# 深入探索Android布局优化（下）



转载自：**深入探索Android布局优化（下）**

## 前言

**成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。**

本篇，为深入探索Android布局优化的下篇。这篇文章包含的主要内容如下所示：

- 6、布局优化常规方案
- 7、布局优化的进阶方案
- 8、布局优化的常见问题

下面，笔者将与大家一起进入进行布局优化的实操环节。

## 六、布局优化常规方案

布局优化的方法有很多，大部分主流的方案笔者已经在Android性能优化之绘制优化里讲解过了。下面，我将介绍一些其它的优化方案。

### 1、布局Inflate优化方案演进

#### 1、代码动态创建View

使用Java代码动态添加控件的简单示例如下：

```
Button button=new Button(this);
button.setBackgroundColor(Color.RED);
button.setText("Hello World");
ViewGroup viewGroup = (ViewGroup) LayoutInflater.from(this).inflate
(R.layout.activity_main, null);
viewGroup.addView(button);
```

#### 2、替换MessageQueue来实现异步创建View

在使用子线程创建视图控件的时候，我们可以把子线程Looper的MessageQueue替换成主线程的MessageQueue，在创建完需要的视图控件后记得将子线程Looper中的MessageQueue恢复为原来的。在Awesome-WanAndroid项目下的UiUtils的Ui优化工具类中，提供了相应的实现，代码如下所示：

```
/**
 * 实现将子线程Looper中的MessageQueue替换为主线程中Looper的
 * MessageQueue，这样就能够在子线程中异步创建UI。
 *
 * 注意：需要在子线程中调用。
 *
 * @param reset 是否将子线程中的MessageQueue重置为原来的，false则表示需要进行
替换
 * @return 替换是否成功
 */
public static boolean replaceLooperWithMainThreadQueue(boolean rese
t) {
    if (CommonUtils.isMainThread()) {
        return true;
    } else {
        // 1、获取子线程的ThreadLocal实例
        ThreadLocal<Looper> threadLocal = ReflectUtils.reflect(Loope
r.class).field("sThreadLocal").get();
        if (threadLocal == null) {
```

```
            return false;
        } else {
            Looper looper = null;
            if (!reset) {
                Looper.prepare();
                looper = Looper.myLooper();
                // 2、通过调用MainLooper的getQueue方法区获取主线程Looper中的MessageQueue实例
                Object queue = ReflectUtils.reflect(Looper.getMainLooper()).method("getQueue").get();
                if (!(queue instanceof MessageQueue)) {
                    return false;
                }
                // 3、将子线程中的MessageQueue字段的值设置为主线的MessageQueue实例
                ReflectUtils.reflect(looper).field("mQueue", queue);
            }
            // 4、reset为false，表示需要将子线程Looper中的MessageQueue重置为原来的。
            ReflectUtils.reflect(threadLocal).method("set", looper);
            return true;
        }
    }
}
```

### 3、AsynclayoutInflater异步创建View

在第三小节中，我们对Android的布局加载原理进行了深入地分析，从中我们得出了布局加载过程中的两个耗时点：

- 1、布局文件读取慢：IO过程。
- 2、创建View慢：使用反射，比直接new的方式要慢3倍。布局嵌套层级越多，控件个数越多，反射的次数就会越频繁。

很明显，我们无法从根本上去解决这两个问题，但是Google提供了一个从侧面解决的方案:使用AsyncLayoutInflater去异步加载对应的布局，它的特点如下：

- 1、工作线程加载布局。
- 2、回调主线程。
- 3、节省主线程时间。

接下来，我将详细地介绍AsynclayoutInflater的使用。

首先，在项目的build.gradle中进行配置：

```
implementation 'com.android.support:asynclayoutinflater:28.0.0'
```

然后，在Activity中的onCreate方法中将setContentView注释：

```
super.onCreate(savedInstanceState);
// 内部分别使用了IO和反射的方式去加载布局解析器和创建对应的View
// setContentView(R.layout.activity_main);
```

接着，在super.onCreate方法前继续布局的异步加载：

```
// 使用AsyncLayoutInflater进行布局的加载
new AsyncLayoutInflater(MainActivity.this).inflate(R.layout.activity_main, null, new AsyncLayoutInflater.OnInflateFinishedListener() {
        @Override
        public void onInflateFinished(@NonNull View view, int i, @Nullable ViewGroup viewGroup) {
            setContentView(view);
            // findViewById、视图操作等
    }
});
super.onCreate(savedInstanceState);
```

接下来，我们来分析下AsyncLayoutInflater的实现原理与工作流程。
由于我们是使用new的方式创建的AsyncLayoutInflater，所以我们先来看看它的构造函数：

```
public AsyncLayoutInflater(@NonNull Context context) {
    // 1
    this.mInflater = new AsyncLayoutInflater.BasicInflater(context);
    // 2
    this.mHandler = new Handler(this.mHandlerCallback);
    // 3
    this.mInflateThread = AsyncLayoutInflater.InflateThread.getInstance();
}
```

在注释1处，创建了一个BasicInflater，它内部的onCreateView并没有使用Factory做AppCompat控件兼容的处理：

```
protected View onCreateView(String name, AttributeSet attrs) throws ClassNotFoundException {
    String[] var3 = sClassPrefixList;
    int var4 = var3.length;
    for(int var5 = 0; var5 < var4; ++var5) {
```

```
        String prefix = var3[var5];
        try {
            View view = this.createView(name, prefix, attrs);
            if (view != null) {
                return view;
            }
        } catch (ClassNotFoundException var8) {
        }
    }
    return super.onCreateView(name, attrs);
}
```

由前面的分析可知，在createView方法中仅仅是做了反射创建出对应View的处理。

接着，在注释2处，创建了一个全局的Handler对象，主要是用于将异步线程创建好的View实例及其相关信息回调到主线程。

最后，在注释3处，获取了一个用于异步加载View的线程实例。

接着，我们继续跟踪AsyncLayoutInflater实例的inflate方法：

```
@UiThread
public void inflate(@LayoutRes int resid, @Nullable ViewGroup parent, @NonNull AsyncLayoutInflater.OnInflateFinishedListener callback) {
    if (callback == null) {
        throw new NullPointerException("callback argument may not be null!");
    } else {
        // 1
        AsyncLayoutInflater.InflateRequest request = this.mInflateThread.obtainRequest();
        request.inflater = this;
        request.resid = resid;
        request.parent = parent;
        request.callback = callback;
        this.mInflateThread.enqueue(request);
    }
}
```

在注释1处，这里使用InflateRequest对象将我们传进来的三个参数进行了包装，并最终将这个InflateRequest对象加入了mInflateThread线程中的一个ArrayBlockingQueue中：

```
public void enqueue(AsyncLayoutInflater.InflateRequest request) {
```

```
    try {
        this.mQueue.put(request);
      } catch (InterruptedException var3) {
        throw new RuntimeException("Failed to enqueue async inflate request", var3);
    }
}
```

并且，在InflateThread这个静态内部类的静态代码块中调用了其自身实例的start方法以启动线程：

```
static {
    sInstance.start();
}
public void run() {
    while(true) {
        this.runInner();
    }
}
public void runInner() {
    AsyncLayoutInflater.InflateRequest request;
    try {
        // 1
        request = (AsyncLayoutInflater.InflateRequest)this.mQueue.take();
    } catch (InterruptedException var4) {
        Log.w("AsyncLayoutInflater", var4);
        return;
    }
    try {
        // 2
        request.view = request.inflater.mInflater.inflate(request.resid, request.parent, false);
    } catch (RuntimeException var3) {
        Log.w("AsyncLayoutInflater", "Failed to inflate resource in the background! Retrying on the UI thread", var3);
    }
    // 3
    Message.obtain(request.inflater.mHandler, 0, request).sendToTarget();
}
```

在run方法中，使用了死循环的方式去不断地调用runInner方法，在runInner方法中，首先在注释1处从ArrayBlockingQueue队列中获取一个InflateRequest对象，然后在注释2处将异步加载好的view对象赋值给了InflateRequest对象，最后，在注释3处，将请求作为消息发送给了Handler的handleMessage：

```
1 private Callback mHandlerCallback = new Callback() {
2     public boolean handleMessage(Message msg) {
3         AsyncLayoutInflater.InflateRequest request = (AsyncLayoutInf
  later.InflateRequest)msg.obj;
4         // 1
5         if (request.view == null) {
6             request.view = AsyncLayoutInflater.this.mInflater.inflat
  e(request.resid, request.parent, false);
7         }
8         request.callback.onInflateFinished(request.view, request.res
  id, request.parent);
9         AsyncLayoutInflater.this.mInflateThread.releaseRequest(reque
  st);
10         return true;
11     }
12 };
```

在handleMessage方法中，当异步加载得到的view为null时，此时在注释1处还做了一个fallback处理，直接在主线程进行view的加载，以此兼容某些异常情况，最后，就调用了回调接口的onInflateFinished方法将view的相关信息返回给Activity对象。

### 小结

由以上分析可知，AsyncLayoutInflater是通过侧面缓解的方式去缓解布局加载过程中的卡顿，但是它依然存在一些问题：

- 1、不能设置LayoutInflater.Factory，通过自定义AsyncLayoutInflater的方式解决，由于它是一个final，所以需要将代码直接拷处进行修改。
- 2、因为是异步加载，所以需要注意在布局加载过程中不能有依赖于主线程的操作。

由于AsyncLayoutInflater仅仅只能通过侧面缓解的方式去缓解布局加载的卡顿，因此，我们下面将介绍一种从根本上解决问题的方案。对于AsynclayoutInflater的改进措施，可以查看祁同伟同学封装之后的代码：

Android AsyncLayoutInflater 限制及改进

#### 4、使用X2C进行布局加载优化

由上分析可知，在布局加载的过程中有两个主要的耗时点，即IO操作和反射，而AsyncLayoutInflater仅仅是缓解，那么有什么方案能从根本上去解决这个问题呢？

### 使用Java代码写布局？

如果使用Java代码写布局，无疑从Xml文件进行IO操作的过程和反射获取View实例的过程都将被抹去。虽然这样从本质上解决了问题，但是也引入了一些新问题，如不便于开发，可维护性差等等。
那么，还有没有别的更好的方式呢?
答案就是X2C。

