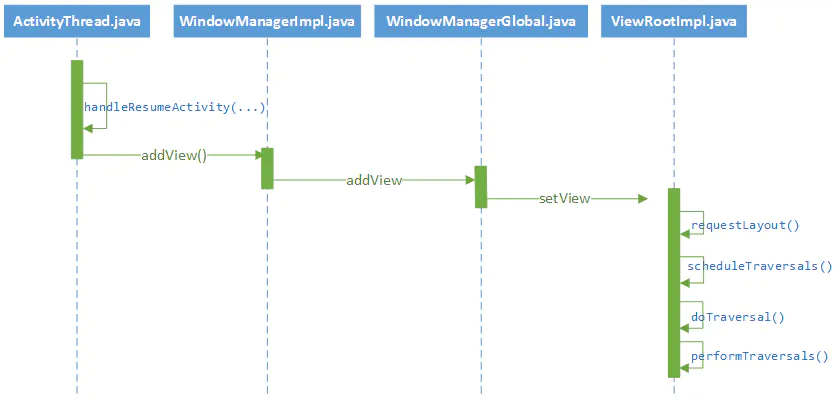
Window概念  
Window表示的是一个窗口的概念，它是站在WindowManagerService角度上的一个抽象的概念，Android中所有的视图都是通过Window来呈现的，不管是Activity、Dialog还是Toast，只要有View的地方就一定有Window。  
注意：抽象的Window概念和PhoneWindow这个类并不是同一个东西，PhoneWindow表示的是手机屏幕的抽象，它充当Activity和DecorView之间的媒介，就算没有PhoneWindow也是可以展示View的。

DecorView概念  
DecorView是整个Window界面的最顶层View，View的测量、布局、绘制、事件分发都是由DecorView往下遍历这个View树。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(具体情况和Android的版本及主题有关)，上面是【标题栏】，下面是【内容栏】。在Activity中我们通过setContentView所设置的布局文件其实就是被加载到【内容栏】中的，而内容栏的id是content，因此指定布局的方法叫setContent()。

ViewRoot概念  
ViewRoot对应于ViewRootImpl类，它是连接WindowManager和DecorView的纽带，View的三大流程均是通过ViewRoot来完成的。在ActivityThread中，当Activity对象被创建完之后，会将DecorView添加到Window中，同时会创建对应的ViewRootImpl，并将ViewRootImpl和DecorView建立关联，并保存到WindowManagerGlobal对象中。

## **绘制的起源点**

Activity启动在ActivityThread.java类中完成，期间会调用到handleResumeActivity(...)方法，这个方法是View绘制的起源点，整个调用链如下图所示：



### 2.1 handleResumeActivity()

关键代码如下：

//=== ActivityThread.java ===final void handleResumeActivity(...) {

......

//跟踪代码后发现其初始赋值为mWindow = new PhoneWindow(this, window, activityConfigCallback);

r.window = r.activity.getWindow();

//从PhoneWindow实例中获取DecorView

View decor = r.window.getDecorView();

......

//跟踪代码后发现，vm值为上述PhoneWindow实例中获取的WindowManager。

ViewManager wm = a.getWindowManager();

......

//当前window的属性，从代码跟踪来看是PhoneWindow窗口的属性

WindowManager.LayoutParams l = r.window.getAttributes();

......

wm.addView(decor, l);

......}

ViewManager是一个接口，addView是其定义的一个方法，其实现类为WindowManagerImpl。wm.addView(decor, l)中两个参数会层层传递，直到ViewRootImpl类中。

#### 2.1.1 参数decor

//=== PhoneWindow.java ===

// This is the top-level view of the window, containing the window decor.private DecorView mDecor;......

public PhoneWindow(...){

......

mDecor = (DecorView) preservedWindow.getDecorView();

......}

@Overridepublic final View getDecorView() {

......

return mDecor;

}

decor是DecorView实例，它是window的顶级视图。其类继承关系为：**DecorView -> FrameLayout -> ViewGroup -> View**

#### 2.1.2 参数l

//=== Window.java ===

private final WindowManager.LayoutParams mWindowAttributes = new WindowManager.LayoutParams();......

public final WindowManager.LayoutParams getAttributes() {

return mWindowAttributes;

}

//=== WindowManager内部类LayoutParams ===

public static class LayoutParams extends ViewGroup.LayoutParams implements Parcelable {

public LayoutParams() {

super(LayoutParams.MATCH\_PARENT, LayoutParams.MATCH\_PARENT);

......

}}

//=== ViewGroup.java内部类LayoutParams ===

public LayoutParams(int width, int height) {

this.width = width;

this.height = height;}

参数l：表示PhoneWindow的LayoutParams属性，其width和height值均为LayoutParams.MATCH\_PARENT。

### 2.2 performTraversals()

performTraversals方法有大约800多行代码，控制着整个绘制流程，关键代码如下：

// === ViewRootImpl.java ===

private void performTraversals() {

......

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

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

......

// Ask host how big it wants to be

performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);

......

performLayout(lp, mWidth, mHeight);

......

