# View的绘制

## 一．相关概念

## 1.1 Window系列

Window：Window是一个抽象类，PhoneWindow继承Window，是实际使用的类。

PhoneWindow：activity就是用于展示界面，控制用户交互的，所以activity中一定维护了用于描述界面的对象，还有就是控制窗口显示的管理器对象，实际上这两个对象都是由PhoneWindow维护的，而每个activity都拥有一个PhoneWindow对象mWindow；

### 1.1.1 Window

public abstract class Window {

private WindowManager mWindowManager;

public void setWindowManager(android.view.WindowManager wm, IBinder appToken, String appName,

boolean hardwareAccelerated) {

...

//调用WindowManagerImpl的静态方法new一个WindowManagerImpl对象

mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);

}

public android.view.WindowManager getWindowManager() {

return mWindowManager;

}

}

### 1.1.2 PhoneWindow

public class PhoneWindow extends Window implements MenuBuilder.Callback {

private DecorView mDecor;

}

## 1.2 WindowManager系列3

WindowManager系列，用于管理窗口视图的显示、添加、移除和状态更新等。

**ViewManager**：WindowManager继承了这个接口，实现了addview等方法，这个addview，是第一次绘制，发起的起点。

**WindowManager**：WindowManager也是一个接口，它是由WindowManagerImpl实现的

WindowManagerImpl：它保存了一个WindowManagerGlobal的单例：

private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();

最终实现addview等方法的地方，都是由mGlobal来实现的，它才是最终实现的地方。

**WindowManagerGlobal**:每个activity都有windowManager，但是每个应用只有一个WindowManagerGlobal，它持有WMS的句柄，负责和WMS通信。WindowManagerGlobal是单例模式，Activity在创建完成调用attach()方法初始化时，会实例化mWindowManager（WindowManager的引用指向WindowManagerImpl的对象），WindowManagerImpl创建之后，就维护了WindowManagerGlobal的单例对象，WindowManagerGlobal中有一个IWindowSession类型的变量sWindowSession，他就是用来和远程服务WindowManagerService通信的代理。

**WindowManagerService**：跟之前讲解Activity启动过程中提到的ActivityManagerService(用于管理Android组件的启动、关闭和状态信息等)一样，WMS也是一个复杂的系统服务。   
  在了解WMS之前，首先要了解窗口（Window）是什么，Android系统中的窗口是只屏幕上一块用于绘制UI元素并能相应用户交互的矩形区域，从原理上讲，窗口是独自占有一个Surface实例的显示区域，例如Dialog、Activity界面、比值、状态栏、Toast等都是窗口。Surface就是一块画布，应用可以通过Cancas或者OpenGL在其上面作画，然后通过surfaceFlinger将多块Surface的内容按照特定的顺序进行混合并输出到FrameBuffer，从而将应用界面显示给用户。   
  既然每个窗口都有一块Surface供自己涂鸦，必然需要一个角色对所有窗口的Surface进行协调管理，WMS就是用来做这个事情的，WMS掌管Surface的显示顺序（Z-order）以及位置尺寸，控制窗口动画，并且还是输入系统的重要中转站。

### 1.2.1 ViewManager

public interface ViewManager{

public void addView(View view, ViewGroup.LayoutParams params);

public void updateViewLayout(View view, ViewGroup.LayoutParams params);

public void removeView(View view);

}

### 1.2.2 WindowManager

public interface WindowManager extends ViewManager {

...

}

### 1.2.3 WindowManagerImpl

public final class WindowManagerImpl implements WindowManager {

//每个WindowManagerImpl对象都保存了一个WindowManagerGlobal的单例

private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();

...

public WindowManagerImpl createLocalWindowManager(android.view.Window parentWindow) {

return new WindowManagerImpl(mDisplay, parentWindow);

}

@Override

public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {

applyDefaultToken(params);

//调用WindowManagerGlobal对象的addView()方法

mGlobal.addView(view, params, mDisplay, mParentWindow);

}

...

}

### 1.2.4 WindowManagerGlobal

public final class WindowManagerGlobal {

private static WindowManagerGlobal sDefaultWindowManager;

private static IWindowManager sWindowManagerService;

private static IWindowSession sWindowSession;

//单例

private WindowManagerGlobal() {

}

public static WindowManagerGlobal getInstance() {

synchronized (WindowManagerGlobal.class) {

if (sDefaultWindowManager == null) {

sDefaultWindowManager = new WindowManagerGlobal();

}

return sDefaultWindowManager;

}

}

/\*\*WindowManagerGlobal类被加载时，就获取到WindowManagerService对象\*/

public static void initialize() {

getWindowManagerService();

}

public static IWindowManager getWindowManagerService() {

//线程安全的

synchronized (WindowManagerGlobal.class) {

if (sWindowManagerService == null) {

sWindowManagerService = IWindowManager.Stub.asInterface(

ServiceManager.getService("window"));

}

return sWindowManagerService;

}

}

public static IWindowSession getWindowSession() {

//线程安全的

synchronized (WindowManagerGlobal.class) {

if (sWindowSession == null) {

try {

InputMethodManager imm = InputMethodManager.getInstance();

IWindowManager windowManager = getWindowManagerService();

sWindowSession = windowManager.openSession(

new IWindowSessionCallback.Stub() {

@Override

public void onAnimatorScaleChanged(float scale) {

ValueAnimator.setDurationScale(scale);

}

},

imm.getClient(), imm.getInputContext());

} catch (RemoteException e) {

Log.e(TAG, "Failed to open window session", e);

}

}

return sWindowSession;

}

}

}

# 二．绘制流程前戏(如何走到doTraversal)

## 2.1 ActivityThread.handleResumeActivity

handleResumeActivity，这段代码前面是处理一些Activity的信息，从这里开始，是处理View，核心的代码是：wm.addView(l, decor)，其中：