## X2C

X2C项目地址
X2C框架保留了XML的优点，并解决了其IO操作和反射的性能问题。开发人员只需要正常写XML代码即可，在编译期，X2C会利用APT工具将XML代码翻译为Java代码。
接下来，我们来进行X2C的使用。
首先，在app的build.gradle文件添加如下依赖：

```
annotationProcessor 'com.zhangyue.we:x2c-apt:1.1.2'
implementation 'com.zhangyue.we:x2c-lib:1.0.6'
```

然后，在对应的MainActivity类上方添加如下注解，让MainActivity知道我们使用的布局是activity_main：

```
@Xml(layouts = "activity_main")
public class MainActivity extends AppCompatActivity implements OnFeed
ShowCallBack {
```

接着，将onCreate方法中setContentView的原始方式改为X2C的设置方式：

```
X2C.setContentView(MainActivity.this, R.layout.activity_main);
```

最后，我们再Rebuild项目，会在build下的generated->source->apt->debug->com.zhangyue.we.x2c下自动生成X2C127_activity_main这个类：

```
public class X2C127_activity_main implements IViewCreator {
@Override
public View createView(Context context) {
    return new com.zhangyue.we.x2c.layouts.X2C127_Activity_Main().cre
ateView(context);
    }
}
```

在这个类中又继续调用了layout目录下的X2C127_Activity_Main实例的createView方法，如下所示：

```
public class X2C127_Activity_Main implements IViewCreator {
    @Override
    public View createView(Context ctx) {
        Resources res = ctx.getResources();
```

```
        ConstraintLayout constraintLayout0 = new ConstraintLayout(ctx);
        RecyclerView recyclerView1 = new RecyclerView(ctx);
        ConstraintLayout.LayoutParams layoutParam1 = new ConstraintLayout.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,ViewGroup.LayoutParams.MATCH_PARENT);
        recyclerView1.setId(R.id.recycler_view);
        recyclerView1.setLayoutParams(layoutParam1);
        constraintLayout0.addView(recyclerView1);
        return constraintLayout0;
    }
}
```

从上可知，里面采用了new的方式创建了相应的控件，并设置了对应的信息。

接下来，我们回到X2C.setContentView(MainActivity.this, R.layout.activity_main)这个方法，看看它内部究竟做了什么处理：

```
/**
 * 设置contentview，检测如果有对应的java文件，使用java文件，否则使用xml
 *
 * @param activity 上下文
 * @param layoutId layout的资源id
 */
public static void setContentView(Activity activity, int layoutId) {
    if (activity == null) {
        throw new IllegalArgumentException("Activity must not be null");
    }
    // 1
    View view = getView(activity, layoutId);
    if (view != null) {
        activity.setContentView(view);
    } else {
        activity.setContentView(layoutId);
    }
}
```

在注释1处，通过getView方法获取到了对应的view，我们继续跟踪进去：

```
public static View getView(Context context, int layoutId) {
    IViewCreator creator = sSparseArray.get(layoutId);
```

```
    if (creator == null) {
        try {
            int group = generateGroupId(layoutId);
            String layoutName = context.getResources().getResourceName(layoutId);
            layoutName = layoutName.substring(layoutName.lastIndexOf("/") + 1);
            String clzName = "com.zhangyue.we.x2c.X2C" + group + "_" + layoutName;
            // 1
            creator = (IViewCreator) context.getClassLoader().loadClass(clzName).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        //如果creator为空，放一个默认进去，防止每次都调用反射方法耗时
        if (creator == null) {
            creator = new DefaultCreator();
        }
        sSparseArray.put(layoutId, creator);
    }
    // 2
    return creator.createView(context);
}
```

可以看到，这里采用了一个sSparseArray集合去存储布局对应的视图创建对象creator，如果是首次创建creator的话，会在注释1处使用反射的方式去加载处对应的creator对象，然后将它放入sSparseArray集中，最后在注释2处调用了creator的createView方法去使用new的方式去创建对应的控件。

但是，X2C框架还存在一些问题：

- 1、部分Java属性不支持。
- 2、失去了系统的兼容（AppCompat）

对于第2个问题，我们需要修改X2C框架的源码，当发现是TextView等控件时，需要直接使用new的方式去创建一个AppCompatTextView等兼容类型的控件。于此同时，它还有如下两个小的点不支持，但是这个问题不大：

- merge标签 ,在编译期间无法确定xml的parent，所以无法支持。
- 系统style,在编译期间只能查到应用的style列表，无法查询系统style，所以只支持应用内style。

### 2、使用ConstraintLayout降低布局嵌套层级

首先，对于Android视图绘制的原理，我们必须要有一定的了解，关于这块，大家可以参考下Android View的绘制流程 这篇文章。

对于视图绘制的性能瓶颈，大概有以下三点：

- 1、测量、布局、绘制每个阶段的耗时。
- 2、自顶而下的遍历，当嵌套层级过多时，遍历耗时会比较明显。
- 3、无效的嵌套布局或不合理使用RelativeLayout可能会导致触发多次绘制。

那么，如何减少布局的层级及复杂度呢?

基本上只要遵循以下两点即可：

- 1、减少View树层级。
- 2、宽而浅，避免窄而深。

为了提升布局的绘制速度，Google推出了ConstraintLayout，它的特点如下：

- 1、实现几乎完全扁平化的布局。
- 2、构建复杂布局性能更高。
- 3、具有RelativeLayout和LinearLayout的特性。

接下来，我们来简单使用一下ConstraintLayout来优化一下我们的布局。

首先，下面是我们的原始布局:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/andr
oid"
    android:id="@+id/ll_out"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="5dp">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
        <com.facebook.drawee.view.SimpleDraweeView
            android:id="@+id/iv_news"
            android:layout_width="80dp"
            android:layout_height="80dp"
            android:scaleType="fitXY" />
        <TextView
            android:id="@+id/tv_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:padding="10dp"
            android:textSize="20dp" />
    </LinearLayout>
    <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:padding="3dp"
        android:text="来自NBA官网"
        android:textSize="14dp" />
</LinearLayout>
```

可以看到,它具有三层嵌套结构,然后我们来使用ConstraintLayout来优化一下这个布局:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/ll_out"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="5dp">
    <com.facebook.drawee.view.SimpleDraweeView
        android:id="@+id/iv_news"
        android:layout_width="80dp"
        android:layout_height="80dp"
        android:scaleType="fitXY"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <TextView
        android:id="@+id/tv_title"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:paddingLeft="10dp"
        android:textSize="20dp"
        app:layout_constraintLeft_toRightOf="@id/iv_news"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="@id/iv_news" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="3dp"
        android:text="来自NBA官网"
```

```
        android:textSize="14dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@id/iv_news" />
</android.support.constraint.ConstraintLayout>
```

经过ConstraintLayout之后，布局的嵌套层级变为了2级，如果布局比较复杂，比如有5，6，7层嵌套层级，使用Contraintlayout之后降低的层级会更加明显。对于其app下的一系列属性，其实都非常简单，这里就不多做介绍了。

除此之外，还有以下方式可以减少布局层级和复杂度：

- 1、不嵌套使用RelativeLayout。
- 2、不在嵌套LinearLayout中使用weight。
- 3、使用merge标签，它能够减少一个层级，但只能用于根View。

### 3、过渡绘制优化

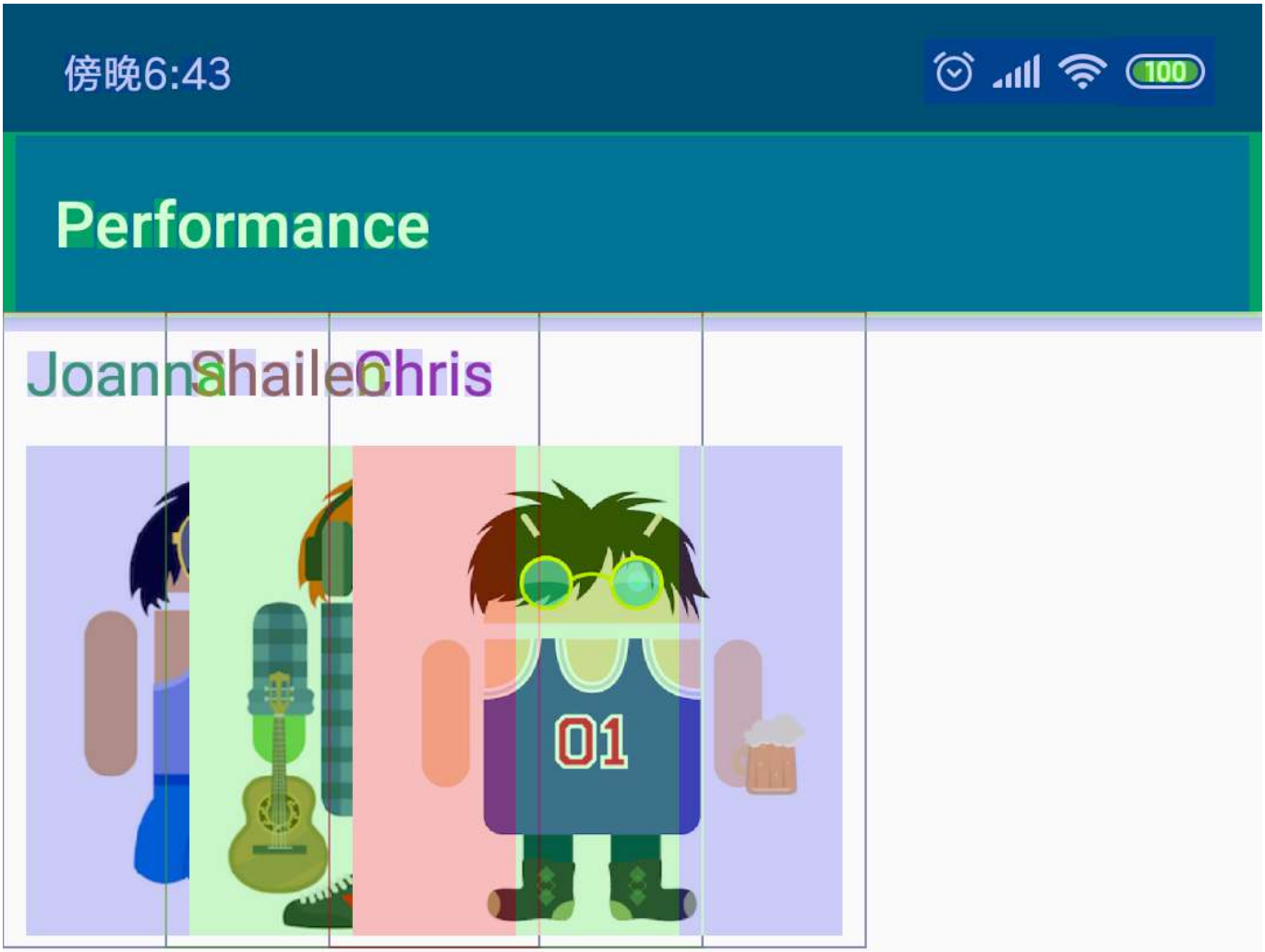
在视图的绘制优化中，还有一个比较重要的优化点，就是避免过渡绘制，这个笔者已经在Android性能优化之绘制优化一文的第四小节详细分析过了。最后这里补充一下自定义View中使用clipRect的一个实例。

