目录

1. Android 属性动画特性

跟早期的 View 动画相比，属性动画具有以下优点：

(1) **对任意对象的属性执行动画操作**：属性动画允许对任意对象的属性执行动画操作，因为属性动画的性质是通过反射实现的。

(2)**可改变背景颜色**。

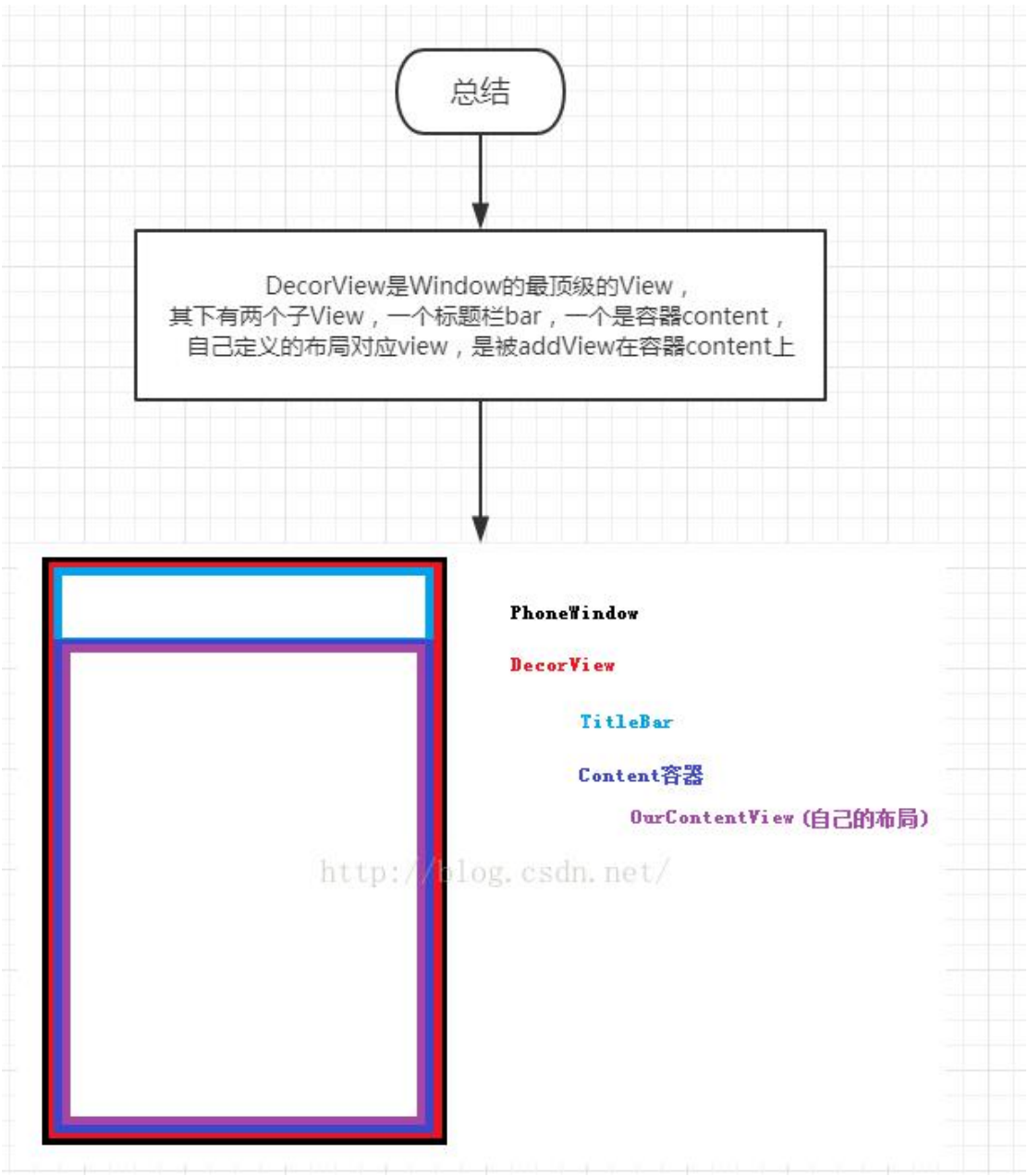
(3)**真正改变 View 本身**：因为是通过反射改变其属性，并刷新，如改变 width，他会搜索 getWidth(),反射获取，再通过进行某种计算，将值通过 setWidth()设置进去并更新。

2. Android 动画框架实现原理(源码)

RootView 只有一个孩子就是 DecorView，这里整个 View Tree 都是 DecorView 的子 View。

在 DecorView 中，标题窗口（TitleBar）以下部分的 FrameLayout(Content 容器）就是为了让程序员通过 setContentView 来设置用户需要的窗口内容。

因为整个 View 的布局就是一棵树，所以绘制的时候也是按照树形结构遍历来让每个 View 进行绘制。



递归的绘制整个窗口需要按顺序执行以下几个步骤：

1. **绘制背景**；

2. **如果需要保存画布（canvas）的层，为淡入或淡出做准备**；

3. **绘制 View 本身的内容**：通过调用 View.onDraw(canvas) 函数实现，通过这个我们应该能看出来 onDraw 函数重载的重要性，onDraw 函数中绘制线条 / 圆 / 文字等功能会调用 Canvas 中对应

的功能。（每个 View 都需要重载该方法，ViewGroup 不需要实现该方法）

4. **如果该 view 是 ViewGroup，则需要绘制自己的孩子**：通过 dispatchDraw(canvas) 实现，参看 ViewGroup.Java 中的代码可知，dispatchDraw() -> drawChild() -> child.draw(canvas)

. 这样的调用过程被用来保证每个子 View 的 draw 函数都被调用，通过这种递归调用从而让整个 View 树中的所有 View 的内容都得到绘制。（在调用每个子 View 的 draw 函数之前，需要绘制的 View 的绘制位置是在 Canvas 通过 translate 函数调用来进行切换的，窗口中的所有 View 是共用一个 Canvas 对象。）

5. **如果需要，我们可以绘制淡入淡出相关的内容，并恢复保存的画布所在的层（layer）**

6. **绘制修饰的内容（例如滚动条）**:这个可知要实现滚动条效果并不需要 ScrollView,可以在 View 中完成的,比如通过 ParentView 中设置 ChildView 的画布来调整 Canvas，进而实现动画效果(安卓动画就是通过父 View 来不断调整子 View 的画不坐标系来实现的)

3. Bitmap 对象的理解(源码)

1. Bitmap 在 Android 中指的是一张图片。
2. 通过 BitmapFactory 类提供的四类方法：

decodeFile（从文件系统加载出一个 Bitmap 对象）

decodeResource（从资源中加载出一个 Bitmap 对象）

decodeStream（从输入流中加载出一个 Bitmap 对象）

decodeByteArray（从字节数组中加载出一个 Bitmap 对象）

其中 decodeFile，decodeResource 又间接调用了 decodeStream 方法，这四类方法最终是在 Android 的底层实现的，对应着 BitmapFactory 类的几个 native 方法。

3. BitmapFactory.Options 的参数

①inSampleSize 参数

上述四类方法都支持 BitmapFactory.Options 参数，而 Bitmap 的按一定采样率进行缩放就是通过 BitmapFactory.Options 参数实现的，主要用到了 inSampleSize 参数，即采样率。通过对 inSampleSize 的设置，对图片的像素的高和宽进行缩放。

当 inSampleSize=1，即采样后的图片大小为图片的原始大小。小于 1，也按照 1 来计算。
当 inSampleSize>1，即采样后的图片将会缩小，缩放比例为 1/(inSampleSize 的二次方)。

