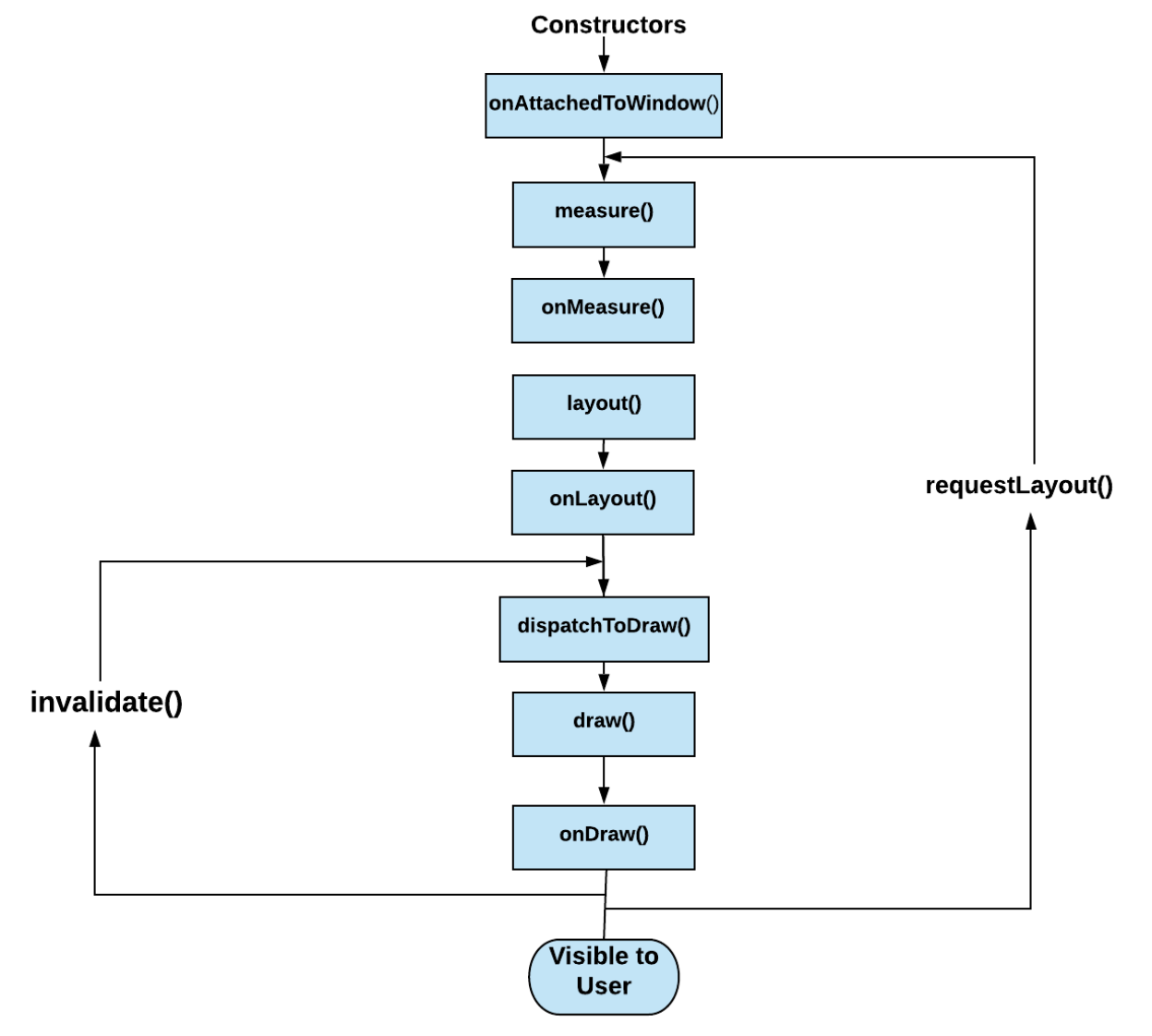
零、用于渲染视图的 Android 内部结构

1. 视图的生命周期

View 类代表用户界面组件的基本构建块。 View 在屏幕上占据一个矩形区域，负责绘图和事件处理。View 是小部件的基类，用于创建交互式 UI 组件（按钮、文本字段等）。 ViewGroup 子类是布局的基类，布局是包含其他View（或其他 ViewGroup）并定义其布局属性的不可见容器。





1.构造函数

View(Context context, @Nullable AttributeSet attrs, int defStyleAttr, int defStyleRes)

1. 附加/分离

这是从 Window附加或分离视图的阶段。在这个阶段，有一些方法可以接收回调来做适当的事情。onAttachedToWindow()：当视图附加到窗口时调用。这是视图知道它可以处于活动状态并具有用于绘制的表面的阶段。所以我们可以开始分配任何资源或设置监听器。onDetachedFromWindow()：当视图与窗口分离时调用。此时，它不再具有用于绘制的表面。需要停止执行任何类型的计划工作或清理分配的资源的地方。当我们在 ViewGroup 上调用 remove view 或 Activity 被销毁等时调用此方法。onFinishInflate()：这个方法会在所有的孩子都被添加之后被调用。

1. 遍历

这被称为遍历阶段，因为视图层次结构就像从父节点（ViewGroup）到叶节点（子视图）的分支的树结构。所以每个方法都从父节点开始，遍历到最后一个节点来定义约束。

1. onMeasure() 是一种调用方法来确定此视图及其所有子视图的大小要求。 onMeasure() 不返回值，而是调用 setMeasuredDimension() 显式设置宽度和高度。

MeasureSpec 封装了从父级传递到子级的布局要求。每个 MeasureSpec 代表对宽度或高度的要求。 MeasureSpec 由大小和模式组成。有三种可能的模式：

MeasureSpec.EXACTLY ：父母已经确定了孩子的确切尺寸。无论孩子想要多大，都将获得这些界限。在 LinearLayout 或 match\_parent 属性等中为视图指定固定宽度或权重的实例。

MeasureSpec.AT\_MOST：孩子可以任意大到指定的大小。

MeasureSpec.UNSPECIFIED：父母没有对孩子施加任何约束。它可以是任何它想要的大小。

1. onLayout() 是一个在视图应该为其所有子视图分配大小和位置时调用的方法。
2. onDraw() 大小和位置是在前面的步骤中计算出来的，因此视图可以根据它们自己绘制。在 onDraw(Canvas canvas) 生成（或更新）的 Canvas 对象有一个 OpenGL-ES 命令列表（displayList）发送到 GPU。永远不要在 onDraw() 中创建对象，因为它会被多次调用。

当特定视图的属性发生变化时，还有另外两种方法可以发挥作用。那些是：

invalidate() 是一种坚持强制重绘我们希望显示更改的特定视图的方法。简单地说，当视图的外观发生变化时，需要调用 invalidate()。

requestLayout() 是给视图系统的信号，它需要重新计算视图的测量和布局阶段（测量→布局→绘制）。简单地说，当视图的边界发生变化时，需要调用 requestLayout()。

在任何View上调用任何方法时，必须始终在 UI 线程上。如果正在其他线程上工作并希望从该线程更新视图的状态，则应使用 Handler。

1. 状态保存/恢复

onSaveInstanceState()

要首先保存视图状态，需要为其提供一个 ID。如果视图层次结构有多个具有相同 ID 的视图，则它们的所有状态都会被保存，以便消除这种维护唯一 ID。其次，需要一个类来扩展 View.BaseSavedState，然后保存它们的属性。

onRestoreInstanceState(Parcelable state)

这里需要重写这个方法并从 Parcelable 中读取数据，然后根据 Parcelable 中可用的数据编写逻辑。

@Override

public Parcelable onSaveInstanceState() {

Bundle bundle = new Bundle();

// The vars you want to save - in this instance a string and a boolean

String someString = "something";

boolean someBoolean = true;

State state = new State(super.onSaveInstanceState(), someString, someBoolean);

bundle.putParcelable(State.STATE, state);

return bundle;

}

@Override

public void onRestoreInstanceState(Parcelable state) {

if (state instanceof Bundle) {

Bundle bundle = (Bundle) state;

State customViewState = (State) bundle.getParcelable(State.STATE);

// The vars you saved - do whatever you want with them

String someString = customViewState.getText();

boolean someBoolean = customViewState.isSomethingShowing());

super.onRestoreInstanceState(customViewState.getSuperState());

return;

}

// Stops a bug with the wrong state being passed to the super

super.onRestoreInstanceState(BaseSavedState.EMPTY\_STATE);

}

protected static class State extends BaseSavedState {

protected static final String STATE = "YourCustomView.STATE";

private final String someText;

private final boolean somethingShowing;

public State(Parcelable superState, String someText, boolean somethingShowing) {

super(superState);

this.someText = someText;

this.somethingShowing = somethingShowing;

}

public String getText(){

return this.someText;

}

public boolean isSomethingShowing(){

return this.somethingShowing;

