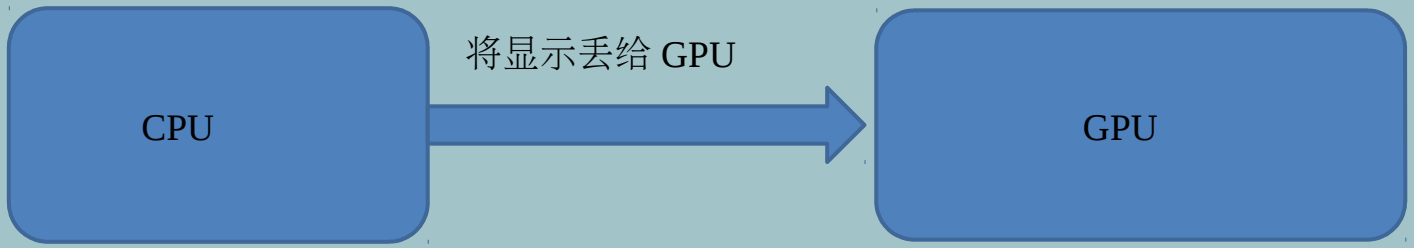
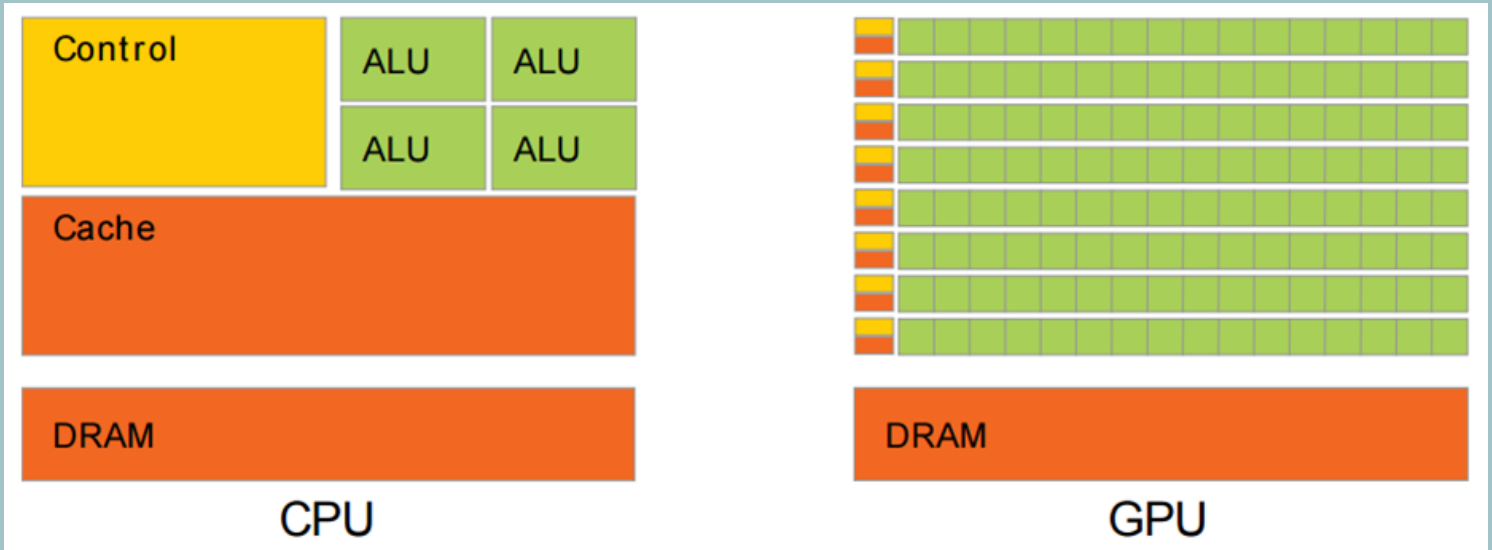
布局渲染流程与原理

