第9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

9.1 View的绘制原理

背景

对于Android开发，在面试的时候，经常会被问到，说一说View的绘制流程？我也经常问面试者，View的绘制流程.

对于3年以上的开发人员来说，就知道onMeasure/onLayout/onDraw基本，知道他们是干些什么的，这样就够了吗？

如果你来我们公司，我是你的面试官，可能我会考察你这三年都干了什么,对于View你都知道些什么，会问一些更细节的问题，比如LinearLayout的onMeasure,onLayout过程?他们都是什么时候被发起的,执行顺序是什么？

如果以上问题你都知道,可能你进来我们公司就差不多了，可能我会考察你draw的 canvas是哪里来的,他是怎么被创建显示到屏幕上呢？看看你的深度有多少？

对于现在的移动开发市场逐渐趋向成熟，趋向饱和，很多不缺人的公司，都需要高级程序员.在说大家也都知道，面试要造飞机大炮，进去后拧螺丝,对于一个3年或者5年以上Android开发不稍微了解一些Android深一点的东西,不是很好混.扯了这么多没用的东西，还是回到今天正题，Android的绘图原理浅析.

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，先学习对应享学课堂相关的知识章节。

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); // 核心代码
            } 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);
    }
```

addView 会将函数转交给WindowManagerGlobal 类中的addView，我们继续看下面的代码：

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) {
            ...
        }
    }
}
```

在WindowManagerGlobal中的addView里面会先创建一个 ViewRootImpl对象，ViewRootImpl大家可以理解为管理viewTree的根布局的一个对象，甚至可以狭义的理解为viewTree根布局的管理者，具体的解析大家可以参考7.8章节关于ViewRootImpl的理解。从上面的代码我们看到WindowManagerGlobal中addView会调用viewRootImpl的setView函数。

1.4 由1.3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();
      ...
}
```

通过上面的代码我们发现，setView中调用了一个非常重要的代码，那就是requestLayout()函数，这个函数是view系统体系中非常重要的一个函数，可以说是viewTree 绘制管理的真正启动，具体的代码，我们看下面的调用流程：

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

上面的代码会调用 scheduleTraversals(),那么这个函数是干什么的呢？继续往下看：

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

通过scheduleTraversals(),我们发现它设置了一个回调函数mTraversalRunnable，回调函数里面又有什么呢？

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

mTraversalRunnable其实就是一个 runnable，所以，她的关键是看run函数中所调用的doTraversal()。

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

在doTraversal()里面出现了一个非常重要的函数performTraversals，为什么说它非常重要呢？

重点来了看下面的代码调用：

```
 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();
            ...
    }
```

在上面代码中，有三个非常重要的函数performMeasure、performLayout、performDraw，相信大家通过名字不难发现这几个函数就是执行onMeasure、onLayout，onDraw的关键入口。那么他们是怎么触发到具体view 的onMeasure，onLyout，onDraw上面的呢？我们下面以performMeasure 为例进行讲解。

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

上面的mView就是DecorView，也就是根view，当调用measure的时候，会调用view 的 measure函数，measure函数

2.绘制流程

2.1 measure过程

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宽高，确定自己宽高。

```
 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);//1
            ...
        }
     }
     ...省略代码段
     setMeasuredDimension(resolveSizeAndState(maxWidth, widthMeasureSpec,
childState),
           resolveSizeAndState(maxHeight, heightMeasureSpec,
                       childState << MEASURED_HEIGHT_STATE_SHIFT));//2
        ...
     }
}
```

上面的代码有两处非常重要的代码，其中代码1处是度量孩子的宽高， 具体的代码我们可以看接下来的解释，但是在这个函数里面还有一处非常重要的代码，那就是代码2，代码2 就是确定当前onMeasure的view的宽高确定的，那么当前的view是怎么进行确定的呢？它就是各种layout 自身的布局算法了,比如FrameLayout，LinearLayout，RelativeLayout，在它们上面摆放的子view以一个什么方式排列，排列完成后，需要多高多宽，这些就是文中标注了省略代码段的代码计算的过程（这个过程就是各个layout的核心），最终计算的结果就是得到当前View的宽高，然后调用setMeasuredDimension将得到的宽高保存起来。

上面代码1处所调用的measureChildWithMargins请看下面的解析。

我们先来关注测量子View即measureChildWithMargins，大概的意思是：先获到宽高的measureSpec，然后再基于这个measureSpec对view进行measure，从而可以将度量动作分发给当前这个child的孩子。

```
 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。

注意：MeasureSpec 是什么？怎么得到？这个也是一个面试常问的点，大家感兴趣可以去查找书籍找到答案，或者可以去享学课堂找到对应的课程进行学习。

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。

小结

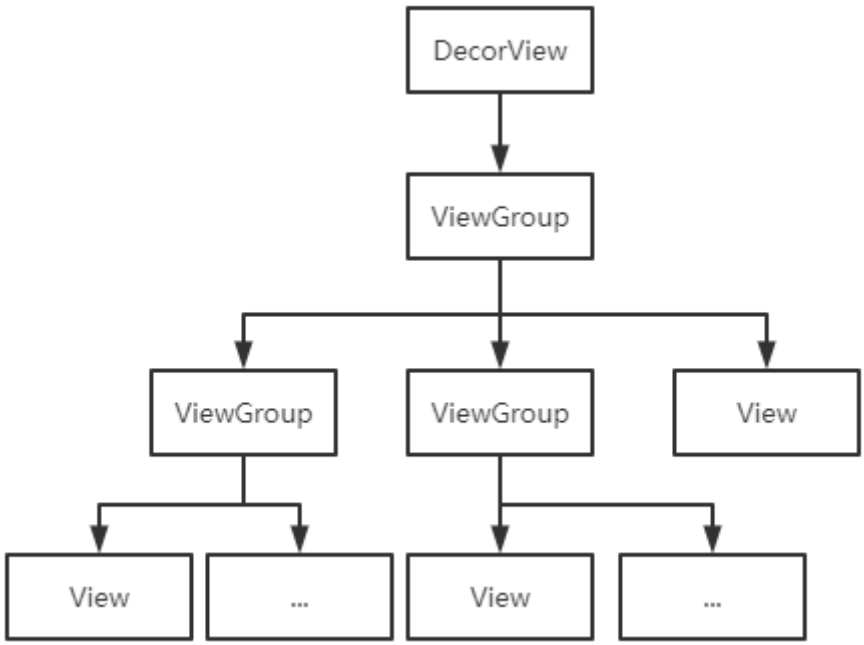


图7-1

view的度量过程就是按照上面的图7-1 viewTree的层次结构进行分发，先从viewRootImpl中执行performMeasure函数，然后再调用view的measure函数，此时的view是rootView属于一个ViewGroup，此时的ViewGroup会执行自己的onMeasure函数：度量孩子并确定自己的宽高；同时在度量孩子的时候，孩子view（viewGroup）就会执行进行同样的分发流程，从而遍历整棵树完成所有view的度量。

2.2 layout过程

layout的过程是基于度量的值，对viewTree上面的节点进行布局的过程，整体流程也是按照图7-1的结构，对7-1所指的树中的节点进行深度遍历，直到所有的树节点完成遍历为止，由于过程基本一致本文就不再赘述，感兴趣的朋友可以自行阅读源码或者通过享学课堂的课程进行学习。

2.3 draw 绘制流程

Draw 的分发过程也是和Measure的过程一样，入口是ViewRootImpl中的performTraversals()，然后再经过 performDraw()将draw的事件逐步通过函数调用分发到 View.java 中的draw(Canvas canvas)函数，所以我们接下来的分析就重点探讨View.java 中的draw函数。

代码中的draw方法注释和解析请大家认真理解

```
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;
        }
    }
```

step1:绘制背景时，首先根据滚动值对canvas的坐标进行调整，然后再恢复坐标， 图中外面是一个任意ViewGroup的实例，内部包含一个TextView对象，粗实线区域代表该TextView 在 ViewGroup中的位置， TextView中的文字由于滚动，一部分已经超出了粗实线区域，从而不可见。此时，如果调用canvas.getClipBoundsO返回的矩形区域是指粗实线所示的区域，该矩形的坐标是相对其 父视图ViewGroup的左上角，并且如果调用canvas的 getHeight()和 getWidth()方法将返回父视图的高度 和宽度，此处分别为200dip和 320dip。如 果ViewGroup中包含多个子视图，那么每个子视图内部的onDraw()函数中参数canvas的大小都是相同的，为父视图的大小。唯一不同的是"剪切区"，这个剪切区正是父视图分配给子视图的显示区 域 。 canvas之所以被设计成这样正是为了 View树的绘制，对于任何一个View而言，绘制时都可以认为原点坐标就是该View本身的原点坐标，从而 对 于View而言，当用户滚动屏幕时，应用程序只需要 调 用View类 的 scrollBy()函数即可，而不需要在onDraw()函数中做任何额外的处理,View的 onDraw() 函数内部可以完全忽略滚动值。 由于背景本身针对的是可视区域的背景，而 不 是 整 个 V iew 内部的背景，因此，本步中先调用translateO将原点移动到粗实线的左上角，从而使得背景Drawable对象内部绘制的是粗实线的区域。当绘制完背景后，还需要重新调用transalte()将原点坐标再移回到TextView本 身 的 （ 0 ,0)坐标。

step2:如果该程序员要求显示视图的渐变框，则需要先为该操作做一点准备，但是大多数情况下都不需要显示渐变框，因此，源码中针对这种情况进行快速处理，即略过该准备。

step3:绘制视图本身，实 际 上 回 调onDraw()函数即可， View 的设计者可以在onDraw()函数中调用canvas的各种绘制函数进行绘制。

step4:调 用 dispatchDraw()绘制子视图。如果该视图内部不包含子视图，则不需要重载该函数，而对所有 的ViewGroup实例而言，都必须重载该函数，否则它也就不是ViewGroup 了。

其他的步骤我们就不再介绍了，相信大家都能搞明白。

总结

View的绘制流程是面试最容易被问到的问题，而且这个问题面试者非常容易满足于一知半解。这道问题的正确的回答方式是从viewRootImpl开始，解析整个viewTree的构建分发流程。

9.2 View绘制流程与自定义View注意点

这道题想考察什么？

是否了解View绘制流程与自定义View注意点与实际场景使用，是否熟悉View绘制流程与自定义View注意事项

考察的知识点

View绘制流程与自定义View注意事项的概念在实际项目中使用

考生应该如何回答

这个问题先需要回答View的绘制流程，然后再回到自定义View的注意点。

9.2.1. View的绘制流程

这个问题的回答请看7-1题，View的绘制原理。

9.2.2. 自定义View注意点？

1）View需要实现四个构造函数

自定义View中构造函数有四种

```
    //  主要是在java代码中new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：在xml中定义的参数内容

int defStyleAttr：主题中优先级最高的属性

int defStyleRes： 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)，只有当defStyleAttr为0或者当前Theme中没有给defStyleAttr属性赋值时才起作用.

在android中的属性可以在多个地方进行赋值，涉及到的优先级排序为：**在布局xml中直接定义 > 在布局xml中通过style定义 > 自定义View所在的Activity的Theme中指定style引用 > 构造函数中defStyleRes指定的默认值**

2）自定义View的种类各不相同，须要根据实际须要选择一种简单低成本的方式来实现，尽可能的减少UI的层级，view的种类如下，开发中需要尽可能的选择适合自己的。

- 继承View重写onDraw方法

主要用于实现不规则的效果，也就是说这种效果不适宜采用布局的组合方式来实现。也就是需要使用canvas，Paint，运用算法去“绘制”了。采用这种方式须要本身支持wrap_content，padding也须要本身处理canvas。

- 继承ViewGroup派生特殊的Layout

主要用于实现自定义的布局，看起来很像几种View组合在一块儿的时候，可使用这种方式。这种方式须要合适地处理ViewGroup的测量和布局，尤其在测量的时候需要注意padding 和margin，比如：自定义一个自动换行的LinerLayout等。

- 继承特定的View，好比TextView

这种方法主要是用于扩展某种已有的View，增长一些特定的功能。这种方法比较简单，也不须要本身支持wrap_content和padding。这种效果就相当于我们使用imageView来实现一个圆形的imageView。

- 继承特定的ViewGroup，好比LinearLayout

这种方式也比较常见，也就是基于原来已经存在的layout，在其上去添加新的功能。和上面的第2种方法比较相似，第2种方法更佳接近View的底层。

```
3）View需要支持padding
```

直接继承View的控件需要在onDraw方法中处理padding，否则用户设置padding属性就不会起作用。直接继承ViewGroup的控件需要在onMeasure和onLayout中考虑padding和子元素的margin对其造成的影响，不然将导致padding和子元素的margin失效。

4) 尽量不要在view中使用handler

如果在view中需要使用handler来处理异步信息，这个时候尽量使用view中的post方法，因为view中给用户提供了post方法，这个方法的处理逻辑是：将Runable 的action 保存到一个队列，在viewRootImpl里面执行度量后再添加进Handler的MessageQueue中，这样就保障了action里面执行的内存是在完成度量 后的结果，避免了使用延迟消息带来的困扰具体的代码如下：

```
public class View implements Drawable.Callback, KeyEvent.Callback,
        AccessibilityEventSource {

    ...

    public boolean post(Runnable action) {
        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.
        getRunQueue().post(action);
        return true;
    }

    public boolean postDelayed(Runnable action, long delayMillis) {
        final AttachInfo attachInfo = mAttachInfo;
        if (attachInfo != null) {
            return attachInfo.mHandler.postDelayed(action, delayMillis);
        }

        // Postpone the runnable until we know on which thread it needs to run.
        // Assume that the runnable will be successfully placed after attach.
        getRunQueue().postDelayed(action, delayMillis);
        return true;
    }

    ...
```

```
}
```

5）在自定义view的onMeasure，onDraw，onLayout方法中尽量少使用局部变量

由于onMeaaure &onDraw&onLayout这3个函数都可能存在频繁调用的可能，尤其在动画里面，onDraw是非常频繁的调度的，在这些频繁被调用的函数里面，开发过程中一定要尽量避免创建局部对象，尤其是比较占用内存的像bitmap等的局部对象，这很容易产生内存抖动，而内存抖动容易带来手机app的卡顿，具体的细节大家可以去享学课堂查找对应的课程进行学习。

9.3 自定义view与viewgroup的区别

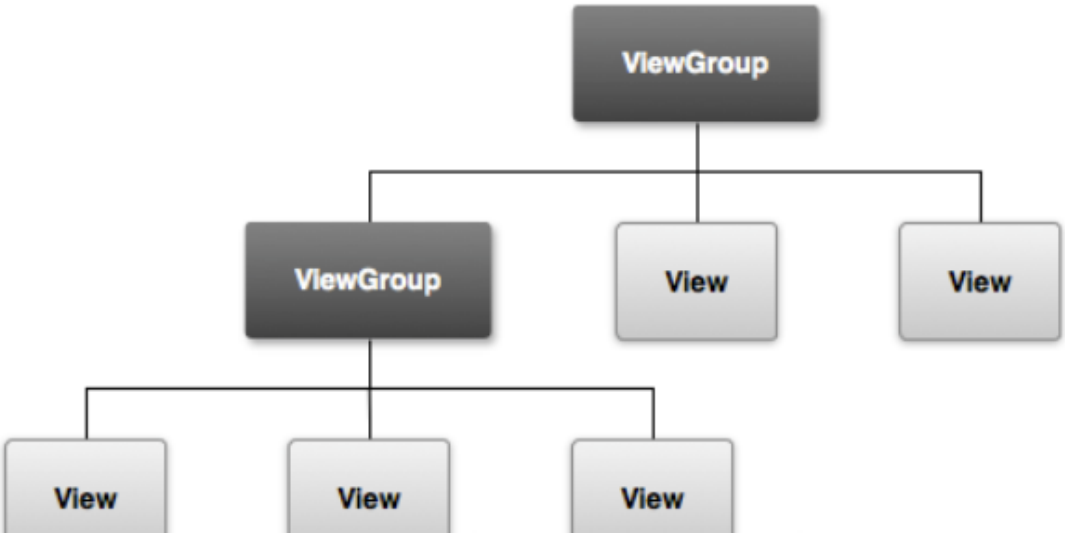
这道题想考察什么？

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？

考生应该如何回答

说说自定义view与viewgroup的区别？

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（一般在写自定义控件的时候使用），而是使用其派生类。



ViewGroup的职责是什么？

ViewGroup相当于一个放置View的容器，在写布局xml的时候，会告诉容器：容器宽度（layout_width）、高度（layout_height）、对齐方式（layout_gravity），还有margin等。因此ViewGroup的职能为：给childView计算出建议的宽和高和测量模式 ，并决定childView的位置，将childView布局到这个Layout上面。

当然我们要知道为什么只是建议的宽和高，而不是直接确定呢？原因很简单，子View的宽和高可以设置为wrap_content，这样只有子View自己才能计算出自己的宽和高。

View的职责是什么？

View的职责，根据测量模式和ViewGroup给出的建议宽和高，计算出自己的宽和高；另外还有个更重要的职责是：在ViewGroup为其指定的区域内绘制自己的形状，所以，view的主要职责会集中在实现onDraw方法。

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；

1）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()：a）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；b）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。

总之，viewGroup处理和分发事件要相对于View而言复杂很多，更多详细的关于事件分发的问题，大家可以参考后面的章节，我们会有详细的说明。

2）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()，当我们自定义View的时候，最主要的需要实现的方法是onDraw(),其他的方法可以使用View.java中的，不一定必须重写；如果需要改变view的大小，那么才需要重写onMeasure()方法；如果需要改变View的（在父控件的）位置，那么才需要重写onLayout()方法。同时，当我们在自定义ViewGroup的时候，最主要的需要实现的是onLayout()和onMeasure()方法，其他放到大多可以基础使用父ViewGroup的。

2.绘制流程基本一直：onMeasure（测量）--》onLayout（布局）--》onDraw（绘制）。

3.绘制按照ViewTree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.onMeasure():用于计算自己及所有子对象的大小。这个方法是所有View、ViewGroup及其派生类都具有的方法。自定义控件时，可以重载该方法，重新计算所有对象的大小。 MeasureSpec包含了测量的模式和测量的大小，通过MeasureSpec.getMode()获取测量模式，通过MeasureSpec.getSize()获取测量大小。mode共有三种情况： 分别为MeasureSpec.UNSPECIFIED（ View想多大就多大），MeasureSpec.EXACTLY（默认模式，精确值模式：将layout_width或layout_height属性指定为具体数值或者match_parent)，MeasureSpec.AT_MOST（ 最大值模式：将layout_width或layout_height指定为wrap_content）。

6.onLayout()：对于View来说，onLayout()只是一个空实现，一般情况下不需要重写；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。

7.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。

8.dispathDraw():ViewGroup及其派生类具有的方法，主要用于控制子View的绘制分发。自定义ViewGroup控件时，重载该方法可以改变子View的绘制，进而实现一些复杂的视效,在自定义View中不存在此方法。

9.drawChild(Canvas canvas, View child, long drawingTime)：ViewGroup及其派生类具有的方法，用于直接绘制具体的子View。自定义控件时，重载该方法可以直接绘制具体的子View。

总结

我们从事件分发和绘制两个角度全面剖析了自定义View和ViewGroup的区别，请大家在平时开发中，一定要多实践，只有在实践中才能真正的明白自定义View背后的逻辑。

9.4 View的绘制流程是从Activity的哪个生命周期方法开始执行的

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。更多细节，请各位去查看7.6题相关的说明。

9.5 Activity,Window,View三者的联系和区别

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；

  通过上面名词的解释以及平时开发中的实际，大家可以发现window其实没有什么实质性的意义，因为平时开发中，我们往往是在activity里面直接处理View。那么为什么还需要window呢？主要是从面向对象的角度来设计代码的结构的，类的职责应该单一，那么activity view都有它们自己的职责，所以activity管理view就需要有它独立类来进行这就是通过window进行。

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。
- WindowManager也是一个接口，它的唯一实现类是WindowManagerImpl，所以往往在管理view的时候，系统是通过WindowManagerImpl来管理view。
- WindowManagerImpl内部的实现极其简单，它会将所有的方法全部转交给WindowManagerGlobal中的方法来进行。

- windowManagerGlobal 通过调用 ViewRootImpl 来完成对view的操作。
- ViewRootImpl 是视图层次结构的顶部，它实现了View与WindowManager之间所需要的协议，并且大部分的WindowManagerGlobal的内部实现是通过ViewRootImpl来进行的。

```
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。

9.6 在onResume中是否可以测量宽高

这道题想考察什么？

这道题想考察同学对 View的绘制流程 的理解。

考生应该如何回答

这个问题答案是：不一定能够正确的获取view的宽高，然后我们要分析原因，然后我们还要讲解解决办法。

如果是activity 启动后第一次进入onResume 生命周期，那么获取到的View的宽高是错误的；如果是从其他activity回到当前activity而执行的onResume方法，那么就能够获取到View的宽高。究其原因如下：