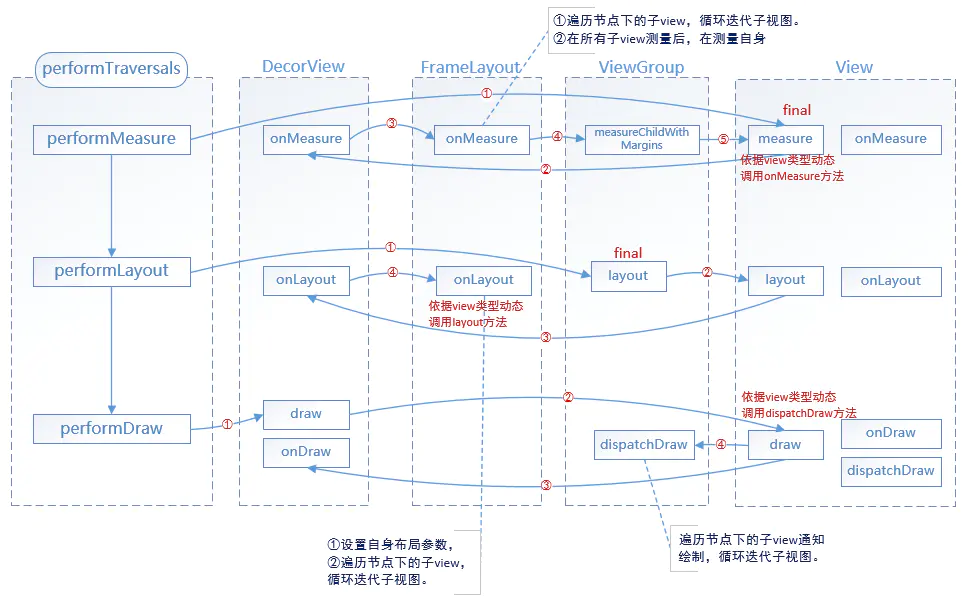
performDraw();}

上述代码是绘制流程的完成过程，涉及三个步骤：  
1）performMeasure()：从根节点向下遍历View树，完成所有ViewGroup和View的测量工作，计算出所有ViewGroup和View显示出来需要的高度和宽度；

2）performLayout()：从根节点向下遍历View树，完成所有ViewGroup和View的布局计算工作，根据测量出来的宽高及自身属性，计算出所有ViewGroup和View显示在屏幕上的区域；

3）performDraw()：从根节点向下遍历View树，完成所有ViewGroup和View的绘制工作，根据布局过程计算出的显示区域，将所有View的当前需显示的内容画到屏幕上。

performTraversals整体过程如下图所示：



## View绘制的三个流程

一个完整的绘制流程包括measure、layout、draw三个步骤，其中：

* measure（测量）  
  系统会先根据xml布局文件和代码中对控件属性的设置，来计算出每个View和ViewGrop的尺寸，并将这些尺寸保存下来。
* layout（布局）  
  根据测量出的结果以及对应的参数，来确定每一个控件应该显示的位置。
* draw（绘制）  
  确定好位置后，就将这些控件绘制到屏幕上。

### 3.1 measure过程分析

#### 3.1.1 MeasureSpec介绍

public static class MeasureSpec {

private static final int MODE\_SHIFT = 30;

private static final int MODE\_MASK = 0x3 << MODE\_SHIFT;

......

/\*\*

\* Measure specification mode: The parent has not imposed any constraint

\* on the child. It can be whatever size it wants.

\*/

public static final int UNSPECIFIED = 0 << MODE\_SHIFT;

/\*\*

\* Measure specification mode: The parent has determined an exact size

\* for the child. The child is going to be given those bounds regardless

\* of how big it wants to be.

\*/

public static final int EXACTLY = 1 << MODE\_SHIFT;

/\*\*

\* Measure specification mode: The child can be as large as it wants up

\* to the specified size.

\*/

public static final int AT\_MOST = 2 << MODE\_SHIFT;

......

/\*\*

\* Creates a measure specification based on the supplied size and mode.

\*......

\*@return the measure specification based on size and mode

\*/

public static int makeMeasureSpec(@IntRange(from = 0, to = (1 << MeasureSpec.MODE\_SHIFT) - 1) int size, @MeasureSpecMode int mode) {

if (sUseBrokenMakeMeasureSpec) {

return size + mode;

} else {

return (size & ~MODE\_MASK) | (mode & MODE\_MASK);

}

......

}

......

/\*\*

\* Extracts the mode from the supplied measure specification.

\*......

\*/

@MeasureSpecMode

public static int getMode(int measureSpec) {

//noinspection ResourceType

return (measureSpec & MODE\_MASK);

}

/\*\*

\* Extracts the size from the supplied measure specification.

\*......

\* @return the size in pixels defined in the supplied measure specification

\*/

public static int getSize(int measureSpec) {

return (measureSpec & ~MODE\_MASK);

}

......}

MeasureSpec含义  
MeasureSpec概括了从父布局传递给子view布局要求。MeasureSpec是32位的int值，高2位代表SpecMode（模式），低30位代表SepcSize（尺寸），这样的打包方式好处是避免过多的对象内存分配。其结构示意图如下：

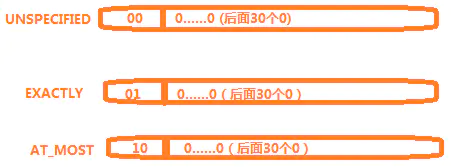


三种模式

UNSPECIFIED：未指定尺寸模式。父容器不对View有任何限制，要多大就给多大。

EXACTLY：精确值模式。父布局决定了子view的准确尺寸，子view无论想设置多大的值，都将限定在那个边界内。

AT\_MOST：最大值模式。子view可以一直大到指定的值。（笔者注：宽高属性设置为wrap\_content，那么它的最大值不会超过父布局给定的值，所以称为最大值模式）



主要方法

| **方法** | **含义** |
| --- | --- |
| makeMeasureSpec | 用于将mode和size打包成一个int型的MeasureSpec |
| getMode | 从指定的measureSpec值中获取其mode |
| getSize | 从指定的measureSpec值中获取其size |

#### 3.1.2 ViewGroup.LayoutParams介绍

//=== ViewGroup.java ===

public static class LayoutParams {

......

