第7章 高级UI面试题汇总

摘要：本章内容，享学课堂在高级UI原理与实战中有系统化-全面完整的直播讲解，详情加微信：xxgfwx03

7.11 事件的分发机制，责任链模式的优缺点 （美团）~leo
这道题想考察什么?
考生应该如何回答
责任链模式的定义
事件分发机制中的责任链模式
责任链模式的优缺点
7.12 动画的分类以及区别（车和家）~colin
这道题想考察什么?
考察的知识点
考生应该如何回答
7.13 属性动画与普通的动画有什么区别? （车和家）~leo
这道题想考察什么?
考生应该如何回答
属性动画和补间动画的基本编写方式
属性动画的使用注意点
属性动画和补间动画工作原理
属性动画
补间动画
为什么属性动画移动一个 View 后，目标位置还可以响应触摸事件呢?
补间动画和属性动画改变的是同一个 Matrix 么?
7.14 插值器 估值器的区别（车和家）~leo
这道题想考察什么?
考生应该如何回答
TimeInterpolator（时间插值器）：
TypeEvaluator（类型估值算法，即估值器）：
自定义插值器
自定义估值器
总结
7.15 RecyclerView与ListView的对比，缓存策略，优缺点。（美团）~colin
这道题想考察什么?
考察的知识点
考生应该如何回答
1.布局效果
2.item点击事件
3.局部刷新
4.动画效果
5.缓存区别
层级不同
缓存内容不同
RV优势
缓存区别
7.16 WebView如何做资源缓存? （字节跳动）
这道题想考察什么?
考察的知识点
考生应该如何回答
1.WebView如何做资源缓存，你知道吗?
2. Application Cache 缓存机制
7.17 WebView和JS交互的几种方式与拦截方法。（字节跳动）~colin
这道题想考察什么?
考察的知识点
考生应该如何回答
步骤2：在Android里通过WebView设置调用JS代码
特别注意：JS代码调用一定要在 `onPageFinished()` 回调之后才能调用，否则不会调用。
3.Android通过WebView调用 JS 代码
方式3：通过 WebChromeClient 的onJsAlert()、onJsConfirm()、onJsPrompt()方法回调拦截JS，对话框alert()、confirm()、prompt()消息
步骤1：加载JS代码，如下：
步骤2：在Android通过`webChromeClient`复写`onJsPrompt()`

7.1 View绘制流程与自定义View注意点。（东方头条、美团）

这道题想考察什么?

1. 是否了解View绘制流程与自定义View注意点操作与真实场景使用，是否熟悉View绘制流程与自定义View注意点

考察的知识点

1. View绘制流程与自定义View注意点的概念在项目中使用与基本知识

考生应该如何回答

1.View绘制流程与自定义View注意点，你知道吗?

> 注意点一：View中关于四个构造函数参数

自定义View中View的构造函数有四个

```
    //  主要是在java代码中生命一个View时所调用，没有任何参数，一个空的View对象
    public ChildrenView(Context context) {
        super(context);
    }

    // 在布局文件中使用该自定义view的时候会调用到，一般会调用到该方法
    public ChildrenView(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    //如果你不需要View随着主题变化而变化，则上面两个构造函数就可以了
    //下面两个是与主题相关的构造函数
   public ChildrenView(Context context, AttributeSet attrs, int defStyleAttr) {
        this(context, attrs, defStyleAttr, 0);
    }

    public ChildrenView(Context context, AttributeSet attrs, int defStyleAttr,
int defStyleRes) {
        super(context, attrs, defStyleAttr, defStyleRes);
    }
```

四个参数解释:

context:上下文

AttributeSet attrs：从xml中定义的参数

intdefStyleAttr：主题中优先级最高的属性

intdefStyleRes： 优先级次之的内置于View的style(这里就是自定义View设置样式的地方)

> 注意点二：自定义控件类型

1. 自定义属性

在`res/values`目录下的`attrs.xml`文件中

```
<resources>
<declare-styleable name="CustomView">
        <attr name="leftIcon" format="reference" />
        <attr name="state" format="boolean"/>
        <attr name="name" format="string"/>
    </declare-styleable>
</resources>
```

1. 布局中使用自定义属性

在布局中使用

```
<com.myapplication.view.CustomView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                app:leftIcon="@mipmap/ic_temp"
                app:name="温度"
                app:state="false" />
```

1. view的构造函数获取自定义属性

```
class DigitalCustomView : LinearLayout {
    constructor(context: Context) : super(context)
    constructor(context: Context, attrs: AttributeSet?) : super(context, attrs)
{
        LayoutInflater.from(context).inflate(R.layout.view_custom, this)
        var ta = context.obtainStyledAttributes(attrs, R.styleable.CustomView)
        mIcon = ta.getResourceId(R.styleable.CustomView_leftIcon, -1) //左图像
        mState = ta.getBoolean(R.styleable.DigitalCustomView_state, false)
        mName = ta.getString(R.styleable.CustomView_name)
        ta.recycle()
        initView()
    }

}
```

上面给出大致的代码 记得获取`context.obtainStyledAttributes(attrs, R.styleable.CustomView)`最后要调用`ta.recycle()`利用对象池回收ta加以复用

7.2 在onResume中可以测量宽高么 ~leo

这道题想考察什么?

这道题想考察同学对 绘制流程 的理解。

考生应该如何回答

首先我们要找到系统调用 onResume 的地方。

```
handleResumeActivity() {
    //...
    r = performResumeActivity(token, clearHide, reason);//调用 onResume() 方法
    //...
    wm.addView(decor, l);//对 mAttachInfo 进行赋值
    //...
}
```

可以看出 onResume() 方法在 addView() 方法前调用。
重点关注：onResume() 方法所处的位置，前后都发生了什么?
从上面总结的流程看出，onResume() 方法是由 handleResumeActivity 触发的，而界面绘制被触发是因为 handleResumeActivity() 中调用了wm.addView(decor, l);

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

    ViewRootImpl root;
    View panelParentView = null;

    synchronized (mLock) {
        //...

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

        view.setLayoutParams(wparams);

        mViews.add(view);
        mRoots.add(root);
        mParams.add(wparams);
    }

    // do this last because it fires off messages to start doing things
    try {
        //触发开发绘制，参考 1
        root.setView(view, wparams, panelParentView);
    } catch (RuntimeException e) {
        //...
        throw e;
    }
}
```

参考 1：setView()

```
public void setView(View view, WindowManager.LayoutParams attrs, View
panelParentView) {
    synchronized (this) {
        if (mView == null) {
            mView = view;

            //...
```

```
            // Schedule the first layout -before- adding to the window
            // manager, to make sure we do the relayout before receiving
            // any other events from the system.
            //触发界面刷新，参考 2
            requestLayout();
            if ((mWindowAttributes.inputFeatures
                    & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL)
== 0) {
                mInputChannel = new InputChannel();
            }
            //...
            try {
                mOrigWindowType = mWindowAttributes.type;
                mAttachInfo.mRecomputeGlobalAttributes = true;
                collectViewAttributes();
                res = mWindowSession.addToDisplay(mWindow, mSeq,
mWindowAttributes,
                        getHostVisibility(), mDisplay.getDisplayId(),
                        mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
                        mAttachInfo.mOutsets, mInputChannel);
            } catch (RemoteException e) {
                //...
                throw new RuntimeException("Adding window failed", e);
            } finally {
                if (restore) {
                    attrs.restore();
                }
            }

            //...

            //这里的 view 是 DecorView
            view.assignParent(this);
            //...
        }
    }
}
```

参考 2：requestLayout()

```
@Override
public void requestLayout() {
    if (!mHandlingLayoutInLayoutRequest) {
        checkThread();
        mLayoutRequested = true;
        //准备刷新，参考 3
        scheduleTraversals();
    }
}
```

参考 3：scheduleTraversals()

```
void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        //设置同步障碍 Message
```

```
        mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
        //屏幕刷新信号 VSYNC 监听回调把 mTraversalRunnable（执行doTraversal()）push 到
主线程了，异步 Message 会优先得到执行 ，具体看下 Choreographer 的实现
        //mTraversalRunnable，参考 4
        mChoreographer.postCallback(Choreographer.CALLBACK_TRAVERSAL,
mTraversalRunnable, null);
        if (!mUnbufferedInputDispatch) {
            scheduleConsumeBatchedInput();
        }
        notifyRendererOfFramePending();
        pokeDrawLockIfNeeded();
    }
}
```

参考 4：mTraversalRunnable

```
final class TraversalRunnable implements Runnable {
    @Override
    public void run() {
        //参考 5
        doTraversal();
    }
}
final TraversalRunnable mTraversalRunnable = new TraversalRunnable();
```

参考 5：doTraversal()

```
void doTraversal() {
    if (mTraversalScheduled) {
        mTraversalScheduled = false;
        //移除同步障碍 Message
        mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);

        if (mProfile) {
            Debug.startMethodTracing("ViewAncestor");
        }

        //参考 6，在上面移除同步障碍后，开始对控件树进行测量、布局、绘制
        performTraversals();

        if (mProfile) {
            Debug.stopMethodTracing();
            mProfile = false;
        }
    }
}
```

参考 6：performTraversals()

```
private void performTraversals() {
    // cache mView since it is used so much below...
    final View host = mView;

    //...

    Rect frame = mWinFrame;
```

```
    if (mFirst) {
        mFullRedrawNeeded = true;
        mLayoutRequested = true;

        //...

        // We used to use the following condition to choose 32 bits drawing
caches:
        // PixelFormat.hasAlpha(lp.format) || lp.format == PixelFormat.RGBX_8888
        // However, windows are now always 32 bits by default, so choose 32 bits
        mAttachInfo.mUse32BitDrawingCache = true;
        mAttachInfo.mHasWindowFocus = false;
        mAttachInfo.mWindowVisibility = viewVisibility;
        mAttachInfo.mRecomputeGlobalAttributes = false;
        mLastConfiguration.setTo(host.getResources().getConfiguration());
        mLastSystemUiVisibility = mAttachInfo.mSystemUiVisibility;
        // Set the layout direction if it has not been set before (inherit is
the default)
        if (mViewLayoutDirectionInitial == View.LAYOUT_DIRECTION_INHERIT) {
            host.setLayoutDirection(mLastConfiguration.getLayoutDirection());
        }
        host.dispatchAttachedToWindow(mAttachInfo, 0);
        mAttachInfo.mTreeObserver.dispatchOnWindowAttachedChange(true);
        dispatchApplyInsets(host);
        //Log.i(mTag, "Screen on initialized: " + attachInfo.mKeepScreenOn);

    } else {
        desiredWindowWidth = frame.width();
        desiredWindowHeight = frame.height();
        if (desiredWindowWidth != mWidth || desiredWindowHeight != mHeight) {
            if (DEBUG_ORIENTATION) Log.v(mTag, "View " + host + " resized to: "
+ frame);
            mFullRedrawNeeded = true;
            mLayoutRequested = true;
            windowSizeMayChange = true;
        }
    }

    //...

    // Execute enqueued actions on every traversal in case a detached view
enqueued an action
    getRunQueue().executeActions(mAttachInfo.mHandler);

    //...

    performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
    performLayout(lp, mWidth, mHeight);
    performDraw();

    //...
}
```

在 onResume() 中 handler.post(Runnable) 获取不到 View 的真实宽高

原因：如果想要获取到宽高，起码要把消息 post 到布局测量消息所在的 Handler，和通过 new 方式创建的 Handler 并没有关系，所以在 onResume() 中 handler.post(Runnable) 中操作 UI 是失效的

View.post(Runnable) 为什么可以获取到 View 的宽高?

```
public boolean post(Runnable action) {
    //mAttachInfo 是在 ViewRootImpl 的构造函数中初始化的
    //而 ViewRootmpl 的初始化是在 addView() 中调用
    //所以此处的 mAttachInfo 为空，所以不会执行该 if 语句
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler.post(action);
    }

    // Postpone the runnable until we know on which thread it needs to run.
    // Assume that the runnable will be successfully placed after attach.
    //保存消息到 RunQueue 中，等到在 performTraversals() 方法中被执行
    getRunQueue().post(action);
    return true;
}
```

由参考 6 可知，我们通过 View.post(Runnable) 的 Message 会在 performMeaure() 之前被调用，那为什么还可以正确的获取到 View 的宽高呢？其实我们的 Message 并没有立即被执行，因为此时主线程的 Handler 正在执行的 Message 是 TraversalRunnable，而 performMeaure() 方法也是在该 Message 中被执行，所以排队等到主线程的 Handler 执行到我们 post 的 Message 时，View 的宽高已经测量完毕，因此我们也就很自然的能够获取到 View 的宽高。

总结

在 onResume() 中直接或者调用创建的 Handler 的 post 方法是获取不到 View 宽高的，需要通过 View.post 才能获取。

7.3 事件分发机制是什么过程？（东方头条）~leo

这道题想考察什么?

这道题想考察同学对事件分发的流程是否掌握了。

考生应该如何回答

此处主要分析正常流程，拦截和冲突的解决等内容在后续文章中讲解。
事件主要包括以下几个：

事件	简介
ACTION_DOWN	手指初次接触到屏幕时触发
ACTION_MOVE	手指在屏幕上滑动时触发，会多次触发
ACTION_UP	手指离开屏幕时触发
ACTION_CANCEL	事件被上层拦截时触发

事件分发总流程为 Activity --> ViewGroup --> ... --> ViewGroup --> View，每一个事件都是按这个流程从上往下分发的。
为了便于分析，我们只分析 ViewGroup(父容器) --> View(子View) 的过程。
注意：这里说的父容器，子View，他们的父子关系指的是包裹关系，不是继承关系。

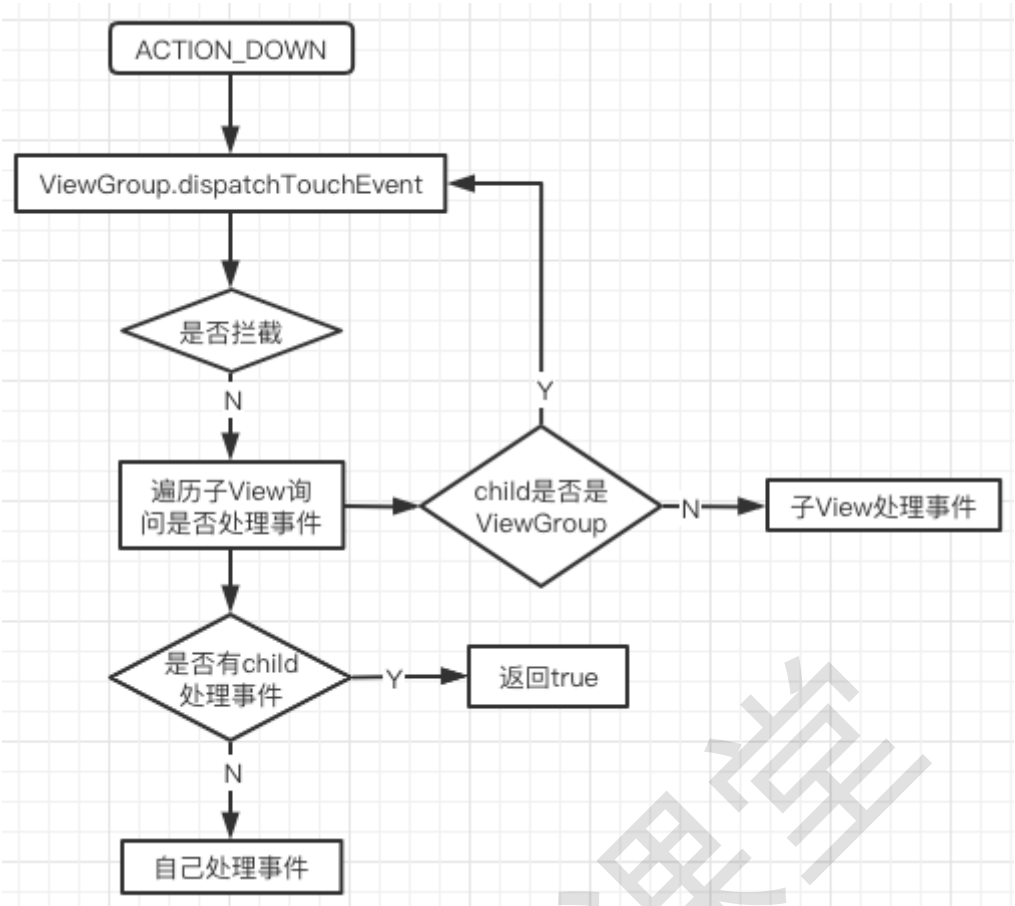
ViewGroup对事件的处理

首先我们来看当事件传到 ViewGroup 后，它的处理流程是怎样的。

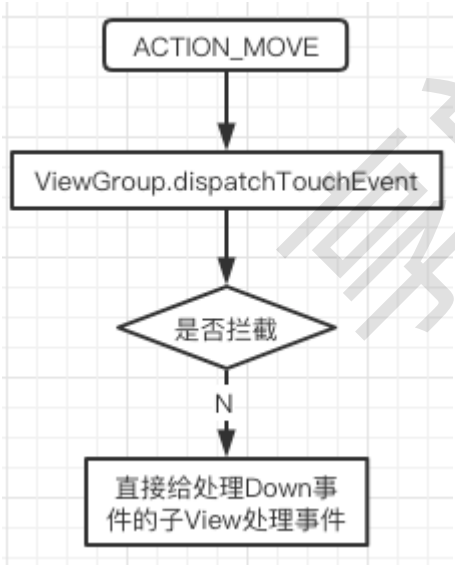
```
// ViewGroup.java
public boolean dispatchTouchEvent(MotionEvent ev) {
    // 返回给父容器，handled 为 true 表示事件被 ViewGroup或它的子孙处理了，false表示不处理该事件。
    boolean handled = false;
    // 事件安全行判断，正常情况都为true
    if (onFilterTouchEventForSecurity(ev)) {
        // 检测是否拦截，即不再向下分发给子View
        final boolean intercepted;
        if (actionMasked == MotionEvent.ACTION_DOWN
            || mFirstTouchTarget != null) {
            final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
            if (!disallowIntercept) {
                // 父容器可以重写 onInterceptTouchEvent 方法，拦截子View事件
                intercepted = onInterceptTouchEvent(ev);
            } else {
                intercepted = false;
            }
        } else {
            intercepted = true;
        }
        // 每来一个事件都会初始化下面两个值
        TouchTarget newTouchTarget = null;
        boolean alreadyDispatchedToNewTouchTarget = false;
        // 判断是否取消或拦截，如果取消或者拦截，则不进入 if 语句
        if (!canceled && !intercepted) {
            // 判断是否是 down 事件或其他的，此处只需关心是否是down事件，如果不是，if 不命中
            if (actionMasked == MotionEvent.ACTION_DOWN
                || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
                || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
                // 子View的个数
                final int childrenCount = mChildrenCount;
                // 有子View，if 语句命中
                if (newTouchTarget == null && childrenCount != 0) {
                    // 遍历所有子View，看是否处理事件
                    for (int i = childrenCount - 1; i >= 0; i--) {
```

```
                        // 拿到子View
                        final View child = getAndVerifyPreorderedView(
                                preorderedList, children, childIndex);
                        // 此处第三个参数是child，所以是询问子View是否处理该事件，进入子View处理事件的流程，
                        // 返回true表示child处理事件，false表示不处理，即处理事件则 if 命中
                        if (dispatchTransformedTouchEvent(ev, false, child,
idBitsToAssign)) {
                            // 关键变量赋值，结果：newTouchTarget == mFirstTouchTarget != null
                            newTouchTarget = addTouchTarget(child,
idBitsToAssign);
                            alreadyDispatchedToNewTouchTarget = true;
                            // 推出for循环，事件已经有子View处理了，不再询问其他的子View
                            break;
                        }
                    }
                }
            }
        }

        // 分发给触摸目标
        if (mFirstTouchTarget == null) {
            // 拦截子View时，或者没有子View处理事件时，则走自己的事件处理流程
            handled = dispatchTransformedTouchEvent(ev, canceled, null,

 TouchTarget.ALL_POINTER_IDS);
        } else {
            // 单指操作，while 只会执行一次
            while (target != null) {
                // down事件时，有子View处理事件，则 if 命中
                if (alreadyDispatchedToNewTouchTarget && target ==
newTouchTarget) {
                    handled = true;
                } else {
                    // 子View处理事件，且是 move 事件时，才走这里，第三个参数，指的就是处理down事件的子View，move事件接着由它处理
                    if (dispatchTransformedTouchEvent(ev, cancelChild,
                                                      target.child,
target.pointerIdBits)) {
                        handled = true;
                    }
                }
            }
        }
    }

    // handled 为true 表示 这个ViewGroup或者它的子孙处理事件，为false表示不处理
    return handled;
}
```

事件分发流程，主要是分析Down和Move事件，是如何分发的。流程图如下：
Down事件流程图：



Move事件流程图：

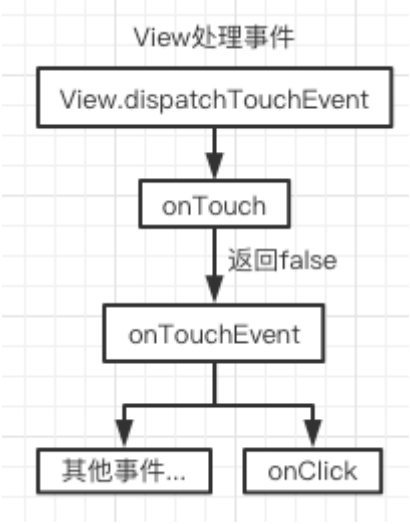


View处理事件分析

```
// View.java
public boolean dispatchTouchEvent(MotionEvent event) {
    if (onFilterTouchEventForSecurity(event)) {
        // 执行 onTouch 的回调方法
        if (li != null && li.mOnTouchListener != null
            && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }
        // onTouch的方法的返回值影响 onTouchEvent 的执行，为false，onTouchEvent才执行
```

```
        // onTouchEvent 中会分别对不同事件进行处理
        if (!result && onTouchEvent(event)) {
            result = true;
        }
    }
    // 返回给父容器，为 true 表示处理该事件，反之不处理
    return result;
}
```

流程图如下：



总结

正常事件分发的流程就是上面介绍的，View的dispatchTouchEvent方法是用来处理事件的，ViewGroup重写了dispatchTouchEvent方法，作用变成了分发事件，一旦要处理事件还是调用的View中的dispatchTouchEvent方法。
但是真实情况下的事件处理是非常复杂的，每一个控件它自己可能又会重写 dispatchTouchEvent 方法，或事件流程中的任一方法。所以只有更好的理解了事件分发流程，才能更好的分析并解决真实项目中遇到的冲突问题。

7.4 事件冲突怎么解决？（东方头条）~leo

这道题想考察什么?

这道题想考察同学对事件冲突的解决是否掌握了。

考生应该如何回答

事件冲突的解决办法分两种：内部拦截法和外部拦截法。

内部拦截法

首先我们来介绍下内部拦截法。它的处理方式是：

```
第一块代码
// 子View中添加如下代码：
@Override
public boolean dispatchTouchEvent(MotionEvent event) {
    int x = (int) event.getX();
```

```
    int y = (int) event.getY();

    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN: {
            getParent().requestDisallowInterceptTouchEvent(true);
            break;
        }
        case MotionEvent.ACTION_MOVE: {
            int deltaX = x - mLastX;
            int deltaY = y - mLastY;
            // 1
            if (Math.abs(deltaX) > Math.abs(deltaY)) {
                getParent().requestDisallowInterceptTouchEvent(false);
            }
            break;
        }
        case MotionEvent.ACTION_UP: {
            break;
        }
        default:
            break;
    }

    mLastX = x;
    mLastY = y;
    return super.dispatchTouchEvent(event);
}
```

第二块代码

```
// 父容器中添加如下代码
public boolean onInterceptTouchEvent(MotionEvent event) {
    // 1
    if (event.getAction() == MotionEvent.ACTION_DOWN){
        super.onInterceptTouchEvent(event);
        return false;
    }
    return true;
}
```

首先我们介绍子View中添加的代码，`getParent().requestDisallowInterceptTouchEvent(true);`的作用，它的代码如下：

第三块代码

```
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {
  if (disallowIntercept == ((mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0)) {
    // We're already in this state, assume our ancestors are too
    return;
  }

  if (disallowIntercept) {
    mGroupFlags |= FLAG_DISALLOW_INTERCEPT;
  } else {
    mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
  }

  // Pass it up to our parent
```

```
  if (mParent != null) {
    mParent.requestDisallowInterceptTouchEvent(disallowIntercept);
  }
}
```

可以看出，通过方法参数disallowIntercept的值，控制了mGroupFlags的值，而这个值是控制父容器是否可以拦截子View的关键代码，如下：

第四块代码

```
public boolean dispatchTouchEvent(MotionEvent ev) {
  if (actionMasked == MotionEvent.ACTION_DOWN
      || mFirstTouchTarget != null) {
    // 1
    final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) !=
0;
    // 2
    if (!disallowIntercept) {
      intercepted = onInterceptTouchEvent(ev);
      ev.setAction(action); // restore action in case it was changed
    } else {
      intercepted = false;
    }
  } else {
    // There are no touch targets and this action is not an initial down
    // so this view group continues to intercept touches.
    intercepted = true;
  }
}
```

通过代码1和2可以知道，当调用 requestDisallowInterceptTouchEvent 传入参数 true 时，disallowIntercept 为 true，就会导致 intercepted = false，从而父容器事件必须要分发给子View。
那父容器中添加的代码是干什么呢？这就要说到事件冲突的解决思路了。我们知道一个事件只能由一个控件处理，而事件冲突实际上就是在不同情况下，按我们的需求分配事件给对应的控件处理。例如：子View是左右滑动的，父容器是上下滑动的，那我们就希望当手指左右滑的时候是子View在动，上下滑的时候是父容器在动。
既然我们的 MotionEvent.ACTION_DOWN 事件是分发给 子View 的，说明事件是由子View根据情况分发的，这个对应的就是 第一块代码的标记1处，此处根据水平和垂直滑动的距离判断出用户是在水平滑动还是垂直滑动。如果事件需要给父容器，则设置 requestDisallowInterceptTouchEvent(false)。然后结合第二块代码中的 onInterceptTouchEvent 返回 true，将分发给子View的事件抢过来，如何做到的呢？代码如下：

第五块代码

```
// 1
final boolean cancelChild = resetCancelNextUpFlag(target.child)
  || intercepted;
// 2
if (dispatchTransformedTouchEvent(ev, cancelChild,
                                  target.child, target.pointerIdBits)) {
  handled = true;
}
```

第六块代码

```
private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel,
```

```
                                                View child, int
desiredPointerIdBits) {
  final int oldAction = event.getAction();
  // 1
  if (cancel || oldAction == MotionEvent.ACTION_CANCEL) {
    event.setAction(MotionEvent.ACTION_CANCEL);
    if (child == null) {
      handled = super.dispatchTouchEvent(event);
    } else {
      handled = child.dispatchTouchEvent(event);
    }
    event.setAction(oldAction);
    return handled;
  }
}
```

第五块代码中的1处，因为 intercepted = true，所以cancelChild = true，从而会进入第六块代码1处，子View这个时候会执行 取消事件，将事件让出来，在下次MOVE事件处理时就会交给父容器处理。从而达到根据需求分配事件的要求。即解决了事件冲突。
那为什么我们还需要在第二块代码中添加标记1处的 if 判断语句呢？原因需要看如下代码：

