**View的工作原理**

本篇主要讲View的工作原理和自定义View。工作原理中比较重要的是View绘制的三大流程，此外还包括常见的回调方法也需要掌握。

* **理解MeasureSpec**

MeasureSpec封装了父布局给子布局的测量规格，每个MesureSpec代表了子View的宽或高的规格。它是由mode（测量模式）和size（在某种模式下测量的规格大小）构成。

* **MesureSpec的三种模式：**

**UNSPECIFIED**：父容器不对View进行任何限制，要多大给多大。

**EXACTLY：**父容器已经检测出View所需要的大小了，这时候View的大小就是SpecSize 所指定的值。只要 mode是EXACTLY，那么View的LayoutParams中对应的宽

或高就一定是match\_parent或者指定的具体的数值。

**AT\_MOST：**表示父View没有给子View具体的大小限制，但是存在一个上限，子View 不能超过这个上限。只要子View设置的宽或高是wrap\_content，那么他的

mode一定是AT\_MOST。

* **方法介绍**

MesureSpec是View下的一个静态公开类，它将SpecMode和SpecSize打包成一个整型值（为了减少对象的内存分配开支）:

static int makeMesureSpec ( int size，int mode );

并且还提供了通过MesureSpec（这里代表打包后生成的那个int值）解包的方法：

static int getMode( int mesureSpec ); 获取mode。

static int getSize( int mesureSpec ); 获取size。

* **生成流程**

**子View的MesureSpec创建过程一定是在ViewGroup中进行的**，下面分下ViewGroup构建子View的MesureSpec源码：

**protected void** measureChildWithMargins(View child,  
 **int** parentWidthMeasureSpec, **int** widthUsed,  
 **int** parentHeightMeasureSpec, **int** heightUsed) {  
 **final** MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();  
  
 **final int** childWidthMeasureSpec = *getChildMeasureSpec*(parentWidthMeasureSpec,  
 mPaddingLeft + mPaddingRight + lp.**leftMargin** + lp.**rightMargin** + widthUsed, lp.**width**);  
 **final int** childHeightMeasureSpec = *getChildMeasureSpec*(parentHeightMeasureSpec,  
 mPaddingTop + mPaddingBottom + lp.**topMargin** + lp.**bottomMargin** + heightUsed, lp.**height**);  
  
 child.measure(childWidthMeasureSpec, childHeightMeasureSpec);  
}

其中参数widthUsed和heightUsed分别表代表其方向上除了这个子View，其他子View占用的空间大小（包括它们的Margins的值）。通过源码可以看到子View的宽、高的MesureSpec是通过getChildMeasureSpec()获取的，下面源码：

**public static int** getChildMeasureSpec(**int** spec, **int** padding, **int** childDimension) {  
 **int** specMode = MeasureSpec.*getMode*(spec);  
 **int** specSize = MeasureSpec.*getSize*(spec);  
  
 **int** size = Math.*max*(0, specSize - padding);  
  
 **int** resultSize = 0;  
 **int** resultMode = 0;  
  
 **switch** (specMode) {  
 *// Parent has imposed an exact size on us* **case** MeasureSpec.***EXACTLY***:  
 **if** (childDimension >= 0) {  
 resultSize = childDimension;  
 resultMode = MeasureSpec.***EXACTLY***;  
 } **else if** (childDimension == LayoutParams.***MATCH\_PARENT***) {  
 *// Child wants to be our size. So be it.* resultSize = size;  
 resultMode = MeasureSpec.***EXACTLY***;  
 } **else if** (childDimension == LayoutParams.***WRAP\_CONTENT***) {  
 *// Child wants to determine its own size. It can't be  
 // bigger than us.* resultSize = size;  
 resultMode = MeasureSpec.***AT\_MOST***;  
 }  
 **break**;  
  
 *// Parent has imposed a maximum size on us* **case** MeasureSpec.***AT\_MOST***:  
 **if** (childDimension >= 0) {  
 *// Child wants a specific size... so be it* resultSize = childDimension;  
 resultMode = MeasureSpec.***EXACTLY***;  
 } **else if** (childDimension == LayoutParams.***MATCH\_PARENT***) {  
 *// Child wants to be our size, but our size is not fixed.  
 // Constrain child to not be bigger than us.* resultSize = size;  
 resultMode = MeasureSpec.***AT\_MOST***;  
 } **else if** (childDimension == LayoutParams.***WRAP\_CONTENT***) {  
 *// Child wants to determine its own size. It can't be  
 // bigger than us.* resultSize = size;  
 resultMode = MeasureSpec.***AT\_MOST***;  
 }  
 **break**;  
  
 *// Parent asked to see how big we want to be* **case** MeasureSpec.***UNSPECIFIED***:  
 **if** (childDimension >= 0) {  
 *// Child wants a specific size... let him have it* resultSize = childDimension;  
 resultMode = MeasureSpec.***EXACTLY***;  
 } **else if** (childDimension == LayoutParams.***MATCH\_PARENT***) {  
 *// Child wants to be our size... find out how big it should  
 // be* resultSize = View.*sUseZeroUnspecifiedMeasureSpec* ? 0 : size;  
 resultMode = MeasureSpec.***UNSPECIFIED***;  
 } **else if** (childDimension == LayoutParams.***WRAP\_CONTENT***) {  
 *// Child wants to determine its own size.... find out how  
 // big it should be* resultSize = View.*sUseZeroUnspecifiedMeasureSpec* ? 0 : size;  
 resultMode = MeasureSpec.***UNSPECIFIED***;  
 }  
 **break**;  
 }  
 *//noinspection ResourceType* **return** MeasureSpec.*makeMeasureSpec*(resultSize, resultMode);  
}

