

Window和DecorView的创建

Activity.setContentView()说起，查看这个方法的源码：

private Window mWindow;

public void setContentView(@LayoutRes int layoutResID) {

//调用window对象的setContentView()

getWindow().setContentView(layoutResID);

//创建ActionBar

initWindowDecorActionBar();

}

public Window getWindow() {

return mWindow;

}

Activity的setContentView()实际上调用的是Window的setContentView()，这时候Activity将View相关的渲染交给Window。

mWindow什么时候被创建的呢？

Activity.java

//只截取部分主要代码

final void attach(...) {

attachBaseContext(context);

mFragments.attachHost(null /\*parent\*/);

//创建PhoneWindow对象

mWindow = new PhoneWindow(this, window, activityConfigCallback);

mWindow.setWindowControllerCallback(this);

mWindow.setCallback(this);

mWindow.setOnWindowDismissedCallback(this);

mWindow.getLayoutInflater().setPrivateFactory(this);

if(info.softInputMode!=WindowManager.LayoutParams.SOFT\_INPUT\_STATE\_UNSPECIFIED) {

mWindow.setSoftInputMode(info.softInputMode);

}

if (info.uiOptions != 0) {

mWindow.setUiOptions(info.uiOptions);

}

...

//通过Context.getSystemService(Context.WINDOW\_SERVICE)的方式获取WindowManager的实例

//给window设置windowManger

mWindow.setWindowManager(

(WindowManager)context.getSystemService(Context.WINDOW\_SERVICE),

mToken, mComponent.flattenToString(),

(info.flags & ActivityInfo.FLAG\_HARDWARE\_ACCELERATED) != 0);

if (mParent != null) {

mWindow.setContainer(mParent.getWindow());

}

mWindowManager = mWindow.getWindowManager();

mCurrentConfig = config;

mWindow.setColorMode(info.colorMode);

}

mWindow是在Activity的attach()中被创建的，它实际上是一个PhoneWindow对象，PhoneWindow是Android提供的Window唯一实现类。且系统会为每个Activity创建一个与之对应的Window

创建Window后，还会通过Context.getSystemService(Context.WINDOW\_SERVICE)的方式获取WindowManager的实例，并设置给Window。WindowManager是一个窗口管理类。

下面来看PhoneWindow的setContentView()。

@Override

public void setContentView(int layoutResID) {

//先判断mContentParent是否初始化

if (mContentParent == null) {

//如果没有初始化调用installDecor

//当前activity第一次调用sentContentView

installDecor();

} else if (!hasFeature(FEATURE\_CONTENT\_TRANSITIONS)) {

//如果Activity没有过度动画，多次调用sentContentView会走这里

//移除mContentParent所有内部view

mContentParent.removeAllViews();

}

//判断是否使用了Activity的过度动画

if (hasFeature(FEATURE\_CONTENT\_TRANSITIONS)) {

//设置动画场景

final Scene newScene = Scene.getSceneForLayout(mContentParent, layoutResID,getContext());

transitionTo(newScene);

} else {

//将资源文件通过LayoutInflater对象装换为View树

mLayoutInflater.inflate(layoutResID, mContentParent);

}

mContentParent.requestApplyInsets();

final Callback cb = getCallback();

if (cb != null && !isDestroyed()) {

//这个方法在Activity是一个空实现

//说明在Activity的布局改动时 (setContentView或者addContentView 方法执行完毕后会调用改方法) 所以各种View的findViewById方法什么的可以放在这里

cb.onContentChanged();

}

mContentParentExplicitlySet = true;

}

这里我们重点关注installDecor()

private void installDecor() {

if (mDecor == null) {

//如果mDecor为空则创建一个DecorView实例

mDecor = generateDecor();

mDecor.setDescendantFocusability(ViewGroup.FOCUS\_AFTER\_DESCENDANTS);

mDecor.setIsRootNamespace(true);

if (!mInvalidatePanelMenuPosted && mInvalidatePanelMenuFeatures != 0) {

mDecor.postOnAnimation(mInvalidatePanelMenuRunnable);

}

}

if (mContentParent == null) {

//如果mContentParent为空则通过generateLayout创建一个

mContentParent = generateLayout(mDecor);

...

}

}

installDecor() 主要干两件事情

如果mDecor没有初始化，则通过generateDecor()初始化

如果mContentParent 没有初始化，则通过generateLayout()初始化

generateDecor()很简单，就是new一个DecorView实例

protected DecorView generateDecor(int featureId) {

Context context;

if (mUseDecorContext) {

Context applicationContext = getContext().getApplicationContext();

if (applicationContext == null) {

context = getContext();

} else {

context=newDecorContext(applicationContext,getContext().getResources());

if (mTheme != -1) {

context.setTheme(mTheme);

}

}

} else {

context = getContext();

}

//创建new DecorView实例

return new DecorView(context, featureId, this, getAttributes());

}

下面来分析，generateLayout()

//只截取部分主要代码

protected ViewGroup generateLayout(DecorView decor) {

//根据当前style修饰相应样式

TypedArray a = getWindowStyle();

...

//一堆if判断，根据设置的主题样式来设置DecorView的风格

if (a.getBoolean(R.styleable.Window\_windowNoTitle, false)) {

requestFeature(FEATURE\_NO\_TITLE);

} else if (a.getBoolean(R.styleable.Window\_windowActionBar, false)) {

// Don't allow an action bar if there is no title.

requestFeature(FEATURE\_ACTION\_BAR);

} else if(...){

...