```
第七块代码
public boolean dispatchTouchEvent(MotionEvent ev) {
  if (actionMasked == MotionEvent.ACTION_DOWN) {
    resetTouchState();
  }
}
private void resetTouchState() {
  mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
}
```

可以看到，resetTouchState 会重制 mGroupFlags 的值，而该代码在事件为 MotionEvent.ACTION_DOWN 的时候一定会执行。这就导致 disallowIntercept 在事件为 MotionEvent.ACTION_DOWN 的时候一定为 false，即 onInterceptTouchEvent 一定会执行。所以父容器只能在其他事件拦截子View。
外部拦截法的思路一样，在此就不赘婿了。

7.5 View分发反向制约的方法？（字节跳动）~leo

这道题想考察什么?

这道题想考察同学对 requestDisallowInterceptTouchEvent 方法的了解。

考生应该如何回答

子View拿到Down事件后，通过调用 requestDisallowInterceptTouchEvent 可以反向制约父容器对事件的拦截。

```
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {
    // 1.通过disallowIntercept的值，给mGroupFlags设置不同值。
    if (disallowIntercept) {
        mGroupFlags |= FLAG_DISALLOW_INTERCEPT;
    } else {
        mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
    }

    // 2.请求传给父类
    if (mParent != null) {
        mParent.requestDisallowInterceptTouchEvent(disallowIntercept);
    }
}
```

通过上面两步，实现了对所有父容器的 mGroupFlags 值的设置。那这个值有什么用呢?
我们知道，父容器是通过调用 onInterceptTouchEvent 方法，来实现对子View的事件拦截，而这个方法的代码如下:

```
final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
if (!disallowIntercept) {
    intercepted = onInterceptTouchEvent(ev);
}
```

可以看到 disallowIntercept 的值，直接影响到 onInterceptTouchEvent 方法的执行，而 mGroupFlags 的值又可以确定 disallowIntercept 的值，由此可知，子 View 通过调用 requestDisallowInterceptTouchEvent 方法，传入 true 即可反向制约父容器对自己的拦截。
不过这个地方有一个注意点，如果是down事件，则该方法没有用，因为在上面 onInterceptTouchEvent 这块代码执行前，会先执行如下代码:

```
if (actionMasked == MotionEvent.ACTION_DOWN) {
    resetTouchState();
}
```

```
private void resetTouchState() {
    mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
}
```

可以看到，当事件为 ACTION_DOWN 时，会重置 mGroupFlags 的值，从而导致 onInterceptTouchEvent 方法肯定会执行。

7.6 自定义Behavior，NestScroll，NestChild。（东方头条）~leo

7.7 View.inflater过程与异步inflater（东方头条）~leo

> View 的异步 Inflate+ 全局缓存：加速你的页面
> Android AsyncLayoutInflater 源码解析

这道题想考察什么?

考察同学对View渲染的理解

考生应该如何回答

1.View.inflate过程

```
// View.java
public static View inflate(Context context, @LayoutRes int resource, ViewGroup
root) {
    LayoutInflater factory = LayoutInflater.from(context);
    return factory.inflate(resource, root);
}
```

1-1.LayoutInflater.inflate

```
// LayoutInflater.java
public View inflate(@LayoutRes int resource, @Nullable ViewGroup root) {
    return inflate(resource, root, root != null);
}

public View inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean
attachToRoot) {
    final Resources res = getContext().getResources();

    // 获取资源解析器 XmlResourceParser
    XmlResourceParser parser = res.getLayout(resource);
    try {
        // 解析View
        return inflate(parser, root, attachToRoot);
    } finally {
        parser.close();
    }
}

public View inflate(XmlPullParser parser, @Nullable ViewGroup root, boolean
attachToRoot) {
    synchronized (mConstructorArgs) {
        final Context inflaterContext = mContext;

        // 存储传进来的根布局
        View result = root;

        try {
            final String name = parser.getName();

            //解析 merge标签，rInflate方法会将 merge标签下面的所有子 view添加到根布局中
            // 这也是为什么 merge 标签可以简化布局的效果
            if (TAG_MERGE.equals(name)) {
                // 必须要有父布局，否则报错
                if (root == null || !attachToRoot) {
```

```
                    throw new InflateException("<merge /> can be used only with a
valid "
                }
                // 解析 merge标签下的所有的 View，添加到根布局中
                rInflate(parser, root, inflaterContext, attrs, false);
            } else {
                // 创建 root View
                final View temp = createViewFromTag(root, name, inflaterContext,
attrs);

                ViewGroup.LayoutParams params = null;

                // 第一种情况：root != null，且attachToRoot = false，root给View设置
params，返回View
                if (root != null) {
                    // 通过该方法创建与root匹配的 LayoutParams
                    params = root.generateLayoutParams(attrs);
                    if (!attachToRoot) {
                        // 将 root 提供的 LayoutParams 设置给View
                        temp.setLayoutParams(params);
                    }
                }

                // 渲染子View
                rInflateChildren(parser, temp, attrs, true);

                // 第二种情况：root != null，且attachToRoot = true，新解析出来的 View
会被 add 到 root 中去，然后将 root 作为结果返回
                if (root != null && attachToRoot) {
                    root.addView(temp, params);
                }

                // 第三种情况：root == null，或者 attachToRoot = false，直接返回View
                if (root == null || !attachToRoot) {
                    result = temp;
                }
            }
        }
        // 返回result，可能是View，也可能是root
        return result;
    }
}
```

inflate 的三个参数，其中第二和第三参数分下面几种情况：

- 当 root != null 且 attachToRoot == false 时，新解析的 View 会直接作为结果返回，而且 root 会为新解析的 View 生成 LayoutParams 并设置到该 View 中去
- 当 root != null 且 attachToRoot == true 时，新解析出来的 View 会被 add 到 root 中去，然后将 root 作为结果返回
- 当 root == null 且 attachToRoot == false 时，新解析的 View 会直接作为结果返回

1-2.createViewFromTag

```
// LayoutInflater.java
private View createViewFromTag(View parent, String name, Context context,
AttributeSet attrs) {
    return createViewFromTag(parent, name, context, attrs, false);
```

```
}

View createViewFromTag(View parent, String name, Context context, AttributeSet
attrs,
                       boolean ignoreThemeAttr) {
    if (view == null) {
        if (-1 == name.indexOf('.')) {
            // 创建 sdk 的View，例如：LinearLayout、Button等，最终还是调用的createView
            view = onCreateView(context, parent, name, attrs);
        } else {
            // 创建 自定义View
            view = createView(context, name, null, attrs);
        }
    }
    return view;
}
```

1-3.createView

```
public final View createView(@NonNull Context viewContext, @NonNull String name,
                             @Nullable String prefix, @Nullable AttributeSet
attrs)
    throws ClassNotFoundException, InflateException {
    // 从缓存中获取constructor
    Constructor<? extends View> constructor = sConstructorMap.get(name);
    Class<? extends View> clazz = null;
    try {
        // 没有缓存，则创建
        if (constructor == null) {
            // 通过全类名获取 Class 对象，如果是sdk的View，需要拼接字符串
            clazz = Class.forName(prefix != null ? (prefix + name) : name,
false,

 mContext.getClassLoader()).asSubclass(View.class);

            // 获取 Constructor 对象
            constructor = clazz.getConstructor(mConstructorSignature);
            constructor.setAccessible(true);
            // 用静态 HashMap保存，优化性能，同一个类，下次就可以直接拿这个constructor创建
            sConstructorMap.put(name, constructor);
        } else {
        }

        try {
            // 创建View对象
            final View view = constructor.newInstance(args);
            // 返回创建的View对象
            return view;
        }
    }
}
```

####1-4.总结

整个View.inflate，就是先经过xml解析，然后通过反射创建View对象的一个流程。

2.异步inflate

Google开发者在v4包中增加了一个用来异步inflate layouts的帮助类。

2-1.AsyncLayoutInflater

AsyncLayoutInflater 是来帮助作异步加载 layout 的，inflate(int, ViewGroup, OnInflateFinishedListener) 方法运行结束以后 OnInflateFinishedListener 会在主线程回调返回 View；这样作旨在 UI 的懒加载或者对用户操做的高响应。
简单的说咱们知道默认状况下 setContentView 函数是在 UI 线程执行的，其中有一系列的耗时动做：Xml的解析、View的反射建立等过程一样是在UI线程执行的，AsyncLayoutInflater 就是来帮咱们把这些过程以异步的方式执行，保持UI线程的高响应。
AsyncLayoutInflater 只有一个构造函数及普通调用函数：inflate(int resid, ViewGroup parent, AsyncLayoutInflater.OnInflateFinishedListener callback)，使用也很是方便。

```
 @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        new AsyncLayoutInflater(AsyncLayoutActivity.this)
                .inflate(R.layout.async_layout, null, new
AsyncLayoutInflater.OnInflateFinishedListener() {
                    @Override
                    public void onInflateFinished(View view, int resid,
ViewGroup parent) {
                        setContentView(view);
                    }
                });
        // 别的操做
    }
```

2-2.AsyncLayoutInflater 构造函数

```
    public AsyncLayoutInflater(@NonNull Context context) {
        mInflater = new BasicInflater(context);
        mHandler = new Handler(mHandlerCallback);
        mInflateThread = InflateThread.getInstance();
    }
```

主要作了三件事情：
1.建立 BasicInflater。BasicInflater 继承自 LayoutInflater，只是覆写了 onCreateView：优先加载这三个前缀的 Layout，而后才按照默认的流程去加载，由于大多数状况下咱们 Layout 中使用的View都在这三个 package 下。

```
private static class BasicInflater extends LayoutInflater {
        private static final String[] sClassPrefixList = {
            "android.widget.",
            "android.webkit.",
            "android.app."
        };
        @Override
        protected View onCreateView(String name, AttributeSet attrs) throws
ClassNotFoundException {
            for (String prefix : sClassPrefixList) {
                try {
                    View view = createView(name, prefix, attrs);
                    if (view != null) {
```

```
                        return view;
                    }
                } catch (ClassNotFoundException e) {
                }
            }
            return super.onCreateView(name, attrs);
        }
    }
```

2.建立 Handler。建立 Handler 和它普通的做用同样，就是为了线程切换，AsyncLayoutInflater 是在异步里 inflate layout，那建立出来的 View 对象须要回调给主线程，就是经过 Handler 来实现的。
3.获取 InflateThread 对象。InflateThread 从名字上就好理解，是来作 Inflate 工做的工做线程，经过 InflateThread.getInstance 能够猜想 InflateThread 里面是一个单例，默认只在一个线程中作全部的加载工做，这个类咱们会在下面重点分析

2-3.inflate

```
public void inflate(@LayoutRes int resid, @Nullable ViewGroup parent,
            @NonNull OnInflateFinishedListener callback) {
        if (callback == null) {
            throw new NullPointerException("callback argument may not be
null!");
        }
        InflateRequest request = mInflateThread.obtainRequest();
        request.inflater = this;
        request.resid = resid;
        request.parent = parent;
        request.callback = callback;
        mInflateThread.enqueue(request);
    }
```

首先会经过 InflateThread 去获取一个 InflateRequest，其中有一堆的成员变量。为何须要这个类呢？由于后续异步 inflate 须要一堆的参数（对应 InflateRequest 中的变量），会致使方法签名过长，而使用 InflateRequest 就避免了不少个参数的传递。

```
private static class InflateRequest {
        AsyncLayoutInflater inflater;
        ViewGroup parent;
        int resid;
        View view;
        OnInflateFinishedListener callback;

        InflateRequest() {
        }
    }
```

接下来对 InflateRequest 变量赋值以后会将其加到 InflateThread 中的一个队列中等待执行。

```
public void enqueue(InflateRequest request) {
        try {
            mQueue.put(request);
        } catch (InterruptedException e) {
            throw new RuntimeException(
                    "Failed to enqueue async inflate request", e);
        }
    }
```

2-4.InflateThread

```
private static class InflateThread extends Thread {
        private static final InflateThread sInstance;
        static {
            // 静态代码块，确保只会建立一次，而且建立即start。
            sInstance = new InflateThread();
            sInstance.start();
        }
       // 单例，避免重复创建线程带来的开销
        public static InflateThread getInstance() {
            return sInstance;
        }

        // 阻塞队列（保存封装过得request）
        private ArrayBlockingQueue<InflateRequest> mQueue = new
ArrayBlockingQueue<>(10);
        private SynchronizedPool<InflateRequest> mRequestPool = new
SynchronizedPool<>(10);

        public void runInner() {
            InflateRequest request;
            try {
                request = mQueue.take(); // 虽然是死循环，但队列中没有数据会阻塞，不占用
cpu
            } catch (InterruptedException ex) {
                // Odd, just continue
                Log.w(TAG, ex);
                return;
            }

            try {
            //解析xml的位置在这
                request.view = request.inflater.mInflater.inflate(
                        request.resid, request.parent, false);
            } catch (RuntimeException ex) {
                // Probably a Looper failure, retry on the UI thread
                Log.w(TAG, "Failed to inflate resource in the background!
Retrying on the UI"
                        + " thread", ex);
            }
            // 发送消息到主线程
            Message.obtain(request.inflater.mHandler, 0, request)
                    .sendToTarget();
        }

        @Override
        public void run() {
```

```
            // 异步加载布局并使用handler进行
            while (true) {
                runInner();
            }
        }

        public InflateRequest obtainRequest() {
            InflateRequest obj = mRequestPool.acquire();
            if (obj == null) {
                obj = new InflateRequest();
            }
            return obj;
        }

        public void releaseRequest(InflateRequest obj) {
            obj.callback = null;
            obj.inflater = null;
            obj.parent = null;
            obj.resid = 0;
            obj.view = null;
            mRequestPool.release(obj);
        }

        public void enqueue(InflateRequest request) {
            try {
                mQueue.put(request);// 添加到缓存队列中
            } catch (InterruptedException e) {
                throw new RuntimeException(
                        "Failed to enqueue async inflate request", e);
            }
        }
    }
```

syncLayoutInflater使用的是ArrayBlockingQueue来实现此模型，所以如果连续大量的调用AsyncLayoutInflater创建布局，可能会造成缓冲区阻塞。
enqueue 函数：只是插入元素到 mQueue 队列中，若是元素过多那么是有排队机制的；
runInner 函数：运行于循环中，从 mQueue 队列取出元素，调用 inflate 方法返回主线程；

7.8 inflater为什么比自定义View慢？（东方头条）~leo

7.9 View中onTouch，onTouchEvent和onClick的执行顺序（58京东）~leo

这道题想考察什么?

这道题考察同学对事件的处理流程是否熟悉。

考生应该如何回答

这些方法的执行主要在 View 类的 dispatchTouchEvent 中。

onTouch的执行

首先会判断用户是否调用了setOnTouchListener，如果调用了，则说明初始化了 ListenerInfo 和 OnTouchListener；接着会判断 View 是否是 enabled，如果是，这个时候才会执行 onTouch 回调方法。

onTouchEvent的执行

onTouchEvent 是否执行，由 onTouch 的返回值影响。如果 onTouch 返回 true，则 result 为true，这个时候由于 短路 && ，所以 onTouchEvent 方法不会执行。反之 onTouchEvent 方法才会执行。

onClick的执行

onTouchEvent 中会对 View 的不同事件进行处理，而 onClick 就是在 ACTION_UP 事件时执行的，执行的前提条件也是需要用户调用了 setOnClickListener 监听方法。

7.10 怎么拦截事件 onTouchEvent如果返回false onClick还会执行么？（58 京东）~leo

这道题想考察什么?

这道题想考察同学对事件分发的了解。

考生应该如何回答

怎么拦截事件

父容器通过重写 onInterceptTouchEvent 方法并返回true 达到拦截事件的目的。代码如下：

```
if (actionMasked == MotionEvent.ACTION_DOWN
                    || mFirstTouchTarget != null) {
    if (!disallowIntercept) {
        intercepted = onInterceptTouchEvent(ev);
    }
}
// 1
if (!canceled && !intercepted) {
    // 处理事件分发给子View
    ...
}
```

可以看出，当 intercepted 为 true，会导致代码1处的if不会被命中，从而事件不会分发给子View。

onTouchEvent如果返回false onClick还会执行么?

就这题来说，因为onTouchEvent执行了，onClick是在 onTouchEvent 方法内部执行的，所以onClick是否执行，与 onTouchEvent 方法的返回值显然是没关系的。

7.11 事件的分发机制，责任链模式的优缺点 （美团）~leo

这道题想考察什么?

这道题想考察同学对责任链模式是否理解了。

考生应该如何回答

责任链模式的定义

责任链模式是行为型设计模式之一。
责任链模式的定义：使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

事件分发机制中的责任链模式

当用户接触屏幕时，Android都会将对应的事件包装成一个事件对象从 ViewTree 的顶部至上而下地分发传递。
我们直接看ViewGroup.java的代码，如下：

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (onFilterTouchEventForSecurity(ev)) {
        // 没有拦截和取消，则向子View传递事件
        if (!canceled && !intercepted) {
            if (actionMasked == MotionEvent.ACTION_DOWN
                || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
                || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
                // children的个数
                final int childrenCount = mChildrenCount;
                if (newTouchTarget == null && childrenCount != 0) {
                    // 寻找能够接收事件的子View，从前往后扫描 children
                    final ArrayList<View> preorderedList =
buildTouchDispatchChildList();
                    // 拿到当前View的子View的集合
                    final View[] children = mChildren;
                    // 循环遍历child
                    for (int i = childrenCount - 1; i >= 0; i--) {
                        // 该方法返回true表示事件被改child处理，事件不再传递，反之继续传递
事件。
                        // 如果child是一个ViewGroup，则递归调用重复此过程。如果child是一
个View，则不再重复此分发过程。
                        if (dispatchTransformedTouchEvent(ev, false, child,
idBitsToAssign)) {
                            ... ...
                        }
                    }
                }
            }
        }
    }
```

```
    }
    return handled;
}
```

责任链模式的优缺点

责任链模式的优点：可以对请求者和处理者关系解耦，提高代码的灵活性。
责任链模式的缺点：处理者太多影响性能，特别是一些递归调用中，要慎用。

7.12 动画的分类以及区别（车和家）~colin

这道题想考察什么?

1. 是否了解安卓的动画?

考察的知识点

1. 动画的分类
2. 动画的区别

考生应该如何回答

1. 首先回答下安卓的动画种类，一共分为三种

- 属性动画：动态的改变属性产生动画效果；
- 补间动画：对View的平移，旋转，缩放，透明产生效果；
- 帧动画：由数张图片连续播放产生的动画效果；

1. 三类动画的区别。

- 属性动画：改变了动画属性，因属性动画在动画过程中动态的改变的对象的属性，从而达到的动画效果；
- 补间动画：无改变动画的属性，只在动画过程中对图像进行了变化（平移、缩放、旋转和透明），从而达到了动画效果；效果连贯自然、但最终会回到原来位置。
- 帧动画：无改变动画的属性，逐帧播放从而达到了动画效果。制作简单但效果单一，且占用空间较大。

7.13 属性动画与普通的动画有什么区别？（车和家）~leo

这道题想考察什么?

这道题想考察同学对 动画 的理解。

考生应该如何回答

属性动画与普通的动画有什么区别：补间动画仅仅是对 View 在视觉效果上做了移动、缩放、旋转和淡入淡出的效果，其实并没有真正改变 View 的属性。属性动画经过动效后响应触摸事件的位置和视觉效果相同。

属性动画和补间动画的基本编写方式

属性动画，你可以用下面的两种方式。

```
ObjectAnimator.ofFloat(tv1, "translationX", 0f, 500f)
                    .setDuration(1000)
                    .start()
// 或者像这样
tv1.animate().setDuration(1000).translationX(500f)
```

补间动画

```
val anim = TranslateAnimation(0f, 500f, 0f, 0f)
anim.duration = 1000
anim.fillAfter = true     // 设置保留动画后的状态
tv1.startAnimation(anim)
```

属性动画的使用注意点

对于属性动画来说，尤其需要注意的是操作的属性需要有 set 和 get 方法，不然你的 ObjectAnimator 操作就不会生效。比如水平平移，我们知道，View 的 translationX 属性设置方法接受的是 float 值，所以你把上面的操作编写为 ofInt 就不会生效，比如：

```
ObjectAnimator.ofInt(tv1, "translationX", 0, 500)
                    .setDuration(1000)
                    .start()
```

对于我们需要用到但又没有写好的属性，比如我们自定义一个进度条 View，我们需要实时展示进度，这时候我们就可以自己定义一个属性，并让它支持 set 和 get，那么在外面就可以对这个自定义的 View 做属性动画操作了。

属性动画和补间动画工作原理

属性动画

属性动画的工作原理很简单，其实就是在一定的时间间隔内，通过不断地对值进行改变，并不断将该值赋给对象的属性，从而实现该对象在属性上的动画效果。
从上述工作原理可以看出属性动画有两个非常重要的类：ValueAnimator 类 & ObjectAnimator 类，二者的区别在于：

- ValueAnimator 类是先改变值，然后 手动赋值 给对象的属性从而实现动画；是 间接 对对象属性进行操作；而 ValueAnimator 类本质上是一种 改变值 的操作机制。
- ObjectAnimator 类是先改变值，然后 自动赋值 给对象的属性从而实现动画；是 直接 对对象属性进行操作；可以理解为：ObjectAnimator 更加智能、自动化程度更高。

补间动画

补间动画，我们可以跟进源码，看看到底做了什么操作。

```
public void startAnimation(Animation animation) {
    animation.setStartTime(Animation.START_ON_FIRST_FRAME);
    setAnimation(animation);
    invalidateParentCaches();
    invalidate(true);
}
```

看到了非常明显 invalidate() 方法，很明显，补间动画在执行的时候，直接导致了 View 执行 onDraw() 方法。总的来说，补间动画的核心本质就是在一定的持续时间内，不断改变 Matrix 变换，并且不断刷新的过程。

为什么属性动画移动一个 View 后，目标位置还可以响应触摸事件呢?

在 ViewGroup 没有重写 onInterceptTouchEvent() 方法进行事件拦截的时候，我们一定会通过其 dispatchTouchEvent() 方法进行事件分发，而决定我们哪一个子 View 响应我们的触摸事件的条件又是我们手指的位置必须在这个子 View 的边界范围内，也就是 left、right、top、bottom 这四个属性形成的矩形区域。
那么，如果我们的 View 已经进行了属性动画后，现在手指响应的触摸位置区域肯定不是 View 自己的 left、right、top、bottom 这四个属性形成的区域了，但这个 View 却神奇的响应了我们的点击事件。

```
/**
 * Returns a MotionEvent that's been transformed into the child's local
coordinates.
 *
 * It's the responsibility of the caller to recycle it once they're finished
with it.
 * @param event The event to transform.
 * @param child The view whose coordinate space is to be used.
 * @return A copy of the the given MotionEvent, transformed into the given
View's coordinate
 *         space.
 */
private MotionEvent getTransformedMotionEvent(MotionEvent event, View child) {
    final float offsetX = mScrollX - child.mLeft;
    final float offsetY = mScrollY - child.mTop;
    final MotionEvent transformedEvent = MotionEvent.obtain(event);
    transformedEvent.offsetLocation(offsetX, offsetY);
    if (!child.hasIdentityMatrix()) {
        transformedEvent.transform(child.getInverseMatrix());
    }
    return transformedEvent;
}

/**
 * Returns true if the transform matrix is the identity matrix.
 * Recomputes the matrix if necessary.
 *
 * @return True if the transform matrix is the identity matrix, false otherwise.
 */
final boolean hasIdentityMatrix() {
    return mRenderNode.hasIdentityMatrix();
}
```

```
/**
 * Utility method to retrieve the inverse of the current mMatrix property.
 * We cache the matrix to avoid recalculating it when transform properties
 * have not changed.
 *
 * @return The inverse of the current matrix of this view.
 * @hide
 */
public final Matrix getInverseMatrix() {
    ensureTransformationInfo();
    if (mTransformationInfo.mInverseMatrix == null) {
        mTransformationInfo.mInverseMatrix = new Matrix();
    }
    final Matrix matrix = mTransformationInfo.mInverseMatrix;
    mRenderNode.getInverseMatrix(matrix);
    return matrix;
}
```

原来，ViewGroup 在 getTransformedMotionEvent() 方法中会通过子 View 的 hasIdentityMatrix() 方法来判断子 View 是否应用过位移、缩放、旋转之类的属性动画。如果应用过的话，那还会调用子 View 的 getInverseMatrix() 做「反平移」操作，然后再去判断处理后的触摸点是否在子 View 的边界范围内。

补间动画和属性动画改变的是同一个 Matrix 么?

肯定不是。
属性动画所影响的 Matrix，是在 View 的 mRenderNode 中的 stagingProperties 里面的，这个 Matrix 在每个 View 之间都是独立的，所以可以各自保存不同的变换状态。
而补间动画所操作的 Matrix，其实是借用了它父容器的一个叫 mChildTransformation 的属性 (里面有 Matrix)，通过 getChildTransformation 获得。
也就是说，一个 ViewGroup 中，无论它有几个子 View 都好，在这些子 View 播放补间动画的时候，都是共用同一个 Transformation 对象的(也就是共用一个 Matrix)，这个对象放在 ViewGroup 里面。
既然是共用，那为什么可以同时播放好几个动画，而互相不受影响呢?
因为在补间动画更新每一帧的时候，父容器的 mChildTransformation 里面的 Matrix，都会被 reset() 。
每次重置 Matrix 而不受影响的原因：这些补间动画，都是基于当前播放进度，来计算出绝对的动画值并应用的，保存旧动画值是没有意义的。
就拿位移动画 TranslateAnimation 来说，比如它要向右移动 500，当前的播放进度是 50%，那就是已经向右移动了 250，在 View 更新帧的时候，就会把这个向右移动了250的 Matrix 应用到 Canvas 上，当下次更新帧时，比如进度是 60%，那计算出来的偏移量就是 300，这时候，已经不需要上一次的旧值 250 了，就算 Matrix 在应用前被重置了，也不影响最后的效果。

7.14 插值器 估值器的区别（车和家）~leo

这道题想考察什么?