首先，我们自定义了一个DroidCardsView，他可以存放多个叠加的卡片，onDraw方法的实现如下：

```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // Don't draw anything until all the Asynctasks are done and all the DroidCards are ready.
    if (mDroids.length > 0 && mDroidCards.size() == mDroids.length) {
        // Loop over all the droids, except the last one.
        int i;
        for (i = 0; i < mDroidCards.size() - 1; i++) {
            mCardLeft = i * mCardSpacing;
            // Draw the card. Only the parts of the card that lie within the bounds defined by
            // the clipRect() get drawn.
            drawDroidCard(canvas, mDroidCards.get(i), mCardLeft, 0);
        }
        // Draw the final card. This one doesn't get clipped.
        drawDroidCard(canvas, mDroidCards.get(mDroidCards.size() - 1),
                mCardLeft + mCardSpacing, 0);
    }
    // Invalidate the whole view. Doing this calls onDraw() if the view is visible.
```

```
18      invalidate();
19  }
```

从以上代码可知，这里是直接进行绘制的，此时显示的布局过渡绘制背景如下所示：



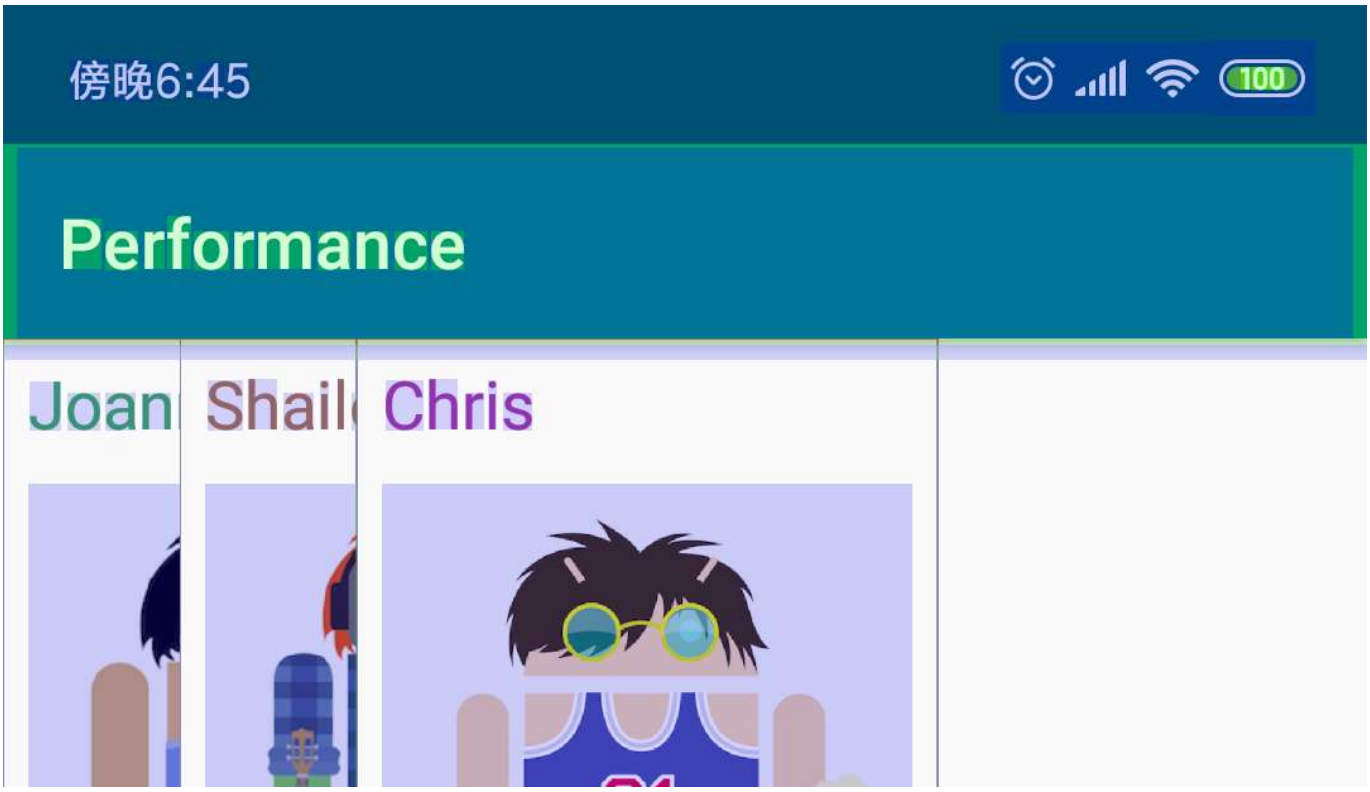
可以看到，图片的背景都叠加起来了，这个时候，我们需要在绘制的时候使用clipRect让系统去识别可绘制的区域，因此我们在自定义的DroidCardsView的onDraw方法去使用clipRect：

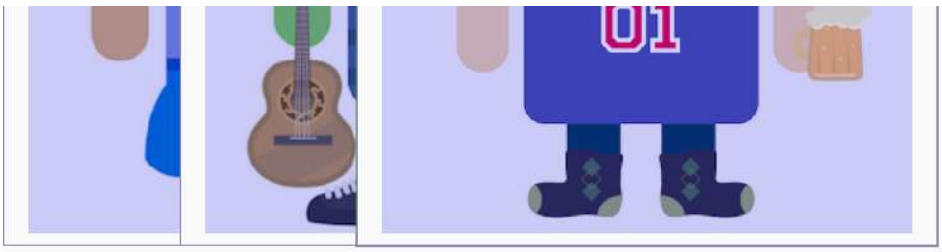
```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // Don't draw anything until all the Asynctasks are done and all the DroidCards are ready.
    if (mDroids.length > 0 && mDroidCards.size() == mDroids.length) {
        // Loop over all the droids, except the last one.
        int i;
        for (i = 0; i < mDroidCards.size() - 1; i++) {
            mCardLeft = i * mCardSpacing;
            // 1、clipRect方法和绘制前后成对使用canvas的save方法与restore方法。
            canvas.save();
```

```
            // 2、使用clipRect指定绘制区域，这里的mCardSpacing是指的相邻卡片最左边的间距，需要在动态创建DroidCardsView的时候传入。
            canvas.clipRect(mCardLeft,0,mCardLeft+mCardSpacing,mDroidCards.get(i).getHeight());
            // 3、Draw the card. Only the parts of the card that lie within the bounds defined by
            // the clipRect() get drawn.
            drawDroidCard(canvas, mDroidCards.get(i), mCardLeft, 0);
            canvas.restore();
        }
        // Draw the final card. This one doesn't get clipped.
        drawDroidCard(canvas, mDroidCards.get(mDroidCards.size() - 1),
                mCardLeft + mCardSpacing, 0);
    }
    // Invalidate the whole view. Doing this calls onDraw() if the view is visible.
    invalidate();
}
```

在注释1处，首先需要在clipRect方法和绘制前后成对使用canvas的save方法与restore方法用来对画布进行操作。接着，在注释2处，使用clipRect指定绘制区域，这里的mCardSpacing是指的相邻卡片最左边的间距，需要在动态创建DroidCardsView的时候传入。最后，在注释3处调用实际绘制卡片的方法。

使用clipRect优化过后的布局过渡绘制背景如下所示：




01

**注意：**

我们还可以通过canvas.quickReject方法来判断是否没和某个矩形相交，以跳过非矩形区域的绘制操作。

当然，对视图的绘制优化还有其它的一些优化操作，比如：

- 1、使用ViewStub、Merge，ViewStub是一种高效占位符，用于延迟初始化。
- 2、onDraw中避免创建大对象，进行耗时操作。
- 3、TextView的优化，比如利用它的drawableLeft属性。此外，也可以使用Android 9.0之后的PrecomputedText，它将文件的measure与layout过程进行了异步化。但是需要注意，如果要显示的文本比较少，反而会造成不必要的Scheduling delay，建议文本字符大于200时才使用，并记得使用其兼容类PrecomputedTextCompat，它在9.0以上使用PrecomputedText进行优化，在5.0~9.0使用StaticLayout进行优化。具体调用代码如下所示：

```
1 Future<PrecomputedTextCompat> future = PrecomputedTextCompat.getTextF
  uture(
2                 “text”, textView.getTextMetricsParamsCompat(), null);
3 textView.setTextFuture(future);
```

到这里，笔者就将常规的布局优化讲解完了，是不是顿时感觉实力大增呢?