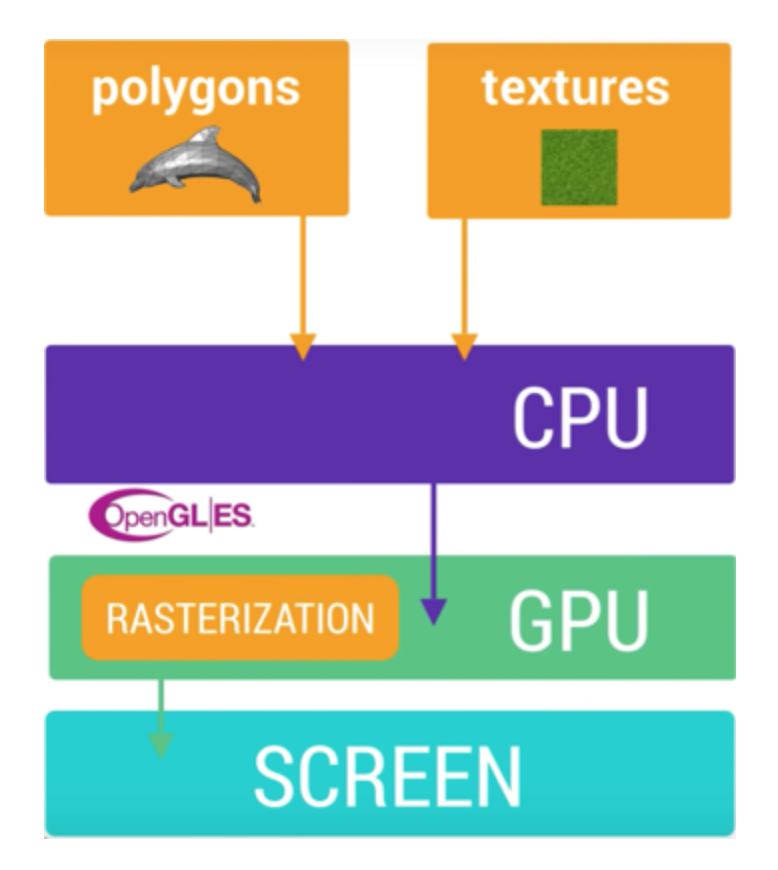
}

}

1. 光栅化

光栅化是获取一些高级对象（如字符串、按钮或形状）并将它们转换为屏幕纹理中的像素的过程。光栅化是一个非常耗时的过程。因此 GPU（图形处理单元）发挥作用以加速光栅化过程。 GPU 的设计方式是只使用特定类型，如多边形、纹理等。

CPU 负责将内容提供给 GPU，然后才能将任何内容绘制到屏幕上。

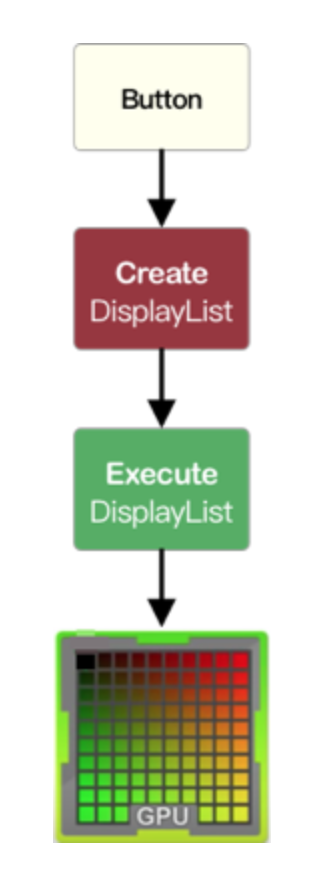


这个将数据从 CPU 传输到 GPU 的过程是由一个称为 OpenGL ES 的通用 API 完成的。在任何时间点，都需要将 UI 对象（如按钮、路径等）绘制到屏幕上。首先，它们被 CPU 转换成多边形和纹理，然后它们被传递到 GPU 进行光栅化。

提示：针对渲染性能进行优化意味着将尽可能多的数据放到 GPU 上，然后将其保留在那里并尽可能长时间地引用它而不进行修改。

1. Display List

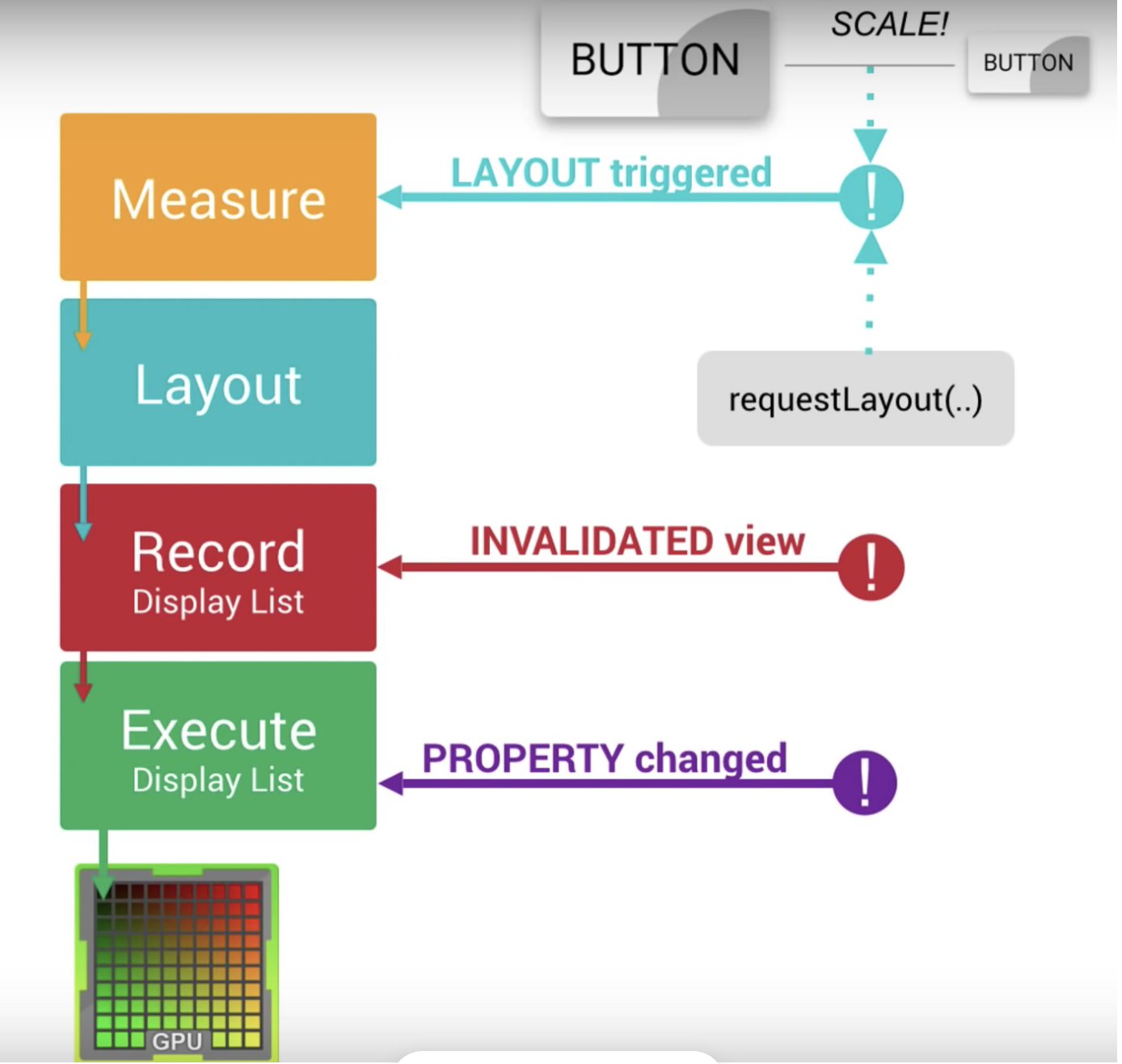
当需要渲染视图时，会为该视图创建一个显示列表。当需要将该视图呈现到屏幕上时，我们需要通过将其绘图命令提交给 GPU 来执行该显示列表。



如果视图的属性发生变化，我们只需要多执行一次该显示列表即可。但是以后如果视图发生了视觉上的变化，那么之前的显示列表就不再有效——所以我们需要重新创建显示列表并再次执行来更新屏幕。

提示：如果视图的内容在任何时候发生变化，它会重复重新创建和重新执行显示列表的过程。因此，性能因视图的复杂程度而异。

假设视图的大小发生了变化。然后当测量阶段开始时，它会询问视图层次结构中的每个视图它们的新尺寸是多少。如果视图的位置发生变化，则调用 requestLayout() 方法。或者，如果视图组重新定位其子级，则调用布局阶段并遍历需要定位每个子级的整个层次结构。



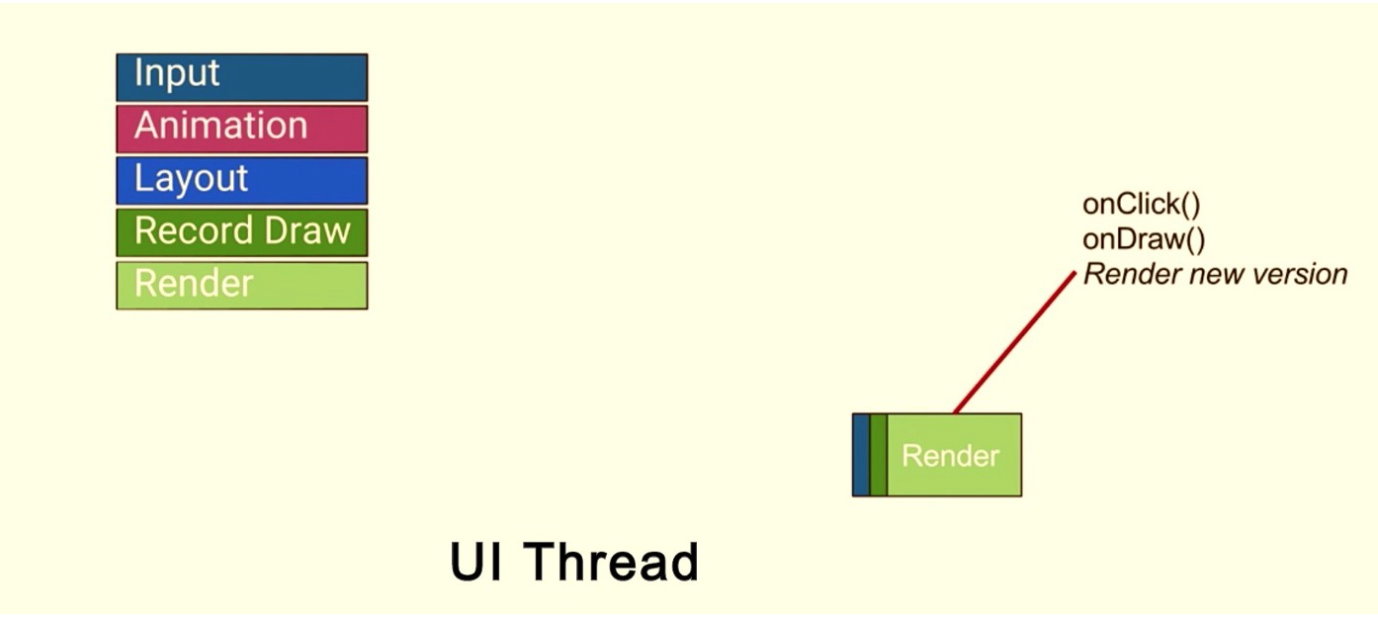
提示：因此，对于性能更好的应用程序，我们需要维护一个平面视图层次结构，这样依赖项就不会受到影响，从而减少更新视图的时间。