/\*\*

\* Special value for the height or width requested by a View.

\* MATCH\_PARENT means that the view wants to be as big as its parent,

\* minus the parent's padding, if any. Introduced in API Level 8.

\*/

public static final int MATCH\_PARENT = -1;

/\*\*

\* Special value for the height or width requested by a View.

\* WRAP\_CONTENT means that the view wants to be just large enough to fit

\* its own internal content, taking its own padding into account.

\*/

public static final int WRAP\_CONTENT = -2;

/\*\*

\* Information about how wide the view wants to be. Can be one of the

\* constants FILL\_PARENT (replaced by MATCH\_PARENT

\* in API Level 8) or WRAP\_CONTENT, or an exact size.

\*/

public int width;

/\*\*

\* Information about how tall the view wants to be. Can be one of the

\* constants FILL\_PARENT (replaced by MATCH\_PARENT

\* in API Level 8) or WRAP\_CONTENT, or an exact size.

\*/

public int height;

......}

LayoutParams含义  
View用LayoutParams告诉父布局，它们想要怎样被布局。其width和height属性对应着layout\_width和layout\_height属性。ViewGroup不同的子类，会定义出不同LayoutParams子类。

LayoutParams取值  
LayoutParams指定三种数值：MATCH\_PARENT、WRAP\_CONTENT、具体数值；

* MATCH\_PARENT：该view希望和父布局尺寸一样大。
* WRAP\_CONTENT：该view希望其大小为仅仅足够包裹住其内容即可。

#### 3.1.3 View测量流程

##### 3.1.3.1 ViewRootImpl.performMeasure()

//=== ViewRootImpl.java ===

public final class ViewRootImpl implements ViewParent, View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {

public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {

......

mView = view;

......

mWindowAttributes.copyFrom(attrs);

......

}

private void performMeasure(int childWidthMeasureSpec, int childHeightMeasureSpec) {

......

mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);

......

}}

① setView的参数view和attrs是ActivityThread类中addView方法传递过来的，可以确定mView指的是DecorView。  
② 在performMeasure()中，其实是DecorView在执行measure()操作。**childWidthMeasureSpec和childHeightMeasureSpec由来**

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

getRootMeasureSpec(int,int)方法的完整源码如下所示：

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;

}

① 基于window的layout params，在window中为root view获取measureSpec。  
② 参数windowSize：window的可用宽度和高度值；参数rootDimension：window的宽/高的layout param值。

##### 3.1.3.2 View.measure()

尽管mView就是DecorView，但是由于measure()方法是final型的，View子类都不能重写该方法，所以这里追踪measure()的时候就直接进入到View类中了，这里贴出关键流程代码：

//=== View.java ===

public final void measure(int widthMeasureSpec, int heightMeasureSpec) {

......

// measure ourselves, this should set the measured dimension flag back

onMeasure(widthMeasureSpec, heightMeasureSpec);

......

}

① 系统将measure方法定义为final，说明系统不希望整个测量流程框架被修改。  
② view的实际测量工作放在onMeasure方法实现的？  
③ 参数widthMeasureSpec：父布局加入的水平空间要求；参数heightMeasureSpec：父布局加入的垂直空间要求。

##### 3.1.3.3 View.onMeasure()

//=== View.java ===

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),

getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));}

① 当重写该方法时，必须调用setMeasuredDimension(int,int)来存储该view测量出的宽和高，否则会触发IllegalStateException，由measure(int,int)抛出。调用基类的onMeasure(int,int)方法是一个有效的方法。  
② ViewGroup的子类必须重写该方法，才能绘制该容器内的子view。如果是自定义一个子控件，extends View，那么并不是必须重写该方法；  
③ 容器类控件都是ViewGroup的子类，如FrameLayout、LinearLayout等，都会重写onMeasure方法，根据自己的特性来进行测量；如果是叶子节点view，即最里层的控件，如TextView等，也可能会重写onMeasure方法，所以当流程走到onMeasure(...)时，流程可能就会切到那些重写的onMeasure()方法中去。  
④ widthMeasureSpec：父布局加入的水平空间要求；heightMeasureSpec：父布局加入的垂直空间要求。  
⑤ 如果该方法被重写，子类负责确保测量的高和宽至少是该view的mininum高度和mininum宽度值（getSuggestedMininumHeight()和getSuggestedMininumWidth()）；

**getSuggestedMinimumWidth()方法**

protected int getSuggestedMinimumWidth() {

return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.getMinimumWidth());

}

"mininum width“指的是在xml布局文件中该view的“android:minWidth"属性值，“background's minimum width”值是指“android:background”的宽度。该方法的返回值就是两者之间较大的那一个值，用来作为该view的最小宽度值。现在应该很容易理解了吧，当一个view在layout文件中同时设置了这两个属性时，为了两个条件都满足，自然要选择值大一点的那个了。

**getDefaultSize()方法**

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;

}

① 参数由widthMeasureSpec变成了measuredWidth，即由“父布局加入的水平空间要求”转变为了view的宽度，measuredHeigh也是一样。  
② 如果父布局没有施加任何限制，即MeasureSpec的mode为UNSPECIFIED，那么返回值为参数中提供的size值。如果父布局施加了限制，则返回的默认尺寸为保存在参数measureSpec中的specSize值。

##### 3.1.3.4 View.setMeasuredDimension()

protected final void setMeasuredDimension(int measuredWidth, int measuredHeight) {

......

setMeasuredDimensionRaw(measuredWidth, measuredHeight);

}

① measuredWidth：该view被测量出宽度值；measuredHeight：该view被测量出的高度值。

private void setMeasuredDimensionRaw(int measuredWidth, int measuredHeight) {

mMeasuredWidth = measuredWidth;

mMeasuredHeight = measuredHeight;

......