首先我们要找到系统调用 onResume 的地方，大家可以看到 ActivityThread 类，在这个类中有一个函数handleResumeActivity，这个函数在下面我们只保留了核心代码。

```
handleResumeActivity() {
    //...
    r = performResumeActivity(token, clearHide, reason);//调用 onResume() 方法
    //...
    wm.addView(decor, l);// WindowManager添加Decor（decor是DecorView）
    //...
}
```

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)，大家看下面的分析;

```
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()，这个函数的主要作用是告诉安卓系统app端需要界面的刷新，并且设置一个回调，用于接收系统发送的同步app到系统的信号。

```
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，当系统审核app端刷新界面的申请后，在合适的时机会调用之前设定的TraversalRunnable回调。

```
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);// code 7

    //...

    performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);//触发viewTree
onMeasure
    ...
    performLayout(lp, mWidth, mHeight);//触发viewTree onLayout
    ...
    performDraw();//触发ViewTree onDraw

    //...
}
```

小结：

在上面的代码中，performTraversals 函数是整个viewTree 的onMeasure 、onLayout、onDraw的发起点。所以，基于上面的代码流程分析大家不难发现activity 的onResume 函数是在 viewTree 的 onMeasure 函数执行之前先执行，也就是在执行onResume函数的时候，viewTree并未完成度量。那么，当我们去onResume函数里面获取view 的宽高自然是获取不到的。

那么如何在onResume中获取view的宽高呢？

在 onResume() 中 handler.post(Runnable) 是无法正确的获取不到 View 的真实宽高

原因：能够获取到view的宽高的前提条件是view要完成onMeasure，那么我们的handler.post 的方式是不能够保证 runnable一定是在view进行了onMeasure后执行的，因此，这个方式不行。同样的道理，运用handler.postDelay也不是最佳的方案，因为，不同型号的手机代码的执行时间不同，导致postDelay到底需要delay多久对于用户来说仍旧是未知数。那么正确的做法是怎样的呢？答案就是调用View.post(Runnable)。

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 中的code 7 可知，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() 中直接获取或者调用创建的 Handler 的 post 方法都是获取不到 View 宽高的，需要通过 View.post 才能获取。

9.7 如何更新UI，为什么子线程不能更新UI？

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;
            }
        ...
}
```

可以看到内部执行了`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);
    }
```

最终然后又执行了`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

9.8 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。
2. 这道题还有一个问法：Window，DecorView，ViewRootImpl的关系

考察的知识点

1. DecorView, ViewRootImpl,View之间的关系的概念，以及他们在源码层级的定位和意义。

考生应该如何回答

这个问题和 7.5 [Activity,Window,View三者的联系和区别]是一个姊妹题，要将这两道题同步学习和看待。

回答这个问题，我们首先要明白，Window，DecorView ，ViewRootImpl，View 到底是什么，什么时候创建的？

1.Window是什么？

window是Android里面 管理view的工具，装载View的实体，每个activity或者dialog 一定要有一个window，而window的唯一实现类就是phoneWindow，我们可以看下面的分析。

```
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”到底是什么？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();
    }
}
```

首先判断了代码1处，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activity的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。

2.DecorView是什么？

创建DecorView的过程如下：

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

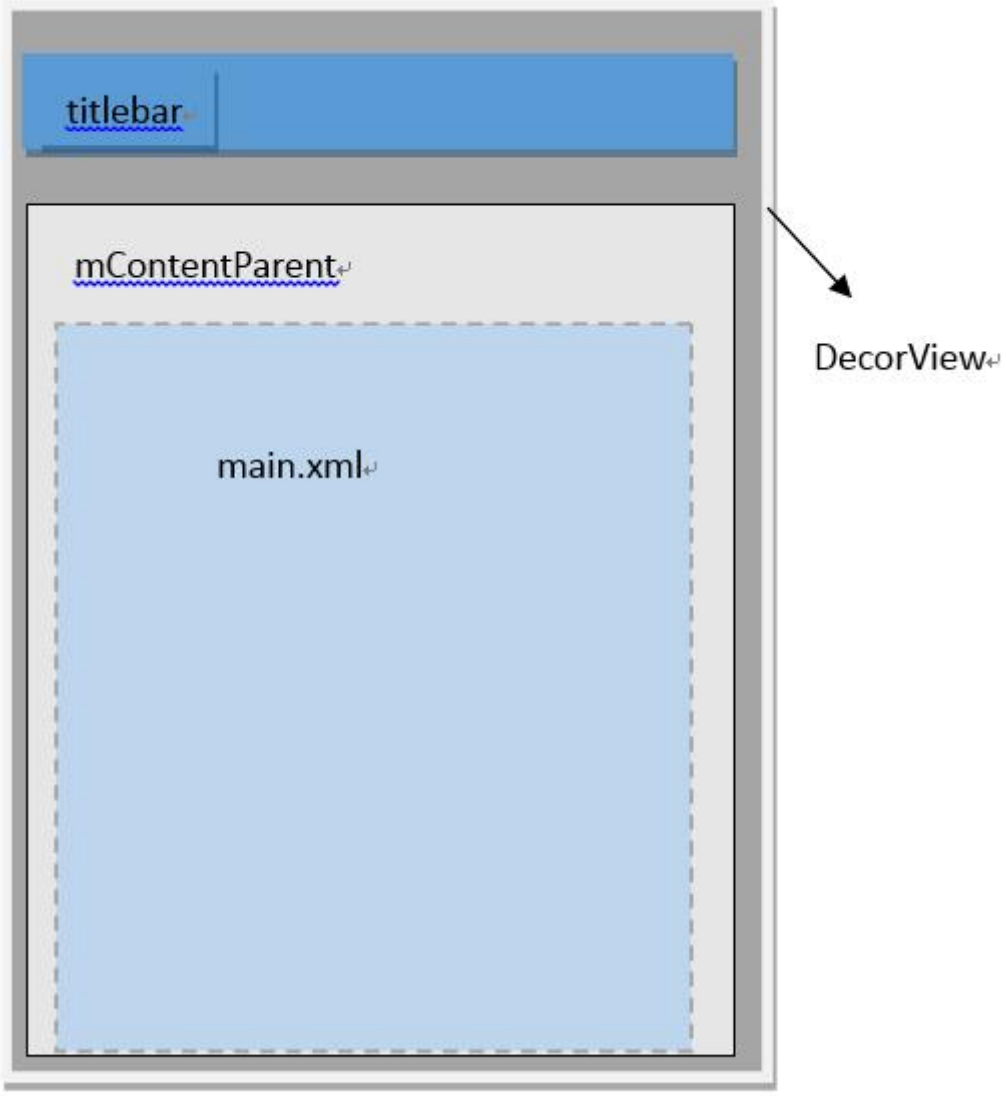
```
protected DecorView generateDecor() {
    return new DecorView(getContext(), -1);
}
```

可以看出，这里实例化了DecorView，而DecorView则是一个继承于FrameLayout的类，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generateLayout方法，在 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，大家如果感兴趣可以参考享学课堂的课程去学习一下。

3.ViewRootImpl是什么

ViewRootImpl 在比较早期的api里面是叫ViewRoot。无论是叫ViewRootImpl 或者是ViewRoot，都可以展示一点：它与View 的root（根）有着紧密的联系。在7.1 分析view 的绘制过程中，我们可以发现ViewRootImpl ViewRootImpl 是一个视图层次结构的顶部如下图7-5所示，它相当于在根布局DecorView上面构建了一个新的View（ViewRootImpl），只是这个“view” 没有自己的具体形态，而是用于管理整个ViewTree的起点而已。同时，ViewRootImpl 实现了 View 与 WindowManager 之间所需要的协议，作为 WindowManagerGlobal 中大部分的内部实现，也就说 WindowManagerGlobal 方法最后还是调用到了 ViewRootImpl。

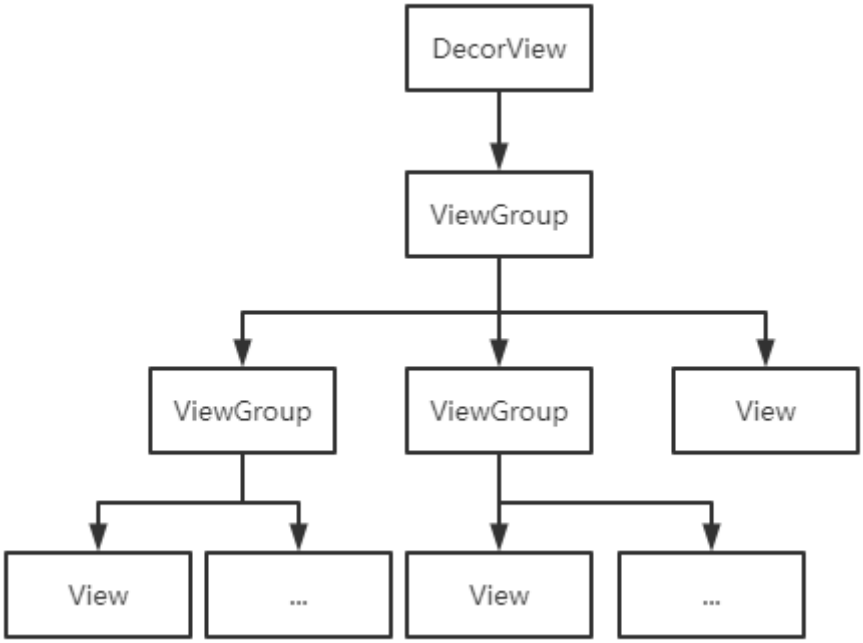


图 7-5

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
            }
            //...
        }
```

```
    }
}
```

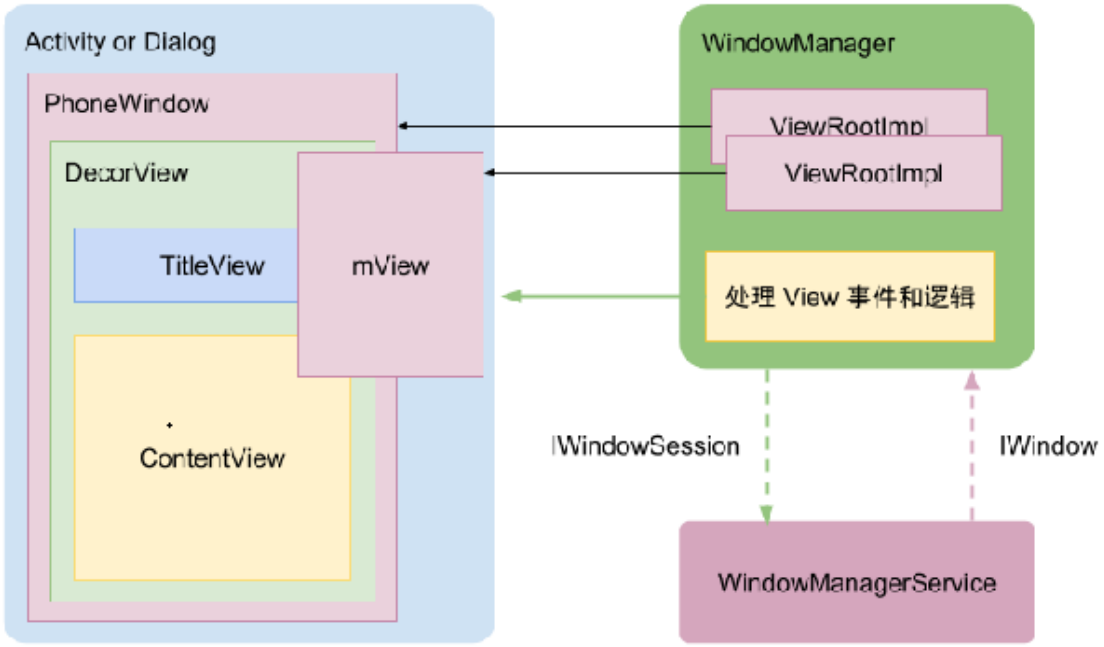
在该方法内部，获取该activity所关联的window对象，DecorView对象，以及windowManager对象，而WindowManager是抽象类，它的实现类是WindowManagerImpl，所以后面调用的是WindowManagerImpl#addView方法，我们看看源码：

```
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。所以，我们又称ViewRootImpl 是沟通 Activity中的View 和 WMS 的桥梁，具体的关系大家可以看下图：



WindowManager 是一个类的集合，这个集合包含WindowManagerImpl、WindowManagerGlobal、ViewRootImpl，其核心是ViewRootImpl。当系统需要进行刷新事件时，这个时候就会调用到ViewRootImpl里面的performTraversals()方法，这个方法就会通过 ViewTree递归的分发onMeasure、onLayout、onDraw。当需要更新界面的数据帧的时候，ViewRootImpl 也会将每一个数据帧 通过IWindowSession（binder）将事件发送给WIndowManagerServcie，然后再通过底层进行渲染。所以，整个过程中ViewRootImpl就是一个桥，此处便是桥接模式。

4. View 和ViewRootImpl的关系

通过上面的分析，相信大家已经很容易发现，ViewRootImpl 内部有一个mView对象，ViewRootImpl一般是通过这个mView来将各种事件如measure、layout，draw分发到 viewTree上面，从而形成一个更新的逻辑。具体的代码调用过如下：

ViewRootImpl#performTraversals() ->ViewRootImpl#performMeasure();

ViewRootImpl#performTraversals() ->ViewRootImpl#performLayout();

ViewRootImpl#performTraversals() ->ViewRootImpl#performDraw();

而performMeasure()的代码如下：

```
private void performMeasure(int childWidthMeasureSpec, int
childHeightMeasureSpec) {
        if (mView == null) {
            return;
        }
        Trace.traceBegin(Trace.TRACE_TAG_VIEW, "measure");
        try {
            mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);//分发
measure
        } finally {
            Trace.traceEnd(Trace.TRACE_TAG_VIEW);
        }
}
```

9.9 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()方法的流程,为什么有时候不会回调onDraw()。

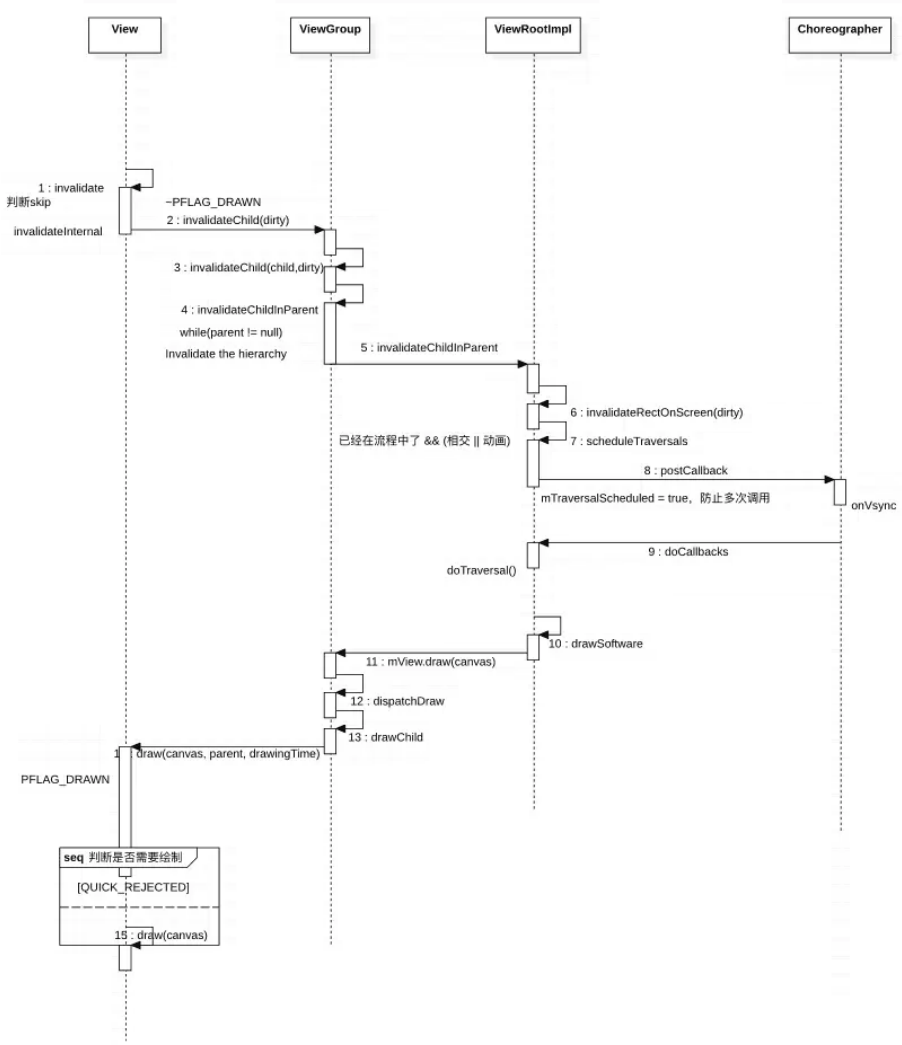
考察的知识点

1. 自定义View执行invalidate()方法的流程,它回调onDraw()的过程细节概念。

考生应该如何回答

首先我们分析一下invalidate()的执行流程，源码是如何从invalidate调用到onDraw()的。由于这部分代码相对较为复杂，那么请大家参考下面的时序图。

1) invalidate软件绘制流程



从上面的流程不难发现：1）view的invalidate会逐层找parent一直找到DecorView，DecorView是顶层view，它有个虚拟父view为ViewRootImpl。ViewRootImpl不是一个view或者viewGroup，它的成员mView就是DecorView，然后再由ViewRootImpl将所有的操作从ViewRootImpl自上而下开始分发，最终分发给所有的View。2）view的invalidate不会导致ViewRootImpl的invalidate被调用，而是递归调用父view的invalidateChildInParent，直到ViewRootImpl的invalidateChildInParent，然后触发peformTraversals，会导致当前view被重绘,由于mLayoutRequested为false，不会导致onMeasure和onLayout被调用，而OnDraw会被调用。

在整个调度流程里面有几个重要的地方 需要拿出来和大家一起探讨一下：

在View.java 类里面代码如下，请大家关注代码中的注解

```
public void invalidate() {
   invalidate(true);
}

public void invalidate(boolean invalidateCache) {
   //invalidateCache 使绘制缓存失效
   invalidateInternal(0, 0, mRight - mLeft, mBottom - mTop, invalidateCache,
true);
}

void invalidateInternal(int l, int t, int r, int b, boolean invalidateCache,
                        boolean fullInvalidate) {
    ...
    //设置了跳过绘制标记
    if (skipInvalidate()) {
```

```
        return;
    }

    //PFLAG_DRAWN 表示此前该View已经绘制过 PFLAG_HAS_BOUNDS表示该View已经layout过，确
定过坐标了
    if ((mPrivateFlags & (PFLAG_DRAWN | PFLAG_HAS_BOUNDS)) == (PFLAG_DRAWN |
PFLAG_HAS_BOUNDS)|| (invalidateCache && (mPrivateFlags &
        PFLAG_DRAWING_CACHE_VALID) == PFLAG_DRAWING_CACHE_VALID) ||
(mPrivateFlags &
        PFLAG_INVALIDATED) != PFLAG_INVALIDATED|| (fullInvalidate && isOpaque()
!=
        mLastIsOpaque)) {
        if (fullInvalidate) {
            //默认true
            mLastIsOpaque = isOpaque();
            //清除绘制标记
            mPrivateFlags &= ~PFLAG_DRAWN;
        }

        //需要绘制
        mPrivateFlags |= PFLAG_DIRTY;

        if (invalidateCache) {
            //1、加上绘制失效标记
            //2、清除绘制缓存有效标记
            //这两标记在硬件加速绘制分支用到
            mPrivateFlags |= PFLAG_INVALIDATED;
            mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
        }

        final AttachInfo ai = mAttachInfo;
        final ViewParent p = mParent;
        if (p != null && ai != null && l < r && t < b) {
           final Rect damage = ai.mTmpInvalRect;
           //记录需要重新绘制的区域 damge，该区域为该View尺寸
           damage.set(l, t, r, b);
           //p 为该View的父布局
           //调用父布局的invalidateChild
           p.invalidateChild(this, damage);
        }
        ...
    }
}
```

从上面代码可以知道，当前要刷新的View确定了刷新区域后即调用了父布局ViewGroup的invalidateChild()方法。

有一个函数很重要，skipInvalidate()：如果当前view不可见而且也没有在执行动画的时候这个时候不能触发invalidate。

```
 private boolean skipInvalidate() {
    return (mViewFlags & VISIBILITY_MASK) != VISIBLE && mCurrentAnimation ==
null &&
            (!(mParent instanceof ViewGroup) ||
                    !((ViewGroup) mParent).isViewTransitioning(this));
}
```

另外一个很重要的函数invalidateChild也需要给大家解析一下：

```
 public final void invalidateChild(View child, final Rect dirty) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null && attachInfo.mHardwareAccelerated) {
        //1、如果是支持硬件加速，则走该分支
        onDescendantInvalidated(child, child);
        return;
    }
    //2、软件绘制
    ViewParent parent = this;
    if (attachInfo != null) {
        //动画相关，忽略
        ...
        do {
           View view = null;
           if (parent instanceof View) {
               view = (View) parent;
           }
           ...
           parent = parent.invalidateChildInParent(location, dirty);
           //动画相关
        } while (parent != null);
    }
}
```