1. 参与组件

1.UI Thread

每个应用程序都有其运行 UI 对象（例如视图对象）的特殊线程；这个线程称为 UI 线程。只有在 UI 线程上运行的对象才能访问该线程上的其他对象。

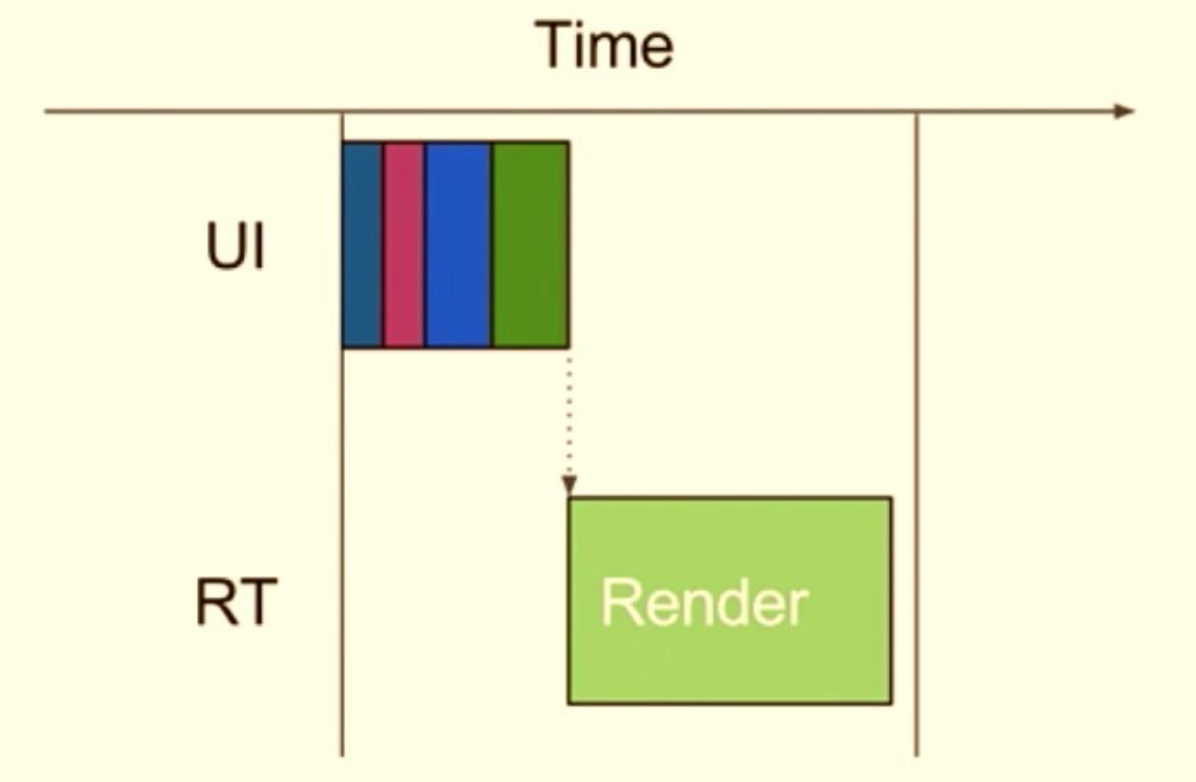
UI 线程准备了要在 Render 线程上执行的命令管道。从技术上讲，UI 线程负责通过处理视图的生命周期来创建信息。这意味着 UI 线程运行视图的 onMeasure()、onLayout()、onDraw() 并收集该信息并将其传递给 Render 线程。



例如，让我们考虑一个视图上有一个点击事件。 UI线程处理输入事件；如果视图中有任何属性更改，那么就会发生遍历来测量视图。在定位和放置视图阶段之后发生。最后一个阶段是根据上述信息绘制视图。如果上述信息准备就绪，那么我们将该信息从 UI 线程同步到 Render 线程。

1. Render Thread

渲染线程是 Android Lollipop 中已经引入的一个组件。它从 UI 线程获取输入并将它们处理到 GPU。我们假设 UI 线程和 Render 线程按顺序工作，因为如果没有来自 UI 线程的输入，渲染线程不知道该做什么。但是，实际上，它们是异步工作的，例如在某些动画和其他东西的情况下。但总的来说，我们确实准备了 UI 线程上的所有信息，并将这些信息传递给渲染线程，渲染线程是一个与 GPU 对话的单独线程。



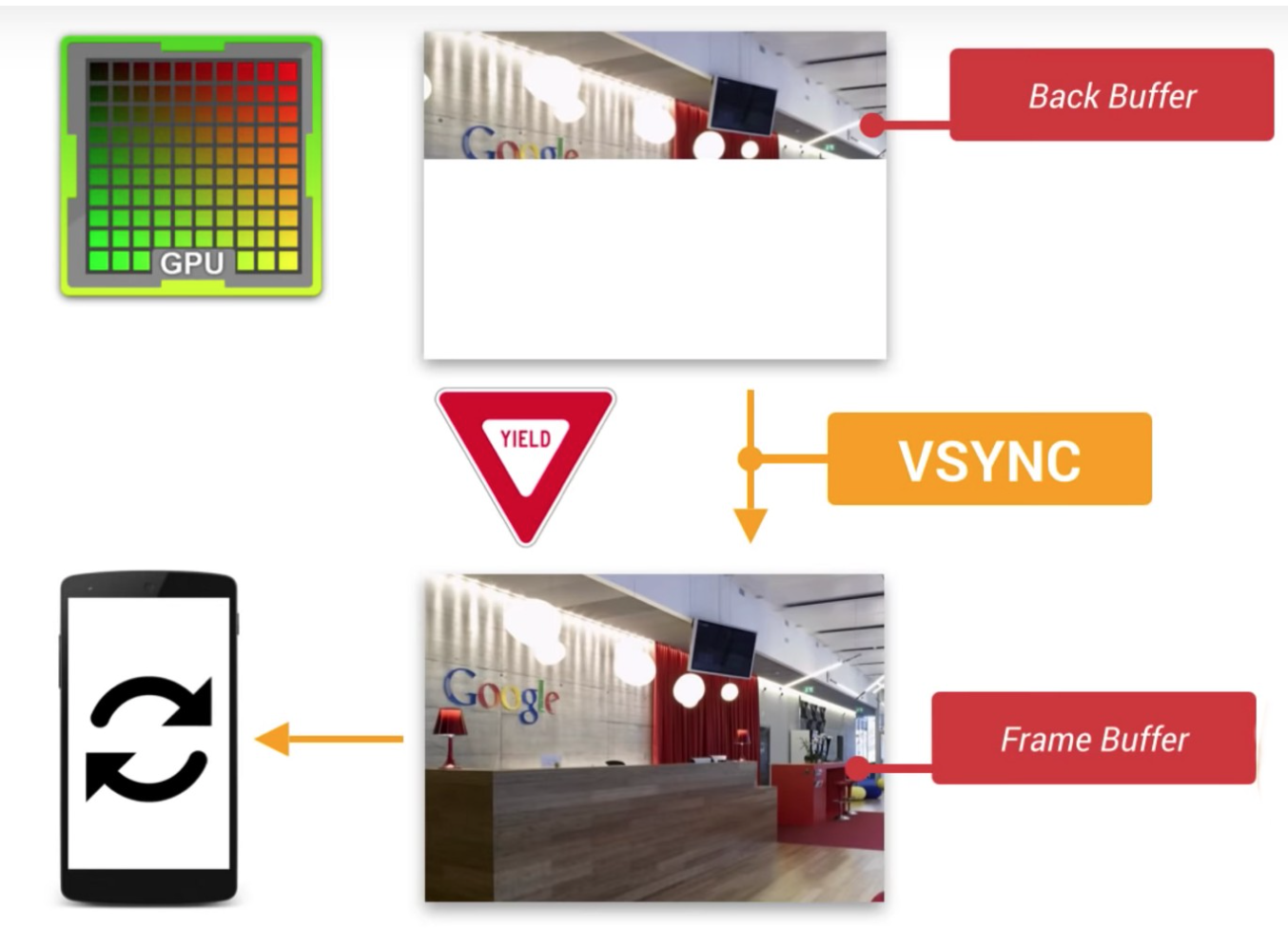
1. Choreographer

我们可以说 Choreographer 是一个充当底层显示系统和应用程序级视图系统之间的桥梁的类。Choreographer 从显示子系统接收定时脉冲，例如垂直同步，然后安排工作发生，作为渲染下一显示帧的一部分。

