Android面试题之View，包括View的事件分发、三大流程、滑动、滑动冲突等内容。

> 本文是我一点点归纳总结的干货，但是难免有疏忽和遗漏，希望不吝赐教。
> 转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81136256

有帮助的话请点个赞！万分感谢！

Android面试题-View(148题)

版本：2018/9/2-3(14:34)

View基础(25题)

1、简述View的绘制流程

> 1. `onMeasure`-测量 ： 从顶层View到子View递归调用 `measure()` 方法， `measure()`内部调用`onMeasure()`，在 `onMeasure()` 中完成 测量工作
> 2. `onLayout`-布局 ： 从顶层View到子View递归调用 `layout()` 方法，layout调用 `onLayout()` ,会根据 测量返回的视图大小 和布局参数将 `View` 放置到合适位置。
> 3. `onDraw`-绘制：ViewRoot会创建Canvas，然后执行 `onDraw()` 进行绘制。

2、onDraw()的绘制顺序

> 1. 绘制背景
> 2. 绘制`View`内容
> 3. 绘制子`View`
> 4. 绘制滚动条

3、requestLayout()的作用

> 1. 请求重新测量、布局
> 2. `View` (requestLayout)-> `ViewGroup` (requestLayout)-> `DecorView` (requestLayout)-> `ViewRootImpl` (requestLayout)。
> 3. 最终会触发 `ViewRootImpl` 的 `performTraversals()`，会触发 `onMeasure()` 和 `onLayout()` ， 不一定会触发 `onDraw()`

4、requestLayout在什么情况下只会触发 测量和布局 ， 而不会触发 绘制 ？

> 如果没有改变控件的left\right\top\bottom就不会触发 `onDraw()`

5、invalidate()的作用

> 1. 请求重新绘制
> 2. 会递归调用 父`View`的`invalidateChildInParent` -> `ViewRootImpl` 的 `invalidateChildInparent()`
> 3. 最终会执行 `ViewRootImpl` 的 `performTraversals()`，不会会触发 `onMeasure()` 和 `onLayout()` ， 会触发 `onDraw()` 也可能不触发 `onDraw()`

6、invalidate()在什么情况下不会触发onDraw?

> 1. 在 `ViewGroup` 中， `invalidate` 默认不重新绘制子view。

7、如何让ViewGroup在invalidate时会触发onDraw?

> 本质需要将`ViewGroup`的`dirtyOpaque`设置为`false`
>
> 1. 在构造函数中调用 `setWillNotDraw(false);`
> 2. 给ViewGroup设置背景。调用 `setBackground`。

8、postInvalidate()的作用

> 1. 与 `invalidate()` 的作用一致。
> 2. 区别在于： 用于在非`UI`线程中请求重新绘制

什么是View

9、什么是View

> 1. View是所有控件的基类

2. View有一个特殊子类ViewGroup，ViewGroup能包含一组View，但ViewGroup的本身也是View。
3. 由于View和ViewGourp的存在，意味着View可以是单个控件也可以是一组控件。这种结构形成了View树。

10、Android坐标系

1. Android坐标系以屏幕左上角为原点，向右X轴为正半轴，向下Y轴为正半轴
2. 触摸事件中getRawX()和getRawY()获得的就是Android坐标系的坐标
3. 通过 `getLocationOnScreen(intlocation[])` 能获得当前视图的左上角在Andriod坐标系中的坐标。

11、视图坐标系(View坐标系)

1. View坐标系是以当前视图的父视图的左上角作为原点建立的坐标系，方向和Android坐标系一致
2. 触摸事件中getX()和getY()获得的就是视图坐标系中的坐标

View的位置参数

12、View的位置参数：top,left,right,bottom

1. top-左上角的y轴坐标(全部是相对坐标，相对于父容器)
2. left-左上角的x轴坐标
3. right-右下角的x轴坐标
4. bottom-右下角的y轴坐标
5. 在View中获取这些成员变量的方法，是getLeft(),getRight(),getTop(),getBottom()即可

13、View从3.0开始新增的参数：x,y,translationX,translationY

1. x,y是View当前左上角的坐标
2. translationX,translationY是在滑动/动画后，View当前位置和View最原始位置的距离。
3. 因此得出等式：x(View左上角当前位置) = left(View左上角初始位置) + translationX(View左上角偏移的距离)

14、View平移时是否改变了left、top等原始参数?

1. View平移时top、left等参数不变，改变的是x,y,tranlsationX和tranlsationY

MotionEvent

15、MotionEvent是什么？有什么用?

1. `MotionEvent` 是 手指触摸事件 。
   16、MotionEvent包含的手指触摸事件
2. ACTION_DOWN\MOVE\UP对应三个触摸事件。
3. getX/getY能获得触摸点的坐标，相当于当前View左上角的(x,y)
4. getRawX/getRawY，获得触摸点相当于手机左上角的(x,y)坐标

ViewRoot

16、ViewRoot是什么?

1. ViewRoot对应于 `ViewRootImpl` 类
2. 是连接 `WindowManager` 和 `DecorView` 的 纽带
3. 发起并完成 `View`的三大流程 (测量、布局、绘制)
4. `ViewRoot` 需要和 `DecorView` 建立联系。

DecorView

17、DecorView的作用

1. DecorView是顶级View，本质就是一个FrameLayout
2. 包含了两个部分，标题栏和内容栏
3. 内容栏id是content，也就是activity中setContentView所设置的部分，最终将布局添加到id为content的FrameLayout中

18、DecorView中如何获取ContentView以及Activity所设置的View?

1. 获取content： `ViewGroup content = findViewById(R.android.id.content)`
2. 获取设置的View： `content.getChidlAt(0)`

19、ViewRootIml如何和DecorView建立联系?

1. Activity对象在ActivityThread中创建完毕后，会将DecorView添加到Window中

2. 同时会创建ViewRootImpl，调用ViewRoot的 `setView` 方法将 `ViewRootImpl` 和 `DevorView` 建立关联

```
root = new ViewRootImpl(view.getContext(), display);
root.setView(view, wparams, panelParentView);
```

20、 `ViewRoot` 为什么要和 `DecorView` 建立关联

1. `DecorView` 等View的 三大流程 需要通过 `ViewRoot` 完成

MeasureSpec

21、MeasureSpec是什么?

1. MeasureSpec是一种“测量规则”或者“测量说明书”，决定了View的测量过程
2. View的MeasureSpec会根据自身的LayoutParamse和父容器的MeasureSpec生成。
3. 最终根据View的MeasureSpec测量出View的宽/高(测量时数据并非最终宽高)

22、MeasureSpec的组成?

1. MeasureSpec代表一个32位int值，高2位是SpecMode，低30位是SpecSize
2. SpecMode是指测量模式
3. SpecSize是指在某种测量模式下的大小
4. 类MesaureSpec提供了用于SpecMode和SpecSize打包和解包的方法

23、测量模式SpecMode的类型和具体含义?

1. UNSPECIFIED：父容器不对View有任何限制，一般用于系统内部
2. EXACTLY：精准模式，View的最终大小就是SpecSize指定的值（对应于LayoutParams的match_parent和具体的数值）
3. AT_MOST：最大值模式，大小不能大于父容器指定的值SpecSize(对应于wrap_content)

24、MeasureSpec和LayoutParams的对应关系

1. View的MeasureSpec是需要通过 自身的`LayoutParams` 和 父容器的`MeasureSpec` 一起才能决定
2. DecorView(顶级View)是例外，其本身MeasureSpec由 窗口尺寸 和 自身`LayoutParams` 共同决定
3. MeasureSpec一旦确定，onMeasure中就可以确定View的测量宽/高

25、普通View的MeasureSpec的创建规则

1. View本身布局参数为具体dp/px数值，模式：EXACTLY，尺寸：自身尺寸(不管父容器的MeasureSpec)
2. View为match_parent， 模式：EXACTLY/AT_MOST由父容器MeasureSpec决定，尺寸：父容器目前可用大小
3. View为wrap_content，模式：AT_MOST,尺寸：父容器可用尺寸(不能超过该尺寸)
4. 当父容器为UNSPECIFIED时，View为具体数值时规则不变；其余match_parent/wrap_content，模式均为：UNSPECIFIED，尺寸：0
5. UNSPECIFIED一般用于系统内部多次measure的情况，不需要关注该模式。

子视图布局参数(LayoutParams) \ 父视图测量模式(mode)	EXACTLY	AT_MOST	UNSPECIFIED
具体数值（dp / px）	EXACTLY + childSize	EXACTLY + childSize	EXACTLY + childSize
match_parent	EXACTLY + parentSize (父容器的剩余空间)	AT_MOST + parentSize (大小不超过父容器的剩余空间)	UNSPECIFIED + 0
wrap_content	AT_MOST + parentSize (大小不超过父容器的剩余空间)	AT_MOST + parentSize (大小不超过父容器的剩余空间)	UNSPECIFIED + 0

View三大流程(28题)

1、ViewRoot如何完成View的三大流程?

1. ViewRoot的 `performTraversals()` 开始View的绘制流程，依次调用 `performMeasure()` 、 `performLayout()` 和 `performDraw()`
2. performMeasure()最终执行父容器的measure()方法，并依此执行所有子View的measure方法。
3. performLayout()和performDraw()同理

2、View三大流程的作用?(3)

> 1. measure决定了View的宽/高，测量后可以通过 `getMeasuredWidth/Height` 来获得View测量后的宽/高，除特殊情况外该值等于View最终的宽/高
> 2. layout决定了View的顶点坐标以及实际View的宽/高：完成后可以通过 `getTop/Bottom/Left/Right` 获取顶点坐标，并通过 `getWidth/Height()` 获得View的最终宽/高
> 3. draw决定了View的显示，最终将View显示出来

3、什么时候测量宽高不等于实际宽高?

> `MeasuredWidth/height ! = getWidth/Height()` 的场景：更改View的布局参数并进行重新布局后，就会导致 `测量宽高 ! = 实际宽高`

measure过程

View

4、View的measure方法的特点?

