

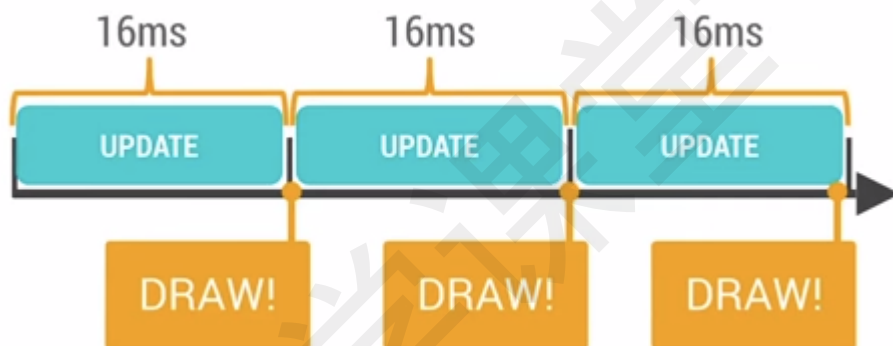
卡顿分析与布局优化

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。Android系统每隔大概16.6ms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。

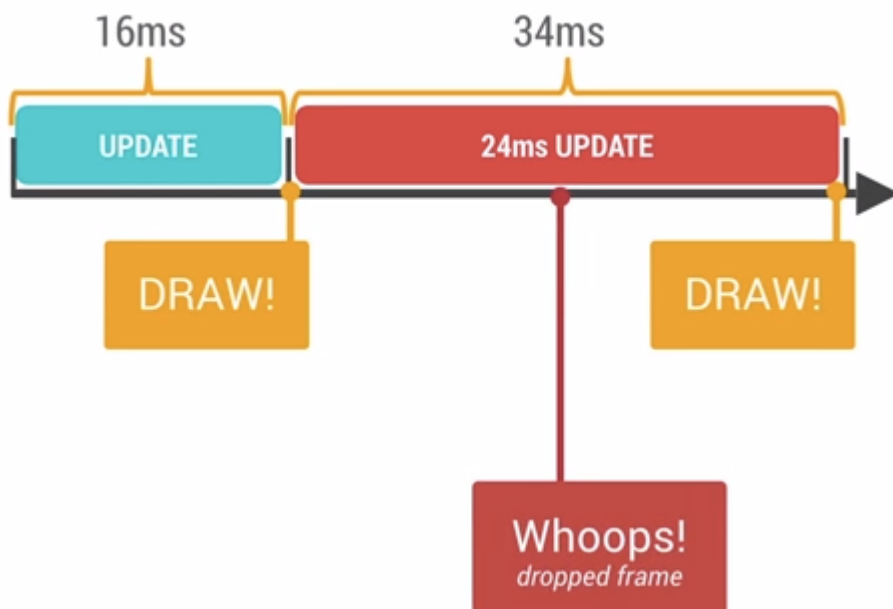
我们通常都会提到60fps与16ms，可是知道为何会是以程序是否达到60fps来作为App性能的衡量标准吗？这是因为人眼与大脑之间的协作无法感知超过60fps的画面更新。

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 $16\text{ms} = 1000/60$ 的时间来处理所有的任务。



如果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。



有很多原因可以导致丢帧，一般主线程过多的UI绘制、大量的IO操作或是大量的计算操作占用CPU，都会导致App界面卡顿。

一般主线程过多的UI绘制、大量的IO操作或是大量的计算操作占用CPU，导致App界面卡顿。

建议阅读：[Android图形显示系统](#)

卡顿分析

Systrace

Systrace 是Android平台提供的一款工具，用于记录短期内的设备活动。该工具会生成一份报告，其中汇总了Android 内核中的数据，例如 CPU 调度程序、磁盘活动和应用线程。Systrace主要用来分析绘制性能方面的问题。在发生卡顿时，通过这份报告可以知道当前整个系统所处的状态，从而帮助开发者更直观的分析系统瓶颈，改进性能。

TraceView可以看出代码在运行时的一些具体信息，方法调用时长，次数，时间比率，了解代码运行过程的效率问题，从而针对性改善代码。所以对于可能导致卡顿的耗时方法也可以通过TraceView检测。

要使用Systrace，需要先安装 **Python2.7**。安装完成后配置环境变量 `path`，随后在命令行输入：`python --version` 进行验证。

```
C:\Users\Administrator>python --version
Python 2.7.16
```