关于 inSampleSize 取值的注意事项：
通常是根据图片宽高实际的大小/需要的宽高大小，分别计算出宽和

高的缩放比。但应该取其中最小的缩放比，避免缩放图片太小，到达指定控件中不能铺满，需要拉伸从而导致模糊。

②inJustDecodeBounds 参数

我们需要获取加载的图片的宽高信息，然后交给 inSampleSize 参数选择缩放比缩放。

想先不加载图片却能获得图片的宽高信息，可通过 inJustDecodeBounds=true，然后加载图片就可以实现只解析图片的宽高信息，并不会真正的加载图片，所以这个操作是轻量级的。

当获取了宽高信息，计算出缩放比后，然后在将 inJustDecodeBounds=false,再重新加载图片，就可以加载缩放后的图片。

4. 高效加载 Bitmap 的流程

①将 BitmapFactory.Options 的 inJustDecodeBounds 参数设为 true 并加载图片。这里要设置 Options.inJustDecodeBounds=true,这时候 decode 的 bitmap 为 null,只是把图片的宽高放在 Options 里，

②从 BitmapFactory.Options 中取出图片的原始宽高信息，它们对应于 outWidth 和 outHeight 参数。

③根据采样率的规则并结合目标 View 的所需大小计算出采样率 inSampleSize。

④将 BitmapFactory.Options 的 inJustDecodeBounds 参数设为 false，然后重新加载图片。

5. Bitmap 高效加载的代码实现

```
     public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,
int reqWidth, int reqHeight){
            BitmapFactory.Options options = new BitmapFactory.Options();
            options.inJustDecodeBounds = true;
            //加载图片
            BitmapFactory.decodeResource(res,resId,options);
            //计算缩放比
            options.inSampleSize =
calculateInSampleSize(options,reqHeight,reqWidth);
            //重新加载图片
            options.inJustDecodeBounds =false;
            return BitmapFactory.decodeResource(res,resId,options);
        }

        private static int calculateInSampleSize(BitmapFactory.Options options,
int reqHeight, int reqWidth) {
            int height = options.outHeight;
            int width = options.outWidth;
            int inSampleSize = 1;
            if(height>reqHeight||width>reqWidth){
                int halfHeight = height/2;
                int halfWidth = width/2;
                //计算缩放比，是 2 的指数

while((halfHeight/inSampleSize)>=reqHeight&&(halfWidth/inSampleSize)>=reqWidth)
{
                    inSampleSize*=2;
                }
            }
            return inSampleSize;

          }
```

这个时候就可以通过如下方式高效加载图片：

mImageView.setImageBitmap(decodeSampledBitmapFromResource(getResources(),R.mipmap.ic_launcher,100,100);

除了 BitmapFactory 的 decodeResource 方法，其他方法也可以类似实现。

4. 自定义控件原理（原理）

一共有四个点

（一）首先是 Android 控件架构

每一个 Activity 包含一个 Window 对象，DecorView 作为整个应用窗口的根 View（根 view 会自动设置成覆盖全屏）

DecorView 其下包含 TitleView 和 ContentView，这里 ContentView 就是 id 为 content 的 FrameLayout，我们平时写的 layout 就是天生包裹着一层 FrameLayout。

在代码中，Activity 的 OnCreate 中调用 setContentView（）方法后，ActivityManagerService 会回调 onResume（）方法，

此时系统会将整个 DecorView 添加到 PhoneWindow 中，并让其显示出来，由此可以见，为了让视图尽快显示，尽量减轻 OnCreate 操作。

（二）然后自定义 View 类型

通常情况下自定义 View 有三种方法

1）对现有控件进行扩展

2）通过组合来实现新控件

3）重写 View 实现全新的控件

（三）接着是 View 中比较重要的回调方法

① onFinishInflate（）：从 XML 加载组件后回调。

② onSizeChanged（）：组件大小改变时回调，一般是准确的、最终测量的值

③ onMeasure（）：

分为 View 测量，以及 ViewGroup 测量

View 的测量

要想绘制一个 View，我们必须先对其进行测量，知道大小。View 的 onMeasure()方法

```
        protected void onMeasure(int widthMeasureSpec, int
heightMeasureSpec) {
            super.onMeasure(widthMeasureSpec,
heightMeasureSpec);
        }
```

在 onMeasure 中向向父 onMeasure（）方法中传递 MeasureSpec 类型的数据，它是一个 32 位的 int 值，高 2 位是测量模式，低 30 位就是测量大小。

测量模式分为下面三个。

1、EXACTLY

精确模式，match_parant 或者具体数据 100dp。

2、AT_MOST

最大值模式，此时控件尺寸不超过父控件允许的最大尺寸即可，wrap_content 就是此模式。

3、UNSPECIFIED

View 想多大就多大。

View 默认提供了 EXACTLY 模式，想要支持 AT_MOST 和 UNSPECIFIED，必须自己自定义了。

父类的 onMeasure（）会将传进来的 widthMeasureSpec 以及 heightMeasureSpec 通过 getDefaultSize（）进行计算，并最终调用

setMeasuredDimension()这个方法，将测量的宽高设置进去从而完成测量工作设值工作

ViewGroup 的测量

ViewGroup 可以放置 n 个 View，

当 ViewGroup 时 wrap_content 模式，那么其大小是通过 ViewGroup 遍历所有的子 View，来获取 View 的大小，从而决定自身的大小，

而在其他模式下，会通过具体的值来自定自身的大小。ViewGroup 遍历所有的 View 会调用所有的 View 的 onMeasure()方法来获取测量结果，

④ **onLayout（）：**绘制位置

在我们自定义 ViewGroup 的时候，一般都要重写 onLayout()方法控制子 View 显示位置的逻辑

当子 View 测量完毕之后，，就需要将子 View 放在合适的地方，这部分是由 onLayout()来进行的，

⑤ **onDraw（）：**

分为 View 的绘制以及 ViewGroup 的绘制

View 的绘制

在 draw（）方法中通过 canva 来进行绘制

如：Canvas canvas = new Canvas(Bitmap);//绘制直线

ViewGroup 的绘制

ViewGroup 是通过 dispathDraw 来绘制其子 View 的，其过程也是通过遍历所有子 View，然后调用子 View 的绘制方法完成绘制的

⑥ onTouchEvent（）：监听触摸事件

（四）事件拦截机制分析

对于 ViewGroup 一般有一下三个方法

dispatchTouchEvent（）

onInterceptTouchEvent（）

onTouchEvent（）

对于 View 一般重写以下两个方法：

dispatchTouchEvent（）

onTouchEvent（）

后序了解深入的做解答

（了解不深）

假设一个场景：

BOSS -- ViewGroupA 最外层的

项目负责人 -- ViewGroupB 中间层的

程序猿 -- View 最底层

对比下看到 ViewGroup 比 View 多了一个方法 onInterceptTouchEvent，事件拦截的核心方法。

什么都不做，正常情况是：

事件传递顺序：BOSS - 项目负责人 - 程序猿，传递过程中会先执行 dispatchTouchEvent（事件分发），在执行 onInterceptTouchEvent（拦截）。

事件处理顺序：程序猿 - 项目负责人 - BOSS 。

事件处理调用的是 onTouchEvent。

onInterceptTouchEvent 返回 True 则表示拦截，不让下分发，自己搞起；返回 false 则表示分发到下一层中，依次传递。
onTouchEvent 返回 true 表示已处理，不用上层处理；false 表示上交上层处理。

简单来说，BOSS 自己想搞起，就把 onInterceptTouchEvent 返回 true，那么就没有项目负责人和程序猿鸟事了；
BOSS 不想搞，就把 onInterceptTouchEvent 返回 false，则分发给项目负责人，

如果项目负责人想搞，则 onInterceptTouchEvent 返回 true，那就没有程序猿鸟事；如果项目负责人也不想搞，则 onInterceptTouchEvent 返回 tfalse，那么苦逼的程序猿只能搞起了。