**1.wm是windowManagerImpl的对象。**

**2.Addview是接口ViewManager的方法，由windowManagerImpl实现,最终会调用WindowManagerGlobal的AddView**

**3.l是decorView。**

**4.decor是layoutParam**

WindowManager wm;

if(r.window == null && !a.mFinished && willBeVisible) {  
 r.window = r.activity.getWindow();  
 View l = r.window.getDecorView();  
 l.setVisibility(View.INVISIBLE);  
 wm = a.getWindowManager();  
 LayoutParams decor = r.window.getAttributes();  
 a.mDecor = l;  
 decor.type = 1;  
 decor.softInputMode |= forwardBit;  
 if(a.mVisibleFromClient) {  
 a.mWindowAdded = true;  
 wm.addView(l, decor);  
 }  
} else if(!willBeVisible) {  
 r.hideForNow = true;  
}

## 2.2 WindowManagerGlobal.addView

重要的代码有两处

### 2.2.1 WindowManagerGlobal.add

因为WindowManagerGlobal是单例的，所以保留了当前进程的所有界面的元素，包括view，viewRoot，param，所以要把新加的view，添加到相应ArrayList里面

this.mViews.add(view);  
this.mRoots.add(viewRoot);  
this.mParams.add(wparams);

### 2.2.2 ViewRootImpl.setView

viewRoot.setView,是最后实现的地方，我们接着这个往下看。

viewRoot.setView(view, wparams, panelParentView);

public void addView(View view, android.view.ViewGroup.LayoutParams params, Display display, Window parentWindow)

{  
 …  
 View panelParentView = null;  
 ViewRootImpl viewRoot;  
 …  
 viewRoot = new ViewRootImpl(view.getContext(), display);  
 view.setLayoutParams(wparams);  
 this.mViews.add(view);  
 this.mRoots.add(viewRoot);  
 this.mParams.add(wparams);  
 }  
  
 try {  
 viewRoot.setView(view, wparams, panelParentView);  
 } catch (RuntimeException var15) {  
 …

}  
 }  
}

## 2.3 ViewRootImpl.setView()

ViewRootImpl实现了ViewParent，它并不是真正的View（没有继承View），它定义了顶层视图的职责API，比如requestLayout()请求布局等等，在setView()方法中，调用了requestLayout()方法，请求对Activity根窗口做第一次布局。

1.requestLayout()方法中首先调用checkThread()检查当前线程是否为UI线程然后调用scheduleTraversals()方法.

2.scheduleTraversals()方法中调用了mChoreographer.postCallback()，mChoreographer中维护了一个Handler，通过handler机制实现消息调度，并传入一个回调mTracersalRunnable。

3.mTracersalRunnable的run方法中调用了doTraversal()方法，doTraversal()方法继续调用performTraversals().

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

synchronized(this) {

…

this.requestLayout();

…

}

}

### 2.3.1 ViewRootImpl. requestLayout

public void requestLayout() {

if(!this.mHandlingLayoutInLayoutRequest) {

//检查是否主线程，就是这里咯  
 this.checkThread();  
 this.mLayoutRequested = true;

//发起绘制的步骤  
 this.scheduleTraversals();  
 }  
}

### 2.3.2 ViewRootImpl.scheduleTraversals

void scheduleTraversals() {

if(!this.mTraversalScheduled) {  
 this.mTraversalScheduled = true;  
 this.mTraversalBarrier = this.mHandler.getLooper().getQueue().postSyncBarrier();

//异步绘制？  
 this.mChoreographer.postCallback(2, this.mTraversalRunnable, (Object)null);  
 if(!this.mUnbufferedInputDispatch) {  
 this.scheduleConsumeBatchedInput();  
 }  
 this.notifyRendererOfFramePending();  
 this.pokeDrawLockIfNeeded();  
 }  
}

### 2.3.4 ViewRootImpl.TraversalRunnable

class TraversalRunnable implements Runnable {

TraversalRunnable() {  
 }  
  
 public void run() {  
 ViewRootImpl.this.doTraversal();  
 }  
}

### 2.3.4 ViewRootImpl.doTraversal

void doTraversal() {

if(this.mTraversalScheduled) {  
 this.mTraversalScheduled = false;  
 this.mHandler.getLooper().getQueue().removeSyncBarrier(this.mTraversalBarrier);  
 if(this.mProfile) {  
 Debug.startMethodTracing("ViewAncestor");  
 }  
 //最终绘制的地方  
 this.performTraversals();  
 if(this.mProfile) {  
 Debug.stopMethodTracing();  
 this.mProfile = false;  
 }  
 }  
}

# 三.绘制流程（measure，layout，draw）

### 3.1 ViewRootImpl.performTraversals()

performTraversals()方法中，分为四部分，

1. 确定Activity窗口的大小，也就是通过各种判断，是否是第一次？是否需要重新计算大小？最终确定Activity根窗口的大小并保存起来。
2. 第二步就是遍历测量子控件的大小；
3. 第三步就是布局；
4. 第四部就是绘制。

private void performTraversals() {

//mView保存在局部变量中，提高访问效率

//这个mView就是decorView  
 View host = this.mView;

//一些debug的打印  
 …  
 if(host != null && this.mAdded) {

//初始化一些其他的标志位  
 …

// mWinFrame表示了窗口的最新尺寸

Rect frame = this.mWinFrame;

//声明本阶段的主角，这两个变量将是mView的SPEC\_SIZE分量的候选  
 int desiredWindowWidth;  
 int desiredWindowHeight;

//mFirst表示了这是第一次遍历，此时窗口刚刚被添加到WMS，尚未进行relayout，

//因此mWinFrame中没有存储有效地窗口尺寸  
 if(this.mFirst) {  
 this.mFullRedrawNeeded = true;  
 this.mLayoutRequested = true;

if(lp.type != WindowManager.LayoutParams.TYPE\_STATUS\_BAR\_PANEL

&& lp.type != WindowManager.LayoutParams. TYPE\_INPUT\_METHOD) {

//第一次“遍历”的测量，采用了应用可以使用的最大尺寸作为SPEC\_SIZE的候选  
 DisplayMetrics packageMetrics =

host.getContext().getResources().getDisplayMetrics();  
 desiredWindowWidth = packageMetrics.widthPixels;  
 desiredWindowHeight = packageMetrics.heightPixels;  
 } else {