如果你此时内心已经YY到这种程度，那我只能说：

下面，笔者就来和大家一起来探索布局优化的进阶方案。

# 七、布局优化的进阶方案

## 1、使用异步布局框架Litho

Litho是Facebook开源的一款在Android上高效建立UI的声明式框架，它具有以下特点：

- 声明式：它使用了声明式的API来定义UI组件。
- 异步布局：它可以提前布局UI，而不会阻塞UI线程。
- 视图扁平化：它使用了Facebook开源的另一款布局引擎Yoga进行布局，以自动减少UI包含的ViewGroup数量。
- 细粒度的回收：可以回收文本或图形等任何组件，并可以在用户界面的任何位置重复使用。
- 内部不仅支持使用View来渲染视图，还可以使用更轻量的Drawable来渲染视图。Litho实现了大量使用Drawable来渲染的基础组件，可以进一步使布局扁平化。

### 简单使用Litho

接下来，我们在项目里面来使用Litho。

1、首先，我们需要配置Litho的相关依赖，如下所示：

```
// 项目下
repositories {
    jcenter()
}
// module下
dependencies {
    // ...
    // Litho
    implementation 'com.facebook.litho:litho-core:0.33.0'
    implementation 'com.facebook.litho:litho-widget:0.33.0'
    annotationProcessor 'com.facebook.litho:litho-processor:0.33.0'
    // SoLoader
    implementation 'com.facebook.soloader:soloader:0.5.1'
    // For integration with Fresco
    implementation 'com.facebook.litho:litho-fresco:0.33.0'
    // For testing
    testImplementation 'com.facebook.litho:litho-testing:0.33.0'
    // Sections （options，用来声明去构建一个list）
    implementation 'com.facebook.litho:litho-sections-core:0.33.0'
    implementation 'com.facebook.litho:litho-sections-widget:0.33.0'
    compileOnly 'com.facebook.litho:litho-sections-annotations:0.33.0'
    annotationProcessor 'com.facebook.litho:litho-sections-processor:0.33.0'
}
```

2、然后，在Application下的onCreate方法中初始化SoLoader：

```
@Override
public void onCreate() {
    super.onCreate();
    SoLoader.init(this, false);
}
```

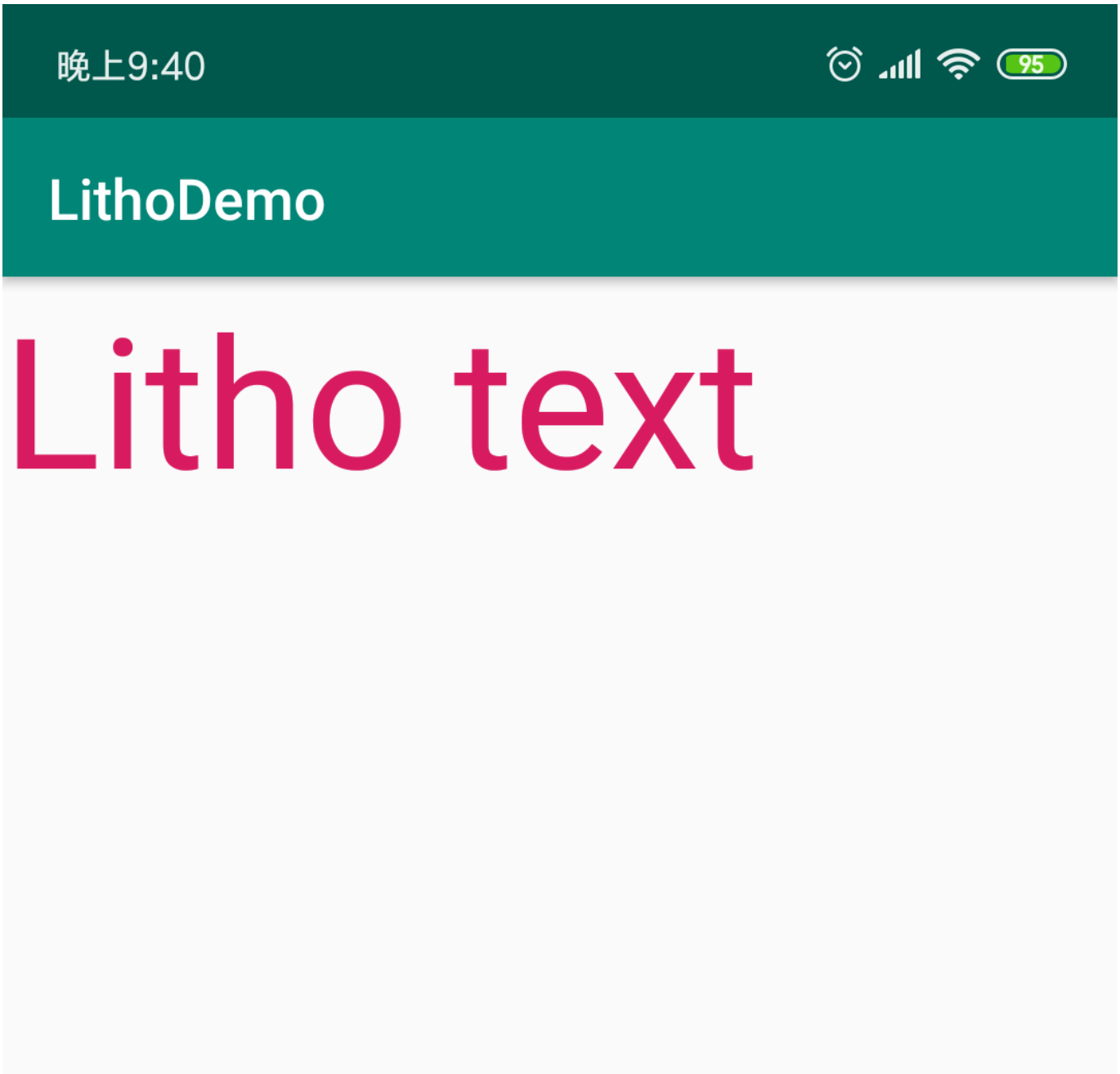
从之前的介绍可知，我们知道Litho使用了Yoga进行布局，而Yoga包含有native依赖，在Soloader.init方法中对这些native依赖进行了加载。

3、最后，在Activity的onCreate方法中添加如下代码即可显示单个的文本视图：

```
// 1、将Activity的Context对象保存到ComponentContext中，并同时初始化
// 一个资源解析者实例ResourceResolver供其余组件使用。
ComponentContext componentContext = new ComponentContext(this);
```

```
// 2、Text内部使用建造者模式以实现组件属性的链式调用，下面设置的text、
// TextColor等属性在Litho中被称为Prop，此概念引申字React。
Text lithoText = Text.create(componentContext)
        .text("Litho text")
        .textSizeDip(64)
        .textColor(ContextCompat.getColor(this, R.color.light_deep_red))
            .build();
// 3、设置一个LithoView去展示Text组件：LithoView.create内部新建了一个
// LithoView实例，并用给定的Component（lithoText）进行初始化
setContentView(LithoView.create(componentContext, lithoText));
```

显示效果如下所示：



在上面的示例中，我们仅仅是将Text这个子组件设置给了LithoView，后续为了实现更复杂的布局，我们需要使用带多个子组件的根组件去替换它。

**使用自定义Component**

由上可知，在Litho中的视图单元叫做Component，即组件，它的设计理念来源于React组件化的思想。每个组件持有描述一个视图单元所必须的属性与状态，用于视图布局的计算工作。视图最终的绘制工作是

由组件指定的绘制单元（View或Drawable）来完成的。接下来，我们使用Litho提供的自定义Component的功能，它能够让我们实现更复杂的Component，这里我们来实现一个类似ListView的列表。
首先，我们先来实现一个ListItem Component，它就如ListView的itemView一样。在下面的实战中，我们将会学习到所有的基础知识，这将会支撑你后续能实现更多更复杂的Component。
然后，在Litho中，我们需要先写一个Spec类去声明Component所对应的布局，在这里需要使用@LayoutSpec规范注解(除此之外，Litho还提供了另一种类型的组件规范：Mount Spec)。代码如下所示：

```
@LayoutSpec
public class ListItemSpec {
    @OnCreateLayout
    static Component onCreateLayout(ComponentContext context) {
        // Column的作用类似于HTML中的<div>标签
        return Column.create(context)
                .paddingDip(YogaEdge.ALL, 16)
                .backgroundColor(Color.WHITE)
                .child(Text.create(context)
                            .text("Litho Study")
                            .textSizeSp(36)
                        .textColor(Color.BLUE)
                            .build())
                .child(Text.create(context)
                            .text("JsonChao")
                            .textSizeSp(24)
                        .textColor(Color.MAGENTA)
                            .build())
                .build();
    }
}
```

然后，框架会使用APT技术去帮助生成对应的ListItem Component 类。最后，我们在Activity的onCreate中将上述第一个例子中的第二步改为如下：

```
// 2、构建ListItem组件
ListItem listItem = ListItem.create(componentContext).build();
```

运行项目，显示界面如下所示：

LithoDemo

# Litho Study

JsonChao

**那上述过程是如何创建与构建的呢?**

它看起来就像有一个LithoSpec的类名，并且在项目构建之后生成了一个与LithoSpec有着同样包名的Litho类，如下所示：

类似于Litho这种类中的所有方法参数都会由Litho进行自动填充。此外，基于这些规格，将会有一些额外的方法由注解处理器自动生成，例如上述示例中Column或Row中的Text的TextSizeSp、backgroundColor等方法。(Row和Column分别对应着Flexox中的行和列，它们都实现了Litho中另一种特殊的组件Layout)