任务

1. CPU 与 GPU 工作流程
2. 卡顿原因分析
3. 实战解决过度优化
3. Hierarchy Viewr 工具使用

CPU 的任务繁多，做逻辑计算外，还要做内存管理、显示操作，因此在实际运算的时候性能会大打折扣，在没有 GPU 的时代，不能显示复杂的图形，其运算速度远跟不上今天复杂三维游戏的要求。即使 CPU 的工作频率超过 2GHz 或更高，对它绘制图形提高也不大。这时 GPU 的设计就出来了





黄色的 Control 为控制器，用于协调控制整个 CPU 的运行，包括取出指令、控制其他模块的运行等；

绿色的 ALU （ Arithmetic Logic Unit ）是算术逻辑单元，用于进行数学、逻辑运算；
橙色的 Cache 和 DRAM 分别为缓存和 RAM ，用于存储信息。

从结构图可以看出， CPU 的控制器较为复杂，而 ALU 数量较少。因此 CPU 擅长各种复杂的逻辑运算，但不擅长数学尤其是浮点运算。

<Button>
layout_width="100dp"
layout_height="100dp"
</Button>

LayoutInflater
加载到内存

Button 对象

对象内 含有 left top right,bottom,width height 信息

cpu 经过计算

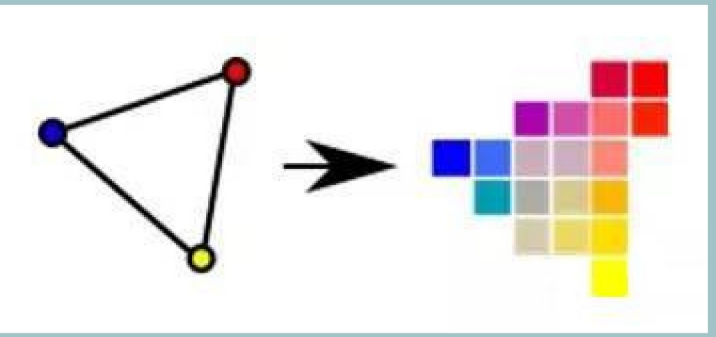
处理成多维的向量图形

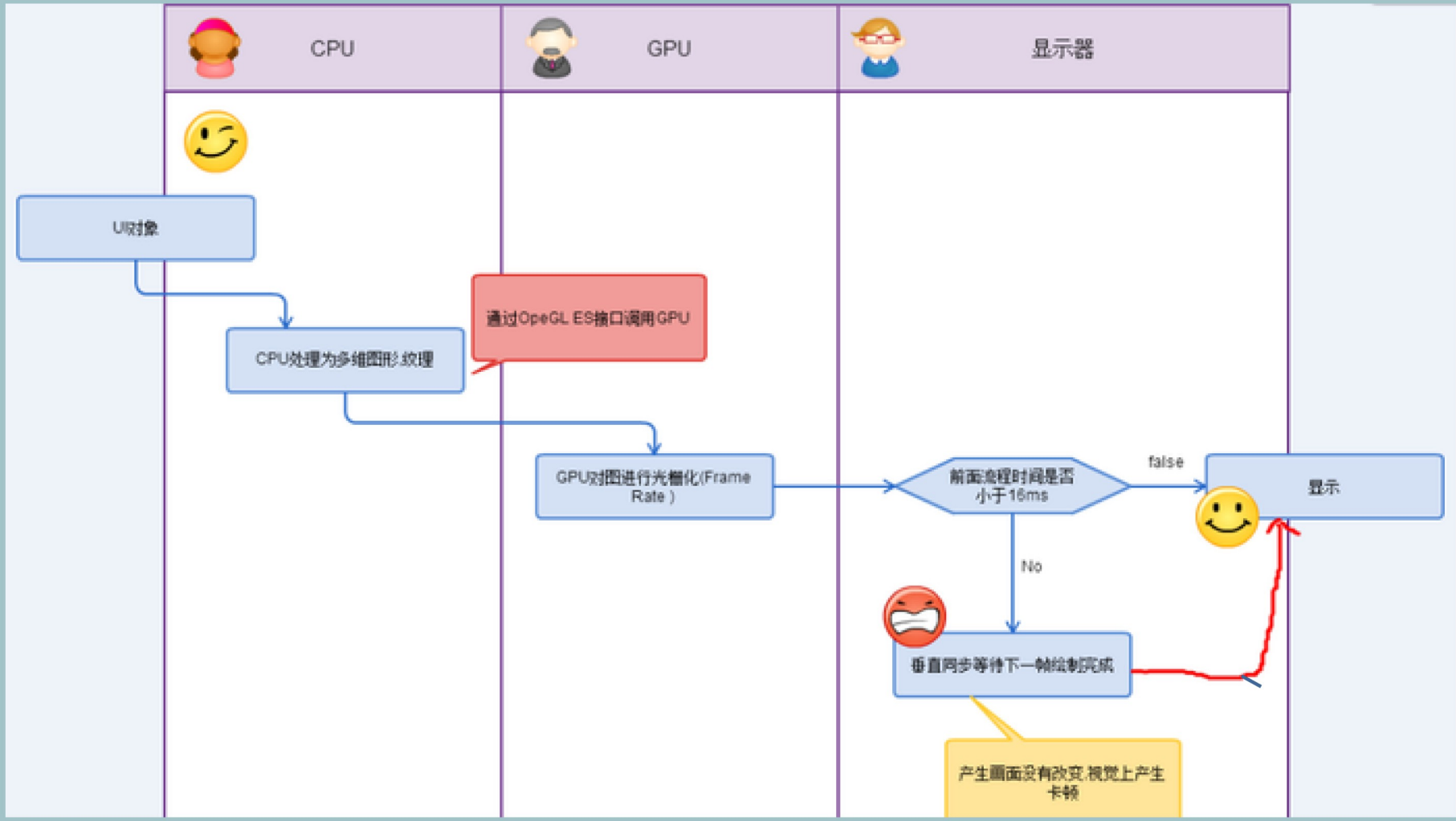
将适量图形交给 GPU
GPU 负责像素填充

栅格化

GPU 绘制图形

定义：栅格化是将向量图形格式表示的图像转换成位图以用于显示器




CPU
GPU
显示器
UI对象
CPU处理为多维图形.纹理
通过OpeGL ES接口调用GPU
GPU对图进行光栅化(Frame Rate)
前面流程时间是否小于16ms
false
显示
No
垂直同步等待下一帧绘制完成
产生画面没有改变,视觉上产生卡顿