从上面的代码注解大家不难发现，该方法分为硬件加速绘制和 软件绘制。

我们先看看硬件，如果该Window支持硬件加速，则走下边流程，ViewGroup.java中

```
public void onDescendantInvalidated(@NonNull View child, @NonNull View target) {
    mPrivateFlags |= (target.mPrivateFlags & PFLAG_DRAW_ANIMATION);

    if ((target.mPrivateFlags & ~PFLAG_DIRTY_MASK) != 0) {
       //此处都会走
        mPrivateFlags = (mPrivateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DIRTY;
        //清除绘制缓存有效标记
        mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
    }

    if (mLayerType == LAYER_TYPE_SOFTWARE) {
        //如果是开启了软件绘制，则加上绘制失效标记
        mPrivateFlags |= PFLAG_INVALIDATED | PFLAG_DIRTY;
        //更改target指向
        target = this;
    }

    if (mParent != null) {
        //调用父布局的onDescendantInvalidated
        mParent.onDescendantInvalidated(this, target);
    }
}
```

onDescendantInvalidated 方法的目的是不断向上寻找其父布局，并将父布局 PFLAG_DRAWING_CACHE_VALID 标记清空，也就是绘制缓存清空。 而我们知道，根View的mParent指向ViewRootImpl对象，因此来看看它里面的onDescendantInvalidated()方法：

```
@Override
public void onDescendantInvalidated(@NonNull View child, @NonNull View
descendant) {
    // TODO: Re-enable after camera is fixed or consider targetSdk checking this
    // checkThread();
    if ((descendant.mPrivateFlags & PFLAG_DRAW_ANIMATION) != 0) {
        mIsAnimating = true;
    }
    invalidate();
}

@UnsupportedAppUsage
void invalidate() {
    //mDirty 为脏区域，也就是需要重绘的区域
    //mWidth，mHeight 为root view的尺寸
    mDirty.set(0, 0, mWidth, mHeight);
    if (!mWillDrawSoon) {
        //开启View 三大流程
        scheduleTraversals();
    }
}
```

invalidate() 对于支持硬件加速来说，会将整个root view区域内的大小都设置为mDirty区域，刷新的时候就会全面的将整个区域进行刷新。所以，当刷新失效的时候，我们往往可以设置硬件刷新来让整个区域都可以刷新，从而达到刷新的效果。

然后，再来分析软件刷新分支，如果该Window不支持硬件加速，那么走软件绘制分支：parent.invalidateChildInParent(location, dirty) 返回mParent，只要mParent不为空那么一直调用invalidateChildInParent()，实际上这也是遍历ViewTree过程，来看看关键invalidateChildInParent()

```
public ViewParent invalidateChildInParent(final int[] location, final Rect
dirty) {
    //dirty 为失效的区域，也就是需要重绘的区域
    if ((mPrivateFlags & (PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID)) != 0) {
        //该View绘制过或者绘制缓存有效
        if ((mGroupFlags & (FLAG_OPTIMIZE_INVALIDATE | FLAG_ANIMATION_DONE))
                != FLAG_OPTIMIZE_INVALIDATE) {
            //修正重绘的区域
            dirty.offset(location[CHILD_LEFT_INDEX] - mScrollX,
                    location[CHILD_TOP_INDEX] - mScrollY);
            if ((mGroupFlags & FLAG_CLIP_CHILDREN) == 0) {
                //如果允许子布局超过父布局区域展示
                //则该dirty 区域需要扩大
                dirty.union(0, 0, mRight - mLeft, mBottom - mTop);
            }
            final int left = mLeft;
            final int top = mTop;
            if ((mGroupFlags & FLAG_CLIP_CHILDREN) == FLAG_CLIP_CHILDREN) {
                //默认会走这
                //如果不允许子布局超过父布局区域展示，则取相交区域
                if (!dirty.intersect(0, 0, mRight - left, mBottom - top)) {
                    dirty.setEmpty();
                }
            }
            //记录偏移，用以不断修正重绘区域，使之相对计算出相对屏幕的坐标
            location[CHILD_LEFT_INDEX] = left;
```

```
            location[CHILD_TOP_INDEX] = top;
        } else {
            ...
        }
        //标记缓存失效
        mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
        if (mLayerType != LAYER_TYPE_NONE) {
            //如果设置了缓存类型，则标记该View需要重绘
            mPrivateFlags |= PFLAG_INVALIDATED;
        }
        //返回父布局
        return mParent;
    }
    return null;
}
```

通过上面的代码发现，最终调用ViewRootImpl 的invalidateChildInParent().

```
public ViewParent invalidateChildInParent(int[] location, Rect dirty) {
    checkThread();
    if (DEBUG_DRAW) Log.v(mTag, "Invalidate child: " + dirty);
    if (dirty == null) {
        //脏区域为空，则默认刷新整个窗口
        invalidate();
        return null;
    } else if (dirty.isEmpty() && !mIsAnimating) {
        return null;
    }
    ...
    invalidateRectOnScreen(dirty);
    return null;
}

private void invalidateRectOnScreen(Rect dirty) {
    final Rect localDirty = mDirty;
    //合并脏区域，取并集
    localDirty.union(dirty.left, dirty.top, dirty.right, dirty.bottom);
    ...
    if (!mWillDrawSoon && (intersected || mIsAnimating)) {
        //开启View的三大绘制流程
        scheduleTraversals();
    }
}
```

invalidate() 对于软件绘制来说，目的就是通过计算找到需要重绘的区域，确定了需要重绘的区域后，然后再调用scheduleTraversals对这个区域触发它的绘制。关于scheduleTraversals的解释，大家可以去看7.1，在7.1详细的解释了它的整个流程。

小结

以上，从硬件加速绘制与软件绘制全面分析了invalidate触发onDraw的整个流程。如果window不支持硬件加速绘制，那么view的invalidate将不会导致ViewRootImpl的invalidate被调用和执行，而是通过软件绘制的方式递归调用父view的invalidateChildInParent，直到ViewRootImpl的invalidateChildInParent，然后触发peformTraversals，会导致当前view被重绘，由于mLayoutRequested为false，不会导致onMeasure和onLayout被调用，而OnDraw会被调用(只绘制需要重绘的视图)。

大家在看代码的时候请注意：1）同一个draw时序内连续调用同一View的invalidate时，会被Flag阻挡，不再向下走。2）同一个draw时序内不同View调用invalidate时只会调动一个，不会重复执行。3)在执行scheduleTraversals方法的时候最终会执行到peformTraversals，由于mLayoutRequested为false，不会导致onMeasure和onLayout被调用，而OnDraw会被调用。4）在ViewGroup中ondraw总是不执行,或者说不被调用，原因是 如果ViewGroup的background是空的，那么onDraw就一定不会执行，但是他的dispatchDraw会执行，所以可以重写dispatchDraw方法;5）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，也就是说view必须是当前Window上面的。

9.10 invalidate() 和 postInvalicate() 区别

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。

9.11 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

讲解他们的工作流程，还有运行后的效果，从而对比他们的区别和联系

我们先来看View中的RequestLayout，看名字大家应该不难判断这个方法就是请求重新布局。

```
public void requestLayout() {
    //清空测量缓存
    if (mMeasureCache != null) mMeasureCache.clear();
    ...
    //添加强制layout 标记，该标记触发layout
    mPrivateFlags |= PFLAG_FORCE_LAYOUT;
    //添加重绘标记
```

```
    mPrivateFlags |= PFLAG_INVALIDATED;

    if (mParent != null && !mParent.isLayoutRequested()) {
        //如果上次的layout 请求已经完成
        //父布局继续调用requestLayout
        mParent.requestLayout();
    }
    ...
}
```

不难看出，这个递归调用和题目7.10 中invalidate是采用了一样的配方，向上寻找其父布局，一直到ViewRootImpl为止，给每个布局设置PFLAG_FORCE_LAYOUT和PFLAG_INVALIDATED标记，注意这个是关键，所有的布局都在递归中被设置了标志位，然后最终会走到ViewRootImpl里面的requestLayout()，看下面的代码：

```
public void requestLayout() {
    //是否正在进行layout过程
    if (!mHandlingLayoutInLayoutRequest) {
        //检查线程是否一致
        checkThread();
        //标记有一次layout的请求
        mLayoutRequested = true;
        //开启View 三大流程
        scheduleTraversals();
    }
}
```

所以，RequestLayout做的具体事情就是这样的：向上寻找父布局，找到后给他们标上强制layout和重绘的标签；然后 调用ViewRootImpl的 scheduleTraversals()开启三大绘制流程。

总结

RequestLayout()方法：会导致调用 onMeasure()方法和 onLayout()，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);

9.12 View的滑动方式

这道题想考察什么？

Android中经常看到一些炫酷的效果，这些效果很多伴随着View的滑动。我们想要做出这样的效果，掌握View的滑动方式必不可少。

考察的知识点

View的滑动方式的概念在项目中使用及相关基本原理的理解。

考生应该如何回答

我们在使用View的过程中，经常需要实现View的滑动效果。比如ListView、跟随手指而移动的自定义View等等，前者的滑动效果是SDK为我们提供的，而对于我们自定义View的滑动效果就需要我们自己来实现，下面介绍四种滑动的方式，和大家一起共享一下。另外，在回答这个问题的时候，希望大家也去学习一下 View 的位置相关内容，View的坐标相关内容。

1、使用scrollTo/scrollBy

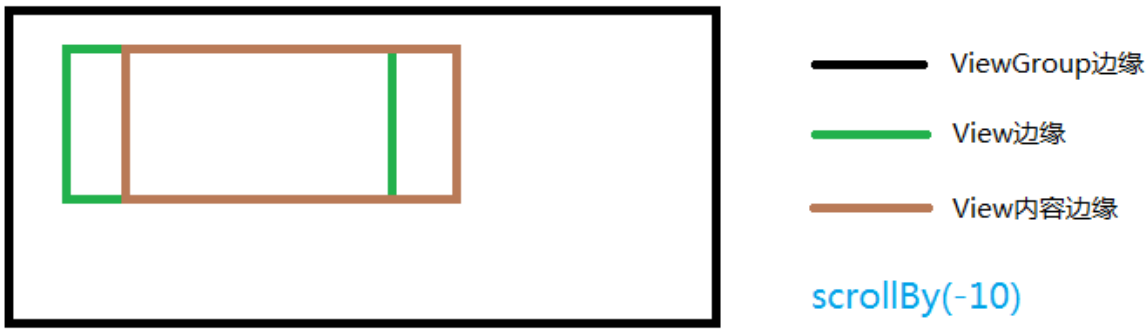
为了实现View滑动，Android专门提供了这两个方法让我们使用。这两个函数的区别是scrollBy提供的是基于当前位置的相对滑动，而scrollTo提供的是基于起始位置的绝对滑动。

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 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();
```

使用动画来实现View的滑动思想：主要通过改变View的translationX和translationY参数来实现，使用动画的好处在于滑动效果是平滑的。

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10dp，那么最简单的方式就是把它LayoutParams里的marginLeft参数的值增加10dp。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());
```

通过改变布局参数来实现View的滑动的思想很简单：比如向右移动一个View，只需要把它的marginLeft参数增大，向其它方向移动同理，只需改变相应的margin参数；还有一种比较拐弯抹角的方法是在要移动的View的旁边预先放一个View（初始宽高设为0），然后比如我们要向右移动View，只需把预先放置的那个View的宽度增大，这样就把View“挤”到右边了。

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();
        }
    }
}
```

大概了解下Scroller实现弹性滑动的原理：invaldate方法会导致View的draw方法被调用，而draw会调用computeScroll方法，因此重写了computeScroll方法，而computeScrollOffset方法会根据时间的流逝动态的计算出很小的一段时间应该滑动多少距离。也就是把一次滑动拆分成无数次小距离滑动从而实现“弹性滑动”。

9.13 事件分发机制是什么过程？

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

Android事件分发是一个老生常谈的知识点。日常开发和求职过程中，都会碰到Android事件分发的问题。

Android的控件分为两类，ViewGoup和View。ViewGroup是控件的容器，可以包含多个子控件。View是控件的最小单位，它不能包裹其它的View。Android的ViewGroup对应的数据结构是树。

网上的事件分发的文章大多数是用线性的思维去分析控件树的事件遍历，我深以为不妥，经常让读者云里雾里，只见树木不见森林。

本文将以树的深度遍历方式来讲解DOWN事件的分发流程，以单链表的线性遍历方式讲解MOVE、UP事件的分发流程。

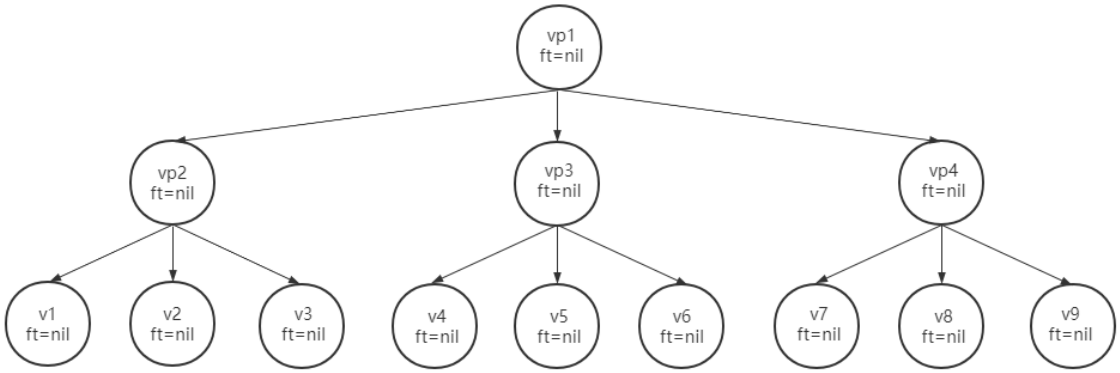
本文大纲

1. Android控件对应的多叉树
2. 手势事件类型
3. 事件分发涉及到的方法
4. DOWN事件的分发流程
5. MOVE、UP事件的分发流程
6. CANCEL事件触发时机以及分发流程

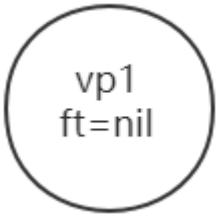
1. Android控件对应的多叉树

假设我们有一个这样的场景

屏幕上有一个FrameLayout名叫vp1。vp1有三个子控件vp2、vp3、vp4，它们的类型是FrameLayout。vp2有三个子控件v1、v2、v3。vp3有三个子控件v4、v5、v6。vp4有三个子控件v7、v8、v9。子控件的类型都是View。为了方便起见我们假设这些控件都是铺满屏幕的。我们将vp1控件树翻译成多叉树。



解释一下树节点ft的含义



ft的是ViewGroup类中的mFirtstTouchTarget成员变量的简称。它对应的数据结构是**单链表**。当DOWN事件在onTouchEvent方法中返回true，会**回溯**设置父View的ft指针。

举个例子

当DOWN事件从vp1开始分发。 假设DOWN事件在v7的onTouchEvent方法中返回true。 那么v7的父控件vp4的ft会指向v7(可以这样简单理解，实际上指向的是v7封装的一个TouchTarget对象)。同理vp1的ft会指向vp4。所以就生成了一条vp1->vp4->v7的分发路径。这很重要。

DOWN事件的分发是从vp1开始，假设所有的ViewGroup都不拦截事件，所有的View都不处理事件。事件会沿着最后一个子控件做深度遍历分发。以下用intercept代替onInterceptTouchEvent，touch代替onTouchEvent。

调用流程如下

vp1(intercept) -> vp4(intercept) -> v9(touch) -> v8(touch) ->v7 (touch) -> vp4(touch)-> vp3 -> v6(touch) -> v5(touch) -> v4(touch) -> vp3(touch)-> vp2 -> v3(touch) -> v2(touch) -> v1(touch) -> vp2(touch)-> vp1(touch)

这只是DOWN事件分发的一个Case。根据是否拦截，View是否处理事件。它的遍历路径也会不一样。

MOVE、UP事件分发也是从vp1开始，不同于DOWN事件的深度遍历方式，它们是通过ft的分发路径线性遍历。深度遍历是比较耗时的。如果vp1有后代View分发了事件。那么必然会通过ft生成一条分发路径。MOVE事件只需沿着分发路径线性分发就可以了。还是用上面的例子。如果v7分发了DOWN事件。那么MOVE、UP事件的分发即vp1(intercept) -> vp4(intercept) -> v7(touch)

2. 手势事件类型

本文主要讲解四种事件类型，DOWN、MOVE、UP、CANCEL。用户划动手机屏幕然后离开。Android系统首先会触发DOWN事件，紧接着一连串的MOVE事件，以UP事件收场。**注意：触摸屏幕，事件之间没有任何依赖。有可能只有其中一种事件被分发。也有可能有多种事件类型被分发**

事件类型	解释
ACTION_DOWN	手指按下屏幕
ACTION_MOVE	手指划动屏幕
ACTION_CANCEL	事件被取消
ACTION_UP	手指离开屏幕

3. 事件分发涉及到的方法

方法名	解释
dispatchTouchEvent	事件分发逻辑
onInterceptTouchEvent	是否拦截事件 **(ViewGroup专属)**
onTouchEvent	是否处理事件

1. dispatchTouchEvent方法是Android系统内部实现的事件分发逻辑。返回值为boolean类型。true表示该View或ViewGroup处理了事件，反之返回false。返回值含义同onTouchEvent的返回值。dispatchTouchEvent与onTouchEvent的区别在于，默认情况下前者的返回值依赖于后者的返回值，而且前者的侧重点在于制定事件分发的流程，后者的侧重点在于View或者ViewGroup是否处理该事件。
2. onInterceptTouchEvent方法是ViewGroup专属的方法。当返回值为true表示ViewGroup(假设vp1)需要拦截掉该事件。这里有两种情况

   2.1　处理DOWN事件时，如果onInterceptTouchEvent返回值为true，那么事件会直接交给vp1的onTouchEvent处理，如果返回false，交由vp1的最后一个View处理。

   2.2　处理MOVE、UP事件时，如果vp1.ft为null，此时不会调用vp1的onInterceptTouchEvent方法。如果vp.ft不为null而且vp1的onInterceptTouchEvent方法返回true，那么将会在vp1处生成CANCEL事件交由vp4分发，先后置空vp4，vp1的ft对象，接下来的MOVE、UP事件只会调用vp1的onTouchEvent方法。
3. onTouchEvent方法返回值同dispatchTouchEvent方法。如果DOWN事件在某个View的onTouchEvent方法中返回true，那么其它的View的onTouchEvent将没有被执行的机会，换句话说对于同一次事件分发有且仅有一个控件能够处理事件。**只有DOWN事件的返回值才有意义**。其它类型事件的返回值并不会影响事件分发的流程。我们以v8的onTouchEvent的DOWN事件返回值为例。

   3.1. v8 DOWN事件返回true。表示v8处理该事件。在v8分发事件之前应该是vp1(onInterceptTouchEvent) -> vp4(onInterceptTouchEvent) -> v9(onTouchEvent)-> v8(onTouchEvent)。此时v8返回true。系统会中断vp4的child遍历(不再将事件交由v7分发)。向上回溯设置vp4的ft指向v8，vp1的ft指向vp4。

   3.2. v8 DOWN事件返回false。事件继续交由v8的亲兄弟v7分发。

4.DOWN事件的分发流程

DOWN事件分发到vp1，会调用vp1的onInterceptTouchEvent。这里有拦截和不拦截两种情况。

1. 如果返回true，vp1拦截DOWN事件，DOWN事件直接交由vp1的onTouchEvent处理。
2. 如果返回false，vp1不拦截DOWN事件，DOWN事件将会交由vp1的最后一个子View分发。即交由vp4分发。

拦截方法以此类推，如果vp4不拦截DOWN事件，将交由v9分发事件。因为v9是View类型。没有拦截方法，所以会直接调用v9的onTouchEvent方法。该方法有处理和不处理两种情况。

1. 如果返回false，v9不处理事件。那么事件继续向前分发交由v8分发，同理调用v8的onTouchEvent方法，v8不处理事件，继续交由v7处理。v7也不处理，vp4的子View到此遍历完成。此时vp4的ft为空，直接调用vp4的onTouchEvent方法。
2. 如果返回true，v9处理事件。系统会中断vp4的子View遍历，DOWN事件分发结束。同时往上递归回溯设置父View的ft对象。vp4的ft指向v9，vp1的ft指向vp4。

小结：DOWN事件是通过深度遍历分发事件的。

5. MOVE、UP事件的分发流程

MOVE事件分发到VP1。它与DOWN事件的区别是，它并不一定会调用onInterceptTouchEvent。它只有当vp1的ft不为空时才会调用onInterceptTouchEvent方法，否则会直接拦截掉事件。

1. 当vp1的ft为空。直接拦截掉MOVE事件。调用VP1的onTouchEvent方法，注意这里onTouchEvent方法的返回值不会影响事件分发的流程。
2. 当vp1的ft不为空(当有后代View的onTouchEvent方法返回了true)。调用onInterceptTouchEvent方法，如果返回true，同上，直接调换用VP1的onTouchEvent方法。如果返回false，会通过ft的单链表线性分发事件。

UP事件同MOVE事件。这里就不分析了。

小结：MOVE、UP事件是通过线性遍历分发的。

6. CANCEL事件触发时机以及分发流程

前面我们讲到的DOWN、MOVE、UP事件都是由手触摸屏幕产生的。并没有讲到CANCEL是如何产生的。CANCEL不是由手触摸屏幕产生的。它是由系统生成。并且分发给View的。有一种场景会触发系统产生CANCEL事件。
还是上面的控件树。假设手机在屏幕的上半部分，所有的ViewGroup都不拦截事件，V9处理DOWN事件，当划动到屏幕的下半部分时，VP1拦截MOVE事件。当手指从上划动到下面时。系统将在vp1处，产生一个CANCEL事件，交由vp4分发。CANCEL事件的分发也是通过ft线性分发。当ViewGroup分发CANCEL事件后，会将ViewGroup的ft置为空。即将VP1，VP4的ft置为空。

9.14 事件冲突怎么解决？

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，这个问题也是同样也是开发中最常遇到的问题。

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，在此就不赘述了。

外部拦截法

外部拦截法直接在父布局中判断是否需要自己处理该事件，如果达到自己需要拦截的条件，直接在onInterceptTouchEvent返回为true，中断该手势后续事件的下发，逻辑上更加简单。实例代码如下：