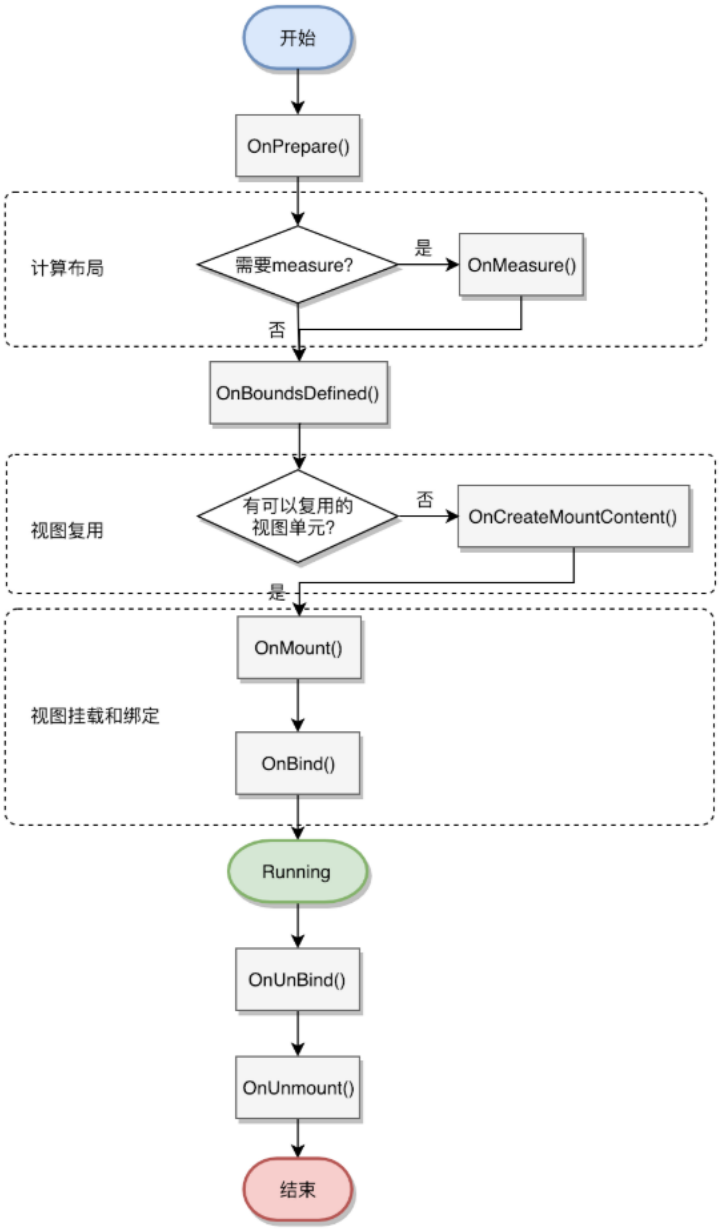
**补充：MountSpec规范**

MountSpec是用来生成可挂载类型组件的一种规范，它的作用是用来生成渲染具体的View或者Drawable的组件。同LayoutSpec类似，它必须使用@MountSpec注解来标注，并实现一个标注了@onCreateMountContent的方法。但是MountSpec的实现要比Layout更加地复杂，因为它拥有自己的生命周期，如下所示：

- @OnPrepare：准备阶段，用于进行一些初始化操作。
- @OnMeasure：负责布局的计算工作。
- @OnBoundsDefined：在布局计算完成之后、挂载视图之前做一些操作。
- @OnCreateMountContent：如果没有可以复用的视图单元，则调用它去创建需要挂载的视图。
- @OnMount：挂载视图，用于完成布局相关的设置。
- @OnBind：绑定视图，用于完成数据和视图的绑定。
- @OnUnBind：解绑视图，与@OnBind相对，主要用于重置视图的数据属性，避免出现数据复用的问题。

- @OnUnmount：卸载视图，与@OnMount相对，主要用于重置视图的布局相关的属性，避免出现布局复用的问题。

MountSpec的生命周期流转图如下所示：




在使用Litho完成了两个实例的开发之后，相信我们已经对Litho的布局方式已经有了一个感性的认知。那么，Litho到底是如何进行布局优化的呢？在布局优化中它所做的核心工作有哪些？

Litho在布局优化中所做的核心工作包括以下三点：

- 1、异步布局化。
- 2、布局自动扁平化。
- 3、更细粒度地优化RecyclerView中组件的缓存与回收的方法。

**1、异步布局化**

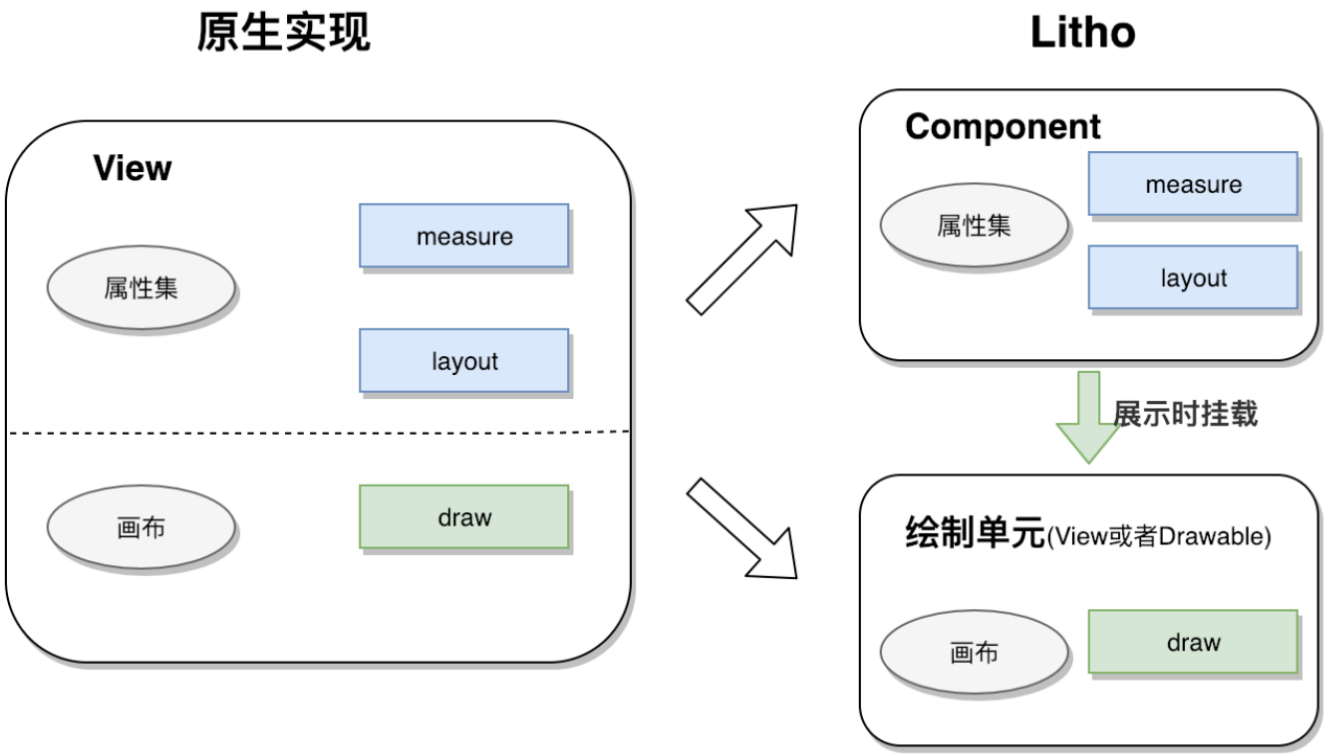
在前文中，我们知道Android的布局加载过程通常会先后涉及到measure、layout、draw过程，并且它们都是在主线程执行的，如果方法执行过程中耗时太多，则主界面必然会产生卡顿现象。

还记得我们在前面介绍的PrecomputedText，它内部将measure与layout的过程放在了异步线程进行初始化，而Litho与PrecomputedText类似，也是将measre与layout的过程进行了异步化，核心原理就是利用CPU的闲置时间提前在异步线程中完成measure和layout的过程，仅在UI线程中完成绘制工作。

**那么Android为什么不自己实现异步布局呢?**

主要有以下两原因：

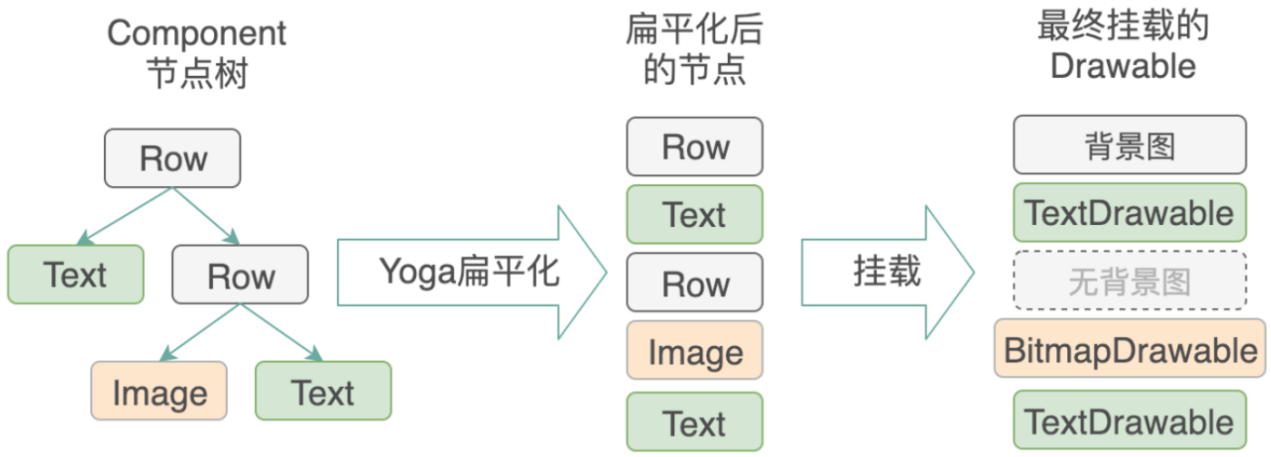
- 1、因为View的属性是可变的，只要属性发生变化就可能导致布局变化，所以需要重新计算布局，那么提前异步去计算布局的意义就不大了。而Litho组件的属性是不可变的，因此它的布局计算结果也是不变的。
- 2、提前异步布局需要去提前创建好接下来用到的若干条目的视图，但是Android原生的View作为视图单元，不仅包含一个视图的所有属性，而且还负责视图的绘制工作。如果要在绘制前提前去计算布局，就需要预先去持有大量未展示的View实例，这将会大大增加App进程的内存占用。对于Litho的组件来说，它只是视图属性的一个集合，仅仅负责计算布局，绘制工作由指定的绘制单元来完成。因此在Litho中，提前创建好下面要用到的多个条目的组件，是不会有性能问题的。两者的绘制原理简图如下所示：