60Hz 刷新频率由来

12 fps ：由于人类眼睛的特殊生理结构，如果所看画面之帧率高于每秒约 10-12 帧的时候，就会认为是连贯的

24 fps ：有声电影的拍摄及播放帧率均为每秒 24 帧，对一般人而言已算可接受

30 fps ：早期的高动态电子游戏，帧率少于每秒 30 帧的话就会显得不连贯，这是因为没有动态模糊使流畅度降低

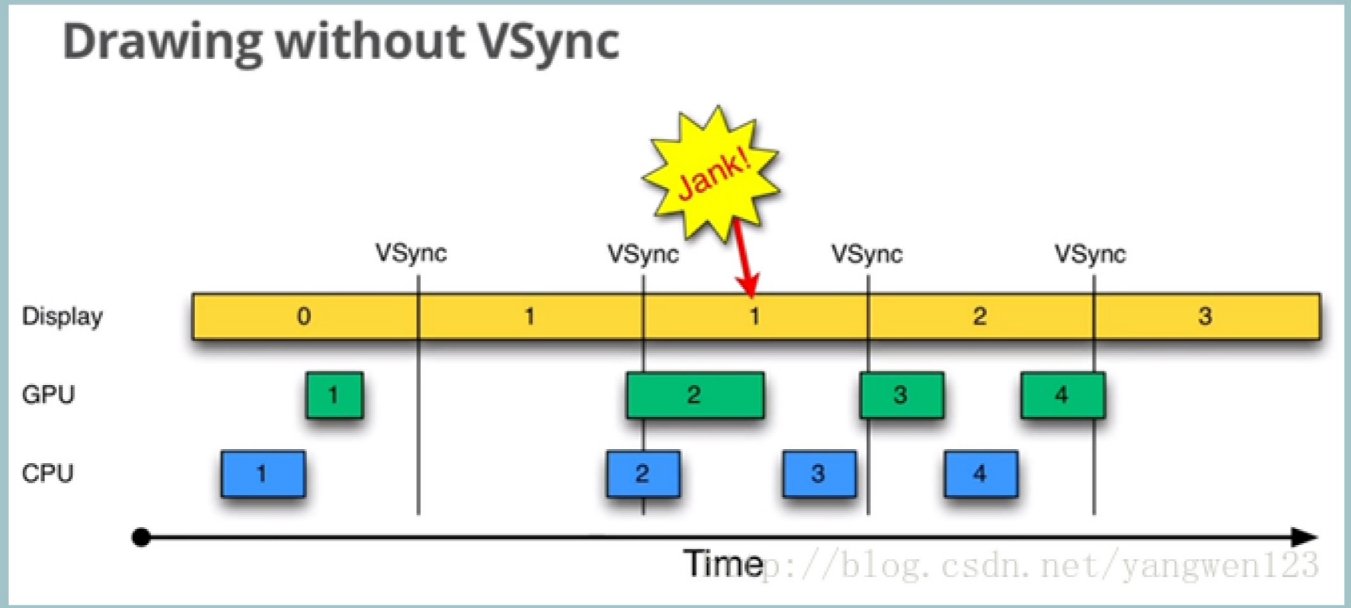
60 fps　在与手机交互过程中，如触摸和反馈　60 帧以下人是能感觉出来的。 60 帧以上不能察觉变化

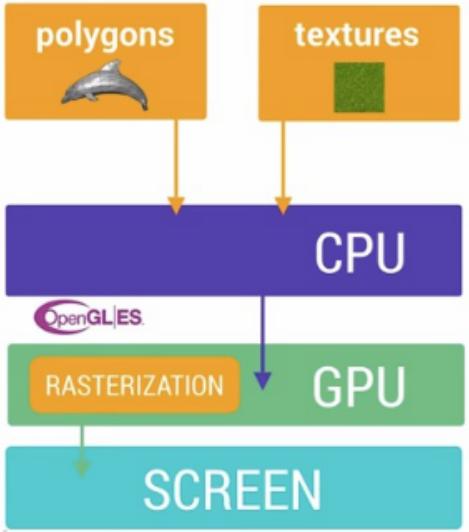
当帧率低于 **60 fps** 时感觉的画面的卡顿和迟滞现象

Android 系统每隔 16ms 发出 VSYNC 信号 (1000ms/60=16.66ms) ，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 ，这意味着计算渲染的大多数操作都必须在 16ms 内完成。

卡顿原理分析

当这一帧画面渲染时间超过 16ms 的时候，垂直同步机制会让显示器硬件 等待 GPU 完成栅格化渲染操作，
这样会让这一帧画面，多停留了 16ms, 甚至更多．这样就这造成了 用户看起来 画面停顿．