//为状态栏设置desiredWindowWidth和desiredWindowHeight，其取值是屏幕尺寸  
 Point insetsChanged = new Point();  
 this.mDisplay.getRealSize(insetsChanged);  
 desiredWindowWidth = insetsChanged.x;  
 desiredWindowHeight = insetsChanged.y;  
 }  
 //由于这是第一次进行“遍历”

//控件树即将第一次被显示在窗口上，因此接下来的代码填充了mAttachInfo中的一些字段

//然后通过mView发起了dispatchAttachedToWindow()的调用

//之后每一个位于控件树中的控件都会回调onAttachedToWindow()  
 attachInfo.mSurface = this.mSurface;  
 attachInfo.mUse32BitDrawingCache = true;  
 …

} else {

//在非第一次遍历的情况下，会采用窗口的最新尺寸作为SPEC\_SIZE的候选  
 desiredWindowWidth = frame.width();  
 desiredWindowHeight = frame.height();

//如果窗口的最新尺寸与ViewRootImpl中的现有尺寸不同

//说明WMS侧单方面改变了窗口的尺寸

//这将产生如下三个结果  
 if(desiredWindowWidth != this.mWidth || desiredWindowHeight != this.mHeight) {  
 // 需要进行完整的重绘以适应新的窗口尺寸

this.mFullRedrawNeeded = true;

// 需要对控件树进行重新布局  
 this.mLayoutRequested = true;

// 控件树有可能拒绝接受新的窗口尺寸

// 比如在随后的预测量中给出了不同于窗口尺寸的测量结果

// 产生这种情况时，就需要在窗口布局阶段尝试设置新的窗口尺寸  
 windowSizeMayChange = true;  
 }  
 }  
 //View的可见性改变时候，做的一些处理  
 …

//执行位于RunQueue中的回调。

//RunQueue是ViewRootImpl的一个静态成员，即是说它是进程唯一的，

//并且可以在进程的任何位置访问RunQueue。

//在进行多线程任务时，开发者可以通过调用View.post()将一个Runnable发送到主线程执行。

//这两个方法的原理是将Runnable对象发送到ViewRootImpl的mHandler去。

//当控件已经加入到控件树时，可以通过AttachInfo轻易获取这个Handler。

//而当控件没有位于控件树中时，则没有mAttachInfo可用

//此时执行View.post()/PostDelay()方法.Runnable将会被添加到这个RunQueue队列中。

//在这里，ViewRootImpl将会把RunQueue中的Runnable发送到mHandler中，进而得到执行。

//无论控件是否显示在控件树中，View.post()/postDelay()方法都是可用的

//除非当前进程中没有任何处于活动状态的ViewRootImpl  
 getRunQueue().executeActions(attachInfo.mHandler);  
 boolean var65 = false;  
 boolean layoutRequested = this.mLayoutRequested &&

!this.mStopped || this.mRequestLayoutByResize;

//仅当layoutRequested为true时才进行预测量。

//layoutRequested为true表示在进行“遍历”之前requestLayout()方法被调用过。

//requestLayout()方法用于要求ViewRootImpl进行一次“遍历”对控件树重新进行测量与布局  
 if(layoutRequested) {  
 Resources windowShouldResize = this.mView.getContext().getResources();  
 if(this.mFirst) {

//确定控件树是否需要进入TouchMode  
 this.mAttachInfo.mInTouchMode = !this.mAddedTouchMode;  
 this.ensureTouchModeLocally(this.mAddedTouchMode);  
 } else {

//检查WMS是否单方面改变了ContentInsets与VisibleInsets。注意对二者的处理的差异

//ContentInsets描述了控件在布局时必须预留的空间，这样会影响控件树的布局

//因此将insetsChanged标记为true，以此作为是否进行控件布局的条件之一。

//VisibleInsets则描述了被遮挡的空间，

//ViewRootImpl在进行绘制时，需要调整绘制位置以保证关键控件或区域

//如正在进行输入的TextView等不被遮挡，这样VisibleInsets的变化并不会导致重新布局

//所以这里仅仅是将VisibleInsets保存到mAttachInfo中，以便绘制时使用  
 if(!this.mPendingOverscanInsets.equals(this.mAttachInfo.mOverscanInsets)) {  
 insetsChanged = true;  
 }   
  
 if(!this.mPendingContentInsets.equals(this.mAttachInfo.mContentInsets)) {  
 insetsChanged = true;  
 }  
  
 if(!this.mPendingVisibleInsets.equals(this.mAttachInfo.mVisibleInsets)) {  
 this.mAttachInfo.mVisibleInsets.set(this.mPendingVisibleInsets);  
 }  
 }  
  
 if(lp.width == ViewGroup.LayoutParams.WRAP\_CONTENT

|| lp.height == ViewGroup.LayoutParams.WRAP\_CONTENT) {

//悬浮窗口的尺寸取决于测量结果。因此有可能需要向WMS申请改变窗口的尺寸  
 windowSizeMayChange = true;  
 if(lp.type != WindowManager.LayoutParams.TYPE\_STATUS\_BAR\_PANEL

&& lp.type != WindowManager.LayoutParams. TYPE\_INPUT\_METHOD) {  
 DisplayMetrics packageMetrics =windowShouldResize.getDisplayMetrics();  
 desiredWindowWidth = packageMetrics.widthPixels;  
 desiredWindowHeight = packageMetrics.heightPixels;  
 } else {  
 Point computesInternalInsets = new Point();  
 this.mDisplay.getRealSize(computesInternalInsets);  
 desiredWindowWidth = computesInternalInsets.x;  
 desiredWindowHeight = computesInternalInsets.y;  
 }  
 }  
 }  
 //进行预测量。通过measureHierarchy()方法以desiredWindowWidth/Height进行测量