}

//加载窗口布局

int layoutResource;

int features = getLocalFeatures();

//根据features选择不同的layoutResource（布局文件资源）

if ((features & (1 << FEATURE\_SWIPE\_TO\_DISMISS)) != 0) {

layoutResource = R.layout.screen\_swipe\_dismiss;

} else if(...){

...

}

//加载layoutResource

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

//DecorView是FrameLayout,往DecorView中添加根布局View

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

mContentRoot = (ViewGroup) in;

//这里获取的就是mContentParent

//ID\_ANDROID\_CONTENT = com.android.internal.R.id.content;

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

if (contentParent == null) {

throw new RuntimeException("Window couldn't find content container view");

}

...

return contentParent;

}

根据设置的Window主题样式来设置DecorView的风格，接着为DecorView添加子View，而这里的子View则是上面提到的mContentParent，如果上面设置了FEATURE\_NO\_ACTIONBAR，那么DecorView就只有mContentParent一个子View。可以看我们文章一开始的那张图，图中就很好的阐述了DecorView、TitleView、mContentParent三者之间的关系

所以这也就是为什么，在设置Activity属性的时候，比如requestWindowFeature(Window.FEATURE\_NO\_TITLE) 需要在setContentView()之前调用才会生效

小结

一个Activity对应一个Window，Window是在activity.attach()的时候被创建的

在Activity中调用setContentView()实际上是调用了Window的setContentView()

调用setContentView()时，会去初始化DecorView以及其内部的TitleView和mContentParent，可以通过设置Window.FEATURE\_NO\_TITLE使得DecorView内部只存在mContentParent

我们在xml中定义的layout\_activity.xml实际上是mContentParent的一个子View

通过WindowManager管理Window

上文中我们主要讲的是Window的创建、DecorView的创建以及DecorView的子View是如何被添加到DecorView中去的。每个Activity都有一个与之关联的Window，那如何管理Window呢？这时候就需要借助我们的WindowMananger类。WindowManager管理Window实际上就是在管理Window中的DecorView

下面来看一下DecorView和WindowManager是如何关联起来的，从Activity的启动开始分析。

使用代理模式启动到ActivityManagerService中执行

创建ActivityRecord到mHistory记录中

通过socket通信到Zgote相关类创建process

通过ApplicatonThread与ActivityManagerService建立通信

ActivityManagerService通知ActiveThread启动Activity的创建

ActivityThread创建Activity加入到mActivities中并开始调度Activity执行

activityThread.handleLaunchActivity()

activityThread.handleLaunchActivity() —> activityThread.performLaunchActivity()—> activity.attach() —> activity.onCreate()—> activityThread.handleResumeActivity() —>activityThread.performResumeActivity() —> activity.onResume() —> windowManager.addView()

//只截取部分主要代码

final void handleResumeActivity(IBinder token, boolean clearHide, boolean isForward) {

...

// 这里会调用到activity.onResume()方法

ActivityClientRecord r = performResumeActivity(token, clearHide);

if (r != null) {

final Activity a = r.activity;

...

if (r.window == null && !a.mFinished && willBeVisible) {

// 获得Window对象

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

// 获得Window中的DecorView对象

View decor = r.window.getDecorView();

decor.setVisibility(View.INVISIBLE);

// 获得WindowManager对象

//这个WindowManager就是在activity.attach()的时候和window一起创建的

ViewManager wm = a.getWindowManager();

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

a.mDecor = decor;

l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION;

l.softInputMode |= forwardBit;

if (a.mVisibleFromClient) {

a.mWindowAdded = true;

// 调用windowManager.addView方法

wm.addView(decor, l);

}

...

if (!r.activity.mFinished && willBeVisible

&& r.activity.mDecor != null && !r.hideForNow) {

r.activity.mVisibleFromServer = true;

mNumVisibleActivities++;

if (r.activity.mVisibleFromClient) {

//设置decorView可见

r.activity.makeVisible();

}

}

}

}

}

通过调用windowManager.addView(decor, lp)，这样一来，DecorView和WindowManager就建立了联系。WindowManager继承了ViewManger接口，但实际上其本身依然是一个接口，实现类是WindowManagerImpl

//WindowManagerImpl implements WindowManager

//WindowManager extends 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);

}

//只截取部分主要代码

public final class WindowManagerImpl implements WindowManager {

private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();

private final Context mContext;

private final Window mParentWindow;

public WindowManagerImpl(Context context) {

this(context, null);

}

private WindowManagerImpl(Context context, Window parentWindow) {

mContext = context;

mParentWindow = parentWindow;

}

public WindowManagerImpl createLocalWindowManager(Window parentWindow) {

return new WindowManagerImpl(mContext, parentWindow);

}

public WindowManagerImpl createPresentationWindowManager(Context displayContext) {

return new WindowManagerImpl(displayContext, mParentWindow);

}

//往Window中添加view

@Override

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

applyDefaultToken(params);

mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);

}

//更新布局

@Override

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

applyDefaultToken(params);

mGlobal.updateViewLayout(view, params);

}

//将删除View的消息发送到MessageQueue中，稍后删除

@Override

public void removeView(View view) {

mGlobal.removeView(view, false);

}

//立刻删除Window中的view

@Override

public void removeViewImmediate(View view) {

mGlobal.removeView(view, true);