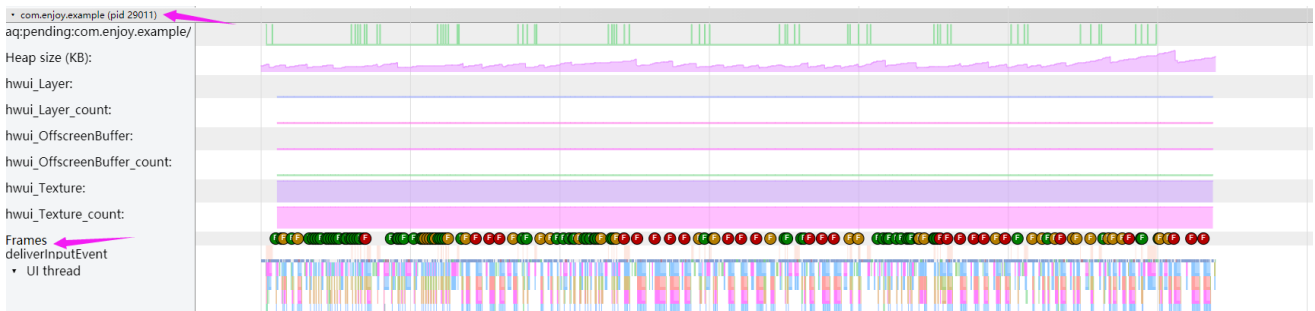
Systrace具体使用可以查看Zero老师博客：<https://www.jianshu.com/p/e73768e66b8d>

执行systrace可以选择配置自己感兴趣的category，常用的有：

标签	描述
gfx	Graphics 图形系统，包括SurfaceFlinger, VSYNC消息, Texture, RenderThread等
input	Input输入系统，按键或者触摸屏输入；分析滑动卡顿等
view	View绘制系统的相关信息，比如onMeasure, onLayout等；分析View绘制性能
am	ActivityManager调用的相关信息；分析Activity的启动、跳转
dalvik	虚拟机相关信息；分析虚拟机行为，如 GC停顿
sched	CPU调度的信息，能看到CPU在每个时间段在运行什么线程，线程调度情况，锁信息。
disk	IO信息
wm	WindowManager的相关信息
res	资源加载的相关信息

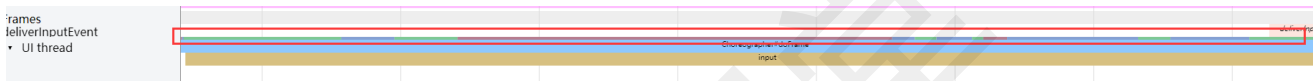
其实Systrace对于应用开发者来说，能看的并不多。主要用于看是否丢帧，与丢帧时系统以及我们应用大致的一个状态。

```
python systrace.py -t 5 -o F:\Lance\optimizer\lsn2_jank\1.a.html gfx input view am dalvik sched wm disk res -a com.enjoy.example
```



我们在抓取systrace文件的时候，切记不要抓取太长时间，也不要太多不同操作。

打开抓取的html文件，可以看到我们应用存在非常严重的掉帧，不借助工具直接用肉眼看应用UI是看不出来的。如果只是单独存在一个红色或者黄色的都是没关系的。关键是连续的红/黄色或者两帧间隔非常大那就需要我们去仔细观察。按“W”放大视图，在UIThread（主线程）上面有一条很细的线，表示线程状态。



Systrace 会用不同的颜色来标识不同的线程状态, 在每个方法上面都会有对应的线程状态来标识目前线程所处的状态。通过查看线程状态我们可以知道目前的瓶颈是什么, 是 CPU 执行慢还是因为 Binder 调用, 又或是进行 IO 操作, 又或是拿不到 CPU 时间片。通过查看线程状态我们可以知道目前的瓶颈是什么, 是 CPU 执行慢还是因为 Binder 调用, 又或是进行 IO 操作, 又或是拿不到 CPU 时间片

线程状态主要有下面几个：

- **绿色：表示正在运行；**
 - 是否频率不够？（CPU处理速度）
 - 是否跑在了小核上？（不可控，但实际上很多手机都会有游戏模式，如果我们应用是手游，那系统会优先把手游中的任务放到大核上跑。）
- **蓝色：表示可以运行，但是CPU在执行其他线程；**
 - 是否后台有太多的任务在跑？Runnable 状态的线程状态持续时间越长，则表示 cpu 的调度越忙，没有及时处理到这个任务
 - 没有及时处理是因为频率太低？
- **紫色：表示休眠，一般表示IO；**