这道题想考察同学对 插值器 估值器 的理解。

考生应该如何回答

我们都知道对于属性动画可以对某个属性做动画，而插值器（Interpolator）和估值器（Evaluator）在其中扮演了重要角色，下面先了解下TimeInterpolator和TypeEvaluator。属性动画不熟悉的见章节《属性动画与普通的动画有什么区别》。

TimeInterpolator（时间插值器）：

1. 作用：根据时间流逝的百分比计算出当前属性值改变的百分比。
2. 系统提供了很多插值器，例如：
    1. LinearInterpolator（线性插值器）：匀速动画。
    2. AccelerateDecelerateInterpolator（加速减速插值器）：动画两头慢，中间快。
    3. DecelerateInterpolator（减速插值器）：动画越来越慢。

TypeEvaluator（类型估值算法，即估值器）：

1. 作用：根据当前属性改变的百分比来计算改变后的属性值。
2. 系统提供了很多估值器，例如：
    1. IntEvaluator：针对整型属性
    2. FloatEvaluator：针对浮点型属性
    3. ArgbEvaluator：针对Color属性

那么TimeInterpolator和TypeEvaluator是怎么协同工作的呢?
它们是实现非匀速动画的重要手段。属性动画是对属性做动画，属性要实现动画，首先由TimeInterpolator（插值器）根据时间流逝的百分比计算出当前属性值改变的百分比，并由插值器将这个百分比返回，这个时候插值器的工作就完成了。比如插值器返回的值是0.5，很显然我们要的不是0.5，而是当前属性的值，即当前属性变成了什么值，这就需要估值器根据当前属性改变的百分比来计算改变后的属性值，根据这个属性值，我们就可以设置当前属性的值了。

自定义插值器

自定义插值器需要实现Interpolator / TimeInterpolator接口 & 复写getInterpolation()，参考代码如下:

```
//弹性插值器

public class SpringInterpolator implements TimeInterpolator {

    private float factor;//参数因子

    public SpringInterpolator(float factor) {

        this.factor = factor;

    }

    // 复写getInterpolation（）

    @Override

    public float getInterpolation(float input) {

        return (float) (Math.pow(2, -10 * input)

        * Math.sin((input - factor / 4)

        * (2 * Math.PI) / factor) + 1);
```

```
    }

}
```

自定义估值器

根据插值器计算出当前属性值改变的百分比 & 初始值 & 结束值 来计算此刻属性变化的具体值，参考代码如下：

```
// 实现TypeEvaluator接口

public class PointEvaluator implements TypeEvaluator<Point> {

    // 复写evaluate（）

    // 在evaluate（）里写入对象动画过渡的逻辑

    @Override

    public Point evaluate(float fraction, Point startValue, Point endValue) {

        // 根据fraction来计算当前动画的x和y的值

        int x = (int) (startValue.x + fraction * (endValue.x - startValue.x));

        int y = (int) (startValue.y + fraction * (endValue.y - startValue.y));

        // 将计算后的坐标封装到一个新的Point对象中并返回

        return new Point(x, y);

    }

}
```

总结

插值器 是根据自己的公式算出当前整个动画已经执行的百分比。可以直接使用系统提供的，也可以按照自己的需求自定义实现。
估值器 是根据当前的百分比 算出当前的属性值。也是可以直接使用系统提供的，或者自定义实现。

7.15 RecyclerView与ListView的对比，缓存策略，优缺点。（美团）~colin

这道题想考察什么?

1. 是否了解RecyclerView、ListView原理知识?

考察的知识点

1. RecyclerView的布局效果、局部刷新、动画效果、缓存知识
2. ListView的布局效果、局部刷新、动画效果、缓存知识

考生应该如何回答

RecyclerView和ListView都是用于加载大量数据的控件，RecyclerView作为listview的改进加强型，相对于ListView，RecyclerView做出了以下优化：

1.布局效果

ListView 的布局比较单一，只有一个纵向效果；

RecyclerView 的布局效果丰富， 可以在 LayoutMananger 中设置：线性布局（纵向，横向），表格布局，瀑布流布局在RecyclerView 中，如果存在的 LayoutManager 不能满足需求，可以自定义 LayoutManager

2.item点击事件

RecyclerView不支持 item 点击事件，只能用回调接口来设置点击事件

ListView的 item 点击事件直接是setOnItemClickListener

3.局部刷新

在ListView中通常刷新数据是用notifyDataSetChanged()，但是这种刷新数据是全局刷新的（每个item的数据都会重新加载一遍），这样一来就会非常消耗资源；
RecyclerView中可以实现局部刷新，例如：notifyItemChanged()；
如果要在ListView实现局部刷新，依然是可以实现的，当一个item数据刷新时，我们可以在Adapter中，实现一个notifyItemChanged()方法，在方法里面通过这个 item 的 position，刷新这个item的数据

4.动画效果

在RecyclerView中，已经封装好API来实现自己的动画效果；并且如果我们需要实现自己的动画效果，我们可以通过相应的接口实现自定义的动画效果（RecyclerView.ItemAnimator类），然后调用RecyclerView.setItemAnimator() (默认的有SimpleItemAnimator与DefaultItemAnimator）；
但是ListView并没有实现动画效果，但我们可以在Adapter自己实现item的动画效果；

5.缓存区别

层级不同

ListView有两级缓存，在屏幕与非屏幕内。mActivityViews + mScrapViews
RecyclerView比ListView多两级缓存：支持开发者自定义缓存处理逻辑，RecyclerViewPool(缓存池)。并且支持多个离屏ItemView缓存（缓存屏幕外2个 itemView）。 mAttachedScrap + mCacheViews + mViewCacheExtension + mRecyclerPool

缓存内容不同

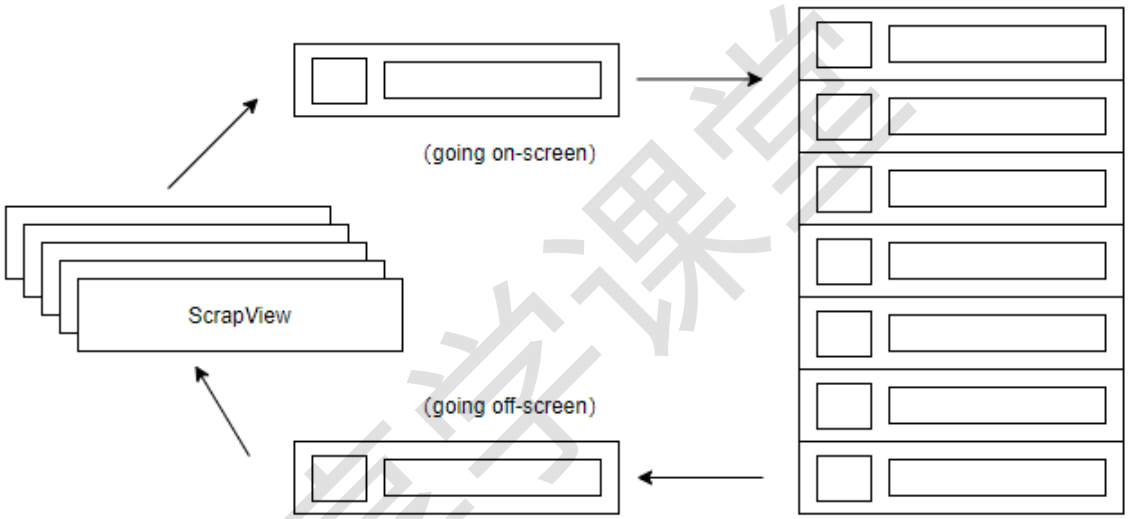
ListView缓存View。
RecyclerView缓存RecyclerView.ViewHolder

RV优势

a.mCacheViews的使用，可以做到屏幕外的列表项ItemView进入屏幕内时也无须bindView快速重用；
b.mRecyclerPool可以供多个RecyclerView共同使用，在特定场景下，如viewpaper+多个列表页下有优势.

缓存区别

1，封装了viewholder，Listview需要自己写ViewHolder缓存，而RecyclerView已经帮我们实现了。
2，RecyclerView的缓存机制有了加强，ListView是2级缓存，而RecyclerView实现了4级缓存。
3，RecyclerView相对于ListView最大的加强是实现了局部刷新，这对于ListView需要刷新全部列表进步很多，特别适用于那些数据源经常发生改变的情况。
下面从缓存机制做出对比：
ListView和RecyclerView的缓存机制大致类似：



过程中，离屏的ItemView即被回收至缓存，入屏的ItemView则会优先从缓存中获取，只是ListView与RecyclerView的实现细节有差异.（这只是缓存使用的其中一个场景，还有如刷新等）。

1. 缓存机制不同
   RecyclerView比ListView多两级缓存，支持多个离ItemView缓存，支持开发者自定义缓存处理逻辑，支持所有RecyclerView共用同一个RecyclerViewPool(缓存池)。

具体来说：
ListView(两级缓存)：

	是否需要回调createView	是否需要回调bindView	生命周期	备注
mActiveViews	否	否	onLayout函数周期内	用于屏幕itemView快速重用
mScrapViews	否	是	与mAdapter一致，当mAdatper被更换时，mScrapViews即被清空	

RecyclerView(四级缓存)：

	是否需要回调createView	是否需要调用bindView	生命周期	备注
mAttachedScrap	否	否	onLayout函数周期中	用于屏幕内itemview快速重用
mCacheViews	否	否	与mAdapter一致,当mAdapter被更换时，mCacheViews即被缓存至mRecyclerPool	默认上限为2，即缓存屏幕外2个itemView
mViewCacheExtension				不直接使用，需要用户在定制，默认不实现
mRecyclerPool	否	是	与自身生命周期一致，不再被引用是即被释放	默认上限为5，技术上可以实现所有RecyclerViewPool共用同一个

ListView和RecyclerView缓存机制基本一致：

1). mActiveViews和mAttachedScrap功能相似，意义在于快速重用屏幕上可见的列表项ItemView，而不需要重新createView和bindView；

2). mScrapView和mCachedViews + mReyclerViewPool功能相似，意义在于缓存离开屏幕的ItemView，目的是让即将进入屏幕的ItemView重用.

3). RecyclerView的优势在于a.mCacheViews的使用，可以做到屏幕外的列表项ItemView进入屏幕内时也无须bindView快速重用；b.mRecyclerPool可以供多个RecyclerView共同使用，在特定场景下，如viewpaper+多个列表页下有优势.客观来说，RecyclerView在特定场景下对ListView的缓存机制做了补强和完善。

2. 缓存不同：

1). RecyclerView缓存RecyclerView.ViewHolder，抽象可理解为：

View + ViewHolder(避免每次createView时调用findViewById) + flag(标识状态)；

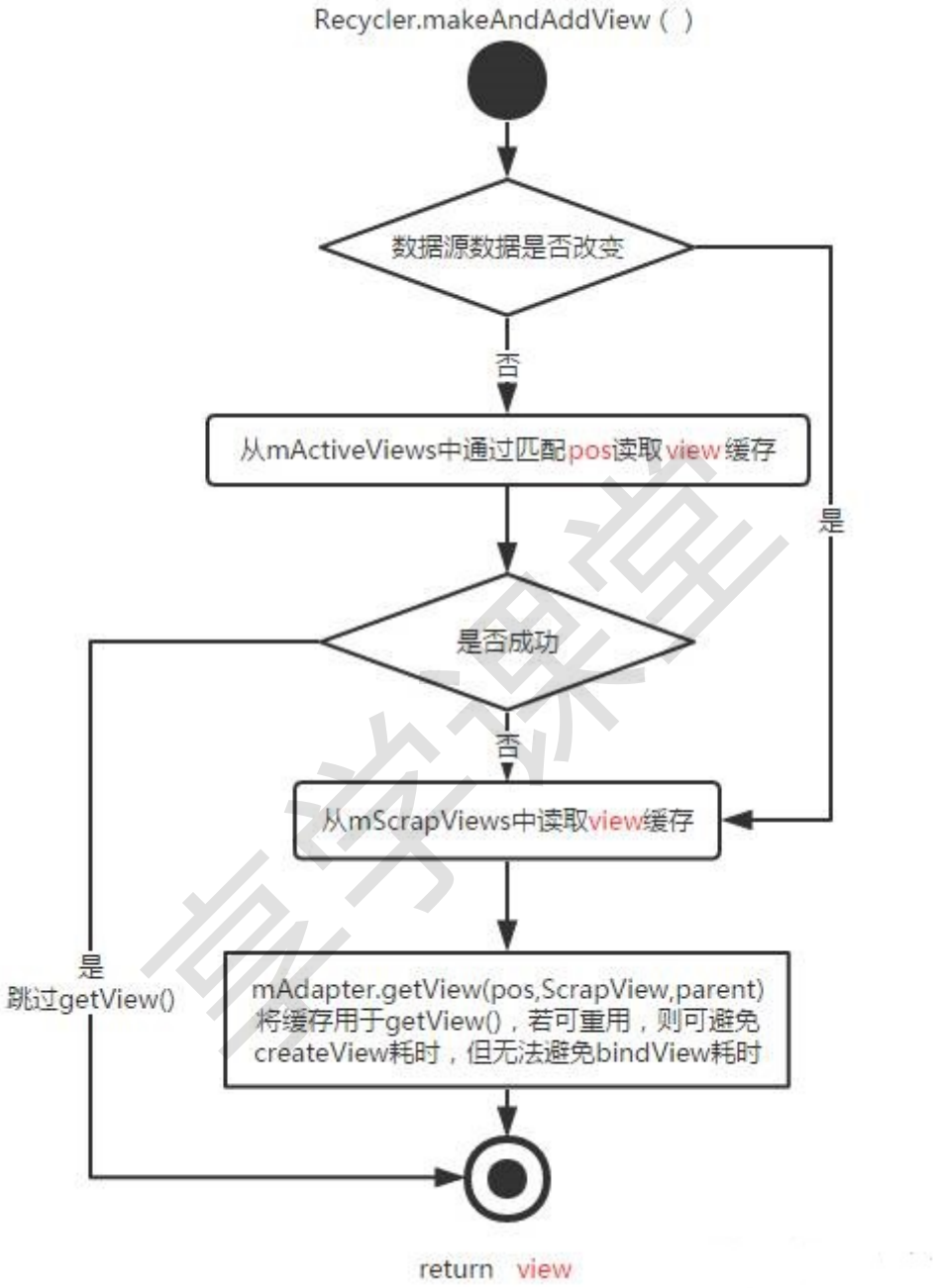
RecyclerView中mCacheViews(屏幕外)获取缓存时，是通过匹配pos获取目标位置的缓存，这样做的好处是，当数据源数据不变的情况下，无须重新bindView：

2). ListView缓存View。而同样是离屏缓存，ListView从mScrapViews根据pos获取相应的缓存，但是并没有直接使用，而是重新getView（即必定会重新bindView）。

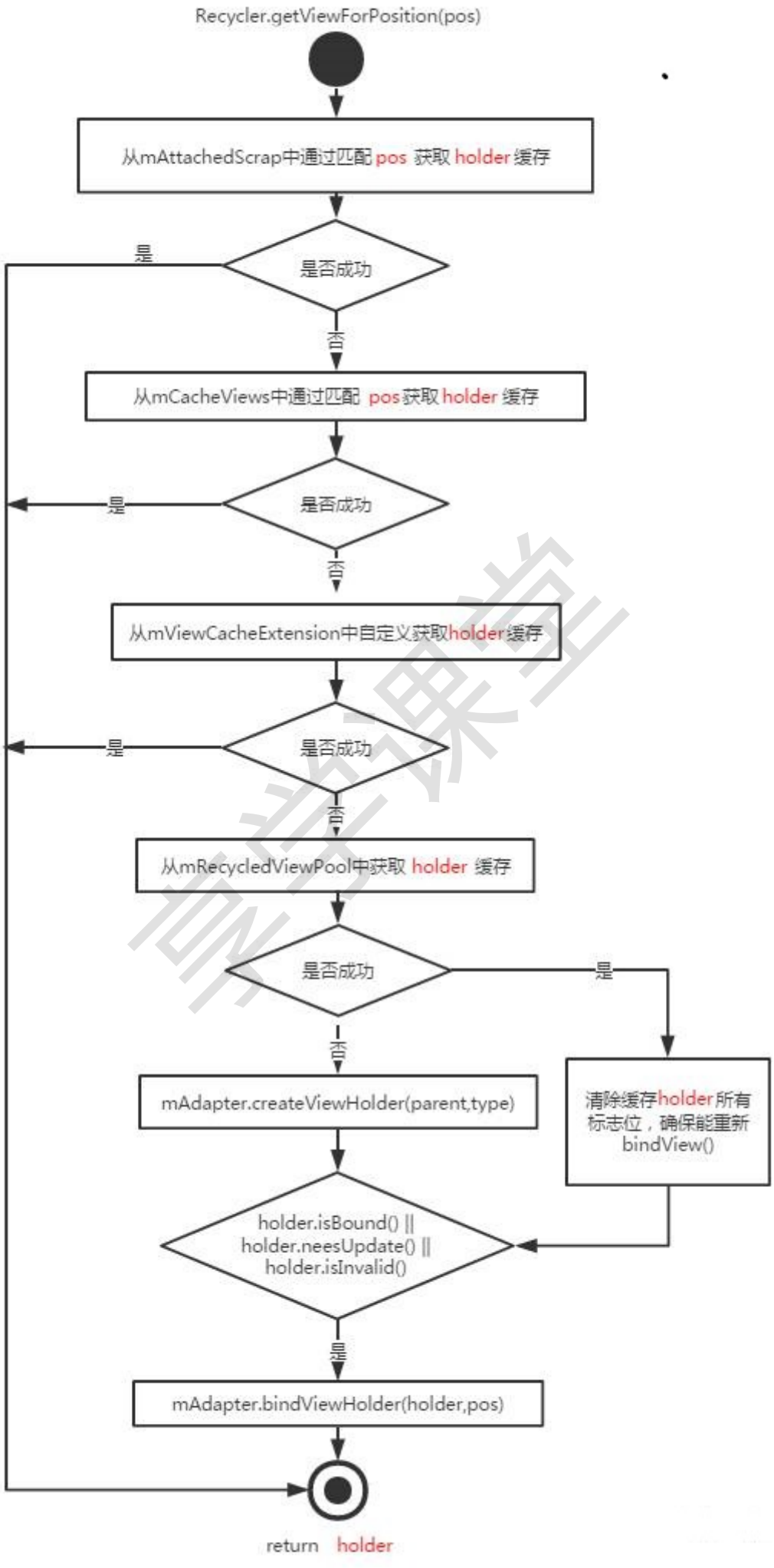
二、 局部刷新

RecyclerView更大的亮点在于提供了局部刷新的接口，通过局部刷新，就能避免调用许多无用的bindView。ListView和RecyclerView最大的区别在于数据源改变时的缓存的处理逻辑，ListView是"一锅端"，将所有的mActiveViews都移入了二级缓存mScrapViews，而RecyclerView则是更加灵活地对每个View修改标志位，区分是否重新bindView。

ListView获取缓存的流程：



RecyclerView获取缓存的流程：


Recycler.getViewForPosition(pos)
从mAttachedScrap中通过匹配 pos 获取 holder 缓存
是否成功
是
否
从mCacheViews中通过匹配 pos 获取 holder 缓存
是否成功
是
否
从mViewCacheExtension中自定义获取holder缓存
是否成功
是
否
从mRecycledViewPool中获取 holder 缓存
是否成功
是
否
mAdapter.createViewHolder(parent,type)
清除缓存holder所有标志位，确保能重新bindView()
holder.isBound() ||
holder.neesUpdate() ||
holder.isInvalid()
是
mAdapter.bindViewHolder(holder,pos)
return holder

结合RecyclerView的缓存机制，看看局部刷新是如何实现的：

以RecyclerView中notifyItemRemoved(1)为例，最终会调用requestLayout()，使整个RecyclerView重新绘制，过程为：

onMeasure()–>onLayout()–>onDraw()

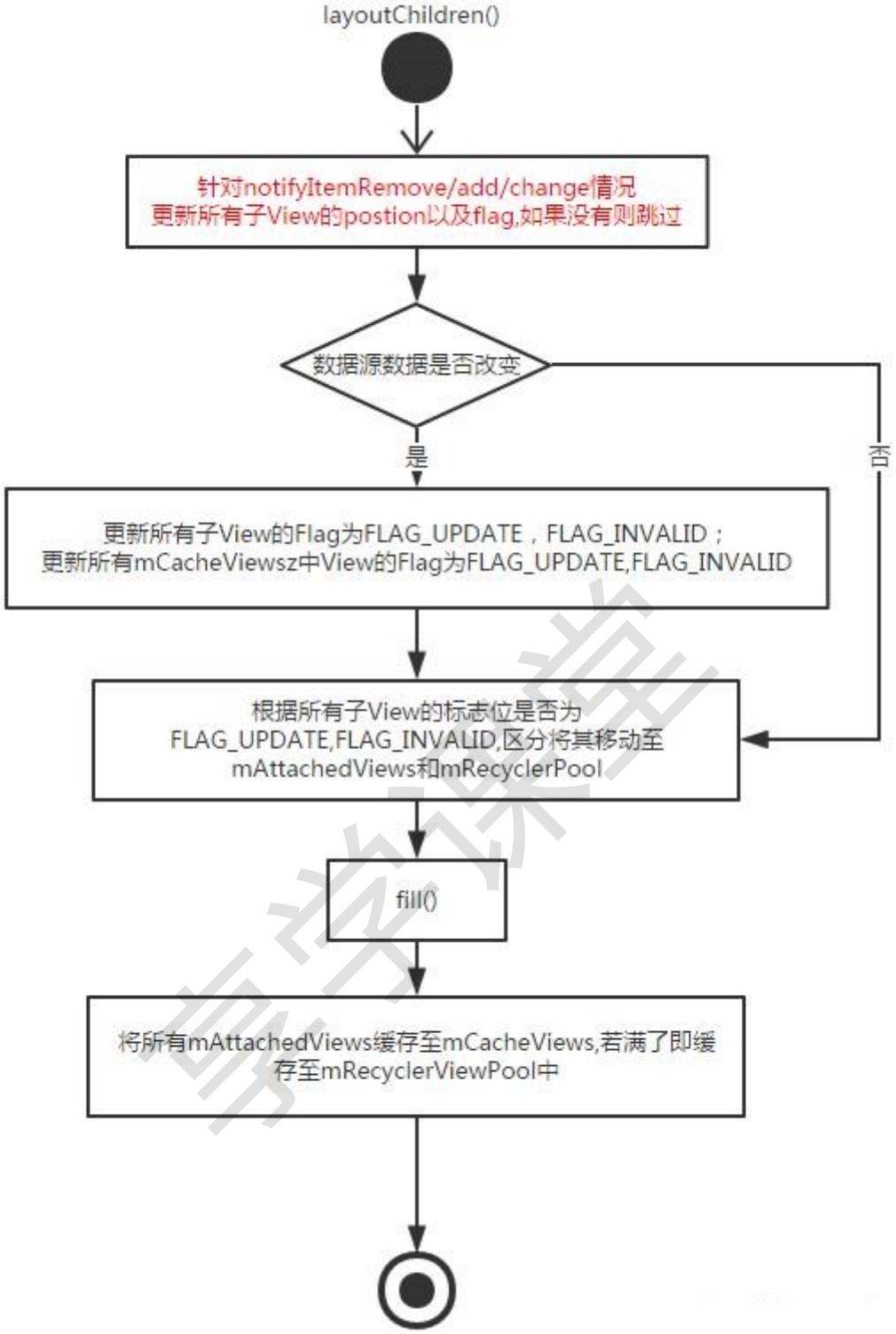
其中，onLayout()为重点，分为三步：

dispathLayoutStep1()：记录RecyclerView刷新前列表项ItemView的各种信息，如Top,Left,Bottom,Right，用于动画的相关计算；

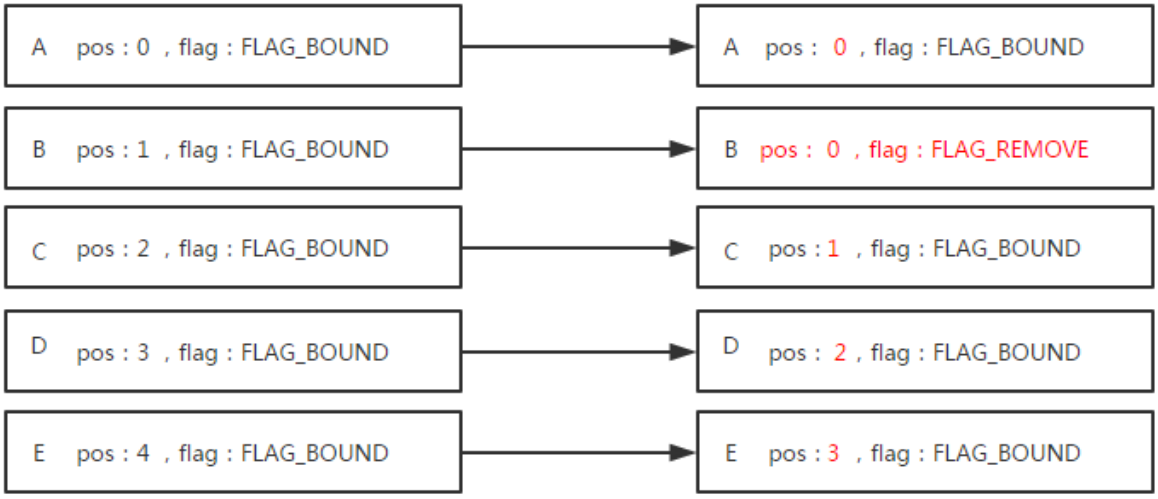
dispathLayoutStep2()：真正测量布局大小，位置，核心函数为layoutChildren()；

dispathLayoutStep3()：计算布局前后各个ItemView的状态，如Remove，Add，Move，Update等，如有必要执行相应的动画.

其中，layoutChildren()流程图：



当调用notifyItemRemoved时，会对屏幕内ItemView做预处理，修改ItemView相应的pos以及flag(流程图中红色部分)：



当调用fill()中RecyclerView.getViewForPosition(pos)时，RecyclerView通过对pos和flag的预处理，使得bindview只调用一次.

7.16 WebView如何做资源缓存？（字节跳动）

这道题想考察什么?

1. 是否了解WebView如何做资源缓存操作与真实场景使用，是否熟悉WebView如何做资源缓存

考察的知识点

1. WebView如何做资源缓存的概念在项目中使用与基本知识

考生应该如何回答

1.WebView如何做资源缓存，你知道吗?

