从XML到View显示在屏幕上，都发生了什么?

程序员江同学 郭霖 2022-08-09 08:00 发表于江苏

/ 今日科技快讯 /

近日，社交网络巨头Meta发布了最新款人工智能聊天机器人BlenderBot3，但这款机器人对Meta首席执行官马克·扎克伯格看法似乎有点儿自相矛盾，并吐槽其“那么有钱还老穿同款衣服”。

/ 作者简介 /

本篇文章转自程序员江同学的博客，文章主要分享了他对Android开发中XML和View之间关系的探索分析，相信会对大家有所帮助！

原文地址：

> https://juejin.cn/post/6991483318625632286

/ 前言 /

View绘制可以说是Android开发的必备技能，但是关于View绘制的的知识点也有些繁杂。

如果我们从头开始阅读源码，往往千头万绪，抓不住要领。

目前当我们写页面时，布局都是写在XML里的，我们可以思考下：布局从XML到显示到屏幕上，都发生了什么，可以分为哪几个部分?

我们将整个显示流程分解为以下几个部分。

1. 代码是怎么从XML转换成View的?
2. View是怎么添加到页面上的?
3. 在内存中View到底是怎么绘制的?
4. View绘制完成后是怎么显示到屏幕上的?

本文目录如下所示:

/ XML是怎么转换成View的? /

我们都知道，在android中写布局一般是通过XML，然后通过setContentView方法配置到页面中。看来XML转换成View就是在这个setContentView中了。

setContentView中做了什么

```
public void setContentView(int resId) {
   ensureSubDecor();
   ViewGroup contentParent = mSubDecor.findViewById(android.R.id.content);
   contentParent.removeAllViews();
   LayoutInflater.from(mContext).inflate(resId, contentParent);
   mAppCompatWindowCallback.getWrapped().onContentChanged();
}
```

可以看到resId传给了我们熟悉的LayoutInflater，看来xml转化成View就是在LayoutInflater方法中实现的了。

LayoutInflater中做了什么?

```
public View inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean attachToRoot) {
   final Resources res = getContext().getResources();
   //预编译直接返回view,目前还未启用
   View view = tryInflatePrecompiled(resource, res, root, attachToRoot);
   if (view != null) {
      return view;
   }
   XmlResourceParser parser = res.getLayout(resource);
   try {
      //真正将`XML`转化为`View`
      return inflate(parser, root, attachToRoot);
   } finally {
      parser.close();
   }
}
```

代码也比较简单，我们一起来分析下。

1. 首先我们需要明确，将XML转化为View牵涉到一些耗时操作，比如XML解析是一个io操作，将XML转化为View涉及到反射，这也是耗时的
2. 我们可以看到在解析前有个tryInflatePrecompiled方法，这个方法就是希望可以在编译阶段直接预编译XML，在运行时直接返回构建好的View，看起来Google希望通过这种方式解决XML的性能问题。不过这个功能目前还没有启用，因此此方法直接返回null，目前生效的还是下面的方法
3. 真正将XML解析为View的还是在inflate方法中，将标签名转化为View的名称，XML中的各种属性转化为AttributeSet对象，然后通过反射生成View对象

由于篇幅原因，这里就不再粘贴inflate方法的源码了，里面主要需要注意下setFactory与setFactory2方法。在真正进行反射前，会先调用这两个方法尝试创建一下View，而且系统开放了API，我们可以自定义解析XML方式。

这就给了我们一些面向切面编程的空间，可以利用这两个API实现换肤，替换字体，替换View，提升View构建速度等操作。

小结

XML转化为View转化为主要是通过LayoutInflator来完成的，将标签名转化为View的名称，XML中的各种属性转化为AttributeSet对象，然后通过反射生成View对象。

这个过程中存在一些耗时操作，比如解析XML的IO操作，通过反射生成View等，我们可以通过多种方式优化这个过程，比如将反向的耗时转移到编译期。

/ View是怎么添加到页面上的？ /

经过上面这步，View已经被创建出来了，但是View又是怎么添加到页面(Activity)上的呢？我们再来看下setContentView方法。

```
public void setContentView(int resId) {
   ensureSubDecor();
   ViewGroup contentParent = mSubDecor.findViewById(android.R.id.content);
   contentParent.removeAllViews();
   LayoutInflater.from(mContext).inflate(resId, contentParent);
   mAppCompatWindowCallback.getWrapped().onContentChanged();
}
```

LayoutInflater有两个参数，第二个参数就是root，即创建出的view要被添加的父view，所以答案也就呼之欲出了，创建出来的view被添加到了contentParent上，即R.id.content上。那么问题来了，这个R.id.content是哪来的呢？

R.id.content从何而来?