```
private int mLastXIntercept;
private int mLastYIntercept;
@Override
public boolean onInterceptTouchEvent(MotionEvent event) {
    int x = (int) event.getX();
    int y = (int) event.getY();
    final int action = event.getAction();
    switch (action) {
        case MotionEvent.ACTION_DOWN:
            mLastXIntercept = (int) event.getX();
            mLastYIntercept = (int) event.getY();
            break;
        case MotionEvent.ACTION_MOVE:
            if (needIntercept) {//判断是否需要拦截的条件
                return true;
            }
            break;
        case MotionEvent.ACTION_UP:
            break;
    }
    return super.onInterceptTouchEvent(event);
}
```

9.15 View分发反向制约的方法？

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 的值又可以影响 disallowIntercept 的值，由此可见，子 View 通过调用 requestDisallowInterceptTouchEvent 方法，传入 true 即可让父容器对子View自己的拦截失效。
不过有一个注意事项，如果是down事件，那么该方法没有效果的，因为在上面 onInterceptTouchEvent 这块代码在执行前，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。

9.16 View中onTouch，onTouchEvent和onClick的执行顺序

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 函数中。

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 onTouchEvent 方法不会执行。反之 onTouchEvent 方法才会执行。

onClick的执行

view内置诸如click事件的实现等等都基于onTouchEvent的performClick方法，假如onTouch返回true，这些事件将不会被触发。

更多细节，大家可以去参考7.4节中View的事件管理

9.17 怎么拦截事件？如果 onTouchEvent返回false，onClick还会执行么？

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。具体拦截的原理说明代码如下：

```
if (actionMasked == MotionEvent.ACTION_DOWN
                    || mFirstTouchTarget != null) {
    if (!disallowIntercept) {
        intercepted = onInterceptTouchEvent(ev);
    }
}
// code 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code1处的if不会被命中，从而事件不会分发给子View。更详细的代码大家可以参考问题7.3 中的相关解析。

onTouchEvent如果返回false onClick还会执行么？

关于这个问题，首先需要强调一下，onClick 是在onTouchEvent方法里面执行的。大家可以看下面的代码

```
public boolean onTouchEvent(MotionEvent event) {
    ...
    performClickInternal（）；
    ...
}
```

performClickInternal（ ）函数里面会调用performClick()，然后在performClick()里面会调用 onClick(), 通过这个调用关系大家应该会发现，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。

9.18 事件的分发机制，责任链模式的优缺点

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

关于事件分发机制，大家可以去参考7.3题事件分发机制详细解释，这里就不再重复讲解了。

责任链模式的定义

责任链模式是行为型设计模式中的一个。
责任链模式的定义：为了避免请求发送者与多个请求处理者耦合在一起，于是将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，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：

1.降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。

2.增强了系统的可扩展性。可以根据需要增加新的请求处理类，满足开闭原则。

3. 增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。

4.责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if…else 语句。

5.责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

责任链模式的缺点：

1.不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理。
2.对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。

3.职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。

9.19 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,这个问题在开发中非常常见，如果我们使用了这种展示列表的方案，基本上都会遇到一些需要解决的冲突问题，所以面试官是问我们的实践情况。所以，建议大家用过这个场景实现app UI的都可以好好的总结一下，实践一下。

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

- ScrollView和RecyclerView两者都会滑动，而且有时候，滑动会冲突
- 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对象通过此方法关闭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
该方法的核心点在于通过重写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
该方法的核心点在于通过重写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。

9.20 View.inflater过程与异步inflater

这道题想考察什么？

考察同学对xml解析过程，以及异步解析过程，但是异步解析在使用过程中有诸多限制，所以使用的并不太多。

考生应该如何回答

Inflate解析

我们先分析一下View.inflate 的代码：

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

上面的代码不难发现是调用了LayoutInflater.inflate，于是我们将目光放到这个函数里面来

```
// 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 {
                // 获取 xml 布局的根 View 对象，比如 LinearLayout 对象，FrameLayout
对象等
                final View temp = createViewFromTag(root, name, inflaterContext,
attrs);
                ViewGroup.LayoutParams params = null;
                // 第一种情况：root != null &&attachToRoot 为false，通过 root 来获取
根节点的布局参数 ViewGroup.LayoutParams 对象,也就是说，把 xml 中的根节点的 layout_ 开头
的属性，如layout_width 和 layout_height 对应的值转为布局参数对象中的字段值，如width 和
height 值
                if (root != null) {
                    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，这意味着此时的temp没有解析xml中的layout属性
```

```
                if (root == null || !attachToRoot) {
                    result = temp;
                }
            }
        }
        // 返回result，可能是View，也可能是root
        return result;
    }
}
```

从上面的代码大家应该不难发现，inflate 主要是解析xml的属性，并生成xml的根布局temp，并且基于inflate的参数情况，来确定是否将temp添加到 解析它的 外层布局上面，也就是root变量上面。

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，注意：**此时的view是没有解析它xml中的layout params属性的，所以，xml中设置的宽高将可能失效。**

接下来，我们分析一下createViewFromTag 源码

```
// 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android 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 用户自定义View
            view = createView(context, name, null, attrs);
        }
    }
    ...
    return view;
}
```

从上面的函数我们发现他们主要是执行了onCreateView函数，那么onCreateView又做了什么呢？

```
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;
        }
    }
}
```

通过上面大家不难发现 onCreateView 是在通过反射的方式去创建XML中定义的view。并将他们构建成

小结

整个Inflater.inflate的过程，其实就是就是先进行xml解析，拿到xml中相关的tag的情况下，然后通过反射创建tag对应的View对象的一个流程。

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。

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作的高响应。

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
    }
```

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 sClassPrefixList 中做描述的这三个前缀的 view，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

inflate

AsyncLayoutInflater 中有一个非常重要的函数：inflate，那么这个函数是怎样的呢？我们一起看代码

```
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 就避免了这一点。那么InflateRequest是什么呢？大家参考下面的代码

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

从上面的代码不难发现，InflateRequest 其实就是一个类，这个类代表了一个xml解析的请求所需要的基本信息。

然后，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);
        }
    }
```

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);
            }
        }
    }
```

总结：

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；

SyncLayoutInflater 就是用于异步加载layout的，可以用于懒加载中，或者是在用户想提升onCreate函数的执行效率的时候。但是，它有几个缺陷：1）异步转换出来的 View 并没有被加到 parent view中，AsyncLayoutInflater 是调用了 LayoutInflater.inflate(int, ViewGroup, false)，因此如果需要加到 parent view 中，就需要我们自己手动添加； 2）不支持加载包含 Fragment 的 layout。

9.21 inflater创建view效率为什么比new View慢？

这道题想考察什么？

这个题目主要是考察大家对Inflate源码的理解，和对xml解析的理解。

考生应该如何回答

在出现Jetpack Compose之前，android的UI界面的设计大多数都是在xml里面定义的。XML只是一种界面布局的文件，这种文件是不能直接在app中显示的，它的显示是需要 inflate 解析后转变成为 java 中的view。那么Inflate是如何去解析xml呢？大家可以到LayoutInflater.java 文件中看到inflate 函数，这个函数就是通过xml里面的各个标签，使用序列化的方法拿到各个标签，然后将标签存储在Java类里面。有了java标签，那么inflate函数就可以大量的使用反射，通过反射找到这些标签所对于的类，从而能够通过反射获取到这些类的对象。也就是说，Inflate 是通过反射xml标签中的类开创建view的，这个反射过程相对比较耗时，因此它比直接在java中创建View的效率要高。

更多的关于view 的inflater的细节可以参考9.20 题。

9.22 动画的分类以及区别

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

动画的种类

首先回答下安卓的动画种类，一共分为三种

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

动画的区别

帧动画

帧动画其实就是简单的收集N张图片，然后依次显示这些图片。由于人眼"视觉暂留"的原因，会让我们造成动画的"错觉"。视觉暂留是指我们看到的画面，会在大脑中停留短暂的时间而不立即消失。

帧动画的特点：帧动画不会改变控件的属性，只是通过播放图片来达到动画效果，制作简单但效果单一，且占用空间较大。

补间动画

补间动画会先声明好两个关键帧，开始帧和结束帧。开始帧是用来描述动画开始时的状态，结束帧是用来描述动画结束时的状态，而动画中间如何由开始帧演变到结束帧是由系统计算而来。

补间动画的特点：安卓中补间动画只改变了View的显示效果，而未改变View的真正属性值。比如说，我们使用补间动画放大View的宽度，在动画运行中View宽度变大了，但仅仅只是显示效果上宽度变大了，并没有真正改变View宽度的值。

属性动画

属性动画与其他动画最大的不同是它会不断的更新View的属性值，从而产生动画效果。它不仅仅显示效果发生了改变，View的属性值也发生了变化。

属性动画和补间动画一样会声明关键帧，关键帧与关键帧之间的属性变化由系统计算而来。它们之间有个小的不同点是，属性动画可以声明一个、两个或者多个关键帧，而补间动画固定是开始帧和结束帧。

9.23 属性动画的原理是怎么样的？

这道题想考察什么？

这道题想考察同学对 动画 的理解。

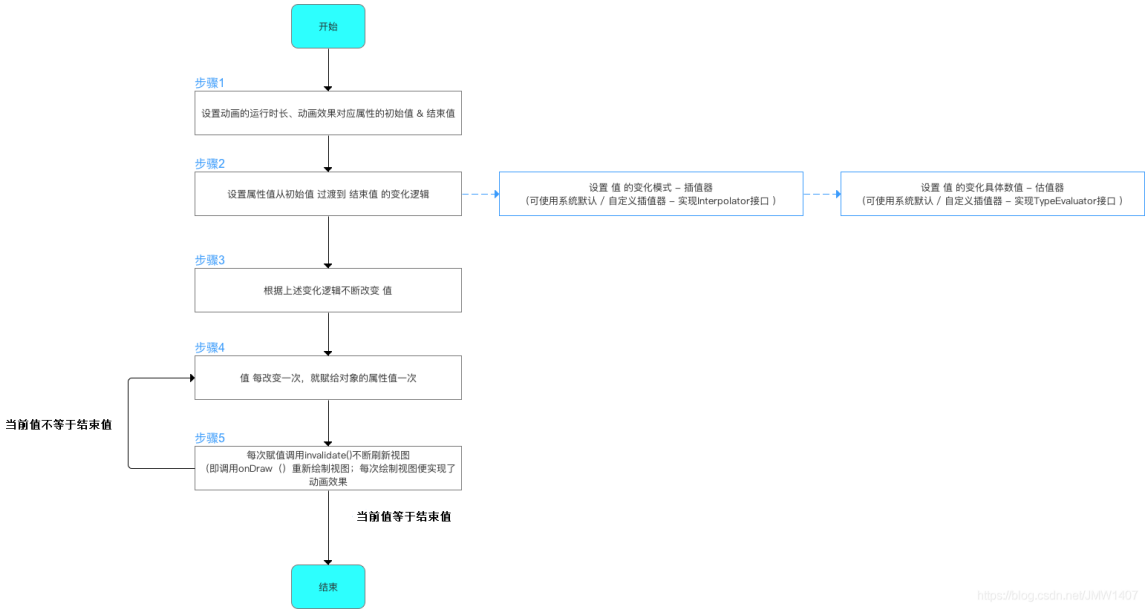
考生应该如何回答

属性动画的原理很简单，可以用一句话描述：最开始先设定好动画的基本属性信息，然后属性的数值按照设定逻辑变化并刷新视图（变化的属性会带来视图效果的变化），接着会判断动画是否达到结束条件，如果没有达到则重复数值的变化和视图刷新直到结束。

具体的细节我们可以看后面的分析。

属性动画可以分解为以下五个步骤：

1. 设置动画的基本信息，包括运行时间、动画效果、开始值、结束值。
2. 设置属性值的变化逻辑，包括插值器、估值器。
3. 根据变化的逻辑不断的改变值
4. 值改变后，就赋给对象的属性
5. 调用invalidate刷新视图，并且判断当前的数值是否为结束值。如果当前值为结束值，则动画结束。如果当前值不等于结束值则重复第4步。



属性动画原理详解

属性动画的重要类

关键帧KeyFrame

关键帧KeyFrame的注释是"This class holds a time/value pair for an animation"，它包含时间和数值两个属性，用来描述动画运行中多个画面中的一个。KeyFrame的时间属性描述了这个画面发生在整个动画的什么时刻，keyFrame的数值属性描述了当前时刻属性的值。

在属性动画的创建过程中，我们会设定开始值和结束值。在框架内部，开始值和结束值最终会转化成为关键帧KeyFrame。

```
public static KeyframeSet ofFloat(float... values) {
        boolean badValue = false;
        int numKeyframes = values.length;
        FloatKeyframe keyframes[] = new FloatKeyframe[Math.max(numKeyframes,2)];
        if (numKeyframes == 1) {
```

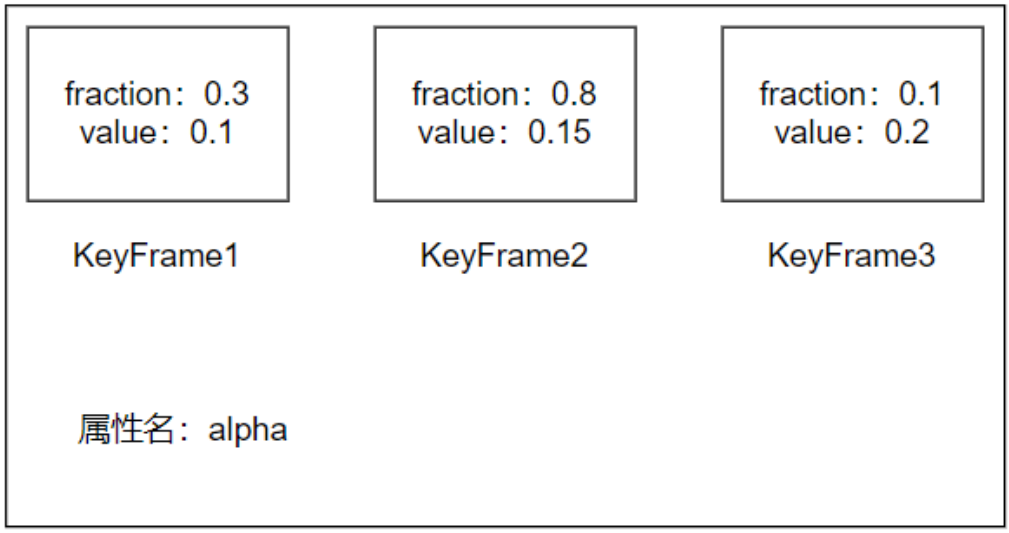
```
            keyframes[0] = (FloatKeyframe) Keyframe.ofFloat(0f);
            keyframes[1] = (FloatKeyframe) Keyframe.ofFloat(1f, values[0]);
            if (Float.isNaN(values[0])) {
                badValue = true;
            }
        } else {
            keyframes[0] = (FloatKeyframe) Keyframe.ofFloat(0f, values[0]);
            for (int i = 1; i < numKeyframes; ++i) {
                keyframes[i] =
                        (FloatKeyframe) Keyframe.ofFloat((float) i /
(numKeyframes - 1), values[i]);
                if (Float.isNaN(values[i])) {
                    badValue = true;
                }
            }
        }
        if (badValue) {
            Log.w("Animator", "Bad value (NaN) in float animator");
        }
        return new FloatKeyframeSet(keyframes);
    }
```

从上述代码过程中可以看到，参数values最终会转化为KeyFrame关键帧。这个里面会有两种情况，一种情况是values长度为一，另外一种是大于一。为一的情况，会生成开始帧和结束帧两个关键帧，开始帧数据都为0，结束帧的数据是传入的参数。不为一的情况，就是比较简单，values长度为多少就生成多少的关键帧。

PropertyValuesHolder

ObjectAnimator只能对单个属性进行操作，如果想实现比较复杂的效果就需要用到PropertyValuesHolder了。

PropertyValuesHolder的注释是"This class holds information about a property and the values that that property should take on during an animation"，它包含了属性，以及该属性在动画运行期间的值。简单来讲就是，包含了属性名和多个关键帧。属性名表明了是哪个属性需要发生变化，关键帧表明动画该如何变化。



属性持有者
PropertyValuesHolder

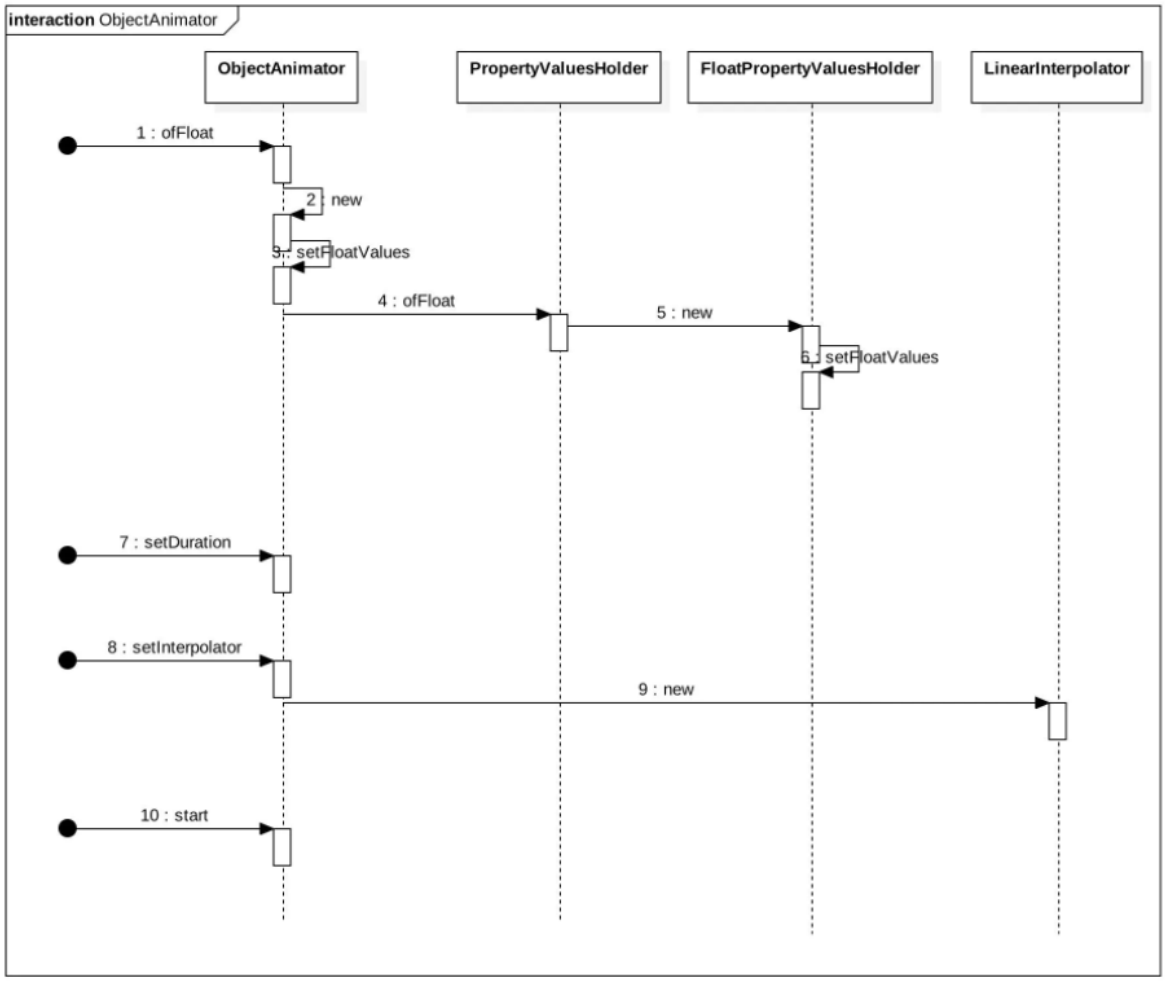
从上图看来，PropertyValuesHolder包含了属性名和多个关键帧。这里面属性是透明度，描述的是透明度的动画变化，在实际运用过程中可以换成其他属性。KeyFrame中的fraction代表的是时间、value表示数值，三个KeyFrame表示三个时间点的属性值。