}

}

注意：同样实现了ViewManger接口的还有ViewGroup，我们知道ViewGroup也有addView方法，但是在ViewGroup中是将普通的view或者viewGroup作为Children加入，而在WindowManagerImpl是将DecorView作为根布局加入到PhoneWindow中去，所以两个方法的作用是截然不同的

我们发现，WindowManagerImpl并没有直接实现操作View的相关方法，而是全部交给WindowManagerGlobal。WindowManagerGlobal是一个单例类—即一个进程中最多仅有一个。创建WindowManagerGlobal对象的方式如下

public static WindowManagerGlobal getInstance() {

synchronized (WindowManagerGlobal.class) {

if (sDefaultWindowManager == null) {

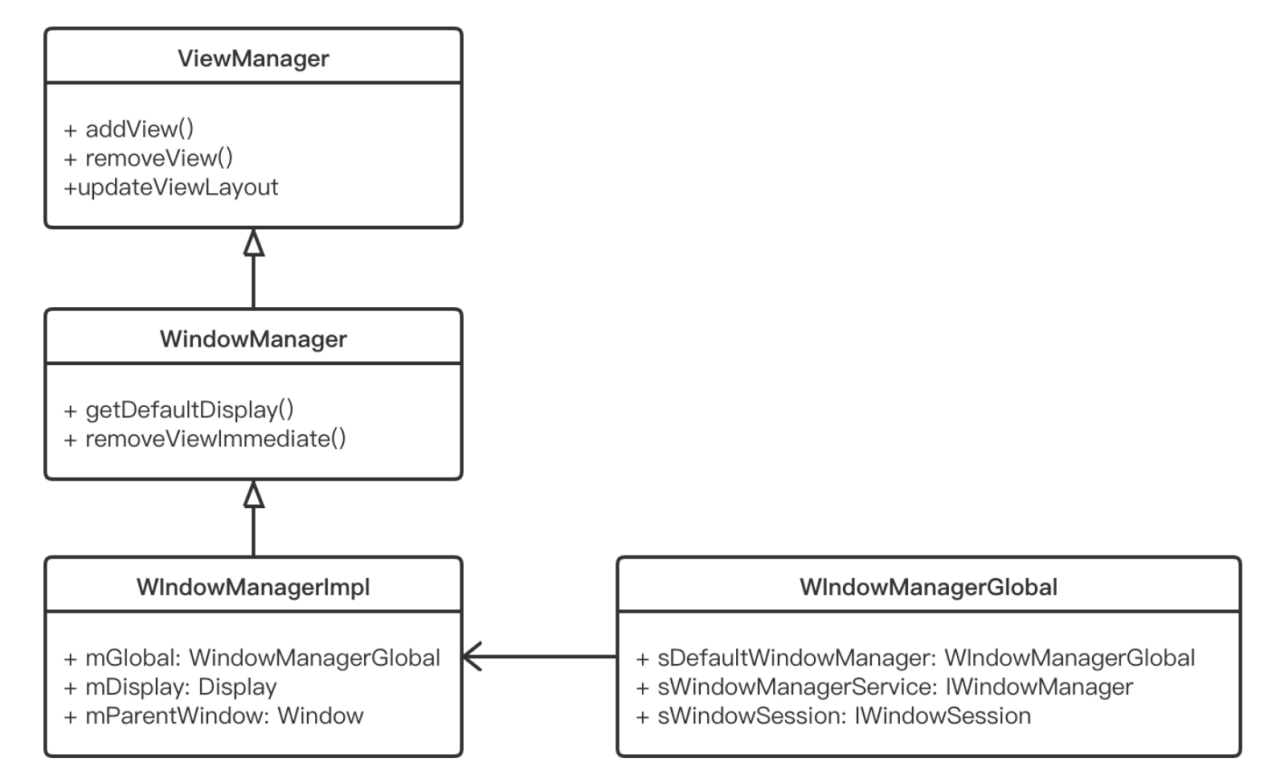
sDefaultWindowManager = new WindowManagerGlobal();

}

return sDefaultWindowManager;

}

}



深入分析WindowManagerGlobal

分析WindowManangerGlobal实际上就是分析其内部的三个方法

addView(View view, ViewGroup.LayoutParams params)

该方法的主要作用是将decorView添加到viewRootImp中，通过viewRootImp对其内部的view进行管理

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

Display display, Window parentWindow) {

//参数检查

if (view == null) {

throw new IllegalArgumentException("view must not be null");

}

...

//判断是否有父Window，从而调整当前窗口布局参数（layoutParams）

final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams) params;

if (parentWindow != null) {

//有，调整title和token

parentWindow.adjustLayoutParamsForSubWindow(wparams);

} else {

//无，应用开启了硬件加速的话，decorview就开启

final Context context = view.getContext();

if (context != null && (context.getApplicationInfo().flags

& ApplicationInfo.FLAG\_HARDWARE\_ACCELERATED) != 0) {

wparams.flags|= WindowManager.LayoutParams.FLAG\_HARDWARE\_ACCELERATED;

}

}

ViewRootImpl root;

View panelParentView = null;

synchronized (mLock) {

if (mSystemPropertyUpdater == null) {

//创建一个Runnable，用来遍历更新所有ViewRootImpl，更新系统参数

mSystemPropertyUpdater = new Runnable() {

@Override public void run() {

synchronized (mLock) {

for (int i = mRoots.size() - 1; i >= 0; --i) {

mRoots.get(i).loadSystemProperties();

}

}

}

};