官方介绍为：

橙色：不可中断的休眠

线程在遇到 I/O 操作时被阻止或正在等待磁盘操作完成。

紫色：可中断的休眠

线程在遇到另一项内核操作（通常是内存管理）时被阻止。

但是实际从Android 9模拟器中拉取数据，遇到IO显示紫色，没有橙色状态显示。

- **白色：表示休眠**，可能是因为线程在互斥锁上被阻塞，如Binder堵塞/Sleep/Wait等

Trace API

其实对于APP开发而言，使用systrace的帮助并不算非常大，大部分内容用于设备真机优化之类的系统开发人员观察。systrace无法帮助应用开发者定位到准确的错误代码位置，我们需要凭借很多零碎的知识点与经验来猜测问题原因。如果我们有了大概怀疑的具体的代码块或者有想了解的代码块执行时系统的状态，还可以结合 **Trace API** 打标签。

Android 提供了Trace API能够帮助我们记录收集自己应用中的一些信息：`Trace.beginSection()` 与 `Trace.endSection()`;

```
public class MainActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        TraceCompat.beginSection("enjoy_launcher"); //Trace.beginSection()  
        setContentView(R.layout.activity_main);  
        TraceCompat.endSection(); //Trace.endSection()  
    }  
}
```



Wall Duration: 执行耗时

CPU Durtation: 占用CPU耗时

如果掌握了更精准的问题在哪个阶段，可以结合**TraceView**查看更详细的方法情况。

App层面监控卡顿

systrace可以让我们了解应用所处的状态，了解应用因为什么原因导致的。若需要准确分析卡顿发生在什么函数，资源占用情况如何，目前业界两种主流有效的app监控方式如下：

- 1、利用UI线程的Looper打印的日志匹配；
- 2、使用Choreographer.FrameCallback

Looper日志检测卡顿

Android主线程更新UI。如果界面1秒钟刷新少于60次，即FPS小于60，用户就会产生卡顿感觉。简单来说，Android使用消息机制进行UI更新，UI线程有个Looper，在其loop方法中会不断取出message，调用其绑定的Handler在UI线程执行。如果在handler的dispatchMessage方法里有耗时操作，就会发生卡顿。

```
public static void loop() {
    //.....
    for (;;) {
        //.....
        Printer logging = me.mLogging;
        if (logging != null) {
            logging.println(">>>> Dispatching to " + msg.target + " " +
                msg.callback + ": " + msg.what);
        }

        msg.target.dispatchMessage(msg);
        if (logging != null) {
            logging.println("<<<< Finished to " + msg.target + " " + msg.callback);
        }
        //.....
    }
}
```

只要检测 `msg.target.dispatchMessage(msg)` 的执行时间，就能检测到部分UI线程是否有耗时的操作。注意到这行执行代码的前后，有两个`logging.println`函数，如果设置了`logging`，会分别打印出**>>>> Dispatching to**和**<<<< Finished to**这样的日志，这样我们就可以通过两次log的时间差值，来计算`dispatchMessage`的执行时间，从而设置阈值判断是否发生了卡顿。

```
public final class Looper {
    private Printer mLogging;
    public void setMessageLogging(@Nullable Printer printer) {
        mLogging = printer;
    }
}

public interface Printer {
    void println(String x);
}
```