//前面都是前戏，这个才是正经给desireWindowWidth赋值的地方  
 windowSizeMayChange |= this.measureHierarchy(host, lp, windowShouldResize,

desiredWindowWidth, desiredWindowHeight);  
 }

……  
 //前面的步骤主要是判断窗口大小（Activity，Dialog）

//后面的步骤，是负责View里面measure，layout，draw  
 boolean didLayout;  
 boolean triggerGlobalLayoutListener;  
 boolean skipDraw;  
 Rect i;  
 boolean var75;  
 int var78;

// 接下来的两段代码都是在满足下面的条件之一的情况下执行的：

// 1. Activity窗口是第一次执行测量、布局和绘制操作，

// 即ViewRoot类的成员变量mFirst的值等于true

// 2. 前面得到的变量windowShouldResize的值等于true，

// 即Activity窗口的大小的确是发生了变化。

// 3. 前面得到的变量insetsChanged的值等于true

// 即Activity窗口的内容区域边衬发生了变化。

// 4. Activity窗口的可见性发生了变化，即变量viewVisibilityChanged的值等于true。

// 5. Activity窗口的属性发生了变化，

// 即变量params指向了一个WindowManager.LayoutParams对象。  
 if(!this.mFirst && ! windowShouldResize&& ! insetsChanged

&& !viewVisibilityChanged && params == null) {  
 //什么也没有变化，就不用更新

…

} else {

//做一些surface的初始化和其他参数的初始化  
 …

try {  
 int cancelDraw = this.mSurface.getGenerationId();  
 i = new Rect(this.mWinFrame);

//★请求WindowManagerService服务计算Activity窗口的大小

//以及内容区域边衬大小和可见区域边衬大小  
 relayoutResult = this.relayoutWindow(params, viewVisibility, var71);  
 if(!this.mDrawDuringWindowsAnimating && (relayoutResult & 8) != 0) {  
 this.mWindowsAnimating = true;  
 }  
  
 …  
  
 //计算得到的Activity窗口的宽度和高度

//保存在ViewRoot类的成员变量mWidth和mHeight中  
 if(this.mWidth != frame.width() || this.mHeight != frame.height()) {  
 this.mWidth = frame.width();  
 this.mHeight = frame.height();  
 }

//surface的相关操作  
 …  
 if(!this.mStopped || this.mRequestLayoutByResize) {  
 focusChangedDueToTouchMode = this.ensureTouchModeLocally((relayoutResult & 1) != 0);  
 if(focusChangedDueToTouchMode || this.mWidth != host.getMeasuredWidth()

|| this.mHeight != host.getMeasuredHeight()

|| triggerGlobalLayoutListener) {  
 int childWidthMeasureSpec = getRootMeasureSpec(this.mWidth, lp.width);  
 int childHeightMeasureSpec = getRootMeasureSpec(this.mHeight, lp.height);

// 一、测量控件大小（根窗口和其子控件树）  
 this.performMeasure(childWidthMeasureSpec, childHightMeasureSpec);  
 int width = host.getMeasuredWidth();  
 int height= host.getMeasuredHeight();  
 measureAgain = false;

//有权重的，要performMeasure两次

//第一次要确定有多少空闲位置

//第二次按照权值分配这些位置   
 if(lp.horizontalWeight > 0.0F) {  
 width += (int)((float)(this.mWidth - width) \* lp.horizontalWeight);  
 childWidthMeasureSpec =

MeasureSpec.makeMeasureSpec(width, View.MeasureSpec.*EXACTLY* );  
 measureAgain = true;  
 }  
  
 if(lp.verticalWeight > 0.0F) {  
 height += (int)((float)(this.mHeight - height) \* lp.verticalWeight);  
 childHeightMeasureSpec =

MeasureSpec.makeMeasureSpec(height, View.MeasureSpec.*EXACTLY*);  
 measureAgain = true;  
 }  
 if(measureAgain) {  
 this.performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);  
 }  
 layoutRequested = true;  
 }  
 }  
 }  
  
 didLayout = layoutRequested && !this.mStopped || this.mRequestLayoutByResize;  
 this.mRequestLayoutByResize = false;  
 triggerGlobalLayoutListener = didLayout || attachInfo.mRecomputeGlobalAttributes;  
   
 if(didLayout) {

//开始布局  
 this.performLayout(lp, desiredWindowWidth, desiredWindowHeight);  
 ……  
 }

……  
 if(!cancelDraw && !newSurface) {  
 if(!skipDraw || this.mReportNextDraw) {  
 ……

//开始draw  
 this.performDraw();  
 }  
 ……  
 this.mIsInTraversal = false;  
 }  
}

### 3.2 measure

#### 3.2.1 ViewRootImpl. performMeasure

private void performMeasure(int childWidthMeasureSpec, int childHeightMeasureSpec) {

Trace.traceBegin(8L, **"measure"**);  
 try {  
 this.mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);  
 } finally {  
 Trace.traceEnd(8L);  
 }

}

#### 3.2.2View. measure

1.View中有两个关于测量的方法measure()和onMeasure()。

2.measure()方法是final的，不能被重写。

3.measure()方法中调用了onMeasure()方法，onMeasure()方法在View中的默认实现是测量自身的大小，所以在我们自定义ViewGroup的时候，如果我们不重写onMeasure()测量子控件的大小，子控件将会显示不出来的。

4.setMeasureDimension,做了两个事情，一是复位标志位mPrivateFlags，二是给mMeasureWeith和mMeasureHeight赋值。measure 在调用onMeasure之后，会很快检查标志位mPrivateFlags，如果发现没有setMeasureDimension，就会抛异常。

