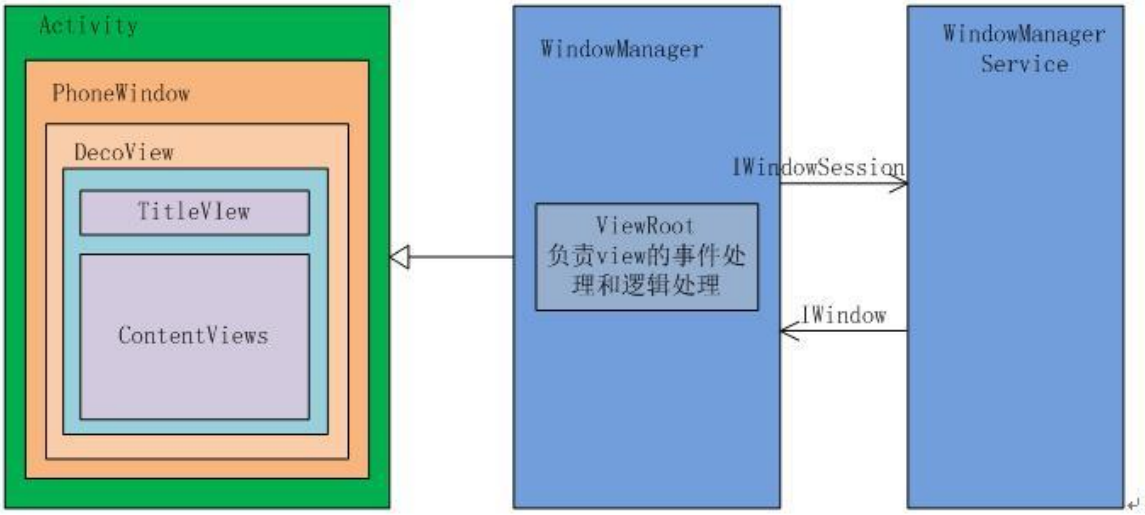
PDF-XChange
Click to buy NOW!
www.docu-track.com