Looper 提供了 `setMessageLogging(@Nullable Printer printer)` 方法，所以我们可以自己实现一个Printer，在通过 `setMessageLogging()` 方法传入即可：

```
public class BlockCanary {
    public static void install() {
        LogMonitor logMonitor = new LogMonitor();
        Looper.getMainLooper().setMessageLogging(logMonitor);
    }
}

public class LogMonitor implements Printer {

    private StackSampler mStackSampler;
    private boolean mPrintingStarted = false;
    private long mStartTimestamp;
    // 卡顿阈值
    private long mBlockThresholdMillis = 3000;
    //采样频率
    private long mSampleInterval = 1000;

    private Handler mLogHandler;

    public LogMonitor() {
        mStackSampler = new StackSampler(mSampleInterval);
        HandlerThread handlerThread = new HandlerThread("block-canary-io");
        handlerThread.start();
        mLogHandler = new Handler(handlerThread.getLooper());
    }

    @Override
    public void println(String x) {
        //从if到else会执行 dispatchMessage，如果执行耗时超过阈值，输出卡顿信息
        if (!mPrintingStarted) {
            //记录开始时间
            mStartTimestamp = System.currentTimeMillis();
            mPrintingStarted = true;
            mStackSampler.startDump();
        } else {
            final long endTime = System.currentTimeMillis();
            mPrintingStarted = false;
            //出现卡顿
            if (isBlock(endTime)) {
                notifyBlockEvent(endTime);
            }
            mStackSampler.stopDump();
        }
    }

    private void notifyBlockEvent(final long endTime) {
        mLogHandler.post(new Runnable() {
            @Override
            public void run() {
                //获得卡顿时 主线程堆栈
            }
        });
    }
}
```

```

        List<String> stacks = mStackSampler.getStacks(mStartTimestamp, endTime);
        for (String stack : stacks) {
            Log.e("block-canary", stack);
        }
    }
});
}

private boolean isBlock(long endTime) {
    return endTime - mStartTimestamp > mBlockThresholdMillis;
}

}

public class StackSampler {
    public static final String SEPARATOR = "\r\n";
    public static final SimpleDateFormat TIME_FORMATTER =
        new SimpleDateFormat("MM-dd HH:mm:ss.SSS");

    private Handler mHandler;
    private Map<Long, String> mStackMap = new LinkedHashMap<>();
    private int mMaxCount = 100;
    private long mSampleInterval;
    //是否需要采样
    protected AtomicBoolean mShouldSample = new AtomicBoolean(false);

    public StackSampler(long sampleInterval) {
        mSampleInterval = sampleInterval;
        HandlerThread handlerThread = new HandlerThread("block-canary-sampler");
        handlerThread.start();
        mHandler = new Handler(handlerThread.getLooper());
    }

    /**
     * 开始采样 执行堆栈
     */
    public void startDump() {
        //避免重复开始
        if (mShouldSample.get()) {
            return;
        }
        mShouldSample.set(true);

        mHandler.removeCallbacks(mRunnable);
        mHandler.postDelayed(mRunnable, mSampleInterval);
    }

    public void stopDump() {
        if (!mShouldSample.get()) {
            return;
        }
    }
}

```

```

    }
    mShouldSample.set(false);

    mHandler.removeCallbacks(mRunnable);
}

public List<String> getStacks(long startTime, long endTime) {
    ArrayList<String> result = new ArrayList<>();
    synchronized (mStackMap) {
        for (Long entryTime : mStackMap.keySet()) {
            if (startTime < entryTime && entryTime < endTime) {
                result.add(TIME_FORMATTER.format(entryTime)
                    + SEPARATOR
                    + SEPARATOR
                    + mStackMap.get(entryTime));
            }
        }
    }
    return result;
}

private Runnable mRunnable = new Runnable() {
    @Override
    public void run() {
        StringBuilder sb = new StringBuilder();
        StackTraceElement[] stackTrace = Looper.getMainLooper().getThread().getStackTrace();
        for (StackTraceElement s : stackTrace) {
            sb.append(s.toString()).append("\n");
        }
        synchronized (mStackMap) {
            //最多保存100条堆栈信息
            if (mStackMap.size() == mMaxCount) {
                mStackMap.remove(mStackMap.keySet().iterator().next());
            }
            mStackMap.put(System.currentTimeMillis(), sb.toString());
        }

        if (mShouldSample.get()) {
            mHandler.postDelayed(mRunnable, mSampleInterval);
        }
    }
};
}

```

其实这种方式也就是 `BlockCanary` 原理。

Choreographer.FrameCallback

Android系统每隔16ms发出VSYNC信号，来通知界面进行重绘、渲染，每一次同步的周期约为16.6ms，代表一帧的刷新频率。通过Choreographer类设置它的FrameCallback函数，当每一帧被渲染时会触发回调

FrameCallback.doFrame (long frameTimeNanos) 函数。frameTimeNanos是底层VSYNC信号到达的时间戳。

```
public class ChoreographerHelper {
    public static void start() {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
            Choreographer.getInstance().postFrameCallback(new Choreographer.FrameCallback() {
                long lastFrameTimeNanos = 0;

                @Override
                public void doFrame(long frameTimeNanos) {
                    //上次回调时间
                    if (lastFrameTimeNanos == 0) {
                        lastFrameTimeNanos = frameTimeNanos;
                        Choreographer.getInstance().postFrameCallback(this);
                        return;
                    }
                    long diff = (frameTimeNanos - lastFrameTimeNanos) / 1_000_000;
                    if (diff > 16.6f) {
                        //掉帧数
                        int droppedCount = (int) (diff / 16.6);
                    }
                    lastFrameTimeNanos = frameTimeNanos;
                    Choreographer.getInstance().postFrameCallback(this);
                }
            });
        }
    }
}
```