}

View中的成员变量mMeasureWidth和mMeasureHeight就被赋值了，这也就意味着，View的测量就结束了。

//=== View.java ===

public static final int MEASURED\_SIZE\_MASK = 0x00ffffff;

public final int getMeasuredWidth() {

return mMeasuredWidth & MEASURED\_SIZE\_MASK;

}

public final int getMeasuredHeight() {

return mMeasuredHeight & MEASURED\_SIZE\_MASK;

}

① 获取原始的测量宽度和高度，这两个方法需在setMeasuredDimension()方法执行后才有效，否则返回值为0。

#### 3.1.4 DecorView测量过程

##### 3.1.4.1 DecorView.onMeasure()

//=== DecorView.java ===

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

......

super.onMeasure(widthMeasureSpec, heightMeasureSpec);

......

}

//=== FrameLayout.java ===

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

int count = getChildCount();

for (int i = 0; i < count; i++) {

final View child = getChildAt(i);

......

measureChildWithMargins(child, widthMeasureSpec, 0, heightMeasureSpec, 0);

......

}

......

setMeasuredDimension(......)

}

① DecorView的继承链：DecorView extends FrameLayout extends ViewGroup extends View。当DecorView第一次调用到measure()方法后，流程就开始切换到重写的onMeasure()中。DecorView在onMeasure()方法做一些事项后，调用父类的onMeasure方法。  
② FrameLayout对OnMeasure()方法进行重写，当所有子view测量完成后，最后调用setMeasuredDimension(...)来测量自己的。

##### 3.1.4.2 ViewGroup.measureChildWithMargins()

**measureChild()方法和measureChildWithMargins()**

protected void measureChild(View child, int parentWidthMeasureSpec,

int parentHeightMeasureSpec) {

final LayoutParams lp = child.getLayoutParams();

final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,

mPaddingLeft + mPaddingRight, lp.width);

final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,

mPaddingTop + mPaddingBottom, lp.height);

child.measure(childWidthMeasureSpec, childHeightMeasureSpec);}

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);

}

① measureChildWithMargins在measureChild的基础上增加：已使用的宽高、margin值。其实它们的功能都是一样的，最后都是生成子View的MeasureSpec，并传递给子View继续测量，即最后一句代码child.measure(childWidthMeasureSpec, childHeightMeasureSpec)。  
② 在FrameLayout和LinearLayout中重写的onMeasure方法中调用的就是后者，而AbsoluteLayout中就是间接地调用的前者。而RelativeLayout中，两者都没有调用，而是自己写了一套方法，不过该方法和后者方法仅略有差别，但基本功能还是一样。

**getChildMeasureSpec()方法**  
目的：将父布局传递来的MeasureSpec和其子view的LayoutParams，整合成子View的MeasureSpec。

// spec参数 表示父View的MeasureSpec

// padding参数 父View的Padding+子View的Margin，父View的大小减去这些边距，才能精确算出子View的MeasureSpec的size

// childDimension参数 表示该子View内部LayoutParams属性的值（lp.width或者lp.height）可以是wrap\_content、match\_parent、一个精确指(an exactly size),

public static int getChildMeasureSpec(int spec, int padding, int childDimension) {

int specMode = MeasureSpec.getMode(spec); //获得父View的mode

int specSize = MeasureSpec.getSize(spec); //获得父View的大小

//父View的大小-自己的Padding+子View的Margin，得到值才是子View的大小。

int size = Math.max(0, specSize - padding);

int resultSize = 0; //初始化值，最后通过这个两个值生成子View的MeasureSpec

int resultMode = 0; //初始化值，最后通过这个两个值生成子View的MeasureSpec

switch (specMode) {

// Parent has imposed an exact size on us

//1、父View是EXACTLY的 ！

case MeasureSpec.EXACTLY:

//1.1、子View的width或height是个精确值 (an exactly size)

if (childDimension >= 0) {

resultSize = childDimension; //size为精确值

resultMode = MeasureSpec.EXACTLY; //mode为 EXACTLY 。

}

//1.2、子View的width或height为 MATCH\_PARENT/FILL\_PARENT

else if (childDimension == LayoutParams.MATCH\_PARENT) {

// Child wants to be our size. So be it.

resultSize = size; //size为父视图大小

resultMode = MeasureSpec.EXACTLY; //mode为 EXACTLY 。

}

//1.3、子View的width或height为 WRAP\_CONTENT

else if (childDimension == LayoutParams.WRAP\_CONTENT) {

// Child wants to determine its own size. It can't be

// bigger than us.

resultSize = size; //size为父视图大小

resultMode = MeasureSpec.AT\_MOST; //mode为AT\_MOST 。

}

break;

// Parent has imposed a maximum size on us

//2、父View是AT\_MOST的 ！

case MeasureSpec.AT\_MOST:

//2.1、子View的width或height是个精确值 (an exactly size)

if (childDimension >= 0) {

// Child wants a specific size... so be it

resultSize = childDimension; //size为精确值

resultMode = MeasureSpec.EXACTLY; //mode为 EXACTLY 。

}

//2.2、子View的width或height为 MATCH\_PARENT/FILL\_PARENT

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; //size为父视图大小

resultMode = MeasureSpec.AT\_MOST; //mode为AT\_MOST

}

//2.3、子View的width或height为 WRAP\_CONTENT

else if (childDimension == LayoutParams.WRAP\_CONTENT) {

// Child wants to determine its own size. It can't be

// bigger than us.