我们先看这个方法的参数分别是，1、父ViewGroup的MeasureSpec；2、父View除去当前测量子View的额外剩余空间；3、子View设置在LayoutParams中的宽或高的值。最终的大小：

1、假如设置了宽或高数值，那么宽高就是设定的这个值；

2、假如没有设定宽或高数值，

宽或高 = 父容器宽或高 - 父容器的padding的值 - 其他的子View的宽或高 - 其他子View的 Margins值 - 测量View的Margins值

通过swich中的判断我们可以看出，子View的MesureSpec的生成是由父容器的MesureSpec的mode和它本身的LayoutParams最终确定的。

这里注意，子View的size假如设置了固定数值，那么它的大小就是他自己设置的数值，假如没有设置数值则是父容器减去padding的值。

|  |  |  |  |
| --- | --- | --- | --- |
| chlidLayoutParams  parentSpecMode | EXACTLY | AT\_MOST | UNSPECIFIED |
| dp/px | EXACTLY  childSize | EXACTLY  ChildSize | EXACTLY  childSize |
| match\_parent | EXACTLY  parentSize | AT\_MOST  parentSize | UNSPECIFIED  0 |
| wrap\_content | AT\_MOST  parentSize | AT\_MOST  parentSize | UNSPECIFIED  0 |

* **DecorView（顶级View）的MeasureSpec创建**

DecorView的MeasureSpec创建是在ViewRootImpl中measureHierarchy方法里

创建的：

childWidthMeasureSpec = *getRootMeasureSpec*(desiredWindowWidth, lp.**width**);  
 childHeightMeasureSpec = *getRootMeasureSpec*(desiredWindowHeight, lp.**height**);  
 performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);

其中desiredWindowWidth和desiredWindowHeight分别是屏幕的宽和高。接着再看*getRootMeasureSpec方法：*

**private static int** getRootMeasureSpec(**int** windowSize, **int** rootDimension) {  
 **int** measureSpec;  
 **switch** (rootDimension) {  
  
 **case** ViewGroup.LayoutParams.***MATCH\_PARENT***:  
 *// Window can't resize. Force root view to be windowSize.* measureSpec = MeasureSpec.*makeMeasureSpec*(windowSize, MeasureSpec.***EXACTLY***);  
 **break**;  
 **case** ViewGroup.LayoutParams.***WRAP\_CONTENT***:  
 *// Window can resize. Set max size for root view.* measureSpec = MeasureSpec.*makeMeasureSpec*(windowSize, MeasureSpec.***AT\_MOST***);  
 **break**;  
 **default**:  
 *// Window wants to be an exact size. Force root view to be that size.* measureSpec =MeasureSpec.*makeMeasureSpec*(rootDimension,MeasureSpec.***EXACTLY***);  
 **break**;  
 }  
 **return** measureSpec;  
}

这样看就很明确了：

LayoutParams.MATCH\_PARENT：mode为EXACTLY。size为屏幕大小。

LayoutParams.WRAP\_CONTENT：mode为AT\_MOST。size为屏幕大小。

固定大小（比如设置100dp）：mode为EXACTLY***。***size为固定的数值。

所以DecorView的MeasureSpec的创建是由窗口的大小和自身的LayoutParams决定。

* **View的绘制流程（****http://www.jianshu.com/p/5a71014e7b1b）**

一个Android应用程序窗口里面包含了很多UI元素，这些UI元素是以树形结构来组织的，所有的UI元素都是存在于ViewGroup中的，不管是View的测量还是ViewGroup的测量，它们都是从measure()方法测量完成的，这个方法是final的不可重写的，它的实现永远在View类里面，measure()方法最终调用的是onMeasure()方法来测量。真正的测量的实现是在onMeasure()方法中的（VeiwGroup类没有实现onMeasure()方法，它的派生类实现了，因为每个派生类实现的逻辑不一样），当调用onMeasure()方法时，VeiwGroup派生类会先测量子View的大小，调用子View的measure()方法，假如子View是一个View，在它的measure方法中会调用onMeasure()方法设置自己的大小，假如是一个ViewGroup派生类，那么会继续调用自己的子View的measure方法，直到它下面的所有的View都测量完，才会调用setMeasuredDimension()方法来设置最终的宽高。

* **View的measure过程**

View的测量是在measure()方法，但是主要的测量工作的是在onMeasure()方法中进行的，**measure()方法final的不可重写**，想要重写View测量需要重写onMeasure()。

**public final void** measure(**int** widthMeasureSpec, **int** heightMeasureSpec) {  
 **...**onMeasure(widthMeasureSpec, heightMeasureSpec);  
 ...