> 1. View的measure方法是final类型方法——表明该方法无法被重载
> 2. View的measure方法会调用onMeasure方法，onMeasure会调用setMeasuredDimension方法设置View宽/高的测量值

5、View的onMeasure源码要点

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        //1. setMeasuredDimension方法设置View宽/高的测量值
        setMeasuredDimension(
                //2. 第一个参数是获得的测量宽/高(通过getDefaultSize获取)
                getDefaultSize(getSuggestedMinimumWidth(),  //3. 获取的建议最小的宽/高
                               widthMeasureSpec),
                getDefaultSize(getSuggestedMinimumHeight(),
                               heightMeasureSpec));
}
```

> 1. setMeasuredDimension方法设置View宽/高的测量值（测量值通过getDefaultSize获取）
> 2. getDefaultSize用于获取View的测量宽/高

6、View的getDefaultSize源码要点(决定了View宽高的测量值)

```
    //1. 获取View宽和高的测量值
    public static int getDefaultSize(int size, int measureSpec) {
        int result = size;
        int specMode = MeasureSpec.getMode(measureSpec);
        int specSize = MeasureSpec.getSize(measureSpec);

        switch (specMode) {
        //2. UNSPECIFIED模式时，宽/高为第一个参数也就是getSuggestedMinimumWidth()获取的建议最小值
        case MeasureSpec.UNSPECIFIED:
            result = size;
            break;
        //3. AT_MOST(wrap_content)和EXACTLY(match_parent/具体值dp等)这两个模式下，View宽高的测量值为当前View的MeasureSpec(测量规格)中指定的尺寸spec
        case MeasureSpec.AT_MOST:
        case MeasureSpec.EXACTLY:
            result = specSize;
            break;
        }
        return result;
    }
```

7、View的getSuggestedMinimumWidth/Height()源码要点

```
//获取建议的最小宽度
protected int getSuggestedMinimumWidth() {
        return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.getMinimumWidth());
}
```

> 1. 如果View没有背景，View的最小宽度就为 `android:minWidth` 这个参数指定的值(mMinWidth),没有指定则默认为0
> 2. 如果View有背景，会从mMinWidth和背景的最小宽度中取最大值。
> 3. 背景的最小宽度(getMinimumWidth())本质就是Drawable的原始宽度(ShapeDrawable无原始宽度,BitmapDrawable有原始宽度——图片的尺寸)

8、View的onMeasure中调用的方法以及作用?

> 1. `setMeasuredDimension`：设置测量宽高
> 2. `getDefaultSize`： 根据 建议获取的最小宽高 和 测量规格 ， 决定实际的 测量宽高

3. `getSuggestedMinimumWidth`：没有背景就使用 `android: minWidth` ， 有背景就在 `View`最小宽度 和 `Drawable`的原始宽度 中取最大值。

9、getDefaultSize方法的处理逻辑?

1. `UNSPECIFIED`模式： 测量宽高 = 建议的最小宽高
2. `EXACTLY / AT_MOST`模式： 测量宽高 = specSize

10、View的wrap_content和match_parent效果一致的原因分析

1. 根据View的onMeasure方法中的getDefaultSize方法，我们可以发现在两种模式下，View的测量值等于该View的测量规格MeasureSpec中的尺寸。
2. View的MeasureSpec本质是由自身的LayoutParams和父容器的MeasureSpec决定的。
3. 当View为wrap_content时，该View的模式为AT_MOST，且尺寸specSize为父容器的剩余空间大小。
4. 当View为match_parent时，该View的模式跟随父容器的模式(AT_MOST/EXACTLY), 且尺寸specSize为父容器的剩余空间大小。
5. 因此getDefaultSize中无论View是哪种模式，最终测量宽/高均等于尺寸specSize，因此两种属性效果是完全一样的(View的大小充满了父容器的剩余空间)
6. 除非给定View固定的宽/高，View的specSize才会等于该固定值。

11、自定义View需要重写onMeasure方法，并写明两种模式的处理方法

```
//1. 重写onMeasure，特殊处理wrap_content的情况
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);
    int widthSpecSize =  MeasureSpec.getSize(widthMeasureSpec);
    int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);
    int heightSpecSize =  MeasureSpec.getSize(heightMeasureSpec);

    if(widthSpecMode == MeasureSpec.AT_MOST && heightSpecMode == MeasureSpec.AT_MOST){
        //2. 均为wrap_content时，将值设置为android:minWidth/Height属性指定的值
        setMeasuredDimension(mWidth, mHeight);
    }else if(widthSpecMode == MeasureSpec.AT_MOST){
        //3. 哪个为wrap_content哪个就用android:minXXX属性给定的最小值
        setMeasuredDimension(mWidth, heightSpecSize);
    }else if(heightSpecMode == MeasureSpec.AT_MOST){
        setMeasuredDimension(widthSpecSize, mHeight);
    }
}
```

ViewGroup

12、ViewGroup(抽象类)的measure流程

1. ViewGroup没有onMeasure方法，只定义了measureChildren方法(onMeasure根据不同布局难以统一)
2. measureChildren中遍历所有子元素并调用measureChild方法
3. measureChild方法中会获取子View的MeasureSpec(`getChildMeasureSpec`)，然后调用子元素View的measure方法进行测量

13、getChildMeasureSpec获取子元素MeasureSpec的要点

1. 子View的MeasureSpec是根据自身的LayoutParams和父容器SpecMode生成
2. 当子View的布局参数为wrap_content，且父容器模式为AT_MOST时，效果与子元素布局为match_parent是一样的。因此当子View的布局参数为wrap_content时，需要给指定默认的宽/高

14、LinearLayout的onMeasure()分析

1. ViewGroup因为布局的不同，无法统一onMeasure方法，具体内容根据布局的不同而不同，这里直接以LinearLayout进行分析
2. onMeasure会根据 `orientation` 选择measureVertical或者measureHorizontal进行测量
3. measureVertical本质是遍历子元素，并执行子元素的measure方法，并获得子元素的总高度以及子元素在竖直方向上的margin等。
4. 最终LinearLayout会测量自己的大小，在orientation的方向上，如果布局是match_parent或者具体数值，测量过程与View一致(高度为specSize)；如果布局是wrap_content，高度是所有子元素高度总和，且不会超过父容器的剩余空间，最终高度需要考虑在竖直方向上的padding

layout过程

15、View的layout过程

1. 使用 `layout` 方法确定View本身的位置
2. `layout` 中调用 `onLayout` 方法确定所有子View的位置

16、View的layout()源码分析

1. 调用setFrame()设置View四个定点位置(即初始化mLeft,mRight,mTop,mBottom的值)
2. 之后调用onLayout确定子View位置，该方法类似于onMeasure，View和ViewGroup中均没有实现，具体实现与具体布局有关。

17、LinearLayout的onLayout方法

> 1. 根据orientation选择调用layoutVertical或者layoutHorizontal
> 2. layoutVertical中会遍历所有子元素并调用setChildFrame(里面直接调用子元素的layout方法)
> 3. 层层传递下去完成了整个View树的layout过程
> 4. setChildFrame中的宽/高实际就是子元素的测量宽/高(getMeasure...后直接传入)

18、View的测量宽高和最终宽高有什么区别?

> 1. 等价于getMeasuredWidth和getWidth有什么区别
> 2. getWidth = mRight - mLeft，结合源码测量值和最终值是完全相等的。
> 3. 区别在于：测量宽高形成于measure过程，最终宽高形成于layout过程(赋值时机不同)
> 4. 也有可能导致两者不一致：强行重写View的layout方法，在传参方面改变最终宽/高（虽然这样毫无实际意义）
> 5. 某些情况下，View需要多次measure才能确定自己的测量宽高，在前几次测量中等到的值可能有最终宽高不一致。但是最终结果上，测量宽高=最终宽高

draw过程

19、draw的步骤

> 1. 绘制背景(drawBackground(canvas))
> 2. 绘制自己(onDraw)
> 3. 绘制children(dispatchDraw)-遍历调用所有子View的draw方法
> 4. 绘制装饰(如onDrawScollBars)

20、View特殊方法setWillNotDraw

> 1. 若一个View不绘制任何内容，需要将该标志置为true，系统会进行相应优化
> 2. 默认View不开启该标志位
> 3. 默认ViewGroup开启该标志位
> 4. 如果我们自定义控件继承自ViewGroup并且本身不进行绘制时，就可以开启该标志位
> 5. 当该ViewGroup明确通过onDraw绘制内容时，就需要显式关闭WILL_NOT_DRAW标志位。

获取View的宽高

21、如何获取View的测量宽/高

> 1. 在measure完成后，可以通过getMeasuredWidth/Height()方法，就能获得View的测量宽高
> 2. 在一定极端情况下，系统需要多次measure，因此得到的值可能不准确，最好的办法是在onLayout方法中获得测量宽/高或者最终宽/高

22、如何在Activity启动时获得View的宽/高

> 1. Activity的生命周期与View的measure不是同步运行，因此在onCreate/onStart/onResume均无法正确得到
> 2. 若在View没有测量好时，去获得宽高，会导致最终结果为0
> 3. 有四种办法去正确获得宽高

23、Activity中获得View宽高的4种办法?

> 1. `onWindowFocusChanged`
> 2. `view.post(runnable)`
> 3. `ViewTreeObserver`
> 4. `view.measure`

24、onWindowFocusChanged获得View的宽/高

```
//1. View已经初始化完毕，可以获得宽高
    @Override
    public void onWindowFocusChanged(boolean hasFocus) {
        super.onWindowFocusChanged(hasFocus);
//2. Activity得到焦点和失去焦点均会调用一次(频繁onResume和onPause会导致频繁调用)
        if(hasFocus){
            int width = view.getMeasuredWidth();
            int height = view.getMeasuredHeight();
        }
    }
```

25、view.post(runnable)获得View的宽/高

```
//1. 通过post将一个runnable投递到消息队列尾部
        view.post(new Runnable() {
            @Override
//2. 等到Looper调用次runnable时，View已经完成初始化
            public void run() {
                int width = view.getMeasuredWidth();
                int height = view.getMeasuredHeight();
            }
        });
```

26、ViewTreeObserver获得View的宽/高（Kotlin版）

```
    val observer = imageView.viewTreeObserver
        //1. 使用ViewTreeObserver的接口，可以在View树状态改变或者View树内部View的可见性改变时，onGlobalLayout会被回调
    observer.addOnGlobalLayoutListener(object :ViewTreeObserver.OnGlobalLayoutListener {
            //2. 能正确获取View宽/高
                    override fun onGlobalLayout() {
                    //3. 随着View树状态改变，会多次调用。因此需要移除监听器
                imageView.viewTreeObserver.removeGlobalOnLayoutListener(this)
                val width = imageView.measuredWidth
                val height = imageView.measuredHeight
            }
        })
```

27、View.measure()获得View的宽/高(Kotlin)

> 1. match_parent的情况下是不可以的，因为需要知道parent的size，这里无法获取。
> 2. 具体数值