resultSize = size; //size为父视图大小

resultMode = MeasureSpec.AT\_MOST; //mode为AT\_MOST

}

break;

// Parent asked to see how big we want to be

//3、父View是UNSPECIFIED的 ！

case MeasureSpec.UNSPECIFIED:

//3.1、子View的width或height是个精确值 (an exactly size)

if (childDimension >= 0) {

// Child wants a specific size... let him have it

resultSize = childDimension; //size为精确值

resultMode = MeasureSpec.EXACTLY; //mode为 EXACTLY

}

//3.2、子View的width或height为 MATCH\_PARENT/FILL\_PARENT

else if (childDimension == LayoutParams.MATCH\_PARENT) {

// Child wants to be our size... find out how big it should

// be

resultSize = 0; //size为0！ ,其值未定

resultMode = MeasureSpec.UNSPECIFIED; //mode为 UNSPECIFIED

}

//3.3、子View的width或height为 WRAP\_CONTENT

else if (childDimension == LayoutParams.WRAP\_CONTENT) {

// Child wants to determine its own size.... find out how

// big it should be

resultSize = 0; //size为0! ，其值未定

resultMode = MeasureSpec.UNSPECIFIED; //mode为 UNSPECIFIED

}

break;

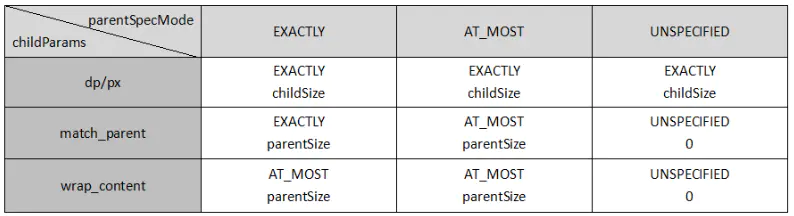
}

//根据上面逻辑条件获取的mode和size构建MeasureSpec对象。

return MeasureSpec.makeMeasureSpec(resultSize, resultMode);

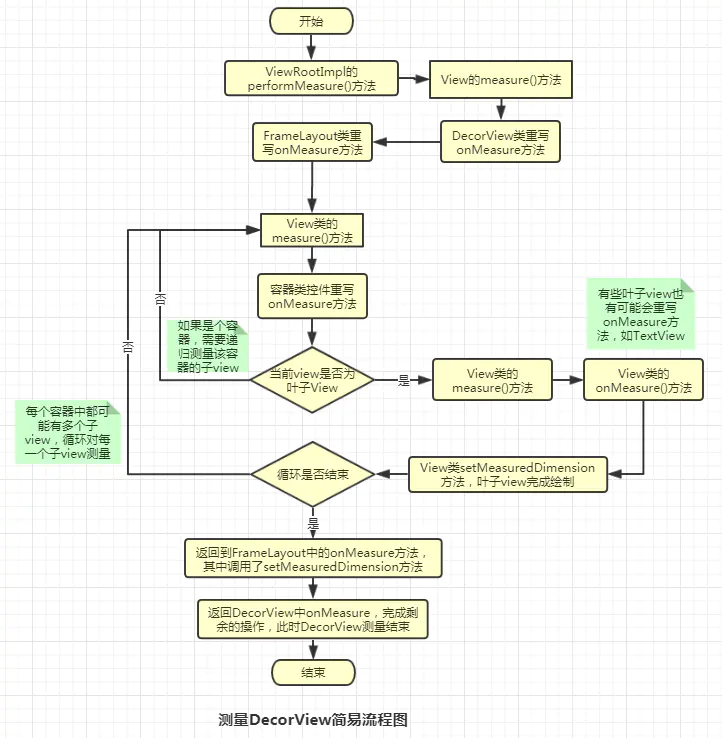
}

pecMode和specSize分别是父布局传下来的要求，size的值是父布局尺寸要求减去其padding值，最小不会小于0。代码最后就是将重新得到的mode和size组合生成一个新的MeasureSpec，传递给子View，一直递归下去。本段代码重难点就是这里新mode和新size值的确定，specMode和childDimension各有3种值，所以最后会有9种组合。总结图形如下：