}

接下来看onMeasure()方法的默认实现：

**protected void** onMeasure(**int** widthMeasureSpec, **int** heightMeasureSpec) {  
 setMeasuredDimension(*getDefaultSize*(getSuggestedMinimumWidth(),

widthMeasureSpec),  
 *getDefaultSize*(getSuggestedMinimumHeight(), heightMeasureSpec));  
 }

onMeasure()的实现很简单调用的是setMeasuredDimension()方法，可以说调用了这个方法就等于给**mMeasuredHeight**和**mMeasuredWidth**赋值了，也就等于View的测量结束。所以我们可以直接调用setMeasuredDimension(100,300)来设置死View的宽和高。但是它只能在onMeasure()方法中调用，在别的方法里调用会抛出异常。

接下来看setMeasuredDimension()方法接受的宽和高是由getDefaultSize()和getSuggestedMinimumWidth()来确认的，这两个方法源码：

**public static int** getDefaultSize(**int** size, **int** measureSpec) {  
 **int** result = size;  
 **int** specMode = MeasureSpec.*getMode*(measureSpec);  
 **int** specSize = MeasureSpec.*getSize*(measureSpec);  
  
 **switch** (specMode) {  
 **case** MeasureSpec.***UNSPECIFIED***:  
 result = size;  
 **break**;  
 **case** MeasureSpec.***AT\_MOST***:  
 **case** MeasureSpec.***EXACTLY***:  
 result = specSize;  
 **break**;  
 }  
 **return** result;  
 }

**protected int** getSuggestedMinimumWidth() {  
 **return** (**mBackground** == **null**) ? **mMinWidth** : *max*(**mMinWidth**,

**mBackground**.getMinimumWidth());  
 }

**protected int** getSuggestedMinimumHeight() {  
 **return** (**mBackground** == **null**) ? **mMinHeight** : *max*(**mMinHeight**,

**mBackground**.getMinimumHeight());  
  
 }

getDefaultSize()的第一个参数size是getSuggestedMinimumXXX()返回的建议宽度或高度，这个建议的宽度或高度是由View的Background和在布局文件设置的minWidth或minHeight共同决定的。返回逻辑：如果View没有设置背景，那么返回设置的minWidth或minHeight。如果设置背景则返回minWidth或minHeight和设置的background的minWidth或minHeight当中最大的一个，而这些值都有可能为0。【这里background的获取的宽高其实是Drawable的原始宽高，假如Drawable没有原始宽高，则就返回0，而drawable什么时候有宽高呢？这里先举例子：ShapeDrawable无原始宽高，而BitmapDrawbale有原始宽高（图片的尺寸）】

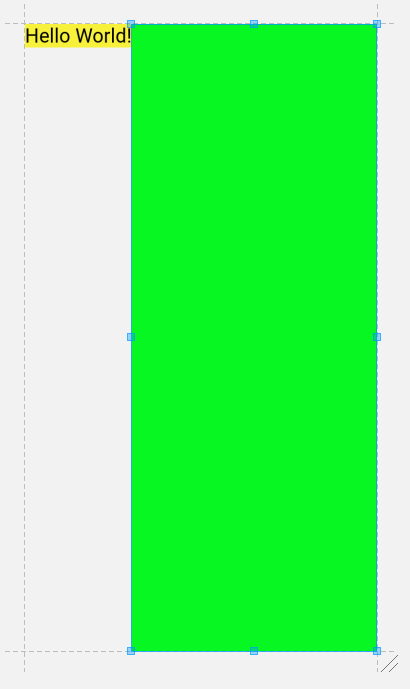
getDefaultSize()的第二个参数，通过上面ViewGroup对子View的MeasureSpec构建源码分析可以知道，它就是子View的MeasureSpec。

getDefaultSize()的源码很简单，假如子View的specMode是AT\_MOST或者EXACTLY那么返回大小就是子View的MeasureSpec的specSize的值，如果是UNSPECIFIED，那么返回的大小就是getSuggestedMinimumXXX的值。

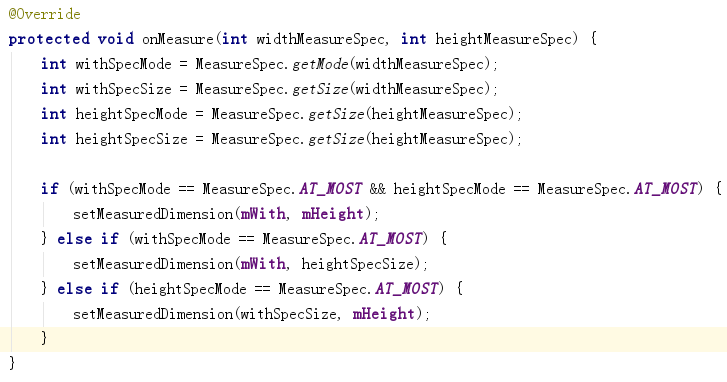
【特别注意】：对于View的默认测量很简单，大部分情况就是拿子View的MeasureSpec的seicSize的值当做最终测量的大小。但是我们通过getDefaultSize()方法可以看出子View的AT\_MOST模式和EXACTLY模式返回的都是子View的MeasureSpec的值（这个值就是父控件目前剩余的大小），那么就是说子View设置的wrap\_content和match\_parent的效果是一样的，可以看下面的示例：



上面设置的布局文件中View的宽高都是wrap\_content，那么他的效果也和我们上面分析的是一样的，占用了父控件剩余的空间，和设置的match\_parent效果是一样的：



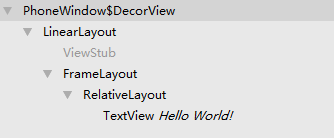
而其他的派生类，如：TextView，Button，ImageView等都是重写了onMeasure()方法的。不是简单的使用MeasureSpec的specSize来当大小，而是针对wrap\_content情形做了特殊处理。那么问题来了，假如我们自定义一个view直接继承Veiw类,我们不想出现上面的情况（设置wrap\_content的结果和match\_parent的效果一样）：我们通常会重写onMeasure()方法，判断出wrap\_content，并且设置大小：