通过 ChoreographerHelper 可以实时计算帧率和掉帧数，实时监测App页面的帧率数据，发现帧率过低，还可以自动保存现场堆栈信息。

Looper比较适合在发布前进行测试或者小范围灰度测试然后定位问题，ChoreographerHelper适合监控线上环境的 app 的掉帧情况来计算 app 在某些场景的流畅度然后有针对性的做性能优化。

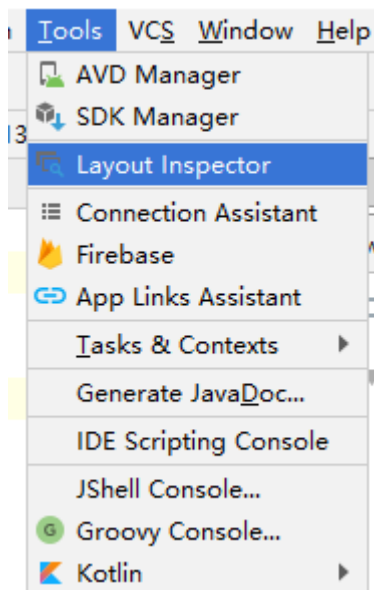
布局优化

层级优化

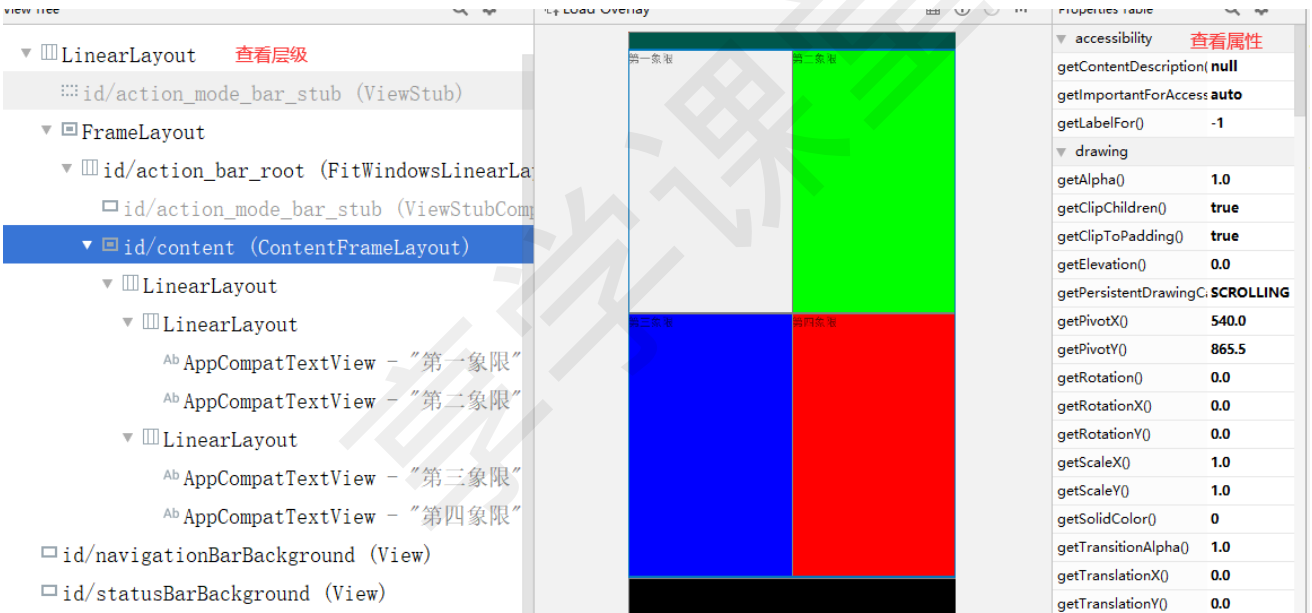
measure、layout、draw这三个过程都包含自顶向下的View Tree遍历耗时，如果视图层级太深自然需要更多的时间来完成整个绘制过程，从而造成启动速度慢、卡顿等问题。而onDraw在频繁刷新时可能多次出发，因此onDraw更不能做耗时操作，同时需要注意内存抖动。对于布局性能的检测，依然可以使用systrace与traceview按照绘制流程检查绘制耗时函数。