5. measure最本质的作用，就是给mMeasureWeith和mMeasureHeight赋值。

6.ViewGroup中没有重写onMeasure()方法，在ViewGroup的子类（LinearLayout、RelativeLayout等）中会根据容器自身的布局特性，比如LinearLayout有两种布局方式，水平或者垂直，分别对onMeasure()方法进行不同的重写。

7.由于View.measure()中调用了onMeasure()方法，所以就调用到了DecorView的onMeasure ()方法，DecorView.onMeasure()中首先对宽高约束做了一些处理后，接着调用了super.onMeasure()，也就是FrameLayout中的onMeasure()方法：

public final void measure(int widthMeasureSpec, int heightMeasureSpec) {  
 //如果view有一个光学的边界比如阴影，发光等等，则optical的值为true

//设置完这个属性之后，在布局的时候，View需要放在一个较大区域的布局内

//以便留出来阴影之类位置

//TODO demo

boolean optical = isLayoutModeOptical(this);  
 int cacheIndex;  
 if(optical != isLayoutModeOptical(this.mParent)) {  
 Insets key = this.getOpticalInsets();  
 int oWidth = key.left + key.right;  
 cacheIndex = key.top + key.bottom;  
 widthMeasureSpec = View.MeasureSpec.adjust(widthMeasureSpec, optical?-oWidth:oWidth);  
 heightMeasureSpec =

View.MeasureSpec.adjust(heightMeasureSpec, optical?-cacheIndex:cacheIndex);  
 }  
  
 long key1 = (long)widthMeasureSpec << 32 | (long)heightMeasureSpec & 4294967295L;  
 if(this.mMeasureCache == null) {  
 this.mMeasureCache = new LongSparseLongArray(2);  
 }  
  
 if((this.mPrivateFlags & PFLAG\_FORCE\_LAYOUT) == PFLAG\_FORCE\_LAYOUT

|| widthMeasureSpec != this.mOldWidthMeasureSpec

|| heightMeasureSpec != this.mOldHeightMeasureSpec) {

//这个标志位，是用来判断，onMeasure中是否执行了setMeasuredDimension的

//复位一下这个标志位  
 this.mPrivateFlags &= ~PFLAG\_MEASURED\_DIMENSION\_SET;  
 this.resolveRtlPropertiesIfNeeded();

//如果需要强制布局的话，cache就是-1，下面就会跳转进onMeasure的case

//如果这次测量的值，在以前的测量中，都没有出现过，cacheIndex也是-1

//如果这次测量的值，在以前的测量中，出现过，那我们先不测量，但是会设置标志位  
 cacheIndex = (this.mPrivateFlags & PFLAG\_FORCE\_LAYOUT) ==

PFLAG\_FORCE\_LAYOUT?-1:this.mMeasureCache.indexOfKey(key1);  
 long value;  
 if(cacheIndex >= 0 && !sIgnoreMeasureCache) {  
 value = this.mMeasureCache.valueAt(cacheIndex);  
 this.setMeasuredDimension((int)(value >> 32), (int)value);

//PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT

//虽然现在不测量，但是后面layout的时候，可能会再测量一下  
 this.mPrivateFlags3 |= PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT;  
 } else {

//如果需要强制布局操作，或者忽略测量历史，将会调用onMeasure对控件进行一次测量

//sIgnoreMeasureCache是一个boolean值，

//初始化为sIgnoreMeasureCache = targetSdkVersion < KITKAT;

//意思是如果Android版本低于19，每次都会调用onMeasure()，

//而19以上时sIgnoreMeasureCache==false,这时候就只考虑cacheIndex了

//调用onMeasure，完成实际的测量  
 this.onMeasure(widthMeasureSpec, heightMeasureSpec);  
 this.mPrivateFlags3 &= ~PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT;  
 }  
 //没有调用setMeasuredDimension就抛异常  
 if((this.mPrivateFlags&PFLAG\_MEASURED\_DIMENSION\_SET) !=PFLAG\_MEASURED\_DIMENSION\_SET)

{  
 throw new IllegalStateException(**"onMeasure() did not set**

**the measured dimension by calling setMeasuredDimension()"**);  
 }  
 this.mPrivateFlags |= PFLAG\_LAYOUT\_REQUIRED;  
 }  
 //保存测量值到mOldWidthMeasureSpec，mOldHeightMeasureSpec

//用来作为下次是否需要重新measure的依据  
 this.mOldWidthMeasureSpec = widthMeasureSpec;  
 this.mOldHeightMeasureSpec = heightMeasureSpec;

// mMeasuredWidth和mMeasuredHeight是在setMeasuredDimension被赋值的  
 this.mMeasureCache.put(key1, (long)this.mMeasuredWidth << 32 | (long)this.mMeasuredHeight & 4294967295L);  
}

### 3.3 layout

#### 3.3.1 ViewRootImpl.performLayout

ViewRootImpl.performLayout()方法中会调用mView（Activity根窗口mDecor）的layout()方法，为窗口中所有的子控件安排显示的位置，由于不同的容器有不同的布局策略，所以在布局之前首先要确定所有子控件的大小，才能适当的为子控件安排位置，这就是为什么测量过程需要在布局过程之前完成。接着我们看看DecorView的layout()方法

private void performLayout(LayoutParams lp, int desiredWindowWidth, int desiredWindowHeight) {  
 this.mLayoutRequested = false;  
 this.mScrollMayChange = true;  
 this.mInLayout = true;  
 View host = this.mView;  
   
 try {

//调用view的layout方法，

//这里用的就是setMeasuredDimension中赋值的mMeasureWidth，mMeasuredHeight  
 host.layout(0, 0, host.getMeasuredWidth(), host.getMeasuredHeight());  
 this.mInLayout = false;  
 int numViewsRequestingLayout = this.mLayoutRequesters.size();  
 if(numViewsRequestingLayout > 0) {  
 final ArrayList validLayoutRequesters =

this.getValidLayoutRequesters(this.mLayoutRequesters, false);  
 if(validLayoutRequesters != null) {  
 this.mHandlingLayoutInLayoutRequest = true;  
 int numValidRequests = validLayoutRequesters.size();  
 for(int finalRequesters = 0; finalRequesters < numValidRequests;