当程序猿加班加点搞定了，不想提交上层，则 onTouchEvent 返回 true 即可，否则提交给项目负责人，依次类推；

5. 自定义 View 如何提供获取 View 属性的接口？（原理）

自定义属性的实现流程：

(1) **在 values 目录下定义一个 attrs.xml**：在 res/values/attr.xml 中定义相关属性。

(2) **在对应的类文件里生成某些组件**：在对应类的构造函数中通过 obtainStyledAttributes（）方法获得自定义属性的相关值

(3) **在 layout 布局文件里为这些属性赋值**：在布局中添加为该自定义组件设置一个命名空间，并且相关属性赋值

[xmlns:myandroid= "http://schemas.android.com/apk/res/cn.com.androidtest"]

也可以采用命名空间写法:xmlns: 空间名 = "http://schemas.android.com/apk/res/自定义组件所在顶级包名"

**

6. View 刷新机制（原理）

刷新原理:

在 Android 的布局体系中，父 View 负责刷新、布局显示子 View；而当子 View 需要刷新时，则是通知父 View 来完成。

这种处理逻辑在 View 的代码中明确的表现出来 :子 View 执行 invalidate（）方法时，首先找到自己的父 View（），将 AttachInfo 中保存的信息告诉父 View 刷新自己，父 View 会将参数代入 invalidateChild（）执行

View 的父子关系的建立分为两种情况：

1) View 加入 ViewGroup 中，执行 addViewInner（）方法

2)DecorView 注册给 WindowManagerImpl 时，产生一个 ViewRoot 作为其父 View。

对于 AttachInfo：

AttachInfo 是在 View 第一次 attach 到 Window 时，ViewRoot 传给自己的子 View 的。这个 AttachInfo 之后，会顺着布局体系一直传递到最底层的 View。

并且在新的 View 被加入 ViewGroup 时 ,也会将该 AttachInfo 传给加入的 View

在 invalidate 中，调用父 View 的 invalidateChild，这是一个从第向上回溯的过程，每一层的父 View 都将自己的显示区域与传入的刷新 Rect 做交集。

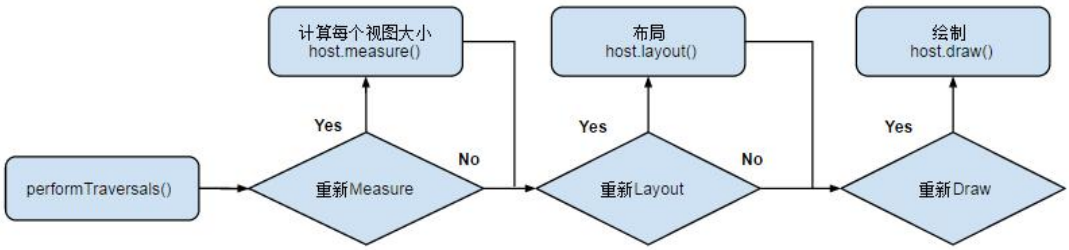
另外：

Invalidate()方法不能放在线程中，因为当 Invalidate()被调用的时候，View 的 OnDraw()就会被调用，Invalidate()必须是在 UI 线程中被调用所以需要把

Invalidate()方法放在 Handler 中。在 MyThread 中只需要在规定时间内发送一个 Message 给 handler，当 Handler 接收到消息就调用 Invalidate()方法。

postInvalidate()可以在非 UI 线程中调用，通知 UI 线程重绘。方法就可以放在线程中做处理，就不需要 Handler。 postInvalidate 底层是也是使用了 Handler,这就是为什么能在子线程更新 UI 的原因。同时 postInvalidate 可以指定一个延迟时间

7. View 绘制流程（原理）




```
*   测量      摆放   绘制
 measure   ->  layout  ->  draw
     |           |          |
 onMeasure -> onLayout -> onDraw 重写这些方法，实现自定义控件

     View流程
     onMeasure() (在这个方法里指定自己的宽高) -> onDraw() (绘制自己的内容)

     ViewGroup流程
     onMeasure() (指定自己的宽高，所有子View的宽高)-> onLayout() (摆放所有子View) -> onDraw() (绘制内容)
*
```

（一）首先可以自定义一些自己的属性，在 res/values/attrs.xml

里面定义，然后在 layout 中使用，在 View 中通过 context.obtainS

tyledAttributes(attrs,R.styleable.自定义属性的名字)进行获取。

（二）**然后再测量 onMeasure**,一般通过他的三个模式(EXACTLY,AT

_MODE,, UNSPECIFIED)进行测量，调用 setMeasuredDimension 进行传入设置的值。

如果是 ViewGroup，为整个 View 树计算实际的大小，即设置实际的宽高（每个 View 的控件的实际宽高都是由父视图和本身视图决定的。）

当 ViewGroup 时 wrap_content 模式，那么其大小是通过 ViewGroup 遍历所有的子 View，来获取 View 的大小，从而决定自身的大小，

而在其他模式下,会通过具体的值来自定自身的大小。ViewGroup 遍历所有的 View 会调用所有的 View 的 onMeasure()方法来获取测量结果

（三）**接着如果是 ViewGroup 的话我们还需要设置下子 View 的位置**，一般是通过 requestLayout 去触发 onLayout 的方法的。

（四）**最后在 onDraw 里面**。

绘制的流程如下

1) 首先绘制该 View 的背景

2) 调用 onDraw()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) 如果该 View 是 ViewGroup，调用 dispatchDraw ()方法绘制子视图

4) 绘制控件，**通过 Canvas 的一些方法进行绘制**

（五）**如果需要进行触摸事件的话，一般需要有实现 onTouchEvent 事件**，注意，如果需要多点触摸，需要实现 ACTION_POINTER_DOWN 和 ACTION_POINTER_UP 进行处理。

8. 计算一个 view 的嵌套层级（原理）

循环追寻父类父类，看看有多少，便有多少嵌套层级

```
while (view.getParents() != null) {
        count++;
        view = view.getParents();
}
```

9. 自定义 View 如何考虑机型适配(源码)

布局类：

1. 合理使用 warp_content，match_parent.
2. 尽可能的是使用 RelativeLayout
3. 引入 android 的百分比布局。
4. 针对不同的机型，使用不同的布局文件放在对应的目录下，android 会自动匹配。

Icon 类：

1. 尽量使用 svg 转换而成 xml。
2. 切图的时候切大分辨率的图，应用到布局当中。在小分辨率的手机上也会有很好的显示效果。
3. 使用与密度无关的像素单位 dp，sp

10. 如何优化自定义 View(源码)

(1)**减少不必要的代码**：对于频繁调用的方法，需要尽量减少不必要的代码。

(2)**不在 onDraw 中做内存分配的事**：先从 onDraw 开始，需要特别注意不应该在这里做内存分配的事情，因为它会导致 GC，从而

导致卡顿。在初始化或者动画间隙期间做分配内存的动作。不要在动画正在执行的时候做内存分配的事情。

(3)**减少 onDraw 被调用的次数**：大多数时候导致 onDraw 都是因为调用了 invalidate().因此请尽量减少调用 invaildate()的次数。如果可能的话，尽量调用含有 4 个参数的 invalidate()方法而不是没有参数的 invalidate()。没有参数的 invalidate 会强制重绘整个 view。

(4)**减少 layout 的次数**：一个非常耗时的操作是请求 layout。任何时候执行 requestLayout()，会使得 Android UI 系统去遍历整个View 的层级来计算出每一个 view 的大小。如果找到有冲突的值，它会需要重新计算好几次。