动画基本信息设定

```
var objectAnimation: ObjectAnimator = ObjectAnimator.ofFloat(textView,
"translationY", 0f, 400f)
objectAnimation.setDuration(5000)
objectAnimation.interpolator = DecelerateInterpolator()
objectAnimation.start()
```

如上面代码所示，我们设定了一个平移的动画。该动画作用于textview这个控件上，从0f的位置平移至400f的位置。动画的总共耗时5000ms，并且运用了DecelerateInterpolator动画变化速率越来越慢的插值器。这样一来我们把动画的目标控件、开始值、结束值、动画时间以及变化逻辑插值器都设定好了，包括了本章节开头提到的步骤一和步骤二的内容。

接下来我们研究一下代码内部的变化。首先通过ofFloat会生成一个ObjectAnimator对象，然后根据传入的属性值生成PropertyValuesHodler，接着根据传入的数值生成对应的关键帧，再设置相关的动画时间、插值器，整个动画的信息就设置完毕。



Start方法开启动画

调用start方法整个动画就开始运作起来，最终会进入了ValueAnimator的start方法。ValueAnimator#start方法中三件重要的事情：一是初始化估值器；二是计算属性值并设置到控件上；三是注册同步信号；

初始化估值器是通过调用startAnimation()，最后了会调用到initAnimation方法。在initAnimation方法里面会遍历PropertyValuesHolder，并逐个进行执行init。

```
 void initAnimation() {
        if (!mInitialized) {
            int numValues = mValues.length;
            for (int i = 0; i < numValues; ++i) {
                mValues[i].init();
            }
            mInitialized = true;
        }
    }
```

其中mValues代表的是PropertyValuesHolder的集合，看完下面的代码，一切都很清晰明了了。

```
 void init() {
        if (mEvaluator == null) {
            // We already handle int and float automatically, but not their
Object
            // equivalents
            mEvaluator = (mValueType == Integer.class) ? sIntEvaluator :
```

```
                    (mValueType == Float.class) ? sFloatEvaluator :
                    null;
        }
        if (mEvaluator != null) {
            // KeyframeSet knows how to evaluate the common types - only give it
a custom
            // evaluator if one has been set on this class
            mKeyframes.setEvaluator(mEvaluator);
        }
    }
```

系统会判断当前是否有估值器对象，如果没有会根据mValueType的类型默认分配一个，接着就是简单的设置。

计算属性值是通过调用setCurrentPlayTime方法，最终会调用到animateValue()。在animateValue()中，会先通过插值器mInterpolator拿到当前动画的完成度，然后把完成度给到每一个PropertyValuesHolder用于计算动画的属性值。

```
 void animateValue(float fraction) {
        //计算完成度
        fraction = mInterpolator.getInterpolation(fraction);
        mCurrentFraction = fraction;
        int numValues = mValues.length;
        for (int i = 0; i < numValues; ++i) {
            //计算动画的数据
            mValues[i].calculateValue(fraction);
        }
        ...
    }
```

mValues[i].calculateValue最终会调用到KeyFrameSet的getValue方法，看完这个代码其实一切都很明了了。

```
public Object getValue(float fraction) {
    if (mNumKeyframes == 2) {
        if (mInterpolator != null) {
            fraction = mInterpolator.getInterpolation(fraction);
        }
        return mEvaluator.evaluate(fraction, mFirstKeyframe.getValue(),
                mLastKeyframe.getValue());
    }
    ...
    for (int i = 1; i < mNumKeyframes; ++i) {
            Keyframe nextKeyframe = mKeyframes.get(i);
            if (fraction < nextKeyframe.getFraction()) {
                final TimeInterpolator interpolator =
nextKeyframe.getInterpolator();
                final float prevFraction = prevKeyframe.getFraction();
                float intervalFraction = (fraction - prevFraction) /
                    (nextKeyframe.getFraction() - prevFraction);
                // Apply interpolator on the proportional duration.
                if (interpolator != null) {
                    intervalFraction =
interpolator.getInterpolation(intervalFraction);
```

```
                }
                return mEvaluator.evaluate(intervalFraction,
prevKeyframe.getValue(),
                        nextKeyframe.getValue());
            }
            prevKeyframe = nextKeyframe;
        }
    ...
}
```

上述代码中分了两种情况，一种情况是关键帧数目为二，一种不为二。为二的情况会传入完成度fraction、开始帧的值、结束帧的值给到估值器，由估值器最终计算数据。不为二的情况会通过fraction完成度寻找最合适的前帧prevKeyframe和后帧nextKeyframe，之后进行的内容就和关键帧为二的情况十分类似，通过完成度和前帧数值、后帧数值计算当前的属性值。

设置属性值的代码在Object的animateValue方法：

```
int numValues = mValues.length;
        for (int i = 0; i < numValues; ++i) {
            mValues[i].setAnimatedValue(target);
 }
```

上面代码其实是简单的遍历PropertyValuesHolder，然后给每一个PropertyValuesHolder设置动画数值。属性动画是通过反射的方式设置数值的，在具体代码中表现是先拿取到属性set方法的Method对象，当需要使用的时候直接invoke。

```
void setAnimatedValue(Object target) {
           ...
            if (mSetter != null) {
                try {
                    mTmpValueArray[0] = mIntAnimatedValue;
                    //反射设置属性
                    mSetter.invoke(target, mTmpValueArray);
                } catch (InvocationTargetException e) {
                    Log.e("PropertyValuesHolder", e.toString());
                } catch (IllegalAccessException e) {
                    Log.e("PropertyValuesHolder", e.toString());
                }
            }
            ...
        }
```

当调用ValueAnimator的start方法的时候，则会开始注册同步服务信号。注册同步服务信号其实就是向系统请求刷新屏幕。ValueAnimator的start方法最终会调用到MyFrameCallbackProvider的postFrameCallback方法，如下所示：

```
 public void postFrameCallback(Choreographer.FrameCallback callback) {
            mChoreographer.postFrameCallback(callback);
}
```

mChoreographer是编舞者，管理app端向系统端申请同步服务信号的事情。当系统执行同步，也就是刷新的时候，会回调callback。我们来看下callback中的代码。

```
 private final Choreographer.FrameCallback mFrameCallback = new
Choreographer.FrameCallback() {
        @Override
        public void doFrame(long frameTimeNanos) {
            doAnimationFrame(getProvider().getFrameTime());
            //判断动画是否结束
            if (mAnimationCallbacks.size() > 0) {
                getProvider().postFrameCallback(this);
            }
        }
    };
```

callback中的代码就只有两个内容：一个是执行doAnimationFrame，主要是执行新一轮的数值计算；另外一个是判断动画是否满足结束条件。

doAnimationFrame函数最终会调用到ValueAnimator#animateValue方法，其实也就是执行新一轮的动画的数值计算，先通过插值器、时间算出完成度，然后通过估值器、前后帧的值算出动画的数值，最后设置到控件上。

mAnimationCallbacks.size() > 0则说明动画未结束，继续申请一下次的同步服务信号，也就是需要继续刷新界面。不满足结束条件就继续刷新，执行回调的时候又判断是否需要继续刷新。类似于一种循环的方式，不断的推动动画的运行，直至满足条件动画结束。

总结

综上所述，你可以把动画的原理看中两个部分：一个部分与数据计算有关，另外一个部分与刷新界面有关。数据计算有关的部分，包括到关键帧、插值器、估值器、动画时间，它控制着动画运行过程中每个画面的数值。刷新界面有关就是不断的向系统申请刷新，当系统下发刷新的时候先运行动画的数据计算，然后判断动画是否结束。如果结束则不再想系统申请刷新，如果没有结束继续申请，直至满足条件结束。

9.24 动画插值器与估值器是什么？

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

插值器和估值器是动画运行过程中非常重要的内容。简单来说插值器是负责计算动画完成的百分比，换一句话说插值器也就是用来控制动画执行的变化速率的，当然也可以叫加速器。而估值器则根据动画完成百分比计算改变后的属性值，具体请参照下面内容。

动画是一系列的图片组合的一个变化过程，那么这个变化应该分为两个部分：1）动画变化的速率也就是“动”或者“移动” 的速率；2）每“移动”或者变化后展示的图片效果。由此，控制动画的变化就有了两个非常重要的工具，由**Interpolator**也就是插值器来控制的变化速率，以及由**Evaluator**估值器来计算的每次“移动”或者变化后展示的图片的效果。

Interpolator（插值器）

定义

- 它是一个接口

- 通俗易懂的说，Interpolator负责控制动画变化的速率，即确定了动画效果变化的模式，使得基本的动画效果能够以匀速、加速、减速、抛物线速率等各种速率变化。

  动画是开发者给定开始和结束的“关键帧”，其变化的“中间帧”是由系统计算决定然后播放出来。因此，动画的每一帧都将在开始和结束之间的特定时间显示。此时动画时间被转换为时间索引，则动画时间轴上的每个点都可以转换成0.0到1.0之间的一个浮点数，然后再将该值用于计算该对象的属性变换。在变换的情况下，y轴上，0.0对应于起始位置，1.0对应于结束位置，0.5对应于起始和结束之间的中间值。

Interpolator 本质上是一个数学函数，其取数字在0.0和1.0之间，并将其转换为另一个数字。

分类

java类	xml资源id	说明
AccelerateDecelerateInterpolator	@android:anim/accelerate_decelerate_interpolator	其变化开始和结束速率较慢，中间加速
AccelerateInterpolator	@android:anim/accelerate_interpolator	其变化开始速率较慢，后面加速
DecelerateInterpolator	@android:anim/decelerate_interpolator	其变化开始速率较快，后面减速
LinearInterpolator	@android:anim/linear_interpolator	其变化速率恒定
AnticipateInterpolator	@android:anim/anticipate_interpolator	其变化开始向后甩，然后向前
AnticipateOvershootInterpolator	@android:anim/anticipate_overshoot_interpolator	其变化开始向后甩，然后向前甩，过冲到目标值，最后又回到了终值
OvershootInterpolator	@android:anim/overshoot_interpolator	其变化开始向前甩，过冲到目标值，最后又回到了终值
BounceInterpolator	@android:anim/bounce_interpolator	其变化在结束时反弹
CycleInterpolator	@android:anim/cycle_interpolator	循环播放，其速率为正弦曲线
TimeInterpolator		一个接口，可以自定义插值器

TimeInterpolator接口只有一个方法getInterpolation(float input)。这个方法会在调度时间内返回从0~1变化值。所以我们变化不同的动画插值器，实际上就是在该方法内使用不同的曲线算法，不断的返回一个趋近于物理规律的曲线时刻点的变化值。在众多的插值器中，匀速插值器是相对容易理解的插值器，适合作为案例讲解，其他插值器采用的是类似的方案，大家感兴趣的可以自行去研究。

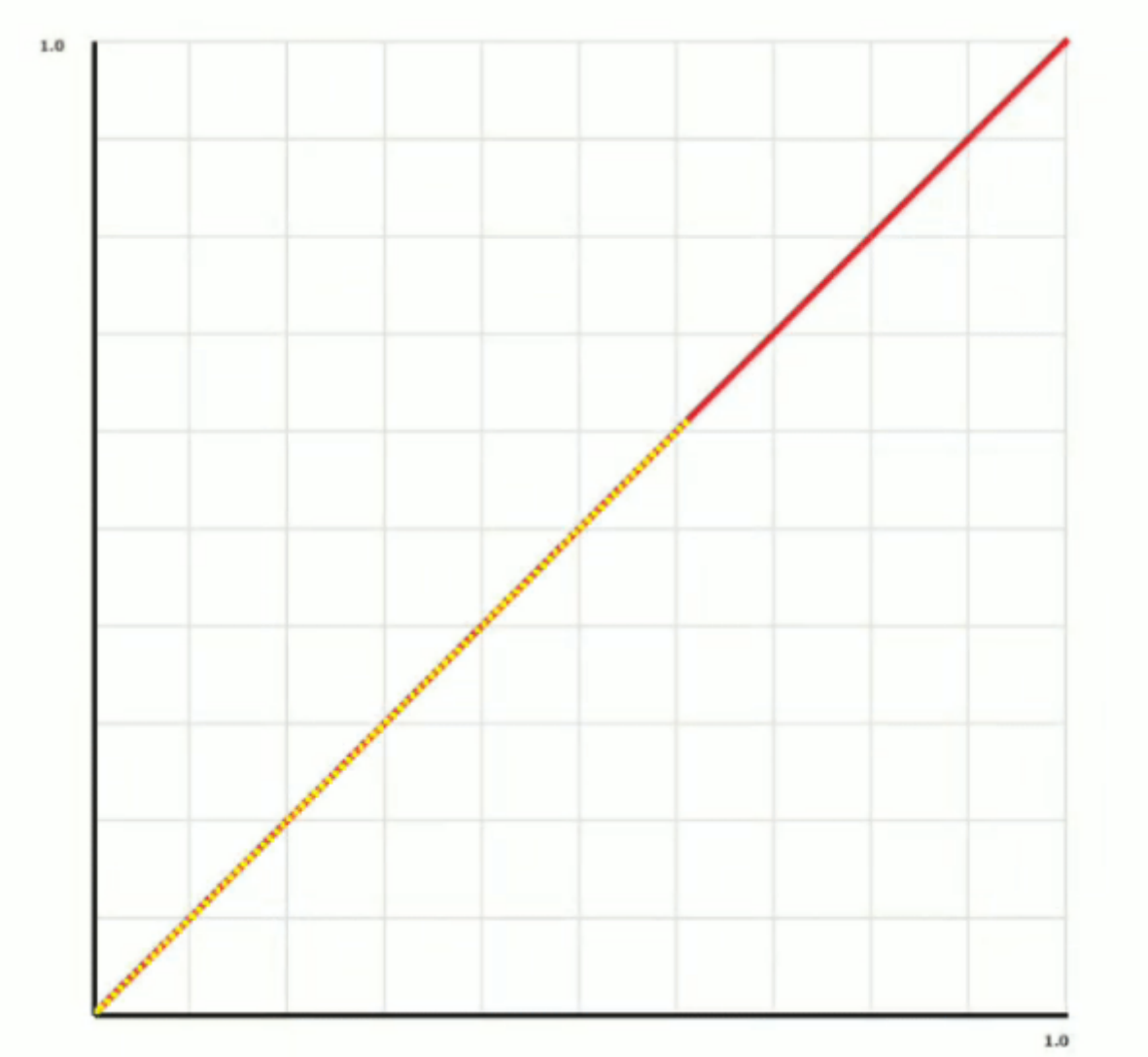
LinearInterpolator匀速插值器

匀速插值器的意思是随着时间变化，动画匀速变化，在整个过程中速率是一样的。

核心代码：

```
public class LinearInterpolator extends BaseInterpolator implements
NativeInterpolator {
    ...//省略部分方法
    //输入多少返回多少，匀速变化
    public float getInterpolation(float input) {
        return input;
    }
}
```

在上述代码中，getInterpolation函数传入了参数input，在未做任何处理的情况下又将input作为参数返回。整个数学模式表达的意思时间和动画变化速率一致，类似于数学函数中的f(x) = x，用坐标轴表示类似于下图，横轴作为时间，纵轴作为动画变化。



使用`Android`内置的插值器能满足大多数的动画需求。不管你是速率的要求是匀速、先加速后减速、先减速后加速、先慢后快再慢、先快后慢再快，内置的插值器都可以满足。如果实在是动画效果特殊，也可以自定义插值器。自定义插值器需要实现`TimeInterpolator`接口并复写getInterpolation()。

插值器（`Interpolator`）决定值的变化规律（匀速、加速），即决定的是变化趋势，而接下来的具体变化数值则交给 而估值器。估值器是协助插值器 实现非线性运动的动画效果。

TypeEvaluator（类型估值算法，即估值器）

什么是估值器：根据当前属性改变的百分比来计算改变后的属性值。也就是基于属性的变化的比率来计算得到当前的属性的值。例如：颜色从红色变成白色的动画，那么这个渐变的过程中，每一步变化都会有一个新的颜色，这个颜色怎么来的呢？它就是估值器计算出来的。估值器中有一个函数叫evaluate，这个函数的返回值，就是返回计算后的颜色。

- 插值器决定属性值随时间变化的规律；而具体变化后属性数值则交给估值器去计算。
- 是一个动画相关的接口。
- 作用：设置属性值从初始值过渡到结束值的变化过程中每一步的具体数值。
- 属性动画特有的属性
- 协助插值器实现非线性运动的动画效果

如果你还是对估值器的概念不清楚的话，不妨以例子作为切入点，我们从例子慢慢分析。

我们用代码初始化ObjectAnimation的时候，有利用到系统默认提供的FloatEvaluator：

```
val objectAnimator =
ObjectAnimator.ofObject(viewHolder,"PropertyName",FloatEvaluator(),0f,1f)
```

这时候，我们定义的值的类型是Float，直接看看FloatEvaluator的源码，只有一个方法：

```
public Float evaluate(float fraction, Number startValue, Number endValue) {
    float startFloat = startValue.floatValue();
    return startFloat + fraction * (endValue.floatValue() - startFloat);
}
```

- startValue 是指初始值
- endValue 是指最终值
- fraction是指从startValue到endValue的动画的进度，这个值是从0到1的float值。其变化的速率和插值器是有关联的，假设插值器使用的是LinearInterpolator线性插值器，那么fraction的变化也是线性的！
- return 生成的具体值。在上个Demo中，就是直接操控View的透明度的值。

那么估值器在这段时间内会一值被调用 evaluate() 方法，fraction始终是从0到1的变化，而startValue和endValue会以下面的方式变化：

startValue	endValue
0	1.0
.....	.....
0	1.0
1.0	2.0
.....	.....
1.0	2.0
动画结束	动画结束

需要注意的是：因为估值器返回的值会直接影响显示的效果，如果把fraction看作是x，startValue和endValue是常量，其返回值y必须满足与`x*(endValue - startValue)`构成的等式为一元一次方程。

使用方法

```
ObjectAnimator anim = ObjectAnimator.ofObject(myView2, "height", new
Evaluator(), 1, 3);
// 在第4个参数中传入对应估值器类的对象
// 系统内置的估值器有3个：
// IntEvaluator：以整型的形式从初始值 - 结束值 进行过渡
// FloatEvaluator：以浮点型的形式从初始值 - 结束值 进行过渡
// ArgbEvaluator：以Argb类型的形式从初始值 - 结束值 进行过渡
```

如果要为 Android 系统无法识别的类型添加动画效果，则可以通过实现 TypeEvaluator 接口来创建您自己的评估程序。Android 系统可以识别的类型为 int、float 或颜色，分别由 IntEvaluator、FloatEvaluator 和 ArgbEvaluator 类型评估程序提供支持。

- IntEvaluator ：Int类型估值器，返回int类型的属性改变
- FloatEvaluator ： Float类型估值器，返回Float类型属性改变
- ArgbEvaluator ：颜色类型估值器，返回16进制颜色值

自定义估值器

> 本质：根据插值器计算出当前属性值改变的百分比 & 初始值 & 结束值来计算当前属性具体的数值

如：动画进行了50%（初始值=100，结束值=200 ），那么匀速插值器计算出了当前属性值改变的百分比是50%，那么估值器则负责计算当前属性值 = 100 + （200-100）x 50% = 150。

具体使用：自定义估值器需要实现 `TypeEvaluator接口` & `复写evaluate()`

```
public interface TypeEvaluator {

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

// 参数说明
// fraction：插值器getInterpolation（）的返回值
// startValue：动画的初始值
// endValue：动画的结束值

        ....// 估值器的计算逻辑

        return xxx;
        // 赋给动画属性的具体数值
        // 使用反射机制改变属性变化

    }
}
```

那么插值器的input值 和 估值器fraction有什么关系呢？

答：input的值决定了fraction的值：input值经过计算后传入到插值器的getInterpolation（ ），然后在getInterpolation（ ）中的基于响应的逻辑算法结合input值来计算出一个返回值，而这个返回值就是fraction了。

总结

属性动画是对属性做动画，属性要实现动画。

- 首先由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，然后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这个值，而是变化效果变化了0.5，那么怎么办呢？

- 插值器算好属性变化百分比之后，由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，我们就可以对View设置当前的属性值了。