1. VSYNC

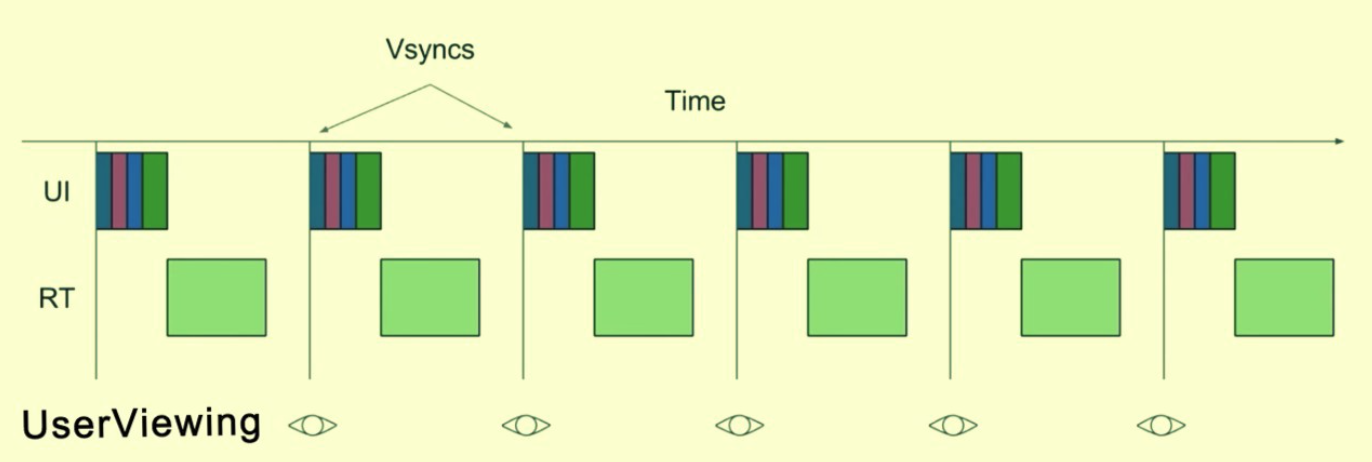
要了解 VSYNC，我们需要了解其他两个术语。一是刷新率，这意味着屏幕每秒可以更新其显示的次数。第二个术语是帧速率，这意味着您的 GPU 每秒可以绘制多少帧。这两者需要协同工作才能将图像正确地绘制到您的屏幕上。

GPU使用两个缓冲区后缓冲区和帧缓冲区。后台缓冲区用于绘制帧。完成后，GPU 将该帧复制到帧缓冲区。简而言之，帧缓冲区保存需要在屏幕上呈现的数据。但是，在将内容从后台缓冲区复制到帧缓冲区时，屏幕的刷新率会启动，这会从帧缓冲区获取数据。

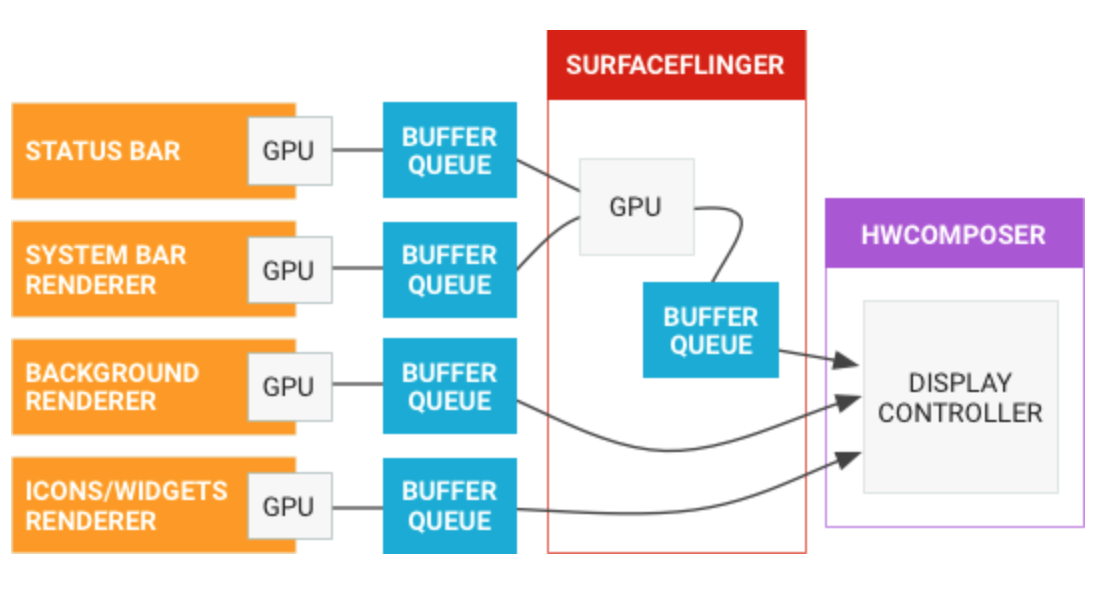


所以这里 VSYNC 或垂直同步就派上用场了。 VSYNC 保持从后台缓冲区到帧缓冲区的复制操作，以防屏幕刷新。因此，帧缓冲区中完全绘制的帧只会在屏幕上呈现。

VSYNC 通常每秒启动 60 次（大约 16.67 毫秒）。在引入 VSYNC 之前，我们经历了 UI 渲染的延迟，这是肉眼可以察觉的。

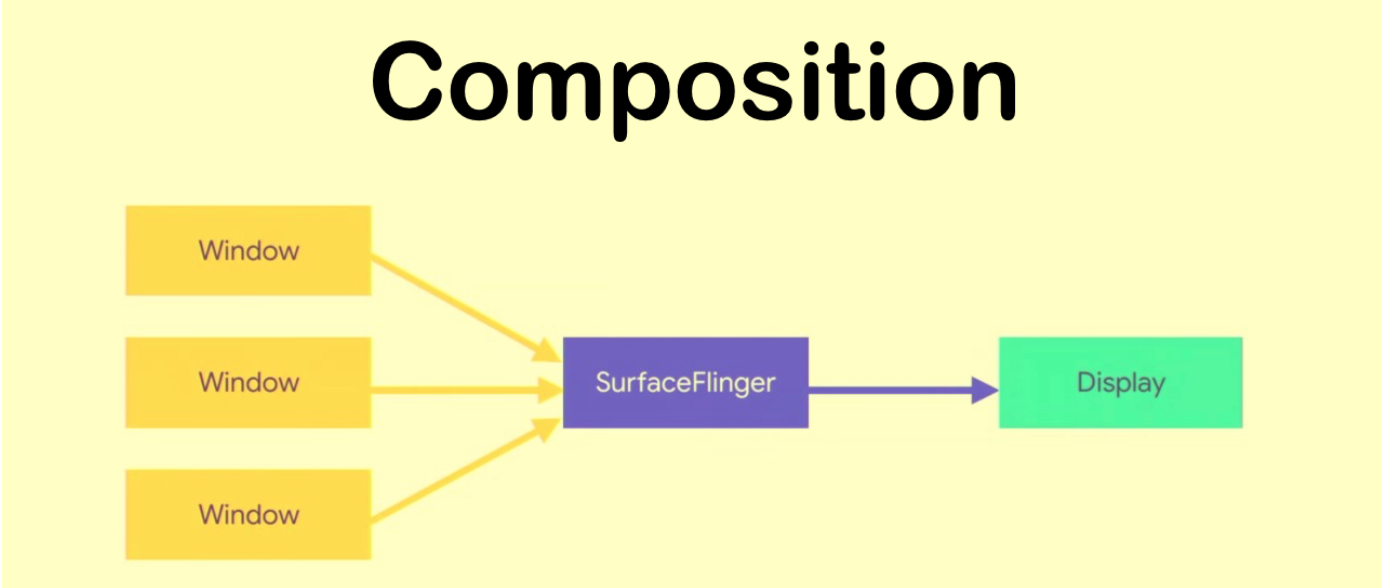


1. Flow



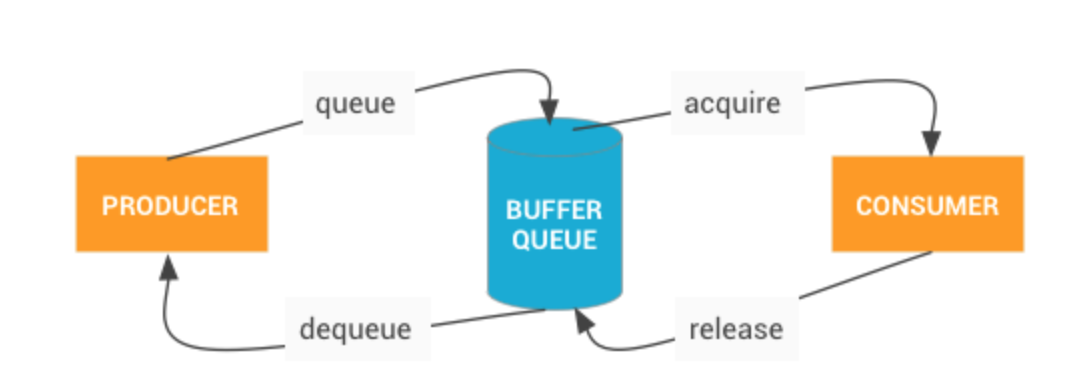
1. SurfaceFlinger

SurfaceFlinger 获取显示数据的缓冲区，将它们组合成一个缓冲区，然后将其传递给硬件抽象层 (HAL)。但是，SurfaceFlinger 专门用于包含图形和显示数据的缓冲区。