```
    //1. 具体数值时(dp/px),让View重新测量
    val widthMeasureSpec = View.MeasureSpec.makeMeasureSpec(100, View.MeasureSpec.EXACTLY)
    val heightMeasureSpec = View.MeasureSpec.makeMeasureSpec(100, View.MeasureSpec.EXACTLY)
    imageView.measure(widthMeasureSpec, heightMeasureSpec)
    //2. 完成后就可以获得宽/高
    val width = imageView.width
    val height = imageView.height
```

> 3. wrap_content

```
    //1. wrap_content,将specSize设置为30位二进制的最大值 (1 << 30) - 1,让View重新测量(在AT_MOST情况下是合理的)
    val widthMeasureSpec = View.MeasureSpec.makeMeasureSpec((1 shl 30) - 1, View.MeasureSpec.AT_MOST)
    val heightMeasureSpec = View.MeasureSpec.makeMeasureSpec((1 shl 30) - 1, View.MeasureSpec.AT_MOST)
    imageView.measure(widthMeasureSpec, heightMeasureSpec)
    //2. 完成后就可以获得宽/高
    val width = imageView.width
    val height = imageView.height
```

Activity启动到加载ViewRoot的流程

28、Activity启动到最终加载ViewRoot(执行三大流程)的流程

> 1. Activity调用startActivity方法，最终会调用ActivityThread的handleLaunchActivity方法
> 2. handleLaunchActivity会调用performLauchActivity方法(会调用Activity的onCreate，并完成DecorView的创建)和handleResumeActivity方法
> 3. handleResumeActivity方法会做四件事：performResumeActivity(调用activity的onResume方法)、getDecorView(获取DecorView)、getWindowManager(获取WindowManager)、WindowManager.addView(decor, 1)
> 4. WindowManager.addView(decor, 1)本质是调用WindowManagerGlobal的addView方法。其中主要做两件事：1、创建ViewRootImpl实例 2、root.setView(decor, ...)将DecorView作为参数添加到ViewRoot中，这样就将DecorView加载到了Window中
> 5. ViewRootImpl还有一个方法performTraveals方法，用于让ViewTree开始View的工作流程：其中会调用performMeasure/Layout/Draw()三个方法，分别对应于View的三大流程。

自定义View(26题)

四种实现方法

1、自定义View实现方法的分类?

分类	注意点1	注意点2	注意点3
1.继承View	重写onDraw()--- 绘制和支持padding	重写onMeasure()--- 解决wrap_content问题	

分类	注意点1	注意点2	注意点3
2.继承ViewGroup	重写onMesaure()---测量子元素，测量自身， 并且需要处理子View的margin和自身的padding	必须实现onLayout()---布局子元素， 并且处理子View的margin和自身的padding属性	实现自身的LayoutParams并且重写Lay --让子View的Margin属性生效
3. 继承特定的View(TextView等)	扩展较容易实现	不需要额外支持 `wrap_content` 和 `padding`	
4. 继承特定的ViewGroup(LinearLayout等)	方法2能实现的效果方法4都能实现	___	

2、自定义View的注意点？(5)

> 1. View需要支持wrap_content、padding
> 2. ViewGroup需要支持子View的margin和自身的padding
> 3. 尽量不要在View中使用Handler，View已经有post系列方法
> 4. View如果有线程或者动画，需要及时停止(onDetachedFromWindow会在View被remove时调用)——避免内存泄露
> 5. View如果有滑动嵌套情形，需要处理好滑动冲突

直接继承View

3、直接继承自View的实现步骤和方法：

> 1. 重写onDraw，在onDraw中处理 `padding`
> 2. 重写onMeasure，额外处理 `wrap_content` 的情况
> 3. 设定自定义属性attrs(属性相关xml文件，以及在onDraw中进行处理)

```
class CustomViewByView(context: Context, attrs: AttributeSet?, defStyleAttr: Int, defStyleRes: Int):
        View(context, attrs, defStyleAttr, defStyleRes){
    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int):this(context, attrs, defStyleAttr, 0)
    constructor(context: Context, attrs: AttributeSet):this(context, attrs, 0, 0)
    constructor(context: Context): this(context, null, 0, 0)

    var mColor = Color.RED

    init {
        //3. 自定义attrs中属性的获取
        val typedArray = context.obtainStyledAttributes(attrs, R.styleable.CustomViewByView)
        mColor = typedArray.getColor(R.styleable.CustomViewByView_circle_color, Color.RED)
        typedArray.recycle()
    }

    //1. 重写onDraw方法
    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        val paint = Paint(Paint.ANTI_ALIAS_FLAG)
        paint.color = mColor //属性attrs给定的颜色
        //2. 需要处理padding
        val width = width - paddingLeft - paddingRight
        val height = height - paddingTop - paddingBottom
        canvas.drawCircle(paddingLeft + width.toFloat() / 2, paddingTop + height.toFloat() / 2,
                Math.min(width, height).toFloat() / 2, paint)
    }

    //3. 特别处理wrap_content的情况，给定一个最小值
    override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec)
        val widthSpecMode = MeasureSpec.getMode(widthMeasureSpec)
        val widthSpecSize = MeasureSpec.getSize(widthMeasureSpec)
        val heightSpecMode = MeasureSpec.getMode(heightMeasureSpec)
        val heightSpecSize = MeasureSpec.getSize(heightMeasureSpec)
        when{
            // 为wrap_content的边均使用最小值mMinWidth/mMinHeight
            widthSpecMode == MeasureSpec.AT_MOST && heightSpecMode == MeasureSpec.AT_MOST -> {
                setMeasuredDimension(minimumWidth, minimumHeight)
            }
            widthSpecMode == MeasureSpec.AT_MOST -> {
                setMeasuredDimension(minimumWidth, heightSpecSize)
            }
            heightSpecMode == MeasureSpec.AT_MOST -> {
                setMeasuredDimension(widthSpecSize, minimumHeight)
            }
        }
    }
}
```

自定义属性

4、自定义属性实现的步骤和源码

> 1. 在values目录下定义一个属性文件 `attrs_circle_view` ， 文件名可任意
> 2. 在控件的布局中使用该属性（需要添加 `xmlns:app="http://schemas.android.com/apk/res-auto"` ）
> 3. 在自定义View中处理自定义的属性

```
<com.example.a6005001819.androiddeveloper.CustomViewByView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@color/colorPrimary"
    android:padding="30dp"
    android:minWidth="100dp"
    android:minHeight="100dp"
    app:circle_color="@color/colorAccent"/>
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="CustomViewByView">
        <attr name="circle_color" format="color"/>
    </declare-styleable>
</resources>
```

直接继承ViewGroup

5、自定义View：继承自ViewGroup

1. 需要重写onMeasure方法，进行测量(测量子元素，测量自身-需要处理子View的margin和自身的padding)
2. 必须实现onLayout方法，并且处理子View的margin和自身的padding属性
3. 要让子View的Margin属性生效，需要实现自身的LayoutParams并且重写LayoutParmas相关的3个方法

```
class CustomViewByViewGroup(context: Context, attrs: AttributeSet?, defStyleAttr: Int, defStyleRes: Int):
        ViewGroup(context, attrs, defStyleAttr, defStyleRes){

    constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int):this(context, attrs, defStyleAttr, 0)
    constructor(context: Context, attrs: AttributeSet):this(context, attrs, 0, 0)
    constructor(context: Context): this(context, null, 0, 0)

    /**
     * 1. 继承ViewGroup必须实现onLayout方法
     */
    override fun onLayout(changed: Boolean, left: Int, top: Int, right: Int, bottom: Int) {
        var childLeft = paddingLeft //需要处理padding
        for(i in 0 until childCount){
            val childView = getChildAt(i)
            if(childView.visibility != View.GONE){
                val childWidth = childView.measuredWidth

                //2. 额外处理margin属性
                val childLayoutParams = childView.layoutParams as MarginLayoutParams
                childLeft += childLayoutParams.leftMargin
                childView.layout(childLeft,
                        childLayoutParams.topMargin + paddingTop,
                        childLeft + childWidth,
                        childLayoutParams.topMargin  + paddingTop + childView.measuredHeight) //一定要根据margin处理好四个顶点坐标
                childLeft += childWidth + childLayoutParams.rightMargin
            }
        }
    }

    /**
     * 2. 定义ViewGroup的布局测量过程(也需要额外处理margin)
     */
    override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec)

        val widthSpecMode = MeasureSpec.getMode(widthMeasureSpec)
        val widthSpecSize = MeasureSpec.getSize(widthMeasureSpec)
        val heightSpecMode = MeasureSpec.getMode(heightMeasureSpec)
        val heightSpecSize = MeasureSpec.getSize(heightMeasureSpec)

        var measureWidth = 0
        var measureHeight = 0

        //2. 需要测量所有子View!
        measureChildren(widthMeasureSpec, heightMeasureSpec)

        //3. 本身宽高的模式均为wrap_content, 需要根据子View来获得
        if(widthSpecMode == MeasureSpec.AT_MOST && heightSpecMode == MeasureSpec.AT_MOST){
            for(i in 0 until childCount){
                val childView = getChildAt(i)
                measureWidth += childView.measuredWidth //测量出总宽度

                //6. 处理marigin
                val childLayoutParams = childView.layoutParams as MarginLayoutParams
                measureWidth += childLayoutParams.leftMargin + childLayoutParams.rightMargin

                val totalCurChildHeight = childView.measuredHeight + childLayoutParams.topMargin + childLayoutParams.bottomMargin
                if(totalCurChildHeight > measureHeight){
                    measureHeight = totalCurChildHeight //选取子View中高度最大的
                }
            }
            //7. 处理padding
            measureWidth += paddingLeft + paddingRight
            measureHeight += paddingTop + paddingBottom
            setMeasuredDimension(measureWidth, measureHeight)
        }
        //4. 仅有高度是wrap_content
        else if(heightSpecMode == MeasureSpec.AT_MOST){
            //获取所有子View最大的高度, 宽度直接用给定的尺寸
            for(i in 0 until childCount){
                val childView = getChildAt(i)

