## View的加载过程

ActivityThread. performLaunchActivity

activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);

Application app = r.packageInfo.makeApplication(false, mInstrumentation);

Context appContext = createBaseContextForActivity(r, activity);

activity.**attach**(appContext, this, getInstrumentation(), r.token,

r.ident, app, r.intent, r.activityInfo, title, r.parent,

r.embeddedID, r.lastNonConfigurationInstances, config,

r.voiceInteractor);

attachBaseContext(context);

mFragments.attachActivity(this, mContainer, null);

mWindow = PolicyManager.**makeNewWindow**(this);

mWindow.setCallback(this);

activity.setTheme(theme);

mInstrumentation.**callActivityOnCreate**(activity, r.state);

Activity. onCreate

setContentView(R.layout.incall\_screen);

getWindow().setContentView(layoutResID);

installDecor();

mDecor = generateDecor();

new DecorView(getContext(), -1);

mContentParent = generateLayout(mDecor);

View in = mLayoutInflater.inflate(layoutResource, null);

decor.addView(in, new ViewGroup.LayoutParams(MATCH\_PARENT, MATCH\_PARENT));

ViewGroup contentParent = (ViewGroup)findViewById(ID\_ANDROID\_CONTENT);

mLayoutInflater.inflate(layoutResID, mContentParent);

ActivityThread. handleResumeActivity

performResumeActivity

ActivityClientRecord r = mActivities.get(token);

r.activity.performResume();

mInstrumentation.callActivityOnResume(this);

wm.**addView**(decor, l);

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

root = new ViewRootImpl(view.getContext(), display); // 保存在WindowManagerGlobal内

mWindowSession = WindowManagerGlobal.getWindowSession(); //WMS的本地功能代理

mWidth = -1;

mHeight = -1;

mDirty = new Rect();

mWinFrame = new Rect();

mWindow = new W(this); //用于接收WMS发过来的控制信息，并送到ViewRootHandler队列中处理

mAttachInfo = new View.AttachInfo(mWindowSession, mWindow, display, this, mHandler, this);

view.setLayoutParams(wparams);

mViews.add(view);

mRoots.add(root);

mParams.add(wparams);

root.**setView**(view, wparams, panelParentView);

ViewRootImpl. setView

mView = view;

mWindowAttributes.copyFrom(attrs);

**requestLayout**();

scheduleTraversals(); //启动view树的遍历

//将mWindow传给WMS，添加之后WMS对mWindowAttributes等属性参数进行赋值

res = **mWindowSession**.**addToDisplay**(**mWindow**, mSeq, **mWindowAttributes**,

getHostVisibility(), mDisplay.getDisplayId(),

mAttachInfo.mContentInsets, mInputChannel);

// WindowManagerService

mService.**addWindow**(this, window, seq, attrs, viewVisibility, displayId,

outContentInsets, outInputChannel);

token = new WindowToken(this, attrs.token, -1, false);

1、View添加到Windows流程，创建surface

2、windows参数设置，view参数设置

3、window token， app token， window id

4、WindowState

prev.app.thread是一个ApplicationThread对象的远程接口，通过调用这个远程接口的schedulePauseActivity来通知Launcher进入Paused状态。

ActivityThread paused之后会通知ActivityManagerService，这个Activity已经进入Paused状态了，ActivityManagerService现在可以完成未竟的事情，即启动MainActivity了。

ActivityManagerNative.getDefault().activityPaused(token, state);

## View的invalidate刷新过程

1. 如何只刷新有变化的区域

2、刷新的遍历流程

**View的绘制，在addWindow的时候，就拿到了一个surface，以后更新画布那些都在这上面操作？**

## ImageView的大小计算

<RelativeLayout

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

<ImageView

android:id=*"@+id/view"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"wrap\_content"*

android:scaleType=*"centerCrop""* />

图片大小2400 1500

屏幕大小 1200 1920

实际大小 w1200 h1500

<ImageView

android:id=*"@+id/view"*

android:layout\_width=*"1500px"*

android:layout\_height=*"wrap\_content"*

android:background=*"#555555"*

android:scaleType=*"centerInside"*

实际大小 1200 1500

<ImageView

android:id=*"@+id/view"*

android:layout\_width=*"1500px"*

android:layout\_height=*"2000px"*

android:background=*"#555555"*

android:scaleType=*"centerInside"*

实际大小 1200 1664

这里非1500 2000是因为外面包裹的是一个RelativeLayout的同时，里面还有其他的View

改为FrameLayout就好了。

从源码分析，只要子View指定了大小，子View的测量规格都是

resultSize = childDimension;

resultMode = MeasureSpec.EXACTLY;