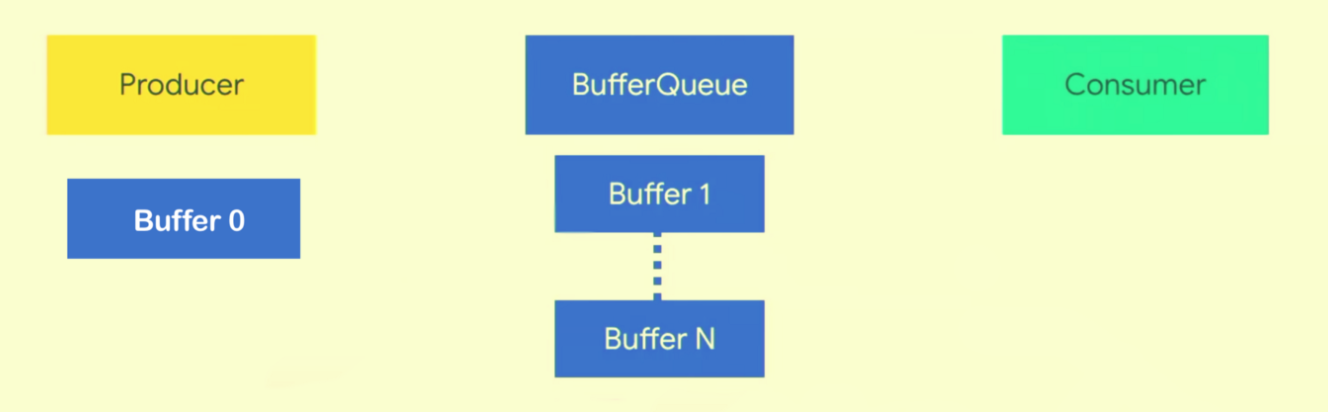


1. BufferQueue

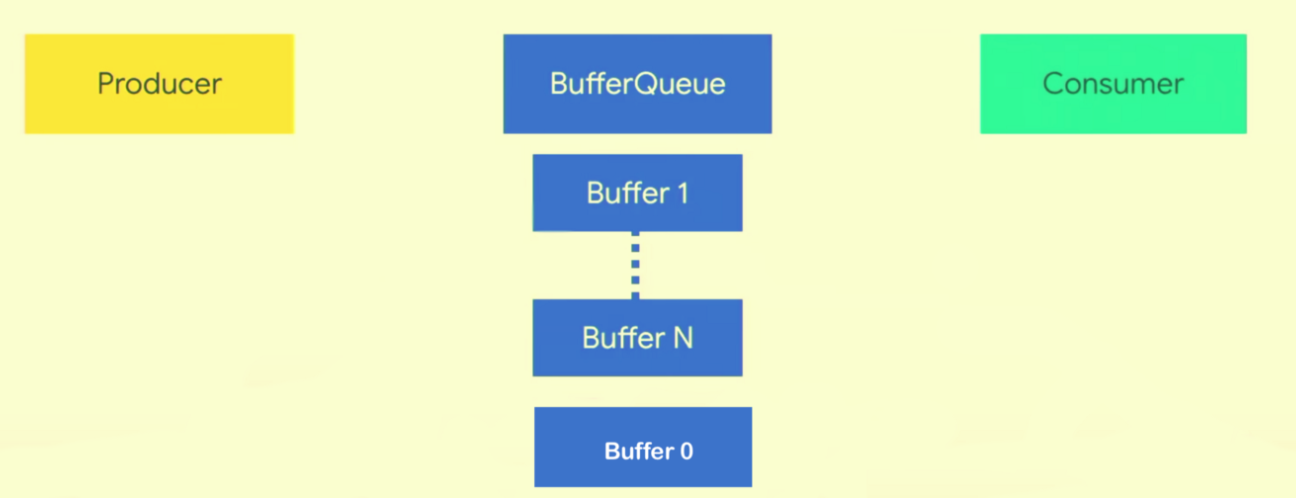
缓冲区只是分配的内存区域，用于在将某些内容移动到另一个地方时临时存储它。缓冲区队列只是我们的图形缓冲区所在的缓冲区队列。通常，我们将拥有一到三个缓冲区。缓冲队列主要有两个端点：一个是生产者，另一个是消费者。



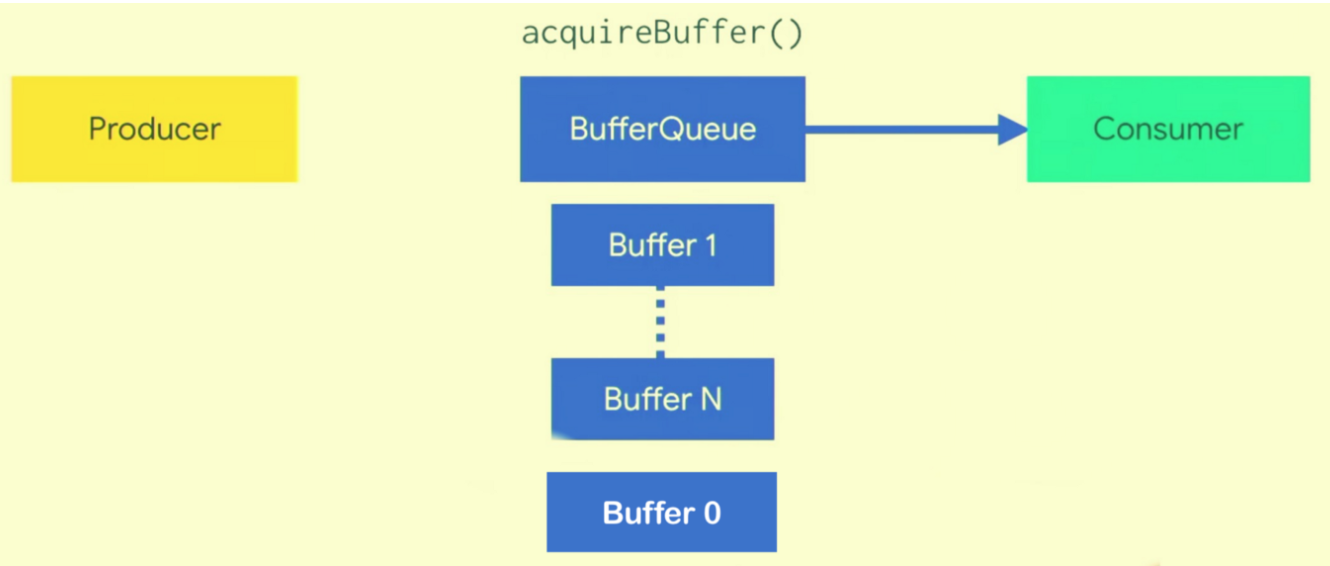
步骤 1：生产者在队列上调用一个名为 dequeueBuffer 的方法，现在缓冲区的状态为 DEQUEUED。现在，缓冲区的所有者就是生产者，它可以通过使用 OpenGL 或 Canvas 对缓冲区进行任何操作，如渲染、设置数据等。



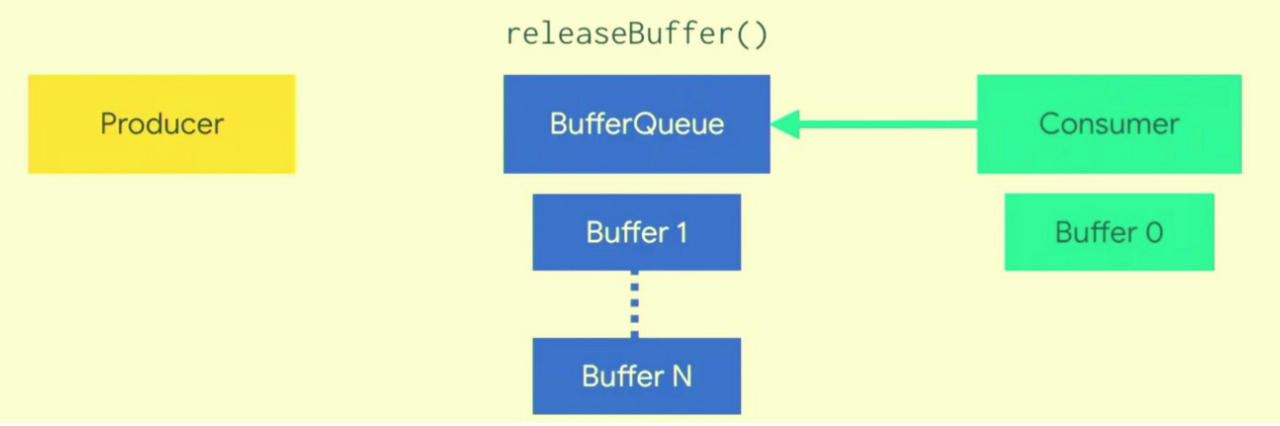
步骤 2：生产者插入数据完成后，它调用队列缓冲区，以便缓冲区重新排队进入 BufferQueue。此时缓冲区的状态由DEQUEUED -> QUEUED 改变，缓冲区所有者变为BufferQueue。



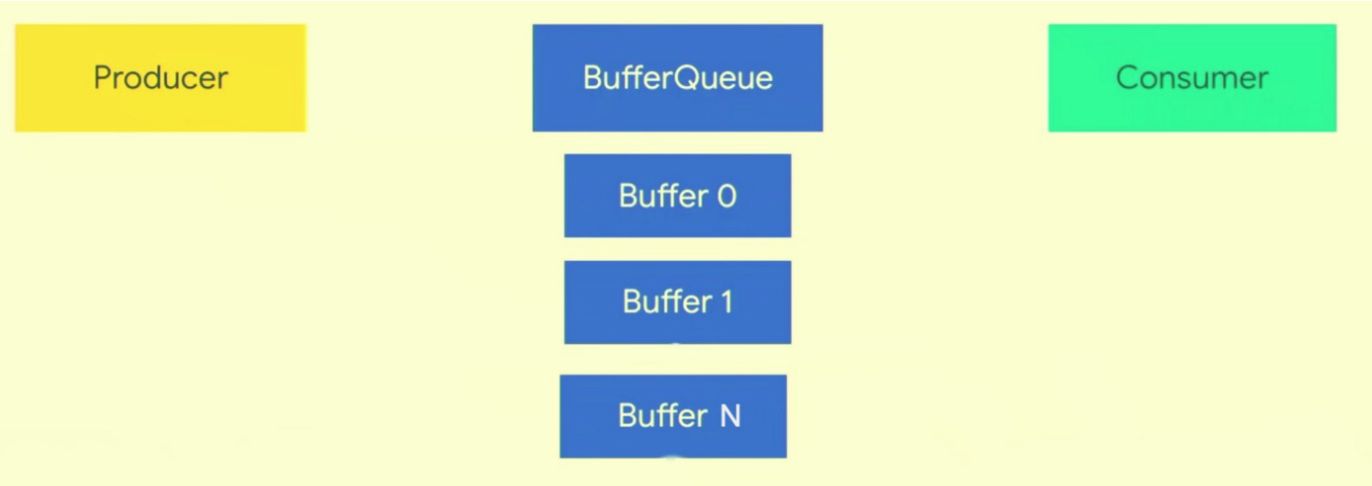
第 3 步：现在消费者将调用 BufferQueue 上的 acquireBuffer 并获取队列中可用的缓冲区。现在消费者是所有者；缓冲区的状态由 QUEUED-> ACQUIRED 更改。