`Android WebView` 自带的缓存机制有2种：

1. 浏览器 缓存机制
2. `Application Cache` 缓存机制

1. **浏览器缓存机制**

   a. 原理
   根据 HTTP 协议头里的 Cache-Control（或 Expires）和 Last-Modified（或 Etag）等字段来控制文件缓存的机制
   下面详细介绍Cache-Control、Expires、Last-Modified & Etag四个字段

   > Cache-Control：用于控制文件在本地缓存有效时长
   > 如服务器回包：`Cache-Control:max-age=600`，则表示文件在本地应该缓存，且有效时长是600秒（从发出请求算起）。在接下来600秒内，如果有请求这个资源，浏览器不会发出HTTP 请求，而是直接使用本地缓存的文件。

2. `Expires`：与`Cache-Control`功能相同，即控制缓存的有效时间

   > 1. `Expires`是`HTTP1.0`标准中的字段，Cache-Control 是`HTTP1.1`标准中新加的字段
   > 2. 当这两个字段同时出现时，`Cache-Control`优先级较高

3. `Last-Modified`：标识文件在服务器上的最新更新时间

> 下次请求时，如果文件缓存过期，浏览器通过 If-Modified-Since 字段带上这个时间，发送给服务器，由服务器比较时间戳来判断文件是否有修改。如果没有修改，服务器返回304告诉浏览器继续使用缓存；如果有修改，则返回200，同时返回最新的文件。

4. `Etag`：功能同`Last-Modified`，即标识文件在服务器上的最新更新时间。

> 1.不同的是，Etag 的取值是一个对文件进行标识的特征字串。
> 2.在向服务器查询文件是否有更新时，浏览器通过If-None-Match 字段把特征字串发送给服务器，由服务器和文件最新特征字串进行匹配，来判断文件是否有更新：没有更新回包304，有更新回包200
> 3.Etag 和 Last-Modified 可根据需求使用一个或两个同时使用。两个同时使用时，只要满足基中一个条件，就认为文件没有更新。

常见用法是：

- `Cache-Control`与`Last-Modified`一起使用；
- `Expires`与`Etag`一起使用；

即一个用于控制缓存有效时间，一个用于在缓存失效后，向服务查询是否有更新

特别注意：浏览器缓存机制 是 浏览器内核的机制，一般都是标准的实现

> 即`Cache-Control`、`Last-Modified`、`Expires`、`Etag`都是标准实现，你不需要操心

b. 特点

- 优点：支持`Http`协议层
- 不足：缓存文件需要首次加载后才会产生；浏览器缓存的存储空间有限，缓存有被清除的可能；缓存的文件没有校验。

> 对于解决以上问题，可以参考手 Q 的离线包

c. 应用场景

静态资源文件的存储，如`JS、CSS`、字体、图片等。

> 1. `Android Webview`会将缓存的文件记录及文件内容会存在当前 app 的 data 目录中。

d. 具体实现

`Android WebView`内置自动实现，即不需要设置即实现

> 1. `Android` 4.4后的`WebView`浏览器版本内核：`Chrome`
> 2. 浏览器缓存机制 是 浏览器内核的机制，一般都是标准的实现

2. Application Cache 缓存机制

a. 原理

- 以文件为单位进行缓存，且文件有一定更新机制（类似于浏览器缓存机制）
- `AppCache`原理有两个关键点：manifest 属性和 manifest 文件。

```
<!DOCTYPE html>
<html manifest="demo_html.appcache">
// HTML 在头中通过 manifest 属性引用 manifest 文件
// manifest 文件：就是上面以 appcache 结尾的文件，是一个普通文件文件，列出了需要缓存的文件
// 浏览器在首次加载 HTML 文件时，会解析 manifest 属性，并读取 manifest 文件，获取
Section：CACHE MANIFEST 下要缓存的文件列表，再对文件缓存
<body>
...
</body>
</html>

// 原理说明如下：
// AppCache 在首次加载生成后，也有更新机制。被缓存的文件如果要更新，需要更新 manifest 文件
// 因为浏览器在下次加载时，除了会默认使用缓存外，还会在后台检查 manifest 文件有没有修改（byte
by byte）
发现有修改，就会重新获取 manifest 文件，对 Section：CACHE MANIFEST 下文件列表检查更新
// manifest 文件与缓存文件的检查更新也遵守浏览器缓存机制
// 如用户手动清了 AppCache 缓存，下次加载时，浏览器会重新生成缓存，也可算是一种缓存的更新
// AppCache 的缓存文件，与浏览器的缓存文件分开存储的，因为 AppCache 在本地有 5MB（分 HOST）
的空间限制
```

b. 特点

方便构建`Web App`的缓存

```
专门为 Web App离线使用而开发的缓存机制
```

c. 应用场景

存储静态文件（如`JS`、`CSS`、字体文件）

> 1. 应用场景 同 浏览器缓存机制
> 2. 但AppCache 是对 浏览器缓存机制 的补充，不是替代。

d. 具体实现

```
        // 通过设置WebView的settings来实现
        WebSettings settings = getSettings();

        String cacheDirPath = context.getFilesDir().getAbsolutePath()+"cache/";
        settings.setAppCachePath(cacheDirPath);
        // 1. 设置缓存路径

         settings.setAppCacheMaxSize(20*1024*1024);
        // 2. 设置缓存大小

        settings.setAppCacheEnabled(true);
        // 3. 开启Application Cache存储机制

// 特别注意
// 每个 Application 只调用一次 WebSettings.setAppCachePath() 和
```

```
WebSettings.setAppCacheMaxSize()
```

7.17 WebView和JS交互的几种方式与拦截方法。（字节跳动）~colin

这道题想考察什么?

1. 是否了解WebView与原生的交互?

考察的知识点

1. Android去调用JS的代码
2. JS去调用Android的代码

考生应该如何回答

1. **交互方式总结**

- 对于Android调用JS代码的方法有2种：
  (1)通过WebView的loadUrl（）
  (2)通过WebView的evaluateJavascript（）

- 对于JS调用Android代码的方法有3种：
  (1)通过WebView的addJavascriptInterface（）进行对象映射
  (2)通过 WebViewClient 的shouldOverrideUrlLoading ()方法回调拦截 url
  (3)通过 WebChromeClient 的onJsAlert()、onJsConfirm()、onJsPrompt()方法回调拦截JS， 对话框alert()、confirm()、prompt()消息

1. **Android通过WebView调用 JS 代码**

方式一：通过`webView`的`loadUrl（）`

实例介绍：点击Android按钮，即调用WebView JS（文本名为`javascript`）中callJS（）

需要加载S代码：javascript.html

```
<!DOCTYPE html>
<html>

   <head>
      <meta charset="utf-8">
      <title>Carson_Ho</title>

// JS代码
     <script>
// Android需要调用的方法
   function callJS(){
      alert("Android调用了JS的callJS方法");
   }
```

```
</script>

   </head>

</html>
```

步骤2：在Android里通过WebView设置调用JS代码

```
public class MainActivity extends AppCompatActivity {

    WebView mWebView;
    Button button;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mWebView =(WebView) findViewById(R.id.webview);

        WebSettings webSettings = mWebView.getSettings();

        // 设置与Js交互的权限
        webSettings.setJavaScriptEnabled(true);
        // 设置允许JS弹窗
        webSettings.setJavaScriptCanOpenWindowsAutomatically(true);

        // 先载入JS代码
        // 格式规定为:file:///android_asset/文件名.html
        mWebView.loadUrl("file:///android_asset/javascript.html");

        button = (Button) findViewById(R.id.button);


        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // 通过Handler发送消息
                mWebView.post(new Runnable() {
                    @Override
                    public void run() {

                        // 注意调用的JS方法名要对应上
                        // 调用javascript的callJS()方法
                        mWebView.loadUrl("javascript:callJS()");
                    }
                });

            }
        });

        // 由于设置了弹窗检验调用结果,所以需要支持js对话框
        // webview只是载体，内容的渲染需要使用webviewChromClient类去实现
        // 通过设置WebChromeClient对象处理JavaScript的对话框
        //设置响应js 的Alert()函数
```

```
        mWebView.setWebChromeClient(new WebChromeClient() {
            @Override
            public boolean onJsAlert(WebView view, String url, String message,
final JsResult result) {
                AlertDialog.Builder b = new
AlertDialog.Builder(MainActivity.this);
                b.setTitle("Alert");
                b.setMessage(message);
                b.setPositiveButton(android.R.string.ok, new
DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        result.confirm();
                    }
                });
                b.setCancelable(false);
                b.create().show();
                return true;
            }

        });


    }
}
```

特别注意：JS代码调用一定要在 `onPageFinished()` 回调之后才能调用，否则不会调用。

方式2：通过WebView的evaluateJavascript()

优点：该方法比第一种方法效率更高、使用更简洁。

```
// 只需要将第一种方法的loadUrl()换成下面该方法即可
    mWebView.evaluateJavascript("javascript:callJS()", new ValueCallback<String>
() {
        @Override
        public void onReceiveValue(String value) {
            //此处为 js 返回的结果
        }
    });
}
```

方法对比：

调用方式	优点	缺点	使用场景
使用loadUrl（）	方便简洁	效率低； 获取返回值麻烦	不需要获取返回值，对性能要求较低时
使用evaluateJavascript（）	效率高	向下兼容性差 （仅Android 4.4以上可用）	Android 4.4以上

3.Android通过WebView调用 JS 代码

方式1：通过 WebView的addJavascriptInterface（）进行对象映射

步骤1：定义一个与JS对象映射关系的Android类：AndroidtoJs

```
// 继承自Object类
public class AndroidtoJs extends Object {

    // 定义JS需要调用的方法
    // 被JS调用的方法必须加入@JavascriptInterface注解
    @JavascriptInterface
    public void hello(String msg) {
        System.out.println("JS调用了Android的hello方法");
    }
}
```

步骤2：将需要调用的JS代码以 .html 格式放到src/main/assets文件夹里

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Carson</title>
      <script>


         function callAndroid(){
        // 由于对象映射，所以调用test对象等于调用Android映射的对象
            test.hello("js调用了android中的hello方法");
         }
      </script>
   </head>
   <body>
      //点击按钮则调用callAndroid函数
      <button type="button" id="button1" onclick="callAndroid()"></button>
   </body>
</html>
```

步骤3：在Android里通过WebView设置Android类与JS代码的映射

```
public class MainActivity extends AppCompatActivity {

    WebView mWebView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mWebView = (WebView) findViewById(R.id.webview);
        WebSettings webSettings = mWebView.getSettings();

        // 设置与Js交互的权限
        webSettings.setJavaScriptEnabled(true);

        // 通过addJavascriptInterface()将Java对象映射到JS对象
        //参数1：Javascript对象名
        //参数2：Java对象名
        mWebView.addJavascriptInterface(new AndroidtoJs(), "test");//AndroidtoJS
类对象映射到js的test对象

        // 加载JS代码
        // 格式规定为:file:///android_asset/文件名.html
        mWebView.loadUrl("file:///android_asset/javascript.html");
```

特点

优点：使用简单

缺点：存在严重的漏洞问题

方式2：通过 WebViewClient 的方法shouldOverrideUrlLoading ()回调拦截 url

- 具体原理：

1. Android通过 `webViewClient` 的回调方法 `shouldOverrideUrlLoading ()` 拦截 url
2. 解析该 url 的协议
3. 如果检测到是预先约定好的协议，就调用相应方法

步骤1：在JS约定所需要的Url协议

```
<!DOCTYPE html>
<html>

   <head>
      <meta charset="utf-8">
      <title>Carson_Ho</title>

     <script>
         function callAndroid(){
            /*约定的url协议为：js://webview?arg1=111&arg2=222*/
            document.location = "js://webview?arg1=111&arg2=222";
```

```
        }
    </script>
</head>

<!-- 点击按钮则调用callAndroid（）方法  -->
   <body>
     <button type="button" id="button1" onclick="callAndroid()">点击调用Android代码
</button>
   </body>
</html>
```

步骤2：在Android通过WebViewClient复写`shouldOverrideUrlLoading （）`

```
public class MainActivity extends AppCompatActivity {

    WebView mWebView;
//    Button button;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mWebView = (WebView) findViewById(R.id.webview);

        WebSettings webSettings = mWebView.getSettings();

        // 设置与Js交互的权限
        webSettings.setJavaScriptEnabled(true);
        // 设置允许JS弹窗
        webSettings.setJavaScriptCanOpenWindowsAutomatically(true);

        // 步骤1：加载JS代码
        // 格式规定为:file:///android_asset/文件名.html
        mWebView.loadUrl("file:///android_asset/javascript.html");


// 复写WebViewClient类的shouldOverrideUrlLoading方法
mWebView.setWebViewClient(new WebViewClient() {
                                    @Override
                                    public boolean
shouldOverrideUrlLoading(WebView view, String url) {

                                        // 步骤2：根据协议的参数，判断是否是所需要的
url
                                        // 一般根据scheme（协议格式） &
authority（协议名）判断（前两个参数）
                                        //假定传入进来的 url = "js://webview?
arg1=111&arg2=222"（同时也是约定好的需要拦截的）

                                        Uri uri = Uri.parse(url);

                                        // 如果url的协议 = 预先约定的 js 协议
                                        // 就解析往下解析参数
                                        if ( uri.getScheme().equals("js")) {
```

```
                                                    // 如果 authority  = 预先约定协议里的
webview，即代表都符合约定的协议
                                                    // 所以拦截url,下面JS开始调用Android需
要的方法
                                                    if
(uri.getAuthority().equals("webview")) {


                                                       //  步骤3：
                                                        // 执行JS所需要调用的逻辑
                                                        System.out.println("js调用了
Android的方法");
                                                        // 可以在协议上带有参数并传递到
Android上
                                                        HashMap<String, String> params
= new HashMap<>();
                                                        Set<String> collection =
uri.getQueryParameterNames();


                                                    }

                                                    return true;
                                                }
                                                return
super.shouldOverrideUrlLoading(view, url);
                                            }
                                        }
        );
    }
        }
```

特点

优点：不存在方式1的漏洞；

缺点：JS获取Android方法的返回值复杂。

方式3：通过 WebChromeClient 的onJsAlert()、onJsConfirm()、onJsPrompt()方法回调拦截JS，对话框alert()、confirm()、prompt()消息

下面的例子将用**拦截 JS的输入框（即`prompt（）`方法）**说明：

1. 常用的拦截是：拦截 JS的输入框（即`prompt（）`方法）
2. 因为只有`prompt（）`可以返回任意类型的值，操作最全面方便、更加灵活；而alert（）对话框没有返回值；confirm（）对话框只能返回两种状态（确定 / 取消）两个值

步骤1：加载JS代码，如下：

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Carson_Ho</title>

     <script>

    function clickprompt(){
    // 调用prompt（）
    var result=prompt("js://demo?arg1=111&arg2=222");
    alert("demo " + result);
}

      </script>
</head>

<!-- 点击按钮则调用clickprompt()  -->
   <body>
     <button type="button" id="button1" onclick="clickprompt()">点击调用Android代码
</button>
   </body>
</html>
```

当使用`mWebView.loadUrl("file:///android_asset/javascript.html")`加载了上述JS代码后，就会触发回调`onJsPrompt（）`，具体如下：

1. 如果是拦截警告框（即`alert()`），则触发回调`onJsAlert（）`；
2. 如果是拦截确认框（即`confirm()`），则触发回调`onJsConfirm（）`；

步骤2：在Android通过`WebChromeClient`复写`onJsPrompt（）`

```
public class MainActivity extends AppCompatActivity {

    WebView mWebView;
//    Button button;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mWebView = (WebView) findViewById(R.id.webview);

        WebSettings webSettings = mWebView.getSettings();

        // 设置与Js交互的权限
        webSettings.setJavaScriptEnabled(true);
        // 设置允许JS弹窗
        webSettings.setJavaScriptCanOpenWindowsAutomatically(true);
```

```
// 先加载JS代码
        // 格式规定为:file:///android_asset/文件名.html
        mWebView.loadUrl("file:///android_asset/javascript.html");


        mWebView.setWebChromeClient(new WebChromeClient() {
                                        // 拦截输入框(原理同方式2)
                                        // 参数message:代表promt（）的内容（不是url）
                                        // 参数result:代表输入框的返回值
                                        @Override
                                        public boolean onJsPrompt(WebView view,
String url, String message, String defaultValue, JsPromptResult result) {
                                            // 根据协议的参数，判断是否是所需要的url(原
理同方式2)
                                            // 一般根据scheme（协议格式） &
authority（协议名）判断（前两个参数）
                                            //假定传入进来的 url = "js://webview?
arg1=111&arg2=222"（同时也是约定好的需要拦截的）

                                            Uri uri = Uri.parse(message);
                                            // 如果url的协议 = 预先约定的 js 协议
                                            // 就解析往下解析参数
                                            if ( uri.getScheme().equals("js")) {

                                                // 如果 authority  = 预先约定协议里
的 webview，即代表都符合约定的协议
                                                // 所以拦截url,下面JS开始调用Android
需要的方法
                                                if
(uri.getAuthority().equals("webview")) {

                                                    //
                                                    // 执行JS所需要调用的逻辑
                                                    System.out.println("js调用了
Android的方法");
                                                    // 可以在协议上带有参数并传递到
Android上
                                                    HashMap<String, String>
params = new HashMap<>();
                                                    Set<String> collection =
uri.getQueryParameterNames();

                                                    //参数result:代表消息框的返回值
(输入值)
                                                    result.confirm("js调用了
Android的方法成功啦");
                                                }
                                                return true;
                                            }
                                            return super.onJsPrompt(view, url,
message, defaultValue, result);
                                        }

// 通过alert()和confirm()拦截的原理相同，此处不作过多讲述

                                        // 拦截JS的警告框
                                        @Override
```

```
                                    public boolean onJsAlert(WebView view,
String url, String message, JsResult result) {
                                        return super.onJsAlert(view, url,
message, result);
                                    }

                                    // 拦截JS的确认框
                                    @Override
                                    public boolean onJsConfirm(WebView view,
String url, String message, JsResult result) {
                                        return super.onJsConfirm(view, url,
message, result);
                                    }
                                }
        );


            }

        }
```

3.总结

类型	调用方式	优点	缺点	使用场景	使用建议
Android 调用 JS	loadUrl（）	方便简洁	效率低、获取返回值麻烦	不需要获取返回值，对性能要求较低时	混合使用，即： Android 4.4以下 用方法1 Android 4.4以上 用方法2
	evaluateJavascript（）	效率高	向下兼容性差 (仅Android 4.4以上可用)	Android 4.4以上	
JS 调用 Android	通过addJavascriptInterface（）进行添加对象映射	方便简洁	Android 4.2以下存在漏洞问题	Android 4.2以上相对简单互调场景	/
	通过 WebViewClient.shouldOverrideUrlLoading ()回调拦截 url	不存在漏洞问题	使用复杂：需要进行协议的约束；从 Native 层往 Web 层传递值比较繁琐	不需要返回值情况下的互调场景 (iOS主要使用该方式)	
	通过 WebChromeClient的onJsAlert()、onJsConfirm()、onJsPrompt（）方法回调拦截JS对话框消息	不存在漏洞问题	使用复杂：需要进行协议的约束；	能满足大多数情况下的互调场景	

7.18 自定义view与viewgroup的区别 ~derry

这道题想考察什么?

1. 是否了解自定义view与viewgroup的区别与真实场景使用，是否熟悉自定义view与viewgroup的区别在工作中的表现是什么?

考察的知识点

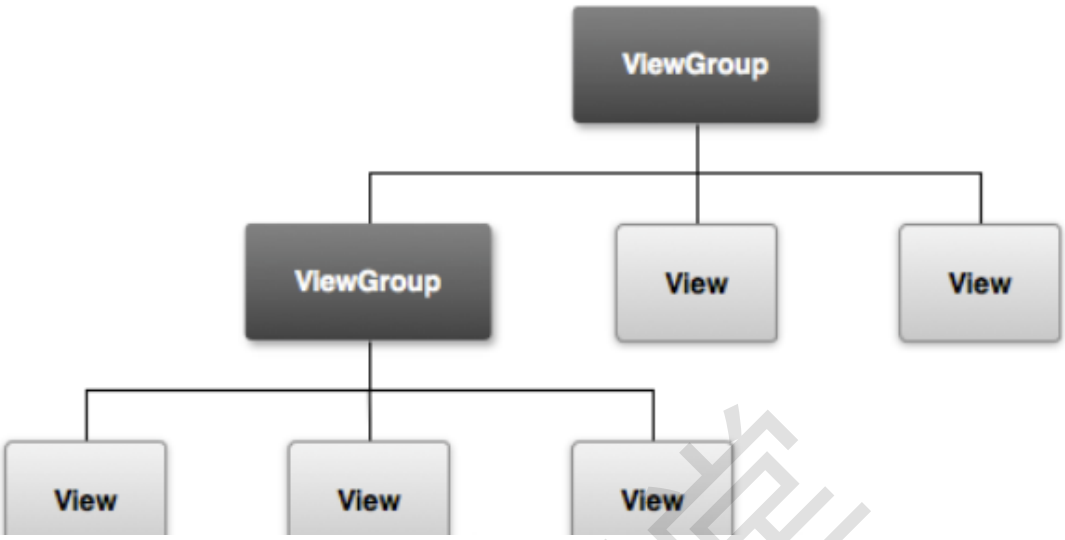
1. 自定义view与viewgroup的区别的概念在项目中使用与基本知识

考生应该如何回答

1.说说自定义view与viewgroup的区别?

答:

Android的UI界面都是由View和ViewGroup及其派生类组合而成的。其中，View是所有UI组件的基类，而ViewGroup是容纳View及其派生类的容器，ViewGroup也是从View派生出来的。一般来说，开发UI界面都不会直接使用View和ViewGroup（自定义控件的时候使用），而是使用其派生类。



View和ViewGroup的区别:

可以从两方面来说:

```
一.事件分发方面的区别;

二.UI绘制方面的区别;
```

事件分发方面的区别:

事件分发机制主要有三个方法：dispatchTouchEvent()、onInterceptTouchEvent()、onTouchEvent()

1.ViewGroup包含这三个方法，而View则只包含dispatchTouchEvent()、onTouchEvent()两个方法，不包含onInterceptTouchEvent()。

2.触摸事件由Action_Down、Action_Move、Action_Up组成，一次完整的触摸事件，包含一个Down和Up，以及若干个Move（可以为0）；

3.在Action_Down的情况下，事件会先传递到最顶层的ViewGroup，调用ViewGroup的dispatchTouchEvent()，①如果ViewGroup的onInterceptTouchEvent()返回false不拦截该事件，则会分发给子View，调用子View的dispatchTouchEvent()，如果子View的dispatchTouchEvent()返回true，则调用View的onTouchEvent()消费事件。②如果ViewGroup的onInterceptTouchEvent()返回true拦截该事件，则调用ViewGroup的onTouchEvent()消费事件，接下来的Move和Up事件将由该ViewGroup直接进行处理。

4.当某个子View的dispatchTouchEvent()返回true时，会中止Down事件的分发，同时在ViewGroup中记录该子View。接下来的Move和Up事件将由该子View直接进行处理。

5.当ViewGroup中所有子View都不捕获Down事件时，将触发ViewGroup自身的onTouch();触发的方式是调用super.dispatchTouchEvent函数，即父类View的dispatchTouchEvent方法。在所有子View都不处理的情况下，触发Acitivity的onTouchEvent方法。

6..由于子View是保存在ViewGroup中的，多层ViewGroup的节点结构时，上层ViewGroup保存的会是真实处理事件的View所在的ViewGroup对象。如ViewGroup0--ViewGroup1--TextView的结构中，TextView返回了true，它将被保存在ViewGroup1中，而ViewGroup1也会返回true，将被保存在ViewGroup0中；当Move和Up事件来时，会先从ViewGroup0传递到ViewGroup1，再由ViewGroup1传递到TextView，最后事件由TextView消费掉。

7.子View可以调getParent().requestDisallowInterceptTouchEvent(),请求父ViewGroup不拦截事件。

UI绘制方面的区别：

UI绘制主要有五个方法：onDraw(),onLayout(),onMeasure()，dispatchDraw(),drawChild()

1.ViewGroup包含这五个方法，而View只包含onDraw(),onLayout(),onMeasure()三个方法，不包含dispatchDraw(),drawChild()。

2.绘制流程：onMeasure（测量）--》onLayout（布局）--》onDraw（绘制）。

3.绘制按照视图树的顺序执行，视图绘制时会先绘制子控件。如果视图的背景可见，视图会在调用onDraw()之前调用drawBackGround()绘制背景。强制重绘，可以使用invalidate();

4.如果发生视图的尺寸变化，则该视图会调用requestLayou()，向父控件请求再次布局。如果发生视图的外观变化，则该视图会调用invalidate()，强制重绘。如果requestLayout()或invalidate()有一个被调用，框架会对视图树进行相关的测量、布局和绘制。

注意：视图树是单线程操作，直接调用其它视图的方法必须要在UI线程里。跨线程的操作必须使用Handler。

5.onLayout()：对于View来说，onLayout()只是一个空实现；而对于ViewGroup来说，onLayout()使用了关键字abstract的修饰，要求其子类必须重载该方法，目的就是安排其children在父视图的具体位置。

6.draw过程：drawBackground()绘制背景--》onDraw()对View的内容进行绘制--》dispatchDraw()对当前View的所有子View进行绘制--》onDrawScrollBars()对View的滚动条进行绘制。

7.19 View的绘制原理~colin

这道题想考察什么?

1. 是否了解View绘制原理的知识?