**2、布局自动扁平化**

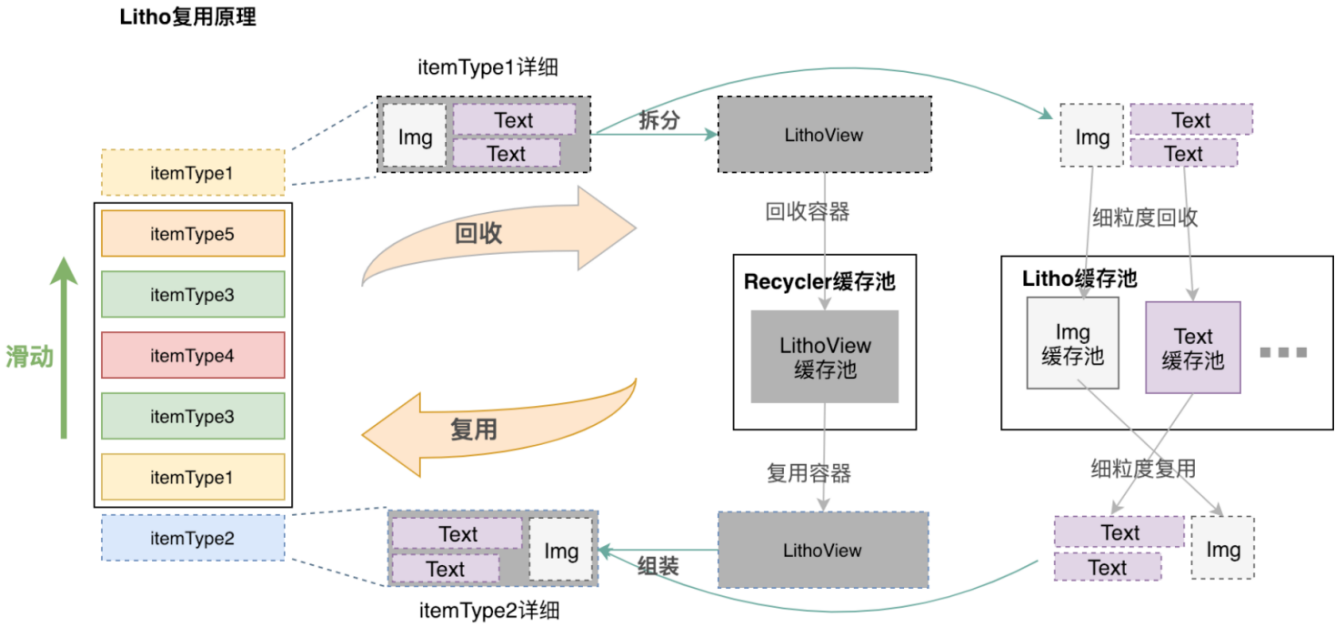
经过之前的学习，我们了解到Litho采用了一套自有的布局引擎Yoga，它会在布局的过程中去检测出不必要的布局嵌套层级，并自动去减少多余的层级以实现布局的扁平化，这可以显著减少渲染时的递归调用，加快渲染速度。例如，在实现一个图片带多个文字的布局中，我们通常会至少有两个布局层级，当然，**你也可以使用TextView的drawableStart方法 + 代码动态布局使用Spannable/Html.fromHtml(用来实现多种不同规格的文字) + lineSpaceExtra/lineSpacingMultiplier（用来调整多行文本的显示间距）**来将布局层级降为一层，但是这种实现方式比较繁琐，而通过使用Litho，我们可以把降低布局嵌套层级的任务全部丢给布局引擎Yoga去处理。由前面可知，Litho是使用Flexbox来创建布局的，并最终生成带有层级结

构的组件树。通过使用Yoga来进行布局计算，可以使用Flexbox的相对布局变成了只有一层嵌套的绝对布局。相比于ConstraintLayout，对于实现复杂布局的时候可读性会更好一些，因为ConstraintLayout此时会有过多的约束条件，这会导致可读性变差。此外，Litho自身还提供了许多挂载Drawable的基本视图组件，相比Viwe组件使用它们可以显著减少内存占用（通常会减少80%的内存占用）。Litho实现布局自动扁平化的原理图如下所示：




### 3、更细粒度地优化RecyclerView中组件的缓存与回收的方法

使用了RecyclerView与ListView这么久，我们明白它是以viewType为粒度来对一个组件集合统一进行缓存与回收的，并且，当viewType的类型越多，其对组件集合的缓存与回收的效果就会越差。相对于RecyclerView与ListView缓存与回收的粗粒度而言，Litho实现了更细粒度的回收机制，它是以Text、image、video等单个Component为粒度来作为其基准的，具体实现原理为在item回收前，会把LithoView中挂载的各个绘制单元进行解绑拆分出来，由Litho自己的缓存池去分类回收，然后在展示前由LithoView按照组件树的样式挂载组装各个绘制单元，这样就达到了细粒度复用的目的。毫无疑问，这不仅提高了其缓存的命中率与内存的使用率，也降低了提高了其滚动刷新的频率。更细粒度复用优化内存的原图如下所示：



由上图可以看出，滑出屏幕的itemType1会被拆分成一个个的视图单元。其中LithoView容器由Recycler缓存池回收，而其他视图单元则由Litho的缓存池分类回收，例如分类为Img缓存池、Text缓存池等等。
现在，我们对Litho已经比较了解了，它似乎很完美，但是任何事物都有其弊端，在学习一个新的事物时，我们不仅仅只去使用与了解它的优势，更应该对它的缺陷与弊端了如指掌。Litho在布局的过程中，使用了类似React的单向数据流设计，并且由于Litho是使用代码进行动态布局，这大大增加了布局的复杂度，而且，代码布局是无法实时预览的，这也增加了开发调试时的难度。
综上，对于某些性能性能要求高的场景，可以先使用Litho布局的方式去替换，特别是利用好RecyclerViewCollectionComponent与sections去充分提升RecylerView的性能。
现在，我们来使用RecyclerViewCollectionComponent与sections去创建一个可滚动的列表单元。
接下来，我们需要使用SectionsAPI，它可以将列表分为多个Section，然后编写GroupSectionSpec注解类来声明每个Section需要呈现的内容及其使用的数据。下面，我们创建一个ListSectoinSpec：

```
// 1、可以理解为一个组合Sectoin规格
@GroupSectionSpec
public class ListSectionSpec {
    @OnCreateChildren
    static Children onCreateChildren(final SectionContext context) {
        Children.Builder builder = Children.create();
        for (int i = 0; i < 20; i++) {
            builder.child(
                    // 单组件区域用来包含一个特定的组件
                    SingleComponentSection.create(context)
                     .key(String.valueOf(i))
                     .component(ListItem.create(context).build())
        };
        return builder.build();
    }
}
```

然后，我们将MainActivity onCreate方法中的步骤2替换为如下代码：

```
// 2、使用RecyclerCollectionComponent去绘制list
   RecyclerCollectionComponent recyclerCollectionComponent = RecyclerCollectionComponent.create(componentContext)
            // 使下拉刷新实现
            .disablePTR(true)
            .section(ListSection.create(new SectionContext(componentContext)).build())
            .build();
```

最终的显示效果如下所示：


Litho Study

JsonChao

Litho Study

JsonChao

Litho Study

JsonChao

Litho Study

JsonChao

# Litho Study

JsonChao

# Litho Study

JsonChao

# Litho Study

**如果我们需要显示不同UI的ListItem该怎么办呢?**

这个时候我们需要去自定义Component的属性，即props，它是一种不可变属性（此外还有一种可变属性称为State，但是其变化是由组件内部进行控制的，例如输入框、Checkbox等都是由组件内部去感知用户的行为，并由此更新组件的State属性），你设置的这些属性将会改变Component的行为或表现。Props是Component Spec中方法的参数，并且使用@Prop注解。

下面，我们使用props将ListItemSpec的onCreateLayout修改为可自定义组件属性的方法，如下所示：

```
@LayoutSpec
public class ListItemSpec {
    @OnCreateLayout
    static Component onCreateLayout(ComponentContext context,
                                    @Prop int bacColor,
                                    @Prop String title,
                                    @Prop String subTitle,
                                    @Prop int textSize,
                                    @Prop int subTextSize) {
        // Column的作用类似于HTML中的<div>标签
        return Column.create(context)
                .paddingDip(YogaEdge.ALL, 16)
                .backgroundColor(bacColor)
```

```
                .child(Text.create(context)
                           .text(title)
                        .textSizeSp(textSize)
                        .textColor(Color.BLUE)
                           .build())
                .child(Text.create(context)
                           .text(subTitle)
                        .textSizeSp(subTextSize)
                        .textColor(Color.MAGENTA)
                           .build())
                .build();
    }
}
```

奇妙之处就发生在我们所定义的@Prop注解与注解处理器之间，注解处理器以一种智能的方对组件构建过程中所关联的属性生成了对应的方法。

接下来，我们再修改ListSectionSpec类，如下所示：

```
@GroupSectionSpec
public class ListSectionSpec {
    @OnCreateChildren
    static Children onCreateChildren(final SectionContext context) {
        Children.Builder builder = Children.create();
        for (int i = 0; i < 20; i++) {
            builder.child(
                    SingleComponentSection.create(context)
                    .key(String.valueOf(i))
                    .component(ListItem.create(context)
                            .bacColor(i % 2 == 0 ? Color.BLUE:Color.MAGENTA)
                            .title("第" + i + "次练习")
                        .subTitle("JsonChao")
                            .textSize(36)
                            .subTextSize(24)
                            .build())
            );
        }
        return builder.build();
    }
}
```

最终的显示效果如下所示：


除此之外，我们还可以有更多的方式去定义@Prop，如下所示：

```
@Prop(optional = true, resType = ResType.DIMEN_OFFSET) int shadowRadius,
```

上面定义了一个可选的Prop，传入的shadowRadius是支持dimen规格的，如px、dp、sp等等。

**小结**

使用Litho，在布局性能上有很大的提升，但是开发成本太高，因为需要自己去实现很多的组件，并且其组件需要在编译时才能生成，不能够进行实时预览，但是可以把Litho封装成Flexbox布局的底层渲染引擎，以此实现上层的动态化，具体实现原理可参见Litho在美团动态化方案MTFlexbox中的实践。