第4步：消费者操作完成后，调用release，让buffer回到BufferQueue。此时，缓冲区的状态由 ACQUIRED -> FREE 改变，缓冲区所有者变为 BufferQueue。



最后，BufferQueue 状态



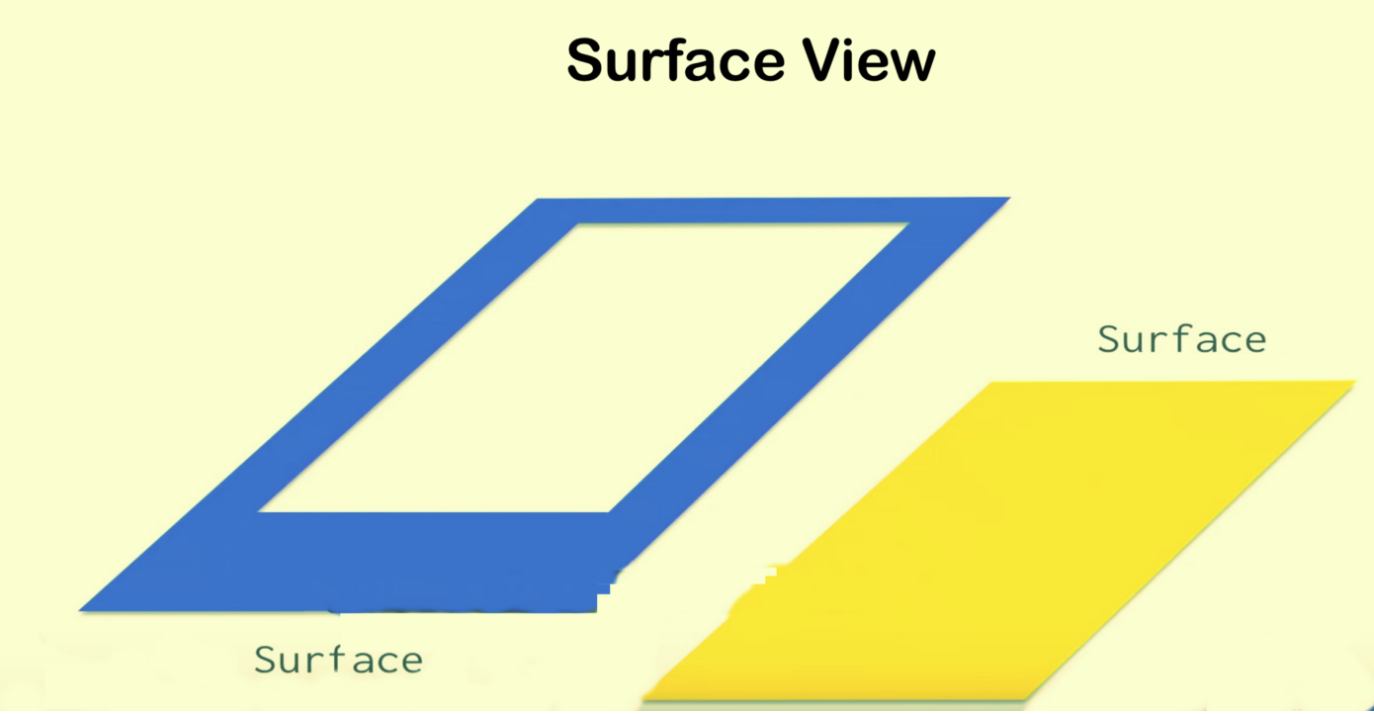
1. Display Pipeline

VSYNC 信号同步显示流水线。显示管道由应用程序渲染、SurfaceFlinger 合成和在显示器上呈现图像的硬件合成器 (HWC) 组成。 VYSNC 同步应用唤醒开始渲染的时间、SurfaceFlinger 唤醒合成屏幕的时间以及显示刷新周期。这种同步消除了卡顿并提高了图形的视觉性能。

HWC 生成 VSYNC 事件并通过回调将事件发送到 SurfaceFlinger。

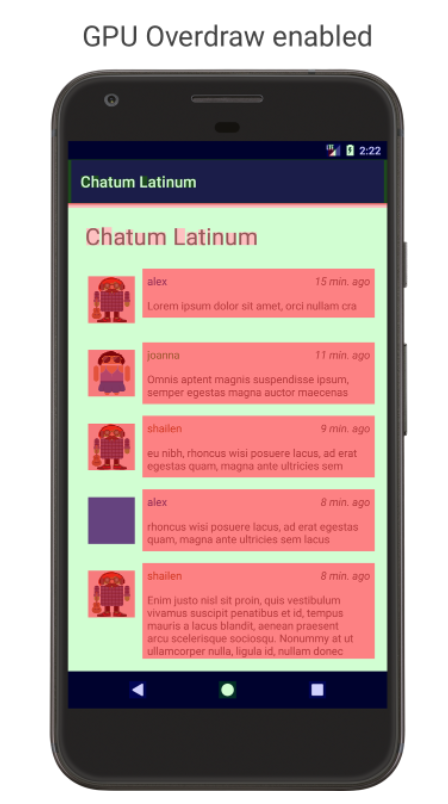
1. SurfaceView

SurfaceView 提供嵌入在视图层次结构中的专用绘图表面。您可以控制此表面的格式，如果您愿意，还可以控制它的大小； SurfaceView 负责将表面放置在屏幕上的正确位置。



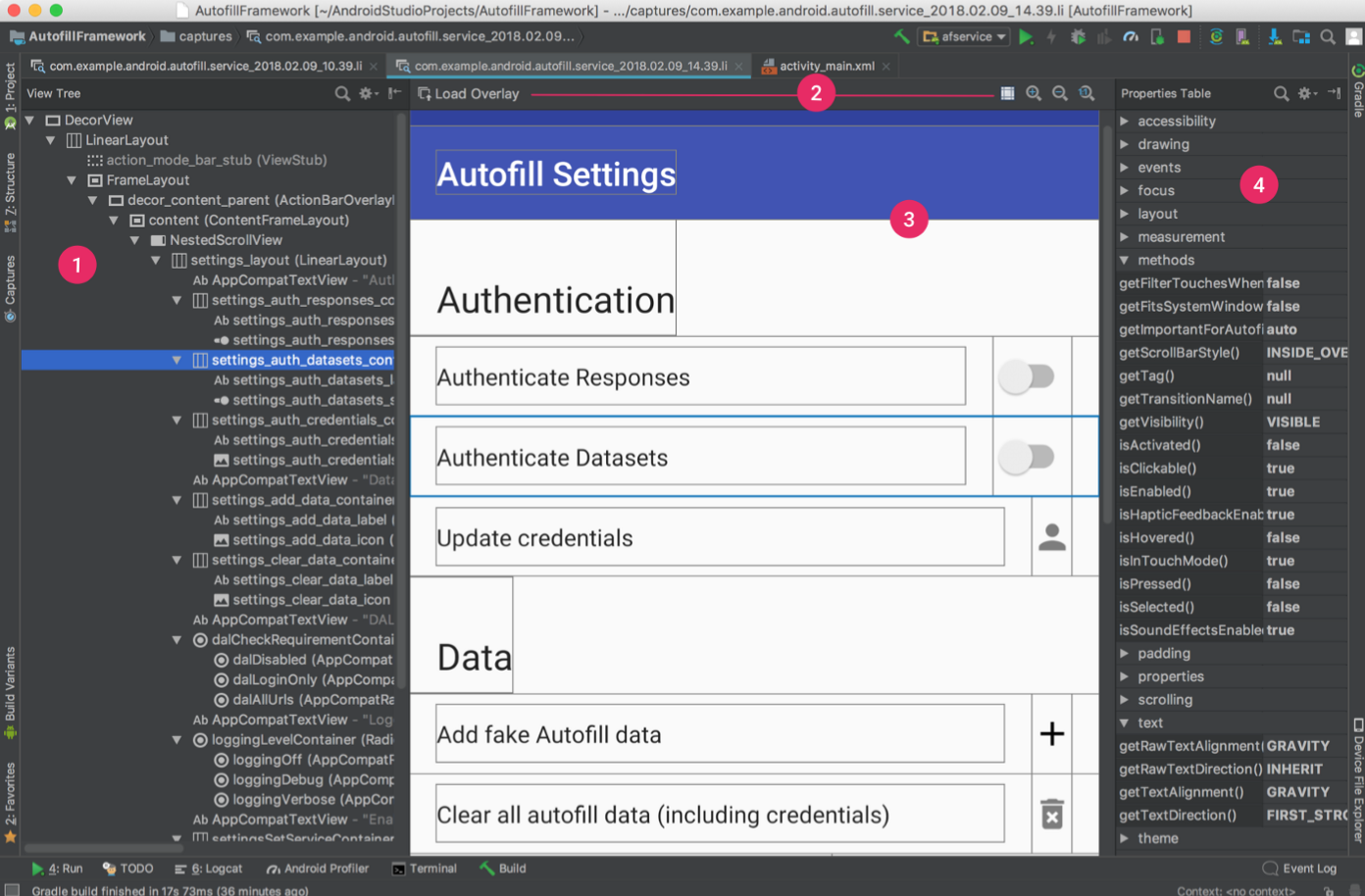
SurfaceView 就像窗口是它的表面一样工作。然后，它在 Surface 上精确地切割孔。我们要求 WindowManager 和 SurfaceFlinger 创建第二个 Surface，然后将其滑到第一个 Surface 的下方。它们是两个不同的 Surface 和不同的 BufferQueue；他们可以是独立的。下面是结果。

1. UI层级
2. 开启GPU过度绘制功能：查看重复绘制的区域；



1. Layout Inspector

Layout Inspector 允许将应用程序布局与设计模型进行比较，显示应用程序的放大视图，还允许在运行时检查其布局的细节。当布局在运行时在类级别而不是完全在 XML 中构建并且布局行为异常时，这将很有用。