当设置了wrap\_content的时候，我们就给它指定一个大小，假如不是，那么就使用系统测量的大小。

* **ViewGroup的measure的过程**

我们知道一个View一定是在ViewGroup当中的，所以一个页面的测量首先是测量的ViewGroup，所以，我们需要从根View入手，即从DecorView开始分析一个页面的具体的绘制流程。



如上图是一个新建项目默认的main\_layout.xml文件的结构，DecorView是一个FrameLayout,包含一个LinearLayout（这里我们设置app的主题是NoActionBar,设置不同的主题，DecorView下的布局内容是不一样的。），LinearLayout它包含一个FrameLayout和一个ViewStub,其中ViewStub我们可以忽略。我们在Activity通过setContent()方法设置的布局就是设置成了FrameLayout的子View，它的id是android.id.content。

所以我们只分析DecorView——>LinearLayout——>FrameLayout——>我们的布局文件。这个流程。

首先DecorView的绘制才是绘制的真正的开始，DecorView的绘制是从ViewRootImpl的performTraversals方法开始：

private void performTraversals() {

......

int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);

int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);

......

mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);

......

mView.layout(0, 0, mView.getMeasuredWidth(), mView.getMeasuredHeight());

......

mView.draw(canvas);

......

}

**private static int** getRootMeasureSpec(**int** windowSize, **int** rootDimension) {  
 **int** measureSpec;  
 **switch** (rootDimension) {  
  
 **case** ViewGroup.LayoutParams.MATCH\_PARENT:  
 *// Window can't resize. Force root view to be windowSize.* measureSpec = MeasureSpec.makeMeasureSpec(windowSize,

MeasureSpec.EXACTLY);  
 **break**;  
 **case** ViewGroup.LayoutParams.WRAP\_CONTENT:  
 *// Window can resize. Set max size for root view.* measureSpec = MeasureSpec.makeMeasureSpec(windowSize,

MeasureSpec.AT\_MOST);  
 **break**;  
 **default**:  
 *// Window wants to be an exact size. Force root view to be that size.* measureSpec = MeasureSpec.makeMeasureSpec(rootDimension,

MeasureSpec.EXACTLY);  
 **break**;  
 }  
 **return** measureSpec;  
}

DecorView的测量流程跟普通的View测量流程是一样的，都是先生成自身的测量规则MeasureSpec，然后执行自己的measure()方法，不过DecorView通过getRootMeasureSpec()方法的生成的MeasureSpec测量规则。这里需要强调一下getRootMeasureSpec()方法的两个参数，第一个参数（mWidth和mHeight）是屏幕的宽或高（包括状态栏），第二个参数lp是指WindowManager.LayoutParams（即lp=**new** WindowManager.LayoutParams()

），他的默认值都是MATCH\_PARENT，size是屏幕宽高。

根据上面源码我们可以知道，DecorView宽和高的MeasureSpec的mode是EXACTLY，size是屏幕宽高。

准备好了MeasureSpec，接下来就是调用View的measure()方法进行测量了。首先DecorView是一个FrameLayout是一个ViewGroup，而measure()方法是final的是不可重写的，所以这里调用measure()方法调用的是View的measure(),通过上面对View的测量源码分析我们知道，measure()方法最终测量是调用onMeasure(),但是ViewGroup也没有重写onMeasure()方法，只有继承了ViewGroup的派生类才重写了onMeasure()方法，这里DecorView也重写了onMeasure()方法,但是最终也是调用**super**.onMeasure(widthMeasureSpec, heightMeasureSpec);也就是FrameLayout的onMeasure()方法。

ViewGroup内部是没有实现onMeasure()方法的，只有他的派生类才会实现，这是因为不同的派生类功能不一样，它的具体实现的方式也不一样，所以就交到派生类去实现，这里因为DecorView继承了FrameLayout，所以我们只分析FrameLayout的onMeasure()方法：

@Override  
**protected void** onMeasure(**int** widthMeasureSpec, **int** heightMeasureSpec) {  
 **int** count = getChildCount();  
  
 **final boolean** measureMatchParentChildren =  
 MeasureSpec.*getMode*(widthMeasureSpec) != MeasureSpec.***EXACTLY*** ||  
 MeasureSpec.*getMode*(heightMeasureSpec) != MeasureSpec.***EXACTLY***;  
 **mMatchParentChildren**.clear();  
  
 **int** maxHeight = 0;  
 **int** maxWidth = 0;  
 **int** childState = 0;  
  
 **for** (**int** i = 0; i < count; i++) {  
 **final** View child = getChildAt(i);  
 **if** (**mMeasureAllChildren** || child.getVisibility() != ***GONE***) {