Android View 绘制流程

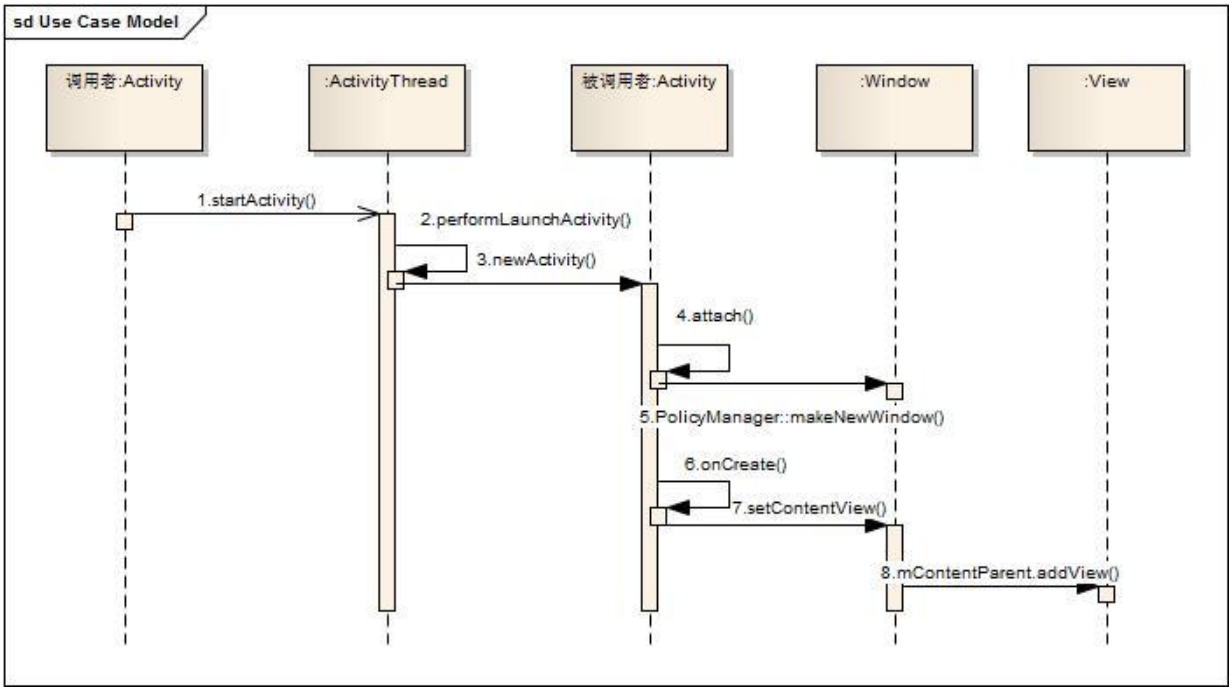
框架分析

在之前的下拉刷新中，小结过触屏消息先到 WindowManagerService（Wms）然后顺次传递给 ViewRoot（派生自 Handler），经 decor view 到 Activity 再传递给指定的 View，这次整理 View 的绘制流程，通过源码可知，这个过程应该没有涉及到 IPC（或者我没有发现），需要绘制时在 UI 线程中通过 ViewRoot 发送一个异步请求消息，然后 ViewRoot 自己接收并不处理这个消息。

在正式进入 View 绘制之前，首先需要明确一下 Android UI 的架构组成，偷图如下：

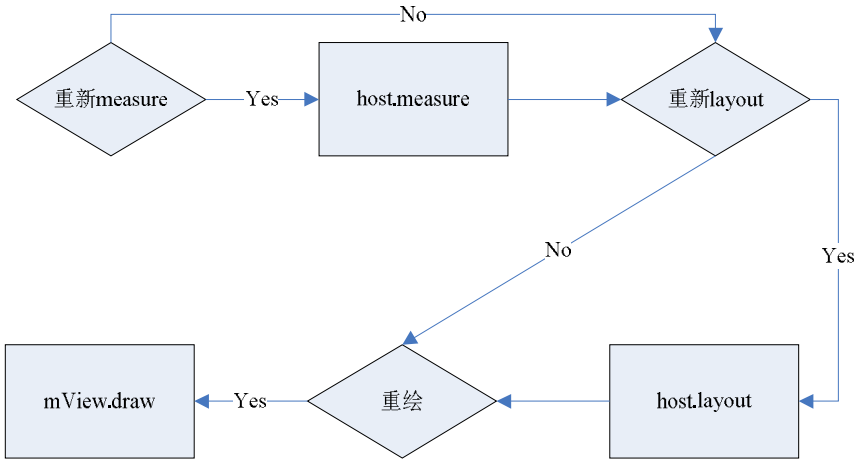


上述架构很清晰的呈现了 Activity、Window、DecorView（及其组成）、ViewRoot 和 WMS 之间的关系，我通过源码简单理了下从启动 Activity 到创建 View 的过程，大致如下



在上图中，performLaunchActivity 函数是关键函数，除了新建被调用的 Activity 实例外，还负责确保 Activity 所在的应用程序启动、读取 manifest 中关于此 activity 设置的主题信息以及上图中对“ 6.onCreate”调用也是通过对 mInstrumentation.callActivityOnCreate 来实现的。图中的“ 8. mContentParent.addView”其实就是架构图中 phoneWindow 内 DecorView 里面的 ContentViews，该对象是一个 ViewGroup 类实例。在调用 AddView 之后，最终就会触发 ViewRoot 中的 scheduleTraversals 这个异步函数，从而进入 ViewRoot 的 performTraversals 函数，在 performTraversals 函数中就启动了 View 的绘制流程。

performTraversals 函数在 2.3.5 版本源码中就有近六百行的代码，跟我们绘制 view 相关的可以抽象成如下的简单流程图



流程图中的 host 其实就是 mView，而 ViewRoot 中的这个 mView 其实就是 DecorView，之所以这么说，又得具体看源码中 ActivityThread 的 handleResumeActivity 函数，在这里我就不展开了。上述流程主要调用了 View 的 measure、layout 和 draw 三个函数。

PDF-XChange
Click to buy NOW!
www.docu-track.com

measure 过程分析

因为 DecorView 实际上是派生自 FrameLayout 的类，也即一个 ViewGroup 实例，该 ViewGroup 内部的 ContentViews 又是一个 ViewGroup 实例，依次内嵌 View 或 ViewGroup 形成一个 View 树。所以 measure 函数的作用是为整个 View 树计算实际的大小，设置每个 View 对象的布局大小（“窗口”大小）。实际对应属性就是 View 中的 mMeasuredHeight（高）和 mMeasureWidth（宽）。