我们看到，setContentView开头调用了ensureSubDecor方法，一起来看下它的源码。

```
private void ensureSubDecor() {
   if (!mSubDecorInstalled) {
```

```
            mSubDecor = createSubDecor();
        }
    }
    private ViewGroup createSubDecor() {
        // Now let's make sure that the Window has installed its decor by retrieving it
        ensureWindow();
        mWindow.getDecorView();

        final LayoutInflater inflater = LayoutInflater.from(mContext);
        ViewGroup subDecor = null;

        //省略其他样式subDecor布局的实例化
        //包含 actionBar floatTitle ActionMode等样式
        subDecor = (ViewGroup) inflater.inflate(R.layout.abc_screen_simple, null);
        final ContentFrameLayout contentView = (ContentFrameLayout) subDecor.findViewById(R.id.action_

        final ViewGroup windowContentView = (ViewGroup) mWindow.findViewById(android.R.id.content);
        // 把`contentView`的id设置为android.R.id.content,把windowContentView的id设置为View.NO_ID
        windowContentView.setId(View.NO_ID);
        contentView.setId(android.R.id.content);

        //将subDecor添加到window
        mWindow.setContentView(subDecor);
        return subDecor;
    }
```

可以看出，主要工作是创建subDecor并添加到window上。

- 步骤一：确认window并attach(设置背景等操作)
- 步骤二：获取DecorView，因为是第一次调用所以会installDecor(创建DecorView和windowContentView)
- 步骤三：从xml中实例化出subDecor布局
- 步骤四：将subDecor的contentView的id设置为R.id.content
- 步骤五：将subDecor添加到window中

现在我们已经知道R.id.content从何而来了，并且知道了subDecor最终会添加到window中。那么问题来了，window又是什么呢?

window到底是什么?

我们上文提到，我们创建的view会被添加到subDecor上，最后会被添加到window中，那么window是什么？为什么要有window?

我们在应用中有多个页面，手机上也有多个应用，这么多页面同时只能有一个页面显示在手机上，这个时候就需要有一个机制来管理当前显示哪个页面。

于是Android在系统进程中创建了一个系统服务WindowManagerService(WMS)专门用来管理屏幕上的窗口，而View只能显示在对应的窗口上，如果不符合规定就不开辟窗口进而对应的View也无法显示 。

window机制就是为了管理屏幕上的view的显示以及触摸事件的传递问题

值得注意的事，上面的window与窗口很容易混淆，Android SDK中的Window是一个抽象类，它有一个唯一实现类PhoneWindow，PhoneWindow内部会持有一个DecorView(根View)，它的职责就是对DecorView做一些标准化的处理，比如标题、背景、导航栏、事件中转等，很显然与我们前面所说的窗口概念不符合。

总得来说PhoneWindow只是提供些标准的UI方案，与窗口不等价。窗口是一个抽象概念，即当前应该显示哪个页面，系统通过WindowManagerService(WMS)来管理。

View什么时候真正可见?

上面提到PhoneWindow只是提供些标准的UI方案，并不是真正的窗口。那么我们的View到底什么时候添加到窗口上，什么时候真正对用户可见?

```
#ActivityThread
public void handleResumeActivity(...) {
   //...
   //注释1
   r.window = r.activity.getWindow();
   View decor = r.window.getDecorView();
   decor.setVisibility(View.INVISIBLE);
   ViewManager wm = a.getWindowManager();
   WindowManager.LayoutParams l = r.window.getAttributes();
   ...
   //注释2
```

```
    wm.addView(decor, l);
    ...
}

#ViewRootImpl.java
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
    synchronized (this) {
        if (mView == null) {
            //记录DecorView
            mView = view;
            //省略
            //开启View的三大流程（measure、layout、draw）
            requestLayout();
            try {
                //添加到WindowManagerService里，这里是真正添加window到底层
                //这里的返回值判断window是否成功添加，权限判断等。
                //比如用Application的context开启dialog，这里会添加不成功
                // 注释3
                res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                        getHostVisibility(), mDisplay.getDisplayId(), mTmpFrame,
                        mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
                        mAttachInfo.mOutsets, mAttachInfo.mDisplayCutout, mInputChannel,
                        mTempInsets);
                setFrame(mTmpFrame);
            } catch (RemoteException e) {
            }
            //省略
            //输入事件接收
        }
    }
}
```