//先测量子View  
 measureChildWithMargins(child, widthMeasureSpec, 0, heightMeasureSpec, 0);  
 **final** LayoutParams lp = (LayoutParams) child.getLayoutParams();  
 maxWidth = Math.*max*(maxWidth,  
 child.getMeasuredWidth() + lp.**leftMargin** + lp.**rightMargin**);  
 maxHeight = Math.*max*(maxHeight,  
 child.getMeasuredHeight() + lp.**topMargin** + lp.**bottomMargin**);  
 childState = *combineMeasuredStates*(childState, child.getMeasuredState());  
 **if** (measureMatchParentChildren) {  
 **if** (lp.**width** == LayoutParams.***MATCH\_PARENT*** ||  
 lp.**height** == LayoutParams.***MATCH\_PARENT***) {  
 **mMatchParentChildren**.add(child);  
 }  
 }  
 }  
 }  
 //FrameLayout测量完子View之后设置自己的宽高了。  
 setMeasuredDimension(*resolveSizeAndState*(maxWidth, widthMeasureSpec, childState),  
 *resolveSizeAndState*(maxHeight, heightMeasureSpec,  
 childState << ***MEASURED\_HEIGHT\_STATE\_SHIFT***));  
  
 count = **mMatchParentChildren**.size();  
 **if** (count > 1) {  
 **for** (**int** i = 0; i < count; i++) {  
 **final** View child = **mMatchParentChildren**.get(i);  
 ...//省略号部分是重置了子View的MeasureSpec  
 child.measure(childWidthMeasureSpec, childHeightMeasureSpec);  
 }  
 }  
}

FrameLayout的onMeasure()方法首先遍历自己的子View，并且调用measureChildWithMargins()方法测量子View,只要子View不是GONE的都会参与测量，注意当调用measureChildWitnMargins()方法时FrameLayout将自己的MeasureSpec传递进去了。当测量完所有的子View后，又调用setMeasuredDimension()方法设置自己最终的宽和高，到此为止FrameLayout的测量结束。

但是我们看到下面代码的执行，又循环测量了**mMatchParentChildren**列表里的View，通过源码我们可以看到这个列表添加的是宽或高为MATCH\_PARENT的View，所以我们就知道了，某些View的测量并不是一次就可以完成的，可能需要多次测量才行。

我们继续看DecorView如何测量子View（即LinearLayout）measureChildWithMargins()方法：

**protected void** measureChildWithMargins(View child,  
 **int** parentWidthMeasureSpec, **int** widthUsed,  
 **int** parentHeightMeasureSpec, **int** heightUsed) {  
 **final** MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();  
  
 **final int** childWidthMeasureSpec = *getChildMeasureSpec*(parentWidthMeasureSpec,  
 mPaddingLeft + mPaddingRight + lp.**leftMargin** + lp.**rightMargin** + widthUsed, lp.**width**);  
 **final int** childHeightMeasureSpec = *getChildMeasureSpec*(parentHeightMeasureSpec,  
 mPaddingTop + mPaddingBottom + lp.**topMargin** + lp.**bottomMargin** + heightUsed, lp.**height**);  
  
 child.measure(childWidthMeasureSpec, childHeightMeasureSpec);  
}

首先获取通过父View的MeasureSpec和子View的LayoutParams获取的到子View的MeasureSpec（具体获取过程请看顶部View的MeasureSpec的获取），然后调用子View（即LinearLayout）的measure()方法，最终依然是调用LinearLayout的onMeasure()方法，然后遍历并测量子View...以此类推，直到最终的子View没有继承ViewGroup的View为止，最后调用setMeasuredDimension()方法来设置最终的宽高。

通过上面我们可以总结出测量流程了，从DecorView开始：

Child.measure()



最后我们来解决在一个activity创建后，如何获取控件的宽高问题：

1. Activity/View#onWindowFocusChanged()

当Activity失去或者获得全局焦点的时候会回调这个方法。这时候获取宽高是没问题的，但是如果Activity频繁的调用onResume()和onPause()方法，那么这个方法也会频繁的被调用。具体使用如下：

@Override  
**public void** onWindowFocusChanged(**boolean** hasFocus) {  
 **super**.onWindowFocusChanged(hasFocus);  
 **if** (hasFocus)  
 {  
 **int** width = **view**.getMeasuredWidth();  
 **int** height = **view**.getMeasuredHeight();  
 }  
}

1. view.post（runnable）

可以将一个Runnable投递到View的消息队列的尾部，然后等待looper调用此runnable的时候，View此时也初始化好了：

**view**.post(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **int** width = **view**.getMeasuredWidth();  
 **int** height = **view**.getMeasuredHeight();  
 }  
});

1. ViewTreeObserver

当View树状态发生改变或者View树内部的View可见性发生改变的时候，view会回调OnGlobalLayoutListener这个接口的onGlobaLayout这个方法，此时是获取View的宽高的一个良好时机，但是这种方式依然会随着View树的改变而被频繁的调用，使用方式：

ViewTreeObserver observer = **view**.getViewTreeObserver();  
observer.addOnGlobalLayoutListener(**new** ViewTreeObserver.OnGlobalLayoutListener() {  
 @RequiresApi(api = Build.VERSION\_CODES.***JELLY\_BEAN***)  
 @Override  
 **public void** onGlobalLayout() {  
 **view**.getViewTreeObserver().removeOnGlobalLayoutListener(**this**);  
 **int** width = **view**.getMeasuredWidth();  
 **int** height = **view**.getMeasuredHeight();  
 }  
});

* **layout过程**

layout过程是确定布局中元素位置（坐标）的过程，涉及到两个方法：layout()和onLayout()。layout()方法确定View本身的位置，onLayout()确定所有子View的位置。这里注意他们的顺序是先确定自己的位置调用layout()，才能确定子View的位置onLayout()。