### 2、使用Flutter实现高性能的UI布局

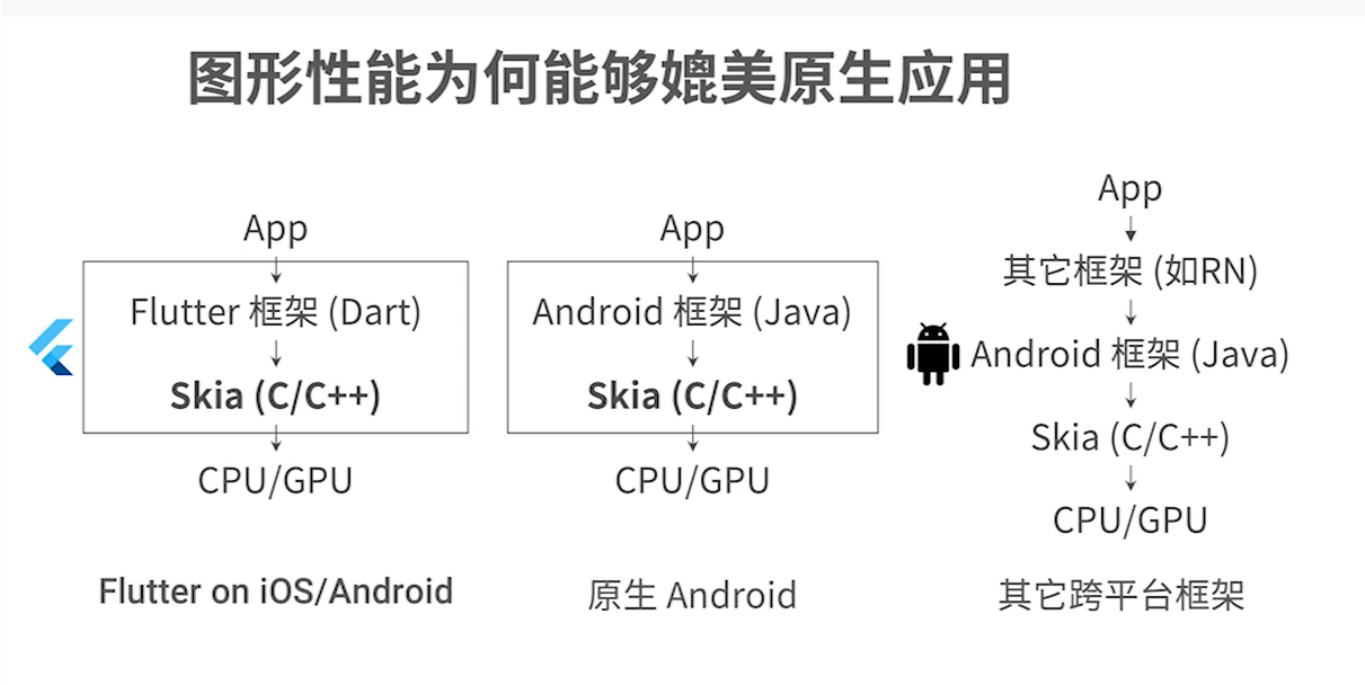
Flutter可以说是2019最火爆的框架之一了，它是 Google 开源的 UI 工具包，帮助开发者通过一套代码库高效构建多平台精美应用，支持移动、Web、桌面和嵌入式平台。对于Android来说，FLutter能够创作媲美原生的高性能应用，应用使用 Dart语言进行 开发。Flutter的架构类似于Android的层级架构，每一层都建立在前一层之上，其架构图如下所示：

在Framework层中，Flutter通过在 widgets 层组合基础 widgets 来构建 Material 层，而 widgets 层本身则是通过对来自 Rendering 层的低层次对象组合而来。而在Engine层，Flutter集成了Skia引擎用于进

行栅格化，并且使用了Dart虚拟机。

**那么Flutter的图形性能为何能够媲美原生应用呢?**

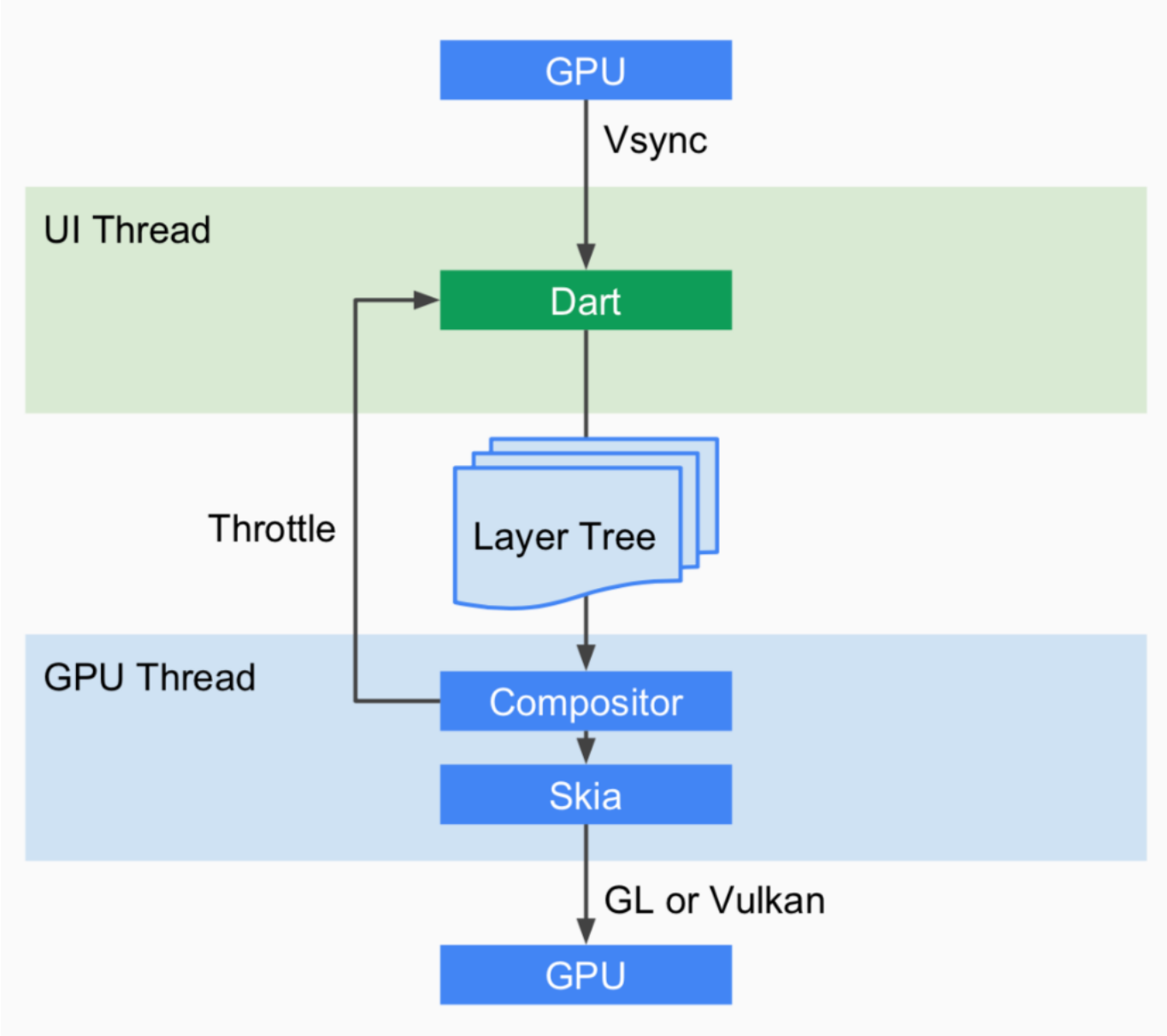
接下来，我们以Flutter、原生Android、其它跨平台框架如RN来做比较，它们的图形绘制调用层级图如下所示：




可以看到，Flutter框架的代码完全取代了Java层的框架代码，所以只要当Flutter框架中Dart代码的效率可以媲美原生框架的Java代码的时候，那么总体的Flutter App的性能就能够媲美原生的APP。而反观其它流行的跨平台框架如RN，它首先需要调用自身的Js代码，然后再去调用Java层的代码，这里比原生和Flutter的App显然多出来一个步骤，所以它的性能肯定是不及原生的APP的。此外，Flutter App不同于原生、RN，它内部是直接包含了Skia渲染引擎的，只要Flutter SDK进行升级，Skia就能够升级，这样Skia的性能改进就能够同步到Flutter框架之中。而对于Android原生和RN来说，只能等到Android系统升级才能同步Skia的性能改进。

**而Flutter又是如何实现高性能UI布局的呢?**

接下来，我们来大致了解一下Flutter的UI绘制原理，它主要是通过VSYNC信号来使UI线程和GPU线程有条不紊的周期性的去渲染界面，其绘制原理图如下所示：




绘制步骤大致如下：

- 1、首先 UI Runner 会执行 root isolate（可简单理解为Dart VM的线程），它会告诉引擎层有帧要渲染，当需要渲染则会调用到Engine的ScheduleFrame()来注册VSYNC信号回调，一旦触发回调doFrame()，并当它执行完成后，便会移除回调方法，也就是说一次注册一次回调。
- 2、当需要再次绘制则需要重新调用到ScheduleFrame()方法，该方法的唯一重要参数regenerate_layer_tree决定在帧绘制过程是否需要重新生成layer tree，还是直接复用上一次的layer tree。
- 3、接着，执行的是UI线程绘制过程中最核心的WidgetsBinding的drawFrame()方法，然后会创建layer tree视图树。
- 4、然后 Layer Tree 会交给 GPU Task Runner 进行合成和栅格化。
- 5、最后，GPU Task Runner会利用Skia库结合GL或Vu'lkan将layer tree提供的信息转化为平台可执行的GPU指令。

此外，Flutter 也采用了类似 Litho 的props属性不可变、Reat单向数据流的方案，用于将视图与数据分离。对于Flutter这一大前端领域的核心技术，笔者也是充满兴趣，后续会有计划对此进行深入研究，敬请

期待。

### 3、使用RenderThread 与 RenderScript

在Android 5.0之后，Android引进了RenderThread，它能够实现动画的异步渲染。但是目前支持RenderThread完全渲染的动画，只有两种，即ViewPropertyAnimator和CircularReveal（揭露动画）。对于CircularReveal使用比较简单且功能较为单一，就不多做过多的描述了。下面我简单说下ViewPropertyAnimator中如何去利用RenderThread。