(5)**选用扁平化的 View**：另外需要尽量保持 View 的层级是扁平化的（去除冗余、厚重和繁杂的装饰效果），这样对提高效率很有帮助。

(6)**复杂的 UI 使用 ViewGroup**：如果你有一个复杂的 UI，你应该考虑写一个自定义的 ViewGroup 来执行他的 layout 操作。（与内置的view 不同,自定义的 view 可以使得程序仅仅测量这一部分，这避免了遍历整个 view 的层级结构来计算大小。）继承 ViewGroup 作为自定义 view 的一部分，有子 views，但是它从来不测量它们。而是根据他自身的 layout 法则，直接设置它们的大小。

11. 自定义 View 注意事项

① 让 view 支持 wrap_content

这是因为直接继承 view 或者 viewgroup 的控件,如果不在 onmeasure 中做特殊的处理，那么在布局中 wrap_content 就不能达到预期的效果。

② 让 view 支持 padding

这是因为直接继承 view 的控件,如果不在 draw 中处理 padding 事件，那么 padding 属性是无法起作用的。另外，直接继承字 viewgroup 的空间需要在 measure 和 layout 中考虑 padding 和 margin。

3、尽量避免在 view 中使用 handler，因为使用 handler 没必要，因为 view 内部提供了 post 方法，可以取代 handler。

③ view 中如果有线程或者动画，需要及时停止，参考 view#ondetachedfromwindow

如果有线程或者动画需要停止时，那么 ondetachedfromwindow 是一个很好的时机，当包含此 view 的 activity 退出或者此 viewremove 时，此方法会被调用，和此相对应的方法是 onattchedwindow，当包含 view 的 activity 启动时，onattchedwindow 方法会被调用，如果不及时停止将会造成内存泄漏。

④ 带有滑动嵌套时，需要处理好滑动冲突。

12. 封装 View 的时候怎么知道 view 的大小(源码)

https://blog.csdn.net/fwt336/article/details/52979876

onMeasure()：决定 View 的大小

分两种方法：

1. 如果测量的是一个 View：可以 getDefaultSize()方法来获取测量宽高。

setMeasuredDimension(

getDefaultSize

（getSuggestedMinimumHeight(),heightMeasureSpec））

在源码中 onMeasure（）只调用了 setMeasuredDimension()来存储测量的宽，高值，其两个宽高参数都是通过 getDefaultSize（） 获得的

getDefaultSize（） 中有两个参数，第一个是 getSuggestedMinimumWidth()/getSuggestedMinimumHeight()， 第二个是 widthMeasureSpec/heightMeasureSpec。

getDefaultSize（）返回的结果就是用 MeasureSpec 中模式（mode）、大小（size）、

getSuggestedMinimumWidth()/getSuggestedMinimumHeight()大小三者进行测量后的大小

其中

getSuggestedMinimumWidth()/getSuggestedMinimumHeight()，就是获取最小宽度作为默认值，原理是先查看 View 有没有设背景来设置参数，如无则默认为 0

2. 如果是多个View或者ViewGroup嵌套我们就需要循环遍历视图中的所有 View

在子类中重写的 onMeasure：

调用 measureChildren（）来设置 widthMeasureSpec 以及 heightMeasureSpec，然后再执行 super.onMeasure(widthMeasureSpec,heightMeasureSpec);

measureChildren() ->measureChild（）->getChildMeasureSpec（）

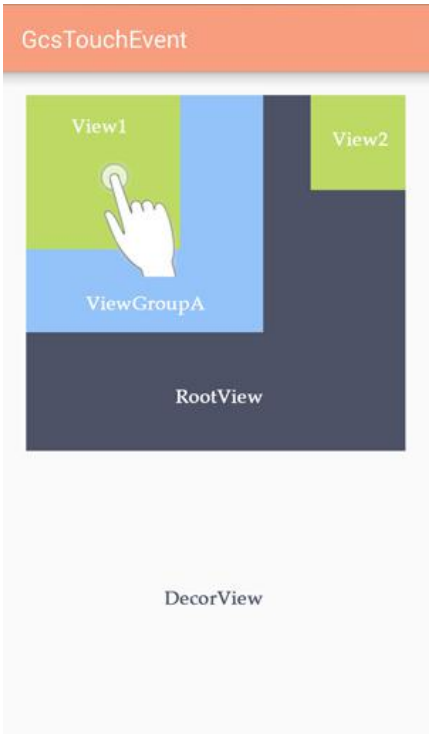
执行 measureChildren()就是遍历所有的子 View，如果 View 的状态不是 GONE 就调用 measureChild（）去进行下一步的测量（measureChild()先测量单个视图，将宽高和 padding 加在一起后交给 getChildMeasureSpec（）去获得最终的测量值 ，再将计算好的宽高详细测量值传入 child 的 measure 方法，完成最后的测量）

getChildMeasureSpec()结合父 view 的 MeasureSpec 与子 view 的 LayoutParams 信息去找到最好的结果 （也就是说子 view 的确切大小由两方面共同决定：1.父 view 的 MeasureSpec 2.子 view 的 LayoutParams 属性）

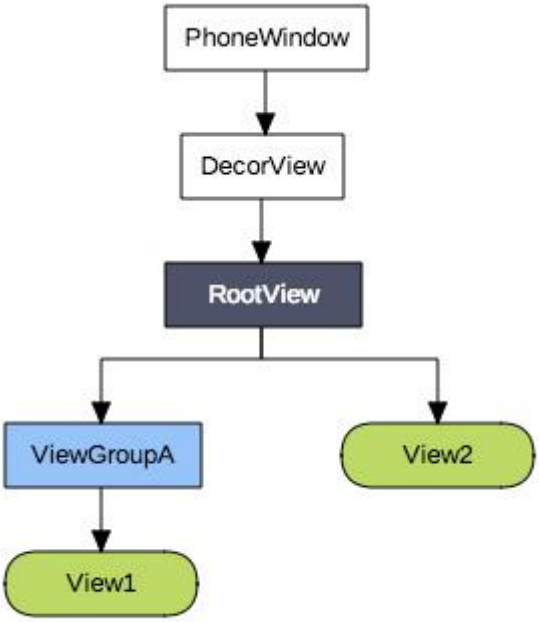
13. 自定义 View 的事件(源码)

14. 请描述一下 View 事件传递分发机制（原理）

15. Touch 事件传递流程（原理）



View 结构:



一、事件流程:

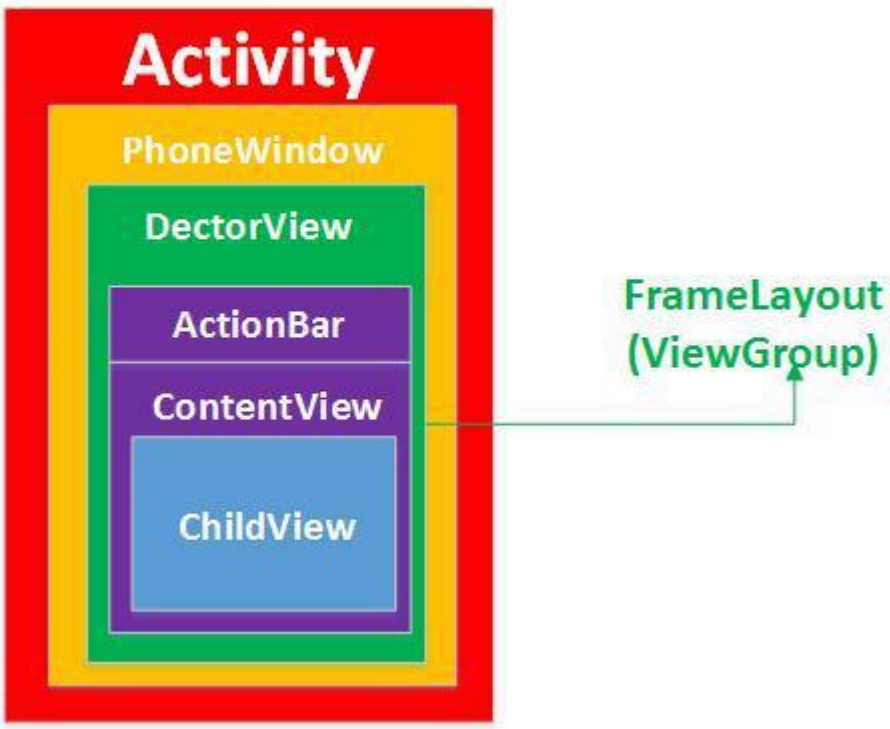
一般我们显示一个界面都是会使用一个 Activity 来显示的，Activity 中会有一个 PhoneWindow。