                // 处理高度(wrap_content)上marigin
                val childLayoutParams = childView.layoutParams as MarginLayoutParams

                val totalCurChildHeight = childView.measuredHeight + childLayoutParams.topMargin + childLayoutParams.bottomMargin
                if(totalCurChildHeight > measureHeight){
                    measureHeight = totalCurChildHeight //选取子View中高度最大的
                }
            }
            measureHeight += paddingTop + paddingBottom //处理高度的padding
```

```
            setMeasuredDimension(widthSpecSize, measureHeight)
        }
        //5. 仅有宽度是wrap_content
        else if(widthSpecMode == MeasureSpec.AT_MOST){
            for(i in 0 until childCount){
                val childView = getChildAt(i)
                measureWidth += childView.measuredWidth

                //  处理宽度(wrap_content)上marigin
                val childLayoutParams = childView.layoutParams as MarginLayoutParams
                measureWidth += childLayoutParams.leftMargin + childLayoutParams.rightMargin
            }
            measureWidth += paddingLeft + paddingRight            //  处理宽度的padding
            setMeasuredDimension(measureWidth, heightSpecSize)//高度直接用给定的尺寸
        }
    }

    /**
     * 3. 要让子View的Margin属性生效，必须要重写方法，并实现自己LayoutParams
     */
    override fun generateDefaultLayoutParams() = MyLayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT)
    override fun generateLayoutParams(attrs: AttributeSet) = MyLayoutParams(context, attrs)
    override fun generateLayoutParams(p: LayoutParams): MyLayoutParams{
        when(p){
            is LayoutParams -> return MyLayoutParams(p)
            is MarginLayoutParams ->  return MyLayoutParams(p)
            else -> return MyLayoutParams(p)
        }
    }

    open class MyLayoutParams : MarginLayoutParams {
        constructor(c: Context, attrs: AttributeSet) : super(c, attrs)
        constructor(width: Int, height: Int) : super(width, height)
        constructor(p: ViewGroup.LayoutParams) : super(p) {}
        constructor(source: ViewGroup.MarginLayoutParams) : super(source)
    }
}
```

6、自定义View的思想

> 面对陌生的自定义View的时候，需要掌握基本功：View的弹性滑动、滑动冲突、绘制原理。个人理解就是处理好三大流程：测量、布局和绘制。

性能优化

7、自定义View性能优化?(12种)

> 1. 避免过度绘制
> 2. 尽量减少或简化计算
> 3. 避免创建大量对象造成频繁GC
> 4. 禁止或避免I/O操作
> 5. onDraw中避免冗余代码、避免创建对象
> 6. 复合View，要减少布局层级。
> 7. 状态和恢复和保存
> 8. 开启硬件加速
> 9. 合理使用invalidate的参数版本。
> 10. 减少冗余代码：不要使用Handler，因为已经有post系列方法.
> 11. 使用的线程和动画，要在 `onDetachedFromWindow` 中进行清理工作。
> 12. 要妥善处理滑动冲突。

8、避免过度绘制

> 1. 像素点能画一次就不要多次绘制，以及绘制看不到的背景
> 2. 开发者选项里内的工具,只对xml布局有效果,看不到自定义View的过度绘制，仍然需要注意。

9、尽量减少或简化计算

> 1. 不要做无用计算。尽可能的复用计算结果。
> 2. 没有数据,或者数据较少的时候应如何处理,没有事件需要响应的时候如何处理。
> 3. 应该避免在for或while循环中做计算。比如：去计算屏幕宽度等信息。

10、避免创建大量对象造成频繁GC

> 1. 应该避免在for或while循环中new对象。这是减少内存占用量的有效方法。

11、禁止或避免I/O操作

> 1. I/O操作对性能损耗极大，不要在自定义View中做IO操作。

12、onDraw的优化

> 1. onDraw中禁止new对象.如：不应该在ondraw中创建Paint对象。Paint类提供了reset方法。可以在初始化View时创建对象。
> 2. 要避免冗余代码，提高效率。

13、invalidate()的高效使用

> 1. 避免任何请款下之际调用默认参数的 `invalidate`
> 2. 调用有参数的 `invalidate` 进行局部和子View刷新，能够提高性能。

14、减少布局层级

> 1. 复合控件：继承自现有的LinearLayout等ViewGroup，然后组合多个控件来实现效果。这种实现方法要注意减少布局层级，层级越高性能越差。

15、状态和恢复和保存

> Activity还会因为内存不足或者旋转屏幕而导致重建Activity，自定义View也要去进行自我状态的保存和读取。

```
// 1、保存状态
@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;
}

// 2、恢复状态
@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);
    }

}
```

16、尽量避免使用Handler

> 1. View已经有post系列方法，没有必要重复去写。
> 2. 可以直接使用，最终会投递到主线程的Handler中

```
post(new Runnable() {
    @Override
    public void run() {
         // 投递到UI线程中的Handler执行
        xxx
    }
});
```

17、线程和动画要及时终止

> 1. View如果有线程或者动画，需要及时停止.
> 2. View的onDetachedFromWindow会在View被remove时调用，在该方法内进行终止
> 3. 这样能避免内存泄露

18、View要妥善处理滑动冲突

> View如果有滑动嵌套情形，需要处理好滑动冲突

19、通过开启硬件减速提高自定View的性能

硬件加速

20、硬件加速的作用和优缺点

> 1. 硬件加速能能够使用GPU来加速2D图形的渲染操作
> 2. Android API11以后才开始支持硬件加速。

> 3. 缺点：不支持所有的渲染操作，可能会出现渲染错位的情况。需要谨慎取舍。

21、硬件加速四种级别的控制

> 1. Application
> 2. Activity
> 3. Windows
> 4. View

22、Application级别的硬件加速

> 针对整个APP

```
   android:hardwareAccelerated="true"
```

23、Activity级别的硬件加速

> 针对单个Activity

```
<activity android:name=".view.activity.LeadActivity"
                  android:hardwareAccelerated="false"
                  android:configChanges="keyboardHidden|orientation|screenSize"
                  android:theme="@style/MyTheme">

        </activity>
```

24、Window级别的硬件加速

> 注意：window级别的硬件加速只能打开，不能关闭。

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED,WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);
```

25、View级别的控制

```
// 硬件加速
myView.setLayerType(View.LAYER_TYPE_HARDWARE, null);
// 软件
myView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

26、如何判断是否开启了硬件加速

```
// 判断View是否开启硬件加速
 mView.isHardwareAccelerated();

// 判断图层是否开启硬件加速
 mCanvas.isHardwareAccelerated();
```

事件分发机制(22题)

1、事件分发

> 1. 点击事件的对象就是MotionEvent，因此事件的分发，就是MotionEvent的分发过程，
> 2. 点击事件有三个重要方法来完成：dispatchTouchEvent、onInterceptTouchEvent和onTouchEvent

三个重要方法

2、简述Android的事件分发机制

> 事件分发顺序： `Activty->ViewGroup->View`
> 主要方法： `dispatchTouchEvent`-分发事件 、 `onInterceptTouchEvent`-当前`View`是否拦截该事件 、 `onTouchEvent`-处理事件
>
> 1. 父View调用 `dispatchTouchEvent` 开启事件分发。
> 2. 父View调用 `onInterceptTouchEvent` 判断是否拦截该事件，一旦拦截后该事件的后续事件(如DOWN之后的MOVE和UP)都直接拦截，不会再进行判断。
> 3. 如果父View进行拦截，父View调用 `onTouchEvent` 进行处理。
> 4. 如果父View不进行拦截，会调用 子`View` 的 `dispatchTouchEvent` 进行事件的层层分发。

dispatchTouchEvent

3、dispatchTouchEvent的作用

> 1. 用于进行事件的分发
> 2. 只要事件传给当前View，该方法一定会被调用
> 3. 返回结果受到当前View的onTouchEvent和下级View的dispatchTouchEvent影响
> 4. 表示是否消耗当前事件

4、ViewGroup事件分发伪代码:

```
public boolean dispatchTouchEvent(MotionEvent ev){
    boolean consume = false;
    boolean intercepted = false;

    intercepted = onInterceptTouchEvent(ev);
    // 1、没有被拦截，分发给子View
    if(intercepted == false){
        consume = child.dispatchTouchEvent(ev);
    }
    // 2、事件被拦截因此自己进行处理 || 子View没有消耗该事件因此自己进行处理
    if(intercepted == true || consume == false){
        // 3、交给当前View进行处理（调用的是View的dispatchTouchEvent，该方法就是处理事件，等效于onTouchEvent）
        consume = super.dispacthTouchEvent(ev);
    }

    return consume;
}
```

> super.dispacthTouchEvent(ev): 就是调用ViewGroup父类View的dispacthTouchEvent方法。该方法是直接对事件的处理。

5、View事件分发伪代码:

```
public boolean dispatchTouchEvent(MotionEvent event) {
    boolean result = false;
    // 1. 判断是否有OnTouchListener，返回true，则处理完成
    if (mOnTouchListener != null){
        result = mOnTouchListener.onTouch(this, event);
    }
    // 2. 事件没有被消耗。并且，如果有代理，会执行代理的onTouchEvent方法
    if (result == false && mTouchDelegate != null) {
        result = mTouchDelegate.onTouchEvent(event);
    }
    // 3. 事件没有被消耗。才会调用onTouchEvent
    if (result == false) {
        result = onTouchEvent(event);