Layout Inspector

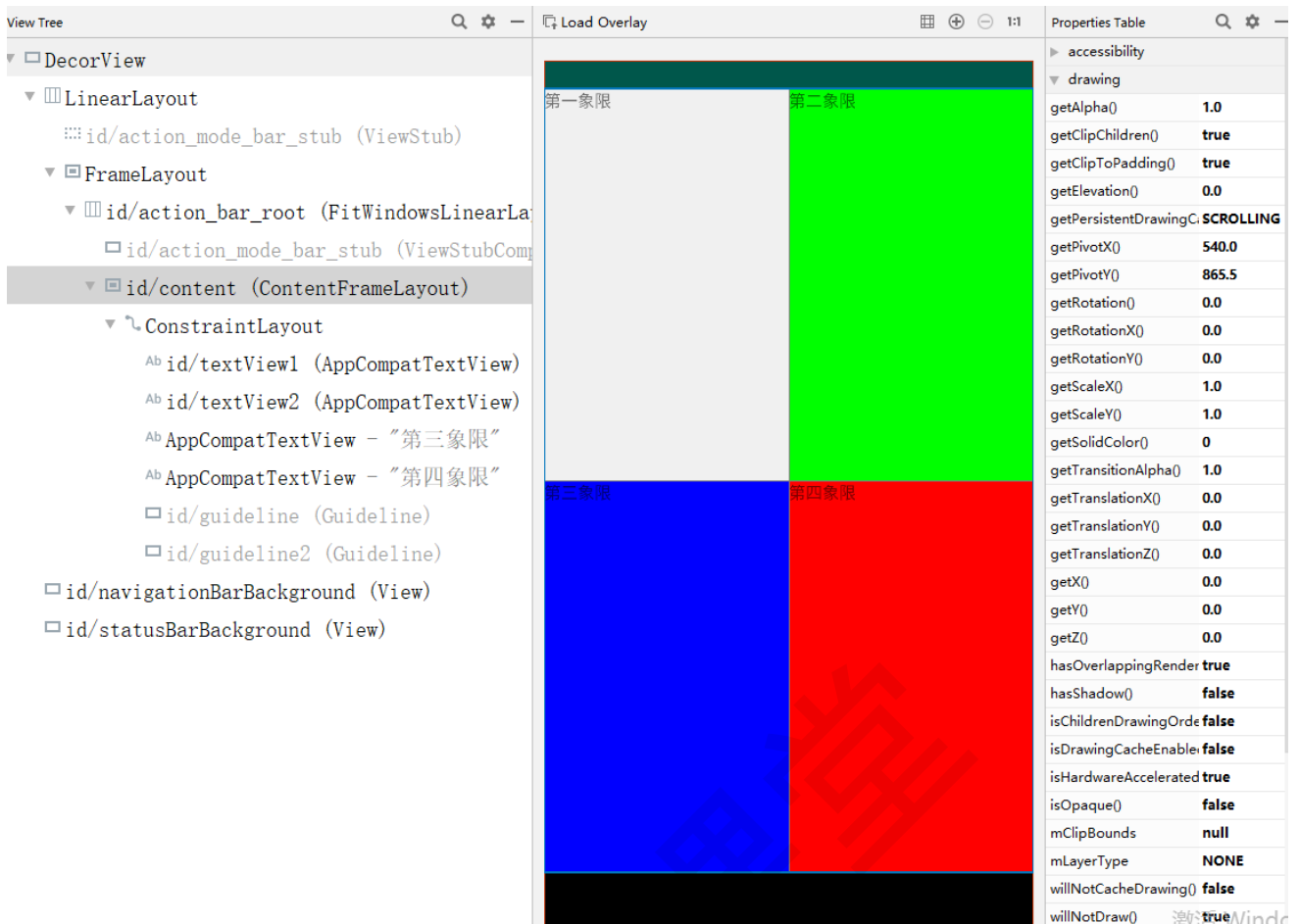
在较早的时代SDK中有一个**hierarchy viewer** 工具，但是早在 Android Studio 3.1 配套的SDK中（具体SDK版本不记得了）就已经被弃用。现在应在运行时改用 **Layout Inspector**来检查应用的视图层次结构。



然后选择需要查看的进程与Activity：



在左侧id为content之下的就是我们写在XML中的布局。可以明显看出，我们的布局中是一个 **LinearLayout** ,其中又包含两个 **LinearLayout** 。我们应该尽量减少其层级，可以使用 **ConstraintLayout** 约束布局使得布局尽量扁平化，移除非必需的UI组件。



使用merge标签

当我们有一些布局元素需要被多处使用时，这时候我们会考虑将其抽取成一个单独的布局文件。在需要使用通过 `include` 加载。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#000000"
    android:orientation="vertical">
    <!-- include layout_merge布局 -->
    <include layout="@layout/layout_merge" />
</LinearLayout>