PhoneWindow 是 Window 的实现类，Window 是一个抽象类，是所有视图的最顶层容器，视图的外观和行为都归他管，但因是抽象类不能使用，因此 PhoneWindow 的权利非常大。

PhoneWindow 类有个 mDecor 的 DectorView 全局变量，用来对界面的 View 元素进行修饰,PhoneWindow 传递信息给 DecorView ，再将其传递给下面的 View

DecorView 是 PhoneWindow 的一个内部类，专门为 PhoneWindow 服务的，接受 PhoneWindow 的指令通过它传递给下面的 View，也可将下面 View 的信息通过它回传给 PhoneWindow。DectorView 继承自 FrameLayout，FrameLayout 就是一个 ViewGroup，可以有多个子 ViewGroup 和子 View。DectorView 用来修饰 ActionBar、ContentView(Activity.setContentView)；ContentView 里面包含了用户自定义的一些子 Layout。

同时 DectorView 用来修饰 ActionBar、ContentView(Activity.setContentView)；ContentView 里面包含了用户自定义的一些子 Layout。



在这流程中：

Activity、ViewGroup、View 都有事件分发（dispathTouchEvent）、事件消费（onTouchEvent）功能

事件拦截（onInterceptTouchEvent）只有 ViewGroup 才有，因为 Acitvity 作为原始的事件分发者，拦截了就导致整个屏幕都无法响应事件，同时 View 作为传递的最末端，根本没有必要进行事件拦截（要么消费，要么不处理进行回传）

事件分发、拦截与消费

下表省略了 PhoneWidow 和 DecorView。

> √ 表示有该方法。
>
> X 表示没有该方法。

类型	相关方法	Activity	ViewGroup	View
事件分发	dispatchTouchEvent	√	√	√
事件拦截	onInterceptTouchEvent	X	√	X
事件消费	onTouchEvent	√	√	√

这个三个方法均有一个 boolean(布尔) 类型的返回值，通过返回 true 和 false 来控制事件传递的流程。

PS: 从上表可以看到 Activity 和 View 都是没有事件拦截的，这是因为：

> Activity 作为原始的事件分发者，如果 Activity 拦截了事件会导致整个屏幕都无法响应事件，这肯定不是我们想要的效果。
>
> View最为事件传递的最末端，要么消费掉事件，要么不处理进行回传，根本没必要进行事件拦截。

二、事件分发流程：

我们的 View 一般是是树形结构的，基于这样的结构，我们的事件可以进行有序的分发。事件分发就是责任链模式（当有多个对象均可以处理同一请求的时候，将这些对象串联成一条链，并沿着这条链传递改请求，直到有对象处理它为止）

事件收集之后最先传递给 Activity， 然后依次向下传递，大致如下：

```
    Activity —> PhoneWindow —> DecorView —> ViewGroup —> ... —> View
```

当用户触发一个 touch 事件的时候,事件首先被分发到 Activity 的 dispatchTouchEvent,activity 首先会将事件分发给 Window 处理,

调用 Window 的 superDispatchTouchEvent;

PhoneWindow 又会调用 DectorViewsuperDispatchTouchEvent 方法;

DectorView 会调用父类 FrameLayout 也就是 ViewGroup 的 dispatchTouchEvent 方法进行事件分发;

接着就会分发发到用户调用 setContentView 传入的 ViewGroup 的 dispatchTouchEvent 中。

ViewGroup 的 dispatchTouchEvent 事件分发时，会先调用 onInterceptTouchEvent 判断是否拦截事件(默认不拦截)，如果拦截了，则 mFirstTouchTarget 为 null；如果不拦截，就会查找对应的 child 进行事件处理(将事件分发给 child 进一步处理)；不论是否找到 child，都和调用 dispatchTransformedTouchEvent

如果没有任何 View 消费掉事件，那么这个事件会按照反方向回传，最终传回给 Activity，如果最后 Activity 也没有处理，本次事件才会被抛弃:

```
Activity <— PhoneWindow <— DecorView <— ViewGroup <— ... <— View
```

1) 点击时没有任何 View 消费事件：事件分发就会从 Activity 走到最末端 View，然后再回传回来
2) 点击时被最末端的 View 消费：事件分发会从 Activity 走到最末端 View，最末端 View 消费掉事件，回传 true 告诉上层 View，上层 View 无需再响应
3) 点击时被 ViewGroup 拦截：拦截后不传递给下层 View

ViewGroup 的事件分发流程又是如何的呢？

在默认的情况下 ViewGroup 事件分发流程是这样的。

1.判断自身是否需要(询问 onInterceptTouchEvent 是否拦截)，如果需要，调用自己的 onTouchEvent。

2.自身不需要或者不确定，则询问 ChildView ，一般来说是调用手指触摸位置的 ChildView。

3. 如果子 ChildView 不需要则调用自身的 onTouchEvent。

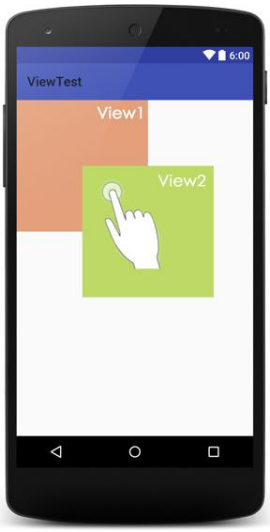
1. ViewGroup 中可能有多个 ChildView，如何判断应该分配给哪一个？

手指触摸点在哪个位置就分发给哪个 View：就是把所有的 ChildView 遍历一遍，如果手指触摸的点在 ChildView 区域内就分发给这个 View。

2. 当该点的 ChildView 有重叠时应该如何分配？

当 ChildView 重叠时，**一般会分配给显示在最上面的 ChildView**。

如何判断哪个是显示在最上面的呢？后面加载的一般会覆盖掉之前的，所以**显示在最上面的是最后加载的**。



当手指点击有重叠区域时，分如下几种情况：

哪个 View 可点击时，事件就会分配给那个 View。当有多个 View 均可点击时，会分发给最上层的 View，也就是最后加载的

三、点击事件：

对于单指触控来说，一次简单的交互流程是这样的:

手指落下(ACTION_DOWN) —> 移动(ACTION_MOVE) —> 离开(ACTION_UP)

为什么 View 会有 dispatchTouchEvent ？

View 可以注册很多事件监听器，例如：单击事件(onClick)、长按事件(onLongClick)、触摸事件(onTouch)，并且 View 自身也有 onTouchEvent 方法，那么这个时候就需要 dispatchTouchEvent，所以 View 也会有事件分发。

与 View 事件相关的各个方法调用顺序是怎样的？

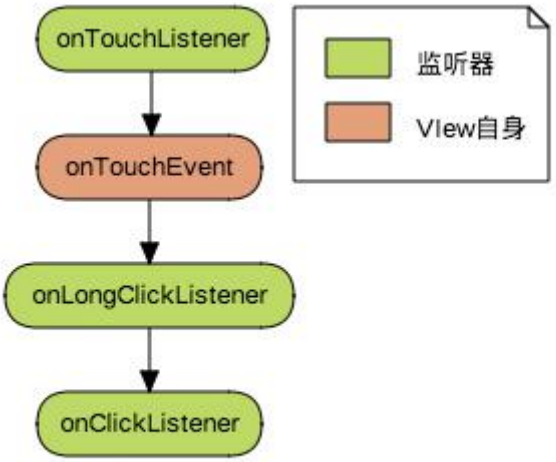
单击事件(onClickListener) 需要两个两个事件(ACTION_DOWN 和 ACTION_UP)才能触发