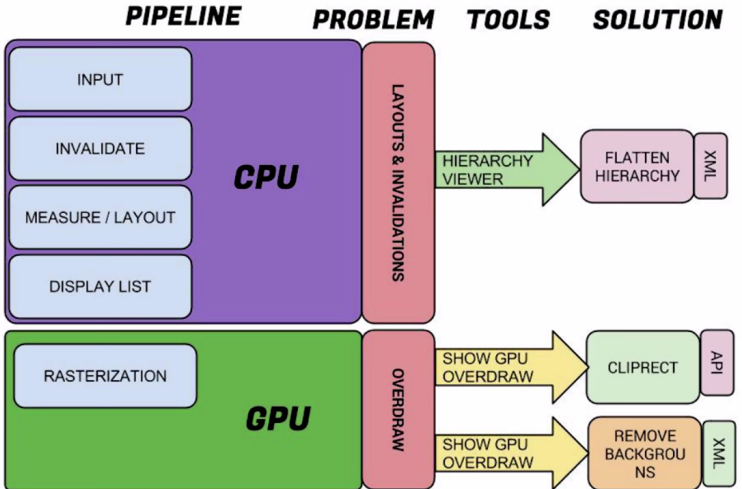




16 毫秒的时间主要被两件事情所占用

第一件：将 UI 对象转换为一系列多边形和纹理（ 1 ）（？）

第二件： CPU 传递处理数据到 GPU 。所以很明显，我们要缩短这两部分的时间，也就是说需要尽量减少对象转换的次数，以及上传数据的次数 （？ 布局　　自定义）



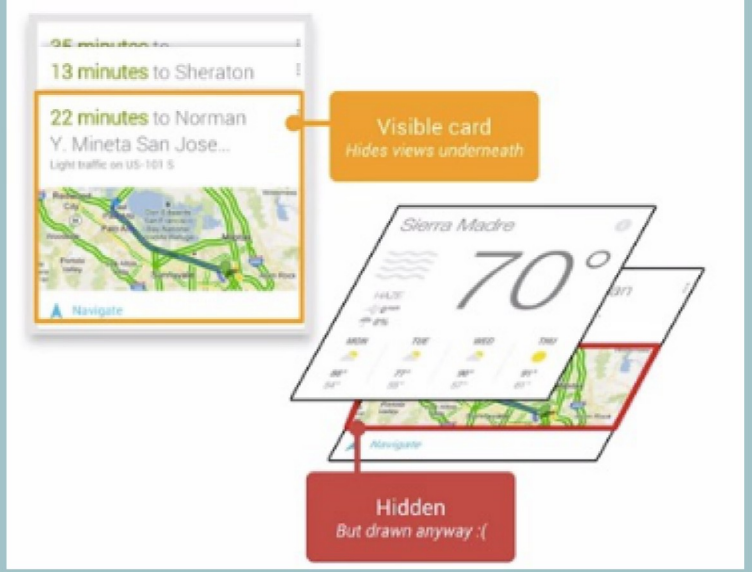
如何减少这两部分的时间 以至于在 **16ms** 完成呢

CPU 减少 xml 转换成对象的时间

GPU 减少重复绘制的时间

什么是过度绘制

GPU 的绘制过程，就跟刷墙一样，一层层的进行，16ms 刷一次。这样就会造成，图层覆盖的现象，即无用的图层还被绘制在底层，造成不必要的浪费。



GPU 过度绘制 几种情况

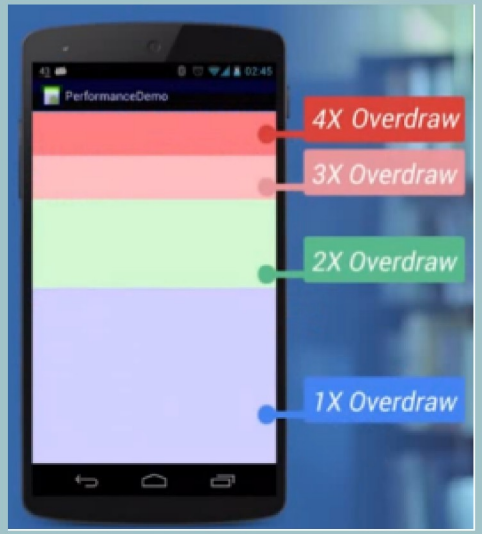
1 自定义控件中 onDraw 方法做了过多重复绘制

2 布局层次太深，重叠性太强。用户看不到的区域 GPU 也会渲染，导致耗时增加

过度绘制查看工具

在手机端的开发者选项里，有 OverDraw 监测工具，调试 GPU 过度绘制工具，

其中颜色代表渲染的图层情况，分别代表 1 层，2 层，3 层，4 层

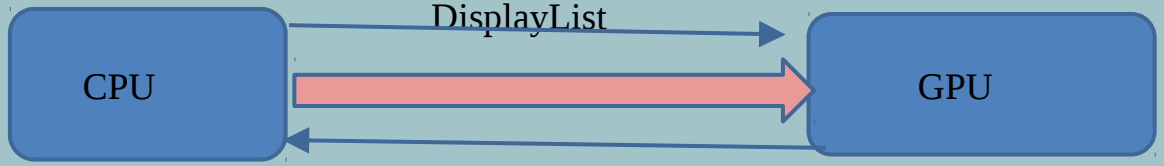


蓝色　过度绘制一次 无过度绘制
淡绿　过度绘制两次
淡红　过度绘制三次
深红　过度绘制四次
代表了 4 种不同程度 Overdraw 情况
我们的目标就是尽量减少红色 Overdraw ，看到更多的蓝色区域。

那 **Android** 系统有没有给我们做优化的操作呢

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。文字的显示比较复杂，需要先经过 CPU 换算成纹理，然后交给 GPU 进行渲染，返回到 CPU 绘制单个字符的时候，再重新引用经过 GPU 渲染的内容。动画则存在一个更加复杂的操作流程。