考察的知识点

1. View的Framework相关知识
2. View的measure、layout、draw

考生应该如何回答

注意： 本文中涉及ActivityThread、WindowManagerImpl、WindowManagerGlobal、ViewRootImpl知识，如果对上述概念不熟悉的同学，先学习对应章节。

1.View的Framework相关知识

先简单说下View的起源，有助于我们后续的分析理解。

1.1 从**ActivityThread.java**开始

下面只贴出源码中的关键代码，重点是**vm.addView(decor, l)**这句，那么有三个对象需要理解：

vm：a.getWindowManager()即通过Activity.java的getWindowManager方法得到的对象；继续跟踪源码，发现此对象由Window.java的getWindowManager方法获得，即是((WindowManagerImpl)wm).createLocalWindowManager(this)。

decor：通过r.activity.getWindow()可知r.window是PhoneWindow对象，在PhoneWindow中找到getDecorView方法，得知decor即DecorView对象。

l：通过下面源码得知，l通过r.window.getAttributes获取到，由于PhoneWindow中没有getAttributes方法，故从他的父类Window中获取，得知宽高均为LayoutParams.MATCH_PARENT

```
ActivityThread.java

    public void handleResumeActivity(IBinder token, boolean finalStateRequest,
boolean isForward, String reason)  {
            ...
            r.window = r.activity.getWindow();
            View decor = r.window.getDecorView();
            decor.setVisibility(View.INVISIBLE);
            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) {
                if (!a.mWindowAdded) {
                    a.mWindowAdded = true;
```

```
                    wm.addView(decor, l);
                } else {
                    ...
                }
            }
     }
```

1.2 接上面wm.addView(decor, l)，通过上面分析wm是**WindowManagerImpl**，则走进此类的addView中

```
  @Override
    public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams
params) {
        applyDefaultToken(params);
        mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);
    }
```

1.3 **WindowManagerGlobal**

```
public void addView(View view, ViewGroup.LayoutParams params,
        Display display, Window parentWindow) {
        ...
        ViewRootImpl root;
        ...
        root = new ViewRootImpl(view.getContext(), display);
        ...
        try {
            root.setView(view, wparams, panelParentView);
        } catch (RuntimeException e) {
            ...
        }
    }
}
```

1.4 有第三步可知，最终走到了**ViewRootImpl.java**，曙光在前方

```
 public void setView(View view, WindowManager.LayoutParams attrs, View
panelParentView) {
                ...
                requestLayout();
                ...
    }
```

```
 public void requestLayout() {
        if (!mHandlingLayoutInLayoutRequest) {
            ...
            scheduleTraversals();
        }
    }
```

```
 void scheduleTraversals() {
      if (!mTraversalScheduled) {
          ...
          mChoreographer.postCallback(
                  Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
          ...
      }
   }
```

```
final class TraversalRunnable implements Runnable {
      @Override
      public void run() {
          doTraversal();
      }
   }
```

```
   void doTraversal() {
          ...
          performTraversals();
          ...
   }
```

重点来了，performMeasure、performLayout、performDraw则是对根view的测量、布局、draw；先说下mWidth、mHeight和lp，其中mWidth、mHeight是window的宽高，lp就是前面提到的WindowManager中的LayoutParam，跟踪代码发现均为LayoutParams.MATCH_PARENT

```
private void performTraversals() {
          ...
          WindowManager.LayoutParams lp = mWindowAttributes;
          ...
          int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);
          int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);
          ...
          performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
          ...
          performLayout(lp, mWidth, mHeight);
          ...
          performDraw();
          ...
   }
```

根据以下源码，那么childWidthMeasureSpec、childHeightMeasureSpec的值就是MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.EXACTLY);

```
/**
     * Figures out the measure spec for the root view in a window based on it's
     * layout params.
     *
     * @param windowSize
     *            The available width or height of the window
     *
     * @param rootDimension
     *            The layout params for one dimension (width or height) of the
     *            window.
```

```
     *
     * @return The measure spec to use to measure the root view.
     */
    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;
        ..
        }
        return measureSpec;
    }
```

下面的mView上面已分析过，就是DecorView，也就是根view

```
  private void performMeasure(int childWidthMeasureSpec, int
childHeightMeasureSpec) {
        ...
        try {
            mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);
        } finally {
            ...
        }
    }
```

2.绘制流程

2.1 measure

接上面说到了performMeasure，即走到了DecorView的measure，而DecorView实际是FrameLayout，FrameLayout的父类是ViewGroup，而ViewGroup的父类是View，所以直接走 到了View的measure里面。

主角登场，接下来分析View的measure，measure是final的，也就是不能不能重写此方法，但是里面有一个onMeasure方法，到这里应该很熟悉了，我们自定义控件都会重写这个方法；

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

由于DecorView的父类的FrameLayout，那么我们来看FrameLayout的onMeasure方法；可以看到会测量所有的子View，最后测量自己。所以有两点：测量子View；测量自己。

```
 protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        int count = getChildCount();
        ...
        for (int i = 0; i < count; i++) {
```

```
            final View child = getChildAt(i);
            ...
            if (mMeasureAllChildren || child.getVisibility() != GONE) {
                measureChildWithMargins(child, widthMeasureSpec, 0,
heightMeasureSpec, 0);
                ...
            }
        }
        ...
        setMeasuredDimension(resolveSizeAndState(maxWidth, widthMeasureSpec,
childState),
                                                resolveSizeAndState(maxHeight,
heightMeasureSpec,
                        childState << MEASURED_HEIGHT_STATE_SHIFT));
        ...
        }
    }
```

我们先来关注测量子View即measureChildWithMargins，大概的意思是：
父View的MeasureSpec+当前View的LayoutParams=得出当前View的MeasureSpec
parentWidthMeasureSpec、parentHeightMeasureSpec还记得是什么吗？就是之前说过的ViewRootImpl中的performTraversals方法里面的performMeasure(**childWidthMeasureSpec, childHeightMeasureSpec**)传过去的，可以回顾下前面的分析；

```
 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);
    }
```

在上面代码中先获取到LayoutParams，然后通过getChildMeasureSpec计算出自定义view的宽高，里面涉及父view的padding与子view的margin，接着调用子View的measure方法传入计算出的宽高MeasureSpec，层层递归直到无子View为止。

```
  public static int getChildMeasureSpec(int spec, int padding, int
childDimension) {
        int specMode = MeasureSpec.getMode(spec);
        int specSize = MeasureSpec.getSize(spec);

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

        int resultSize = 0;
```

```
        int resultMode = 0;

        switch (specMode) {
        // Parent has imposed an exact size on us
        case MeasureSpec.EXACTLY:
            if (childDimension >= 0) {
                resultSize = childDimension;
                resultMode = MeasureSpec.EXACTLY;
            } else if (childDimension == LayoutParams.MATCH_PARENT) {
                // Child wants to be our size. So be it.
                resultSize = size;
                resultMode = MeasureSpec.EXACTLY;
            } else if (childDimension == LayoutParams.WRAP_CONTENT) {
                // Child wants to determine its own size. It can't be
                // bigger than us.
                resultSize = size;
                resultMode = MeasureSpec.AT_MOST;
            }
            break;

        // Parent has imposed a maximum size on us
        case MeasureSpec.AT_MOST:
            if (childDimension >= 0) {
                // Child wants a specific size... so be it
                resultSize = childDimension;
                resultMode = MeasureSpec.EXACTLY;
            } else if (childDimension == LayoutParams.MATCH_PARENT) {
                // Child wants to be our size, but our size is not fixed.
                // Constrain child to not be bigger than us.
                resultSize = size;
                resultMode = MeasureSpec.AT_MOST;
            } else if (childDimension == LayoutParams.WRAP_CONTENT) {
                // Child wants to determine its own size. It can't be
                // bigger than us.
                resultSize = size;
                resultMode = MeasureSpec.AT_MOST;
            }
            break;

        // Parent asked to see how big we want to be
        case MeasureSpec.UNSPECIFIED:
            if (childDimension >= 0) {
                // Child wants a specific size... let him have it
                resultSize = childDimension;
                resultMode = MeasureSpec.EXACTLY;
            } else if (childDimension == LayoutParams.MATCH_PARENT) {
                // Child wants to be our size... find out how big it should
                // be
                resultSize = View.sUseZeroUnspecifiedMeasureSpec ? 0 : size;
                resultMode = MeasureSpec.UNSPECIFIED;
            } else if (childDimension == LayoutParams.WRAP_CONTENT) {
                // Child wants to determine its own size.... find out how
                // big it should be
                resultSize = View.sUseZeroUnspecifiedMeasureSpec ? 0 : size;
                resultMode = MeasureSpec.UNSPECIFIED;
            }
            break;
        }
```

```
        //noinspection ResourceType
        return MeasureSpec.makeMeasureSpec(resultSize, resultMode);
    }
```

分析完测量子View，接下来看测量自己，即上面源码中提到的setMeasuredDimension

```
 protected final void setMeasuredDimension(int measuredWidth, int
measuredHeight) {
        ...
        setMeasuredDimensionRaw(measuredWidth, measuredHeight);
    }
```

```
  private void setMeasuredDimensionRaw(int measuredWidth, int measuredHeight) {
        mMeasuredWidth = measuredWidth;
        mMeasuredHeight = measuredHeight;
        ...
    }
```

最后为mMeasureWidth、mMeasureSpec赋值，测量完毕。

我们来看下View类中的onMeasure（即若不覆写onMeasure的默认逻辑）,同上调用了setMeasureSpec为测量结果赋值；这里有需要**注意**的地方，当我们自定义View覆写onMeasure时，最后一定要为测量结果赋值(setMeasuredDimension)，否则会报错。

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(),
widthMeasureSpec),
                getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
    }
```

看下getSuggestedMinimumWidth，mMinWidth就是我们在layout中设置的android:minWidth，也就是有background的话则取background的宽度，否则去minWidth；高度也一样。

```
    protected int getSuggestedMinimumWidth() {
        return (mBackground == null) ? mMinWidth : max(mMinWidth,
mBackground.getMinimumWidth());
    }
```

getDefaultSize获取结果，首先通过之前算好的MeasureSpec取出mode、size，若mod是AT_MOST、WXACTLY则结果为specSize，若是UNSPECIFIED则结果为getSuggestedMinimumWidth的默认值。

```
  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;
    }
```

2.2 layout

上面说过的view的入口是ViewRootImpl

```
 private void performTraversals() {
            ...
            WindowManager.LayoutParams lp = mWindowAttributes;
            ...
            int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);
            int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);
            ...
            performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
            ...
            performLayout(lp, mWidth, mHeight);
            ...
            performDraw();
            ...
    }
```

performMeasureSpec说完了，接下来看performLayout，measure负责测量、layout负责位置摆放，performLayout里面调用了View.java的layout

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

        final View host = mView;
        if (host == null) {
            return;
        }
        ...
        try {
            host.layout(0, 0, host.getMeasuredWidth(),
host.getMeasuredHeight());
            ...
            }
        } finally {
            ...
        }
        ...
    }
```

这里根据传入的left、top、right、bottom值来确定摆放位置，right和bottom一般分别为left、top的值加上当前view的宽、高。

```
public void layout(int l, int t, int r, int b) {
        ...
        int oldL = mLeft;
```

```
        int oldT = mTop;
        int oldB = mBottom;
        int oldR = mRight;

        boolean changed = isLayoutModeOptical(mParent) ?
                setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);

        if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) ==
PFLAG_LAYOUT_REQUIRED) {
            onLayout(changed, l, t, r, b);
            ...
        }
        ...
    }
```

通过setFrame来设置上下左右四个值，setOpticalFrame最终也会走到setFrame，这里就不贴源码了

```
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;
    }
```

我们发现view.java的onLayout是空方法，但是在viewgroup中是抽象方法，也就是父类均需要重写该方法，说明该方法是用来摆放子view位置的。

```
View.java
    protected void onLayout(boolean changed, int left, int top, int right, int
bottom) {
    }
```

```
ViewGroup.java
    protected abstract void onLayout(boolean changed,
            int l, int t, int r, int b);
```

2.3 draw

上面说过的view的入口是ViewRootImpl

```
  private void performTraversals() {
            ...
            WindowManager.LayoutParams lp = mWindowAttributes;
            ...
            int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);
            int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);
            ...
            performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
            ...
            performLayout(lp, mWidth, mHeight);
            ...
            performDraw();
            ...
    }
```

performMeasureSpec、performLayout说完了，measure负责测量、layout负责位置摆放，draw负责绘制。下面只贴出了关键代码，最后走到了mView的draw里面

```
private void performDraw() {
        ...
        try {
            boolean canUseAsync = draw(fullRedrawNeeded);
            ...
        }
        ...
    }
```

```
 private boolean draw(boolean fullRedrawNeeded) {
         ...
                if (!drawSoftware(surface, mAttachInfo, xOffset, yOffset,
                        scalingRequired, dirty, surfaceInsets)) {
                    return false;
                }
        ...
        return useAsyncReport;
    }
```

```
 private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int xoff,
int yoff,
            boolean scalingRequired, Rect dirty, Rect surfaceInsets) {
            ...
                mView.draw(canvas);
            ...
        return true;
    }
```

View中的draw方法注释和清晰，分为6步，其中2、5可以跳过：
1、绘制背景
2、跳过
3、绘制view内容
4、绘制子view
5、跳过
6、绘制

```
View.java
    public void draw(Canvas canvas) {
        ...

        /*
         * Draw traversal performs several drawing steps which must be executed
         * in the appropriate order:
         *
         *      1. Draw the background
         *      2. If necessary, save the canvas' layers to prepare for fading
         *      3. Draw view's content
         *      4. Draw children
         *      5. If necessary, draw the fading edges and restore layers
         *      6. Draw decorations (scrollbars for instance)
         */

        // Step 1, draw the background, if needed
        int saveCount;

        if (!dirtyOpaque) {
            drawBackground(canvas);
        }

        // skip step 2 & 5 if possible (common case)
        final int viewFlags = mViewFlags;
        boolean horizontalEdges = (viewFlags & FADING_EDGE_HORIZONTAL) != 0;
        boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;
        if (!verticalEdges && !horizontalEdges) {
            // Step 3, draw the content
            if (!dirtyOpaque) onDraw(canvas);

            // Step 4, draw the children
            dispatchDraw(canvas);

            drawAutofilledHighlight(canvas);

            // Overlay is part of the content and draws beneath Foreground
            if (mOverlay != null && !mOverlay.isEmpty()) {
                mOverlay.getOverlayView().dispatchDraw(canvas);
            }

            // Step 6, draw decorations (foreground, scrollbars)
            onDrawForeground(canvas);

            // Step 7, draw the default focus highlight
            drawDefaultFocusHighlight(canvas);
```

```
            if (debugDraw()) {
                debugDrawFocus(canvas);
            }

            // we're done...
            return;
        }
    }
```

7.20 View的滑动方式

这道题想考察什么?

1. 是否了解View的滑动方式与真实场景使用，是否熟悉View的滑动方式

考察的知识点

1. View的滑动方式的概念在项目中使用与基本知识

考生应该如何回答

1.View的滑动方式，你知道吗?

1、使用scrollTo/scrollBy

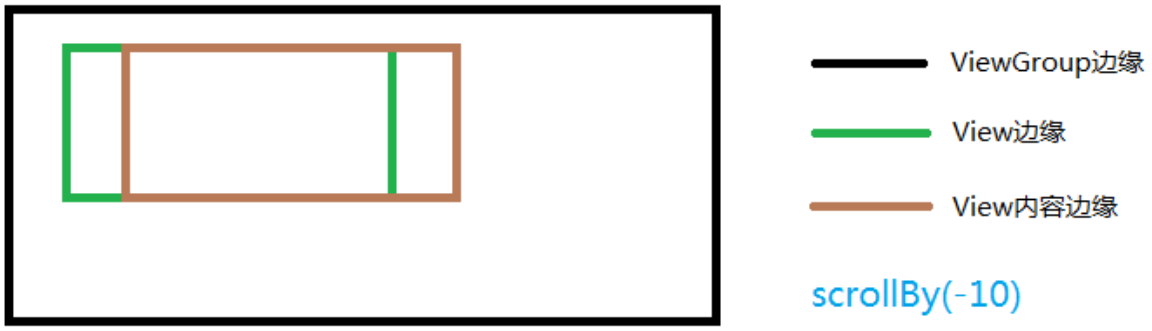
为了实现View滑动，Android专门提供了这两个方法让我们使用。这两个函数的区别是scrollBy提供的是基于当前位置的相对滑动，而scrollTo提供的是基于起始位置的绝对滑动。

需要注意的是实际的滑动方向与我们想当然的方向不同，这个问题与View的内部变量mScrollX和mScrollY的含义有关，scrollTo函数与scrollBy函数实际上就是对这两个变量做出修改。

mScrollX：View的左边缘坐标减去View内容的左边缘坐标。

mScrollY：View的上边缘坐标减去View内容的上边缘坐标。

另外一个需要注意的地方是超出View边缘范围的内容是不会显示的。



2、使用动画

动画的方式主要是操作View的translationX与translationY属性。可以使用XML文件自定义动画效果也可以使用属性动画实现。

2.1、自定义动画

```xml
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true">
    <translate
        android:duration="100"
        android:fromXDelta="0"
        android:fromYDelta="0"
        android:toXDelta="100"
        android:toYDelta="0"
        android:interpolator="@android:anim/linear_interpolator"/>
</set>
```

```
View view = findViewById(R.id.target_view);
Animation animation = AnimationUtils.loadAnimation(context,
R.anim.target_animation);
view.startAnimation(animation);
```

注意在XML文件中有一个fillAfter属性，如果设置为false的话当动画结束时View会恢复到原来的位置。

2.2、属性动画

```
View view = findViewById(R.id.target_view);
ObjectAnimator.ofFloat(view, "translationX", 0, 100)
    .setDuration(100)
    .start();
```

3、改变布局参数

第三种方法是改变View的LayoutParams，与之前的方法相比，这种方法显得不是很正统，但是也可以实现我们的需求。举个例子，假如我们想把一个View右移100dp，那么最简单的方式就是把它LayoutParams里的marginLeft参数的值增加100dp。但这种方法要根据View所在的父布局灵活调整，在一些情况下改变margin值并不能改变View的位置。

```
View view = findViewById(R.id.target_view);
ViewGroup.MarginLayoutParams params = (ViewGroup.MarginLayoutParams) view
.getLayoutParams();
params.leftMargin += 10;
mTargetTextView.setLayoutParams(params);
```

还有一个相似的方法，但可用性要好一些，那就是调用View的layout方法，直接修改View相对于父布局的位置。

```
int offsetX = 10;
View view = findViewById(R.id.target_view);
view.layout(mTargetTextView.getLeft() + offsetX,
            view.getTop(),
            view.getRight() + offsetX,
            view.getBottom());
```

4、使用Scroller实现渐进式滑动

上边提到的滑动方式有一个共同的缺点那是他们都不是渐进式的滑动。实现渐进式滑动的共同思想就是将一个滑动行为分解为若干个，在一段时间内按顺序完成。

这里介绍使用Scroller类的实现方式。

```
public class ScrollerTextView extends TextView {

    private Scroller mScroller;

    public ScrollerTextView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
        mScroller = new Scroller(context);
    }

    public void smoothScrollBy(int dx, int dy) {
        int scrollX = getScrollX();
        int scrollY = getScrollY();
        mScroller.startScroll(scrollX, scrollY, dx, dy);
        invalidate();
    }

    @Override
    public void computeScroll() {
        super.computeScroll();
        if (mScroller.computeScrollOffset()) {
            scrollTo(mScroller.getCurrX(), mScroller.getCurrY());
            invalidate();
        }
    }
}
```

7.21 invalidate() 和 postInvalicate() 区别 ~leo

这道题想考察什么?

这道题想考察同学对这两种刷新的理解。

考生应该如何回答

二者的相同点

都是用来刷新界面。

二者的不同点

invalide是在UI线程中刷新View，要想在非UI线程中刷新View，就要用postInvalidate，因为postInvalidate底层是使用了Handler，同时postInvalidate可以指定一个延迟时间。
postInvalidate的调用流程：postInvalidate --> postInvalidateDelayed --> dispatchInvalidateDelayed --> mHandler.sendMessageDelayed --> 最后执行 Handler 的 MSG_INVALIDATE_RECT 消息，而这个消息里面实际上就是调用了 invalidate 方法。

7.22 View的绘制流程是从Activity的哪个生命周期方法开始执行的 ~leo

这道题想考察什么?

考察同学对Activity的生命周期和View的绘制流程是否熟悉

考生应该如何回答

View的绘制流程是从Activity的 onResume 方法开始执行的。
首先我们找到 onResume 在哪儿执行的，代码如下：

```
// ActivityThread.java
public void handleResumeActivity(IBinder token, boolean finalStateRequest,
boolean isForward,
            String reason) {
    // 1 执行 onResume 流程
    final ActivityClientRecord r = performResumeActivity(token,
finalStateRequest, reason);

    // 2 执行 View 的流程
    wm.addView(decor, l);
}
```

由上面1代码进入，我们继续跟进：

```
public ActivityClientRecord performResumeActivity(IBinder token, boolean
finalStateRequest,
            String reason) {
    r.activity.performResume(r.startsNotResumed, reason);
}
```

```
// Activity.java
final void performResume(boolean followedByPause, String reason) {
    mInstrumentation.callActivityOnResume(this);
}
```

```
public void callActivityOnResume(Activity activity) {
    activity.onResume();
}
```

到这儿我们就找到了onResume方法的执行位置。而View的绘制就是由2代码进入：`wm.addView` 中的 wm 就是 WindowManager，但是WindowManger是一个接口，实际调用的是 WindowManagerImpl 的 addView 方法

```
// WindowManagerImpl.java
public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params)
{
    mGlobal.addView(view, params, mContext.getDisplayNoVerify(), mParentWindow,
                mContext.getUserId());
}
```

mGlobal 是 WindowManagerGlobal 对象

```
// WindowManagerGlobal.java
public void addView(View view, ViewGroup.LayoutParams params,
            Display display, Window parentWindow, int userId) {
    root = new ViewRootImpl(view.getContext(), display);
    root.setView(view, wparams, panelParentView, userId);
}
```

```
public void setView(View view, WindowManager.LayoutParams attrs, View
panelParentView,
            int userId) {
    requestLayout();
}
```

到这儿我们可以看到，通过 requestLayout 开始绘制 View。
所以通过以上分析可以知道，在调用了 onResume 生命周期方法后，开始执行 View 的绘制。

7.23 Activity,Window,View三者的联系和区别 ~colin

这道题想考察什么?

1. 是否了解Activity,Window,View的原理?

考察的知识点

1. Window和View的关系
2. Activity与Window的关系

考生应该如何回答

1. Activity,Window,View分别是什么?

- Activity是安卓四大组件之一，负责界面、交互和业务逻辑处理;
- Window对安卓而言，是窗体的一种抽象，是顶级Window的外观与行为策略。目前仅有的实现类是PhoneWindow;
- View是放在Window容器的元素，Window是View的载体，View是Window的具体展示;

2. Window和View的关系。

- Window是一个界面的窗口，适用于存放View的容器，即Window是View的管理者。安卓中所有的视图都是通过Window来呈现的，比如Activity、Dialog、Toast;
- Window的添加、删除和更改是通过WindowManager来实现的，而WindowManager又是继承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);
}
```

3. Activity与Window的关系。

- Activity是向用户展示一个界面，并可以与用户进行交互。我们通过分析原理发现Activity内部持有一个Window对象，用户管理View。

```
final void attach(Context context, ActivityThread aThread,
            Instrumentation instr, IBinder token, int ident,
            Application application, Intent intent, ActivityInfo info,
            CharSequence title, Activity parent, String id,
            NonConfigurationInstances lastNonConfigurationInstances,
            Configuration config, String referrer, IVoiceInteractor
voiceInteractor,
            Window window, ActivityConfigCallback activityConfigCallback, IBinder
assistToken) {
        attachBaseContext(context);

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

        mWindow = new PhoneWindow(this, window, activityConfigCallback);
        mWindow.setWindowControllerCallback(mWindowControllerCallback);
        mWindow.setCallback(this);
        mWindow.setOnWindowDismissedCallback(this);
        mWindow.getLayoutInflater().setPrivateFactory(this);
}
```

- 从Activity的attch函数中是可以发现，新建了一个PhoneWindow对象并赋值给了mWindow。Window相当于Activity的管家，用于管理View的相关事宜。事件的分发，其实也是先交予Window再向下分发。

```
 public boolean dispatchTouchEvent(MotionEvent ev) {
        if (ev.getAction() == MotionEvent.ACTION_DOWN) {
            onUserInteraction();
        }
        if (getWindow().superDispatchTouchEvent(ev)) {
            return true;
        }
        return onTouchEvent(ev);
    }
```

- 从上面的代码中可以发现，事件被传递到了Window的superDispatchTouchEvent方法，再接着会传递给我们的DecorView。

7.24 如何实现Activity窗口快速变暗

这道题想考察什么?

考察同学对Activity的窗口属性是否熟悉

考生应该如何回答

这个主要是用到窗口的 alpha属性，然后使用属性动画实现，代码如下：

```
private void dimBackground(final float from, final float to) {
    final Window window = getWindow();
    ValueAnimator valueAnimator = ValueAnimator.ofFloat(from, to);
    valueAnimator.setDuration(500);
    valueAnimator.addUpdateListener(new AnimatorUpdateListener() {
        @Override
        public void onAnimationUpdate(ValueAnimator animation) {
            WindowManager.LayoutParams params = window.getAttributes();
            params.alpha = (Float) animation.getAnimatedValue();
            window.setAttributes(params);
        }
    });

    valueAnimator.start();
}
```

变暗

```
dimBackground(1.0f,0.5f);
```

变亮

```
dimBackground(0.5f,1.0f);
```

7.25 ListView卡顿的原因以及优化策略 ~derry

这道题想考察什么?

1. 是否了解ListView卡顿的原因以及优化策略与真实场景使用，是否熟悉ListView卡顿的原因以及优化策略在工作中的表现是什么?

考察的知识点

1. ListView卡顿的原因以及优化策略的概念在项目中使用与基本知识

考生应该如何回答

1.你在工作中是如何对ListView卡顿的原因以及优化策略？

答：

先大概分析出，有哪些原因会引发ListView卡顿：

原因一：Adapter的getView方法里面convertView没有使用setTag和getTag方式；
原因二：在getView方法里面ViewHolder初始化后的赋值或者是多个控件的显示状态和背景的显示没有优化好，抑或是里面含有复杂的计算和耗时操作；
原因三：在getView方法里面 inflate的row 嵌套太深（布局过于复杂）或者是布局里面有大图片或者背景所致；
原因四：Adapter多余或者不合理的notifySetDataChanged；
原因五：listview 被多层嵌套，多次的onMessure导致卡顿，如果多层嵌套无法避免，建议把listview的高和宽设置为fill_parent. 如果是代码继承的listview，那么也请你别忘记为你的继承类添加上LayoutPrams，注意高和宽都是fill_parent的;

在给出解决的思路：

在构造方法中耗时会导致ListView第一次加载时比较慢, 但是如果在getView方法中耗时就会导致整个ListView卡顿

原因一的解决思路：Adapter的getView方法里面convertView没有使用setTag和getTag方式；

每次getView都要执行 findViewById, 这是相当耗时的

原因二的解决思路：在getView方法里面ViewHolder初始化后的赋值或者是多个控件的显示状态和背景的显示没有优化好，抑或是里面含有复杂的计算和耗时操作；

如果使用了ViewHolder, 那么viewHolder初始化, 或者控件操作太复杂,导致耗时

原因三的解决思路：在getView方法里面 inflate的row 嵌套太深（布局过于复杂）或者是布局里面有大图片或者背景所致；

单个item的布局太复杂, 或者 加载大图片, 或者从网上获取图片

原因四的解决思路：Adapter多余或者不合理的notifySetDataChanged；

通知所有的观察者, 底层数据已经改变, 所有显示这些数据的View都应该刷新

原因五的解决思路：listview 被多层嵌套，多次的onMessure导致卡顿，如果多层嵌套无法避免，建议把listview的高和宽设置为fill_parent. 如果是代码继承的listview，那么也请你别忘记为你的继承类添加上LayoutPrams，注意高和宽都是fill_parent的;

View在Draw的时候分成两个阶段：measure和layout，在measure阶段时主要就是为了计算两个参数：height和width。而且要注意的是，这是个递归的过程，从顶向下，DecorView开始依次调用自己子元素的measure。计算完成这两个参数后就开始layout，最后再是draw的调用。
对于ListView，当然每一个Item都会被调用measure方法，而在这个过程中getView和getCount会被调用，而且看用户的需求，可能会有很多次调用。

而为什么会有很多组次调用呢?

问题就在于在layout中的决定ListView或者它的父元素的height和width属性的定义了。fill_parent会好一点，计算方法会比较简单，只要跟父元素的大小相似就行，但是即使是fill_parent，也不能给View当饭吃，还是要计算出来具体的dip，所以measure还是会被调用，只是可能比wrap_content的少一点。至于自适应的它会一直考量它的宽和高，根据内容（也就是它的子Item）计算宽高。可能这个measure过程会反复执行，如果父元素也是wrap_content，这个过程会更加漫长。

所以，解决方法就是尽量避免自适应，除非是万不得已，固定大小或者填充的效果会比较好一些。

我们把listview与他父控件的所有高度与宽度都设置为fill_parent,果然getview调用正常了，注意是所有的高度和宽度情况

7.26 ViewHolder为什么要被声明成静态内部类

这道题想考察什么?

考察同学对静态内部类和非静态内部类的主要区别

考生应该如何回答

这个是考静态内部类和非静态内部类的主要区别之一。非静态内部类会隐式持有外部类的引用，就像大家经常将自定义的adapter在Activity类里，然后在adapter类里面是可以随意调用外部activity的方法的。
当你将内部类定义为static时，你就调用不了外部类的实例方法了，因为这时候静态内部类是不持有外部类的引用的。声明ViewHolder静态内部类，可以将ViewHolder和外部类解引用。
可能你会说ViewHolder都很简单，不定义为static也没事吧。确实如此，但是如果你将它定义为static的，说明你懂这些含义。万一有一天你在这个ViewHolder加入一些复杂逻辑，做了一些耗时工作，那么如果ViewHolder是非静态内部类的话，就很容易出现内存泄露。
如果是静态的话，你就不能直接引用外部类，迫使你关注如何避免相互引用。 所以将 ViewHolder内部类 定义为静态的，是一种良好的编程习惯。

7.27 Android中的动画有哪些? 动画占用大量内存，如何优化

这道题想考察什么?

考察同学对Android的动画是否熟悉。

考生应该如何回答

Android动画主要有三种：逐帧动画、补间动画、属性动画。

逐帧动画

将多张图片组合起来进行播放，很多 App 的加载动画是采用这种方式。
实现它的方式有几种，一种是直接作出 gif 或 webP 格式的图片，直接播放；一种则是 android 系统提供封装好的方法，将动画定义在 xml 中，用 animation-list 标签来实现它， 元素是必要的，可以包含n个 元素，每个 item 代表一帧动画。 以上两种都能实现，但比较耗费内存，想想看，假如说帧动画有 100 张图片，每张图片都比较大的情况下，此时执行动画，瞬间加载这么多图片，内存会出现什么问题，很大的几率会 OOM。

有没有节省内存的方法？帧动画说白了就是固定时间的刷新一个ImageView，显示不同的图片，既然知道了这些，那么我们可以使用 Handler ，每隔100毫秒就发送一个通知，接收到通知后获取一个指定的图片资源，刷新到ImageView上面，这样就避免了瞬间加载多张图片资源的问题了；如果再进一步优化，我们可以获取图片资源时，使用缓存把获取到的图片存起来，这样就不必每次都从资源里面获取，先判断缓存中有没有；再进一步，我们可以把中封装起来，熟悉自定义控件的同学，我们可以直接继承View，来实现这个逻辑。图片绘制说白了就是对 Drawable 的绘制，之前文章中也提到过，这里就不多说了，直接上代码，主要是提供一个思路。