- 注释1处会从Activity中取出PhoneWindow，DecorView，WindowManager
- 注释2处调用了WindowManager的addView方法，顾名思义就是将DecorView添加至窗口当中
- 最后会调到ViewRootImpl中注释3处，这里才是真正的通过WMS在屏幕上开辟一个窗口，到这一步我们的View也就可以显示到屏幕上了

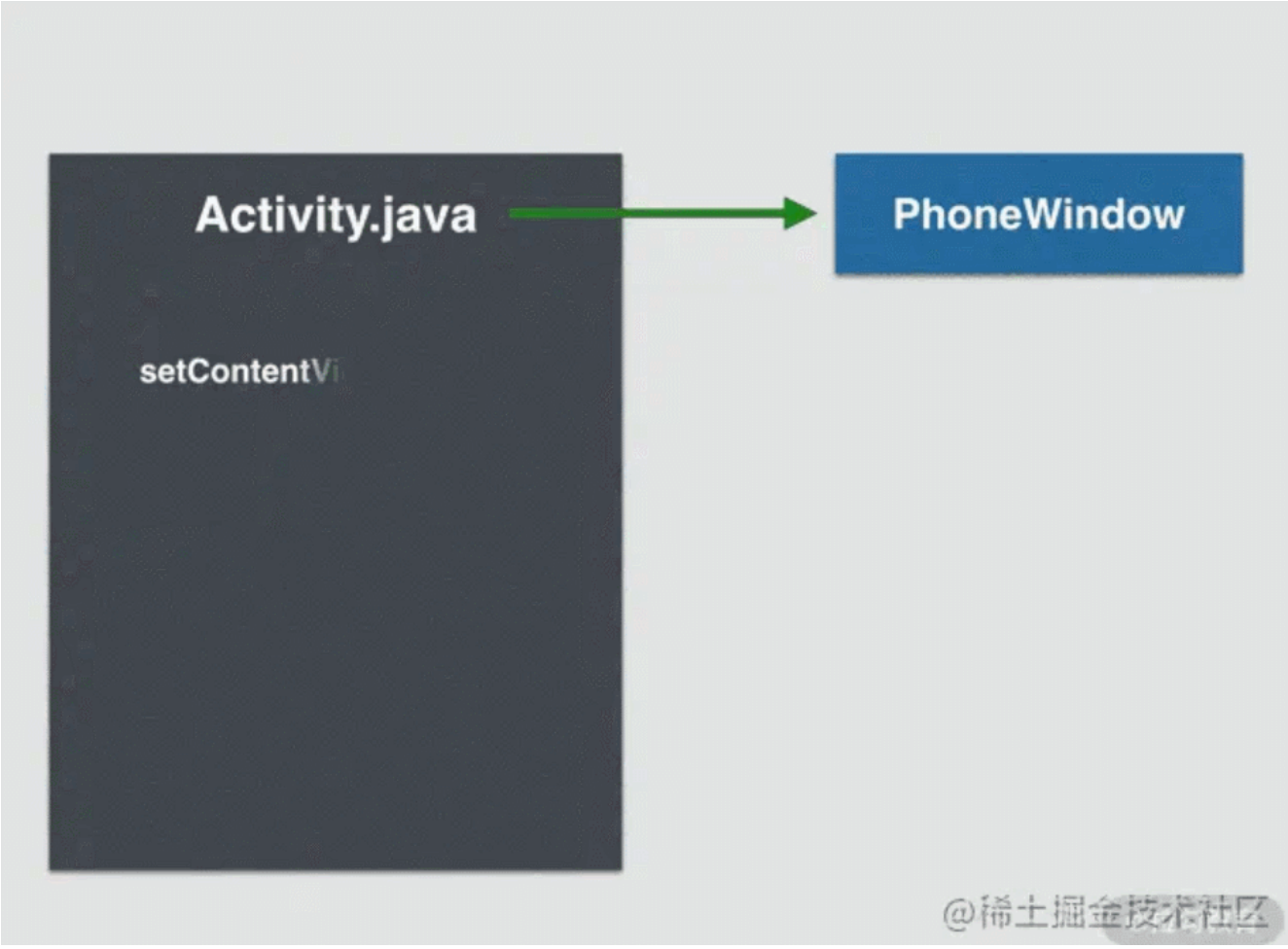
可以看出，当我们打开一个Activity时，界面真正可见是在onResume之后。