所以，顶多是显示不全而已。

如果把高变成wrap\_content。则View的大小为1500 1500。View的测量规格为：

resultSize = size; //整屏

resultMode = MeasureSpec.AT\_MOST;

这个时候，就需要看ImageView的onMeasure方法了。

**首先是decorView的测量规格：**

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;

**然后是Content**

<com.android.internal.widget.**ActionBarOverlayLayout**

android:layout\_width="**match\_parent**"

android:layout\_height="**match\_parent**"

android:theme="?attr/actionBarTheme">

<**FrameLayout** android:id="@android:id/content"

android:layout\_width="**match\_parent**"

android:layout\_height="**match\_parent**" />

如果指定的都是**match\_parent，那么在measureChildWithMargins计算子类测量规格的时候：**

父View的测量规格是整屏大小和**EXACTLY，子View的是：**

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

switch (specMode) {

// Parent has imposed an exact size on us

case MeasureSpec.EXACTLY:

if (childDimension >= 0) { 手动指定的情况

resultSize = childDimension;

resultMode = MeasureSpec.EXACTLY;

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

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

resultSize = **size**; //传递父view sise

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;

**总结：**

1. 不管父View给子View的测量规则是什么，只要子View有直接指定宽高，那么子View的测量规格就会变为，自己指定的宽高和Exactly。对于再下一级的子View来说，父View下发的最大size就可能缩小了。
2. 如果父View给子类的测量规格是size1 Exactly，那么子View如果是:

\* 有指定大小或着和父View一样，那么子View的大小也是确定的，即测量规则是指定的大小或父View的大小，Exactly。

\* WrapContent，可变的。那么子View的测量规则就是，父View的大小，AtMost。给你一个最大范围，范围内你自己想要多大就多大。

1. 如果父View给子类的测量规格是size1 AtMost，那么子View

\* 有指定大小，那么子View的大小也是确定的，即测量规则是指定的大小，Exactly。

\* 父View一样或者WrapContent，那么子View的大小也不确定的，即测量规则就是，父View的大小，AtMost。给你一个最大范围，范围内你自己想要多大就多大。

4、如果父View给子类的测量规格是size1 Unspecified，那么子View只有自己指定了大小的时候，测量规格才为Exactly。其他情况都为Unspecified。

拓展：

1. 如果父view V1为AtMost，子View V2也是AtMost，那么是不是可能出现子View大小比父View要大。因为他们都使用了来自上级View的最大size。
2. 当然，子View在OnMeasure的时候，不按照规格来，那大小就任意了。
3. Tips：onMeasure中计算所有childView的宽和高，然后根据childView的宽和高，计算自己的宽和高。（当然，如果不是wrap\_content，直接使用父ViewGroup传入的计算值即可）

getWidth() 是实际显示的宽度。

getMeasureWidth() 是测量宽度，在布局之前计算出来的。

getIntrinsicWidth() 是原有宽度，有时候原有宽度可能很大，但是实际上空间不够，所有效果上并没有那么大，这个方法可以获得原有宽度，可以辅助测量的时候选择合适的展示宽度。

getMinimumWidth() 是最小宽度，是XML参数定义里的 minWidth，也是一个辅助测量展示的参数。

**#View的Measure**

缓存机制。如果不是强制layout或者宽高的测量规格变化，就不需要重新测量。

if ((mPrivateFlags & PFLAG\_FORCE\_LAYOUT) == PFLAG\_FORCE\_LAYOUT ||

widthMeasureSpec != mOldWidthMeasureSpec ||

heightMeasureSpec != mOldHeightMeasureSpec) {

int cacheIndex = (mPrivateFlags & PFLAG\_FORCE\_LAYOUT) == PFLAG\_FORCE\_LAYOUT ? -1 :

mMeasureCache.indexOfKey(key);

//什么情况下会出现不是强制layout，但宽高的测量规格变化呢？但这个时候还使用cache会不会有问题。

if (**cacheIndex** < 0 || **sIgnoreMeasureCache**) {

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

**onMeasure**(widthMeasureSpec, heightMeasureSpec);

mPrivateFlags3 &= ~PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT;

} else {

long value = **mMeasureCache**.valueAt(cacheIndex);