```
public class AnimationView extends View {

    private final static String KEY_MARK = "0";

    private int[] mFrameRess;
    private int mDuration;
    private int mLastFrame;
    private int mCurrentFrame;
    private boolean mPause;
    private Rect mDst;
    private BitmapLRU mLruCache;

    public AnimationView(Context context) {
        this(context, null);
    }

    public AnimationView(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public AnimationView(Context context, AttributeSet attrs, int defStyleAttr)
{
        super(context, attrs, defStyleAttr);
        init(context, attrs);
    }

    private void init(Context context, AttributeSet attrs) {
        TypedArray typedArray =
getResources().obtainTypedArray(R.array.animation_1);
        int len = typedArray.length();
        int[] resId = new int[len];
        for (int i = 0; i < len; i++) {
            resId[i] = typedArray.getResourceId(i, -1);
        }
        typedArray.recycle();

        this.mFrameRess = resId;
        this.mDuration = 50;
        this.mLastFrame = resId.length - 1;

        mLruCache = BitmapLRU.getInstance();
    }

    @Override
    protected void onDetachedFromWindow() {
        super.onDetachedFromWindow();
        pauseAnimation();
    }
```

```
    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        super.onSizeChanged(w, h, oldw, oldh);
        mDst = new Rect(0, 0, w, h);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        canvas.save();
        Drawable d = takeDrawable();
        d.setBounds(mDst);
        d.draw(canvas);
        canvas.restore();

    }

    private Drawable takeDrawable(){
        int resid = mFrameRess[mCurrentFrame];
        String key = KEY_MARK + resid;
        Drawable drawable = mLruCache.get(key);
        if(drawable == null){
            drawable = getResources().getDrawable(resid);
            mLruCache.put(key, drawable);
        }
        return drawable;
    }

    public void play(final int i) {
        postDelayed(new Runnable() {

            @Override
            public void run() {
                mCurrentFrame = i;
                if (mPause) {
                    return;
                }
                invalidate();
                if (i == mLastFrame) {
                    play(0);
                } else {
                    play(i + 1);
                }
            }
        }, mDuration);
    }

    public void pauseAnimation() {
        this.mPause = true;
    }
}
```

```
public class BitmapLRU {

    private int maxMemory = (int) Runtime.getRuntime().maxMemory();
```

```
    private int cacheSize = maxMemory / 18;
    private LruCache<String, Drawable> mCache;

    private BitmapLRU(){
        mCache = new LruCache<String, Drawable>(cacheSize){
            @Override
            protected int sizeOf(String key, Drawable value) {
                return value.getIntrinsicWidth() * value.getIntrinsicHeight();
            }
        };
    }

    private static BitmapLRU single = new BitmapLRU();

    public static BitmapLRU getInstance(){
        return single;
    }


    public void put(String key, Drawable value){
        mCache.put(key, value);
    }

    public Drawable get(String key){
        return mCache.get(key);
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:tools="http://schemas.android.com/tools"
tools:ignore="MissingTranslation">


    <array name="animation_1">
        <item>@drawable/da_00</item>
        <item>@drawable/da_01</item>
        <item>@drawable/da_02</item>
        <item>@drawable/da_03</item>
        <item>@drawable/da_04</item>
        <item>@drawable/da_05</item>
        <item>@drawable/da_06</item>
        <item>@drawable/da_07</item>
        <item>@drawable/da_08</item>
        <item>@drawable/da_09</item>
        <item>@drawable/da_10</item>
    </array>

</resources>
```

在外面使用 animationView.play(0); 即可。

补间动画

补间动画由 Animation 类来实现具体效果，包括平移（TranslateAnimation）、缩放(ScaleAnimation)、旋转(RotateAnimation)、透明度(AlphaAnimation)四个子类，四种变化，但补间动画只是达到了其视觉效果，并不是真正的位置上的变化。（属性动画出来之后，补间动画就不那么常用了）。

属性动画

最为强大的动画，弥补了补间动画的缺点，实现位置+视觉的变化。
属性动画中有一类无限循环的动画，如果在Activity中播放此类动画且没有在 onDestroy中去停止动画，那么动画会一直播放下去，尽管已经无法在界面上看见动画效果了，并且这个时候 Activity的 View会被动画持有，而View又持有了Activty,最终Activity无法释放。下面的动画是无限动画，会泄露当前的Activity,解决方法是在Activity的onDestroy中调用animator.cancel()来停止动画。

7.28 自定义View执行invalidate()方法,为什么有时候不会回调onDraw()

这道题想考察什么?

1. 是否了解自定义View执行invalidate()方法,为什么有时候不会回调onDraw()与真实场景使用，是否熟悉自定义View执行invalidate()方法,为什么有时候不会回调onDraw()

考察的知识点

1. 自定义View执行invalidate()方法,为什么有时候不会回调onDraw()的概念在项目中使用与基本知识

考生应该如何回答

1.自定义View执行invalidate()方法,为什么有时候不会回调onDraw()，你知道吗?

自定义一个view时，重写onDraw。

调用view.invalidate(),会触发onDraw和computeScroll()。前提是该view被附加在当前窗口上

view.postInvalidate(); // 注意: 是在非UI线程上调用的

自定义一个ViewGroup，重写onDraw。

onDraw可能不会被调用，原因是需要先设置一个背景(颜色或图)。

表示这个group有东西需要绘制了，才会触发draw，之后是onDraw。

因此，一般直接重写dispatchDraw来绘制viewGroup

自定义一个ViewGroup

dispatchDraw会调用drawChild

```
自定义一个view时，重写onDraw。调用view.invalidate(),会触发onDraw和computeScroll()。前提是该view被附加在当前窗口.
view.postInvalidate(); //是在非UI线程上调用的

自定义一个ViewGroup，重写onDraw。onDraw可能不会被调用，原因是需要先设置一个背景(颜色或图)。表示这个group有东西需要绘制了，才会触发draw，之后是onDraw。因此，一般直接重写dispatchDraw来绘制viewGroup.自定义一个ViewGroup,dispatchDraw会调用drawChild.
```

7.29 DecorView, ViewRootImpl,View之间的关系

这道题想考察什么?

1. 是否了解DecorView, ViewRootImpl,View之间的关系与真实场景使用，是否熟悉DecorView, ViewRootImpl,View之间的关系

考察的知识点

1. DecorView, ViewRootImpl,View之间的关系的概念在项目中使用与基本知识

考生应该如何回答

1.DecorView, ViewRootImpl,View之间的关系，你知道吗?

```
public void setContentView(@LayoutRes int layoutResID) {
     getWindow().setContentView(layoutResID);  //调用getWindow方法，返回mWindow
     initWindowDecorActionBar();
}
...
public Window getWindow() {
     return mWindow;
}
```

从上面看出，里面调用了mWindow的setContentView方法，那么这个“mWindow”是何方神圣呢？尝试追踪一下源码，发现mWindow是Window类型的，但是它是一个抽象类，setContentView也是抽象方法，所以我们要找到Window类的实现类才行。我们在Activity中查找一下mWindow在哪里被赋值了，可以发现它在Activity#attach方法中有如下实现：

```
    final void attach(Context context, ActivityThread aThread,
            Instrumentation instr, IBinder token, int ident,
            Application application, Intent intent, ActivityInfo info,
            CharSequence title, Activity parent, String id,
            NonConfigurationInstances lastNonConfigurationInstances,
            Configuration config, String referrer, IVoiceInteractor
voiceInteractor) {
        ...
        mWindow = new PhoneWindow(this);
        ...
    }
```

我们只看关键部分，这里实例化了PhoneWindow类，由此得知，PhoneWindow是Window的实现类，那么我们在PhoneWindow类里面找到它的setContentView方法，看看它又实现了什么，**PhoneWindow#setContentView**:

```
@Override
public void setContentView(int layoutResID) {
    // Note: FEATURE_CONTENT_TRANSITIONS may be set in the process of installing
the window
    // decor, when theme attributes and the like are crystalized. Do not check
the feature
    // before this happens.
    if (mContentParent == null) { // 1
        installDecor();
    } else if (!hasFeature(FEATURE_CONTENT_TRANSITIONS)) {
        mContentParent.removeAllViews();
    }

    if (hasFeature(FEATURE_CONTENT_TRANSITIONS)) {
        final Scene newScene = Scene.getSceneForLayout(mContentParent,
layoutResID,
                getContext());
        transitionTo(newScene);
    } else {
        mLayoutInflater.inflate(layoutResID, mContentParent); // 2
    }
    final Callback cb = getCallback();
    if (cb != null && !isDestroyed()) {
        cb.onContentChanged();
    }
}
```

首先判断了mContentParent是否为null，如果为空则执行installDecor()方法，那么这个mContentParent又是什么呢？我们看一下它的注释：

```
// This is the view in which the window contents are placed. It is either
// mDecor itself, or a child of mDecor where the contents go.
private ViewGroup mContentParent;
```

它是一个ViewGroup类型，结合②号代码处，可以得知，这个mContentParent是我们设置的布局(即main.xml)的父布局。注释还提到了，这个mContentParent是mDecor本身或者是mDecor的一个子元素，这句话什么意思呢？这里先留一个疑问，下面会解释。

这里先梳理一下以上的内容：Activity通过PhoneWindow的setContentView方法来设置布局，而设置布局之前，会先判断是否存在mContentParent，而我们设置的布局文件则是mContentParent的子元素。

创建DecorView

接着上面提到的installDecor()方法，我们看看它的源码，**PhoneWindow#installDecor**:

```
private void installDecor() {
    if (mDecor == null) {
        mDecor = generateDecor(); // 1
        mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCENDANTS);
        mDecor.setIsRootNamespace(true);
```

```
        if (!mInvalidatePanelMenuPosted && mInvalidatePanelMenuFeatures != 0) {
            mDecor.postOnAnimation(mInvalidatePanelMenuRunnable);
        }
    }
    if (mContentParent == null) {
        mContentParent = generateLayout(mDecor); // 2
        ...
        }
    }
}
```

首先，会执行①号代码，调用**PhoneWindow#generateDecor**方法：

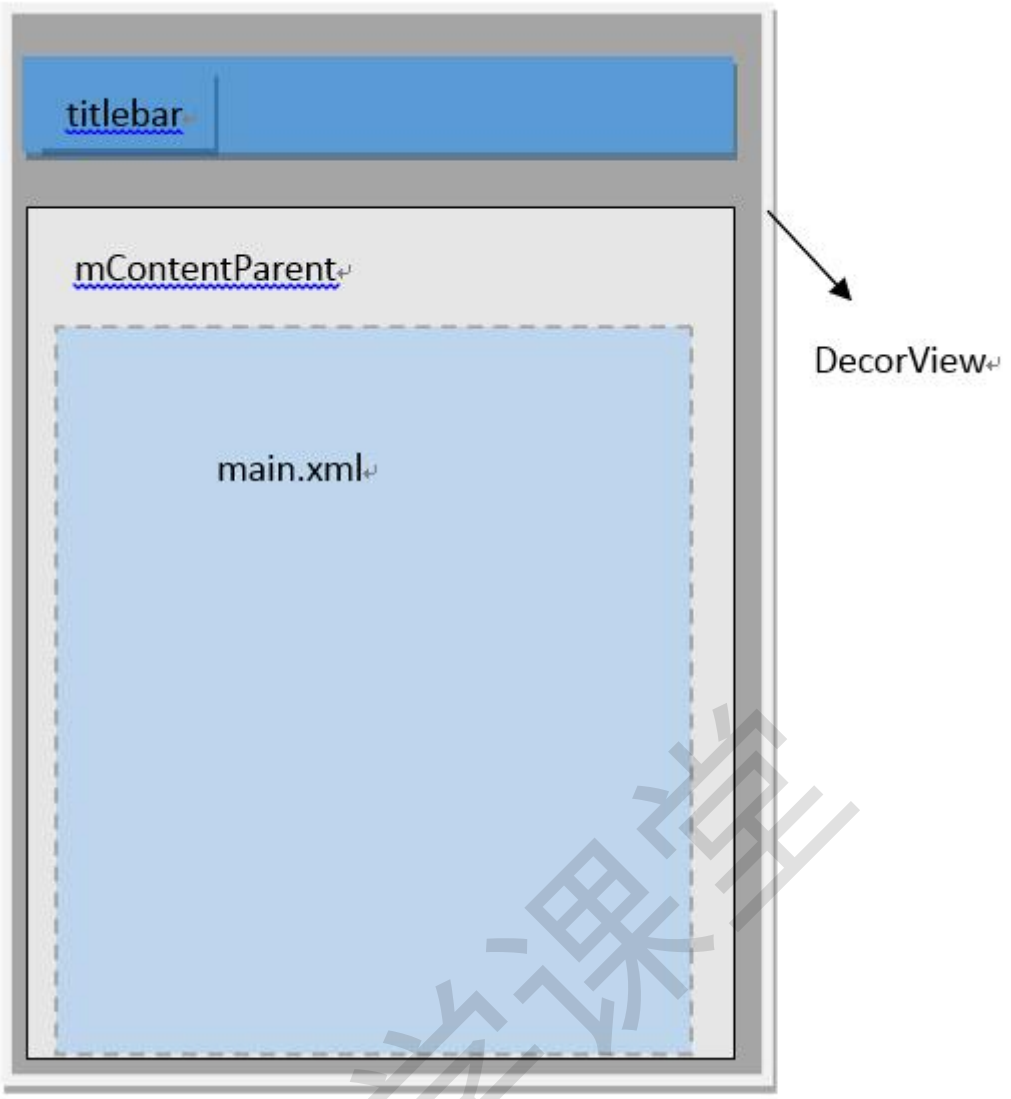
```
protected DecorView generateDecor() {
    return new DecorView(getContext(), -1);
}
```

可以看出，这里实例化了DecorView，而DecorView则是PhoneWindow类的一个内部类，继承于FrameLayout，由此可知它也是一个ViewGroup。
那么，DecroView到底充当了什么样的角色呢?
其实，DecorView是整个ViewTree的最顶层View，它是一个FrameLayout布局，代表了整个应用的界面。在该布局下面，有标题view和内容view这两个子元素，而内容view则是上面提到的mContentParent。我们接着看②号代码，PhoneWindow#generateLayout方法

```
protected ViewGroup generateLayout(DecorView decor) {
        // Apply data from current theme.
        // 从主题文件中获取样式信息
        TypedArray a = getWindowStyle();

        ...

        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);
        }

        if(...){
            ...
        }

        // Inflate the window decor.
        // 加载窗口布局
        int layoutResource;
        int features = getLocalFeatures();
        // System.out.println("Features: 0x" + Integer.toHexString(features));
        if ((features & (1 << FEATURE_SWIPE_TO_DISMISS)) != 0) {
            layoutResource = R.layout.screen_swipe_dismiss;
        } else if(...){
            ...
        }

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

```
        decor.addView(in, new ViewGroup.LayoutParams(MATCH_PARENT,
MATCH_PARENT)); //往DecorView中添加子View，即mContentParent
        mContentRoot = (ViewGroup) in;

        ViewGroup contentParent = (ViewGroup)findViewById(ID_ANDROID_CONTENT);
// 这里获取的就是mContentParent
        if (contentParent == null) {
            throw new RuntimeException("Window couldn't find content container
view");
        }

        if ((features & (1 << FEATURE_INDETERMINATE_PROGRESS)) != 0) {
            ProgressBar progress = getCircularProgressBar(false);
            if (progress != null) {
                progress.setIndeterminate(true);
            }
        }

        if ((features & (1 << FEATURE_SWIPE_TO_DISMISS)) != 0) {
            registerSwipeCallbacks();
        }

        // Remaining setup -- of background and title -- that only applies
        // to top-level windows.
        ...

        return contentParent;
    }
```

由以上代码可以看出，该方法还是做了相当多的工作的，首先根据设置的主题样式来设置DecorView的风格，比如说有没有titlebar之类的，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，这也解释了上面的疑问：mContentParent是DecorView本身或者是DecorView的一个子元素。
用一幅图来表示DecorView的结构如下：



小结：DecorView是顶级View，内部有titlebar和contentParent两个子元素，contentParent的id是content，而我们设置的main.xml布局则是contentParent里面的一个子元素。

在DecorView创建完毕后，让我们回到PhoneWindow#setContentView方法，直接看②号代码：mLayoutInflater.inflate(layoutResID, mContentParent);这里加载了我们设置的main.xml布局文件，并且设置mContentParent为main.xml的父布局，至于它怎么加载的，这里就不展开来说了。

到目前为止，通过setContentView方法，创建了DecorView和加载了我们提供的布局，但是这时，我们的View还是不可见的，因为我们仅仅是加载了布局，并没有对View进行任何的测量、布局、绘制工作。在View进行测量流程之前，还要进行一个步骤，那就是把DecorView添加至window中，然后经过一系列过程触发ViewRootImpl#performTraversals方法，在该方法内部会正式开始测量、布局、绘制这三大流程。至于该一系列过程是怎样的，因为涉及到了很多机制，这里简单说明一下：

将DecorView添加至Window
每一个Activity组件都有一个关联的Window对象，用来描述一个应用程序窗口。每一个应用程序窗口内部又包含有一个View对象，用来描述应用程序窗口的视图。上文分析了创建DecorView的过程，现在则要把DecorView添加到Window对象中。而要了解这个过程，我们首先要简单先了解一下Activity的创建过程：
首先，在ActivityThread#handleLaunchActivity中启动Activity，在这里面会调用到Activity#onCreate方法，从而完成上面所述的DecorView创建动作，当onCreate()方法执行完毕，在handleLaunchActivity方法会继续调用到ActivityThread#handleResumeActivity方法，我们看看这个方法的源码：

```
final void handleResumeActivity(IBinder token, boolean clearHide, boolean
isForward) {
    //...
    ActivityClientRecord r = performResumeActivity(token, clearHide); // 这里会调
用到onResume()方法

    if (r != null) {
        final Activity a = r.activity;

        //...
        if (r.window == null && !a.mFinished && willBeVisible) {
            r.window = r.activity.getWindow(); // 获得window对象
            View decor = r.window.getDecorView(); // 获得DecorView对象
            decor.setVisibility(View.INVISIBLE);
            ViewManager wm = a.getWindowManager(); // 获得windowManager对象
            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;
                wm.addView(decor, l); // 调用addView方法
            }
            //...
        }
    }
}
```

在该方法内部，获取该activity所关联的window对象，DecorView对象，以及windowManager对象，而WindowManager是抽象类，它的实现类是WindowManagerImpl，所以后面调用的是WindowManagerImpl#addView方法，我们看看源码：

```
public final class WindowManagerImpl implements WindowManager {
    private final WindowManagerGlobal mGlobal =
WindowManagerGlobal.getInstance();
    ...
    @Override
    public void addView(View view, ViewGroup.LayoutParams params) {
        mGlobal.addView(view, params, mDisplay, mParentWindow);
    }
}
```

接着调用了mGlobal的成员函数，而mGlobal则是WindowManagerGlobal的一个实例，那么我们接着看**WindowManagerGlobal#addView**方法：

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

        ViewRootImpl root;
        View panelParentView = null;

        synchronized (mLock) {
            ...

            root = new ViewRootImpl(view.getContext(), display); // 1
```

```
            view.setLayoutParams(wparams);

            mViews.add(view);
            mRoots.add(root);
            mParams.add(wparams);
        }

        // do this last because it fires off messages to start doing things
        try {
            root.setView(view, wparams, panelParentView); // 2
        } catch (RuntimeException e) {
            // BadTokenException or InvalidDisplayException, clean up.
            synchronized (mLock) {
                final int index = findViewLocked(view, false);
                if (index >= 0) {
                    removeViewLocked(index, true);
                }
            }
            throw e;
        }
    }
```

先看①号代码处，实例化了ViewRootImpl类，接着，在②号代码处，调用ViewRootImpl#setView方法，并把DecorView作为参数传递进去，在这个方法内部，会通过跨进程的方式向WMS（WindowManagerService）发起一个调用，从而将DecorView最终添加到Window上，在这个过程中，ViewRootImpl、DecorView和WMS会彼此关联，至于详细过程这里不展开来说了。
最后通过WMS调用ViewRootImpl#performTraverals方法开始View的测量、布局、绘制流程

7.30 如何更新UI，为什么子线程不能更新UI？(美团) ~derry

这道题想考察什么?

1. 是否了解为什么子线程不能更新UI的概念与真实场景使用，是否熟悉为什么子线程不能更新UI的本质区别?

考察的知识点

1. 为什么子线程不能更新UI的概念在项目中使用与基本知识

考生应该如何回答

1.你工作这么些年，有注意到子线程不能更新UI?

答:

其实Android开发者都在说，子线程不能更新UI，难道一定就不能在子线程更新UI吗，那也未必:

下面代码是可以的，运行结果并无异常，可以正常的在子线程中更新了`TextView`控件

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tv = findViewById(R.id.tv);
        new Thread(new Runnable() {
            @Override
            public void run() {
                tv.setText("Test");
            }
        }).start();
}
```

但是假如让线程休眠`1000ms`,就会发生错误:

```
Only the original thread that created a view hierarchy can touch its views.
```

上面报错的意思是只有创建视图层次结构的原始线程才能更新这个视图，也就是说只有主线程才有权力去更新`UI`，其他线程会直接抛异常的;
从`at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:7905)`的异常路径可以看到抛出异常的最终在`ViewRootIml`的`checkThread`方法里，`ViewRootIml`是`View`的根类，其控制着`View`的测量、绘制等操作，那么现在我们转到`ViewRootImpl.java`源码观察:

```
@Override
public void requestLayout() {
        if (!mHandlingLayoutInLayoutRequest) {
            checkThread();
            mLayoutRequested = true;
            scheduleTraversals();
        }
}

void checkThread() {
        if (mThread != Thread.currentThread()) {
            throw new CalledFromWrongThreadException(
          "Only the original thread that created a view hierarchy can touch its
views.");
        }
}
```

上面的`scheduleTraversals()`函数里是对`View`进行绘制操作，而在绘制之前都会检查当前线程是否为主线程`mThread`，如果不是主线程，就抛出异常；这样做法就限制了开发者在子线程中更新UI的操作;
但是为什么最开始的在`onCreate()`里子线程对`UI`的操作没有报错呢，可以设想一下是因为`ViewRootImp`此时还没有创建，还未进行当前线程的判断;
现在，我们寻找`ViewRootImp`在何时创建，从`Activity`启动过程中寻找源码,通过分析可以查看`ActivityThread.java`源码,并找到`handleResumeActivity`方法:

```
final void handleResumeActivity(IBinder token,boolean clearHide, boolean
isForward, boolean reallyResume) {
        ...
        // TODO Push resumeArgs into the activity for consideration
```

```
        ActivityClientRecord r = performResumeActivity(token, clearHide);
        if (r.window == null && !a.mFinished && willBeVisible) {
                r.window = r.activity.getWindow();
                View decor = r.window.getDecorView();
                decor.setVisibility(View.INVISIBLE);
                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;
                    wm.addView(decor, l);
                }

            } else if (!willBeVisible) {
                if (localLOGV) Slog.v(
                    TAG, "Launch " + r + " mStartedActivity set");
                r.hideForNow = true;
            }
        ...
}
```

可以看到内部调用了`performResumeActivity`方法:

```
public final ActivityClientRecord performResumeActivity(IBinder token, boolean
clearHide) {
    if (r != null && !r.activity.mFinished) {
        r.activity.performResume();
    省略...
}
```

会发现`在内部调用了`Activity 的 performResume `方法，可以肯定应该是要回调生命周期的`onResume方法了:

```
    final void performResume() {
        ...
        mCalled = false;
        // mResumed is set by the instrumentation
        mInstrumentation.callActivityOnResume(this);
        if (!mCalled) {
            throw new SuperNotCalledException(
                "Activity " + mComponent.toShortString() +
                " did not call through to super.onResume()");
        }
        ...
    }
```

接着然后又调用了`Instrumentation`的`callActivityOnResume`方法:

```
    public void callActivityOnResume(Activity activity) {
        activity.mResumed = true;
        activity.onResume();

        if (mActivityMonitors != null) {
```

```
            synchronized (mSync) {
                final int N = mActivityMonitors.size();
                for (int i=0; i<N; i++) {
                    final ActivityMonitor am = mActivityMonitors.get(i);
                    am.match(activity, activity, activity.getIntent());
                }
            }
        }
    }
```

然后就可以看到执行了 `activity.onResume()` 方法，也就是回调了 `Activity` 生命周期的 `onResume` 方法;现在让我们回头看看 `handleResumeActivity` 方法，会执行这段代码：

```
...
r.activity.mVisibleFromServer = true;
                mNumVisibleActivities++;
                if (r.activity.mVisibleFromClient) {
                    r.activity.makeVisible();
                }
```

发现在内部调用了 `Activity` 的 `makeVisible` 方法：

```
void makeVisible() {
        if (!mWindowAdded) {
            ViewManager wm = getWindowManager();
            wm.addView(mDecor, getWindow().getAttributes());
            mWindowAdded = true;
        }
        mDecor.setVisibility(View.VISIBLE);
    }
```

源码内部调用了 `WindowManager` 的 `addView` 方法，而 `WindowManager` 方法的实现类是 `WindowManagerImp` 类，直接找 `WindowManagerImp` 的 `addView` 方法：

```
    @Override
    public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams
params) {
        applyDefaultToken(params);
        mGlobal.addView(view, params, mDisplay, mParentWindow);
    }
```

最终然后又调用了 `WindowManagerGlobal` 的 `addView` 方法，在该方法中，终于看到了 `ViewRootImpl` 的创建；

```
public void addView(View view, ViewGroup.LayoutParams params,Display display,
Window parentWindow) {
        ...
            root = new ViewRootImpl(view.getContext(), display);
            view.setLayoutParams(wparams);
            mViews.add(view);
            mRoots.add(root);
            mParams.add(wparams);
        }
        ...
    }
```

真相大白：

刚刚源码分析可得知，`ViewRootImpl`对象是在`onResume`方法回调之后才创建，那么就说明了为什么在生命周期的`onCreate`方法里，甚至是`onResume`方法里都可以实现子线程更新UI，因为此时还没有创建`ViewRootImpl`对象，并不会进行是否为主线程的判断；

个人理解

必须要在主线程更新`UI`，实际是为了提高界面的效率和安全性，带来更好的流畅性；你反推一下，假如允许多线程更新`UI`，但是访问`UI`是没有加锁的，一旦多线程抢占了资源，那么界面将会乱套更新了，体验效果就不言而喻了；所以在`Android`中规定必须在主线程更新`UI`，很合理吧

总结源码分析：

第一点：子线程可以在ViewRootImpl还没有创建之前更新UI的

第二点：访问UI是没有加对象锁的，在子线程环境下更新UI，会造成各种未知风险的

第三点：Android开发者一定要在主线程更新UI操作，这个是职业习惯哦

7.31 RecyclerView是什么？如何使用？如何返回不一样的Item

这道题想考察什么？

考察同学是否对RecyclerView的使用熟悉。

考生应该如何回答

RecyclerView是什么？

官方解释是提供一个固定的View让有限的窗口显示一个大数据集。

如何使用?