在 View 类中 measure 过程主要涉及三个函数，函数原型分别为

public final void measure(int widthMeasureSpec, int heightMeasureSpec)

protected final void setMeasuredDimension(int measuredWidth, int measuredHeight)

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec)

前面两个函数都是 final 类型的，不能重载，为此在 ViewGroup 派生的非抽象类中我们必须重载 onMeasure 函数，实现 measure 的原理是：假如 View 还有子 View，则 measure 子 View，直到所有的子 View 完成 measure 操作之后，再 measure 自己。ViewGroup 中提供的 measureChild 或 measureChildWithMargins 就是实现这个功能的。

在具体介绍测量原理之前还是先了解些基础知识，即 measure 函数的参数由类 measureSpec 的 makeMeasureSpec 函数方法生成的一个 32 位整数，该整数的高两位表示模式（Mode），低 30 位则是具体的尺寸大小（specSize）。

MeasureSpec 有三种模式分别是 UNSPECIFIED, EXACTLY 和 AT_MOST，各表示的意义如下

如果是 AT_MOST，specSize 代表的是最大可获得的尺寸；

如果是 EXACTLY，specSize 代表的是精确的尺寸；

如果是 UNSPECIFIED，对于控件尺寸来说，没有任何参考意义。

那么对于一个 View 的上述 Mode 和 specSize 值默认是怎么获取的呢，他们是根据 View 的 LayoutParams 参数来获取的：

参数为 fill_parent/match_parent 时，Mode 为 EXACTLY，specSize 为剩余的所有空间；

参数为具体的数值，比如像素值（px 或 dp），Mode 为 EXACTLY，specSize 为传入的值；

参数为 wrap_content，Mode 为 AT_MOST，specSize 运行时决定。

具体测量原理

上面提供的 Mode 和 specSize 只是程序员对 View 的一个期望尺寸，最终一个 View 对象能从父视图得到多大的允许尺寸则由子视图期望尺寸和父视图能力尺寸（可提供的尺寸）两方面决定。关于期望尺寸的设定，可以通过在布局资源文件中定义的 android:layout_width 和 android:layout_height 来设定，也可以通过代码在 addView 函数调用时传入的 LayoutParams 参数来设定。父 View 的能力尺寸归根到最后就是 DecorView 尺寸，这个尺寸是全屏，由手机的分辨率决定。期望尺寸、能力尺寸和最终允许尺寸的关系，我们可以通过阅读 measureChild 或 measureChildWithMargins 都会调用的 getChildMeasureSpec 函数的源码来获得，下面简单列表说明下三者的关系

父视图能力尺寸	子视图期望尺寸	子视图最终允许尺寸
EXACTLY + Size1	EXACTLY + Size2	EXACTLY + Size2
EXACTLY + Size1	fill_parent/match_parent	EXACTLY+Size1

PDF-XChange Click to buy NOW! www.docu-track.com

EXACTLY + Size1	wrap_content	AT_MOST+Size1
AT_MOST+Size1	EXACTLY + Size2	EXACTLY+Size2
AT_MOST+Size1	fill_parent/match_parent	AT_MOST+Size1
AT_MOST+Size1	wrap_content	AT_MOST+Size1
UNSPECIFIED+Size1	EXACTLY + Size2	EXACTLY + Size2
UNSPECIFIED+Size1	fill_parent/match_parent	UNSPECIFIED+0
UNSPECIFIED+Size1	wrap_content	UNSPECIFIED+0

上述表格展现的是子视图最终允许得到的尺寸，显然 1、4、7 三项没有对 Size1 和 Size2 进行比较，所以允许尺寸是可以大于父视图的能力尺寸的，这个时候最终的视图尺寸该是多少呢？AT_MOST 和 UNSPECIFIED 的 View 又该如何决策最终的尺寸呢？

通过 Demo 演示的得到的结果，假如 Size2 比 Size1 的尺寸大，假如不使用滚动效果的话，子视图超出部分将被裁剪掉，该父视图中如果在该子视图后面还有其他视图，那么也将被裁剪掉，但是通过调用其 getVisibility 还是显示该控件是可见的，所以裁剪后控件依然是有的，只是用户没办法观察到；在使用滚动效果的情况下，就能将原本被裁剪掉的控件通过滚动显示出来。