<!-- layout_merge -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:background="#ffffff"
```

```

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="测试merge" />

</LinearLayout>

```

这时候我们的主布局文件是垂直的LinearLayout，include的"layout_merge"也是垂直的LinearLayout，这时候include的布局中使用的LinearLayout就没意义了，使用的话反而减慢你的UI表现。这时可以使用merge标签优化。

```

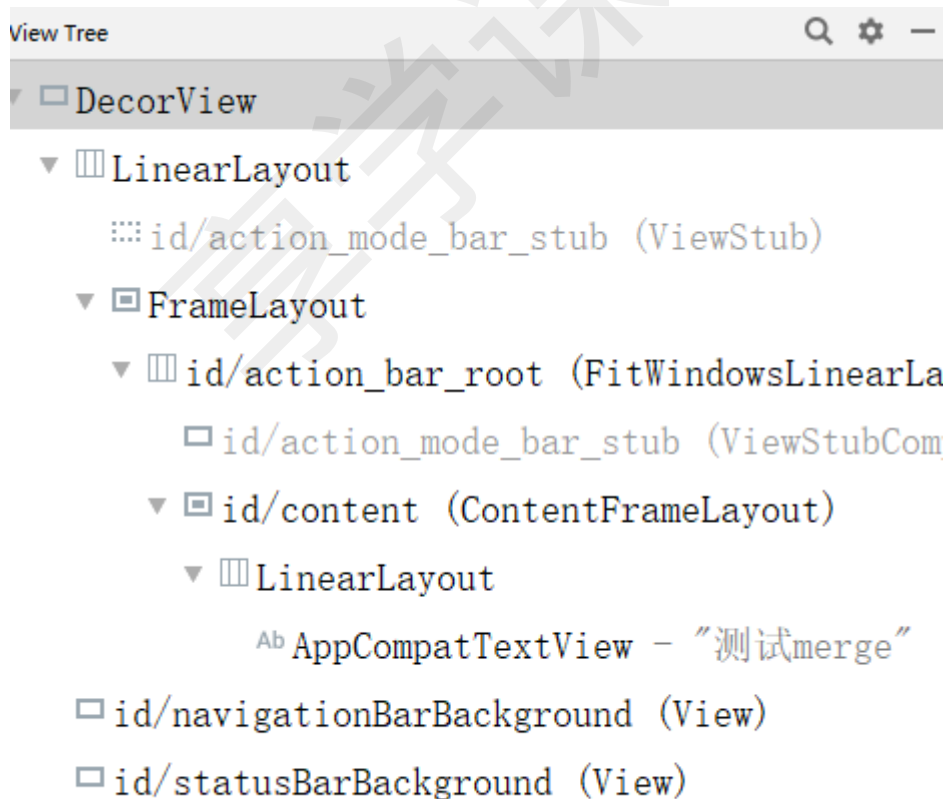
<!-- layout_merge -->
<merge xmlns:android="http://schemas.android.com/apk/res/android">

    <TextView
        android:background="#ffffff"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="测试merge" />

</merge>

```

修改为merge后，通过LayoutInspector能够发现，include的布局中TextView直接被加入到父布局中。



使用ViewStub 标签

当我们布局中存在一个View/ViewGroup，在某个特定时刻才需要他的展示时，可能会有同学把这个元素在xml中定义为invisible或者gone，在需要显示时再设置为visible可见。比如在登陆时，如果密码错误在密码输入框上显示提示。

invisible

view设置为invisible时，view在layout布局文件中会占用位置，但是view为不可见，该view还是会创建对象，会被初始化，会占用资源。

gone

view设置gone时，view在layout布局文件中不占用位置，但是该view还是会创建对象，会被初始化，会占用资源。

如果view不一定会显示，此时可以使用 *ViewStub* 来包裹此View 以避免不需要显示view但是又需要加载view消耗资源。

viewstub是一个轻量级的view，它不可见，不用占用资源，只有设置viewstub为visible或者调用其inflater()方法时，其对应的布局文件才会被初始化。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#000000"
    android:orientation="vertical">

    <ViewStub
        android:id="@+id/viewStub"
        android:layout_width="600dp"
        android:layout_height="500dp"
        android:inflatedId="@+id/textView"
        android:layout="@layout/layout_viewstub" />

</LinearLayout>

<!-- layout_viewstub -->
<?xml version="1.0" encoding="utf-8"?>

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="#ffffff"
    android:text="测试viewStub" />
```