//添加到执行队列中

SystemProperties.addChangeCallback(mSystemPropertyUpdater);

}

//看需要add的view是否已经在mViews中

//WindowManager不允许同一个View被添加两次

int index = findViewLocked(view, false);

if (index >= 0) {

// 如果被添加过，就看是否在死亡队列里也存在

if (mDyingViews.contains(view)) {

// 如果存在，就将这个已经存在的view对应的window移除

mRoots.get(index).doDie();

} else {

//否则，说明view已经被添加，不需要重新添加了

throw new IllegalStateException("View " + view

+ " has already been added to the window manager.");

}

}

// 如果属于子Window层级

if(wparams.type >= WindowManager.LayoutParams.FIRST\_SUB\_WINDOW && wparams.type <= WindowManager.LayoutParams.LAST\_SUB\_WINDOW) {

final int count = mViews.size();

//遍历viewRootImpl集合，看是否存在一个window的IBinder对象和需要添加的window的token一致，之后赋值引用

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

if (mRoots.get(i).mWindow.asBinder() == wparams.token) {

panelParentView = mViews.get(i);

}

}

}

//创建一个ViewRootImpl对象并保存在root变量中

root = new ViewRootImpl(view.getContext(), display);

//给需要添加的view设置params，也就是decorView

view.setLayoutParams(wparams);

//将decoView、布局参数以及新建的ViewRootImpl保存在三个集合中

mViews.add(view);

mRoots.add(root);

mParams.add(wparams);

try {

//将decorView设置给ViewRootImpl

//ViewRootImpl向WMS添加新的窗口、申请Surface以及decorView在Surface上的重绘动作这才是真正意义上完成了窗口的添加操作

root.setView(view, wparams, panelParentView);

} catch (RuntimeException e) {

// BadTokenException or InvalidDisplayException, clean up.

if (index >= 0) {

removeViewLocked(index, true);

}

throw e;