* 如果specMode的值为MeasureSpec.EXACTLY，即父布局对子view的尺寸要求是一个精确值，这有两种情况，父布局中layout\_width属性值被设置为具体值，或者match\_parent，它们都被定义为精确值。子view的childDimension讨论：  
  ① childDimension值为具体数值时，此时resultSize为childDimension的精确值，resultMode理所当然为MeasureSpec.EXACTLY。这里不知道读者会不会又疑问，如果子View的layout\_width值比父布局的大，那这个结论还成立吗？按照我们的经验，似乎不太能理解，因为子view的宽度再怎么样也不会比父布局大。事实上，我们平时经验看到的，是最后布局后绘制出来的结果，而当前步骤为测量值，是有差别的。读者可以自定义一个View，将父布局layout\_width设置为100px，该自定义的子view则设置为200px，然后在子view中重写的onMeasure方法中打印出getMeasuredWidth()值看看，其值一定是200。甚至如果子view设置的值超过屏幕尺寸，其打印值也是设置的值。  
  ② childDimension值为LayoutParams.MATCH\_PARENT时。这个容易理解，它的尺寸和父布局一样，也是个精确值，所以resultSize为前面求出的size值，由父布局决定，resultMode为MeasureSpec.EXACTLY。  
  ③ childDimension值为LayoutParams.WRAP\_CONTENT时。当子view的layout\_width被设置为wrap\_content时，子view最多能够达到父视图的大小，所以resultSize值为size大小，resultMode为MeasureSpec.AT\_MOST。
* 如果specMode值为MeasureSpec.AT\_MOST，父视图对应于layout\_width为wrap\_content。子view的childDimension讨论：  
  ① childDimension为精确值时。很容易明确specSize为自身的精确值，specMode为MeasureSpec.EXACTLY。  
  ② childDimension为LayoutParams.MATCH\_PARENT时。specSize由父布局决定，specSize为size，specMode为MeasureSpec.AT\_MOST。  
  ③ childDimension为LayoutParams.WRAP\_CONTENT时。specSize由父布局决定，specSize为size，specMode为MeasureSpec.AT\_MOST。
* 如果specMode值为MeasureSpec.UNSPECIFIED。前面说过，平时很少用，一般用在系统中，不过这里还是简单说明一下。这一段有个变量View.sUseZeroUnspecifiedMeasureSpec，它是用于表示当前的目标api是否低于23（对应系统版本为Android M）的，低于23则为true，否则为false。现在系统版本基本上都是Android M及以上的，所以这里该值我们当成false来处理。  
  ① childDimension为精确值时。很容易明确specSize为自身的精确值，specMode为MeasureSpec.EXACTLY。  
  ② childDimension为LayoutParams.MATCH\_PARENT时。specSize由父布局决定为size，specMode和父布局一样，为MeasureSpec.UNSPECIFIED。  
  ③ childDimension为LayoutParams.WRAP\_CONTENT时。specSize由父布局决定为size，specMode和父布局一样，为MeasureSpec.UNSPECIFIED。

##### 3.1.4.3 DecorView视图树measure流程图



### 3.2 layout过程分析

#### 3.2.1 View布局流程

##### 3.2.1.1 ViewRootImpl.performLayout()

//=== ViewRootImpl.java ===

private void performLayout(WindowManager.LayoutParams lp, int desiredWindowWidth, int desiredWindowHeight) {

......

final View host = mView;

......

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

......

}

mView就是DecorView，lp中width和height均为LayoutParams.MATCH\_PARENT。

##### 3.2.1.2 ViewGroup.layout()

//=== ViewGroup.java ===

@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;

}}

① 由于DecorView是一个容器，是ViewGroup子类，所以跟踪代码的时候，实际上是先进入到ViewGroup类中的layout方法中。在layout方法中调用View.layout()方法。  
② layout方法是final的，说明系统不希望自定的ViewGroup子类破坏layout流程。

##### 3.2.1.3 View.layout()

/\*\*

\* Assign a size and position to a view and all of its

\* descendants

\*

\* <p>This is the second phase of the layout mechanism.

\* (The first is measuring). In this phase, each parent calls

\* layout on all of its children to position them.

\* This is typically done using the child measurements

\* that were stored in the measure pass().</p>

\*

\* <p>Derived classes should not override this method.

\* Derived classes with children should override

\* onLayout. In that method, they should

\* call layout on each of their children.</p>

\*

\* @param l Left position, relative to parent

\* @param t Top position, relative to parent

\* @param r Right position, relative to parent

\* @param b Bottom position, relative to parent

\*/@SuppressWarnings({"unchecked"})

public void layout(int l, int t, int r, int b) {

......

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);

......

}

......

}

目的 ：根据子视图的大小以及布局参数将View树放到合适的位置上。  
① 给view和它的所有后代分配尺寸和位置。  
② 派生类不应该重写该方法，容器类应该重写onLayout方法。在重写的onLayout方法中，它们应该为每一子view调用layout方法进行布局。  
③ 参数依次为：Left、Top、Right、Bottom四个点相对父布局的位置。  
④ setOpticalFrame方法最后会调用setFrame方法，将布局信息进行保持。

**setFrame方法**

//=================View.java================/\*\*

\* Assign a size and position to this view.

\*

\* This is called from layout.

\*

\* @param left Left position, relative to parent

\* @param top Top position, relative to parent

\* @param right Right position, relative to parent

\* @param bottom Bottom position, relative to parent

\* @return true if the new size and position are different than the

\* previous ones

\* {@hide}

\*/

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;

......

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);

mLeft = left;

mTop = top;

mRight = right;

mBottom = bottom;

......

}

return changed;}

① setFrame(l, t, r, b) 可以理解为给mLeft 、mTop、mRight、mBottom赋值，然后基本就能确定View自己在父视图的位置了，这几个值构成的矩形区域就是该View显示的位置，这里的具体位置都是相对与父视图的位置。  
② 返回值：如果新的尺寸和位置和之前的不同，返回true。

##### 3.2.1.4 View.onLayout()

//============View.java============

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

//=============ViewGroup.java===========

@Override

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

由于layout时已经将布局信息通过setFrame方法进行保存起来，在onLayout方法已经无须做额外事项，因此方法对于叶子view意义不大。但是对于容器类来说，需要一种遍历所有子view的机制，所以ViewGroup的子类需要重写此方法。

#### 3.2.2 DecorView布局过程

##### 3.2.2.1 DecorView.onLayout()

//==============DecorView.java================

@Override

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

super.onLayout(changed, left, top, right, bottom);

......

}

DecorView的onLayout方法会调用父类FrameLayout的onLayout方法。

##### 3.2.2.2 FrameLayout.onLayout()

//================FrameLayout.java==============

@Override

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

layoutChildren(left, top, right, bottom, false /\* no force left gravity \*/);

}

void layoutChildren(int left, int top, int right, int bottom, boolean forceLeftGravity) {

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();

......