++finalRequesters) {  
 View view = (View)validLayoutRequesters.get(finalRequesters);  
 view.requestLayout();  
 }

// TODO  
 // measureHierarchy很重要  
 this.measureHierarchy(host, lp, this.mView.getContext().getResources(),

desiredWindowWidth, desiredWindowHeight);  
 this.mInLayout = true;  
 host.layout(0, 0, host.getMeasuredWidth(), host.getMeasuredHeight());  
 this.mHandlingLayoutInLayoutRequest = false;  
 validLayoutRequesters = this.getValidLayoutRequesters(this.mLayoutRequesters,

true);  
 if(validLayoutRequesters != null) {  
 getRunQueue().post(new Runnable() {  
 public void run() {  
 int numValidRequests = validLayoutRequesters.size();  
 for(int i = 0; i < numValidRequests; ++i) {  
 View view = (View)validLayoutRequesters.get(i);  
 view.requestLayout();  
 }  
 }  
 });  
 }  
 }  
 }  
 }   
 this.mInLayout = false;  
}

#### 3.3.2 View.layout

从下面的代码中，我们注意到关于布局有两个重要的方法，View.layout()和View.onLayout()，这两个方法有什么关系？各自的作用是什么呢？他们都是定义在View中的，不同的是layout()方法中有很长一段实现的代码，而onLayout()确实一个空的实现，里面什么事也没做。   
  首先我们要明确布局的本质是什么，布局就是为View设置四个坐标值，这四个坐标值保存在View的成员变量mLeft、mTop、mRight、mBottom中，方便View在绘制（onDraw）的时候知道应该在那个区域内绘制控件。而我们看到layout()方法中实际上就是为这几个成员变量赋值的，所以到底真正设置坐标的是layout()方法，那onLayout()的作用是什么呢？   
  onLayout()都是由ViewGroup的子类实现的，他的作用就是确定容器中每个子控件的位置，由于不同的容器有不容的布局策略，所以每个容器对onLayout()方法的实现都不同，onLayout()方法会遍历容器中所有的子控件，然后计算他们左上右下的坐标值，最后调用child.layout()方法为子控件设置坐标；由于layout()方法中又调用了onLayout()方法，如果子控件child也是一个容器，就会继续为它的子控件计算坐标，如果child不是容器，onLayout()方法将什么也不做，这样下来，只要Activity根窗口mDecor的layout()方法执行完毕，窗口中所有的子容器、子控件都将完成布局操作。

  其实布局过程的调用方式和测量过程是一样的，ViewGroup的子类都要重写onMeasure()方法遍历子控件调用他们的measure()方法，measure()方法又会调用onMeasure()方法，如果子控件是普通控件就完成了测量，如果是容器将会继续遍历其孙子控件。

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

//layout里面，如果需要重新measure，会重新调用onMeasure方法

//这个就是看measure的时候，cache里面，是否有过之前  
 if((this.mPrivateFlags3 & PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT) != 0) {  
 this.onMeasure(this.mOldWidthMeasureSpec, this.mOldHeightMeasureSpec);  
 this.mPrivateFlags3 &= ~ PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT;  
 }  
 //存储旧的坐标  
 int oldL = this.mLeft;  
 int oldT = this.mTop;  
 int oldB = this.mBottom;  
 int oldR = this.mRight;

// setFrame大概是这个意思：

// 如果老边界和新边界不同，就返回true，设置新边界

// if(this.mLeft != l||this.mRight != r||this.mTop != t||this.mBottom != b)

// change = true;

// this.mLeft = l;

// this.mTop = t;

// this.mRight = r;

// this.mBottom = b;

// setOpticalFrame最后也会调用到setFrame，这个区别不大。  
 boolean changed = isLayoutModeOptical(this.mParent)?this.setOpticalFrame(l, t, r, b):this.setFrame(l, t, r, b);

//布局改变或者强制要求布局  
 if(changed || (this.mPrivateFlags & PFLAG\_LAYOUT\_REQUIRED) == PFLAG\_LAYOUT\_REQUIRED) {  
 //调用onlayout

this.onLayout(changed, l, t, r, b);  
 this.mPrivateFlags &= ~PFLAG\_LAYOUT\_REQUIRED;  
 View.ListenerInfo li = this.mListenerInfo;  
 if(li != null && li.mOnLayoutChangeListeners != null) {  
 ArrayList listenersCopy = (ArrayList)li.mOnLayoutChangeListeners.clone();  
 int numListeners = listenersCopy.size();  
 for(int i = 0; i < numListeners; ++i) {  
 ((View.OnLayoutChangeListener)listenersCopy.get(i)).onLayoutChange(this, l, t, r, b, oldL, oldT, oldR, oldB);  
 }  
 }  
 }  
  
 this.mPrivateFlags &= ~PFLAG\_FORCE\_LAYOUT;  
 this.mPrivateFlags3 |= PFLAG3\_IS\_LAID\_OUT;  
   