长按事件(onLongClickListener) 同理，也是需要长时间等待才能出结果，肯定不能排到前面，但因为不需要 ACTION_UP，应该排在 onClick 前面。(onLongClickListener > onClickListener)

触摸事件(onTouchListener) 如果用户注册了触摸事件，说明用户要自己处理触摸事件了

View 自身处理(onTouchEvent) 提供了一种默认的处理方式，如果用户已经处理好了，也就不需要了，所以应该排在 onTouchListener 后面。(onTouchListener > onTouchEvent)

所以事件的调度顺序应该是 `onTouchListener > onTouchEvent > onLongClickListener > onClickListener`。



1. 不论 View 自身是否注册点击事件，只要 View 是可点击的就会消费事件。

2. 事件是否被消费由返回值决定，true 表示消费，false 表示不消费，与是否使用了事件无关。

注意:

上面说的是可点击,可点击包括很多种情况,只要你给View注册了 onClickListener 、 onLongClickListener 、 OnContextClickListener 其中的任何一个监听器或者设置

了 android:clickable=”true” 就代表这个 View 是可点击的。另外，某些 View 默认就是可点击的，例如，Button，CheckBox 等。

给 View 注册 OnTouchListener 不会影响 View 的可点击状态。即使给 View 注册 OnTouchListener ，**只要不返回 true 就不会消费事件**。

3. ViewGroup 和 ChildView 同时注册了事件监听器(onClick 等)，哪个会执行?

事件优先给 ChildView，会被 ChildView 消费掉，ViewGroup 不会响应。

4. 所有事件都应该被同一 View 消费

在上面的例子中我们分析后可以了解到，同一次点击事件只能被一个 View 消费，这是为什呢？主要是为了防止事件响应混乱，如果再一次完整的事件中分别将不同的事件分配给了不同的 View 容易造成事件响应混乱。

安卓为了保证所有的事件都是被一个 View 消费的，对第一次的事件（ ACTION_DOWN ）进行了特殊判断，View 只有消费了 ACTION_DOWN 事件，才能接收到后续的事件(可点击控件会默认消费所有事件)，并且会将后续所有事件传递过来，不会再传递给其他

View，除非上层 View 进行了拦截。

如果上层 View 拦截了当前正在处理的事件，会收到一个 ACTION_CANCEL，表示当前事件已经结束，后续事件不会再传递过来。

四、核心要点：

1. 事件分发原理：责任链模式，事件层层传递，直到被消费。
2. View 的 `dispatchTouchEvent` 主要用于调度自身的监听器和 onTouchEvent。
3. View 的事件的调度顺序是 onTouchListener > onTouchEvent > onLongClickListener > onClickListener 。
4. 不论 View 自身是否注册点击事件，只要 View 是可点击的就会消费事件。
5. 事件是否被消费由返回值决定，true 表示消费，false 表示不消费，与是否使用了事件无关。
6. ViewGroup 中可能有多个 ChildView 时，将事件分配给包含点击位置的 ChildView。
7. ViewGroup 和 ChildView 同时注册了事件监听器(onClick 等)，由 ChildView 消费。
8. 一次触摸流程中产生事件应被同一 View 消费，全部接收或者全部拒绝。

9. 只要接受 ACTION_DOWN 就意味着接受所有的事件，拒绝 ACTION_DOWN 则不会收到后续内容。

10. 如果当前正在处理的事件被上层 View 拦截，会收到一个 ACTION_CANCEL，后续事件不会再传递过来。

16. 事件分发中的 onTouch 和 onTouchEvent 有什么区别，又该如何使用？（原理）

OnTouch 方法：

onTouch()是 OnTouchListener 接口的方法，它是获取某一个控件的触摸事件，因此使用时。通过 getAction()方法可以获取当前触摸事件的状态：

如：ACTION_DOWN：表示按下了屏幕的状态。

OnTouchEvent()方法：

onTouchEvent 是手机屏幕事件的处理方法，是获取的对屏幕的各种操作，比如向左向右滑动，点击返回按钮等等。

通过查看安卓源码中 View 对 dispatchTouchEvent 的实现，可以知道 onTouchListener(onTouch 方法在其中)的接口的执行顺序是要先于 onTouchEvent 的，onTouch 方法会先触发。

如果 onTouchListener 中的 onTouch 方法返回 true，表示此次事件已经被消费了，那 onTouchEvent 是接收不到消息的。（内置诸如 click 事件的实现等等都基于 onTouchEvent,这些事件将不会被触发）

Ruguo onTouch 方法返回 false 会接着触发 onTouchEvent。

17. View 和 ViewGroup 分别有哪些事件分发相关的回调方法（原理）

类型	相关方法	Activity	ViewGroup	View
事件分发	dispatchTouchEvent	√	√	√
事件拦截	onInterceptTouchEvent	X	√	X
事件消费	onTouchEvent	√	√	√

18. AndroidManifest 的作用与理解(源码)

<manifest/>标签层：

1. package="应用包名"

整个应用的包名。这里有个坑，当我们通过 ComponentName 来启动某个 Activity 时，所用的包名一定是这个应用的包名，而不是当前 Activity 的包名。

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

命名空间的声明，使得各种 Android 系统级的属性能让我们使用。当我们需要使用自定义属性时，可以将其修改为 res-auto，编译时会为我们自动去找到该自定义属性。

3.android:sharedUserId="android.uid.system"

将当前应用进程设置为系统级进程。

4.uses-permission

为我们的应用添加必须的权限。同时我们也可以该层声明自定义的权限。

<application/>标签层：

应用层标签，用来配置我们的 apk 的整体属性，也可以统一指定所有界面的主题。

1."android:name"、"android:icon"、"android:label"

顾名思义，用来指定应用的名称、在桌面的启动图标、应用的标签名

2."android:theme"

为当前应用的每个界面都默认设置一个主题，可以后续在 activity 标签层单独覆盖此 Theme。

3."android:allowBackup"

关闭应用程序数据的备份和恢复功能，注意该属性值默认为 true，如果你不需要你的应用被恢复导致隐私数据暴露，必须手动设置此属性。

4.android:hardwareAccelerated="true"

开启硬件加速，一般应用不推介使用。就算非要使用也最好在某个 Activity 单独开启，避免过大的内存开销。

5.android:taskAffinity

设置 Activity 任务栈的名称,可忽略。

<具体组件/>标签层：

因为</provider>、</service>在实际开发中接触得不多，这部分主要讲解 </activity> 、</receiver>标签。

关于 Activity 标签的属性，个人最觉得绕和难掌握的就是 Intent-filter 的匹配规则了，每次使用错了都要去查资料修改，所以这边总结得尽可能仔细。

</activity>

1. android:configChanges

当我们的界面大小，方向，字体等 config 参数改变时，我们的 Activity 就会重新执行 onCreate 的生命周期。而当我们设置此属性后，就可以强制让 Activity 不重新启动，而是只会调用一次 onConfigurationChanged 方法，所以我们可以在这里做一些相关参数改变的操作。

2. "android.intent.category.LAUNCHER"、"android.intent.action.MAIN"

这两个属性共同将当前 Activity 声明为了我们应用的入口，将应用注册至系统的应用列表中，缺一不可。

注：这里还有一点需要注意，如果希望我们的应用有多个入口，每个入口能进入到 app 的不同 Activity 中时，光设置这两个属性还不够，还要为它指定一个进程和启动模式。

android:process=".otherProcess"

android:launchMode ="singleInstance"

