应用程序的性能和响应灵敏性，因此请记住，我们应该了解我们的应用程序的行为模式，现在我们要返回到Android Device Monitor，我已经打开Hierarchy Viewer视窗，和以前一样我们跳转到这里的窗格，选择我们的设备然后选择想要查看的活动，在这个例子中我们要看一下这个根节点，这是我们的线性布局，这是跟视图群组将会显示这两行，我们在这里可以看到它们，请注意来自于这个父级线性布局的两个不同子元素，其中一个表示我们聊天界面第一行，但是它使用嵌套线性布局实现，第二个对应于布局中的第二行，这次不使用嵌套设计，我们使用扁平化设计，使用相对布局视图群组，这对应于XML中的代码，进入Android Studio查看我们的源代码，打开activity\_compare\_layout.xml文件,现在重新在屏幕上显示以便于进行比较，我们看到一个父级容器它是一个由垂直方向组成的线性布局，因此这些控件将会从上到下排列，现在我想让我们注意这里，这是我们的第一个聊天行，我们的实现方法不是使用结构化或嵌套布局，这种方法更加直观，逻辑性很强，例如，我们从一个水平性质的父级线性布局开始，在左侧我们将设置一个ImageView，在右侧创建另外一个嵌套线性布局以容纳我们的文本，但是在这个例子中，方向是垂直的而不是水平的，它代表第一个条目，接下来我们可以看到聊天模板的第二行，与使用嵌套结构不同我们决定采用扁平化布局，使用相对位置来描述它们。这样做对于性能有什么影响？让我们返回Hierarchy Viewer

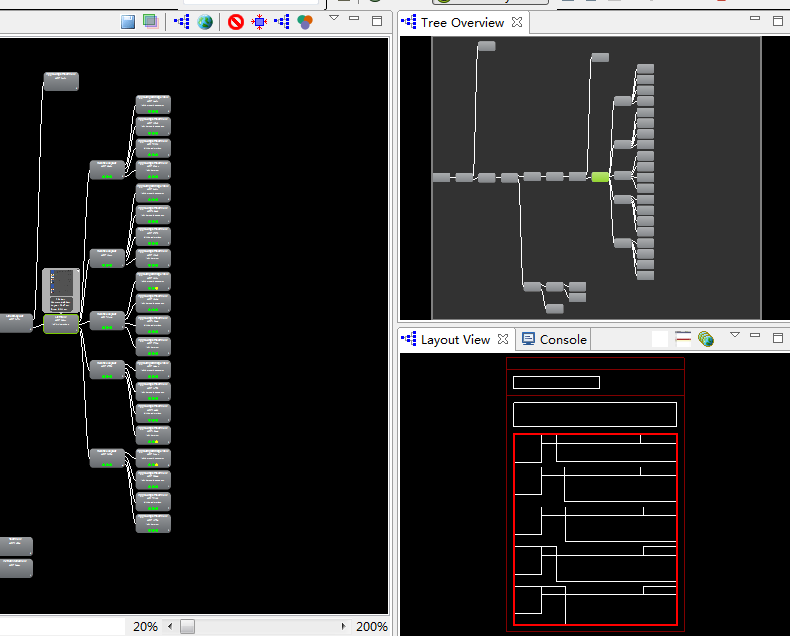


从渲染过程的角度来说线性布局设计比相对布局更慢一些，与相对布局比较，线性布局需要更多的资源开销，这里全部是绿色。如果我们有机会采取扁平化布局，我们应该想办法尽可能使用它。

**优化Chatum Latinum**



优化前



优化后