加载viewStub后，可以通过 `inflatedId` 找到 `layout_viewstub` 中的根View。

过度渲染

过度绘制是指系统在渲染单个帧的过程中多次在屏幕上绘制某一个像素。例如，如果有若干界面卡片堆叠在一起，每张卡片都会遮盖其下面一张卡片的部分内容。但是，系统仍然需要绘制堆叠中的卡片被遮盖的部分。


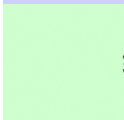
GPU 过度绘制检查


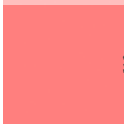
手机开发者选项中能够显示过度渲染检查功能，通过对界面进行彩色编码来帮我们识别过度绘制。开启步骤如下：

1. 进入**开发者选项** (Developer Options)。
2. 找到**调试 GPU 过度绘制**(Debug GPU overdraw)。
3. 在弹出的对话框中，选择**显示过度绘制区域** (Show overdraw areas) 。



Android 将按如下方式为界面元素着色，以确定过度绘制的次数：

- **真彩色**：没有过度绘制
-  **蓝色**：过度绘制 1 次
-  **绿色**：过度绘制 2 次

-  粉色：过度绘制 3 次
-  红色：过度绘制 4 次或更多次

请注意，这些颜色是半透明的，因此您在屏幕上看到的确切颜色取决于界面内容。

有些过度绘制是不可避免的。在优化应用的界面时，应尝试达到大部分显示真彩色或仅有 1 次过度绘制（蓝色）的视觉效果。

解决过度绘制问题

可以采取以下几种策略来减少甚至消除过度绘制：

- 移除布局中不需要的背景。
 - 默认情况下，布局没有背景，这表示布局本身不会直接渲染任何内容。但是，当布局具有背景时，其可能会导致过度绘制。

移除不必要的背景可以快速提高渲染性能。不必要的背景可能永远不可见，因为它会被应用在该视图上绘制的任何其他内容完全覆盖。例如，当系统在父视图上绘制子视图时，可能会完全覆盖父视图的背景。
- 使视图层次结构扁平化。
 - 可以通过优化视图层次结构来减少重叠界面对象的数量，从而提高性能。
- 降低透明度。
 - 对于不透明的 `view`，只需要渲染一次即可把它显示出来。但是如果这个 `view` 设置了 `alpha` 值，则至少需要渲染两次。这是因为使用了 `alpha` 的 `view` 需要先知道混合 `view` 的下一层元素是什么，然后再结合上层的 `view` 进行 Blend 混色处理。透明动画、淡入淡出和阴影等效果都涉及到某种透明度，这就会造成了过度绘制。可以通过减少要渲染的透明对象的数量，来改善这些情况下的过度绘制。例如，如需获得灰色文本，可以在 `TextView` 中绘制黑色文本，再为其设置半透明的透明度值。但是，简单地通过用灰色绘制文本也能获得同样的效果，而且能够大幅提升性能。

布局加载优化

异步加载

`LayoutInflater` 加载 xml 布局的过程会在主线程使用 IO 读取 XML 布局文件进行 XML 解析，再根据解析结果利用反射创建布局中的 `View/ViewGroup` 对象。这个过程随着布局的复杂度上升，耗时自然也会随之增大。Android 为我们提供了 `AsyncLayoutInflater` 把耗时的加载操作在异步线程中完成，最后把加载结果再回调给主线程。

```
dependencies {  
    implementation "androidx.asynclayoutinflater:asynclayoutinflater:1.0.0"  
}
```

```
new AsyncLayoutInflater(this)
    .inflate(R.layout.activity_main, null, new AsyncLayoutInflater.OnInflateFinishedListener() {
        @Override
        public void onInflateFinished(@NonNull View view, int resid, @Nullable ViewGroup parent) {
            setContentView(view);
            //.....
        }
    });
```

- 1、使用异步 inflate，那么需要这个 layout 的 parent 的 generateLayoutParams 函数是线程安全的；
- 2、所有构建的 View 中必须不能创建 Handler 或者是调用 Looper.myLooper；（因为是在异步线程中加载的，异步线程默认没有调用 Looper.prepare）；
- 3、AsyncLayoutInflater 不支持设置 LayoutInflater.Factory 或者 LayoutInflater.Factory2；
- 4、不支持加载包含 Fragment 的 layout
- 5、如果 AsyncLayoutInflater 失败，那么会自动回退到UI线程来加载布局；

掌阅X2C思路

https://github.com/iReaderAndroid/X2C/blob/master/README_CN.md