为了能够使得 App 流畅，我们需要在每帧 16ms 以内处理完所有的 CPU 与 GPU 的计算，绘制，渲染等等操作。



位置

颜色

缓存

invalidate()　　onDraw()

toolbar
include.xml

<include>

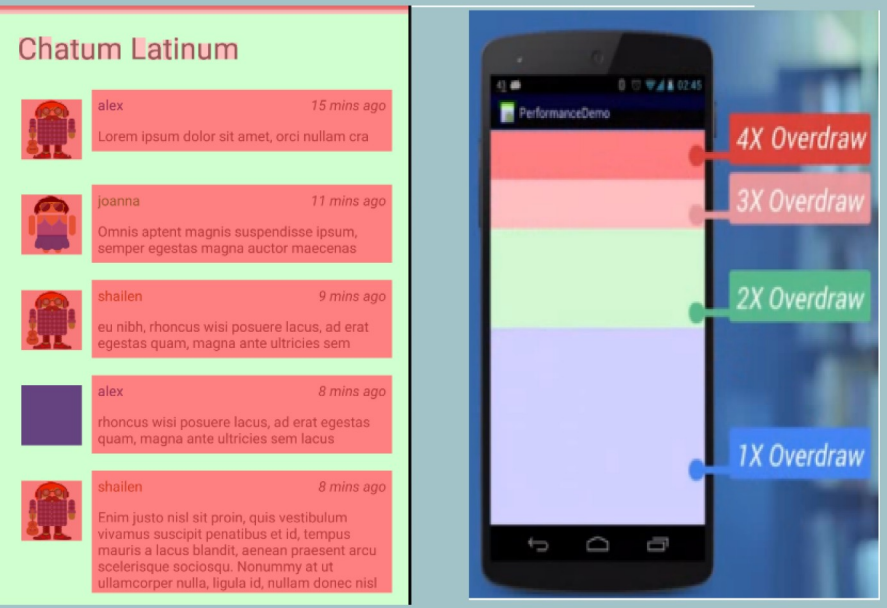
布局优化一　去掉默认 bei'jing

```
<!-- Base application theme. -->
<style name="Theme.Base" parent="Theme.AppCompat.Light">
    <!-- Customize your theme here. -->
    <item name="android:windowBackground">@null</item>
</style>
```

去掉二层容器背景

```
android:layout_width="match_parent"
android:layout_height="match_parent"
android:background="@android:color/white"
tools:context=".MainActivity"
```

过度绘制实例



如何避免过度绘制

1

布局优化原则

1. 减少不必要嵌套
2. 使用 merger 避免与父容器重叠

总结

性能优化其实不仅仅是一种技术，而是一种思想，你只听过它的高大上，却不知道它其实就是各个细节处的深入研究和处理。就像挤牙膏一样

1 布局里面是否有背景

2 是否可以删除多余的布局（ 容器 只有一个子节点 容器） （ srcoView--》 lineralayout ）

3 自定义 View 是否进行了裁剪 （ include ）

4 布局是否扁平化

感 谢 观 看