Activity，PhoneWindow，View的关系

- Phonewindow是activity的一个成员变量，会在Activity.attatch时初始化

- PhoneWindow是View的容器，对DecorView做一些标准化的处理，比如标题、背景、导航栏、事件中转等
- Activity则提供了窗口的生命周期，屏蔽了窗口机制的复杂细节，开发者只需要基于模板方法开发即可。

如下图所示：



小结

View添加到页面上，主要经过了这么几个过程：

1. 启动activity
2. 创建PhoneWindow
3. 设置布局setContentView,将layoutId转化为View
4. 确认subDecorView的初始化，将subDecorView添加到PhoneWindow中

5. 添加layoutId转化后的View到android.R.id.content上
6. 在onResume中将DecorViewView添加到WindowManager中
7. View真正显示到屏幕上了

/ View到底是怎么绘制的? /

经过上一步，View已经添加到window上了，接下来就是View本身的绘制了。View的绘制主要经过以下几步：

1、首先需要确定View占的空间尺寸(measure)
2、确定了空间尺寸，就需要确定摆放在哪个位置(layout)
3、确认了摆放位置，就需要确定在上面展示些什么东西(draw)

这几个阶段，View已经封装了模板方法给我们，我们直接重写onMeasure，onLayout，onDraw这几个方法就好了。而绘制的入口，就是上面ViewRootImpl.setView中的requestLayout。