}

### FrameLayout. onLayout

所有的布局容器的onLayout方法都是一样的流程，都是先遍历子控件，然后计算子控件的坐标，最后调用子控件的layout()方法设置布局坐标，但是不同的布局容器有不同的布局策略，所以区别就在于计算子控件坐标时的差异。比如LinearLayout线性布局，如果是水平布局，第一个子控件的l值是0，r是100，那第二个子控件的l就是101（只是打个比方），而FrameLayout，如果没有设置padding，子控件也没设置margin，第一个子控件的l值就是0，第二个子控件的l还是0，这就是不同容器的计算区别。

protected void onLayout(boolean changed, int left, int top, int right, int bottom) {  
 this.layoutChildren(left, top, right, bottom, false);  
}  
  
void layoutChildren(int left, int top, int right, int bottom, boolean forceLeftGravity) {  
 int count = this.getChildCount();  
 int parentLeft = this.getPaddingLeftWithForeground();  
 int parentRight = right - left - this.getPaddingRightWithForeground();  
 int parentTop = this.getPaddingTopWithForeground();  
 int parentBottom = bottom - top - this.getPaddingBottomWithForeground();  
 this.mForegroundBoundsChanged = true;  
  
 for(int i = 0; i < count; ++i) {  
 View child = this.getChildAt(i);  
 if(child.getVisibility() != 8) {  
 FrameLayout.LayoutParams lp = (FrameLayout.LayoutParams)child.getLayoutParams();  
 int width = child.getMeasuredWidth();  
 int height = child.getMeasuredHeight();  
 int gravity = lp.gravity;  
 if(gravity == -1) {  
 gravity = 8388659;  
 }  
  
 int layoutDirection = this.getLayoutDirection();  
 int absoluteGravity = Gravity.getAbsoluteGravity(gravity, layoutDirection);  
 int verticalGravity = gravity & 112;  
 int childLeft;  
 switch(absoluteGravity & 7) {  
 case 1:  
 childLeft = parentLeft + (parentRight - parentLeft - width) / 2 + lp.leftMargin - lp.rightMargin;  
 break;  
 case 5:  
 if(!forceLeftGravity) {  
 childLeft = parentRight - width - lp.rightMargin;  
 break;  
 }  
 case 2:  
 case 3:  
 case 4:  
 default:  
 childLeft = parentLeft + lp.leftMargin;  
 }  
  
 int childTop;  
 switch(verticalGravity) {  
 case 16:  
 childTop = parentTop + (parentBottom - parentTop - height) / 2 + lp.topMargin - lp.bottomMargin;  
 break;  
 case 48:  
 childTop = parentTop + lp.topMargin;  
 break;  
 case 80:  
 childTop = parentBottom - height - lp.bottomMargin;  
 break;  
 default:  
 childTop = parentTop + lp.topMargin;  
 }  
  
 child.layout(childLeft, childTop, childLeft + width, childTop + height);  
 }  
 }  
  
}

### 3.4 draw

#### 3.4.1 ViewRootImpl. performDraw

ViewRootImpl的performDraw()方法调用draw(boolean)，在这个过程中主要完成一些条件判断，surface的设置准备，以及判断使用硬件渲染还是软件渲染等操作，由于我们主要研究绘制代码流程层面，所以直接看drawSoftware()方法，对于硬件渲染具体是怎样的有兴趣可以跟踪一下。drawSoftware()方法中，通过mSurface.locakCanvas(dirty)拿到画布，然后调用mView.draw(canvas)，这里的mView就是Activity的根窗口DecorView类型的对象。

private void performDraw() {  
 if(this.mAttachInfo.mScreenOn || this.mReportNextDraw) {  
 ……  
 try {

//draw方法  
 this.draw(fullRedrawNeeded);  
 } finally {  
 this.mIsDrawing = false;  
 }  
 ……  
}

#### 3.4.2 ViewRootImpl .draw

private void draw(boolean fullRedrawNeeded) {  
 Surface surface = this.mSurface;  
 if(surface.isValid()) {  
 AttachInfo mAttachInfo = this.mAttachInfo;  
 boolean var15 = this.mScroller != null && this.mScroller.computeScrollOffset();  
 ……

//获取mDirty这个是需要重新绘制的区域  
 Rect dirty = this.mDirty;  
 if(this.mSurfaceHolder != null) {  
 ……  
 } else {  
 ……  
 this.invalidateDisplayLists();  
 mAttachInfo.mTreeObserver.dispatchOnDraw();  
 if(!dirty.isEmpty() || this.mIsAnimating) {  
 if(mAttachInfo.mHardwareRenderer != null

&& mAttachInfo.mHardwareRenderer.isEnabled()) {  
 this.mIsAnimating = false;  
 this.mHardwareYOffset = yoff;  
 this.mResizeAlpha = resizeAlpha;  
 this.mCurrentDirty.set(dirty);  
 dirty.setEmpty();

//硬件绘制  
 mAttachInfo.mHardwareRenderer.draw(this.mView, mAttachInfo, this,

var15?null:this.mCurrentDirty);  
 } else {  
 if(mAttachInfo.mHardwareRenderer != null

&& !mAttachInfo.mHardwareRenderer.isEnabled()

&& mAttachInfo.mHardwareRenderer.isRequested()) {  
 try {  
 mAttachInfo.mHardwareRenderer.initializeIfNeeded(this.mWidth,

this.mHeight, this.mHolder.getSurface());  
 } catch (OutOfResourcesException var12) {  
 this.handleOutOfResourcesException(var12);  
 return;  
 }  
  
 this.mFullRedrawNeeded = true;  
 this.scheduleTraversals();  
 return;  
 }  
 //软件绘制  
 if(!this.drawSoftware(surface, mAttachInfo, yoff, scalingRequired, dirty))

{  
 return;  
 }  
 }  
 }  
 if(var15) {  
 this.mFullRedrawNeeded = true;  
 this.scheduleTraversals();  
 }  
 }  
 }  
}