9.25 优化帧动画之SurfaceView逐帧解析

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

逐帧动画在动画刚开始的时候会加载所有帧的图片，等于预加载了全部的图片。如果所含图片过大的话会影响app的性能，严重的可能会导致OOM。我们可以才有SurfaceView的方式逐帧解析加载避免上述情况的发生。

Android 提供了`AnimationDrawable`用于实现帧动画。在动画开始之前，所有帧的图片都被解析并占用内存，一旦动画较复杂帧数较多，在低配置手机上容易发生 OOM。即使不发生 OOM，也会对内存造成不小的压力。下面代码展示了一个帧数为4的帧动画：

```
AnimationDrawable drawable = new AnimationDrawable();
drawable.addFrame(getDrawable(R.drawable.frame1), frameDuration);
drawable.addFrame(getDrawable(R.drawable.frame2), frameDuration);
drawable.addFrame(getDrawable(R.drawable.frame3), frameDuration);
drawable.addFrame(getDrawable(R.drawable.frame4), frameDuration);
drawable.setOneShot(true);

ImageView ivFrameAnim = ((ImageView) findViewById(R.id.frame_anim));
ivFrameAnim.setImageDrawable(drawable);
drawable.start();
```

有没有什么办法让帧动画的数据逐帧加载，而不是一次性全部加载到内存？`SurfaceView`就提供了这种能力。

SurfaceView

屏幕的显示机制和帧动画类似，也是一帧一帧的连环画，只不过刷新频率很高，感觉像连续的。为了显示一帧，需要经历计算和渲染两个过程，CPU 先计算出这一帧的图像数据并写入内存，然后调用 OpenGL 命令将内存中数据渲染成图像存放在 GPU Buffer 中，显示设备每隔一定时间从 Buffer 中获取图像并显示。

上述过程中的计算，对于View来说，就好比在主线程遍历 View树 以决定视图画多大（measure），画在哪（layout），画些啥（draw），计算结果存放在内存中，SurfaceFlinger 会调用 OpenGL 命令将内存中的数据渲染成图像存放在 GPU Buffer 中。每隔16.6ms，显示器从 Buffer 中取出帧并显示。所以自定义 View 可以通过重载onMeasure()、onLayout()、onDraw()来定义帧内容，但不能定义帧刷新频率。

SurfaceView可以突破这个限制。而且它可以将计算帧数据放到独立的线程中进行。下面是自定义SurfaceView的模版代码：

```
public abstract class BaseSurfaceView extends SurfaceView implements
SurfaceHolder.Callback {
    public static final int DEFAULT_FRAME_DURATION_MILLISECOND = 50;
    //用于计算帧数据的线程
    private HandlerThread handlerThread;
    private Handler handler;
    //帧刷新频率
    private int frameDuration = DEFAULT_FRAME_DURATION_MILLISECOND;
    //用于绘制帧的画布
    private Canvas canvas;
    private boolean isAlive;
```

```
    public BaseSurfaceView(Context context) {
        super(context);
        init();
    }

    protected void init() {
        getHolder().addCallback(this);
        //设置透明背景，否则SurfaceView背景是黑的
        setBackgroundTransparent();
    }

    private void setBackgroundTransparent() {
        getHolder().setFormat(PixelFormat.TRANSLUCENT);
        setZOrderOnTop(true);
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        isAlive = true;
        startDrawThread();
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int
height) {
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        stopDrawThread();
        isAlive = false;
    }

    //停止帧绘制线程
    private void stopDrawThread() {
        handlerThread.quit();
        handler = null;
    }

    //启动帧绘制线程
    private void startDrawThread() {
        handlerThread = new HandlerThread("SurfaceViewThread");
        handlerThread.start();
        handler = new Handler(handlerThread.getLooper());
        handler.post(new DrawRunnable());
    }

    private class DrawRunnable implements Runnable {

        @Override
        public void run() {
            if (!isAlive) {
                return;
            }
            try {
                //1.获取画布
                canvas = getHolder().lockCanvas();
                //2.绘制一帧
```

```
                onFrameDraw(canvas);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                //3.将帧数据提交
                getHolder().unlockCanvasAndPost(canvas);
                //4.一帧绘制结束
                onFrameDrawFinish();
            }
            //不停的将自己推送到绘制线程的消息队列以实现帧刷新
            handler.postDelayed(this, frameDuration);
        }
    }

    protected abstract void onFrameDrawFinish();

    protected abstract void onFrameDraw(Canvas canvas);
}
```

- 用HandlerThread作为独立帧绘制线程，好处是可以通过与其绑定的Handler方便地实现“每隔一段时间刷新”，而且在`Surface`被销毁的时候可以方便的调用HandlerThread.quit()来结束线程执行的逻辑。
- DrawRunnable.run()运用模版方法模式定义了绘制算法框架，其中帧绘制逻辑的具体实现被定义成两个抽象方法，推迟到子类中实现，因为绘制的东西是多样的，对于本文来说，绘制的就是一张张图片，所以新建BaseSurfaceView的子类FrameSurfaceView：

逐帧解析 & 及时回收

```
public class FrameSurfaceView extends BaseSurfaceView {
    public static final int INVALID_BITMAP_INDEX = Integer.MAX_VALUE;
    private List<Integer> bitmaps = new ArrayList<>();
    //帧图片
    private Bitmap frameBitmap;
    //帧索引
    private int bitmapIndex = INVALID_BITMAP_INDEX;
    private Paint paint = new Paint();
    private BitmapFactory.Options options = new BitmapFactory.Options();
    //帧图片原始大小
    private Rect srcRect;
    //帧图片目标大小
    private Rect dstRect = new Rect();
    private int defaultWidth;
    private int defaultHeight;

    public void setDuration(int duration) {
        int frameDuration = duration / bitmaps.size();
        setFrameDuration(frameDuration);
    }

    public void setBitmaps(List<Integer> bitmaps) {
        if (bitmaps == null || bitmaps.size() == 0) {
            return;
        }
        this.bitmaps = bitmaps;
        //默认情况下，计算第一帧图片的原始大小
        getBitmapDimension(bitmaps.get(0));
```

```
    }

    private void getBitmapDimension(Integer integer) {
        final BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeResource(this.getResources(), integer, options);
        defaultWidth = options.outWidth;
        defaultHeight = options.outHeight;
        srcRect = new Rect(0, 0, defaultWidth, defaultHeight);
        requestLayout();
    }

    public FrameSurfaceView(Context context) {
        super(context);
    }

    @Override
    protected void onLayout(boolean changed, int left, int top, int right, int
bottom) {
        super.onLayout(changed, left, top, right, bottom);
        dstRect.set(0, 0, getWidth(), getHeight());
    }

    @Override
    protected void onFrameDrawFinish() {
        //在一帧绘制完后，直接回收它
        recycleOneFrame();
    }

    //回收帧
    private void recycleOneFrame() {
        if (frameBitmap != null) {
            frameBitmap.recycle();
            frameBitmap = null;
        }
    }

    @Override
    protected void onFrameDraw(Canvas canvas) {
        //绘制一帧前需要先清画布，否则所有帧都叠在一起同时显示
        clearCanvas(canvas);
        if (!isStart()) {
            return;
        }
        if (!isFinish()) {
            drawOneFrame(canvas);
        } else {
            onFrameAnimationEnd();
        }
    }

    //绘制一帧，是张Bitmap
    private void drawOneFrame(Canvas canvas) {
        frameBitmap = BitmapUtil.decodeOriginBitmap(getResources(),
bitmaps.get(bitmapIndex), options);
        canvas.drawBitmap(frameBitmap, srcRect, dstRect, paint);
        bitmapIndex++;
    }
```

```
    private void onFrameAnimationEnd() {
        reset();
    }

    private void reset() {
        bitmapIndex = INVALID_BITMAP_INDEX;
    }

    //帧动画是否结束
    private boolean isFinish() {
        return bitmapIndex >= bitmaps.size();
    }

    //帧动画是否开始
    private boolean isStart() {
        return bitmapIndex != INVALID_BITMAP_INDEX;
    }

    //开始播放帧动画
    public void start() {
        bitmapIndex = 0;
    }

    private void clearCanvas(Canvas canvas) {
        paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.CLEAR));
        canvas.drawPaint(paint);
        paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.SRC));
    }
}
```

- FrameSurfaceView继承自BaseSurfaceView，所以它复用了基类的绘制框架算法，并且定了自己每一帧的绘制内容：一张Bitmap。
- Bitmap资源 id 通过setBitmaps()传递进来， **绘制一帧解析一张** ，在每一帧绘制完毕后，调用Bitmap.recycle()释放图片 native 内存并去除 java 堆中图片像素数据的引用。这样当 GC 发生时，图片像素数据可以及时被回收。

一切都是这么地能够自圆其说，我迫不及待地运行代码并打开AndroidStudio的Profiler标签页，切换到MEMORY，想用真实内存数据验证下性能。但残酷的事实狠狠地打了下脸，多次播放帧动画后，内存占用居然比原生AnimationDrawable还大，而且每播放一次，内存中都会多出 N 个Bitmap对象（N为帧动画总帧数）。唯一令人欣慰的是，手动触发 GC 后帧动画图片能够被回收。（AnimationDrawable中的图片数据不会被 GC）

原因就在于自作聪明地及时回收，每一帧绘制完后帧数据被回收，那下一帧解析Bitmap时只能新申请一块内存。帧动画每张图片大小是一致的，是不是能复用上一帧Bitmap的内存空间？于是乎有了下面这个版本的FrameSurfaceView：

逐帧解析 & 帧复用

```
public class FrameSurfaceView extends BaseSurfaceView {
    public static final int INVALID_BITMAP_INDEX = Integer.MAX_VALUE;
    private List<Integer> bitmaps = new ArrayList<>();
    private Bitmap frameBitmap;
    private int bitmapIndex = INVALID_BITMAP_INDEX;
    private Paint paint = new Paint();
    private BitmapFactory.Options options;
    private Rect srcRect;
```

```
    private Rect dstRect = new Rect();

    public void setDuration(int duration) {
        int frameDuration = duration / bitmaps.size();
        setFrameDuration(frameDuration);
    }

    public void setBitmaps(List<Integer> bitmaps) {
        if (bitmaps == null || bitmaps.size() == 0) {
            return;
        }
        this.bitmaps = bitmaps;
        getBitmapDimension(bitmaps.get(0));
    }

    private void getBitmapDimension(Integer integer) {
        final BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeResource(this.getResources(), integer, options);
        defaultWidth = options.outWidth;
        defaultHeight = options.outHeight;
        srcRect = new Rect(0, 0, defaultWidth, defaultHeight);;
    }

    public FrameSurfaceView(Context context) {
        super(context);
    }

    @Override
    protected void init() {
        super.init();
        //定义解析Bitmap参数为可变类型，这样才能复用Bitmap
        options = new BitmapFactory.Options();
        options.inMutable = true;
    }

    @Override
    protected void onLayout(boolean changed, int left, int top, int right, int
bottom) {
        super.onLayout(changed, left, top, right, bottom);
        dstRect.set(0, 0, getWidth(), getHeight());
    }

    @Override
    protected int getDefaultWidth() {
        return defaultWidth;
    }

    @Override
    protected int getDefaultHeight() {
        return defaultHeight;
    }

    @Override
    protected void onFrameDrawFinish() {
        //每帧绘制完毕后不再回收
//        recycle();
    }
```

```
    public void recycle() {
        if (frameBitmap != null) {
            frameBitmap.recycle();
            frameBitmap = null;
        }
    }

    @Override
    protected void onFrameDraw(Canvas canvas) {
        clearCanvas(canvas);
        if (!isStart()) {
            return;
        }
        if (!isFinish()) {
            drawOneFrame(canvas);
        } else {
            onFrameAnimationEnd();
        }
    }

    private void drawOneFrame(Canvas canvas) {
        frameBitmap = BitmapUtil.decodeOriginBitmap(getResources(),
bitmaps.get(bitmapIndex), options);
        //复用上一帧Bitmap的内存
        options.inBitmap = frameBitmap;
        canvas.drawBitmap(frameBitmap, srcRect, dstRect, paint);
        bitmapIndex++;
    }

    private void onFrameAnimationEnd() {
        reset();
    }

    private void reset() {
        bitmapIndex = INVALID_BITMAP_INDEX;
    }

    private boolean isFinish() {
        return bitmapIndex >= bitmaps.size();
    }

    private boolean isStart() {
        return bitmapIndex != INVALID_BITMAP_INDEX;
    }

    public void start() {
        bitmapIndex = 0;
    }

    private void clearCanvas(Canvas canvas) {
        paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.CLEAR));
        canvas.drawPaint(paint);
        paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.SRC));
    }
}
```

- 将Bitmap的解析参数inBitmap设置为已经成功解析的Bitmap对象以实现复用。

这一次不管重新播放多少次帧动画，内存中Bitmap数量只会增加1，因为只在解析第一张图片时分配了内存。而这块内存可以在FrameSurfaceView生命周期结束时手动调用recycle()回收。

总结

采用逐帧解析的方式可以有效的解决原生帧动画一次性加载全部动画图片带来的问题，而且按需加载的方式也更符合实际开发场景，不占用过多没有必要的内存资源。

9.26 WebView如何做资源缓存？

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

一般H5的加载流程：

1. 加载开始前做初始化工作，包括Runtime初始化，创建WebView等；
2. 完成初始化之后，WebView开始去请求资源加载H5页面；
3. 页面发起CGI请求对应的数据（或者通过本地缓存获取），拿到数据后对DOM进行操作更新。

从流程上看存在的直观的问题：

1. 终端耗时：终端初始化阶段的时间里（网络资料显示耗时在1s以上），网络完全是处于空闲等待状态的，浪费时间；
2. 资源和数据动态拉取，获取的速度受制于网络速度；

那么解决方案是怎样的呢？

针对上述问题，主流的解决方案有：

1. WebView自带的H5缓存机制
2. 预加载
3. 离线包

下面我们就重点分析一下缓存机制的解决方案。

1.缓存机制

H5中有很多的特性，其中就包括离线存储（也可称为缓存机制）这个重要的特性。加入了缓存机制，意味着web应用可以进行缓存，在没有网络的情况下进行离线访问。缓存机制带来的好处包括：

1. 离线访问：用户可以在离线环境下使用
2. 提高速度：缓存在本地的资源加载速度更快
3. 减少服务器压力：只需下载更新过的文件，无需每次加载页面都进行一次完整的请求流程

到目前为止，H5的缓存机制一共有六种，分别是：

1. 浏览器缓存机制
2. Application Cache（AppCache）机制

3. Dom Storgage（Web Storage）存储机制
4. Web SQL Database存储机制
5. Indexed Database（IndexedDB）
6. File System API

下面我们详细分析一下这几种缓存机制

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。

   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

   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。

   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。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。WebView内置自动实现，使用默认的CacheMode就可以实现。

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()
```

3.Dom Storage存储机制

Dom Storage的官方描述为：DOM 存储是一套在 Web Applications 1.0 规范中首次引入的与存储相关的特性的总称，现在已经分离出来，单独发展成为独立的 W3C Web 存储规范。 DOM存储被设计为用来提供一个更大存储量、更安全、更便捷的存储方法，从而可以代替掉将一些不需要让服务器知道的信息存储到 cookies里的这种传统方法。 Dom Storage机制类似Cookies，但有一些优势。Dom Storage是通过存储字符串的Key-Value对来提供的， Dom Storage存储的数据在本地，不像Cookies，每次请求一次页面，Cookies都会发送给服务器。 DOM Storage分为sessionStorage和localStorage，二者使用方法基本相同，区别在于作用范围不同：前者具有临时性，用来存储与页面相关的数据，它在页面关闭后无法使用，后者具备持久性，即保存的数据在页面关闭后也可以使用。

- sessionStorage 是个全局对象，它维护着在页面会话(page session)期间有效的存储空间。只要浏览器开着，页面会话周期就会一直持续。当页面重新载入(reload)或者被恢复(restores)时，页面会话也是一直存在的。每在新标签或者新窗口中打开一个新页面，都会初始化一个新的会话。
- localStorage保存的数据是持久性的。当前PAGE关闭（Page Session结束后），保存的数据依然存在。重新打开PAGE，上次保存的数据可以获取到。另外，Local Storage 是全局性的，同时打开两个 PAGE 会共享一份存数据，在一个PAGE中修改数据，另一个 PAGE 中是可以感知到的。

Dom Storage的优势在于：存储空间（5M）大，远远大于Cookies（4KB），而且数据存储在本地无需经常和服务器进行交互，存储安全、便捷。可用于存储临时的简单数据。作用机制类似于`SharedPreference`。但是，如果要存储结构化的数据，可能要借助JSON了，将要存储的对象转为JSON 串。不太适合存储比较复杂或存储空间要求比较大的数据，也不适合存储静态的文件。 使用方法如下：

```
webSettings.setDomStorageEnabled(true);
```

4. Web SQL Database存储机制

Web SQL Database基于SQL的数据库存储机制，用于存储适合数据库的结构化数据，充分利用数据库的优势，存储适合数据库的结构化数据，Web SQL Database存储机制提供了一组可方便对数据进行增加、删除、修改、查询。 Android系统也使用了大量的数据库用来存储数据，比如联系人、短消息等；数据库的格式为SQLite。Android也提供了API来操作SQLite。Web SQL Database存储机制就是通过提供一组API，借助浏览器的实现，将这种Native的功能提供给了Web。 实现方法为：

```
String cacheDirPath = context.getFilesDir().getAbsolutePath()+"cache/";
// 设置缓存路径
webSettings.setDatabasePath(cacheDirPath);
webSettings.setDatabaseEnabled(true);
```

Web SQL Database存储机制官方已不再推荐使用，也已经停止了维护，取而代之的是IndexedDB缓存机制

5. Indexed Database（IndexedDB）

IndexedDB也是一种数据库的存储机制，但不同于已经不再支持 Web SQL Database缓存机制。IndexedDB不是传统的关系数据库，而是属于NoSQL数据库，通过存储字符串的Key-Value对来提供存储（类似于Dom Storage，但功能更强大，且存储空间更大）。其中Key是必需的，且唯一的，Key可以自己定义，也可由系统自动生成。Value也是必需的，但Value非常灵活，可以是任何类型的对象。一般Value通过Key来存取的。 IndexedDB提供了一组异步的API，可以进行数据存、取以及遍历。IndexedDB有个非常强大的功能：index（索引），它可对Value对象中任何属性生成索引，然后可以基于索引进行Value对象的快速查询。 IndexedDB集合了Dom Storage和Web SQL Database的优点，用于存储大块或复杂结构的数据，提供更大的存储空间，使用起来也比较简单。可以作为 Web SQL Database的替代。但是不太适合静态文件的缓存。 Android在4.4开始支持IndexedDB，开启方法如下：

```
webSettings.setJavaScriptEnabled(true);
```

6. File System

File System是H5新加入的存储机制。它为Web App提供了一个运行在沙盒中的虚拟的文件系统。不同WebApp的虚拟文件系统是互相隔离的，虚拟文件系统与本地文件系统也是互相隔离的。Web App在虚拟的文件系统中，通过File System API提供的一组文件与文件夹的操作接口进行文件（夹）的创建、读、写、删除、遍历等操作。 浏览器给虚拟文件系统提供了两种类型的存储空间：临时的和持久性的：

- 临时的存储空间是由浏览器自动分配的，但可能被浏览器回收；
- 持久性的存储空间需要显示的申请，申请时浏览器会给用户一提示，需要用户进行确认。持久性的存储空间是 WebApp 自己管理，浏览器不会回收，也不会清除内容。存储空间大小通过配额管理，首次申请时会一个初始的配额，配额用完需要再次申请。

File System的优势在于：

- 可存储数据体积较大的二进制数据
- 可预加载资源文件
- 可直接编辑文件

遗憾的是：由于File System是H5新加入的缓存机制，目前Android WebView暂时还不支持。

总结

缓存机制汇总

名称	原理	优点	适用对象	说明
浏览器缓存	使用HTTP协议头部字段进行缓存控制	支持HTTP协议层	存储静态资源	Android默认实现
Dom Storage	通过存储键值对实现	存储空间大，数据在本地，安全便捷	类似Cookies，存储临时的简单数据	类似Android中的SP
Web SQL DataBase	基于SQL	利用数据库优势，增删改查方便	存储复杂、数据量大的结构化数据	不推荐使用，用IndexedDB替代
IndexedDB	通过存储键值对实现（NoSQL）	存储空间大、使用简单灵活	存储复杂、数据量大的结构化数据	集合Dom Storage和Web SQL DataBase的有点
AppCache	类似浏览器缓存，以文件为单位进行缓存	构建方便	离线缓存，存储静态资源	对浏览器缓存的补充
File System	提供一个虚拟的文件系统	可存储二进制数据、预加载资源和之间编辑文件	通过文件系统管理数据	目前Android不支持

9.27 WebView和JS交互的几种方式。

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

2.具体分析

2.1 Android通过WebView调用 JS 代码

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（）`

2.1.1 方式1：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（ ）

具体使用如下所示：

步骤1：将需要调用的JS代码以.html格式放到src/main/assets文件夹里

1. 为了方便展示，本文是采用Andorid调用本地JS代码说明；
2. 实际情况时，Android更多的是调用远程JS代码，即将加载的JS代码路径改成url即可

需要加载JS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

Android代码：MainActivity.java