对于第二个问题，根据源码 View 的 OnMeasure 函数调用的 getDefaultSize 函数获知，默认情况下，控件都有一个最小尺寸，该值可以通过设置 android:minHeight 和 android:minWidth 来设置(无设置时缺省为 0)；在设置了背景的情况下，背景 drawable 的最小尺寸与前面设置的最小尺寸比较，两者取大者，作为控件的最小尺寸。在 UNSPECIFIED 情况下就选用这个最小尺寸，其它情况则根据允许尺寸来。不过这个是默认规则，通过 demo 发现，TextView 在 AT_MOST+Size 情况下，并不是以 Size 作为控件的最终尺寸，结果发现在 TextView 的源码中，重载了 onMeasure 函数，有价值的代码如下：

```
……
int widthSize = MeasureSpec.getSize(widthMeasureSpec);
int heightSize = MeasureSpec.getSize(heightMeasureSpec);
……
if (widthMode == MeasureSpec.AT_MOST) {
    width = Math.min(widthSize, width);
}
……
if (heightMode == MeasureSpec.AT_MOST) {
    height = Math.min(desired, heightSize);
}
……
```

至于其中的 width 和 desired 值，感兴趣的同学可以具体关注下。虽然 FrameWork 提供了视图默认的尺寸计算规则，但是最终的视图布局大小可以重载 onMeasure 函数来修改计算规则，当然也可以不计算直接通过 setMeasuredDimension 来设置（需要注意的是，如果通过 setMeasuredDimension 的同时还要调用父类的 onMeasure 函数，那么在调用父类函数之前调用的 setMeasuredDimension 会无效果）。

layout 过程分析

上述 measure 过程达到的结果是设定了视图的高和宽，layout 过程的作用就是设定视图

PDF-XChange Click to buy NOW! www.docu-track.com

在父视图中的四个点（分别对应 View 四个成员变量 mLeft，mTop，mLeft，mBottom）。同样 layout 也是被 fianl 修饰符限定为不能重载，不过在 ViewGroup 中 onLayout 函数被 abstract 修饰，即所有派生自 ViewGroup 的类必须实现 onLayout 函数，从而实现对其包含的所有子视图的布局设定。

那么上述的 measure 结果与 layout 有什么关系，截取 ViewRoot 和 FrameLayout 两个类中 onLayout 函数的部分代码如下：

```
//ViewRoot 的 performTraversals 函数 measure 之后对 layout 的调用代码
host.layout(0, 0, host.mMeasuredWidth, host.mMeasuredHeight);
//FrameLayou 的 onLayout 函数部分源码
    protected void onLayout(boolean changed, int left, int top, int right, int bottom) {
        final int count = getChildCount();
        ……
        for (int i = 0; i < count; i++) {
            final View child = getChildAt(i);
            if (child.getVisibility() != GONE) {
                final LayoutParams lp = (LayoutParams) child.getLayoutParams();
                final int width = child.getMeasuredWidth();
                final int height = child.getMeasuredHeight();
                int childLeft = parentLeft;
                int childTop = parentTop;
                final int gravity = lp.gravity;

                if (gravity != -1) {
                    final int horizontalGravity = gravity & Gravity.HORIZONTAL_GRAVITY_MASK;
                    final int verticalGravity = gravity & Gravity.VERTICAL_GRAVITY_MASK;

                    switch (horizontalGravity) {
                        case Gravity.LEFT:
                            childLeft = parentLeft + lp.leftMargin;
                            break;
                        case Gravity.CENTER_HORIZONTAL:
                            childLeft = parentLeft + (parentRight - parentLeft - width) / 2 + lp.leftMargin - lp.rightMargin;
                            break;
                        case Gravity.RIGHT:
                            childLeft = parentRight - width - lp.rightMargin;
                            break;
                        default:
                            childLeft = parentLeft + lp.leftMargin;
                    }

                    switch (verticalGravity) {
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
                                case Gravity.TOP:
                                    childTop = parentTop + lp.topMargin;
                                    break;
                                case Gravity.CENTER_VERTICAL:
                                    childTop = parentTop + (parentBottom - parentTop -
height) / 2 + lp.topMargin - lp.bottomMargin;
                                    break;
                                case Gravity.BOTTOM:
                                    childTop = parentBottom - height - lp.bottomMargin;
                                    break;
                                default:
                                    childTop = parentTop + lp.topMargin;
                            }
                        }

                        child.layout(childLeft, childTop, childLeft + width, childTop + height);
                    }
                }
            }
```