**View**：实现了layout()方法来确定自己的位置，但是onLayout()是个空实现。

**ViewGroup**：重写了layout()方法,并且用final修饰，所以它的子类不能在重写

layout()这个方法了，但是ViewGroup的子类的layout()最终调用的是super.layout()，也就是View类中的layout()。当ViewGroup的位置确定后，它又会调用onLayout()方法来遍历它的子View，然后调用子View的layout()方法，确定子View的位置。ViewGroup中onLayout()方法只是用abstract修饰，要求它的子类必须实现onLayout()方法。

**ViewGroup子类**：只能且必须实现onLayout()方法。

还是从DecorView开始分析，当测量完View的大小后，就开始进行layout：

**Private void** performLayout(WindowManager.LayoutParams lp, **int**

desiredWindowWidth,**int** desiredWindowHeight) {  
 ...  
 **final** View host = mView;  
 ...  
 host.layout(0, 0, host.getMeasuredWidth(), host.getMeasuredHeight());

...

}

mView指的是DecorView，调用的layout()，我们知道DecorView继承自FrameLayout，继承自ViewGroup，直接看ViewGroup中layout()方法的源码：

@Override  
**public final void** layout(**int** l, **int** t, **int** r, **int** b) {  
 **if** (!**mSuppressLayout** && (**mTransition** == **null** || !**mTransition**.isChangingLayout())) {  
 **if** (**mTransition** != **null**) {  
 **mTransition**.layoutChange(**this**);  
 }  
 **super**.layout(l, t, r, b);  
 } **else** {  
 *// record the fact that we noop'd it; request layout when transition finishes* **mLayoutCalledWhileSuppressed** = **true**;  
 }  
}

看源码大体意思是，如果当前ViewGroup没有添加LayoutTransition动画，或者LayoutTransition动画并没有运行，那么就调用super.layout()方法。ViewGroup中layout是final的所以FrameLayout方法是不会重写的，但是ViewGroup中的layout()最终调用的是View类的layout()方法：

**public void** layout(**int** l, **int** t, **int** r, **int** b) {  
 **if** ((**mPrivateFlags3** & ***PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT***) != 0) {  
 onMeasure(**mOldWidthMeasureSpec**, **mOldHeightMeasureSpec**);  
 **mPrivateFlags3** &= ~***PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT***;  
 }  
  
 **int** oldL = **mLeft**;  
 **int** oldT = **mTop**;  
 **int** oldB = **mBottom**;  
 **int** oldR = **mRight**;  
  
 **boolean** changed = *isLayoutModeOptical*(**mParent**) ?  
 setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);  
  
 **if** (changed || (**mPrivateFlags** & ***PFLAG\_LAYOUT\_REQUIRED***) == ***PFLAG\_LAYOUT\_REQUIRED***) {  
 onLayout(changed, l, t, r, b);  
 ...  
 }  
  
}

通过setFrame(l,t,r,b)方法来给mLeft、mTop、mRight、mBottom赋值，mLeft、mRight、mTop和mBottom分别用来描述当前视图的的四个顶点在父容器中的位置，这几个值构成的矩形区域就能确定View在父视图的位置了，这里的显示位置是相对于父视图来说的。但是通过从DcorView中传过来的参数可以看出来，它的位置就是整个屏幕，而且没有父视图。

这里注意，setFrame(l,t,r,b)方法会返回一个boolean的值，然后传递到onLayout()方法中具体看注释：

**protected boolean** setFrame(**int** left, **int** top, **int** right, **int** bottom) {

//**默认为false**  
 **boolean** changed = **false**;  
 //**判断四个点的坐标是否改变**

**if** (**mLeft** != left || **mRight** != right || **mTop** != top || **mBottom** != bottom) {

//**当它的四个点的坐标改变的时候才会为true**  
 changed = **true**;

**int** oldWidth = **mRight** - **mLeft**;  
 **int** oldHeight = **mBottom** - **mTop**;  
 **int** newWidth = right - left;  
 **int** newHeight = bottom - top;  
 **boolean** sizeChanged = (newWidth != oldWidth) || (newHeight != oldHeight);  
  
 *// Invalidate our old position* invalidate(sizeChanged);  
 //**更新mLeft、mTop、mRight、mBottom的值**  
 **mLeft** = left;  
 **mTop** = top;  
 **mRight** = right;  
 **mBottom** = bottom;  
 **mRenderNode**.setLeftTopRightBottom(**mLeft**, **mTop**, **mRight**, **mBottom**);  
  
 **mPrivateFlags** |= ***PFLAG\_HAS\_BOUNDS***;  
  
 //**如果大小改变则View的大小**

**if** (sizeChanged) {  
 sizeChange(newWidth, newHeight, oldWidth, oldHeight);  
 }

....  
 }  
 **return** changed;  
}

接下来调用onLayout()方法，对于View来说就是空实现：

**protected void** onLayout(**boolean** changed, **int** left, **int** top, **int** right, **int** bottom) {}

对于ViewGroup来说虽然重写了，但是添加了abstract来修饰，并且要求子类必须去实现：

@Override  
**protected abstract void** onLayout(**boolean** changed,**int** l, **int** t, **int** r, **int** b);

这是因为ViewGroup的子类的功能不同所以实现也不同，而DecorView也重写了这个方法，但是它首先调用的是super.onLayout()方法，所以调用的是FrameLayout的onLayout()方法：