}

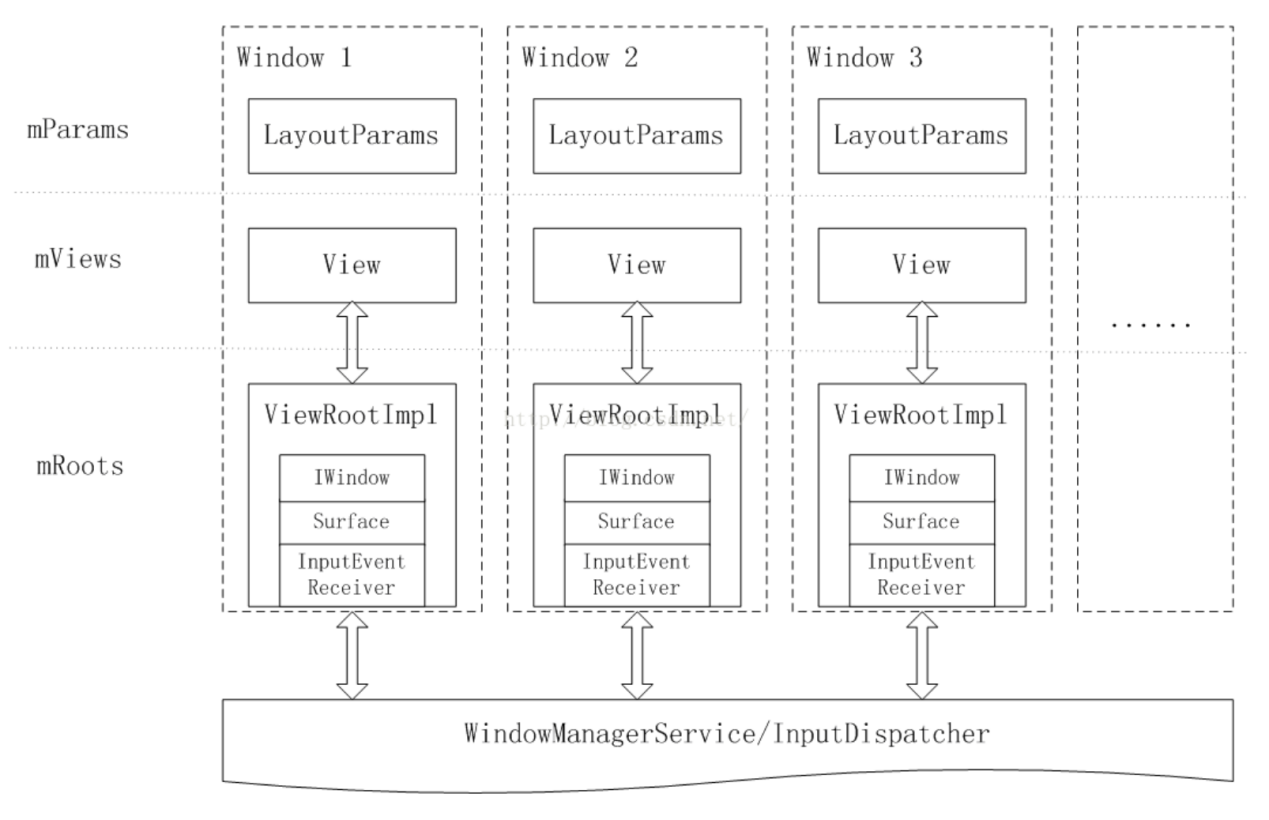
}

}

父窗口修改新窗口的布局参数。可能修改的只有LayoutParams.token和LayoutParams.mTitle两个属性。mTitle属性不必赘述，仅用于调试。而token属性则值得一提，每一个新窗口必须通过LayoutParams.token向WMS出示相应的令牌才可以。在addView()函数中通过父窗口修改这个token属性的目的是为了减少开发者的负担。开发者不需要关心token到底应该被设置为什么值，只需将LayoutParams丢给一个WindowManager，剩下的事情就不用再关心了。父窗口修改token属性的原则是：如果新窗口的类型为子窗口(其类型大于等于LayoutParams.FIRST\_SUB\_WINDOW并小于等于LayoutParams.LAST\_SUB\_WINDOW)，则LayoutParams.token所持有的令牌为其父窗口的ID（也就是IWindow.asBinder()的返回值）。否则LayoutParams.token将被修改为父窗口所属的Activity的ID(也就是AppToken)，这对类型为TYPE\_APPLICATION的新窗口来说非常重要。从这点来说，当且仅当新窗的类型为子窗口时addView()的parentWindow参数才是真正意义上的父窗口。这类子窗口有上下文菜单、弹出式菜单以及游标等等，在WMS中，这些窗口对应的WindowState所保存的mAttachedWindow既是parentWindow所对应的WindowState。然而另外还有一些窗口，如对话框窗口，类型为TYPE\_APPLICATION， 并不属于子窗口，但需要AppToken作为其令牌，为此parentWindow将自己的AppToken赋予了新窗口的的LayoutParams.token中。此时parentWindow便并不是严格意义上的父窗口了

为新窗口创建一个ViewRootImpl对象。顾名思义，ViewRootImpl实现了一个控件树的根。它负责与WMS进行直接的通讯，负责管理Surface，负责触发控件的测量与布局，负责触发控件的绘制，同时也是输入事件的中转站。总之，ViewRootImpl是整个控件系统正常运转的动力所在。

将控件、布局参数以及新建的ViewRootImpl对象以相同的索引值添加到三个对应的集合mViews、mParams以及mRoots中，以供之后的查询之需。控件、布局参数以及ViewRootImpl三者共同组成了客户端的一个窗口。或者说，在控件系统中的窗口就是控件、布局参数与ViewRootImpl对象的一个三元组



updateViewLayout(View view, ViewGroup.LayoutParams params)

该方法的主要作用是更新decorView的layoutParams，如layoutParams.width从100变为了200，则需要将这个变化通知给WMS使其调整Surface的大小，并让窗口进行重绘

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

//参数检查

if (view == null) {

throw new IllegalArgumentException("view must not be null");

}

if (!(params instanceof WindowManager.LayoutParams)) {

throw new IllegalArgumentException("Params must be WindowManager.LayoutParams");

}

final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;

//将layoutparams保存到decorView中

view.setLayoutParams(wparams);

synchronized (mLock) {

// 获取decorView在三个集合中的索引

int index = findViewLocked(view, true);

ViewRootImpl root = mRoots.get(index);

mParams.remove(index);

//更新layoutParams到集合中

mParams.add(index, wparams);

//调用viewRootImpl的setLayoutParams()使得新的layoutParams生效

root.setLayoutParams(wparams, false);

}

}

removeView(View view)

该方法的作用是从3个集合中删除此Window所对应的元素，包括decorView、layoutPrams以及viewRootImpl，并要求viewRootImpl从WMS中删除对应的Window，并释放一切需要回收的资源

public void removeView(View view, boolean immediate) {

if (view == null) {

throw new IllegalArgumentException("view must not be null");

}

synchronized (mLock) {

int index = findViewLocked(view, true);

View curView = mRoots.get(index).getView();

removeViewLocked(index, immediate);//其内部会调用root.die(immediate)

if (curView == view) {

return;

}

throw new IllegalStateException("Calling with view " + view

+ " but the ViewAncestor is attached to " + curView);

}

}

要求viewRootImpl从WMS中删除窗口并释放资源的方法是调用viewRootImpl.die()函数。因此可以得出这样一个结论：viewRootImpl的生命从setView()开始，到die()结束

通过ViewRootImp管理View

ViewRootImpl实现了ViewParent接口，它是WindowManagerGlobal工作的实际实现者

我们先来看setView()做了什么

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

synchronized (this) {

if (mView == null) {

//mView保存了decorView

mView = view;

//mWindowAttributes保存了窗口所对应的LayoutParams

mWindowAttributes.copyFrom(attrs);

attrs = mWindowAttributes;

...

//请求UI开始绘制

requestLayout();

//初始化mInputChannel

//InputChannel是窗口接受来自InputDispatcher的输入事件的管道

//注意，仅当窗口的属性inputFeatures不含有INPUT\_FEATURE\_NO\_INPUT\_CHANNEL时

//才会创建InputChannel，否则mInputChannel为空，从而导致此窗口无法接受任何输入事件

mInputChannel = new InputChannel();

try {

//通知WindowManagerService添加一个窗口，注册一个事件监听管道

//用来监听 按键(KeyEvent)和触摸(MotionEvent)事件

res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes, getHostVisibility(), mDisplay.getDisplayId(), mAttachInfo.mContentInsets, mAttachInfo.mStableInsets, mInputChannel);

}

...