        // 4. 接收到UP事件，就会执行OnClickListener的onClick方法
        if(MotionEvent.ACTION_UP == action && mOnClickListener != null){
            mOnClickListener.onClick(event);
        }
    }
    // 5. 返回事件处理的结果(是否消耗该事件)
    return result;
}
```

> 1. mTouchDelegate和mOnClickListener本质都是在onTouchEvent中执行的，作为伪代码就忽视这些细节了。并不影响整个流程的层级。

6、View和ViewGroup在dispatchTouchEvent上的区别

> 1. ViewGroup在dispatchTouchEvent()中会进行事件的分发。
> 2. View在dispatchTouchEvent()中会对该事件进行处理。

onInterceptTouchEvent

7、onInterceptTouchEvent的作用

> 1. 在dispatchTouchEvent的内部调用，用于判断是否拦截某个事件

8、View和ViewGroup在onInterceptTouchEvent上的区别

> 1. View没有该方法，View会处理所有收到的事件，但不一定会消耗该事件。
> 2. onInterceptTouchEvent是ViewGroup中添加的方法，用于判断是否拦截该事件。

onTouchEvent

9、onTouchEvent的作用

1. 在dispatchTouchEvent的中调用，用于处理点击事件
2. 返回结果表示是否消耗当前事件

事件传递规则与要点

事件传递规则

10、事件的传递规则：

1. 点击事件产生后，会先传递给根ViewGroup，并调用dispatchTouchEvent
2. 之后会通过onInterceptTouchEvent判断是否拦截该事件，如果true，则表示拦截并交给该ViewGroup的onTouchEvent方法进行处理
3. 如果不拦截，则当前事件会传递给子元素，调用子元素的dispatchTouchEvent，如此反复直到事件被处理

11、View处理事件的优先级

1. 在View需要处理事件时，会先调用OnTouchListener的onTouch方法，并判断onTouch的返回值
2. 返回true，表示处理完成，不会调用onTouchEvent方法
3. 返回false，表示未完成，调用onTouchEvent方法进行处理
4. 可见，onTouchEvent的优先级没有OnTouchListener高
5. onTouchEvent没有消耗的话就会交给 `TouchDelegate`的`onTouchEvent` 去处理。
6. 如果最后事件都没有消耗，会在 `onTouchEvent` 中执行 `performClick()` 方法，内部会执行OnClickListener的onClick方法，优先级最低，属于事件传递尾端

12、点击事件传递过程遵循如下顺序：

1. Activity->Window->View->分发
2. 如果View的onTouchEvent返回false，则父容器的onTouchEvent会被调用，最终可以传递到Activity的onTouchEvent

13、事件传递规则要点

1. View一旦拦截事件，则整个事件序列都由它处理(ACTION_DOWN\UP等)，onInterceptTouchEvent不会再调用(因为默认都拦截了)
2. 但是一个事件序列也可以通过特殊方法交给其他View处理(onTouchEvent)
3. 如果View开始处理事件(已经拦截)，如果不消耗ACTIO_DOWN事件(onTouchEvent返回false)，则同一事件序列的剩余内容都直接交给父onTouchEvent处理
4. View消耗了ACTION_DOWN，但不处理其他的事件，整个事件序列会消失(父onTouchEvent)不会调用。这些消失的点击事件最终会传给Activity处理。
5. ViewGroup默认不拦截任何事件(onInterceptTouchEvent默认返回false)
6. View没有onInterceptTouchEvent方法，一旦有事件传递给View，onTouchEvent就会被调用
7. View的onTouchEvent默认都会消耗事件return true, 除非该View不可点击(clickable和longClickable同时为false)
8. View的enable属性不影响onTouchEvent的默认返回值。即使是disable状态。
9. onClick的发生前提是当前View可点击，并且收到了down和up事件
10. 事件传递过程是由父到子，层层分发，可以通过requestDisallowInterceptTouchEvent让子元素干预父元素的事件分发(ACTION_DOWN除外)

Activity的事件分发

14、Activity事件分发的过程

1. 事件分发过程：Activity->Window->Decor View(当前界面的底层容器，setContentView的View的父容器)->ViewGroup->View
2. Activity的dispatchTouchEvent，会交给Window处理(`getWindow().superDispatchTouchEvent()`)，
3. 返回true：事件全部结束
4. 返回false：所有View都没有处理(onTouchEvent返回false)，则调用Activity的onTouchEvent

Window的事件分发

15、Window事件分发

1. Window和superDispatchTouchEvent分别是抽象类和抽象方法
2. Window的实现类是PhoneWindow
3. PhoneWindow的 `superDispatchTouchEvent()` 直接调用 `mDecor.superDispatchTouchEvent()` ,也就是直接传给了DecorView

DecorView的事件分发

16、DecorView的事件分发

1. DecorView继承自FrameLayout
2. DecorView的 `superDispatchTouchEvent()` 会调用 `super.dispatchTouchEvent()` ——也就是 `ViewGroup` 的 `dispatchTouchEvent` 方法，之后就会层层分发下去。

根View的事件分发

17、根View的事件分发

> 1. 顶层View调用dispatchTouchEvent
> 2. 调用onInterceptTouchEvent方法
> 3. 返回true，事件由当前View处理。如果有onTouchiListener，会执行onTouch，并且屏蔽掉onTouchEvent。没有则执行onTouchEvent。如果设置了onClickListener，会在onTouchEvent后执行onClickListener
> 4. 返回false，不拦截，交给子View重复如上步骤。

ViewGroup的事件分发

18、ViewGroup的dispatchTouchEvent事件分发解析

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    boolean handled = false;
    //1. 过滤掉不符合安全策略的事件
    if (onFilterTouchEventForSecurity(ev)) {
        final boolean intercepted;
        /**==========================================
         * 2. 一旦一系列事件中的某个事件被拦截，后续的事件都会直接拦截，不会再判断
         *  情景1：为MotionEvent.ACTION_DOWN，等式为true，进入判断是否拦截
         *  情景2：不为ACTION_DOWN，(mFirstTouchTarget!=null)系列事件都没有被拦截，等式为true，进入判断是否拦截
         *  情景3：不为ACTION_DOWN，(mFirstTouchTarget=null)前面事件被拦截，等式为false
         *==========================================*/
        if (actionMasked == MotionEvent.ACTION_DOWN || mFirstTouchTarget != null) {
            /**==========================================
             *3. 由子View请求ViewGroup不要拦截该事件
             * 1-子View会通过`requestDisallowInterceptTouchEvent`设置FLAG_DISALLOW_INTERCEPT标志位
             * 2-ACTION_DOWN会重置FLAG_DISALLOW_INTERCEPT标志位，因此无法被子View影响
             *==========================================*/
            final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
            if (!disallowIntercept) {
                //4. 判断ViewGroup是否拦截该事件
                intercepted = onInterceptTouchEvent(ev);
                ev.setAction(action); // restore action in case it was changed
            } else {
                //5. 由子View控制不拦截该事件(前提是DOWN没有被拦截)
                intercepted = false;
            }
        } else {
            //6. ACTION_UP\MOVE等系列事件被拦截过，因此后续的全部拦截，不会重新判断
            intercepted = true;
        }
        ......
        //7. 没有被拦截，交给子View处理
        if (!canceled && !intercepted) {
            //8. 遍历所有子元素，并判断是否能接受点击事件，以及点击事件坐标是否在子元素内。
            for (int i = childrenCount - 1; i >= 0; i--) {
                //判断是否能接受点击事件，不能就直接continue
                if (childWithAccessibilityFocus != null) {
                    if (childWithAccessibilityFocus != child) {
                        continue;
                    }
                }
                //判断点击事件坐标是否在子元素内，不在就直接continue
                if (!canViewReceivePointerEvents(child) || !isTransformedTouchPointInView(x, y, child, null)) {
                    continue;
                }
                //分发给子View处理，内部就是调用子元素的`dispatchTouchEvent`方法
                if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {
                    //子View消耗并且处理该事件
                    alreadyDispatchedToNewTouchTarget = true;

                    break;
                }
            }
        }
        //9. 事件被拦截或者子View未消耗该事件：自己处理该事件
        if (mFirstTouchTarget == null) {
            handled = dispatchTransformedTouchEvent(ev, canceled, null, TouchTarget.ALL_POINTER_IDS);
        }
        .......
    }
    return handled;
}
```

View的事件分发和事件处理

19、View的事件处理中的优先级(super.onTouchEvent后执行自定义内容)

```
public class TestView extends View{

    @Override
    public boolean onTouchEvent(MotionEvent event) {
       super.onTouchEvent(event); // 2
       result = xxx自定义处理xxx; // 4
       return result;
    }
}
```

方法			优先级
OnTouchListener的onTouch			1
onTouchEvent			
	super.onTouchEvent		
		TouchDelegate的onTouchEvent	2
		OnClickListener的onClick(会通过post投递到主线程的消息队列中)	4
	onTouchEvent中 自定义处理的内容		3

1. 1: OnTouchListener如果消耗了该事件(return true),后续的方法都不会执行。
2. 2: TouchDelegate如果消耗了该事件(return true),后续的方法都不会执行。
3. 3: OnTouchListener和TouchDelegate没有消耗事件：会执行onTouchEvent中 自定义处理的内容
4. 4: OnTouchListener和TouchDelegate没有消耗事件：只要接收到UP事件，就会执行OnClickListener的onClick方法

20、View的事件处理中的优先级(super.onTouchEvent后执行自定义内容)

```
public class TestView extends View{

    @Override
    public boolean onTouchEvent(MotionEvent event) {
       result = xxx自定义处理xxx;  //2
       super.onTouchEvent(event); // 3
       return result;
    }
}
```

方法			优先级
OnTouchListener的onTouch			1
onTouchEvent			
	onTouchEvent中 自定义处理的内容		2
	super.onTouchEvent		
		TouchDelegate的onTouchEvent	3
		OnClickListener的onClick(会通过post投递到主线程的消息队列中)	4

1. 1: OnTouchListener如果消耗了该事件(return true),后续的方法都不会执行。
2. 2: OnTouchListener没有消耗事件：会执行onTouchEvent中 自定义处理的内容
3. 3: OnTouchListener没有消耗事件：会执行TouchDelegate的onTouchEvent。如果消耗了该事件(return true),后续的方法都不会执行。
4. 4: OnTouchListener和TouchDelegate没有消耗事件：只要接收到UP事件，就会执行OnClickListener的onClick方法

21、TouchDelegate是什么?