#### 3.4.4 ViewRootImpl. drawSoftware

private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int yoff, boolean scalingRequired, Rect dirty) {  
 Canvas canvas;  
 try {  
 int left = dirty.left;  
 int top = dirty.top;  
 int right = dirty.right;  
 int bottom = dirty.bottom;  
 canvas = this.mSurface.lockCanvas(dirty);  
 if(left != dirty.left || top != dirty.top || right != dirty.right

|| bottom != dirty.bottom) {  
 attachInfo.mIgnoreDirtyState = true;  
 }  
  
 canvas.setDensity(this.mDensity);  
 } catch (OutOfResourcesException var28) {  
 this.handleOutOfResourcesException(var28);  
 return false;  
 } catch (IllegalArgumentException var29) {  
 Log.e(**"ViewRootImpl"**, **"Could not lock surface"**, var29);  
 this.mLayoutRequested = true;  
 return false;  
 }  
  
 try {  
   
 if(!canvas.isOpaque() || yoff != 0) {  
 canvas.drawColor(0, Mode.CLEAR);  
 }  
  
 dirty.setEmpty();  
 this.mIsAnimating = false;  
 attachInfo.mDrawingTime = SystemClock.uptimeMillis();  
 this.mView.mPrivateFlags |= 32;  
   
 try {  
 canvas.translate(0.0F, (float)(-yoff));  
 if(this.mTranslator != null) {  
 this.mTranslator.translateCanvas(canvas);  
 }  
  
 canvas.setScreenDensity(scalingRequired?this.mNoncompatDensity:0);  
 attachInfo.mSetIgnoreDirtyState = false;

//调用view.draw  
 this.mView.draw(canvas);  
 this.drawAccessibilityFocusedDrawableIfNeeded(canvas);  
 } finally {  
 if(!attachInfo.mSetIgnoreDirtyState) {  
 attachInfo.mIgnoreDirtyState = false;  
 }  
  
 }  
  
 } finally {  
 try {  
 surface.unlockCanvasAndPost(canvas);  
 } catch (IllegalArgumentException var25) {  
 this.mLayoutRequested = true;  
 return false;  
 }  
 }  
 return true;  
}

#### 3.4.5 View.draw

View的draw()方法中有六个步骤，其中最为重要的就是第1步绘制背景、第2步绘制内容、第3步绘制子控件；绘制背景我们就不看了，我们看看绘制内容，调用的是onDraw()方法，View中的onDraw()方法是一个空的实现，可以想象，因为View是所有控件的超类，它并不知道他的孩子要怎样画自己，所以绘制的具体操作由孩子重写onDraw()方法自己绘制，所以DecorView重写了onDraw()方法来绘制Activity的跟窗口，具体怎么画的请跟踪DecorView的onDraw()方法:

public void draw(Canvas canvas) {

//1如果有设置背景，则绘制背景

//2保存canvas层

//3绘制自身内容

//4如果有子元素则绘制子元素

//5绘制效果

//6绘制装饰品(scrollbars)

final int privateFlags = mPrivateFlags;  
final boolean dirtyOpaque = (privateFlags & PFLAG\_DIRTY\_MASK) == PFLAG\_DIRTY\_OPAQUE &&  
 (mAttachInfo == null || !mAttachInfo.mIgnoreDirtyState);  
 mPrivateFlags = (privateFlags & ~PFLAG\_DIRTY\_MASK) | PFLAG\_DRAWN;  
  
// Step 1, draw the background, if needed  
int saveCount;  
if (!dirtyOpaque) {  
 drawBackground(canvas);  
}  
...省略  
// Step 2, save the canvas' layers  
int paddingLeft = mPaddingLeft;  
final boolean offsetRequired = isPaddingOffsetRequired();  
if (offsetRequired) {  
 paddingLeft += getLeftPaddingOffset();  
}  
 ...省略  
// Step 3, draw the content  
if (!dirtyOpaque) onDraw(canvas);  
// Step 4, draw the children  
dispatchDraw(canvas);

...省略  
// Step 5, draw the fade effect and restore layers  
final Paint p = scrollabilityCache.paint;  
final Matrix matrix = scrollabilityCache.matrix;  
final Shader fade = scrollabilityCache.shader;  
 ...省略  
 // Step 6, draw decorations (scrollbars)  
onDrawScrollBars(canvas);  
if (mOverlay != null && !mOverlay.isEmpty()) {  
 mOverlay.getOverlayView().dispatchDraw(canvas);  
}

}

#### 3.4.6 View.dispatchDraw

dispatchDraw()方法中首先获取子控件的数量childrenCount，然后遍历所有子控件，调用drawChild()方法，drawChild()方法中只是简单的调用child.draw()，这样就完成了子控件的绘制。细心的可以发现child的draw()方法又会执行之前DecorView.draw()的六步（draw()是在View里面实现的），所以说，所有的控件在绘制的时候都会调用draw()方法，draw()方法中会先调用onDraw()方法绘制自己，然后调用dispatchDraw()绘制它的子控件，如果此控件不是ViewGroup的子类，也就是说是叶子控件，dispatchDraw()`什么也不做。

protected void dispatchDraw(Canvas canvas) {  
 ...  
 final int childrenCount = mChildrenCount;  
 final View[] children = mChildren;  
  
 if ((flags & FLAG\_RUN\_ANIMATION) != 0 && canAnimate()) {  
 ...  
 for (int i = 0; i < childrenCount; i++) {  
 while (transientIndex >= 0 && mTransientIndices.get(transientIndex) == i) {  
 final View transientChild = mTransientViews.get(transientIndex);  
 if ((transientChild.mViewFlags & VISIBILITY\_MASK) == VISIBLE ||  
 transientChild.getAnimation() != null) {  
 more |= drawChild(canvas, transientChild, drawingTime);  
 }  
 transientIndex++;  
 if (transientIndex >= transientCount) {  
 transientIndex = -1;  
 }  
 }  
 int childIndex = customOrder ? getChildDrawingOrder(childrenCount, i) : i;  
 final View child = (preorderedList == null)?

children[childIndex]：preorderedList.get(childIndex);  
 if ((child.mViewFlags & VISIBILITY\_MASK) == VISIBLE || child.getAnimation() != null) {  
 more |= drawChild(canvas, child, drawingTime);  
 }  
 }  
 }  
 ...  
}  
  
protected boolean drawChild(Canvas canvas, View child, long drawingTime) {  
 return child.draw(canvas, this, drawingTime);  
}