**for** (**int** i = 0; i < count; i++) {  
 **final** View child = getChildAt(i);  
 **if** (child.getVisibility() != ***GONE***) {  
 //**中间省略部分是结合child的measureSize还有父布局的padding，Gravity等等属性综 合来计算child.layout(l,t,r,b)方法的四个参数。每个ViewGroup的子类实现方式是**

**不一样的这里就不详细分析了。**  
 child.layout(childLeft, childTop, childLeft + width, childTop + height);  
 }

}

很简单就是遍历子View并且调用子View的layout方法确定自己的位置。当所有的子View确定好位置，整个layout过程也就结束。

**测量measure过程是必须的吗？**

不是必须的！我们看当child.layout()的时候，chaild的四个顶点已经通过child的MeasureSpec算出来了，但是我们依然可以在View的子类中重写layout()方法，来指定它的四个顶点，这样一来我们之前的测量值就没有用了。

measure完成后得到的是View的测量的宽高，而layout完成后得到的是View最终的宽高，如果我们不干预layout过程中四个顶点的值，那么getMeasureHeight()/getMeasureWidth的值与getHeight()/getWidth()的值相等：

**measure过程后才能获得**：

**public final int** getMeasuredHeight() {  
 **return mMeasuredHeight** & ***MEASURED\_SIZE\_MASK***;  
 }

**public final int** getMeasuredWidth() {  
 **return mMeasuredWidth** & ***MEASURED\_SIZE\_MASK***;  
 }

**layout过程后才能获得**：

**public final int** getWidth() {  
 **return mRight** - **mLeft**;  
 }

**public final int** getHeight() {  
 **return mBottom** - **mTop**;  
 }

* **draw过程**

draw过程就是将View绘制到屏幕上，我们从DecorView源码可以看到当measure和layout过程执行完了之后就会调用DecorView的draw(canvas);方法了,所以我们先看View类中draw()源码：

**public void** draw(Canvas canvas) {  
 **...**  
 */\*  
 \* Draw traversal performs several drawing steps which must be executed  
 \* in the appropriate order:  
 \*  
 \* 1. Draw the background  
 \* 2. If necessary, save the canvas' layers to prepare for fading  
 \* 3. Draw view's content  
 \* 4. Draw children  
 \* 5. If necessary, draw the fading edges and restore* *layers  
 \* 6. Draw decorations (scrollbars for instance)  
 \*/  
  
 // Step 1, draw the background, if needed* **int** saveCount;  
  
 **if** (!dirtyOpaque) {  
 drawBackground(canvas);  
 }  
  
 *// skip step 2 & 5 if possible (common case)* **...**  
 **if** (!verticalEdges && !horizontalEdges) {  
 *// Step 3, draw the content* **if** (!dirtyOpaque) onDraw(canvas);  
 *// Step 4, draw the children* dispatchDraw(canvas);  
 *// Overlay is part of the content and draws beneath Foreground* **if** (**mOverlay** != **null** && !**mOverlay**.isEmpty()) {  
 **mOverlay**.getOverlayView().dispatchDraw(canvas);  
 }  
 *// Step 6, draw decorations (foreground, scrollbars)* onDrawForeground(canvas);  
 *// we're done...* **return**;  
 }

通过View的draw()方法以及英文注释很清楚的知道draw()方法执行的是一系列的绘制流程：

drawBackground(canvas):绘制背景

onDraw(canvas):绘制自己（在View中是个空实现，由子View去实现）

dispatchDraw(canvas):绘制children（在View中是个空实现，由子View去实现）

onDrawScrollBars:绘制View的滚动条（不是重点）

在VeiwGroup中只实现了dispatchDraw()方法，并没有实现onDraw()方法，另外ViewGroup中也没有draw()方法。

我们按流程来分析：

第一步：

**private void** drawBackground(Canvas canvas) {  
 **final** Drawable background = **mBackground**;  
 ...

//利用layout的四个顶点来确认绘制区域  
 setBackgroundBounds();

...

//调用Drawable的draw()方法绘制背景。  
 background.draw(canvas);

...  
 }

第二步：

调用onDraw()方法是绘制自己，因为每个View的子类都不一样所以这个方法为空，让子类自己去实现。而ViewGroup也没有实现这个类，因为它的子类也需要自己实现（在五大布局中只有LinearLayout实现了这个方法）。

第三步：

dispatchDraw()绘制子View，因为View类没有子View所以他是个空实现。VeiwGroup实现了这个方法（五大布局都没有实现这个方法！），我们在自定义控件时除非比较特别，不然一般也不需要去重写它：

@Override  
 **protected void** dispatchDraw(Canvas canvas) {

**final int** childrenCount = **mChildrenCount**;  
 **final** View[] children = **mChildren**;

...

**for** (**int** i = 0; i < childrenCount; i++) {  
 ...  
 **if** ((child.mViewFlags & VISIBILITY\_MASK) == ***VISIBLE*** ||