1. 用于 增加触摸区域 ---比如在Button的范围之外去点击，也能触发点击事件。

22、View对点击事件的处理过程(不包括ViewGroup)

```
/**=========================
 * 1. 事件分发(OnTouchListener或者onTouchEvent直接处理)
 *=========================*/
public boolean dispatchTouchEvent(MotionEvent event) {
    boolean result = false;
    ......
    //1. 采用安全策略过滤事件
    if (onFilterTouchEventForSecurity(event)) {
        ListenerInfo li = mListenerInfo;
        //2. 判断是否有OnTouchListener，返回true，则处理完成
        if (li != null && li.mOnTouchListener != null
                && (mViewFlags & ENABLED_MASK) == ENABLED
                && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }
        //3. 如果OnTouch返回true，不会调用onTouchEvent
        if (!result && onTouchEvent(event)) {
            result = true;
        }
    }
    ......
    return result;
}
/**=========================
 * 2. 事件处理onTouchEvent
 *=========================*/
public boolean onTouchEvent(MotionEvent event) {
    //0. 获取点击状态
    final boolean clickable = ((viewFlags & CLICKABLE) == CLICKABLE
            || (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)
            || (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE;
    //1. View不可用状态下(可点击状态-会消耗该事件,不可点击不消耗)
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (action == MotionEvent.ACTION_UP && (mPrivateFlags & PFLAG_PRESSED) != 0) {
            setPressed(false);
        }
        return clickable;//根据是否可点击决定是否消耗
    }
    //2. 如果有代理，会执行代理的onTouchEvent方法（会消耗该事件）
    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }
    //3. View可点击(消耗该事件)
    if (clickable || (viewFlags & TOOLTIP) == TOOLTIP) {
        switch (action) {
            case MotionEvent.ACTION_UP:
                  ......
                //5. 如果设置了`OnClickListener`，performClick内部会调用onClick方法
                  performClick();
                  ......
                break;
        }
        return true;
    }
    //4. 可用状态&&没有代理&&不可点击：不消耗该事件
    return false;
}
```

滑动冲突(8题)

滑动冲突的三种场景

1、滑动冲突的三种场景

> 1. 内层和外层滑动方向不一致：一个垂直，一个水平
> 2. 内存和外层滑动方向一致：均垂直or水平
> 3. 前两者层层嵌套

滑动冲突处理原则和解决办法

2、 滑动冲突处理原则

> 1. 对于内外层滑动方向不同，只需要根据滑动方向来给相应控件拦截
> 2. 对于内外层滑动方向相同，需要根据业务来进行事件拦截

> 3. 前两者嵌套的情况，根据前两种原则层层处理即可。

3、 滑动冲突解决办法

> 1. 外部拦截：在父容器进行拦截处理，需要重写父容器的onInterceptTouchEvent方法
> 2. 内部拦截：父容器不拦截任何事件，事件都传递给子元素。子元素需要就处理，否则给父容器处理。需要配合 `requestDisallowInterceptrtTouchEvent` 方法。

外部拦截

4、外部拦截法要点

> 1. 父容器的 `onInterceptTouchEvent` 方法中处理
> 2. ACTION_DOWN不拦截，一旦拦截会导致后续事件都直接交给父容器处理。
> 3. ACTION_MOVE中根据情况进行拦截，拦截：return true，不拦截：return false（外部拦截核心）
> 4. ACTION_UP不拦截，如果父控件拦截UP，会导致子元素接收不到UP进一步会让onClick方法无法触发。此外UP拦截也没什么用。

5、onClick方法生效的两个条件?

> 1. `View`可以点击
> 2. 接收到了`DOWN`和`UP`事件

6、外部拦截，自定义ScrollView

```
//Kotlin
class CustomScrollView(context: Context,
                       attrs: AttributeSet?,
                       defStyleAttr: Int,
                       defStyleRes: Int): ScrollView(context, attrs, defStyleAttr, defStyleRes) {

    constructor(context: Context) : this(context, null, 0, 0)
    constructor(context: Context, attrs: AttributeSet?) : this(context, attrs, 0, 0)
    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int) : this(context, attrs, defStyleAttr, 0)

    var lastX: Int = 0
    var lastY: Int = 0

    override fun dispatchTouchEvent(ev: MotionEvent): Boolean {

        val curX = ev.x.toInt()
        val curY = ev.y.toInt()

        when(ev.action){
            ACTION_DOWN -> {
                parent.requestDisallowInterceptTouchEvent(true)
            }
            ACTION_MOVE -> {
                //如果是水平滑动则交给父容器处理
                if(Math.abs(curX - lastX) > Math.abs(curY - lastY)){
                    parent.requestDisallowInterceptTouchEvent(false)
                }
            }
            ACTION_UP -> null
            else -> null
        }
        lastX = curX
        lastY = curY
        return super.dispatchTouchEvent(ev)
    }
}
```

内部拦截

7、内部拦截法要点

> 1. 子View的 `dispatchTouchEvent` 方法处理
> 2. ACTION_DOWN，让父容器不拦截(也不能拦截，否则会导致后续事件都无法传递到子View)- `parent.requestDisallowInterceptTouchEvent(true)`
> 3. ACTION_MOVE,如父容器需要该事件，则父容器拦截requestDisallowInterceptTouchEvent(false)
> 4. ACTION_UP，无操作，正常执行

8、内部拦截Kotlin代码

```kotlin
//Kotlin
class CustomHorizontalScrollView(context: Context,
                                 attrs: AttributeSet?,
                                 defStyleAttr: Int,
                                 defStyleRes: Int): HorizontalScrollView(context, attrs, defStyleAttr, defStyleRes){
  //构造器
    constructor(context: Context): this(context, null, 0, 0)
    constructor(context: Context, attrs: AttributeSet?): this(context, attrs, 0, 0)
    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int): this(context, attrs, defStyleAttr, 0)

    var downX: Int = 0
    var downY: Int = 0
  //拦截处理
    override fun onInterceptTouchEvent(ev: MotionEvent): Boolean {
        var intercepted = super.onInterceptTouchEvent(ev)
        when(ev.action){
        //不拦截
            ACTION_DOWN -> {
                downX = ev.x.toInt()
                downY = ev.y.toInt()
                intercepted = false
            }
        //判断是否拦截
            ACTION_MOVE -> {
                val curX = ev.x.toInt()
                val curY = ev.y.toInt()
                //水平滑动进行拦截
                if(Math.abs(curX - downX) > Math.abs(curY - downY)){
                    intercepted = true
                }
            }
        //不拦截
            ACTION_UP -> intercepted = false
            else -> null
        }
        return intercepted
    }
}
```

滑动(39题)

滑动的7种实现方法

1、View滑动的7种方法：

> 1. layout：对View进行重新布局定位。在onTouchEvent()方法中获得控件滑动前后的偏移。然后通过layout方法重新设置。
> 2. offsetLeftAndRight和offsetTopAndBottom:系统提供上下/左右同时偏移的API。onTouchEvent()中调用
> 3. LayoutParams: 更改自身布局参数
> 4. scrollTo/scrollBy: 本质是移动View的内容，需要通过父容器的该方法来滑动当前View
> 5. Scroller: 平滑滑动，通过重载 `computeScroll()` ，使用 `scrollTo/scrollBy` 完成滑动效果。
> 6. 属性动画: 动画对View进行滑动
> 7. ViewDragHelper: 谷歌提供的辅助类，用于完成各种拖拽效果。

2、Layout实现滑动

```
/*================================*
* onTouchEvent-进行偏移计算，之后调用layout
*================================*/
 public boolean onTouchEvent(MotionEvent event) {
     float curX = event.getX(); //手指实时位置的X
     float curY = event.getY(); //Y
     switch(event.getAction()){
        case MotionEvent.ACTION_MOVE:
            int offsetX = (int)(curX - downX); //X偏移
            int offsetY = (int)(curY - downY); //Y偏移
    /**=============================================
     * 变化后的距离=getLeft(当前控件距离父控件左边的距离)+偏移量—调用layout重新布局
     *=============================================*/
            layout(getLeft() + offsetX, getTop() + offsetY, getRight() + offsetX, getBottom() + offsetY);
            break;
        case MotionEvent.ACTION_DOWN:
            downX = curX; //按下时的坐标
            downY = curY;
            break;
     }
     return true;
 }
```

3、offsetLeftAndRight和offsetTopAndBottom实现滑动

```
/*================================*
* onTouchEvent-进行偏移计算，直接调用
*================================*/
 public boolean onTouchEvent(MotionEvent event) {
     float curX = event.getX(); //手指实时位置的X
     float curY = event.getY(); //Y
     switch(event.getAction()){
        case MotionEvent.ACTION_MOVE:
            int offsetX = (int)(curX - downX); //X偏移
            int offsetY = (int)(curY - downY); //Y偏移
     /**=============================================
      * 对left和right，top和bottom同时偏移
      *=============================================*/
            offsetLeftAndRight(offsetX);
            offsetTopAndBottom(offsetY);
            break;
        case MotionEvent.ACTION_DOWN:
            downX = curX; //按下时的坐标
            downY = curY;
            break;
     }
     return true;
 }
```

4、LayoutParams实现滑动：

> 1. 通过父控件设置View在父控件的位置，但需要指定父布局的类型，不好
> 2. 用ViewGroup的MariginLayoutParams的方法去设置margin

```
//方法一：通过布局设置在父控件的位置。但是必须要有父控件，而且要指定父布局的类型，不好的方法。
RelativeLayout.LayoutParams layoutParams = (RelativeLayout.LayoutParams) getLayoutParams();
layoutParams.leftMargin = getLeft() + offsetX;
layoutParams.topMargin = getTop() + offsetY;
setLayoutParams(layoutParams);

/**=============================================
 * 方法二：用ViewGroup的MarginLayoutParams的方法去设置marign
 * 优点：相比于上面方法，就不需要知道父布局的类型。
 * 缺点：滑动到右侧控件会缩小
 *=============================================*/
ViewGroup.MarginLayoutParams mlayoutParams = (ViewGroup.MarginLayoutParams) getLayoutParams();
mlayoutParams.leftMargin = getLeft() + offsetX;
mlayoutParams.topMargin = getTop() + offsetY;
setLayoutParams(mlayoutParams);
```

5、scrollTo\scrollBy实现滑动

> 1. 都是View提供的方法。
> 2. scrollTo-直接到新的x,y坐标处。
> 3. scrollBy-基于当前位置的相对滑动。
> 4. scrollBy-内部是调用scrollTo.
> 5. `scrollTo\scrollBy`，效果是移动View的内容，因此需要在View的父控件中调用。

```
// 1、移动到目标位置
((View)getParent()).scrollTo(dstX, dstY);
// 2、相对滑动：且scrollBy是父容器进行滑动，因此偏移量需要取负
((View)getParent()).scrollBy(-offsetX, -offsetY);
```