3. android:exported="true"

将当前组件暴露给外部。属性决定它是否可以被另一个Application的组件启动。

当我们通过intent去隐式调用一个Activity时，需要同时匹配注册activity中的action、category、data才能正常启动，而这三个属性的匹配规则也略有不同。

1. action

action是最简单的匹配项，我们将其理解为一个区分大小写的字符串即可。

一般用来代表某一种特定的动作，隐式调用时intent必须setAction。一个过滤器中可以有多个action属性，只要我们的itent和其中任意一项equal则就算匹配成功。

2. category

category属性也是一个字符串，匹配时也必须和过滤器中定义的值相同。

当我们没有为intent设置addCategory时，系统为帮我们默认添加一个值为"android.intent.category.DEFAULT"的category。

反过来说，如果我们需要我们自己写的 Activity 能接受隐式 intent 启动，我们就必须在它的过滤器中添加"android.intent.category.DEFAULT"，否则无法成功启动。

3.data

data 比较复杂，幸运地是我们几乎用不到它。

额外扩展一些关于 activity 的属性：

<meta-data/>标签：

标签<meta-data>是提供组件额外的数据用的，它本身是一个键值对，写在清单文件中之后，可以在代码中获取。

android:excludeFromRecents="true"

设置为 true 后，当用户按了“最近任务列表”时候，该 activity 不会出现在最近任务列表中，可达到隐藏应用的目的。

</receiver>

关于 receiver，广播接收器，也可以给他设置接收权限。一个 permission 问题：

```
<receiver
 android:name="com.android.settings.AliAgeModeReceiver"
 android:permission="com.android.settings.permission.SWITH_SETTING">
```

```
        <intent-filter>
            <action android:name="com.android.settings.action.SWITH_AGED_MODE"/>
        </intent-filter>
    </receiver>
```

可在 receiver 标签中添加权限， 发广播时可以设置相应权限的应用接收

19. LinearLayout、RelativeLayout、FrameLayout 的特性及对比，并介绍使用场景。

FrameLayout 忽略不说，因为很少用 ，不是很了解

LinearLayout、RelativeLayout 两者绘制同样的界面时 layout 和 draw 的过程时间消耗相差无几，关键在于 measure 过程。

RelativeLayout 比 LinearLayout 慢了一些。我们知道 ViewGroup 是没有 onMeasure 方法的，这个方法是交给子类自己实现的。因为不同的 ViewGroup 子类布局都不一样，那么 onMeasure 索性就全部交给他们自己实现好了。

RelativeLayout 的 onMeasure 过程

根据源码我们发现 RelativeLayout 会根据 2 次排列的结果对子 View 各做一次 measure。

首先RelativeLayout中子View的排列方式是基于彼此的依赖关系，在确定每个子 View 的位置的时候，需要先给所有的子 View 排序一下，所以需要横向纵向分别进行一次排序测量。

LinearLayout 的 onMeasure 过程

LinearLayout 会先做一个简单横纵方向判断

需要注意的是在每次对 child 测量完毕后，都会调用 child.getMeasuredHeight()/getMeasuredWidth()获取该子视图最终的高度，并将这个高度添加到 mTotalLength 中。

但是 getMeasuredHeight 暂时避开了 lp.weight>0 且高度为 0 子 View，因为后面会将把剩余高度按 weight 分配给相应的子 View。因此可以得出以下结论：

（1）如果我们在 LinearLayout 中不使用 weight 属性，将只进行一次 measure 的过程。（如果使用 weight 属性，则遍历一次 wiew 测量后，再遍历一次 view 测量）

（2）如果使用了 weight 属性，LinearLayout 在第一次测量时获取所有子 View 的高度，之后再将剩余高度根据 weight 加到 weight>0 的子 View 上。

由此可见，weight 属性对性能是有影响的。

所以（总结），

1） RelativeLayout 慢于 LinearLayout 是因为它会让子 View 调用 2 次 measure 过程，而 LinearLayout 只需一次，但是有 weight 属性存在时，LinearLayout 也需要两次 measure。

2)在不响应层级深度的情况下，使用 Linearlayout 而不是 RelativeLayout。

20. 介绍下 SurfaceView

我们在使用普通 View 时，如果我们绘制过程逻辑很复杂，我们的界面更新还非常频繁，这时候就会造成界面的卡顿，影响用户体验，为此我们可以用 SurfaceView 来解决这一问题。

SurfaceView 更适合于频繁刷新界面，而且还会开启一个子线程来对页面进行刷新。同时在底层机制中就实现了双缓冲机制。（双缓冲技术是把要处理的图片在内存中处理好之后，再将其显示在屏幕上。双缓冲主要是为了解决 反复局部刷屏带来的闪烁。把要画的东西先画到一个内存区域里，然后整体的一次性画出来）

使用 SurfaceView 的话需要三个步骤，分别是创建、初始化、使用。

1. 创建 SurfaceView

我们需要自定义一个类继承自 SurfaceView，并且实现两个接口以及接口定义的方法。

重写 3 个构造函数，以及 surfaceCreated()（创建时调用）、surfaceChanged()（改变时调用）、surfaceDestroyed()（销毁时调用）、run()方法，在 run（）方法中写我们子线程中执行的绘图逻辑即可

```
public class SurfaceViewTemplate extends SurfaceView implements SurfaceHolder.Callba
    public SurfaceViewTemplate(Context context) {
        this(context, null);
    }

    public SurfaceViewTemplate(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public SurfaceViewTemplate(Context context, AttributeSet attrs, int defStyleAttr
        super(context, attrs, defStyleAttr);
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
       //创建
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int heig
        //改变
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        //销毁
    }

    @Override
    public void run() {
        //子线程
    }
}
```

2. 初始化 surfaceView

这一步我们主要是定义成员变量以备后面绘图时使用，然后初始化这三个成员变量并且注册对应的回调方法。

```
private SurfaceHolder mSurfaceHolder;
//绘图的Canvas
private Canvas mCanvas;
//子线程标志位
private boolean mIsDrawing;

/**
 * 初始化View
 */
private void initView(){
    mSurfaceHolder = getHolder();
    //注册回调方法
    mSurfaceHolder.addCallback(this);
    //设置一些参数方便后面绘图
    setFocusable(true);
    setKeepScreenOn(true);
    setFocusableInTouchMode(true);
}

public SurfaceViewSinFun(Context context, AttributeSet attrs, int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    //在三个参数的构造方法中完成初始化操作
    initView();
}
```

3. 使用 SurfaceView

这一步又可以分为 3 步来完成:

(1) 通过 lockCanvas()方法获得 Canvas 对象

(2) 在子线程中使用 Canvas 对象进行绘制

(3) 使用 unlockCanvasAndPost()方法将画布内容进行提交

注意：lockCanvas() 方法获得的 Canvas 对象仍然是上次绘制的对象，由于我们是不断进行绘制，但是每次得到的 Canvas 对象都是第一次创建的 Canvas 对象。

21. RecycleView 的使用

RecycleView 是 ListView 的升级版，与经典的 ListView 相比，同样具有 item 的回收复用功能，但 RecycleView 更加高级灵活。

Recycle 基本用法

1. 实现 RecycleView 布局（xml 文件布局）。
2. 设置布局管理器。布局主要有三种实现方式(线性布局、网格、流式布局)
3. 初始化数据
4. 设置 Adapter 与 ViewHolder 进行事件绑定。
5. 设置 setAdapter，如果需要，可以设置分割线（可自定义封装分割线）、Item 动画。
6. 如果需要，可以设置 Item 点击事件（增加 Item）。
7. 如果需要，可以设置长按点击事件（移除 Item）。RecycleView 采用更高级的方法 notifyItemInserted(position) 和 notifyItemRemoved(position)来进行添加和删除的数据刷新。