```
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;
            }
        });

    }
}
```

小结：

上面的代码不难发现，基本流程是：1）进行websettings 的基本设置；2）导入js代码；3）进行其他操作相关的设置。

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。`onPageFinished()` 属于WebViewClient类的方法，主要在页面加载结束时调用

2.1.2 方式2：通过WebView的evaluateJavascript（）

优点：因为该方法的执行不会使页面刷新，而第一种方法（loadUrl ）的执行则会，因此该方法比第一种方法效率更高、使用更简洁。

具体使用方法

```
// 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
        }
    });
}
```

2.1.3 两种方法对比：

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

通过上面的对比，我们一般建议采用evaluateJavascript。

2.2 .JS通过WebView调用 Android 代码

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

2.2.1 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，仅将Android对象和JS对象映射即可。

缺点：存在严重的漏洞问题。

2.2.2 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

   即JS需要调用Android的方法

具体使用：

步骤1：在JS约定所需要的Url协议 *JS*代码：*javascript.html*

```
<!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>
```

当该JS通过Android的mWebView.loadUrl("file:///android_asset/javascript.html")加载后，就会回调shouldOverrideUrlLoading（ ），接下来继续看步骤2：

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);
        }
    });
  }
}
```

以上方式的特点

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。

如果JS想要得到Android方法的返回值，只能通过 WebView 的 loadUrl（ ）去执行 JS 方法把返回值传递回去，相关的代码如下：

```
// Android：MainActivity.java
mWebView.loadUrl("javascript:returnResult(" + result + ")");

// JS：javascript.html
function returnResult(result){
    alert("result is" + result);
}
```

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

在JS中，有三个常用的对话框方法：

方法	作用	返回值	备注
alert（）	弹出警告框	没有	在文本加入\n可换行
confirm（）	弹出确认框	两个返回值	• 返回布尔值； • 通过该值可判断点击时确认还是取消：true表示点击了确认，false表示点击了取消。
prompt（）	弹出输入框	任意设置返回值	• 点击“确认”：返回输入框中的值； • 点击“取消”：返回null。

方式3的原理：Android通过 WebChromeClient 的onJsAlert()、onJsConfirm()、onJsPrompt（ ）方法回调分别拦截JS对话框 （即上述三个方法），得到他们的消息内容，然后解析即可。

下面的例子将用拦截 JS的输入框（即prompt（ ）方法）说明 ：

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 *javascript.html*，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);
            }
```

```
            });
        }
    }
```

2.2.4 小结

对以上三种方式的对比&应用场景分析

调用方式	优点	缺点	使用场景
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 的方法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

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	

9.28 Android WebView拦截请求的方式

这道题想考察什么？

考察WebView 的基本用法中拦截请求的方式，这个也是开发中常用的技能

考生应该如何回答？

如果想做webview的本地缓存或者拦截请求做其他的事情就设计到了拦截webview的请求。

如何去做呢？

我们先简单的回顾一下Webview loadUrl的方式。

WebView的基本用法相信大多数android开发者都是会使用的，最简单的就是调用个loadUrl方法，但是记得要在清单文件中添加网络权限。

```
mWebview = (WebView) findViewById(R.id.my_webview);
mWebview.setWebViewClient(new WebViewClient() {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest
request)         {
        return false;
    }
});
mWebview.loadUrl("https://www.baidu.com/");
```

在Android自带的WebView中，如果需要对访问的URL或者资源进行拦截，主要涉及到WebViewClient中的三个方法：onPageStarted、shouldOverrideUrlLoading、shouldInterceptRequest。首先来分析onPageStarted方法和shouldOverrideUrlLoading方法，分别在两个方法以及onPageFinished方法中打印log。

a) 当用户使用WebView的loadUrl方法开启一个网页时，其中onPageStarted方法会执行，而shouldOverrideUrlLoading则不会执行
b) 当用户继续点击网页内的链接时，onPageStarted和shouldOverrideUrlLoading均会执行，并且shouldOverrideUrlLoading要先于onPageStarted方法执行
c) 当用户点击网页中的链接后，点击back，返回历史网页时，onPageStarted会执行，而shouldOverrideUrlLoading不会执行
综上所述，当需要对访问的网页进行策略控制时，需要在onPageStarted方法中进行拦截，如下示例代码:

```
@Override
public void onPageStarted(WebView view, String url, Bitmap favicon) {
    Log.d(TAG, "onPageStarted url is " + url);
    boolean res = checkUrl(url);
    //根据对URL的检查结果，进行不同的处理，
    //例如，当检查的URL不符合要求时，
    //可以加载本地安全页面，提示用户退出
    if (!res) {
        //停止加载原页面
        view.stopLoading();
        //加载安全页面
        view.loadUrl(LOCAL_SAFE_URL);
    }
}
```

然后，来分析一下shouldInterceptRequest(WebView view, String url)，此方法从Android API 11（3.0）开始提供，位于WebViewClient 内，当用户使用WebView的loadUrl方法打开网页、点击网页中的链接、返回历史网页时，所有资源的加载均会调用shouldInterceptRequest方法

进行资源替换时，可以将网页资源，例如html、css、js、图片等存放在本地，在shouldInterceptRequest对WebView加载的资源进行拦截，当符合某种策略时，替换为本地的资源，资源的MIME类型可以采用以下方法获取：
MimeTypeMap.getSingleton().getMimeTypeFromExtension(MimeTypeMap.getFileExtensionFromUrl(url))
示例：

```
@Override
public WebResourceResponse shouldInterceptRequest(WebView view, String url) {
    Log.d(TAG, "shouldInterceptRequest : " + url);
    Uri uri = Uri.parse(url);
```

```
    String localPath = "file://" +
Environment.getExternalStorageDirectory().getAbsoluteFile() + "/www" +
uri.getPath();
    File file = new File(localPath);
    try {
        URL localUri = new URL(localPath);
        if (localUri != null) {
            InputStream is = localUri.openConnection().getInputStream();
            WebResourceResponse resourceResponse = new WebResourceResponse(
 MimeTypeMap.getSingleton().getMimeTypeFromExtension(MimeTypeMap.getFileExtensio
nFromUrl(url)), "UTF-8", is);
            Log.d(TAG, "replace " + MimeTypeMap.getFileExtensionFromUrl(url));
            return resourceResponse;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return super.shouldInterceptRequest(view, url);
}
```

WebView中调用的每个请求都会经过那个拦截器，所以如果一个页面中又有超链接，那么依然会经过那个拦截器，所以上面Import new中有些图片没有加载出来。因为我是以本文形式获取响应的,并且是以utf-8作为编码的，所以有时会出现乱码。

9.29 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

每个Activity都有一个属于自己的window，也就是一个PhoneWindow，所以activity的窗口相关的控制都是由这个window来统一的处理的，因此，对activity窗口的亮度变化需要看window中是否有对应的属性提供给开发者。而恰好window属性中恰好就有这个变量：alpha属性，所有，我们可以通过alpha属性来控制Activity窗口亮度。具体的代码大家可以参考下面的代码。

```
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);
```

9.30 RecyclerView与ListView的对比，缓存策略，优缺点。

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，用来实现列表的功能。接下来我们从布局、局部刷新、动画效果、缓存四个方面对比RecyclerView和ListView。

1.布局效果

ListView 的布局方式比较单一，只有一个纵向效果；

RecyclerView 的布局会多样一些， 线性布局（纵向布局，横向布局），表格布局，瀑布流布局，如果不能满足需求可以自定义LayoutManager布局管理器来实现各种需求。

2.局部刷新

在ListView只有全局刷新，即使只想刷新一个列表项，也要所有的列表项都更新，这是比较消耗性能的；
RecyclerView中实现局部刷新十分的方便，例如调用notifyItemChanged()函数，这个得益于RecyclerView的创建列表项onCreateViewHolder和绑定数据onBindViewHolder的逻辑分离，同时也得益于缓存设计非常便利，具体关于缓存可以先阅读RecyclerView的回收复用原理一题的解答。

3.动画效果

ListView并没有实现动画效果，可以在Adapter自己实现item的动画效果；

在RecyclerView中，实现自己的动画效果有现成的API，如果我们想自定义动画效果，可以通过相应的接口实现（RecyclerView.ItemAnimator类），然后调用RecyclerView.setItemAnimator方法 (默认的有SimpleItemAnimator与DefaultItemAnimator）来设置动画；

4.缓存区别

层级不同

ListView有两级缓存，在屏幕中与屏幕外：mActivityViews 、 mScrapViews
RecyclerView有四级缓存：有屏幕内的缓存Scrap，包括具体的mAttachedScrap、mChangedScrap，他们是在notify来修改数据的时候应用；有mCacheViews缓存，它用于保存离开屏幕的列表项Item；有自定义的缓存mViewCacheExtension，是缓存拓展的帮助类，额外提供了一层缓存给开发者，开发者可以自己具有缓存的需要进行设置；有mRecyclerPool，它是终极的缓存池，上述缓存都没有找到的情况下才会读取mRecyclerPool缓存；

缓存内容不同

ListView是对Item的最外层的View进行缓存。
RecyclerView是对ViewHolder进行缓存，ViewHolder是对Item的View的封装，ViewHolder在构造方法里面会通过findViewById将各个需要的View的引用设置好，是为了之后复用的时候不需要重新findViewById，提升效率。

缓存机制

ListView和RecyclerView的缓存机制大致类似：

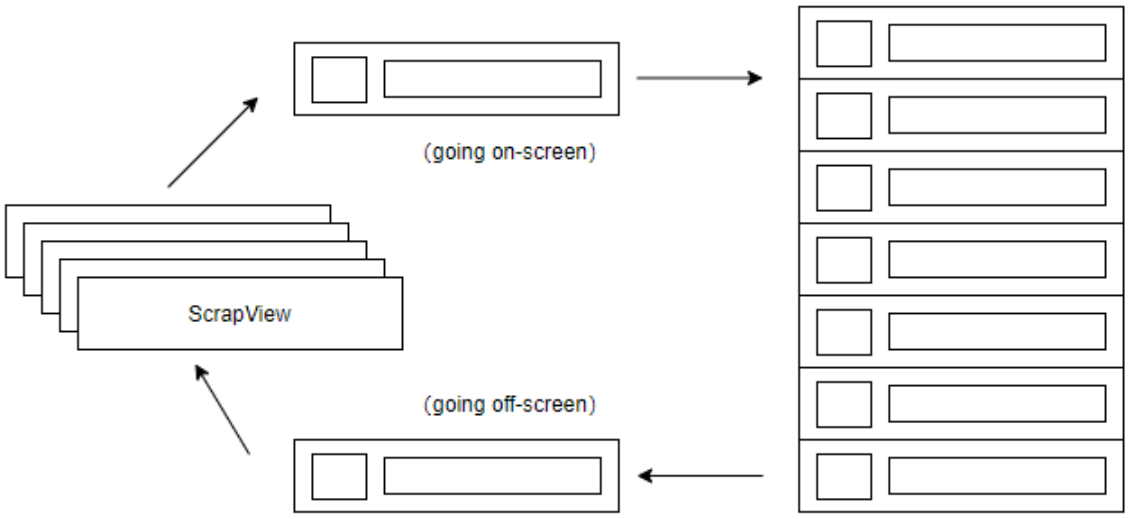


图7.15.1

如图7.15.1所示，在整个滑动的过程中，离开屏幕的Item立即被回收至缓存ScrapView，滑入屏幕的Item则会优先从缓存ScrapView中获取，只是ListView与RecyclerView的实现细节有差异。

ListView与RecyclerView缓存级别的对比

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快速复用
mScrapViews	否	是	与mAdapter一致，当mAdatper被更换时，mScrapViews即被清除	

ListView获取缓存的流程：

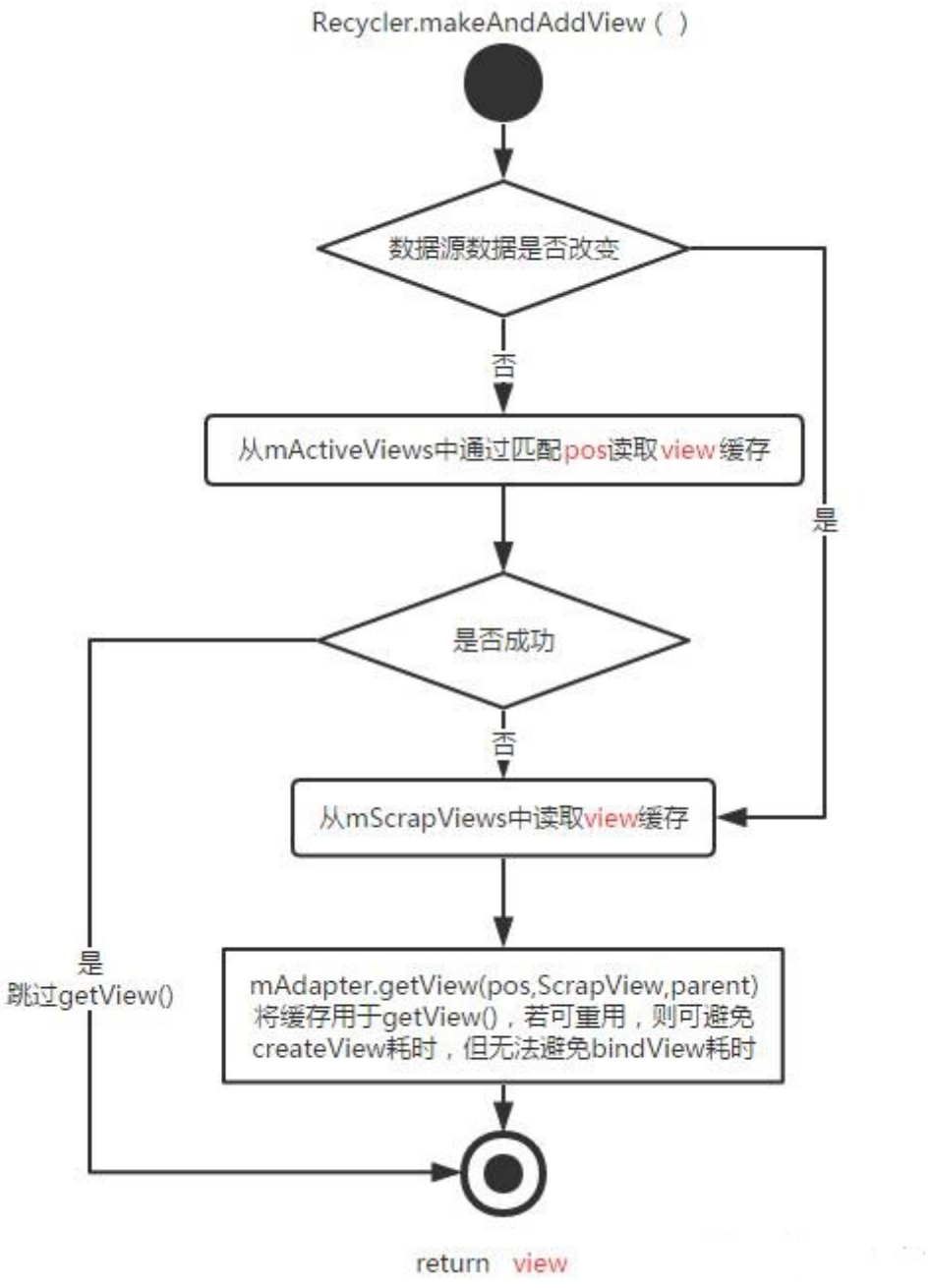


图7.15.2

如图7.15.2所示，ListView会先通过position读取mActiveViews中的缓存，从这一级读取的缓存是可以快速复用，不需要重新绑定数据。如果mActiveViews中没有找到则会从下一级mScrapViews中读取，这一级读取的缓存是需要重新绑定数据。

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快速复用
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

RecyclerView获取缓存的流程，请阅读RecyclerView的回收复用原理一题，其中有关缓存流程的相关内容。

ListView和RecyclerView缓存机制基本一致：

1). listView 的mActiveViews和RecyclerView 的mAttachedScrap功能类似，设计的目的是快速复用屏幕上可见的列表项ItemView，而不需要重新createView和bindView；

2). listView的 mScrapView和RecyclerView 的mCachedViews + mReyclerViewPool功能类似，设计的目的是让即将进入屏幕的Item可以从缓存容器中读取缓存数据，减少耗时。

3). RecyclerView的优点在于两点内容：第一点mCacheViews的使用，可以实现屏幕外的列表项ItemView进入屏幕内时也不需要bindView，可以快速复用；第二点mRecyclerPool可以供多个RecyclerView共同使用；

客观来说，RecyclerView在特定场景下对ListView的缓存机制做了强加和完善。

9.31 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。
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。

另一个方面，将viewHolder 定义为静态内部类，那么这个viewHolder 可以不仅仅在当前类可以用，在其他类里面也可以直接复用这个ViewHolder。

9.32 ListView卡顿的原因以及优化策略

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

导致ListView卡顿的原因有很多，主要包括：Item没有复用、层级过深、数据绑定逻辑过多、滑动时不必要的图片刷新以及频繁的notifyDataSetChanged。

Item没有复用

ListView的Item没有复用是导致卡顿的常见原因。在滑动的过程中，有些Item离开屏幕，有些Item需要进入屏幕。离开屏幕的Item一般会加入到缓存容器中，而不是让item直接被GC的回收。如果有缓存，那么滑动进入屏幕的Item会优先从缓存容器中读取。读取到的缓存会通过给convertView赋值来更新UI。如果没有课复用的item，那么之后每次都会重新创建这些Item，也就是通过LayoutInflater进行Item的创建，LayoutInflater创建Item是采用反射去解析xml因此是比较耗费时间的，这就会带来性能损耗。

布局的层级过深

布局的层级过深是非常容易引起卡顿，原因是ViewGroup会对子View进行多次测量。假设有一个这样的场景：父布局的布局属性是wrap_content、子布局是match_parent，此时的布局过程是：

1. 父布局先以0为强制宽度测量子View、然后继续测量剩下的其他子View
2. 再用其他子View里最宽的宽度，二次测量这个match_parent的子 View，最终得出它的尺寸，并把这个宽度作为自己最终的宽度。

即 这个过程就对单个子View进行了二次测量。

「而布局嵌套对性能影响则是指数形式的」，即：父布局会对每个子view做两次测量，子view也会对下面的子view进行两次测量，即相当于是 $O(2^n)$测量。

总的来说，层级过深会导致多次测量，不必要的测量造成了多余的性能消耗，最终有可能会引起页面的卡顿。

数据绑定逻辑过多

数据绑定逻辑过多也容易导致卡顿。在适配器中，我们通过onCreateViewHolder创建Item，创建之后的Item要通过onBindViewHolder进行数据绑定。过程大致如下：先通过position找到数据源中对应的数据，然后再将数据设置到控件的属性中。简单的设置以及一些简单的计算是不会消耗很多的时间的，如果是包含大量的控件属性设置、数据遍历、转化的话时间就会成倍的增长，最终可能就会导致ListView的卡顿。

滑动时不必要的图片刷新

图片的刷新也有可能导致卡顿。如果Item中包含图片并且我们以非常快的速度滑动列表，那么代表着有Item快速离开屏幕，也有Item快速的进入屏幕。快速进入屏幕的Item必须以极短的时间完成数据绑定和图片加载刷新，但图片的加载是需要耗费较大的性能和时间的，频繁的图片加载会延缓Item的数据绑定过程，容易造成卡顿。

频繁的notifyDataSetChanged

频繁的notifyDataSetChanged也是导致ListView卡顿的一大原因。notifyDataSetChanged是刷新ListView的所有Item的界面上展示的数据，所有的Item都刷新数据肯定是十分消耗性能的，如果是频繁的notifyDataSetChanged，那带来的性能消耗肯定更加严重，最终非常有可能导致到卡顿。

解决办法

Item没有复用，这个问题的解决就是加上复用代码，让item的界面和展示的数据可以得以复用，减少之后创建新的item和绑定数据的次数。

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
        final ViewHolder holder;
        ListViewItem itemData = items.get(position);
        if(convertView == null){
            convertView = View.inflate(context, R.layout.list_item_layout,
null);
            holder = new ViewHolder();
            holder.userImg = (ImageView)
convertView.findViewById(R.id.user_header_img);
            holder.userName = (TextView)
convertView.findViewById(R.id.user_name);
            holder.userComment = (TextView)
convertView.findViewById(R.id.user_coomment);
            convertView.setTag(holder);
        }else{
            holder = (ViewHolder) convertView.getTag();
        }
```

```
        holder.userImg.setImageResource(itemData.getUserImg());
        holder.userName.setText(itemData.getUserName());
        holder.userComment.setText(itemData.getUserComment());
        return convertView;
}

static class ViewHolder{
        ImageView userImg;
        TextView userName;
        TextView userComment;
}
```