**1、在ViewPropertyAnimator类系中，有一个ViewPropertyAnimatorRT ，它的主要作用就把动画交给RenderThread去处理。因此，我们需要先去创建对应view的ViewPropertyAnimatorRT，代码如下所示：**

```
/**
 * 使用反射的方式去创建对应View的ViewPropertyAnimatorRT(非hide类)
 */
private static Object createViewPropertyAnimatorRT(View view) {
    try {
        Class<?> animRtClazz = Class.forName("android.view.ViewPropertyAnimatorRT");
        Constructor<?> animRtConstructor = animRtClazz.getDeclaredConstructor(View.class);
        animRtConstructor.setAccessible(true);
        Object animRt = animRtConstructor.newInstance(view);
        return animRt;
    } catch (Exception e) {
        Log.d(TAG, "创建ViewPropertyAnimatorRT出错,错误信息:" + e.toString());
        return null;
    }
}
```

**2、接下来，我们需要将ViewPropertyAnimatorRT设置给ViewPropertyAnimator的mRTBackend字段，这样ViewPropertyAnimator才能利用它去将动画交给RenderThread处理，如下所示：**

```
private static void setViewPropertyAnimatorRT(ViewPropertyAnimator animator, Object rt) {
 try {
        Class<?> animClazz = Class.forName("android.view.ViewPropertyAnimator");
        Field animRtField = animClazz.getDeclaredField("mRTBackend");
```

```
            animRtField.setAccessible(true);
            animRtField.set(animator, rt);
        } catch (Exception e) {
            Log.d(TAG, "设置ViewPropertyAnimatorRT出错,错误信息:" + e.toStr
ing());
        }
}
/**
 * 在animator.start()即执行动画开始之前配置的方法
 */
public static void onStartBeforeConfig(ViewPropertyAnimator animato
r, View view) {
    Object rt = createViewPropertyAnimatorRT(view);
    setViewPropertyAnimatorRT(animator, rt);
}
```

**3、最后，在开启动画之前将ViewPropertyAnimatorRT实例设置进去即可，如下所示：**

```
ViewPropertyAnimator animator = v.animate().scaleY(2).setDuration(200
0);
AnimHelper.onStartBeforeConfig(animator, v);
animator.start();
```

现在，如果是做音视频或图像处理的工作，经常需要对图片进行高斯模糊、放大、锐化等操作，但是这里涉及大量的图片变换操作，例如缩放、裁剪、二值化以及降噪等。而图片的变换又涉及大量的计算任务，这个时候我们可以通过RenderScript去充分利用手机的GPU计算能力，以实现高效的图片处理。

而RenderScript的工作流程需要经历如下三个步骤：

- 1、RenderScript运行时API：提供进行运算的API。
- 2、反射层：相当于NDK中的JNI胶水代码，它是一些由Android编译工具自动生成的类，对我们写的RenderScript代码进行包装，以使得安卓层能够和RenderScript进行交互。
- 3、安卓框架：通过调用反射层来访问RenderScript运行时。

由于RenderScript主要是用于音视频、图像处理等细分领域，这里笔者就不继续深入扩展了，对于NDK、音视频领域的知识，笔者在今年会有一系列学习计划，目前大纲已经定制好了，如果有兴趣的朋友，可以了解下：Awesome-Android-NDK。

# 八、布局优化的常见问题

## 1、你在做布局优化的过程中用到了哪些工具？

我在做布局优化的过程中，用到了很多的工具，但是每一个工具都有它不同的使用场景，不同的场景应该使用不同的工具。下面我从线上和线下两个角度来进行分析。

比如说，我要统计线上的FPS，我使用的就是Choreographer这个类，它具有以下特性：

- 1、能够获取整体的帧率。
- 2、能够带到线上使用。
- 3、它获取的帧率几乎是实时的，能够满足我们的需求。

同时，在线下，如果要去优化布局加载带来的时间消耗，那就需要检测每一个布局的耗时，对此我使用的是AOP的方式，它没有侵入性，同时也不需要别的开发同学进行接入，就可以方便地获取每一个布局加载的耗时。如果还要更细粒度地去检测每一个控件的加载耗时，那么就需要使用LayoutInflaterCompat.setFactory2这个方法去进行Hook。

此外，我还使用了LayoutInspector和Systrace这两个工具，Systrace可以很方便地看到每帧的具体耗时以及这一帧在布局当中它真正做了什么。而LayoutInspector可以很方便地看到每一个界面的布局层级，帮助我们对层级进行优化。

### 2、布局为什么会导致卡顿，你又是如何优化的?

分析完布局的加载流程之后，我们发现有如下四点可能会导致布局卡顿：

- 1、首先，系统会将我们的Xml文件通过IO的方式映射的方式加载到我们的内存当中，而IO的过程可能会导致卡顿。
- 2、其次，布局加载的过程是一个反射的过程，而反射的过程也会可能会导致卡顿。
- 3、同时，这个布局的层级如果比较深，那么进行布局遍历的过程就会比较耗时。
- 4、最后，不合理的嵌套RelativeLayout布局也会导致重绘的次数过多。

对此，我们的优化方式有如下几种：

- 1、针对布局加载Xml文件的优化，我们使用了异步Inflate的方式，即AsyncLayoutInflater。它的核心原理是在子线程中对我们的Layout进行加载，而加载完成之后会将View通过Handler发送到主线程来使用。所以不会阻塞我们的主线程，加载的时间全部是在异步线程中进行消耗的。而这仅仅是一个从侧面缓解的思路。
- 2、后面，我们发现了一个从根源解决上述痛点的方式，即使用X2C框架。它的一个核心原理就是在开发过程我们还是使用的XML进行编写布局，但是在编译的时候它会使用APT的方式将XML布局转换为Java的方式进行布局，通过这样的方式去写布局，它有以下优点：1、它省去了使用IO的方式去加载XML布局的耗时过程。2、它是采用Java代码直接new的方式去创建控件对象，所以它也没有反射带来的性能损耗。这样就从根本上解决了布局加载过程中带来的问题。
- 3、然后，我们可以使用ConstraintLayout去减少我们界面布局的嵌套层级，如果原始布局层级越深，它能减少的层级就越多。而使用它也能避免嵌套RelativeLayout布局导致的重绘次数过多。
- 4、最后，我们可以使用AspectJ框架（即AOP）和LayoutInflaterCompat.setFactory2的方式分别去建立线下全局的布局加载速度和控件加载速度的监控体系。

### 3、做完布局优化有哪些成果产出?

- 1、首先，我们建立了一个体系化的监控手段，这里的体系还指的是线上加线下的一个综合方案，针对线下，我们使用AOP或者ARTHook，可以很方便地获取到每一个布局的加载耗时以及每一个控件的加载耗时。针对线上，我们通过Choreographer.getInstance().postFrameCallback的方式收集到了FPS，这样我们可以知道用户在哪些界面出现了丢帧的情况。
- 2、然后，对于布局监控方面，我们设立了FPS、布局加载时间、布局层级等一系列指标。

- 3、最后，在每一个版本上线之前，我们都会对我们的核心路径进行一次Review，确保我们的FPS、布局加载时间、布局层级等达到一个合理的状态。

## 九、总结

对于Android的布局优化，笔者以一种自顶向下，层层递进的方式和大家一起深入地去探索了Android中如何将布局优化做到极致，其中主要涉及以下八大主题：

- 1、绘制原理：CPU\GPU、Android图形系统的整体架构、绘制线程、刷新机制。
- 2、屏幕适配：OLED 屏幕和 LCD 屏幕的区别、屏幕适配方案。
- 3、优化工具：使用Systrace来进行布局优化、利用Layout Inspector来查看视图层级结构、采用Choreographer来获取FPS以及自动化测量 UI 渲染性能的方式（gfxinfo、SurfaceFlinger等dumpsys命令）。
- 4、布局加载原理：布局加载源码分析、LayoutInflater.Factory分析。
- 5、获取界面布局耗时：使用AOP的方式去获取界面加载的耗时、利用LayoutInflaterCompat.setFactory2去监控每一个控件加载的耗时。
- 6、布局优化常规方案：使用AOP的方式去获取界面加载的耗时、利用LayoutInflaterCompat.setFactory2去监控每一个控件加载的耗时。
- 7、布局优化的进阶方案：使用异步布局框架Litho、使用Flutter实现高性能的UI布局、使用RenderThread实现动画的异步渲染与 利用RenderScript实现高效的图片处理。
- 8、布局优化的常见问题。

可以看到，布局优化看似是Android性能优化中最简单的专项优化项，但是笔者却花费了整整三万字的篇幅才能比较完整地将其核心知识传授给大家。因此，不要小看每一个专项优化点，深入进去，必定满载而归。

**参考链接：**

---


1、国内Top团队大牛带你玩转Android性能分析与优化 第五章 布局优化
2、极客时间之Android开发高手课 UI优化
3、手机屏幕的前世今生 可能比你想的还精彩
4、OLED 和 LCD 什么区别?
5、Android 目前稳定高效的UI适配方案
6、骚年你的屏幕适配方式该升级了!-smallestWidth 限定符适配方案
7、dimens_sw github
8、一种极低成本的Android屏幕适配方式
9、骚年你的屏幕适配方式该升级了!-今日头条适配方案
10、今日头条屏幕适配方案终极版正式发布!
11、使用Systrace分析UI性能
12、GAPID-Graphics API Debugger
13、Android性能优化之渲染篇
14、Android 屏幕绘制机制及硬件加速
15、Android 图形处理官方教程

16、Vulkan – 高性能渲染
17、Android Vulkan Tutorial
18、Test UI performance-gfxinfo
19、使用dumpsys gfxinfo 测UI性能（适用于Android6.0以后）
20、TextureView API
21、PrecomputedText API
22、Litho Tutorial
23、基本功 | Litho的使用及原理剖析
24、Flutter官方文档中文版
25、[Google Flutter 团队出品] 深入了解 Flutter 的高性能图形渲染
26、Flutter渲染机制—UI线程
27、RenderThread:异步渲染动画
28、RenderScript官方文档
29、RenderScript :简单而快速的图像处理
30、RenderScript渲染利器