1.视图树：布局中视图的层次结构

2.布局检查器工具栏：布局检查器的工具

3.屏幕截图：显示在您设备上的应用程序布局的屏幕截图，显示每个视图的布局边界

4.属性表：选定视图的布局属性

1. ConstraintLayout

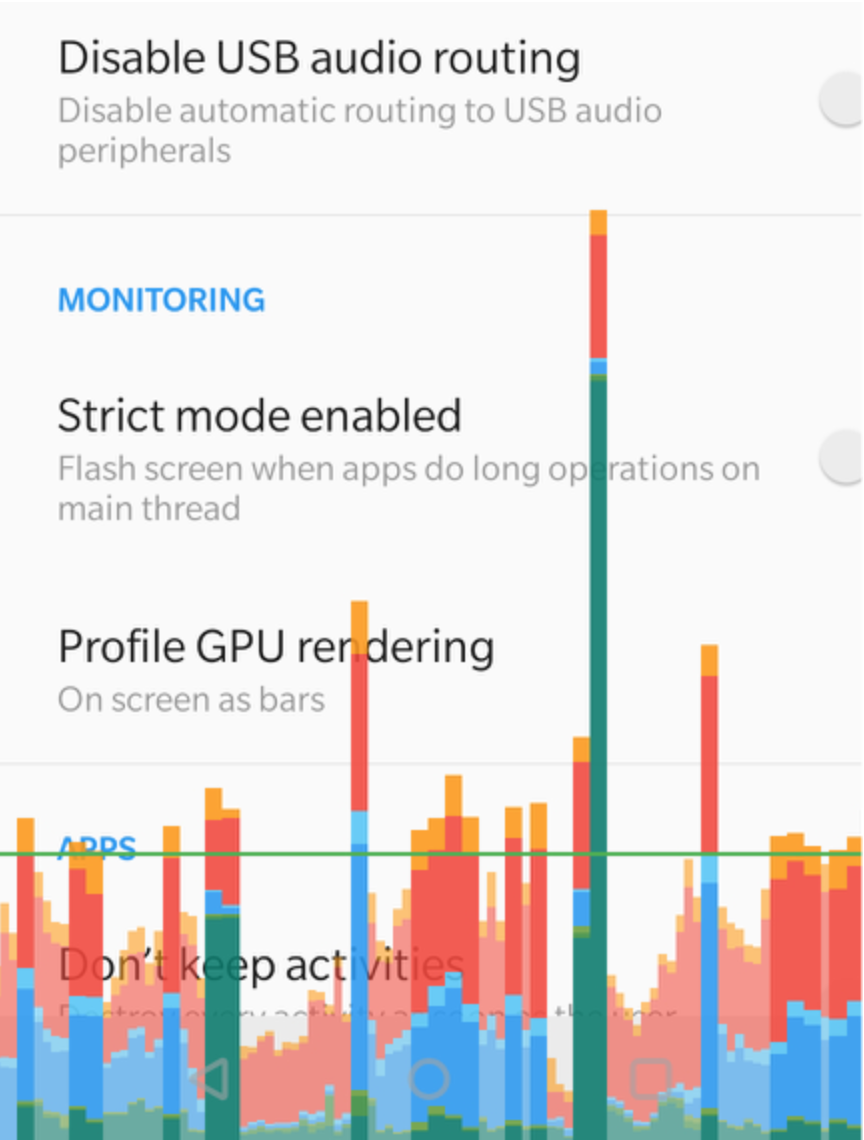
ConstraintLayout 允许创建具有平面视图层次结构的大型复杂布局，这意味着没有嵌套视图组。它类似于RelativeLayout，所有视图都根据兄弟视图和父布局之间的关系进行布局。

渲染视图主要分为三个阶段：测量阶段、布局阶段和绘制阶段。在这些阶段的每一个中，都会有一个从上到下的遍历。嵌套视图越多，渲染视图所需的时间和计算量就越多。因此，通过在我们的应用布局中保持平面视图层次结构，我们可以创建一个快速响应的用户界面。

ConstraintLayout 有许多属性，如 Guideline、Barrier 和 Chains，有助于创建平面视图层次结构。

二、流程分析

1、开启GPU渲染模式分析：“Profile GPU 渲染工具以滚动直方图的形式显示渲染 UI 窗口的帧相对于每帧 16 毫秒的基准所需的时间。”



水平绿线代表 16 毫秒的阈值。要达到每秒 60 帧，每帧的竖线需要保持在这条线以下。每当条形超过这条线时，动画可能会暂停。

下表提供了在使用运行 Android 6.0 及更高版本的设备时分析器输出中垂直条的每一段的描述。



GPU 渲染模式分析工具可以显示渲染流水线的每个阶段渲染前一帧所用的相对时间。这些信息有助于确定流水线中的瓶颈所在，从而了解应该优化哪些方面来提高应用的渲染性能。

1.输入处理

流水线的输入处理阶段测量的是应用处理输入事件所用的时间。该指标表示应用执行输入事件回调给出的代码所用的时间。

当此分段较长时

此区域中的值较高通常是由于输入处理程序事件回调中的工作过多或过复杂导致的。 由于这些回调总是发生在主线程上，因此该问题的解决方法主要是直接优化工作，或者将工作转移到其他线程。

另外值得注意的是，此阶段可能出现 RecyclerView 滚动。当 RecyclerView 处理轻触事件时，它会立即滚动。因此，它可能会膨胀或填充新的项目视图。由于这个原因，尽可能加快此操作的执行速度就非常重要。TraceView 或 Systrace 等分析工具可以帮助您进一步调查相关问题。

2.动画

动画阶段显示的是评估在该帧中运行的所有 Animator 所用的时间。最常见的 Animator 有 ObjectAnimator、ViewPropertyAnimator 和转换。

当此分段较长时

造成此区域中的值较高的原因，通常是因动画的某种属性更改而执行的工作。例如，投掷动画会滚动 ListView 或 RecyclerView，从而导致大量的视图膨胀和填充。

3.测量/布局

为了在屏幕上绘制视图项，Android 会在视图层次结构的布局和视图中执行两项特定操作。

首先，系统会测量视图项。每个视图和布局都包含描述对象在屏幕上的尺寸的具体数据。有些视图具有确切的尺寸，而有些视图的尺寸则可以根据父级布局容器的尺寸进行调整。

然后，系统会对视图项进行布局。系统计算出子视图的尺寸后，就可以进行布局，调整视图在屏幕上的尺寸和位置。

系统不仅会对要绘制的视图进行测量和布局，还会逐层向上对这些视图的父级层次结构执行测量和布局，一直到根视图。

当此分段较长时

如果您的应用在这方面对每帧花费的时间较多，通常是由于需要布局的视图数量过多，或者出现了其他问题（例如在层次结构的错误位置发生了 Double Taxation 等）。在这两种情况下，要解决性能问题，都需要改善视图层次结构的性能。

此外，添加到 onLayout(boolean, int, int, int, int) 或 onMeasure(int, int) 的代码也可能会导致性能问题。Traceview 和 Systrace 可以帮助您检查调用堆栈，以找出代码可能存在的问题。

4.绘制

绘制阶段将视图的渲染操作（如绘制背景或文本）转换为一系列原生绘制命令。系统会将这些命令捕获到一个显示列表中。

“绘制”条记录了将这些命令捕获到显示列表中所用的时间，包括这一帧中需要在屏幕上更新的所有视图对应的命令。测量的时间涵盖了您添加到应用中的界面对象的所有代码。此类代码的示例包括 onDraw()、dispatchDraw() 以及属于 Drawable 类的子类的各种 draw ()methods。

当此分段较长时

简单地说，您可以将该指标理解为针对每个失效视图运行所有 onDraw() 调用所用的时间。该测量值涵盖了将绘制命令分配给可能存在的子项和可绘制对象所用的任何时间。因此，当您看到此竖条出现峰值时，可能是因为大量视图突然失效了。失效意味着必须重新生成视图的显示列表。或者，一些自定义视图的 onDraw() 方法中存在某种极其复杂的逻辑，也可能会导致此时间很长。

5.同步/上传

“同步和上传”指标表示在当前帧中将位图对象从 CPU 内存传输到 GPU 内存所需的时间。

作为两种不同的处理器，CPU 和 GPU 将不同的 RAM 区域用于执行处理。当您在 Android 中绘制位图时，系统会先将位图传输到 GPU 内存，然后 GPU 才能将其渲染到屏幕上。之后，GPU 会缓存位图，以便系统无需再次传输数据，除非纹理被移出 GPU 纹理缓存。

当此分段较长时

帧的所有资源都必须位于 GPU 内存中才能用来绘制帧。这意味着，该指标的值较高可能表示需要加载大量小型资源或少量大型资源。一种常见的情况是应用要显示一个接近屏幕尺寸的位图。另一种情况是应用要显示大量缩略图。

要减小该值，您可以采用以下技巧：

确保位图的分辨率不会比位图的显示尺寸大很多。例如，您的应用应避免将 1024x1024 的图片显示为 48x48 的图片。

利用 prepareToDraw() 在下一个同步阶段之前异步预上传位图。

6.发出命令

“发出命令”分段表示的是发出将显示列表绘制到屏幕上所需的全部命令所需的时间。

为了将显示列表绘制到屏幕上，系统会向 GPU 发送必要的命令。通常，系统会通过 OpenGL ES API 执行此操作。

此过程需要一些时间，因为在将命令发送给 GPU 之前，系统会对每个命令执行最后的转换和裁剪。然后 GPU 会计算最终的命令，使得 GPU 端的开销增加。这些命令包括最后的转换和额外的裁剪。

当此分段较长时

在此阶段花费的时间直接反映了系统在给定帧中渲染的显示列表的复杂程度和数量。例如，如果有大量绘制操作，特别是在每个绘制基元都有较少的固有开销的情况下，将会使得这项时间增加。 例如：

for (i in 0 until 1000) {

canvas.drawPoint()

}

发出以上命令的开销会远远大于发出以下命令的开销：

canvas.drawPoints(thousandPointArray)

发出命令与实际绘制显示列表之间并不总是存在 1 对 1 的关系。“发出命令”阶段捕获的是将绘制命令发送到 GPU 所用的时间，而“绘制”指标表示的是将已发出的命令捕获到显示列表所用的时间。