// Casting a long to int drops the high 32 bits, no mask needed

**setMeasuredDimensionRaw**((int) (value >> 32), (int) value);

mPrivateFlags3 |= PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT;

}

**PFLAG\_FORCE\_LAYOUT的设置时机**

forceLayout()，requestLayout()时会设置mPrivateFlags |= PFLAG\_FORCE\_LAYOUT;

Layout最后会清空mPrivateFlags &= ~PFLAG\_FORCE\_LAYOUT;

**#View的Layout**

对比上面View的layout和ViewGroup的layout方法可以发现，View的layout方法是可以在子类重写的，而ViewGroup的layout是不能在子类重写的，言外之意就是说ViewGroup中只能通过重写onLayout方法。

注意View的坐标都是以父View为(0, 0)。

ViewGroup的onLayout()方法竟然是一个抽象方法，这就是说所有ViewGroup的子类都必须重写这个方法。所以在自定义ViewGroup控件中，onLayout配合onMeasure方法一起使用可以实现自定义View的复杂布局。自定义View首先调用onMeasure进行测量，然后调用onLayout方法动态获取子View和子View的测量大小，然后进行layout布局。重载onLayout的目的就是安排其children在父View的具体位置，重载onLayout通常做法就是写一个for循环调用每一个子视图的layout(l, t, r, b)函数，传入不同的参数l, t, r, b来确定每个子视图在父视图中的显示位置。

**#提高View的绘制性能**

**http://blog.csdn.net/lmj623565791/article/details/45556391**

1. Measure、Layout、Draw，每个环节遍历的效率，以及代码的效率。
2. Overdraw的检测。移除不必要的background。clipRect的妙用。减少不必要的层次：巧用Hierarchy Viewer。

**#从Scroller谈View的坐标系**

View内部有变量mScrollX和mScrollY表明View内部元素偏移自己的位置。**X大于0表示上移。**这两个值在绘制的时候，用来平移画布以及裁剪画布。

View在Draw方法绘制之前，会通过computeScroll()计算当前的偏移。一般而言，ViewPager这种子类会重写，以便在平滑滑动的时候，动态计算偏移。

computeScroll就是判断mScroller是不是处于结束状态，或者此次已经滑动结束了。

if (!mScroller.isFinished() && mScroller.computeScrollOffset()) {

scrollTo(x, y);//调用scrollTo改变当前的mScrollX和mScrollY

mScrollX = x;

mScrollY = y;

invalidateParentCaches(); //清除父View缓存

postInvalidateOnAnimation(); //继续下次动画，也就是下一次Draw。直到动画结束。

**#RelativeLayout 和LinearLayout 的性能分析**

http://www.jianshu.com/p/8a7d059da746

1. RelativeLayout 布局灵活一些。

2. 实现复杂的布局时，由于 RelativeLayout 所需要的嵌套层次少，所以可能大概也许性能会好一些。 但是你应该根据实际情况测试，主观判断不准确，请参考下面第三点。

3. 对于简单的布局，或者使用 LinearLayout 的嵌套布局层次不多的情况下。还是推荐 使用 LinearLayout 来实现。 因为 LinearLayout 布局性能更高一点，为什么？ 因为 RelativeLayout 布局需要至少计算两次。

**#动画之Rotate**

将一个button旋转之后，child的位置似乎并没有改变？但点击事件和背景又是按旋转的来

D/rotate ( 3214): position is left=0, right=540, top=528, bottom=600

D/rotate ( 3214): position is left=0, right=540, top=528, bottom=600

所以在判断子View在不在点击范围的时候，会将触点位置转换到View的坐标系。

protected boolean isTransformedTouchPointInView(float x, float y, View child,

PointF outLocalPoint) {

final float[] point = getTempPoint();

point[0] = x;

point[1] = y;

transformPointToViewLocal(point, child);

final boolean isInView = child.pointInView(point[0], point[1]);

既然如此，那PageTransformer在计算偏移的时候，就可以愉快的使用如下公式了

final float transformPos = (float) (**child.getLeft()** - scrollX) / getPaddedWidth();

mPageTransformer.transformPage(child, transformPos);

Gradle

##The SDK platform-tools version (24.0.1) is too old to check APIs compiled with API 25; please update

升级sdk搞定

## shrinkResources true

> File 'D:\git\_workspace\cgeo-appcompat\main\build\intermediates\res\resources-basic-debug-stripped.ap\_' specified for property 'resourceFile' does not exist.

改为false