}

}

setView()内部的执行过程viewRootImp.setView() —> viewRootImp.requestLayout() —> viewRootImp.scheduleTraversals() —> viewRootImp.doTraversal() —> viewRootImp.performTraversals()—>进入view的绘制流程

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

//线程检查，这里就是判断更新UI的操作是否在主线程的方法

checkThread();

mLayoutRequested = true;

scheduleTraversals();

}

}

void scheduleTraversals() {

if (!mTraversalScheduled) {

mTraversalScheduled = true;

mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();

mChoreographer.postCallback(

Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, null);

if (!mUnbufferedInputDispatch) {

scheduleConsumeBatchedInput();

}

notifyRendererOfFramePending();

pokeDrawLockIfNeeded();

}

}

final class TraversalRunnable implements Runnable {

@Override

public void run() {

doTraversal();

}

}

void doTraversal() {

if (mTraversalScheduled) {

mTraversalScheduled = false;

mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

if (mProfile) {

Debug.startMethodTracing("ViewAncestor");

}

//开始view的绘制

performTraversals();

if (mProfile) {

Debug.stopMethodTracing();

mProfile = false;

}

}

}

我们打开一个Activity，经过Window和DecorView的创建过程后，绘制会从 ViewRootImp 的 performTraversals() 方法开始，从上到下遍历整个视图树进行View的绘制。

ViewRootImp.performTraversals()

performTraversals()的源码非常的长，但是核心代码就是下面三个步骤。

private void performTraversals() {

......

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

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

......

//这里的mView就是DecorView

mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);

......

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

......

mView.draw(canvas);

......

}

所以，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，来获取或者计算出每个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

每个 View 负责绘制自己，而 ViewGroup 还需要负责通知自己的子 View 进行绘制操作

Measure — 测量

MeasureSpec

从上面的源码我们可以发现，系统会用一个int型的值（childWidthMeasureSpec和childHeightMeasureSpec）来存储View的宽高的信息

上文中用于生成childWidthMeasureSpec和childHeightMeasureSpec的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;

}

这个方法实际上调用的就是MeasureSpec.makeMeasureSpec()

public static class MeasureSpec {

private static final int MODE\_SHIFT = 30;

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

@IntDef({UNSPECIFIED, EXACTLY, AT\_MOST})

@Retention(RetentionPolicy.SOURCE)

public @interface MeasureSpecMode {}

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

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

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

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);

}

}

public static int makeSafeMeasureSpec(int size, int mode) {

if (sUseZeroUnspecifiedMeasureSpec && mode == UNSPECIFIED) {

return 0;

}

return makeMeasureSpec(size, mode);

}

@MeasureSpecMode

public static int getMode(int measureSpec) {

//noinspection ResourceType

return (measureSpec & MODE\_MASK);

}

public static int getSize(int measureSpec) {

return (measureSpec & ~MODE\_MASK);

}

static int adjust(int measureSpec, int delta) {

final int mode = getMode(measureSpec);

int size = getSize(measureSpec);

if (mode == UNSPECIFIED) {

// No need to adjust size for UNSPECIFIED mode.

return makeMeasureSpec(size, UNSPECIFIED);

}

size += delta;

if (size < 0) {

Log.e(VIEW\_LOG\_TAG, "MeasureSpec.adjust: new size would be negative! (" + size +

") spec: " + toString(measureSpec) + " delta: " + delta);

size = 0;

}

return makeMeasureSpec(size, mode);

}

}

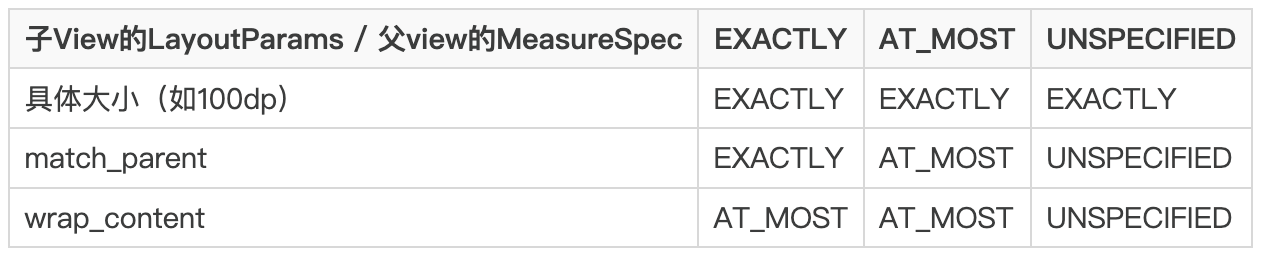
那么这个MeasureSpec到底是什么意思呢？MeasureSpec概括了从父布局传递给子view的布局要求，包括了测量模式和测量大小。分析代码可知，int长度为32位，高2位表示mode（模式），后30位用于表示size（大小）

有三种mode：

UNSPECIFIED：不对View大小做限制，如：ListView，ScrollView

EXACTLY：确切的大小，如：100dp或者march\_parent

AT\_MOST：大小不可超过某数值，如：wrap\_content



当View采用固定宽高时（即设置固定的dp/px），不管父容器是什么模式，View都是EXACTLY模式，并且大小遵循我们设置的值

当View的宽高是match\_parent时，如果父容器的是EXACTLY模式，那么View也是EXACTLY模式且其大小是父容器的剩余空间；如果父容器是AT\_MOST模式那么View也是AT\_MOST模式并且其大小不会超过父容器的剩余空间

当View的宽高是wrap\_content时，View都是AT\_MOST模式并且其大小不能超过父容器的剩余空间

只要提供父容器的MeasureSpec和子元素的LayoutParams，就可以确定出子元素的MeasureSpec，进一步便可以确定出测量后的大小

onMeasure

mView.measure()内部会调用onMeasure()

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

//仅列出关键代码

...

onMeasure(widthMeasureSpec, heightMeasureSpec);

...