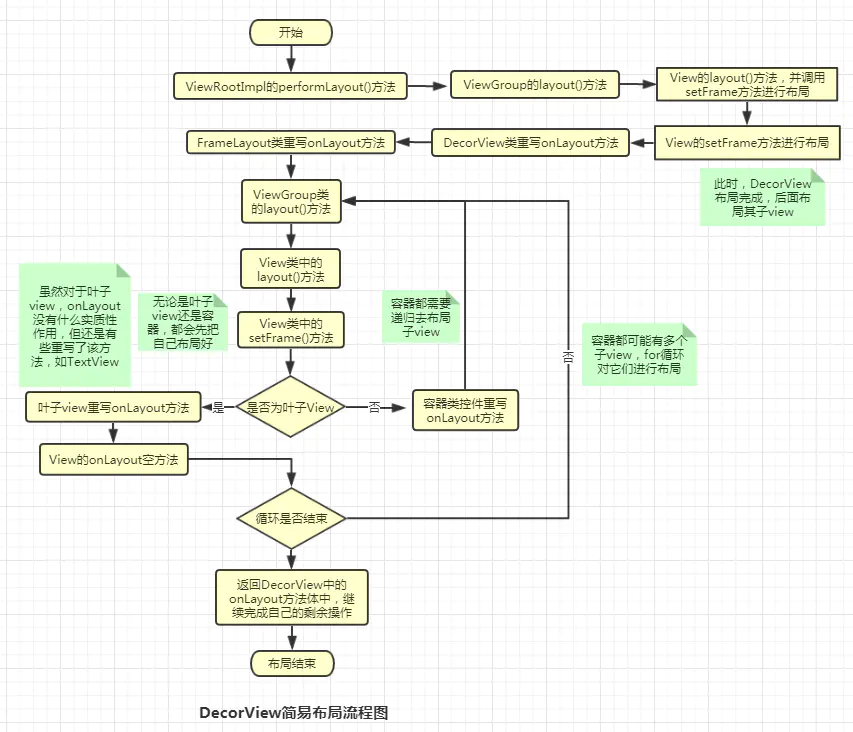
child.layout(childLeft, childTop, childLeft + width, childTop + height);

}

}}

① 对每一个child调用layout方法的，如果该child仍然是父布局，会继续递归下去；如果是叶子view，则会走到view的onLayout空方法，该叶子view布局流程走完。  
② width和height分别来源于measure阶段存储的测量值，如果这里通过其它渠道赋给width和height值，那么measure阶段就不需要了。

##### 3.2.2.3 DecorView的布局流程图



### 3.3 draw过程分析

#### 3.3.1 View绘制流程

##### 3.3.1.1 ViewRootImpl.performDraw()

//=== ViewRootImpl.java ===

private void performDraw() {

......

boolean canUseAsync = draw(fullRedrawNeeded);

......

}

private boolean draw(boolean fullRedrawNeeded) {

......

if (!drawSoftware(surface, mAttachInfo, xOffset, yOffset, scalingRequired, dirty, surfaceInsets)) {

return false;

}

......

}

private boolean drawSoftware(...){

......

mView.draw(canvas);

......

}

mView就是DecorView，这样就开始了DecorView视图树的draw流程了。

##### 3.3.1.2 DecorView.draw()

//================DecorView.java==============

@Override

public void draw(Canvas canvas) {

super.draw(canvas);

if (mMenuBackground != null) {

mMenuBackground.draw(canvas);

}}

用完super.draw后，还画了菜单背景，由于FrameLayout和ViewGroup都没有重写该方法，所以就直接进入都了View类中的draw方法了。

##### 3.3.1.3 View.draw()

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

...

background.draw(canvas);

...

// skip step 2 & 5 if possible (common case)

...

// Step 2, save the canvas' layers

...

if (solidColor == 0) {

final int flags = Canvas.HAS\_ALPHA\_LAYER\_SAVE\_FLAG;

if (drawTop) {

canvas.saveLayer(left, top, right, top + length, null, flags);

}

...

// Step 3, draw the content

if (!dirtyOpaque) onDraw(canvas);

// Step 4, draw the children

dispatchDraw(canvas);

// Step 5, draw the fade effect and restore layers

if (drawTop) {

matrix.setScale(1, fadeHeight \* topFadeStrength);

matrix.postTranslate(left, top);

fade.setLocalMatrix(matrix);

canvas.drawRect(left, top, right, top + length, p);

}

...

// Step 6, draw decorations (scrollbars)

onDrawScrollBars(canvas);}

从代码上看，这里做了很多工作，咱们简单说明一下，有助于理解这个“画”工作。

第1步：背景绘制  
对应我我们在xml布局文件中设置的“android:background”属性，这是整个“画”过程的第一步，这一步是不重点，知道这里干了什么就行。

第3步，对View的内容进行绘制  
onDraw(canvas) 方法是view用来draw 自己的，具体如何绘制，颜色线条什么样式就需要子View自己去实现，View.java 的onDraw(canvas) 是空实现，ViewGroup 也没有实现，每个View的内容是各不相同的，所以需要由子类去实现具体逻辑。

第4步：对当前View的所有子View进行绘制  
dispatchDraw(canvas) 方法是用来绘制子View的，View.java 的dispatchDraw()方法是一个空方法,因为View没有子View,不需要实现dispatchDraw ()方法，ViewGroup就不一样了，它实现了dispatchDraw ()方法。

第6步：画装饰。  
这里指画滚动条和前景，其实平时的每一个view都有滚动条，只是没有显示而已。同样这也不是重点，知道做了这些事就行。

##### 3.3.1.4 View.onDraw()

//=== View.java ===/\*\*

\* Implement this to do your drawing.