RecyclerView的使用类似ListView，只是更加灵活。xml与ListView一样，这里就不介绍了。
Adapter的创建如下：

```
public class CustomAdapter extends
RecyclerView.Adapter<CustomAdapter.CustomViewHolder> {

    private final Context context;
    private List<String> list;

    public CustomAdapter(Context context, List<String> list) {
        this.context = context;
        this.list = list;
    }

    @Override
    public CustomViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(context).inflate(R.layout.item_rv,
parent, false);
        return new CustomViewHolder(view);
    }

    @Override
    public void onBindViewHolder(CustomViewHolder holder, int position) {
        holder.tv.setText(list.get(position));
    }

    @Override
    public int getItemCount() {
        return list == null ? 0 : list.size();
    }

    public static class CustomViewHolder extends RecyclerView.ViewHolder {
        private TextView tv;

        public CustomViewHolder(View itemView) {
            super(itemView);
            tv = itemView.findViewById(R.id.tv);
        }
    }
}
```

然后在Activity中的使用如下：

```
RecyclerView rv = findViewById(R.id.rv);

LinearLayoutManager linearLayoutManager = new LinearLayoutManager(this);
rv.setLayoutManager(linearLayoutManager);

final List<String> list = new ArrayList<>();
for (int i = 0; i < 1000; i++) {
    list.add("" + i);
}

final CustomAdapter adapter = new CustomAdapter(this, list);
rv.setAdapter(adapter);
```

如何返回不一样的Item

这个主要是通过 viewType 参数处理，只需改动 RecyclerView.Adapter即可，代码如下：

```
public class CustomAdapter extends RecyclerView.Adapter<RecyclerView.ViewHolder>
{

    private final Context context;
    private List<String> list;

    private static final int VIEW_TYPE_ONE = 0;
    private static final int VIEW_TYPE_TWO = 1;

    public CustomAdapter(Context context, List<String> list) {
        this.context = context;
        this.list = list;
    }

    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int
viewType) {
        RecyclerView.ViewHolder viewHolder = null;
        switch (viewType) {
            case VIEW_TYPE_ONE:
                viewHolder = new
CustomViewHolder(LayoutInflater.from(context).inflate(R.layout.item_rv, parent,
false));
                break;
            case VIEW_TYPE_TWO:
                viewHolder = new
CustomViewHolder2(LayoutInflater.from(context).inflate(R.layout.item_rv2,
parent, false));
                break;
            default:
                throw new IllegalStateException("Unexpected value: " +
viewType);
        }
        return viewHolder;
    }

    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
        switch (holder.getItemViewType()) {
```

```
            case VIEW_TYPE_ONE:
                ((CustomViewHolder) holder).tv.setText(list.get(position));
                break;
            case VIEW_TYPE_TWO:
                ((CustomViewHolder2) holder).tv.setText(list.get(position));
                ((CustomViewHolder2) holder).tv2.setText(list.get(position));
                break;
        }
    }

    @Override
    public int getItemViewType(int position) {
        if (position % 2 == 0) {
            return VIEW_TYPE_ONE;
        } else {
            return VIEW_TYPE_TWO;
        }
    }

    @Override
    public int getItemCount() {
        return list == null ? 0 : list.size();
    }

    public static class CustomViewHolder extends RecyclerView.ViewHolder {
        private TextView tv;

        public CustomViewHolder(View itemView) {
            super(itemView);
            tv = itemView.findViewById(R.id.tv);
        }
    }

    public static class CustomViewHolder2 extends RecyclerView.ViewHolder {
        private TextView tv;
        private TextView tv2;

        public CustomViewHolder2(View itemView) {
            super(itemView);
            tv = itemView.findViewById(R.id.tv);
            tv2 = itemView.findViewById(R.id.tv2);
        }
    }
}
```

7.32 RecyclerView的回收复用机制

这道题想考察什么?

考察同学是否对RecyclerView的Recycler熟悉。

考生应该如何回答

RecyclerView的缓存复用机制主要有Recycler管理。
Recycler的成员变量如下:

```
public final class Recycler {
    final ArrayList<ViewHolder> mAttachedScrap = new ArrayList<>();
    ArrayList<ViewHolder> mChangedScrap = null;

    final ArrayList<ViewHolder> mCachedViews = new ArrayList<>();

    RecycledViewPool mRecyclerPool;

    private ViewCacheExtension mViewCacheExtension;
}
```

这些缓存集合可以分为 4 个级别，按优先级从高到底为:

- 一级缓存：mAttachedScrap 和 mChangedScrap ，用来缓存还在屏幕内的 ViewHolder
  - mAttachedScrap 存储的是当前还在屏幕中的 ViewHolder；按照 id 和 position 来查找 ViewHolder
  - mChangedScrap 表示数据已经改变的 ViewHolder 列表, 存储 notifyXXX 方法时需要改变的 ViewHolder
- 二级缓存：mCachedViews ，用来缓存移除屏幕之外的 ViewHolder，默认情况下缓存容量是 2，可以通过 setViewCacheSize 方法来改变缓存的容量大小。如果 mCachedViews 的容量已满，则会根据 FIFO 的规则移除旧 ViewHolder
- 三级缓存：ViewCacheExtension ，开发给用户的自定义扩展缓存，需要用户自己管理 View 的创建和缓存。个人感觉这个拓展脱离了 Adapter.createViewHolder 使用的话会造成 View 创建 与 数据绑定及其它代码太分散，不利于维护，使用场景很少仅做了解

```
/*
 * Note that, Recycler never sends Views to this method to be cached. It is
developers
  * responsibility to decide whether they want to keep their Views in this
custom cache or let
  * the default recycling policy handle it.
  */
public abstract static class ViewCacheExtension {
 public abstract View getViewForPositionAndType(...);
}
```

- 四级缓存：RecycledViewPool ，ViewHolder 缓存池，在有限的 mCachedViews 中如果存不下新的 ViewHolder 时，就会把 ViewHolder 存入RecyclerViewPool 中。
  - 按照 Type 来查找 ViewHolder
  - 每个 Type 默认最多缓存 5 个
  - 可以多个 RecyclerView 共享 RecycledViewPool

接下来我们看下这四级缓存是怎么工作的

复用

RecyclerView 作为一个 “平平无奇” 的 View，子 View 的排列和布局当然是从 onLayout 入手了，调用链：

```
   RecyclerView.onLayout(...)
-> RecyclerView.dispatchLayout()
-> RecyclerView.dispatchLayoutStep2() // do the actual layout of the views for
the final state.
-> mLayout.onLayoutChildren(mRecycler, mState) // mLayout 类型为 LayoutManager
-> LinearLayoutManager.onLayoutChildren(...) // 以 LinearLayoutManager 为例
-> LinearLayoutManager.fill(...) // The magic functions :) 填充给定的布局，注释很自信
的说这个方法很独立，稍微改动就能作为帮助类的一个公开方法，程序员的快乐就是这么朴实无华。
-> LinearLayoutManager.layoutChunk(recycler, layoutState) // 循环调用，每次调用填充一
个 ItemView 到 RV
-> LinearLayoutManager.LayoutState.next(recycler)
-> RecyclerView.Recycler.getViewForPosition(int) // 回到主角了,通过 Recycler 获取指
定位置的 ItemView
-> Recycler.getViewForPosition(int, boolean) // 调用下面方法获取 ViewHolder,并返回上
面需要的 viewHolder.itemView
-> Recycler.tryGetViewHolderForPositionByDeadline(...) // 终于找到你，还好没放弃~
```

可以看出最终调用 tryGetViewHolderForPositionByDeadline，来看看这个方法是怎么拿到相应位置上的 ViewHolder：

```
ViewHolder tryGetViewHolderForPositionByDeadline(int position, ...) {
    if (mState.isPreLayout()) {
        // 0) 预布局从 mChangedScrap 里面去获取 ViewHolder，动画相关
        holder = getChangedScrapViewForPosition(position);
    }

    if (holder == null) {
        // 1) 分别从 mAttachedScrap、 mHiddenViews、mCachedViews 获取 ViewHolder
        //    这个 mHiddenViews 是用来做动画期间的复用
        holder = getScrapOrHiddenOrCachedHolderForPosition(position, dryRun);
    }

    if (holder == null) {
        final int type = mAdapter.getItemViewType(offsetPosition);
        // 2) 如果 Adapter 的 hasStableIds 方法返回为 true
        //    优先通过 ViewType 和 ItemId 两个条件从 mAttachedScrap 和 mCachedViews
寻找
        if (mAdapter.hasStableIds()) {
            holder =
getScrapOrCachedViewForId(mAdapter.getItemId(offsetPosition), type, dryRun);
        }

        if (holder == null && mViewCacheExtension != null) {
            // We are NOT sending the offsetPosition because LayoutManager does
not know it.
            // 3) 从自定义缓存获取，别问，问就是别用
            View view = mViewCacheExtension
                .getViewForPositionAndType(this, position, type);
            holder = getChildViewHolder(view);
        }
    }
```

```
    if (holder == null) {
        // 4) 从 RecycledViewPool 获取 ViewHolder
        holder = getRecycledViewPool().getRecycledView(type);
    }

    if (holder == null) {
        // 缓存全取过了，没有，那只好 create 一个咯
        holder = mAdapter.createViewHolder(RecyclerView.this, type);
    }
}
```

到这复用就分析完了。

缓存

所谓的缓存，就是看一下是怎么样往前面提到的四级缓存添加数据的

- mAttachedScrap 和 mChangedScrap
- mCachedViews
- ViewCacheExtension 前面说了，这个的创建和缓存完全由开发者自己控制，系统未往这里添加数据
- RecycledViewPool

mAttachedScrap 和 mChangedScrap

如果调用了 Adapter 的 notifyXXX 方法，会重新回调到 LayoutManager 的onLayoutChildren 方法里面, 而在 onLayoutChildren 方法里面，会将屏幕上所有的 ViewHolder 回收到 mAttachedScrap 和 mChangedScrap。

```
// 调用链
   LinearLayoutManager.onLayoutChildren(...)
-> LayoutManager.detachAndScrapAttachedViews(recycler)
-> LayoutManager.scrapOrRecycleView(..., view)
-> Recycler.scrapView(view);

private void scrapOrRecycleView(Recycler recycler, int index, View view) {
    final ViewHolder viewHolder = getChildViewHolderInt(view);
    if (viewHolder.isInvalid() && !viewHolder.isRemoved()
            && !mRecyclerView.mAdapter.hasStableIds()) {
        // 后面会讲，缓存到 mCacheViews 和 RecyclerViewPool
        recycler.recycleViewHolderInternal(viewHolder);
    } else {
        // 缓存到 scrap
        recycler.scrapView(view);
    }
}

void scrapView(View view) {
    final ViewHolder holder = getChildViewHolderInt(view);
    if (holder.hasAnyOfTheFlags(ViewHolder.FLAG_REMOVED |
ViewHolder.FLAG_INVALID)
        || !holder.isUpdated() || canReuseUpdatedViewHolder(holder)) {
        // 标记为移除或失效的 || 完全没有改变 || item 无动画或动画不复用
        mAttachedScrap.add(holder);
    } else {
        // 相反就得出放入 mChangedScrap 的条件啦
```

```
            mChangedScrap.add(holder);
        }
    }
```

其实还有一种情况会调用到 scrapView，从 mHiddenViews 获得一个 ViewHolder 的话（发生在支持动画的操作），会先将这个 ViewHolder 从 mHiddenViews 数组里面移除，然后调用:

```
   Recycler.tryGetViewHolderForPositionByDeadline(...)
-> Recycler.getScrapOrHiddenOrCachedHolderForPosition(...)
-> Recycler.scrapView(view)
```

mCacheViews 和 RecyclerViewPool

这两级缓存的代码都在 Recycler 的这个方法里:

```
void recycleViewHolderInternal(ViewHolder holder) {
    if (forceRecycle || holder.isRecyclable()) {
        if(mViewCacheMax > 0
             && !holder.hasAnyOfTheFlags(ViewHolder.FLAG_INVALID
                 | ViewHolder.FLAG_REMOVED
                 | ViewHolder.FLAG_UPDATE
                 | ViewHolder.FLAG_ADAPTER_POSITION_UNKNOWN)) {
          int cachedViewSize = mCachedViews.size();
          if (cachedViewSize >= mViewCacheMax && cachedViewSize > 0) {
              // 1. mCacheViews 满了，最早加入的不要了放 RecyclerViewPool
              recycleCachedViewAt(0);
          }
           mCachedViews.add(targetCacheIndex, holder);
           cached = true;
        }

        if (!cached) {
            // 2. 不能放进 mCacheViews 的放 RecyclerViewPool
            addViewHolderToRecycledViewPool(holder, true);
        }
    }
}

// Recycles a cached view and removes the view from the list
void recycleCachedViewAt(int cachedViewIndex) {
    ViewHolder viewHolder = mCachedViews.get(cachedViewIndex);
    addViewHolderToRecycledViewPool(viewHolder, true);
    mCachedViews.remove(cachedViewIndex);
}
```

在这我们知道 recycleViewHolderInternal 会把 ViewHolder 缓存到 mCacheViews，而不满足加到 mCacheViews 的会缓存到 RecycledViewPool 。那又是什么时候调用的 recycleViewHolderInternal 呢？有以下三种情况:

1. 重新布局，主要是调用 Adapter.notifyDataSetChange 且 Adapter 的 hasStableIds 方法返回为 false 时调用。从这边也可以看出为什么一般情况下 notifyDataSetChange 效率比其它 notifyXXX 方法低（使用二级缓存及优先级更低的缓存），同时也知道了，如果我们设置 Adapter.setHasStableIds(ture) 以及其它相关需要的实现，则可以提高效率（使用一级缓存）
2. 在复用时，从一级缓存里面获取到 ViewHolder，但是此时这个 ViewHolder 已经不符合一级缓存的特点了(比如 Position 失效了，跟 ViewType 对不齐)，就会从一级缓存里面移除这个 ViewHolder，添加到这两级缓存里面

3. 当调用 removeAnimatingView 方法时，如果当前 ViewHolder 被标记为 remove ,会调用 recycleViewHolderInternal 方法来回收对应的 ViewHolder。调用 removeAnimatingView 方法的时机表示当前的 ItemAnimator 已经做完了

总结

到这里，RecyclerView 的缓存复用机制就分析完了，总结一下：

- RecyclerView 的缓存复用机制，主要是通过内部类 Recycler 来实现
- Recycler 有 4 级缓存，每一级的缓存都有各自的作用，会按优先级使用。
- ViewHolder 会从某一级缓存移至其它级别的缓存
- mHideenViews 的存在是为了解决在动画期间进行复用的问题。
- 缓存复用 ViewHolder 时会针对内部不同的状态 (mFlags) 进行相应的处理。

7.33 如何给ListView & RecyclerView加上拉刷新 & 下拉加载更多机制

> 参考：Android RecyclerView实现下拉刷新和上拉加载更多

这道题想考察什么?

考察同学是否对列表下拉刷新与上拉加载的实现熟悉。

考生应该如何回答

ListView

下面我们通过自定义ListView，实现下拉刷新与上拉加载，实现的关键点：

- 为ListView添加头布局和底布局。
- 通过改变头布局的paddingTop值，来控制控件的显示和隐藏
- 根据我们滑动的状态，动态修改头部布局和底部布局。

代码如下：

```
public class CustomRefreshListView extends ListView implements OnScrollListener{
    /**
     * 头布局
     */
    private View headerView;

    /**
     * 头部布局的高度
     */
    private int headerViewHeight;

    /**
     * 头部旋转的图片
     */
    private ImageView iv_arrow;

    /**
     * 头部下拉刷新时状态的描述
     */
```

```
    private TextView tv_state;

    /**
     * 下拉刷新时间的显示控件
     */
    private TextView tv_time;

    /**
     * 底部布局
     */
    private View footerView;

    /**
     * 底部旋转progressbar
     */
    private ProgressBar pb_rotate;

    /**
     * 底部布局的高度
     */
    private int footerViewHeight;

    /**
     * 按下时的Y坐标
     */
    private int downY;

    private final int PULL_REFRESH = 0;//下拉刷新的状态
    private final int RELEASE_REFRESH = 1;//松开刷新的状态
    private final int REFRESHING = 2;//正在刷新的状态

    /**
     * 当前下拉刷新处于的状态
     */
    private int currentState = PULL_REFRESH;

    /**
     * 头部布局在下拉刷新改变时，图标的动画
     */
    private RotateAnimation upAnimation,downAnimation;

    /**
     * 当前是否在加载数据
     */
    private boolean isLoadingMore = false;

    public CustomRefreshListView(Context context) {
        this(context,null);
    }

    public CustomRefreshListView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    private void init(){
        //设置滑动监听
        setOnScrollListener(this);
```

```
        //初始化头布局
        initHeaderView();
        //初始化头布局中图标的旋转动画
        initRotateAnimation();
        //初始化为尾布局
        initFooterView();
    }

    /**
     * 初始化headerView
     */
    private void initHeaderView() {
        headerView = View.inflate(getContext(), R.layout.head_custom_listview,
null);
        iv_arrow = (ImageView) headerView.findViewById(R.id.iv_arrow);
        pb_rotate = (ProgressBar) headerView.findViewById(R.id.pb_rotate);
        tv_state = (TextView) headerView.findViewById(R.id.tv_state);
        tv_time = (TextView) headerView.findViewById(R.id.tv_time);

        //测量headView的高度
        headerView.measure(0, 0);
        //获取高度，并保存
        headerViewHeight = headerView.getMeasuredHeight();
        //设置paddingTop = -headerViewHeight;这样，该控件被隐藏
        headerView.setPadding(0, -headerViewHeight, 0, 0);
        //添加头布局
        addHeaderView(headerView);
    }

    /**
     * 初始化旋转动画
     */
    private void initRotateAnimation() {

        upAnimation = new RotateAnimation(0, -180,
                RotateAnimation.RELATIVE_TO_SELF, 0.5f,
                RotateAnimation.RELATIVE_TO_SELF, 0.5f);
        upAnimation.setDuration(300);
        upAnimation.setFillAfter(true);

        downAnimation = new RotateAnimation(-180, -360,
                RotateAnimation.RELATIVE_TO_SELF, 0.5f,
                RotateAnimation.RELATIVE_TO_SELF, 0.5f);
        downAnimation.setDuration(300);
        downAnimation.setFillAfter(true);
    }

    //初始化底布局，与头布局同理
    private void initFooterView() {
        footerView = View.inflate(getContext(), R.layout.foot_custom_listview,
null);
        footerView.measure(0, 0);
        footerViewHeight = footerView.getMeasuredHeight();
        footerView.setPadding(0, -footerViewHeight, 0, 0);
        addFooterView(footerView);
    }

    @Override
```

```
    public boolean onTouchEvent(MotionEvent ev) {
        switch (ev.getAction()) {
        case MotionEvent.ACTION_DOWN:
            //获取按下时y坐标
            downY = (int) ev.getY();
            break;
        case MotionEvent.ACTION_MOVE:

            if(currentState==REFRESHING){
                //如果当前处在滑动状态，则不做处理
                break;
            }
            //手指滑动偏移量
            int deltaY = (int) (ev.getY() - downY);

            //获取新的padding值
            int paddingTop = -headerViewHeight + deltaY;
            if(paddingTop>-headerViewHeight && getFirstVisiblePosition()==0){
                //向下滑，且处于顶部，设置padding值，该方法实现了顶布局慢慢滑动显现
                headerView.setPadding(0, paddingTop, 0, 0);

                if(paddingTop>=0 && currentState==PULL_REFRESH){
                    //从下拉刷新进入松开刷新状态
                    currentState = RELEASE_REFRESH;
                    //刷新头布局
                    refreshHeaderView();
                }else if (paddingTop<0 && currentState==RELEASE_REFRESH) {
                    //进入下拉刷新状态
                    currentState = PULL_REFRESH;
                    refreshHeaderView();
                }
                return true;//拦截TouchMove，不让listview处理该次move事件,会造成
listview无法滑动
            }
            break;
        case MotionEvent.ACTION_UP:
            if(currentState==PULL_REFRESH){
                //仍处于下拉刷新状态，未滑动一定距离，不加载数据，隐藏headView
                headerView.setPadding(0, -headerViewHeight, 0, 0);
            }else if (currentState==RELEASE_REFRESH) {
                //滑倒一定距离，显示无padding值得headcView
                headerView.setPadding(0, 0, 0, 0);
                //设置状态为刷新
                currentState = REFRESHING;

                //刷新头部布局
                refreshHeaderView();

                if(listener!=null){
                    //接口回调加载数据
                    listener.onPullRefresh();
                }
            }
            break;
        }
        return super.onTouchEvent(ev);
    }
```

```
    /**
     * 根据currentState来更新headerView
     */
    private void refreshHeaderView(){
        switch (currentState) {
        case PULL_REFRESH:
            tv_state.setText("下拉刷新");
            iv_arrow.startAnimation(downAnimation);
            break;
        case RELEASE_REFRESH:
            tv_state.setText("松开刷新");
            iv_arrow.startAnimation(upAnimation);
            break;
        case REFRESHING:
            iv_arrow.clearAnimation();//因为向上的旋转动画有可能没有执行完
            iv_arrow.setVisibility(View.INVISIBLE);
            pb_rotate.setVisibility(View.VISIBLE);
            tv_state.setText("正在刷新...");
            break;
        }
    }

    /**
     * 完成刷新操作，重置状态,在你获取完数据并更新完adater之后，去在UI线程中调用该方法
     */
    public void completeRefresh(){
        if(isLoadingMore){
            //重置footerView状态
            footerView.setPadding(0, -footerViewHeight, 0, 0);
            isLoadingMore = false;
        }else {
            //重置headerView状态
            headerView.setPadding(0, -headerViewHeight, 0, 0);
            currentState = PULL_REFRESH;
            pb_rotate.setVisibility(View.INVISIBLE);
            iv_arrow.setVisibility(View.VISIBLE);
            tv_state.setText("下拉刷新");
            tv_time.setText("最后刷新：+getCurrentTime());
        }
    }

    /**
     * 获取当前系统时间，并格式化
     * @return
     */
    private String getCurrentTime(){
        SimpleDateFormat format = new SimpleDateFormat("yy-MM-dd HH:mm:ss");
        return format.format(new Date());
    }

    private OnRefreshListener listener;
    public void setOnRefreshListener(OnRefreshListener listener){
        this.listener = listener;
    }
    public interface OnRefreshListener{
        void onPullRefresh();
        void onLoadingMore();
    }
```

```
    /**
     * SCROLL_STATE_IDLE:闲置状态，就是手指松开
     * SCROLL_STATE_TOUCH_SCROLL：手指触摸滑动，就是按着来滑动
     * SCROLL_STATE_FLING：快速滑动后松开
     */
    @Override
    public void onScrollStateChanged(AbsListView view, int scrollState) {
        if(scrollState==OnScrollListener.SCROLL_STATE_IDLE
                && getLastVisiblePosition()==(getCount()-1) &&!isLoadingMore){
            isLoadingMore = true;

            footerView.setPadding(0, 0, 0, 0);//显示出footerView
            setSelection(getCount());//让listview最后一条显示出来，在页面完全显示出底布
局


            if(listener!=null){
                listener.onLoadingMore();
            }
        }
    }

    @Override
    public void onScroll(AbsListView view, int firstVisibleItem,
            int visibleItemCount, int totalItemCount) {
    }
}
```

1.下拉刷新的实现逻辑如下：
下拉刷新是通过设置setOnTouchListener()方法，监听触摸事件，通过手指滑动的不同处理实现相应逻辑。
实现比较复杂，分为了三个情况，初始状态（显示下拉刷新），释放刷新状态，刷新状态。
其中下拉刷新状态和释放状态的改变，是由于手指滑动的不同距离，是在MotionEvent.ACTION_MOVE中进行判断，该判断不处理任何数据逻辑，只是根据手指滑动的偏移量该表UI的显示。
刷新状态的判断是在MotionEvent.ACTION_UP手指抬起时判断的。这很好理解，因为最终下拉刷新是否加载数据的确定，是由我们手指离开屏幕时与初始值的偏移量确定的。如果我们的偏移量小于了头布局的高度，代表不刷新，继续隐藏头布局。如果偏移量大于了头布局的高度，代表刷新，修改UI，同时通过接口回调，让其持有者进行加载数据。

2.上拉加载的实现逻辑如下：
上拉加载和下拉刷新不同，他的实现较为简单，我们通过ListView的滚动监听进行处理相应逻辑。即setOnScrollListener(this)。
该方法需要实现两个回调方法：