6、scrollTo/By内部的mScrollX和mScrollY的意义

> 1. mScrollX的值，相当于手机屏幕相对于View左边缘向右移动的距离，手机屏幕向右移动时，mScrollX的值为正；手机屏幕向左移动(等价于View向右移动)，mScrollX的值为负。
> 2. mScrollY和X的情况相似，手机屏幕向下移动，mScrollY为+正值；手机屏幕向上移动，mScrollY为-负值。
> 3. mScrollX/Y是根据第一次滑动前的位置来获得的，例如：第一次向左滑动200(等于手机屏幕向右滑动200)，mScrollX = 200；第二次向右滑动50, mScrollX = 200 + （-50）= 150，而不是（-50）。

7、动画实现滑动的方法

> 1. 可以通过传统动画或者属性动画的方式实现
> 2. 传统动画需要通过设置fillAfter为true来保留动画后的状态(但是无法在动画后的位置进行点击操作，这方面还是属性动画好)
> 3. 属性动画会保留动画后的状态，能够点击。

8、ViewDragHelper

> 1. 通过 `ViewDragHelper` 去自定义 `ViewGroup` 让其 子`View` 具有滑动效果。

弹性滑动

Scroller

9、Scroller的作用

> 1. 用于 封装滑动
> 2. 提供了 基于时间的滑动偏移值 ， 但是实际滑动需要我们去负责。

10、Scroller的要点

> 1. 调用startScroll方法时， `Scroller`只是单纯的保存参数
> 2. 之后的invalidate方法导致的View重绘
> 3. View重绘之后draw方法会调用自己实现的computeScroll()，才真正实现了滑动

11、Scroller的使用

```
// 1、初始化
Scroller mScroller = new Scroller(getContext());

// 2、重写View的方法computeScroll
public void computeScroll() {
        super.computeScroll();
        //判断scroller是否执行完毕。
        if(mScroller.computeScrollOffset()){
            ((View)getParent()).scrollTo(mScroller.getCurrX(), mScroller.getCurrY());
            //通过重绘来不断调用 computeScroll
            invalidate();
        }
}

// 3、开始滑动
case MotionEvent.ACTION_UP:
      View viewGroup = (View) getParent();
      mScroller.startScroll(viewGroup.getScrollX(), viewGroup.getScrollY(),
                        -viewGroup.getScrollX(), -viewGroup.getScrollY());
      invalidate();
       break;
```

12、Scroller工作原理

> 1. Scroller本身不能实现View的滑动，需要配合View的computeScroll方法实现弹性滑动
> 2. 不断让View重绘，每一次重绘距离滑动的开始时间有一个时间间隔，通过该时间可以得到View当前的滑动距离
> 3. View的每次重绘都会导致View的小幅滑动，多次小幅滑动就组成了弹性滑动

动画

13、通过动画实现弹性滑动

延时策略

14、通过延时策略实现弹性滑动。

> 1. 通过handler、View的postDelayed、或者线程的sleep方法。
> 2. 实现思路：例如将View滑动100像素，通过Handler可以每100ms发送一次消息让其滑动10像素，最终会在1000ms内滑动100像素。

侧滑菜单

DraweLayout

15、DrawerLayout是什么?

> 1. `Google` 推出的 侧滑菜单 。

16、DrawerLayout的使用

> 1. 侧滑菜单 的布局需要用 `layout_gravity` 属性指定：
> 2. `left/start` ： 菜单位于左侧
> 3. `top/bottom` ： 菜单位于右侧

17、DrawerLayout的方法

> 1-打开： `drawerLayout.openDrawer(button);`
> 2-关闭： `drawerLayout.closeDrawer(button);`
> 3-设置监听器(DrawerListener) `drawerLayout.setDrawerListener(xxx);`

SlidingPanelLayout

18、SlidingPaneLayout是什么

> 1. 提供一种类似于 `DrawerLayout` 的侧滑菜单效果，“效果并不好”
> 2. `xml` 布局中第一个 `ChildView` 就是 左侧菜单的内容 ， 第二个 `ChildView` 就是 主体内容

NavigationView

19、NavigationView的作用

> 1. 配合 `DrawerLayout` 使用用于实现其中的 左侧菜单效果
> 2. Google在5.0之后推出NavigationView，
> 3. 左侧菜单效果 整体上分为两部分，上面一部分叫做 `HeaderLayout` ， 下面的那些点击项都是 `menu` 。

ViewDragHelper

20、ViewDragHelper的作用

> 1. 用于 编写自定义`ViewGroup` 的 工具类，能轻易实现 `QQ`侧滑菜单 的效果。
> 2. 位于 `android.support.v4.widget.` 。
> 3. 提供一系列 操作和状态追踪 用于帮助用户进行 拖拽和定位子`View`

21、ViewDragHelper的简单实例

> 实现ChildView可以自由拖拽的ViewGroup
>
> 1. 创建 `ViewDragHelper`
> 2. 将 `ViewGroup` 的事件处理在 `onTouchEvent` 中交给 `ViewDragHelper`
> 3. 自定义 `ViewDragHelper.Callback` 实现一些触摸回调，用于实现效果。

22、ChildView为Button或者 `clickable = true` 时无法拖动的解决办法

> 1. 正常流程: 如果子View不消耗事件，那么整个手势（DOWN-MOVE-UP）都是直接进入onTouchEvent，在onTouchEvent的DOWN的时候就确定了captureView。
> 2. 子View消耗事件：会先走onInterceptTouchEvent方法，判断是否可以捕获，而在判断的过程中会去判断另外两个回调的方法：getViewHorizontalDragRange和getViewVerticalDragRange，只有这两个方法返回大于0的值才能正常的捕获。

```
/**
 * 返回子View水平滑动范围。
 * return 0:  则该ChildView不会滑动。
 */
@Override
public int getViewHorizontalDragRange(View child)
{
    return getMeasuredWidth()-child.getMeasuredWidth();
}

/**
 * 返回子View垂直滑动范围。
 * return 0:  则该ChildView不会滑动。
 */
@Override
public int getViewVerticalDragRange(View child)
{
    return getMeasuredHeight()-child.getMeasuredHeight();
}
```

ViewDragHelper.Callback

23、ViewDragHelper.Callback的方法和作用

方法	作用
onViewDragStateChanged()	当ViewDragHelper状态发生变化时回调（ IDLE，DRAGGING，SETTING-自动滚动时 ）
onViewPositionChanged()	ChildView位置改变时回调
onViewCaptured()	捕获ChildView时回调
onViewReleased()	松开ChildView时回调
onEdgeTouched()	当触摸到边界时回调
onEdgeLock()	true的时候会锁住当前的边界，false则unLock。
onEdgeDragStarted()	边缘拖拽开始时回调
getOrderedChildIndex()	在同一个坐标（x,y）下应该去获取哪一个View。（mViewDragHelper.findTopChildUnder中需要用到）
getViewHorizontalDragRange()	获取水平方向上的拖拽范围
getViewVerticalDragRange()	获取垂直方向上的拖拽范围
tryCaptureView()	判断是否捕获当前View
clampViewPositionHorizontal()	控制Child在水平方向上的边界
clampViewPositionVertical()	控制Child在垂直方向上的边界

GestureDetector

24、GestureDetector作用和注意点

> 1. 探测 手势 和 事件 ， 需要通过提供的 MotionEvent
> 2. 该类仅能用于 touch触摸 提供的 MotionEvent ， 不能用于 traceball events(追踪球事件)
> 3. 可以在 自定义View 中重写 onTouchEvent() 方法并在里面用 GestureDetector 接管。
> 4. 可以在 View的setOnTouchListener的onTouch 中将 点击事件 交给 GestureDetector 接管。

25、GestureDetector提供的接口

> 1. OnGestureListener
> 2. OnDoubleTapListener
> 3. OnContextClickListener
> 4. SimpleOnGestureListener

OnGestureListener

26、OnGestureListener作用

> 1. 用于在 手势 产生时，去通知监听者。
> 2. 该 监听器 会监听所有的手势，如果只需要监听一部分可以使用 SimpleOnGestureListener

27、OnGestureListener能监听哪些手势(6种)?

> 1. 按下操作。
> 2. 按下之后，Move和Up之前。用于提供视觉反馈告诉用户已经捕获了他们的行为。
> 3. 抬起操作。
> 4. 滑动操作(由Down MotionEvent e1触发，当前是Move MotionEvent e2)
> 5. 长按操作。
> 6. 猛扔操作。
>
> 所有有返回值的回调方法， `return true` -消耗该事件； `return false` -不消耗该事件

OnDoubleTapListener

28、OnDoubleTapListener作用

> 1. 监听“双击操作”
> 2. 监听“确认的单击操作”---该单击操作之后的操作无法构成一次双击。

29、OnDoubleTapListener能监听哪些手势(3种)?

> 1. 单击操作。
> 2. 双击操作.
> 3. 双击操作之间发生了down、move或者up事件。

OnContextClickListener

30、OnContextClickListener的作用

> 1. 鼠标/触摸板的右击操作

31、OnContextClickListener的方法

```
/**=======================================
 * 鼠标/触摸板 右键点击
 * 1. 需要确保在View的onGenericMotionEvent中进行拦截
 * 2. 最终交给GestureDetector的onGenericMotionEvent方
 *=======================================*/
public interface OnContextClickListener {
    boolean onContextClick(MotionEvent e);
}
```

32、OnContextClickListener的使用

> 需要在View的

```
// 1、设置OnContextClickListener监听器
GestureDetector gestureDetector = new GestureDetector(...);
gestureDetector.setContextClickListener(new GestureDetector.OnContextClickListener() {...});

// 2、拦截View的onGenericMotion方法
imageView.setOnGenericMotionListener(new View.OnGenericMotionListener() {
    @Override
    public boolean onGenericMotion(View v, MotionEvent event) {
        return gestureDetector.onGenericMotionEvent(event);
    }
});
```

SimpleOnGestureListener

33、SimpleOnGestureListener的作用

> 1. 实现了 `GestureDetector` 的所有监听器，可以选择性实现需要的方法。
> 2. 不需要去实现那些无关的方法。

6.7-辅助类

ViewConfiguration

34、ViewConfiguration的作用

> 1. 定义所有 `UI` 所需要用的标准常量。
> 2. 包括双击时间间隔、滑动最小距离等等。
> 3. 获取常量需要通过类的静态方法或者成员方法获得。

> 4. 静态方法：与设备无关
> 5. 成员方法: 与设备有关

35、ViewConfiguration的使用方法

```
//类的静态方法
ViewConfiguration.getDoubleTapTimeout(); //构成双击的时间间隔

//类的成员方法
ViewConfiguration configuration = ViewConfiguration.get(getBaseContext());
configuration.getScaledTouchSlop(); //滑动的最小距离
```