requestLayout如何触发绘制

上文说到requestLayout会触发绘制，我们一起来看下源码。

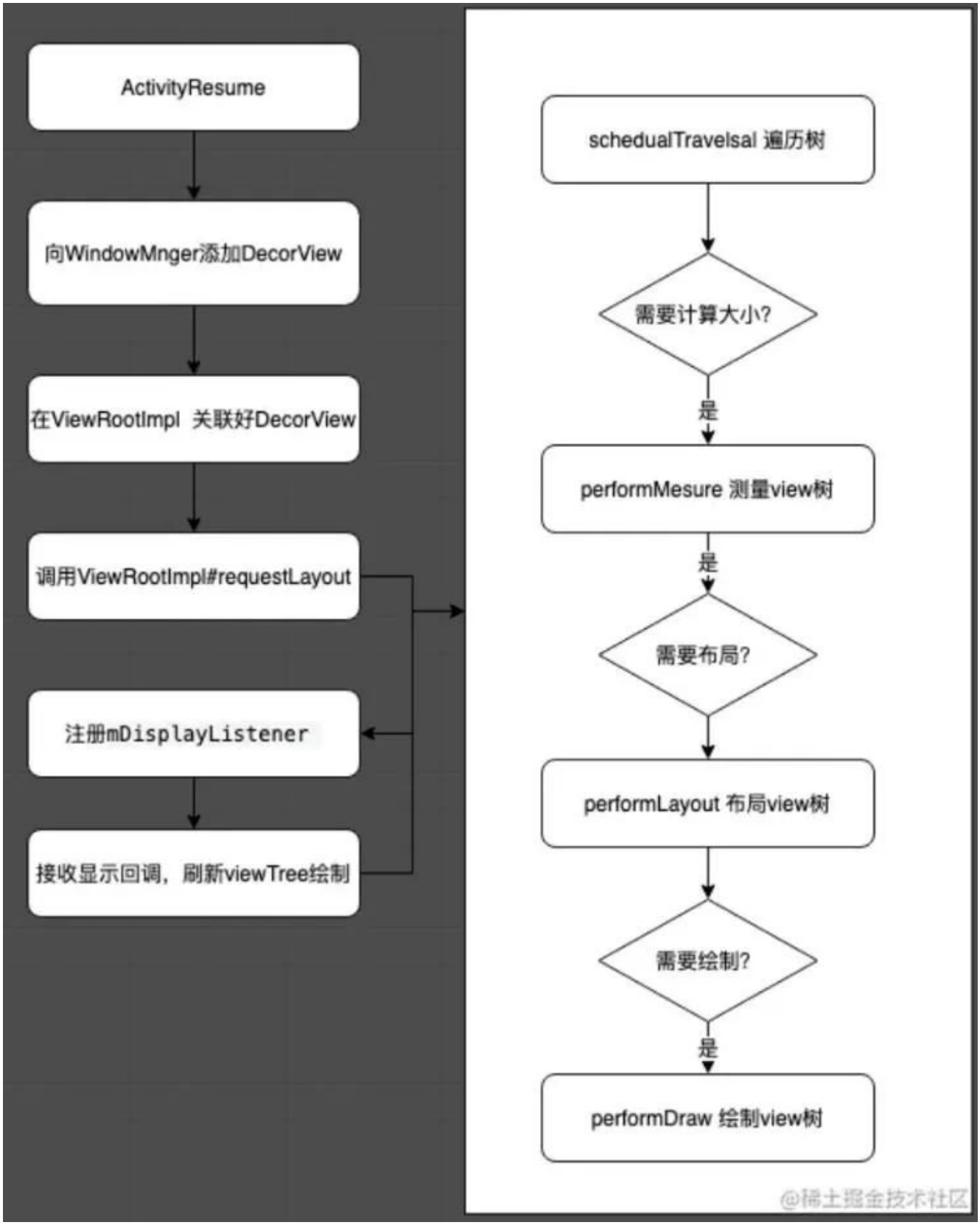
```
ViewRootImpl.java
public void requestLayout() {
   if (!mHandlingLayoutInLayoutRequest) {
      //检查是否是主线程，如果不是则直接抛出异常，ViewRootImpl创建的时候生成一个主线程引用
      //用当前线程和引用比较，如果是同一个则是主线程
      //这也是为什么在子线程对View进行更新、绘制会报错的原因
      checkThread();
      //用来标记需要进行layout
      mLayoutRequested = true;
      //绘制请求
      scheduleTraversals();
   }
}

void scheduleTraversals() {
   if (!mTraversalScheduled) {
      //标记一次绘制请求，用来屏蔽短时间内的重复请求
```

```
            mTraversalScheduled = true;
            //往主线程Looper队列里放同步屏障消息，用来控制异步消息的执行
            mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
            //放入mChoreographer队列里
            //主要是将mTraversalRunnable放入队列
            mChoreographer.postCallback(
                    Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
            //省略
        }
    }

    final class TraversalRunnable implements Runnable {
        @Override
        public void run() {
            doTraversal();
        }
    }

    void doTraversal() {
        //没有取消绘制的话则开始绘制
        if (mTraversalScheduled) {
            mTraversalScheduled = false;
            //移除同步屏障
            mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

            //真正开始执行measure、layout、draw等方法
            performTraversals();
        }
    }
```

requestLayout中其实主要也是做了以下几件事：

1. 检查绘制的线程与View创建的线程是否是同一个线程
2. 通过Handler同步屏障机制，保证UI绘制消息优先级是最高的
3. 将mTraversalRunnable传入Choreographer，监听vsync信号
4. 收到vsync信号后会回调TraversalRunnable，移除同步屏障并开始真正的measure，layout，draw

View绘制流程图如下：



MeasureSpec分析在测量过程中，会传入一个MeasureSpec参数，MeasureSpec封装了View的规格尺寸参数，包括View的宽高以及测量模式。

它的高2位代表测量模式，低30位代表尺寸。其中测量模式总共有3种。

- UNSPECIFIED：未指定模式不对子View的尺寸进行限制。
- AT_MOST：最大模式对应于wrap_content属性，父容器已经确定子View的大小，并且子View不能大于这个值。
- EXACTLY：精确模式对应于match_parent属性和具体的数值，子View可以达到父容器指定大小的值。