- public void onScroll(AbsListView view, int firstVisibleItem,int visibleItemCount, int totalItemCount):滚动监听的调用。
- public void onScrollStateChanged(AbsListView view, int scrollState)：滑动状态改变的回调。其中scrollState为回调的状态，可能值为
    - SCROLL_STATE_IDLE:闲置状态，手指松开后的状态回调
    - SCROLL_STATE_TOUCH_SCROLL：手指触摸滑动的状态回调
    - SCROLL_STATE_FLING：手指松开后惯性滑动的回调

我们在onScrollStateChanged中进行判断，主要判断一下条件;

- 是否是停止状态
- 是否滑倒最后

- 是否正在加载数据

如果符合条件，则开始加载数据，通过接口回调。

RecyclerView

使用官方的刷新控件SwipeRefreshLayout来实现下拉刷新，当RecyclerView滑到底部实现下拉加载（进度条效果用RecyclerView加载一个布局实现）
需要完成控件的下拉监听和上拉监听，其中，下拉监听通过SwipRefreshLayout的setOnRefreshListener()方法监听，而上拉刷新，需要通过监听列表的滚动，当列表滚动到底部时触发事件，具体代码如下：

```
public class MainActivity extends AppCompatActivity implements
SwipeRefreshLayout.OnRefreshListener {
    private SwipeRefreshLayout refreshLayout;
    private RecyclerView recyclerView;
    private LinearLayoutManager layoutManager;

    private RecyclerAdapter mAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initViews();
    }

    private void initViews() {
        refreshLayout = (SwipeRefreshLayout) findViewById(R.id.refresh_layout);
        recyclerView = (RecyclerView) findViewById(R.id.recycler_list);
        layoutManager = new LinearLayoutManager(this);

        // 设置刷新时进度条颜色，最多四种
        refreshLayout.setColorSchemeResources(R.color.colorAccent,
R.color.colorPrimary);
        // 设置监听，需要重写onRefresh()方法，顶部下拉时会调用这个方法
        refreshLayout.setOnRefreshListener(this);

        mAdapter = new RecyclerAdapter();//自定义的适配器
        recyclerView.setAdapter(mAdapter);
        recyclerView.setLayoutManager(layoutManager);
        recyclerView.addOnScrollListener(new OnRecyclerScrollListener());
    }

    /**
      * 用于下拉刷新
      */
    @Override
    public void onRefresh() {
    }

    /**
      * 用于上拉加载更多
      */
    public class OnRecyclerScrollListener extends RecyclerView.OnScrollListener
{
        int lastVisibleItem = 0;
```

```
        @Override
        public void onScrollStateChanged(RecyclerView recyclerView, int
newState) {
            super.onScrollStateChanged(recyclerView, newState);

            if (mAdapter != null && newState == RecyclerView.SCROLL_STATE_IDLE
                && lastVisibleItem + 1 == mAdapter.getItemCount()) {
                //滚动到底部了，可以进行数据加载等操作
            }
        }

        @Override
        public void onScrolled(RecyclerView recyclerView, int dx, int dy) {
            super.onScrolled(recyclerView, dx, dy);
            lastVisibleItem = layoutManager.findLastVisibleItemPosition();
        }
    }
}
```

下面是实现上拉时进度条转动的效果

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
                android:layout_width="match_parent"
                android:layout_height="match_parent">

    <TextView
              android:id="@+id/tv_item_footer_load_more"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:layout_margin="16dp"
              android:gravity="center"
              android:text="上拉加载更多"
              />

    <ProgressBar
                 android:id="@+id/pb_item_footer_loading"
                 android:layout_width="match_parent"
                 android:layout_height="wrap_content"
                 android:layout_margin="16dp"
                 android:visibility="gone"/>
</RelativeLayout>
```

```
public class RecyclerAdapter extends RecyclerView.Adapter<ViewHolder> {
    private static final int TYPE_CONTENT = 0;
    private static final int TYPE_FOOTER = 1;

    private ArrayList<DataBean> dataList;

    private ProgressBar pbLoading;
    private TextView tvLoadMore;

    public RecyclerAdapter() {
        dataList = new ArrayList<>();
    }
```

```
    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        if (viewType == TYPE_CONTENT) {
            return new
ContentViewHolder(LayoutInflater.from(parent.getContext())
                                        .inflate(R.layout.item_list_content,
parent, false));
        } else if (viewType == TYPE_FOOTER) {//加载进度条的布局
            return new FooterViewHolder(LayoutInflater.from(parent.getContext())
                                       .inflate(R.layout.item_list_footer,
parent, false));
        }
        return null;
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        int type = getItemViewType(position);
        if (type == TYPE_CONTENT) {
            DataBean bean = dataList.get(position);
            ((ContentViewHolder) holder).tvId.setText("" + bean.getId());
            ((ContentViewHolder) holder).tvName.setText(bean.getName());
        } else if (type == TYPE_FOOTER) {
            pbLoading = ((FooterViewHolder) holder).pbLoading;
            tvLoadMore = ((FooterViewHolder) holder).tvLoadMore;
        }
    }

    /**
      * 获取数据集加上一个footer的数量
      */
    @Override
    public int getItemCount() {
        return dataList.size() + 1;
    }


    @Override
    public int getItemViewType(int position) {
        if (position + 1 == getItemCount()) {
            return TYPE_FOOTER;
        } else {
            return TYPE_CONTENT;
        }
    }

    /**
      * 获取数据集的大小
      */
    public int getListSize() {
        return dataList.size();
    }



    /**
      * 内容的ViewHolder
```

```
     */
    public static class ContentViewHolder extends ViewHolder {
        private TextView tvId, tvName;

        public ContentViewHolder(View itemView) {
            super(itemView);
            tvId = (TextView) itemView.findViewById(R.id.tv_item_id);
            tvName = (TextView) itemView.findViewById(R.id.tv_item_name);
        }
    }

    /**
     * footer的ViewHolder
     */
    public static class FooterViewHolder extends ViewHolder {
        private TextView tvLoadMore;
        private ProgressBar pbLoading;

        public FooterViewHolder(View itemView) {
            super(itemView);
            tvLoadMore = (TextView)
itemView.findViewById(R.id.tv_item_footer_load_more);
            pbLoading = (ProgressBar)
itemView.findViewById(R.id.pb_item_footer_loading);
        }
    }

    /**
     * 显示正在加载的进度条，滑动到底部时，调用该方法，上拉就显示进度条，隐藏"上拉加载更多"
     */
    public void showLoading() {
        if (pbLoading != null && tvLoadMore != null) {
            pbLoading.setVisibility(View.VISIBLE);
            tvLoadMore.setVisibility(View.GONE);
        }
    }

    /**
     * 显示上拉加载的文字，当数据加载完毕，调用该方法，隐藏进度条，显示“上拉加载更多”
     */
    public void showLoadMore() {
        if (pbLoading != null && tvLoadMore != null) {
            pbLoading.setVisibility(View.GONE);
            tvLoadMore.setVisibility(View.VISIBLE);
        }
    }
}
```

7.34 如何对ListView & RecycleView进行局部刷新的?

这道题想考察什么?

考察同学是否对列表局部刷新清楚。

考生应该如何回答

ListView

我们要进行单条刷新就要手动调用这个方法。

```
public View getView(int position, View convertView, ViewGroup parent)
```

google 2011年开发者大会上提出了ListView的单条刷新方法，代码如下：

```
private void updateSingleRow(ListView listView, long id) {

        if (listView != null) {
            int start = listView.getFirstVisiblePosition();
            for (int i = start, j = listView.getLastVisiblePosition(); i <= j;
i++)
                if (id == ((Messages) listView.getItemAtPosition(i)).getId()) {
                    View view = listView.getChildAt(i - start);
                    getView(i, view, listView);
                    break;
                }
        }
    }
```

RecyclerView

RecyclerView 局部刷新一般使用 notifyItemChanged(position); 方法。
但是这个方法有个坑，同样会刷新item中所有东西，并且在不断刷新的时候会执行刷新Item的动画，导致滑动的时候会错开一下，还可能会造成图片重新加载或者不必要的消耗。
所以下面我们看下源码如何解决这个问题。直接找到 RecyclerView 的 notifyItemChanged(position) 方法，如下：

```
public final void notifyItemChanged(int position) {
    mObservable.notifyItemRangeChanged(position, 1);
}
```

这个方法里执行了notifyItemRangeChanged方法，继续跟踪

```
public void notifyItemRangeChanged(int positionStart, int itemCount) {
    notifyItemRangeChanged(positionStart, itemCount, null);
}
```

发现这里执行了另一个重载方法
最后一个参数为null？继续跟进

```
public void notifyItemRangeChanged(int positionStart, int itemCount,
        @Nullable Object payload) {
    for (int i = mObservers.size() - 1; i >= 0; i--) {
        mObservers.get(i).onItemRangeChanged(positionStart, itemCount, payload);
    }
}
```

发现这里的参数为Object payload，然后我看到了notifyItemChanged的另一个重载方法，这里也有一个Object payload参数，看一下源码：

```
public final void notifyItemChanged(int position, @Nullable Object payload) {
    mObservable.notifyItemRangeChanged(position, 1, payload);
}
```

payload 的解释为：如果为null，则刷新item全部内容，言外之意就是不为空就可以局部刷新了！
继续从payload跟踪，发现在RecyclerView中有另一个onBindViewHolder的方法，多了一个参数，payload。

```
public void onBindViewHolder(@NonNull VH holder, int position,
        @NonNull List<Object> payloads) {
    onBindViewHolder(holder, position);
}
```

发现它调用了另一个重载方法，而另一个重载方法就是我们在写adapter中抽象的方法，那我们就可以直接从这里入手了。
1.重写onBindViewHolder(VH holder, int position, List payloads)这个方法

```
@Override
public void onBindViewHolder(MyViewHolder holder, final int position,
List<Object> payloads) {
    super.onBindViewHolder(holder, position, payloads);
    if (payloads.isEmpty()){
        //全部刷新
    }else {
        //局部刷新
    }
}
```

2.执行两个参数的notifyItemChanged，第二个参数只要不让它为空，随便什么都行，这样就可以实现只刷新item中某个控件了。

7.35 ScrollView下嵌套一个RecycleView通常会出现什么问题?

这道题想考察什么?

考察同学对ScrollView和RecyclerView嵌套的理解。

考生应该如何回答

ScrollView下嵌套一个RecycleView通常会导致如下几个问题

- 页面滑动卡顿
- ScrollView高度显示不正常
- RecyclerView内容显示不全

滑动卡顿解决方案

若只存在滑动卡顿这一问题，可以采用如下两种简单方式快速解决
方式1：利用RecyclerView内部方法

```
recyclerView.setHasFixedSize(true);
recyclerView.setNestedScrollingEnabled(false);
```

其中，setHasFixedSize(true)方法使得RecyclerView能够固定自身size不受adapter变化的影响；而setNestedScrollingeEnabled(false)方法则是进一步调用了RecyclerView内部NestedScrollingChildHelper对象的setNestedScrollingeEnabled(false)方法，如下

```
public void setNestedScrollingEnabled(boolean enabled) {
    getScrollingChildHelper().setNestedScrollingEnabled(enabled);
}
```

进而，NestedScrollingChildHelper对象通过该方法关闭RecyclerView的嵌套滑动特性，如下

```
public void setNestedScrollingEnabled(boolean enabled) {
    if (mIsNestedScrollingEnabled) {
        ViewCompat.stopNestedScroll(mView);
    }
    mIsNestedScrollingEnabled = enabled;
}
```

如此一来，限制了RecyclerView自身的滑动，整个页面滑动仅依靠ScrollView实现，即可解决滑动卡顿的问题

方式2：重写LayoutManager

```
LinearLayoutManager linearLayoutManager = new LinearLayoutManager(this) {
    @Override
    public boolean canScrollVertically() {
        return false;
    }
};
```

这一方式使得RecyclerView的垂直滑动始终返回false，其目的同样是为了限制自身的滑动

综合解决方案

若是需要综合解决上述三个问题，则可以采用如下几种方式。
方式1：插入LinearLayout/RelativeLayout
在原有布局中插入一层LinearLayout/RelativeLayout，形成如下布局

```
ScrollView
    LinearLayout/RelativeLayout
        RecyclerView
```

方式2：重写LayoutManager
该方法的核心思想在于通过重写LayoutManager中的onMeasure()方法，即

```
@Override
public void onMeasure(RecyclerView.Recycler recycler, RecyclerView.State state,
int widthSpec, int heightSpec) {
    super.onMeasure(recycler, state, widthSpec, heightSpec);
}
```

重新实现RecyclerView高度的计算，使得其能够在ScrollView中表现出正确的高度.

方式3：重写ScrollView
该方法的核心思想在于通过重写ScrollView的onInterceptTouchEvent(MotionEvent ev)方法，拦截滑动事件，使得滑动事件能够直接传递给RecyclerView，具体重写方式可参考如下

```
public class RecyclerScrollView extends ScrollView {
    private int slop;
    private int touch;

    public RecyclerScrollView(Context context) {
        super(context);
        setSlop(context);
    }

    public RecyclerScrollView(Context context, AttributeSet attrs) {
        super(context, attrs);
        setSlop(context);
    }

    public RecyclerScrollView(Context context, AttributeSet attrs, int
defStyleAttr) {
        super(context, attrs, defStyleAttr);
        setSlop(context);
    }

    /**
     * 是否intercept当前的触摸事件
     * @param ev 触摸事件
     * @return true：调用onMotionEvent()方法，并完成滑动操作
     */
    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        switch (ev.getAction()) {
            case MotionEvent.ACTION_DOWN:
                //  保存当前touch的纵坐标值
                touch = (int) ev.getRawY();
```

```
                break;
            case MotionEvent.ACTION_MOVE:
                //  滑动距离大于slop值时，返回true
                if (Math.abs((int) ev.getRawY() - touch) > slop) return true;
                break;
        }

        return super.onInterceptTouchEvent(ev);
    }

    /**
     * 获取相应context的touch slop值（即在用户滑动之前，能够滑动的以像素为单位的距离）
     * @param context ScrollView对应的context
     */
    private void setSlop(Context context) {
        slop = ViewConfiguration.get(context).getScaledTouchSlop();
    }
}
```

事实上，尽管我们能够采用多种方式解决ScrollView嵌套RecyclerView所产生的一系列问题，但由于上述解决方式均会使得RecyclerView在页面加载过程中一次性显示所有内容，因此当RecyclerView下的条目过多时，将会对影响整个应用的运行效率。基于此，在这种情况下我们应当尽量避免采用ScrollView嵌套RecyclerView的布局方式。

7.36 一个ListView或者一个RecyclerView在显示新闻数据的时候，出现图片错位，可能的原因有哪些 & 如何解决?

这道题想考察什么?

考察同学对ListView和RecyclerView的错位问题是否知道。

考生应该如何回答

图片错位

什么是显示错位?
例如本来应该显示在item1的图片，由于上下滑动，导致显示在item10上面，这就是显示错位。
ListView和RecyclerView错位的原因是一样的，都是因为ListView和RecyclerView有复用的功能，才导致的错位显示。

如何解决

方案一：ListView和RecyclerView都可以，直接设置一个tag，并预设一张图片。

```
// 给 ImageView 设置一个 tag
holder.img.setTag(imgUrl);
// 预设一个图片
holder.img.setImageResource(R.drawable.ic_launcher);

// 通过 tag 来防止图片错位
if (imageView.getTag() != null && imageView.getTag().equals(imageUrl)) {
    imageView.setImageBitmap(result);
}
```

方案二：RecyclerView的解决方案
重写下面方法，类似方案一的tag一样。

```
@Override
public long getItemId(int position) {
    return position;
}
```

7.37 Requestlayout，onlayout，onDraw，DrawChild区别与联系

这道题想考察什么?

考察同学对这几个方法背后的执行原理是否熟悉。

考生应该如何回答

RequestLayout()方法：会导致调用 Measure()方法和 layout()。将会根据标志位判断是否 需要 onDraw();
onLayout()：摆放 viewGroup 里面的子控件，只有自定义ViewGroup需要重写。
onDraw()：绘制视图本身； （ViewGroup 还需要绘制里面的所有子控件）
drawChild(): 重新回调每一个子视图的 draw 方法 。 child.draw(canvas, this, drawingTime);

7.38 如何优化自定义View

这道题想考察什么?

考察同学对自定义View的注意点是否熟悉。

考生应该如何回答

降低刷新频率

为了提高view的运行速度，减少来自于频繁调用的程序的不必要的代码。从onDraw()方法开始调用，这会给你带来最好的回报。特别地，在onDraw()方法中你应该减少冗余代码，冗余代码会带来使你view不连贯的垃圾回收。初始化的冗余对象，或者动画之间的，在动画运行时，永远都不会有所贡献。
加之为了使onDraw()方法更有依赖性，你应该尽可能的不要频繁的调用它。大部分时候调用 onDraw()方法就是调用invalidate()的结果，所以减少不必要的调用invalidate()方法。有可能的，调用四种参数不同类型的invalidate()，而不是调用无参的版本。无参变量需要刷新整个view，而四种参数类型的变量只需刷新指定部分的view.这种高效的调用更加接近需求，也能减少落在矩形屏幕外的不必 要刷新的页面。
另外一个非常耗时的操作是请求layout。任何时候执行requestLayout()，会使得Android UI系统去遍历整个View的层级来计算出每一个view的大小。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持View的层级是扁平化的，这样对提高效率很有帮助。 如果你有一个复杂的UI，你应该写一个自定义的ViewGroup类来表现它的布局。不同于内置的 view类，你的自定义view能关于尺寸和它子控件的形状做出应用特定的假想，同时避免通过它子类来 计算尺寸。这圆图例子显示怎样作为自定义view一部分来继承ViewGroup类，圆图有它子view类， 但是重来都不测量他们。相反地，它直接地通过自己自定义的布局算法来设定他们的尺寸大小。
如果你有一个复杂的UI，你应该考虑写一个自定义的ViewGroup来执行他的layout操作。与内置的view不同，自定义的view可以使得程序仅仅测量这一部分，这避免了遍历整个view的层级结构来计算大小。这个PieChart 例子展示了如何继承ViewGroup作为自定义view的一部分。PieChart 有子views，但是它从来不测量它们。而是根据他自身的layout法则，直接设置它们的大小。

使用硬件加速

作为Android3.0，Android2D图表系统可以通过大部分新的Android装置自带GPU（图表处理单元）来增加，对于许多应用程序 来说，GPU硬件加速度能带来巨大的性能增加，但是对于每一个应用来讲，并不都是正确的选择。Android框架层更好地为你提供了控制应用程序部分硬件 是否增加的能力。
怎样在你的应用，活动，或者窗体级别中使用加速度类，请查阅Android开发者指南中的Hardware Acceleration类。注意到在开发者指南中的附加说明，你必须在你的AndroidManifest.xml 文件中的中将应用目标API设置到11或者更高的级别。
一旦你使用硬件加速度类，你可能没有看到性能的增长，手机GPUs非常擅长某些任务，例如测量，翻转，和平移位图类的图片。特别地，他们不擅长其他的任务，例如画直线和曲线。为了利用GPU加速度类，你应该增加GPU擅长的操作数量，和减少GPU不擅长的操作数量。

减少过度渲染

当UI之间有重叠的时候，虽然后面的UI用户看不到，但是还是会绘制，这部分的绘制就是过度渲染。
一般通过Canvas.clipXxx 方法来裁剪区域，让看不到的地放不绘制，从而达到减少过度渲染的目的。

初始化时创建对象

不要在onDraw方法内创建绘制对象，一般都在构造函数里面初始化对象；

```
@Override
protected void onDraw(Canvas canvas) {
    if (getDrawable() == null) {
        return;
    }
    setUpShader();
    canvas.setDrawFilter(new PaintFlagsDrawFilter(0, Paint.ANTI_ALIAS_FLAG |
Paint.FILTER_BITMAP_FLAG));
    if (mType == TYPE_ROUND) {
        canvas.drawRoundRect(mRoundRect, mBorderRadius, mBorderRadius,
mBitmapPaint);
    } else {
        canvas.drawCircle(mRadius, mRadius, mRadius, mBitmapPaint);
    }
}
```

状态的存储与恢复

如果内存不足，而恰好我们的Activity置于后台，不幸被重启，或者用户旋转屏幕造成Activity重启，我们的View应该也能尽可能的去保存自己的属性。

```
@Override
protected Parcelable onSaveInstanceState() {
    Bundle bundle = new Bundle();
    bundle.putParcelable(STATE_INSTANCE, super.onSaveInstanceState());
    bundle.putInt(STATE_TYPE, mType);
    bundle.putInt(STATE_BORDER_RADIUS, mBorderRadius);
    return bundle;
}

@Override
protected void onRestoreInstanceState(Parcelable state) {
    if (state instanceof Bundle) {
        Bundle bundle = (Bundle) state;
        super.onRestoreInstanceState(((Bundle)
state).getParcelable(STATE_INSTANCE));
        this.mType = bundle.getInt(STATE_TYPE);
        this.mBorderRadius = bundle.getInt(STATE_BORDER_RADIUS);
    } else {
        super.onRestoreInstanceState(state);
    }
}
```

7.39 Android属性动画实现原理，补间动画实现原理 ~colin

这道题想考察什么?

1. 是否了解动画的实现原理?

考察的知识点

1. 属性动画内部实现
2. 补间动画内部实现

考生应该如何回答

1. 首先回答下安卓的动画种类，一共分为三种

- 属性动画，动态的改变属性产生动画效果；
- 补间动画，对View的平移，旋转，缩放，透明产生效果；
- 帧动画，由数张图片连续播放产生的动画效果；

1. 属性动画的原理，是在一定的时间间隔内，不停的对值进行改变并将该值赋给对象的属性，从而实现对象上特定属性的动画效果。

   主要有三种方式：

- ValueAnimator.ofInt(int values)：估值器是整型估值器IntEaluator；
- ValueAnimator.ofFloat(float values):估值器是浮点型估值器FloatEaluator；
- ValueAnimator.ofObject(ObjectEvaluator, start, end):将初始值以对象的形式过渡到结束值，通过操作对象实现动画效果，需要实现Interpolator接口。

1. 补间动画的原理，对View的平移，旋转，缩放，透明的操作，从而产生的动画效果。需要特别说明的是补间动画只调整了View在Canvas上的显示，并没有真实的改变View原来的位置。

- 调用View的startAnimation()方法出发动画事件，invalidate(true)会触发draw方法对View进行重新绘制。

```
public void startAnimation(Animation animation) {
        animation.setStartTime(Animation.START_ON_FIRST_FRAME);
        setAnimation(animation);
        invalidateParentCaches();
        invalidate(true);
    }
```

- 把当前画布中View放置于矩阵中，然后通过矩阵变换对View在画布中影像进行改变，Transformation 定义的矩阵对象

```
 boolean draw(Canvas canvas, ViewGroup parent, long drawingTime) {
        ...
        if ((parentFlags & ViewGroup.FLAG_CLEAR_TRANSFORMATION) != 0) {
            parent.getChildTransformation().clear();
            parent.mGroupFlags &= ~ViewGroup.FLAG_CLEAR_TRANSFORMATION;
        }
        ...
        //矩阵，把当前的View放置到矩阵中
        Transformation transformToApply = null;
        //对View进行平移，旋转，缩放，透明等事件处理
        if (transformToApply != null
                || alpha < 1
                || !hasIdentityMatrix()
                || (mPrivateFlags3 & PFLAG3_VIEW_IS_ANIMATING_ALPHA) != 0) {
            if (transformToApply != null || !childHasIdentityMatrix) {
```

```
                int transX = 0;
                int transY = 0;

                if (offsetForScroll) {
                    transX = -sx;
                    transY = -sy;
                }

                if (transformToApply != null) {
                    if (concatMatrix) {
                        if (drawingWithRenderNode) {

 renderNode.setAnimationMatrix(transformToApply.getMatrix());
                        } else {
                            // Undo the scroll translation, apply the
transformation matrix,
                            // then redo the scroll translate to get the correct
result.
                            //画布的转化
                            canvas.translate(-transX, -transY);
                            //获取矩阵
                            canvas.concat(transformToApply.getMatrix());
                            //恢复
                            canvas.translate(transX, transY);
                        }
                        parent.mGroupFlags |=
ViewGroup.FLAG_CLEAR_TRANSFORMATION;
                    }

                    float transformAlpha = transformToApply.getAlpha();
                    if (transformAlpha < 1) {
                        alpha *= transformAlpha;
                        parent.mGroupFlags |=
ViewGroup.FLAG_CLEAR_TRANSFORMATION;
                    }
                }

                if (!childHasIdentityMatrix && !drawingWithRenderNode) {
                    //画布的转化
                    canvas.translate(-transX, -transY);
                    //获取矩阵
                    canvas.concat(getMatrix());
                    //恢复
                    canvas.translate(transX, transY);
                }
            }

            // Deal with alpha if it is or used to be <1
            if (alpha < 1 || (mPrivateFlags3 & PFLAG3_VIEW_IS_ANIMATING_ALPHA)
!= 0) {
                if (alpha < 1) {
                    mPrivateFlags3 |= PFLAG3_VIEW_IS_ANIMATING_ALPHA;
                } else {
                    mPrivateFlags3 &= ~PFLAG3_VIEW_IS_ANIMATING_ALPHA;
                }
                parent.mGroupFlags |= ViewGroup.FLAG_CLEAR_TRANSFORMATION;
                if (!drawingWithDrawingCache) {
                    final int multipliedAlpha = (int) (255 * alpha);
```

```
                    if (!onSetAlpha(multipliedAlpha)) {
                        if (drawingWithRenderNode) {
                            renderNode.setAlpha(alpha * getAlpha() *
getTransitionAlpha());
                        } else if (layerType == LAYER_TYPE_NONE) {
                            canvas.saveLayerAlpha(sx, sy, sx + getWidth(), sy +
getHeight(),
                                    multipliedAlpha);
                        }
                    } else {
                        // Alpha is handled by the child directly, clobber the
layer's alpha
                        mPrivateFlags |= PFLAG_ALPHA_SET;
                    }
                }
            }
        } else if ((mPrivateFlags & PFLAG_ALPHA_SET) == PFLAG_ALPHA_SET) {
            onSetAlpha(255);
            mPrivateFlags &= ~PFLAG_ALPHA_SET;
        }

        if (!drawingWithRenderNode) {
            // apply clips directly, since RenderNode won't do it for this draw
            if ((parentFlags & ViewGroup.FLAG_CLIP_CHILDREN) != 0 && cache ==
null) {
                if (offsetForScroll) {
                    canvas.clipRect(sx, sy, sx + getWidth(), sy + getHeight());
                } else {
                    if (!scalingRequired || cache == null) {
                        canvas.clipRect(0, 0, getWidth(), getHeight());
                    } else {
                        canvas.clipRect(0, 0, cache.getWidth(),
cache.getHeight());
                    }
                }
            }

            if (mClipBounds != null) {
                // clip bounds ignore scroll
                canvas.clipRect(mClipBounds);
            }
        }
        ...
        return more;
    }
```