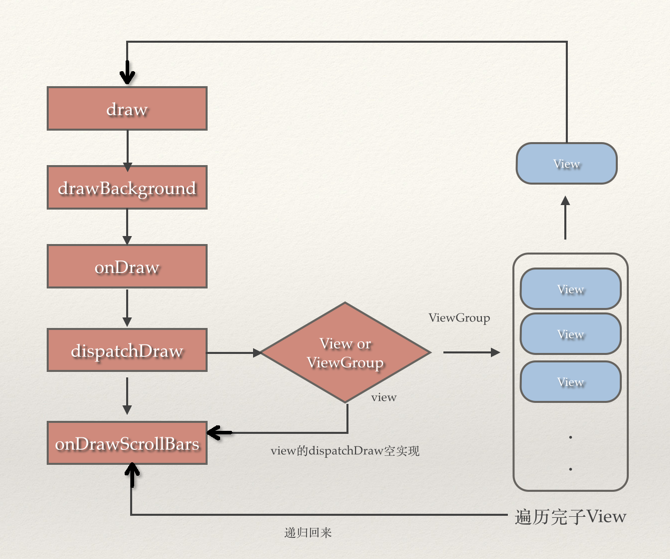
child.getAnimation() != **null**) {  
 more |= drawChild(canvas, child, drawingTime);  
 }  
 }

}

很显然他就是遍历自己的子View然后调用child.draw(canvas,this,drawingTime)方法这里的this是指ViewGroup，而这View的draw(canvas,this,drawingTime)方法的核心过程就是为子视图分配合适的剪切区，剪切去的大小是由layout过程决定的，而剪切区的位置取决于滚动值以及子视图当前动画，设置完剪切区后就开始调用draw(canvas)方法进行具体的绘制流程了。

第四步：对View的滚动条绘制（不重要）

draw流程图:



注意当ViewGroup执行dispatchDraw()方法绘制child的之前先调用View类的draw(canvas,this,drawingTime)绘制剪切区，然后才是调用View类的draw(canvas)方法。

* **自定义View的自定义属性**

自定义属性比较简单，按下列步骤即可：

**第一步**：在values文件夹下创建自定义属性的XML，一般习惯叫attrs.xml，但是这个名字可以随意取：

*<?***xml version="1.0" encoding="utf-8"***?>* <**resources**>  
 <**declare-styleable name="MyView"**>  
 <**attr name="cus\_color" format="color"** />  
 </**declare-styleable**>  
 </**resources**>

<declare-styleable>：属性集合，集合的name可以任意取名字。

<arrt> ：属性，name为属性名称，format为属性类型。还有其它属性可以自行差文档。

**第二步**：在自定义View的构造方法中解析属性：

**private int cus\_color**;  
**public** ViewTest(Context context, AttributeSet attrs) {  
 **super**(context, attrs);  
 TypedArray array = context.obtainStyledAttributes(attrs, R.styleable.***MyView***);  
 **cus\_color** = array.getColor(R.styleable.***MyView\_cus\_color***,Color.***RED***);  
 array.recycle();  
}

首先加载自定义属性集合MyView，然后在集合中取出想要的属性，最后释放集合资源。

**第三步：**在布局文件中使用自定属性：

*<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:myView="http://schemas.android.com/apk/res-auto"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical"**>  
   
 <**com.work.project.myapplication.ViewTest  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="20dp"  
 myView:cus\_color="@color/colorAccent"** />  
</**LinearLayout**>

首先为了使用自定义属性，必须在布局文件中添加schemas声明**xmlns:myView="http://schemas.android.com/apk/res-auto"。**

myView：是自定义属性的前缀，在自定义控件中设置属性，必须与这个前缀一样。

* **自定义View解惑**

**1、onSizeChanged(int w，int h，int oldw，int oldh)方法。**

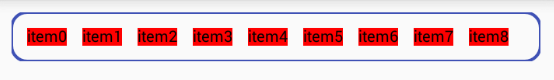
这个方法是当View的尺寸发生改变的时候调用，具体调用位置在View的layout()方法中调用的setFrame()方法中，如果传入的left，top，right，bottom与当前mLeft，mTop，mRight，mBottom不同，则调用sizeChange()方法：

**protected boolean** setFrame(**int** left, **int** top, **int** right, **int** bottom) {  
 **boolean** changed = **false**;  
 。。。  
 **if** (**mLeft** != left || **mRight** != right || **mTop** != top || **mBottom** != bottom) {  
 changed = **true**;  
 。。。  
 **if** (sizeChanged) {  
 sizeChange(newWidth, newHeight, oldWidth, oldHeight);  
 }  
 。。。  
 }  
 **return** changed;  
}

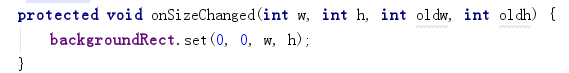
然后在sizeChange()方法中会调用onSizeChange()方法，这个方法是一个空实现。

1. **onDraw方法中绘制边框**

在自定义ViewGroup的onDraw方法中绘制边框时遇到的问题，如下图：



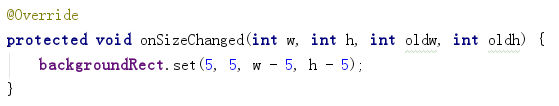
当边框设置了角度后边框的四个角变粗。出现这个问题的原因就是因为绘制边框时没有控制好绘制的区域RectF，RectF设置的上下左右的值超出了ViewGroup大小：



这里RectF设置的正好是ViewGroup的大小，这样绘制时画笔的起点会超出ViewGroup的大小一些，我们设置一下ViewGroup的背景颜色就能很好的看出来：



由上图可以看出其实不是边框的四个圆角变粗，它们只是正常的粗细，而是因为直线有一部分没有画出来导致的。解决办法很简单只要在设置RectF绘制区域时将边框线的粗细算进去就可以了，例如设置的边框粗细为5，即paint.setStrokeWidth(5);则RectF应如下设置：



这样就能将边框绘制在ViewGroup之中了，效果如如下：