普通view的MeasureSpec创建规则如下：

View 布局参数 / ViewGroup 测量模式	精确值	MATCH_PARENT	WRAP_CONTENT
EXACTLY	SpecSize: 精确值 SpecMode：EXACTLY	SpecSize: ViewGroup Size SpecMode：EXACTLY	SpecSize: ViewGroup Size SpecMode：AT_MOST
AT_MOST	SpecSize: 精确值 SpecMode：EXACTLY	SpecSize: ViewGroup Size SpecMode：AT_MOST	SpecSize: ViewGroup Size SpecMode：AT_MOST
UNSPECIFIED	SpecSize: 精确值 SpecMode：EXACTLY	SpecSize: ViewGroup Size SpecMode：UNSPECIFIED	SpecSize: ViewGroup Size SpecMode：UNSPECIFIED

ViewGroup 获取子 View 测量规格

结合这个表，我们可以一起来看一个问题。

```
<FrameLayout
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:background="@android:color/red"
  xmlns:android="http://schemas.android.com/apk/res/android">
  <View
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@android:color/blue"/>
</FrameLayout>
```

请问这样一个布局，最后是什么颜色呢?

答案是蓝色，并且占满屏幕。

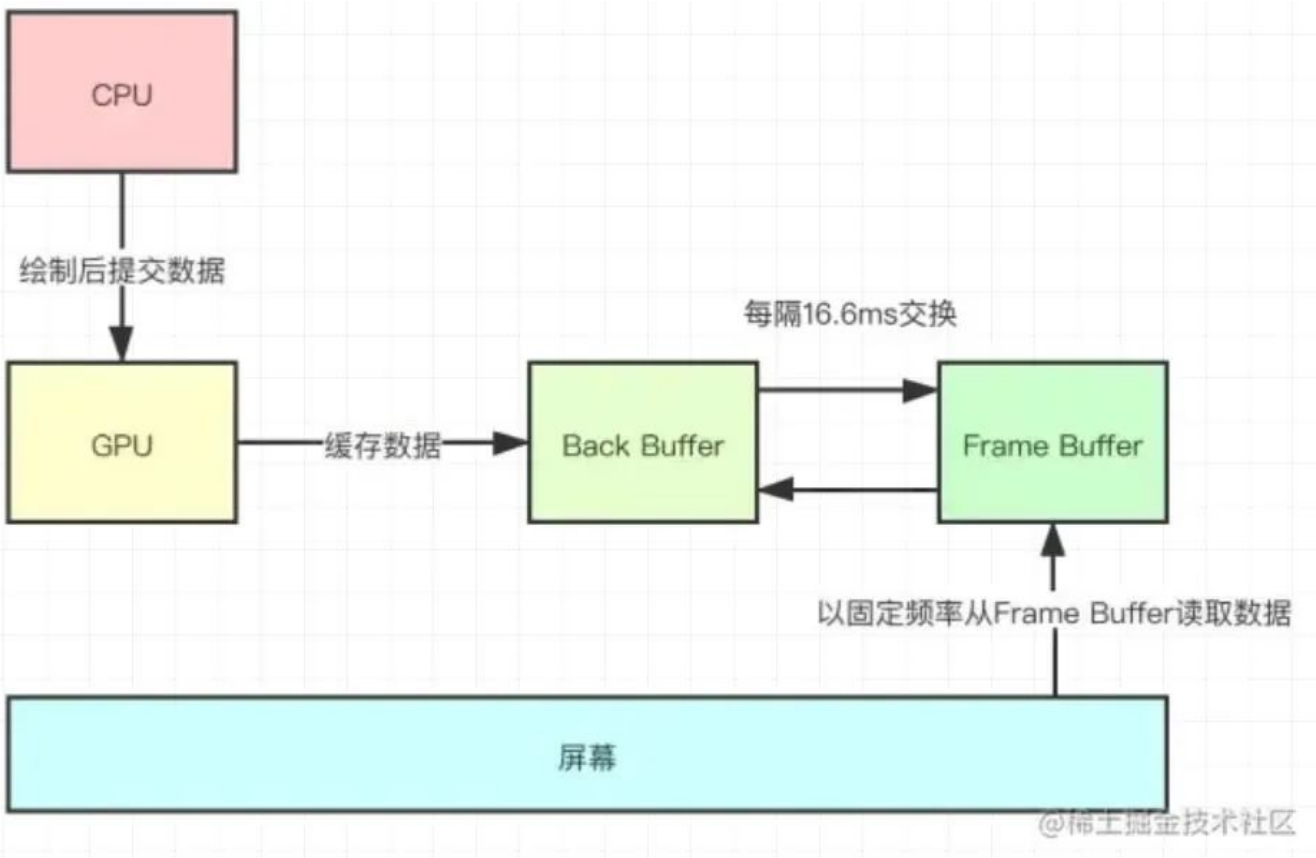
简单来说，当我们自定义View 时，如果没有对MODE做处理，设置wrap_content和match_content结果其实是一样的，View 的宽高都是取父 View 的宽高。