之所以会出现这种差异，是因为系统会尽可能地缓存显示列表。因此，在某些情况下，滚动、转换或动画会要求系统重新发送显示列表，但不必实际重新构建它（即重新捕获绘制命令）。因此，您可能会看到“发出命令”条较高，但“绘制命令”条并不高。

7.处理/交换缓冲区

当 Android 将其所有显示列表提交给 GPU 后，系统会发出最后一条命令，告诉图形驱动程序它已完成当前帧的处理。此时，驱动程序即可将更新后的图像显示到屏幕上。

当此分段较长时

有一点必须要注意：GPU 与 CPU 是并行工作的。Android 系统向 GPU 发出绘制命令，然后继续执行下一个任务。GPU 从队列中读取并处理这些绘制命令。

如果 CPU 发出命令的速度快于 GPU 处理命令的速度，这两个处理器之间的通信队列就会被占满。出现这种情况时，CPU 会阻塞并等待，直到队列中有位置来放置下一个命令。这种队列占满状态通常出现在“交换缓冲区”阶段，因为此时已提交了整个帧的命令。

缓解此问题的关键是降低 GPU 工作的复杂度，就像您在“发出命令”阶段所做的那样。

8.其他

除了渲染系统执行其工作所用的时间外，主线程上还会执行一些与渲染无关的工作。这些工作耗费的时间被报告为“其他时间”。“其他时间”通常表示可能在两个连续渲染帧之间的界面线程上执行的工作。

当此分段较长时

如果该值很高，则表示您的应用可能包含应在其他线程上执行的回调、Intent 或其他工作。通过方法跟踪或 Systrace 等工具可以查看主线程上运行的任务。此信息可帮助您有针对性地改进性能。

1. Systrace

Systrace 是一个工具，它允许我们收集有关我们设备上正在运行的内容的精确时间信息，然后将其可视化。记录 Systrace 的方法有以下三种：通过我们手机上可用的应用程序、通过系统命令行和通过 Perfetto 命令行工具。

Systrace 生成一个包含一系列部分的输出文件。该报告列出了每个进程的线程。如果给定线程呈现 UI 帧，则报告还会沿时间线指示呈现的帧。当您在报告中从左向右移动时，时间会向前流逝。

最佳实践

1、在测量 UI 性能时，请在发布版本上使用任何工具，而不是调试给出适当结果的版本。

2、尽量避免阴影渲染，这会使 onDraw() 成本更高。

3、保持平面视图层次结构，因为在复杂和深度视图层次结构的情况下，布局遍历需要更多时间。

ConstraintsLayout 有助于通过简单的约束在很大程度上减少布局的嵌套。可以通过优化视图层次结构来减少重叠 UI 对象的数量来提高性能。避免不必要的布局。

4、避免不必要的背景，这会导致一些渲染开销，例如，父母和孩子的背景相同。

<LinearLayout android:id="@+id/parent" android:background=”@android:color/black”>

<android.support.v4.widget.NestedScrollView>

<LinearLayout android:id="@+id/child" android:background=”@android:color/black”>

<! — some stuff here →

</LinearLayout>

</android.support.v4.widget.NestedScrollView>

</LinearLayout>

1. 尽量避免 Alpha 渲染。

在屏幕上渲染透明像素称为 Alpha 渲染。透明动画、淡出和阴影等视觉效果都涉及某种透明度，因此会显着影响过度绘制。在这些情况下，可以通过减少渲染的透明对象的数量来改善过度绘制。

例如，可以通过在 TextView 中绘制黑色文本并设置半透明 alpha 值来获取灰色文本。但是您可以通过简单地将文本绘制为灰色来获得相同的效果和更好的性能。

1. Double Taxation

通常情况下，框架会在一次遍历中快速执行布局或度量阶段。但在一些情况比较复杂的布局中，在最终放置元素之前，框架可能必须对层次结构中需要多次遍历才能解析的部分执行多次迭代。必须执行不止一次“布局和度量”迭代的情况称为“Double Taxation”。

例如，当您使用 RelativeLayout 容器时（该容器允许您根据其他 View 对象的位置来放置 View 对象），框架会执行以下操作：

1.执行一次“布局和度量”遍历。在此过程中，框架会根据每个子对象的请求计算该子对象的位置和大小。

2.结合此数据和对象的权重确定关联视图的恰当位置。

3.执行第二次布局遍历，以最终确定对象的位置。

4.进入渲染过程的下一个阶段。

此外，RelativeLayout 以外的容器也可能会导致 Double Taxation。例如：

1.将 LinearLayout 视图设置为水平方向，可能会导致执行双重“布局和度量”遍历。如果您添加 measureWithLargestChild，则垂直方向上也可能会发生双重“布局和度量”遍历，因为在这种情况下，框架可能需要执行第二次遍历才能正确解析对象的大小。

2.GridLayout 也有类似的问题。虽然该容器也允许相对定位，但它通常会通过预处理子视图之间的位置关系来避免 Double Taxation。不过，如果布局使用权重或使用 Gravity 类来填充，则会失去该预处理带来的好处，当容器为 RelativeLayout 时，框架可能必须执行多次遍历。

多次“布局和度量”遍历本身并不是性能负担，但如果发生在错误的地方，就可能会变成负担。您应该警惕容器存在以下情况：

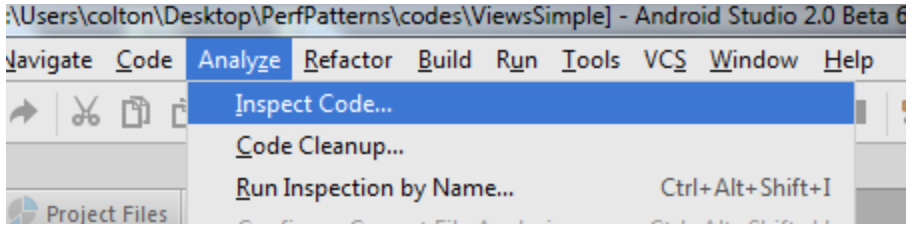
1.它是视图层次结构中的根元素。

2.它下面有较深的视图层次结构。

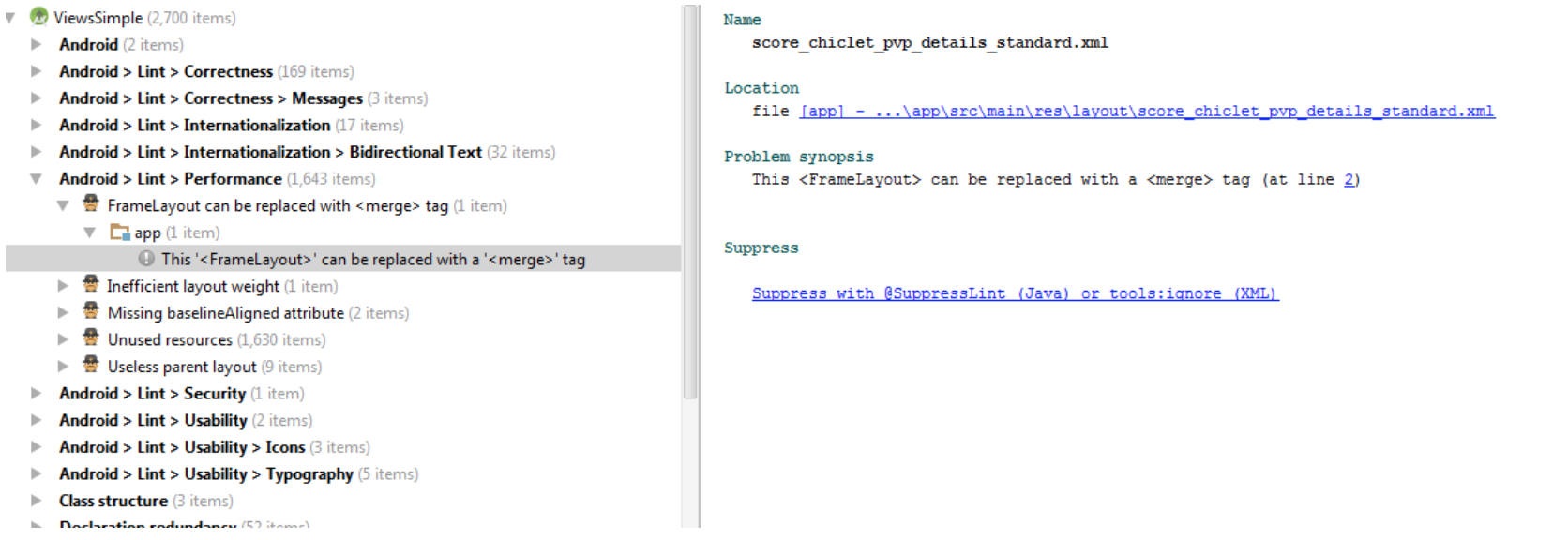
3.屏幕中填充了它的许多实例，类似于 ListView 对象中的子对象。

7、Lint

Android Studio 的 Lint 工具可以帮助您了解视图层次结构中的低效问题。如需使用此工具，请依次选择 Analyze > Inspect Code。



Android > Lint > Performance 下面显示了有关各种布局项目的信息。如需查看更多详情，可以点击各个项目将其展开，然后在屏幕右侧的窗格中会显示详细信息。图 2 显示了此界面的一个示例。



1. 采用 merge/include

造成多余嵌套布局的一个常见原因就是 <include> 标记。例如，您可以定义一个类似如下的可重复使用的布局：

<LinearLayout>

<!-- some stuff here -->

</LinearLayout>

然后，定义一个 include 标记将此项目添加到父容器中：

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:background="@color/app\_bg"

android:gravity="center\_horizontal">

<include layout="@layout/titlebar"/>

<TextView android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="@string/hello"

android:padding="10dp" />

</LinearLayout>

该 include 会将第一个布局嵌套在第二个布局中，而此嵌套是不必要的。merge 标记可以避免此问题。

9、X2C