从代码显然可知具体 layout 布局时，就是根据 measure 过程设置的高和宽，结合视图在父视图中的起始位置，再外加视图的 layoutgravity 属性来设置四个点的具体位置（在 LinearLayout 中还会增加对 layoutweight 属性的考虑）。这个过程相对没有 measure 那么复杂。

需要注意的是在自定义组合控件的时候，我们可以根据需要不用或只用部分 measure 过程计算得到的尺寸，具体可以看下之前做的下拉刷新控件直接重载的 onLayout 函数：

```
    protected void onLayout(boolean changed, int left, int top, int right, int bottom) {
        if (getChildCount() > 2) {
            throw new IllegalStateException("NPullToFreshContainer can host only two direct
child");
        }

        View headView = getChildAt(0);
        View contentView = getChildAt(1);
        if(headView != null){
            headView.layout(0, -HEAD_VIEW_HEIGHT + mTatolScroll,
getMeasuredWidth(), mTatolScroll);// mTatolScroll 是下拉的位移值
        }

        if(contentView != null){
            contentView.layout(0, mTatolScroll, getMeasuredWidth(), getMeasuredHeight());
        }

        if (mFirstLayout) {
            HEAD_VIEW_HEIGHT = getChildAt(0).getMeasuredHeight();
            mFirstLayout = false;
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
        }
    }
```

draw 过程分析

View 的 Draw 过程，其实相对来说应该比 measure 过程更为复杂，正因为其很复杂，所以 android 框架层已经将 draw 过程考虑得相当周全，虽然 view 类的 Draw 函数没用 final 修饰，但是我们自定义的 View，一般也不需要去重载实现它，自己目前也没有自己去 draw 过界面，对整个过程，只能偷别人整理的逻辑，结合源码浏览了一下，在这里做个标注。

draw()方法实现的功能流程如下：

1、调用 background.draw(canvas)绘制该 View 的背景

2、调用 onDraw(canvas)方法绘制视图本身(每个 View 都需要重载该方法，ViewGroup 不需要实现该方法)

3、调用 dispatchDraw(canvas)方法绘制子视图(ViewGroup 类已经为我们重写了 dispatchDraw ()的功能实现，其内部会遍历每个子视图，调用 drawChild()去重新回调每个子视图的 draw()方法)

4、调用 onDrawScrollBars(canvas)绘制滚动条

为了说明 measure、layout 和 draw 过程的连续性，摘得 draw 中的源码如下

```
……
if (mBackgroundSizeChanged) {
        background.setBounds(0, 0,   mRight - mLeft, mBottom - mTop);
        mBackgroundSizeChanged = false;
}
……
```

上述的 mLeft，mTop，mLeft，mBottom 就是我们在 layout 是设定的结果值，这里之所以要用减法获取高宽尺寸而不用 measure 过程设定的 mMeasuredHeight 和 mMeasureWidth，个人感觉就是因为我们可以在代码中通过直接调用 View 的 layout 函数避开 measure 测算结果而导致真实高宽不等于 mMeasuredHeight 和 mMeasureWidth 这种情况。

上述代码中的 mBackgroundSizeChanged 是个私有成员变量，源码中只能在 View 的 onScrollChanged(int l, int t, int oldl, int oldt) 、layout 过程调用的 setFrame(int left, int top, int right, int bottom) 和 setBackgroundDrawable(Drawable d)这三个函数中对其修改为 true。

到这里，除了具体的绘制外，我们对从 Activity 到 View 的绘制流程应该比较清楚了。

本文除了参阅源码，发现下面两篇博文帮助很大，有兴趣可以详细阅读

http://blog.csdn.net/qinjuning/article/details/7110211

http://www.cnblogs.com/bastard/archive/2012/04/10/2440577.html

ViewDemo.rar

验证 View measure 现象的 demo 见 rar 附件