}

一个View的实际测量工作是在onMeasure()中实现的，onMeasure()已经默认为我们的控件测量了宽高

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),

getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));

}

/\*\*

\* 为宽度获取一个建议最小值

\*/

protected int getSuggestedMinimumWidth () {

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

}

/\*\*

\* 获取默认的宽高值

\*/

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;

}

在自定义ViewGroup时，默认的onMeasure()往往不能满足我们的需求，这时候就要重写该方法，在该方法内测量子View的尺寸。当重写onMeasure()时，必须调用setMeasuredDimension(width,height)来存储该View测量出的宽和高。如果不这样做将会触发IllegalStateException

ViewGroup提供了三个方法测量子View的宽高

/\*\*

\*遍历ViewGroup中所有的子控件，调用measuireChild测量宽高

\*/

protected void measureChildren (int widthMeasureSpec, int heightMeasureSpec) {

...

}

/\*\*

\* 测量某一个child的宽高

\*/

protected void measureChild (View child, int parentWidthMeasureSpec,

...

}

/\*\*

\* 测量某一个child的宽高，考虑margin值

\*/

protected void measureChildWithMargins (View child,

...

}

View和ViewGroup重写onMeasure的差异

下面用两个例子分别来展示一下View和ViewGroup重写onMeasure的差异

View

View一般只关心自身尺寸的测量

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

super.onMeasure(widthMeasureSpec, heightMeasureSpec);

int width = measureWidth(widthMeasureSpec);

int height = measureHeight(heightMeasureSpec);

setMeasuredDimension(width, height);

}

private int measureWidth(int measureSpec) {

int result = 0;

int specMode = MeasureSpec.getMode(measureSpec);

int specSize = MeasureSpec.getSize(measureSpec);

Log.e("CustomTextViewWidth", "---speSize = " + specSize + "");

switch (specMode) {

case MeasureSpec.AT\_MOST:

result = (int) mPaint.measureText(mTextStr) + getPaddingLeft() + getPaddingRight();

Log.e("CustomTextViewWidth", "---speMode = AT\_MOST");

break;

case MeasureSpec.EXACTLY:

Log.e("CustomTextViewWidth", "---speMode = EXACTLY");

result = specSize;

break;

case MeasureSpec.UNSPECIFIED:

Log.e("CustomTextViewWidth", "---speMode = UNSPECIFIED");

result = Math.max(result, specSize);

}

Log.e("CustomTextViewWidth", "---result = "+result);

return result;

}

private int measureHeight(int measureSpec) {

int result = 0;

int specMode = MeasureSpec.getMode(measureSpec);

int specSize = MeasureSpec.getSize(measureSpec);

Log.e("CustomTextViewHeight", "---speSize = " + specSize + "");

switch (specMode) {

case MeasureSpec.AT\_MOST:

result =

(int) (-mPaint.ascent() + mPaint.descent()) + getPaddingTop() + getPaddingBottom();

Log.e("CustomTextViewHeight", "---speMode = AT\_MOST");

break;

case MeasureSpec.EXACTLY:

result = specSize;

Log.e("CustomTextViewHeight", "---speSize = EXACTLY");

break;

case MeasureSpec.UNSPECIFIED:

result = Math.max(result, specSize);

Log.e("CustomTextViewHeight", "---speSize = UNSPECIFIED");

break;

}

Log.e("CustomTextViewHeight", "---result = "+result);

return result;

}

ViewGroup

ViewGroup一般会先遍历子View，调用子View的测量方法，然后在再结合子View的尺寸来确定自身的大小

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

super.onMeasure(widthMeasureSpec, heightMeasureSpec);

int modeWidth = MeasureSpec.getMode(widthMeasureSpec);

int sizeWidth = MeasureSpec.getSize(widthMeasureSpec);

int modeHeight = MeasureSpec.getMode(heightMeasureSpec);

int sizeHeight = MeasureSpec.getSize(heightMeasureSpec);

// 如果是wrap\_content,定义width，height统计FlowLayout的宽高

int width = 0;

int height = 0;

// 记录每一行的宽度与高度

int lineWidth = 0;

int lineHeight = 0;

/\*\*

\* 1、通过getChildCount，获取子View的个数view个数

\*/

int childCount = getChildCount();

/\*\*

\* 2、遍历childCount，通过getChildAt获取到对应的view

\*/

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

//获取i对应的子View，通过获取他的宽高，确定

View childView = getChildAt(i);

/\*\*

\* 3、对childView进行测量

\*/

measureChild(childView, widthMeasureSpec, heightMeasureSpec);

MarginLayoutParams lp = (MarginLayoutParams) childView.getLayoutParams();

int childWidth = childView.getMeasuredWidth() + lp.leftMargin + lp.rightMargin;

int childHeight = childView.getMeasuredHeight() + lp.topMargin + lp.bottomMargin;

// 判断是否换行，如果换行则高度累加，如果不换行则宽度累加

if (lineWidth + childWidth > sizeWidth - getPaddingLeft() - getPaddingRight()) {

// 对比得到最大的宽度

width = Math.max(width, lineWidth);

// 重置lineWidth

lineWidth = childWidth;

// 记录行高

height += lineHeight;

// 重置lineHeight

lineHeight = childHeight;