如上面代码所示，可以先对convertView进行判断，如果为空再通过LayoutInflater创建新的Item，如果不为空则可以复用。然后通过ViewHolder进一步优化，不用ViewHolder的情况下每次设置Item的值都会需要通过convertView.findViewById寻找相关的控件，不方便且耗时，如果在创建Item时将各类控件引用设置到ViewHolder中，以后就可以直接从ViewHolder找控件，是十分节约性能的。

布局的层级过深的解决方式是减少层级，尽量使用约束布局ConstraintLayout。约束布局ConstraintLayout是Google 参考了ios的约束布局创建的，他的功能相当于 RelativeLayout + LinearLayout，而性能相对于他们提升了40%左右，所以通过使用ConstraintLayout可以达到优化性能的目的。

数据绑定逻辑过多的话，首先是区分出与界面有关的逻辑计算和与界面无关的逻辑计算，与界面有关的逻辑计算只能通过优化算法逻辑、数据结构处理，与界面无关的逻辑计算不影响界面的显示可以开线程用于计算。

滑动时不必要的图片刷新这个解决办法就十分明确，如果快速滑动带来了很多不必要的刷新，那么可以在快速滑动的时候设置为不刷新图片，这样可以减少非常多不必要的性能消耗。

频繁的notifyDataSetChanged是容易导致卡顿的，这种情况可以优化代码逻辑，减少频繁的notifyDataSetChanged。

9.33 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

问题分析

RecyclerView是一个大的概念，从界面层次分析一下它的组成部分，它是由多个Item。Item的创建、赋值、回收、复用与RecyclerView息息相关，所以列表项是研究RecyclerView研究的重点切入口。

列表项Item有两个非常重要的内容，一个是Item的界面生成，这个和适配器中的onCreateViewHolder相关；另外一个是列表项Item的数据绑定，这个和适配器中的onBindViewHolder相关。onCreateViewHolder是创建ViewHolder，在这个过程中会通过LayoutInflater去生成界面，而LayoutInflater是通过反射的方式实例化View的，基于这个情况可知创建ViewHolder是比较耗时的。onBindViewHolder是绑定相关的视图数据，这个过程中如果设置内容过多或者计算过多都会比较耗时。

当RecyclerView的设计者分析出onCreateViewHolder、onBindViewHolder这两个函数是比较耗时的情况下，很自然会采取策略就是减少onCreateViewHolder和onBindViewHolder调用，达到性能最优的效果。

情况分析

如果当前需要创建一个列表项Item，可以选择onCreateViewHolder创建，也可以选择先去读取缓存。根据缓存的界面、数据的不同，会存在两种情况：第一种情况，缓存中有与需要创建列表项item同类型（适配器中的ViewType）的缓存，那么就不需要重新创建，也就是减少了onCreateViewHolder的调用。第二种情况，如果存在一种缓存，它与需要创建的列表项Item类型相同，并且数据相同，那么就不需要重新绑定数据，更不需要重新创建，也就是同时减少onCreateViewHolder、onBindViewHolder的调用。

缓存类型

据上面的情况分析可以得出缓存的两种类型：第一类是与需要创建Item的类型相同但数据不相同，这种类型的缓存是RecycledViewPool，它给到每个Type（Type指的是适配器Adapter中的ItemType）的缓存容量默认是5。第二类是与需要创建列表项Item的类型相同并且数据相同，这种类型的缓存包括Scrap、CacheViews；

使用场景

当RecyclerView的数据或者界面发生变化的时候，就会用到缓存，大体分为两类情况，一类是进行滑动列表的时候，另外一类就是主动的数据更新。滑动列表的啥时候，一部分Item需要离开屏幕，另外一部分Item需要进入屏幕，进入屏幕这部分Item就非常有可能是从缓存中读取的而来。 主动更新数据，比如通过notifyDataSetChanged更新所有Item数据、通过notifyItemChanged进行局部Item更新，会将数据先放置到缓存容器中，然后在进行相关的更新操作。

滑动

Item回收

在滑动的过程中是有item离开屏幕，有item进入屏幕，这个过程了就会涉及回收数据和读取缓存，两部分的内容，本文也是从这两个方面给大家讲解。

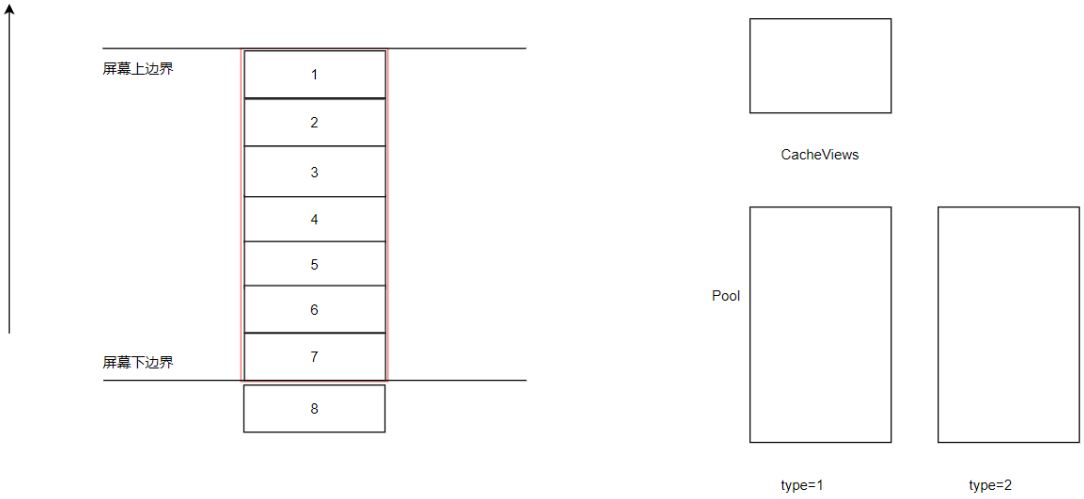


图7.32.1

上图7.32.1描绘的是RecyclerView的一个上滑的过程，左边用来表示一个RecyclerView的界面部分，上下两根横线分别代表着屏幕的上边界和下边界方便大家观察效果。右边是缓存包括CacheViews和RecycledViewPool两种。

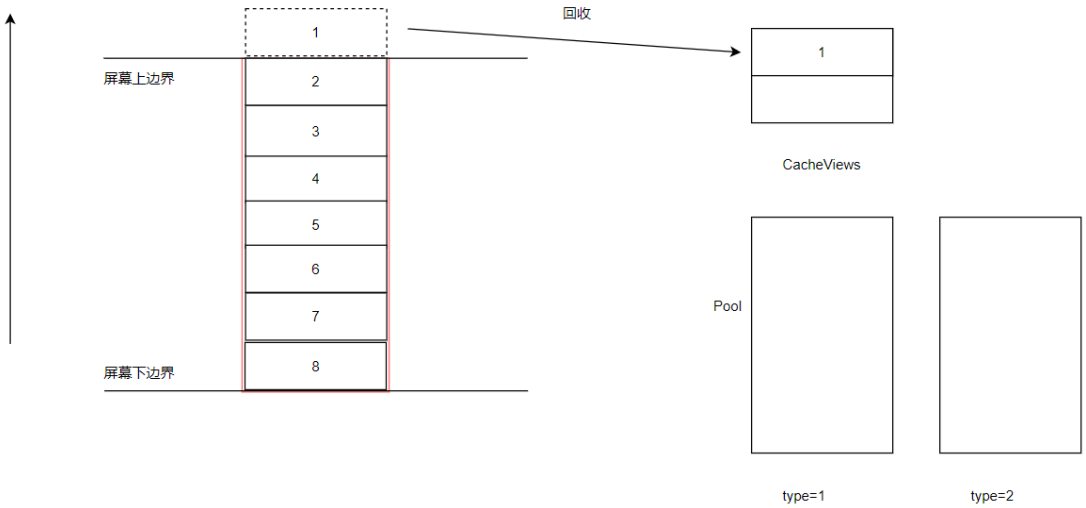


图7.32.2

如图7.32.2所示，当列表不断上滑的过程中了，Item1会离开屏幕，Item如果离开屏幕就会加入到缓存，首先加入的就是图7.32.2左边的CacheViews缓存。

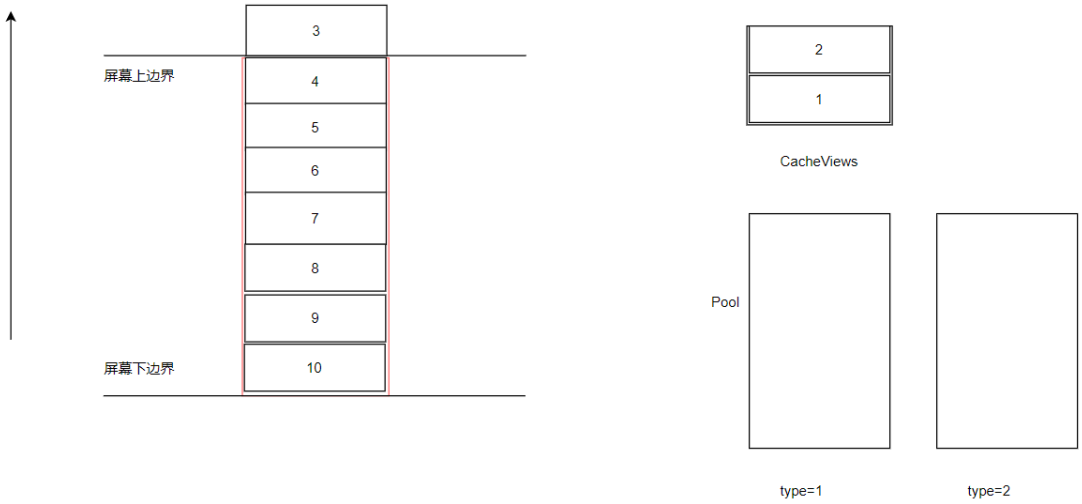


图7.32.3

如图7.32.3所示，当列表项Item3离开屏幕时，CacheViews保存着Item1和Item2。这个时候就会出现一个问题，CacheViews已经装满了，Item3又需要加入这个容器中来，如何处理？其实很简单，新的加入旧的出局，那么CacheViews里面装的就是Item3和Item2。Item1何处何从？它会安排进去RecycledViewPool里面，不过这里会出现一个变化，那就是Item的数据失效了。如下图7.32.4所示，所有加入RecycledViewPool的缓存如果被读取了，都会需要重新绑定数据即会调用onBindViewHolder。

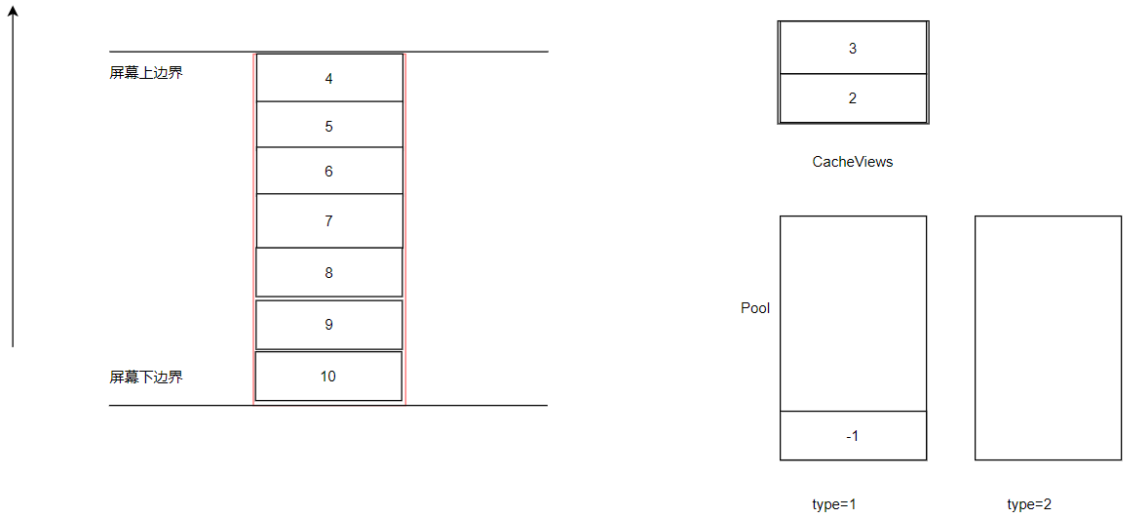


图7.32.4

Item缓存读取

在不断向上滑动的过程中，有新的Item进入屏幕，那新的列表项从何而来？有可能是通过onCreateViewHolder新创建的，也有可能是从缓存中读取而来。

有新的Item需要进入屏幕的时候，会先读取的是CacheViews的缓存，那是任意一个缓存的Item都可以？肯定不是，必须要符合位置position相同或ID相同的条件。整个过程会先匹配position（对应适配器里面的position），如果有同position缓存，说明正式需要寻找的数据。如果通过position没有寻找到数据，那就会通过id（对应适配器中getViewId中返回的id，默认是0）再寻找一遍。如果找到相同id的缓存则读取结束，没有读取到的话，就会在RecycledViewPool中选中一个同类型（对应适配器中getViewType返回的Type）的缓存，RecyclerViewPool找到的缓存是需要重新绑定数据的。

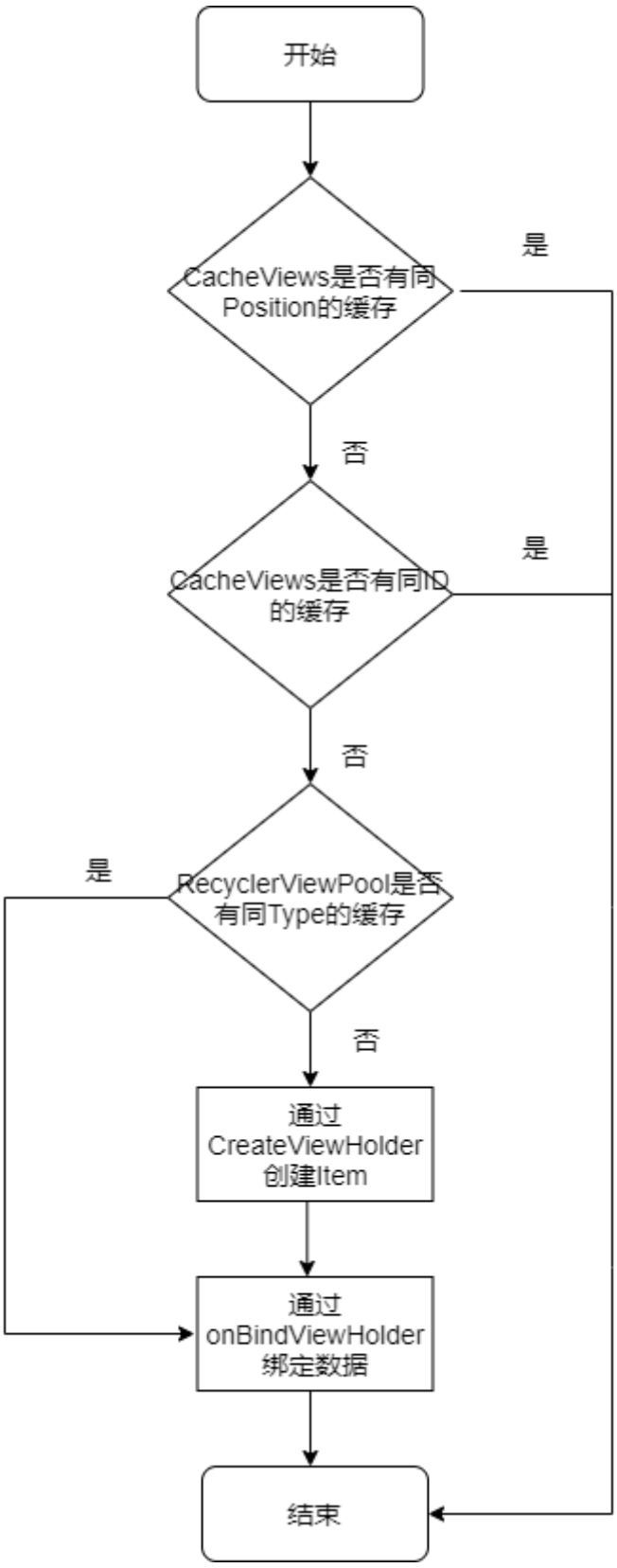


图7.32.5

如果RecyclerViewPool中也没有找到缓存，则只能通过CreateViewHolder创建item，然后通过onBindViewHolder绑定数据。

Notify更新数据

NotifyDataSetChanged

如果调用notifyDataSetChanged这个api，这个函数调用的结果是更新界面上所有item的显示内容。这个时候由于复用机制的存在，所以表项Item的界面（View本身）可以复用，但是Item展示的数据是需要重新更新的。在上述过程中Item会先加入缓存，然后再从缓存中读取出来重新绑定数据，符合要求的缓存容器只有RecycledViewPool，因为它保存的Item是需要重新绑定数据的，符合当前的刷新要求。

RecycledViewPool对每个Type的缓存设置的大小是5。如果调用notifyDataSetChanged这个api，说明有五个Item会加入到RecycledViewPool这一级的缓存里面来，然后在被读取重新赋予新的数据用于展示，这部分的Item是可以有效地回收复用。如果屏幕中展示的列表项Item大于五，由于缓存容器大小的缘故，只能有五个Item可以加入到缓存容器RecycledViewPool，可以回收复用，其余的部分Item由于不能及时加入缓存容器RecycledViewPool，那么它的空间就不会被回收复用，导致空间的浪费，导致之后需要重新创建。

如果一定要通过 notifyDataSetChange 方法更新数据，可以通过下面这种方式，在更新前调大缓存空间大小，更新完成后再调小缓。这种方式可以最大程度地复用已有的 ViewHolder，达到性能最优的效果。

```
mRecyclerView.getRecycledViewPool().setMaxRecycledViews(0, 屏幕显示的item总数+1 );
mAdapter.notifyDataSetChanged();
new Handler().post(new Runnable() {
    @Override
    public void run() {
        mRecyclerView.getRecycledViewPool()
                .setMaxRecycledViews(0, 5);
    }
});
```

setMaxRecycledViews中填入的第一个参数是Type，第二个参数是数目，一般是当前屏幕显示Item的总数加一，加一的原因是因为RecyclerView除了加载当前屏幕的Item以外了，还会额外再加载一个Item。

之所以重新又将MaxRecycledViews的值设置回5，是因为缓存空间变大是满足当前 notifyDataSetChange 的需求，后面的一些业务并不是缓存空间越大越好，调回来也是为了减少对后续操作的影响。

NotifyItemChanged

如何只需要更新一个Item的话，可以调用notifyItemChanged，这样的话就只是局部更新，相对来说性能消耗会小很多。如果当前屏幕内有六个列表项Item1...Item7，我们点击其中的Item4，在这个过程中了，Item4的界面可以复用，数据需要重新绑定，Item1、Item2、Item3、 Item5、Item6、Item7界面和数据都可以复用不需要做任何的改变，那RecyclerView是如何处理这个操作的呢？不着急我们慢慢讲，Item1、Item2、Item3、 Item5、Item6、Item7会被加入到Scrap中的mAttachedScrap里面，需要更换数据的Item4会被加入到mChangedScrap缓存中，相信大家看完很清楚，不要变的放一起，需要变化下的又放另外一个容器。之后再通过比对position、id等方式读取出缓存的内容，进行相关的测量布局设置工作，那整个过程就讲清楚了。

总结

1. AttachedScrap、ChangedScrap只与Notify更新数据有关，与滑动过程的回收无关，二者的空间是没有限定大小的。在一次Notify更新的过程中不需要发生变化的列表项Item会存放在AttachedScrap内，需要更新的列表项Item会放置在ChangedScrap，所以AttachedScrap、ChangedScrap被称之为屏幕内的缓存。
2. CacheViews是用于保存最新被移除(`remove`)的列表项Item，一般指的是滑动过程中离开屏幕的列表项Item，会精准的通过位置Position、ID匹配当前缓存是否为需要的内容。因为是精准匹配，所以读取出来的缓存是不需要重新绑定数据的。
3. RecycledViewPool是一个终极回收站，真正存放着被标识废弃(其他池都不愿意回收)的ViewHolder的缓存池，如果上述AttachedScrap、ChangedScrap、CachedViews都找不到ViewHolder的情况下，就会从RecycledViewPool返回一个废弃的ViewHolder实例。

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

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来确定该表UI的显示。
刷新状态的判断是在MotionEvent.ACTION_UP手指抬起时判断的。这很容易理解，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，则意味着刷新，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，它的实现十分简单，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);
        }
    }
}
```

9.35 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。
但是这个方法有个问题，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);
}
```

这个方法里调用了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，话外之音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。

9.36 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错位的原因是一致的，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都是一致，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;
}
```

9.37 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。
另外一个十分耗时的操作是请求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，它会需要重新计算很多次。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。 如果你有一个十分复杂的界面，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， 但是从来都不测量他们。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，你应该思考写一个自定义的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设置为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，而正好我们的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);
    }
}
```