小结

1. View的绘制需要定位，测量，绘制三个步骤，为了简化自定义View的过程，官方已经提供了模板方法，我们重写相关方法即可
2. ViewRootImpl中的requestLayout是绘制的入口，当然我们在View中调用invalidate或者requestLayout也会触发重绘
3. 绘制过程本质上也是通过Handler发送消息，为了提高绘制消息的优先级，会开启同步屏蔽机制
4. 将mTraversalRunnable传入Choreographer，监听vsync信号。注意，vsync信号注册了才会监听。
5. 收到vsync信号后会回调TraversalRunnable，移除同步屏障并开始真正的measure，layout，draw过程
6. 接下来就是回调各个View的onMeasure，onLayout，onDraw过程

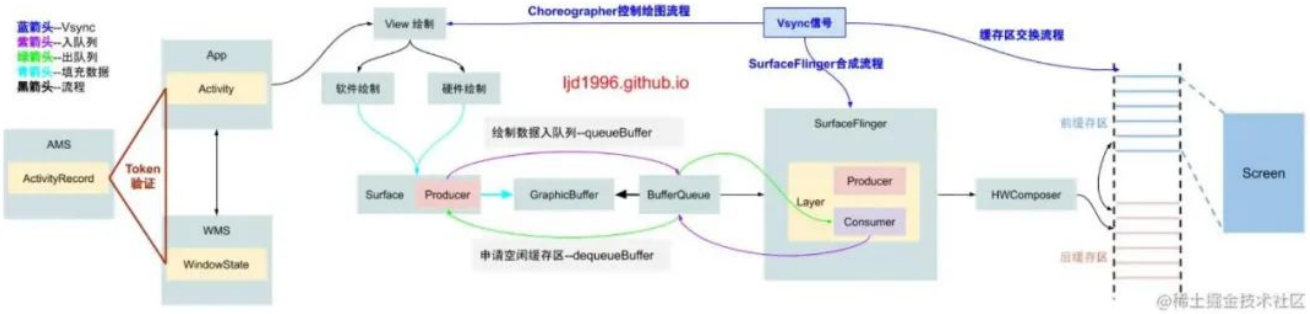
/ View绘制完成后是怎么显示到屏幕上的？ /

目前我们已经知道了，从XML到调用View.onDraw的过程，但是从onDraw到显示到屏幕上似乎还有些距离我们知道，View最后要显示在屏幕上，CPU负责计算帧数据，把计算好的数据交给GPU，GPU会对图形数据进行渲染，渲染好后放到buffer(图像缓冲区)里存起来，然后Display（屏幕或显示器）负责把buffer里的数据呈现到屏幕上。



那么问题来了，canvas.draw是怎么转化成Graphic Buffer的呢?

其大概流程如图所示:



可以看出，这个过程还是相当复杂的，由于篇幅原因，这里就不展开了。

/ 总结 /

从XML到View显示到屏幕上主要涉及到以下知识点:

1. Activity的启动
2. LayoutInflater填充View的原理
3. PhoneWindow，Activity，View的关系
4. Android窗口机制与WindowManagerService管理窗口
5. View的绘制流程，measure，layout，draw等与Handler同步屏障机制
6. Android屏幕刷新机制，VSync信号监听，三级缓冲等
7. Android图形绘制，包括SurfaceFinger工作流程，软件绘制，硬件加速等

这篇文章其实已经比较长了，但是要完全了解从XML到显示到屏幕上的过程，还是不够详细，有很多地方只做了简述。

推荐阅读：

我的新书，《第一行代码 第3版》已出版！

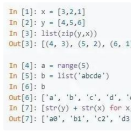
在微软工作365天，还你一个我眼中更加真实的微软

一个Android沉浸式状态栏上的黑科技

张大鹏520

Python的一些日常高频写法

猿大侠



Spring Boot 之 MDC 实现全链路调用日志跟踪

BUG弄潮儿