36、ViewConfiguration常量汇总

常量	介绍	作用	类方法or成员方法
configuration.getScaledTouchSlop()	滑动的最小距离， 低于该值则认为没有滑动。	在两次滑动距离小于该值时可以判断未滑动， 以提高用户体验。	成员方法 (该值与设备有关)
configuration.hasPermanentMenuKey()	设备是否具有实体按键 (返回按键等)。		成员方法 (该值与设备有关)
ViewConfiguration.getKeyRepeatTimeout()	重复按键的间隔时间。	两次按键小于该事件则表示属于同一次按键	类方法 (该值与设备无关)

VelocityTracker

37、VelocityTracker的作用

> 1. 速度追踪：手指滑动中水平和竖直方向的速度
> 2. 速度是指：在给定时间内手机滑过的像素数，如果从右到左，就是负值(例如1000ms内速度为100，就是在1s内滑过100个像素)
> 3. 使用完毕时需要调用 `clear` 和 `recycle` 方法进行清理并回收内存

38、VelocityTracker的使用

> 1. 在View的onTouchEvent中追踪当前点击事情的速度
> 2. 通过VelocityTracker的computeCurrentVelocity方法先计算速度
> 3. 再获取VelocityTracker的xVelocity/yVelocity获取速度

39、VelocityTracker代码如下

```
        //追踪速度
        val velocityTracker = VelocityTracker.obtain()
        velocityTracker.addMovement(event)

        //获取当前速度，但必须在获取前进行速度计算
        velocityTracker.computeCurrentVelocity(1000) //时间单位
        val xVelocity = velocityTracker.xVelocity
        val yVelocity = velocityTracker.yVelocity

        velocityTracker.clear()
        velocityTracker.recycle()
```

面试题：考考你

1. 如果设置了onClickListener, 但是onClick()没有调用，可能产生的原因?

> 1. 父View拦截了事件，没有传递到当前View
> 2. View的Enabled = false(`setEnabled(false)`): view处于不可用状态，会直接返回。
> 3. View的Clickable = false(`setClickable\setLongClickable(false)`):view不可以点击，不会执行onClick
> 4. View设置了onTouchListener，且消耗了事件。会提前返回。
> 5. View设置了TouchDelegate，且消耗了事件。会提前返回。

扩展知识

通过XML创建View的原理

1、通过XML创建View并且进行换肤的原理

> 1. Activity是通过 `Factory` 进行View的创建
> 2. 自定义 `Factory` 就能拦截创建过程，创建自己的 `View`

2、AppCompatActivity的OnCreate流程

```
//AppCompatActivity.java
protected void onCreate(@Nullable Bundle savedInstanceState) {
    final AppCompatDelegate delegate = getDelegate();
    //1. 初始化LayoutInflater，并且设置过Factory(没有设置过就新建)
    delegate.installViewFactory();
    //2. 执行正常的onCreate流程
    delegate.onCreate(savedInstanceState);
    //xxx
    super.onCreate(savedInstanceState);
}
//AppCompatDelegateImpV9.java
public void installViewFactory() {
    LayoutInflater layoutInflater = LayoutInflater.from(mContext);
    //1. 没有Factory,系统会创建一个Factory去进行XML到View的转换
    if (layoutInflater.getFactory() == null) {
        LayoutInflaterCompat.setFactory2(layoutInflater, this);
    } else {
        if (!(layoutInflater.getFactory2() instanceof AppCompatDelegateImplV9)) {
            Log.i(TAG, "The Activity's LayoutInflater already has a Factory installed" + " so we can not install AppCompat's");
        }
    }
}
//LayoutInflaterCompat.java
public static void setFactory2(@NonNull LayoutInflater inflater, @NonNull LayoutInflater.Factory2 factory) {
    //1. 能将Factory接口绑定到创建View的LayoutInflater(IMPL类型为LayoutInflaterCompatBaseImpl)
    IMPL.setFactory2(inflater, factory);
}
//LayoutInflaterCompat.java内部类LayoutInflaterCompatBaseImpl:
static class LayoutInflaterCompatBaseImpl {
    //xxx
    public void setFactory2(LayoutInflater inflater, LayoutInflater.Factory2 factory) {
        inflater.setFactory2(factory);
        //xxx
    }
}
//LayoutInflater.java-完成Factory的创建
public void setFactory2(Factory2 factory) {
    //xxx
    if (mFactory == null) {
        mFactory = mFactory2 = factory;
    } else {
        mFactory = mFactory2 = new FactoryMerger(factory, factory, mFactory, mFactory2);
    }
}
```

3、AppCompatActivity的OnCreate中setContentView()的流程

```
//AppCompatDelegateImplV9.java
    public void setContentView(int resId) {
        //xxx
        //1. 获取到父容器Content
        ViewGroup contentParent = (ViewGroup) mSubDecor.findViewById(android.R.id.content);
        contentParent.removeAllViews();
        //2. 通过LayoutInflater加载布局文件
        LayoutInflater.from(mContext).inflate(resId, contentParent);
        mOriginalWindowCallback.onContentChanged();
    }

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

//LayoutInlfater.java
    public View inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean attachToRoot) {
        final Resources res = getContext().getResources();
        //xxx
        final XmlResourceParser parser = res.getLayout(resource);
        //1. 重点
        return inflate(parser, root, attachToRoot);
    }
//LayoutInlfater.java
    public View inflate(XmlPullParser parser, @Nullable ViewGroup root, boolean attachToRoot) {
        ...
        final String name = parser.getName();//控件名
        //1. 将XmlPullParser转换为View的属性AttributeSet,给其他方法使用
        final AttributeSet attrs = Xml.asAttributeSet(parser);
        //2. Temp是XML文件中的根布局(name为"LinearLayout"等等)
        final View temp = createViewFromTag(root, name, inflaterContext, attrs);
        //3. 将XML根布局中temp下面所有的子View都进行加载
        rInflateChildren(parser, temp, attrs, true);
        //4. 将根布局tmp中找到的所有View贴到root中(content view)
        if (root != null && attachToRoot) {
            root.addView(temp, params);
        }
        ...
    }

/**=====================================
* 通过提供的属性AttributeSet attrs, 创建View
* // LayoutInlfater.java
*======================================*/
    View createViewFromTag(View parent, String name, Context context, AttributeSet attrs, boolean ignoreThemeAttr) {
        //1. 彩蛋?<blink>标签会进行闪烁
        if (name.equals(TAG_1995)) {
            // Let's party like it's 1995!
            return new BlinkLayout(context, attrs);
        }
        //2. 通过Factory创建View
        View view;
        view = mFactory2.onCreateView(parent, name, context, attrs);

        //xxx
        return view;
    }

//AppCompatDelegateImplV9.java
    public final View onCreateView(View parent, String name, Context context, AttributeSet attrs) {
       ...
        //创建View
        return createView(parent, name, context, attrs);
    }
//AppCompatDelegateImplV9.java
    public View createView(View parent, final String name, @NonNull Context context, @NonNull AttributeSet attrs) {
        ...
        return mAppCompatViewInflater.createView(parent, name, context, attrs, inheritContext,
                IS_PRE_LOLLIPOP, /* Only read android:theme pre-L (L+ handles this anyway) */
                true, /* Read read app:theme as a fallback at all times for legacy reasons */
                VectorEnabledTintResources.shouldBeUsed() /* Only tint wrap the context if enabled */
        );
    }
//AppCompatViewInflater.java-最终完成从XML到View的转变
    public final View createView(View parent, final String name, Context context, AttributeSet attrs, ...) {

        View view = null;
        switch (name) {
            case "TextView":
                view = new AppCompatTextView(context, attrs);
```

```
                break;
            case "ImageView":
                view = new AppCompatImageView(context, attrs);
                break;
            case "Button":
                view = new AppCompatButton(context, attrs);
                break;
            case "EditText":
                view = new AppCompatEditText(context, attrs);
                break;
            case "Spinner":
                view = new AppCompatSpinner(context, attrs);
                break;
            case "ImageButton":
                view = new AppCompatImageButton(context, attrs);
                break;
            case "CheckBox":
                view = new AppCompatCheckBox(context, attrs);
                break;
            case "RadioButton":
                view = new AppCompatRadioButton(context, attrs);
                break;
            case "CheckedTextView":
                view = new AppCompatCheckedTextView(context, attrs);
                break;
            case "AutoCompleteTextView":
                view = new AppCompatAutoCompleteTextView(context, attrs);
                break;
            case "MultiAutoCompleteTextView":
                view = new AppCompatMultiAutoCompleteTextView(context, attrs);
                break;
            case "RatingBar":
                view = new AppCompatRatingBar(context, attrs);
                break;
            case "SeekBar":
                view = new AppCompatSeekBar(context, attrs);
                break;
        }
        ...
        return view;
    }
```

4、自定义Activity中通过Factory对控件的创建进行拦截，实现“换肤”效果：

```
public class SkinActivity extends AppCompatActivity{
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        LayoutInflaterCompat.setFactory2(LayoutInflater.from(this), new LayoutInflater.Factory2() {
            @Override
            public View onCreateView(View parent, String name, Context context, AttributeSet attrs) {
                AppCompatDelegate delegate = getDelegate();
                View view = delegate.createView(parent, name, context, attrs);
                return view;
            }

            @Override
            public View onCreateView(String name, Context context, AttributeSet attrs) {
                View view = null;
                switch (name) {
                    case "TextView":
                        view = new AppCompatTextView(context, attrs);
                        break;
                    case "ImageView":
                        view = new AppCompatImageView(context, attrs);
                        break;
                    case "Button":
                        view = new AppCompatButton(context, attrs);
                        break;
                    case "EditText":
                        view = new AppCompatEditText(context, attrs);
                        break;
                    //...
                }
                return view;
            }
        });
    }
}
```

参考资料

1. 从requestLayout()初探View的绘制原理
2. Android ViewDragHelper完全解析 自定义ViewGroup神器
3. 事件分发-TouchDelegate的简单使用
4. Android 性能优化<七>自定义view绘制优化
5. 自定义View起步：硬件加速对绘图的影响
6. View.post()源码解析