} else {

// 宽度累加

lineWidth += childWidth;

// 得到当前行最大的高度

lineHeight = Math.max(lineHeight, childHeight);

}

// 最后一个的时候，不管是换行，还是未换行，前面都没有处理

if (i == childCount - 1) {

width = Math.max(width, lineWidth);

height += lineHeight;

}

/\*\*

\* 4、确定父布局（FlowLayout）的宽高

\*/

setMeasuredDimension(modeWidth == MeasureSpec.EXACTLY ? sizeWidth :

width + getPaddingLeft() + getPaddingRight(),

modeHeight == MeasureSpec.EXACTLY ? sizeHeight :

height + getPaddingTop() + getPaddingBottom());

}

}

Layout - 布局

前面measure的作用是测量每个View的尺寸，而layout的作用是根据前面测量的尺寸以及设置的其它属性值，共同来确定View的位置

ViewGroup的layout()

@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);

}

//内部实际上是调用View的layout()

super.layout(l, t, r, b);

} else {

// record the fact that we noop'd it; request layout when transition finishes

mLayoutCalledWhileSuppressed = true;

}

}

View的layout()

/\*\*

\* 作用：确定View本身的位置，即设置View本身的四个顶点位置

\*/

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

//根据一些flag，如果有需要则进一步measure

if ((mPrivateFlags3 & PFLAG3\_MEASURE\_NEEDED\_BEFORE\_LAYOUT) != 0) {

onMeasure(mOldWidthMeasureSpec, mOldHeightMeasureSpec);

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

}

//暂存旧的位置信息

int oldL = mLeft;

int oldT = mTop;

int oldB = mBottom;

int oldR = mRight;

// 1. 确定View的位置：setFrame（） / setOpticalFrame（）

// 即初始化四个顶点的值、判断当前View大小和位置是否发生了变化

boolean changed = isLayoutModeOptical (mParent) ?

setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);

// 2. 若视图的大小 & 位置发生变化

// 会重新确定该View所有的子View在父容器的位置：onLayout（）

if (changed || (mPrivateFlags & PFLAG\_LAYOUT\_REQUIRED) == PFLAG\_LAYOUT\_REQUIRED) {

// 对于单一View的laytou过程：由于单一View是没有子View的，故onLayout（）是一个空实现->>分析3

// 对于ViewGroup的laytou过程：由于确定位置与具体布局有关，所以onLayout（）在ViewGroup为1个抽象方法，需重写实现（后面会详细说）

onLayout(changed, l, t, r, b);

//回调layoutChange事件

......

}

//标记为已经执行过layout;

......

}

/\*\*

\* 作用：确定View本身的位置，即设置View本身的四个顶点位置

\* @return 如果新的尺寸和位置和之前的不同，返回true

\*/

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;

}

从源码可以看出layout()最终通过setFrame()方法对view的四个属性（mLeft、mTop、mRight、mBottom）进行了赋值，从而去确定View的大小和位置，如果发生改变则调用onLayout()方法

onLayout

我们先来看看ViewGroup的onLayout()方法，该方法是一个抽象方法。因为layout过程是父布局容器布局子View的过程，onLayout()方法对子View没有意义，只有ViewGroup才有用，所以ViewGroup应该重写该方法并为每一个子View调用layout()

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

@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);

}

}

我们可以看到，这里面会对每一个child调用layout()方法。如果该child仍然是ViewGroup，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布局流程就走完了。另外，width和height分别来源于measure阶段存储的测量值

Draw - 绘制

当layout完成后，就进入到draw阶段了，在这个阶段，会根据layout中确定的各个view的位置将它们画出来

前面说过，mView就是DecorView，所以我们直接来看DecorView的draw()方法

@Override

public void draw(Canvas canvas) {

super.draw(canvas);

if (mMenuBackground != null) {

mMenuBackground.draw(canvas);

}

}

调用完super.draw()后，还画了菜单背景。我们继续关注super.draw()方法，会发现FrameLayout和ViewGroup都没有重写该方法，直接进到了View的draw()方法

@CallSuper

public void draw(Canvas canvas) {

......

int saveCount;

// Step 1, draw the background, if needed

if (!dirtyOpaque) {

drawBackground(canvas);

}

// Step 2, If necessary, save the canvas' layers to prepare for fading

......

// Step 3, draw the content

if (!dirtyOpaque) onDraw(canvas);

// Step 4, draw the children

dispatchDraw(canvas);

//Step 5, If necessary, draw the fading edges and restore layers

......

// Step 6, draw decorations (foreground, scrollbars)

onDrawForeground(canvas);45 ......

}

主要是就是如下几步，其中最重要的就是画内容和画子View

画背景。对应我我们在xml布局文件中设置的android:background属性

画内容。通过重写onDraw()方法

画子View。dispatchDraw()方法用于帮助ViewGroup来递归画它的子View

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

onDraw()

当自定义View需要进行绘制的时候，我们往往会重写onDraw()方法，这里放一个简单的例子感受一下

Paint mPaint = new Paint(Paint.ANTI\_ALIAS\_FLAG);

@Override

protected void onDraw(Canvas canvas) {

mPaint.setColor(Color.YELLOW);

canvas.drawRect(0, 0, getWidth(), getHeight(), mPaint);

mPaint.setColor(Color.BLUE);

mPaint.setTextSize(20);

String text = "Hello View";

canvas.drawText(text, 0, getHeight() / 2, mPaint);

}