\*

\* @param canvas the canvas on which the background will be drawn

\*/

protected void onDraw(Canvas canvas) {}

实现该方法来做“画”工作。也就是说，具体的view需要重写该方法，来画自己想展示的东西，如文字，线条等。

##### 3.3.1.4 ViewGroup.dispathcDraw()

**View.dispatchDraw()**

//=== View.java ===/\*\*

\* Called by draw to draw the child views. This may be overridden

\* by derived classes to gain control just before its children are drawn

\* (but after its own view has been drawn).

\* @param canvas the canvas on which to draw the view

\*/

protected void dispatchDraw(Canvas canvas) {}

view没有子视图，不需要进行绘制派发。

**ViewGroup.dispatchDraw()**

//=== ViewGroup.java ===@Overrideprotected void dispatchDraw(Canvas canvas) {

final int childrenCount = mChildrenCount;

final View[] children = mChildren;

......

for (int i = 0; i < childrenCount; i++) {

more |= drawChild(canvas, child, drawingTime);

......

}

......

}

平时常用的LinearLayout、FrameLayout、RelativeLayout等常用的布局控件，都没有再重写该方法，DecorView中也一样，而是只在ViewGroup中实现了dispatchDraw方法的重写。所以当DecorView执行完onDraw方法后，流程就会切到ViewGroup中的dispatchDraw方法了。

**View.drawChild()**

//=== ViewGroup.java ===

protected boolean drawChild(Canvas canvas, View child, long drawingTime) {

return child.draw(canvas, this, drawingTime);

}

画当前ViewGroup中的某一个子view，其中参数drawingTime表示“画”动作发生的时间点。

**View.draw(...)**

//=== View.java ===boolean draw(Canvas canvas, ViewGroup parent, long drawingTime) {

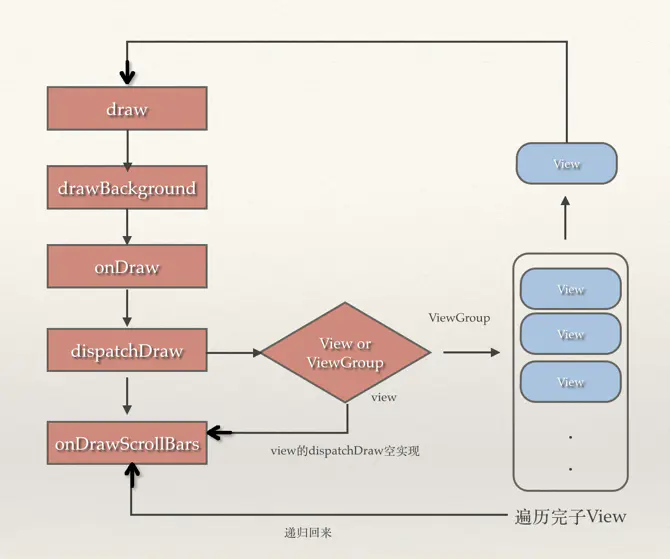
......

draw(canvas);

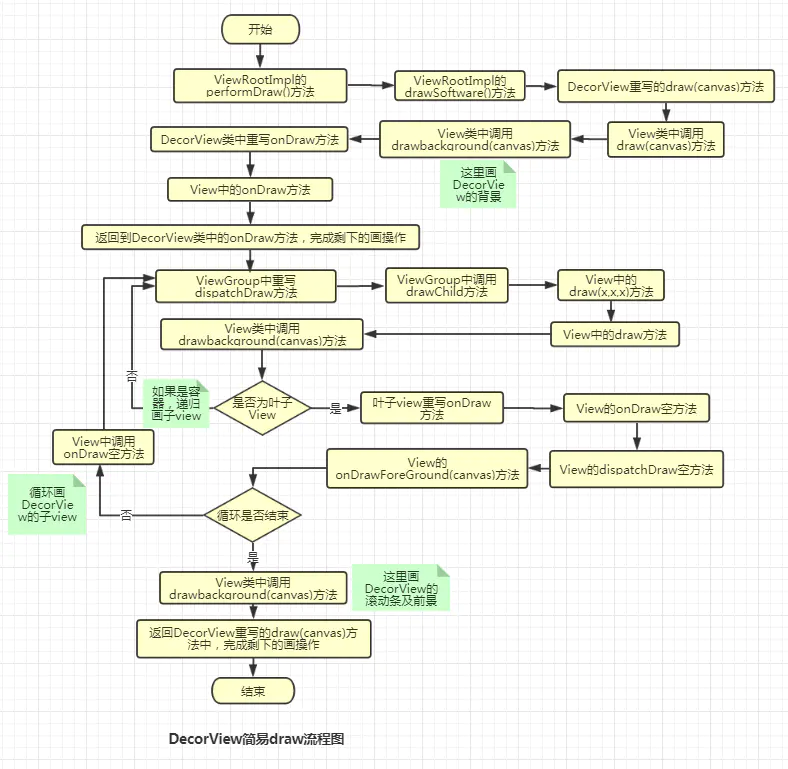
......

}

##### 3.3.1.5 draw的绘制流程



#### 3.3.2 DecorView绘制流程图



### 3.4 绘制过程小结

到目前为止，View的绘制流程就介绍完了。根节点是DecorView，整个View体系就是一棵以DecorView为根的View树，依次通过遍历来完成measure、layout和draw过程。而如果要自定义view，一般都是通过重写onMeasure()，onLayout()，onDraw()来完成要自定义的部分，整个绘制流程也基本上是围绕着这几个核心的地方来展开的。整个绘制过程流程示意图如下：