22. RecycleView 原理(源码)

看原理图

23. Activity-Window-View 三者的差别

Activity 是整个模型的控制单元，

Window 属于承载模型，负责承载视图，

View 是视图显示模型。

一个比喻总结下 Activity Window View 三只之间的关系：Activity 像一个工匠（控制单元），Window 像窗户（承载模型），View 像窗花（显示视图）。

1. 在 Activity 的 attach 方法里，系统会创建 Activity 所属的 Window 对象并为其设置回调接口，Window 对象的创建时通过PolicyManager的makeNewWindow方法实现的。由于Activity 实现了 Window 的 Callback 接口，因此当 Window 接收到外界的状态改变就会回调到 Activity 的方法。

2. View 是 Android 中的视图呈现方式，但是 View 不能单独存在，它必须附着在 Window 这个抽象的概念上面，因此有视图的地方就有 Window。

3. View 借助 ViewRoot 这个纽带，绑定在 Window 上 ViewRoot 对应于 ViewRootImpl 类，它是连接 WindowManager 和 Decorview 的纽带，View 的三大流程（measure，layout，draw）均是通过 ViewRoot 来完成的。

在 ActivityThread 中，当 Activity 对象被创建完毕后，会将 DecorVidew 添加到 Window 中，同时会创建 ViewRootImpl 对象，并将 ViewRootImpl 对象和 DecorView 建立关联。

一个 Activity 中视图的绘制流程：

在 onCreat()中，我们只有一个 setContentView()的操作，

1. 在 Activity 的 setContentView（）中，Activity 将具体实现交给了 Window 处理，
2. 而 Window 的具体实现是 PhoneWindow 在 setContentView 中创建了 DecorView，DecorView 是整棵 View 树的顶级 View，
3. 然后将 View 添加到 DecorView 的 mContentParent 中，
4. 最后回调 Activity 的 onContentChanged 方法通知 Activity 视图已经发生改变。

DecorView 作为顶级 View，一般情况下它内部会包含一个竖直方向的 LinearLayout，在这个 LinearLayout 里面有上下两个部分，上面是标题栏，下面是内容栏，其中标题栏一般由 Activity 的 Theme 样式所决定。在 Activity 中我们通过 setContentView 所设置的布局文件其实就是被加到内容栏之中的。

用户的布局加载

View 的绘制流程是从 ViewRoot 的 performTraversals 方法开始的，它经过 measure，layout 和 draw 三个过程最终将一个 View 绘制出来，

其中 measure 用来测量 View 的宽和高，

layout 用来确定 View 在父容器中的放置位置，

而 draw 则负责将 View 绘制在屏幕上。

如此反复完成一棵 View 树的遍历，整个 View 视图就显示在屏幕上了。

24. requestLayout()与 onLayout()；onDraw()与 drawChild()的区别和联系(源码)

requestLayout()定义： 调用 requestLayout()方法的时机是：当前 View 发生了一些改变，这个改变使得现有的 View 失效，所以调用 requestLayout()方法对 View 树进行重新布局，过程包括了 measure()和 layout()过程，但不会调用 draw()过程，即不会发生重新绘制视图过程。

只是重新测量和布局

onLayout()定义：调用 onLayout()的时机是：View 需要给自己设置大小和位置了或者 ViewGroup 需要给子 View 和 ViewGroup 自身时调用。

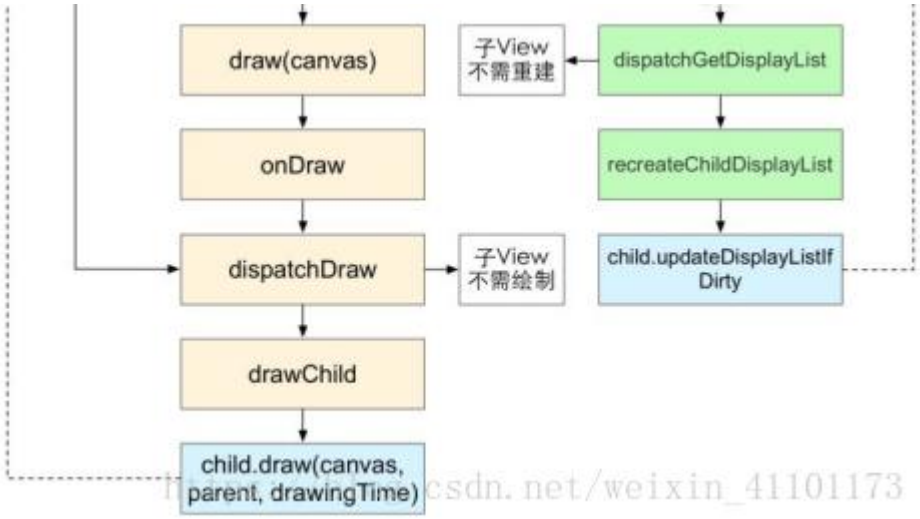
requestLayout()与 onLayout（）方法之间的联系：其实两个方法之间的联系很少，调用时机不同，调用者不同，调用时处理的方式和结果都不同。

onDraw()定义：

onDraw()方法就是在绘制 View 第三步时发生,已经测量好 View 的大小，设置好 View 的布局，剩下最后一步就是，具体画出这个布局。画的方法就是 onDraw()，每个 View 都需要利用这个方法画出自己，ViewGroup 除了要设置背景，不然一般都不会调用该方法

drawChild()定义：

去重新回调每个子视图的 draw()方法。



onDraw()方法和 drawChild()方法之间的联系：

①绘制 VIew 本身的内容，通过调用 View.onDraw(canvas)函数实现；

②ViewGroup 绘制自己的孩子通过 dispatchDraw（canvas）实现，这个方法中调用 drawChild()，而这个方法会调用每个 View 中 onDraw（）来绘制

25. invalidate 和 postInvalidate 的区别及使用(源码)

invalidate 和 postInvalidate **都可以实现界面刷新**

Invalidate 不能直接在线程中调用，违背了单线程模型，必须在 UI 线程中调用。

如果要在子线程中刷新界面，实例化一个 Handler 对象，并重写 handleMessage 方法调用 invalidate()实现界面刷新;而在线程中通过 sendMessage 发送界面更新消息。

postInvalidate()可以在子线程中被调用，使用起来比较简单，不需要 handler，直接在线程中调用 postInvalidate 即可。

区别与联系

`invalidate()` 方法在 UI 线程中调用，重绘当前 UI。

`postInvalidate()` 方法在非 UI 线程中调用，通知 UI 线程重绘。

postInvalidate（）中底层实现还是通过 Hanlder +invalidate（）,只不过被封装起来，因此 postInvalidate（）可以在子线程中刷新界面

26. ListView 重用的是什么？（原理）

有两种情况：

1. 如果复用的 View 为 null 时，我们要创建一个新的 item 以及 ViewHolder，

然后把 item 视图中的控件通过 findViewById 方法寻找到，并添加到

ViewHolder 中， setTag 方法，将 viewholder 传进去，完成 viewholder 与 item 之间的绑定

2.**如果复用的 View 不是为 null**,那么通过 getTag（）方法直接拿过来用，并且从里面拿出 ViewHolder,因为每一个复用的 ViewHolder 肯定是经过处创建并且返回的

（之前的 View）

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    ViewHolder viewHolder;
    if (convertView == null) {
        viewHolder = new ViewHolder();
        convertView = LayoutInflater.from(MainActivity.this).inflate(
                R.layout.adapter_item, null, true);
        viewHolder.img_show = (ImageView) convertView
                .findViewById(R.id.adapter_img);
        viewHolder.tv_text = (TextView) convertView
                .findViewById(R.id.adapter_tv);
        convertView.setTag(viewHolder);
    } else {
        viewHolder = (ViewHolder) convertView.getTag();
    }
```