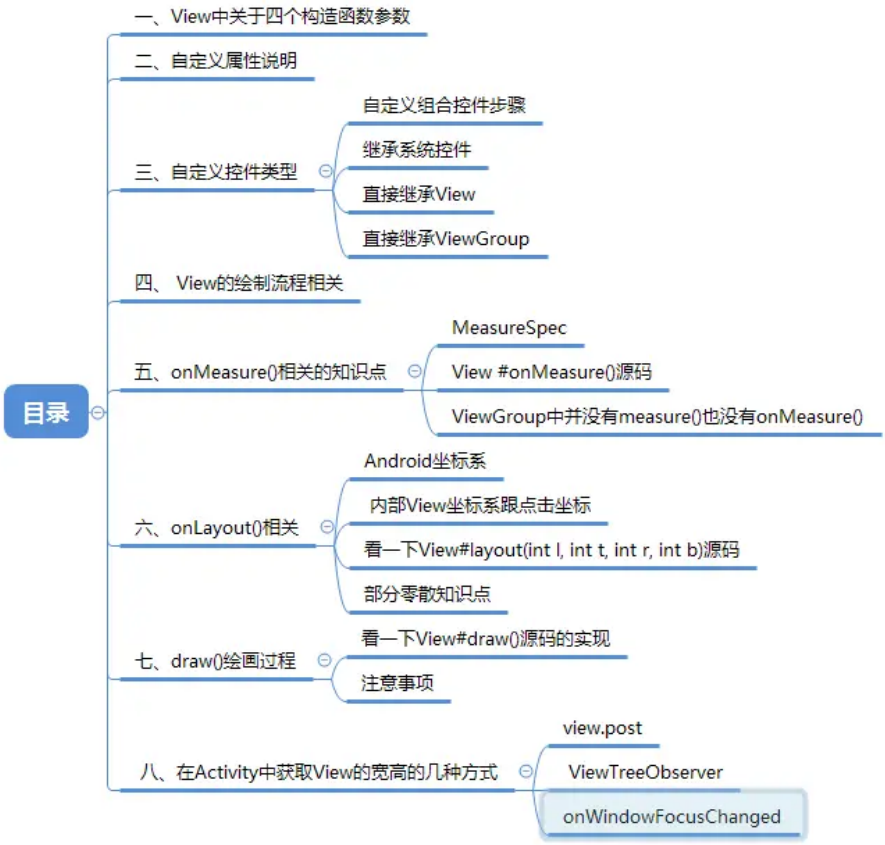
Android Study Plan

deepwaterooo

2021 年 12 月 24 日

目录

1 View 相关



1.1 自定义 View

1.1.1 一、View 中关于四个构造函数参数

- 自定义 View 中 View 的构造函数有四个

```
// 主要是在 java 代码中生命一个 View 时所调用，没有任何参数，一个空的 View 对象
public ChildrenView(Context context) {
    super(context);
}
// 在布局文件中使用该自定义 view 的时候会调用到，一般会调用到该方法
public ChildrenView(Context context, AttributeSet attrs) { // AttributeSet from .xml 设置
    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);
}
```

- 构造函数的传入参数说明
  - context: 上下文
  - AttributeSet attrs：从 xml 中定义的参数
  - intdefStyleAttr：主题中优先级最高的属性
  - intdefStyleRes：优先级次之的内置于 View 的 style(这里就是自定义 View 设置样式的地方)
- 多个地方定义属性,优先级排序 Xml 直接定义 > xml 中 style 引用 > defStyleAttr>defStyleRes > theme 直接定义

1.2 二、.xml 中的自定义属性

- 基本类型包括：integer, boolean, color, string, float, dimension, enum, flags, fraction, reference
- 基本类型略过，其它相对重要一点儿的

1.2.1 color ：引用颜色

1.2.2 dimension: 引用字体大小

- 定义

```
<attr name = "text_size" format = "dimension" />
```

- 使用：

```
<app:text_size = "28sp"  />
<app:text_size = "@android:dimen/app_icon_size" />
```

1.2.3 enum：枚举值

- 定义

```
<attr name="orientation">
  <enum name="horizontal" value="0" />
  <enum name="vertical" value="1" />
</attr>
```

- 使用：

```
<app:orientation = "vertical" />
```

1.2.4 **flags:** 标志（位或运行）主要作用 **=** 可以多个值

- 定义

```
<attr name="gravity">
  <flag name="top" value="0x01" />
  <flag name="bottom" value="0x02" />
  <flag name="left" value="0x04" />
  <flag name="right" value="0x08" />
  <flag name="center_vertical" value="0x16" />
</attr>
```

- 使用

```
<app:gravity = Top|left />
```

1.2.5 **fraction:** 百分数：

- 定义：

```
<attr name = "transparency" format = "fraction" />
```

- 使用：

```
<app:transparency = "80%"  />
```

1.2.6 **reference:** 参考/引用某一资源 **ID**

- 定义：

```
<attr name="leftIcon" format="reference" />
```

- 使用：

```
<app:leftIcon = "@drawable/图片 ID" />
```

1.2.7 混合类型：属性定义时指定多种类型值

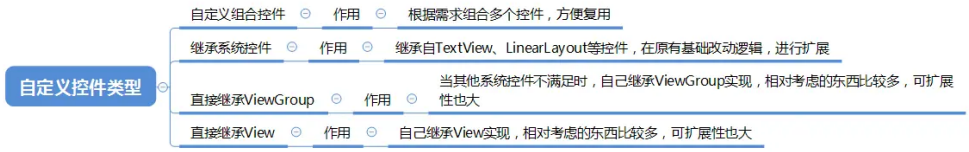
- 属性定义

```
<attr name = "background" format = "reference|color" />
```

- 使用

```
<android:background = "@drawable/图片 ID"  />
<android:background = "#FFFFFF"  />
```

1.3 三、自定义控件类型



1.3.1 自定义组合控件步骤

1. 1. 自定义属性
    - 在 res/values 目录下的 attrs.xml 文件中

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

2. 2. 布局中使用自定义属性
    - 在布局中使用

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

3. 3. view 的构造函数获取自定义属性

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

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

1.3.2 继承系统控件

- 就是继承系统已经提供好给我们的控件例如 TextView、LinearLayout 等, 分为 View 类型或者 ViewGroup 类型的两种。主要根据业务需求进行实现，实现重写的空间也很大主要看需求。
- 比如需求：在文字后面加个颜色背景

- 根据需要一般这种情况下我们是希望可以复用系统的 onMeaseur 和 onLayout 流程. 直接复写 onDraw 方法

```
class Practice02BeforeOnDrawView : AppCompatTextView {
    internal var paint = Paint(Paint.ANTI_ALIAS_FLAG)
    internal var bounds = RectF()
    constructor(context: Context) : super(context) {}
    constructor(context: Context, attrs: AttributeSet?) : super(context, attrs) {}
    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int) : super(context, attrs, defStyleAttr) {}
    init {
        paint.color = Color.parseColor("#FFC107")
    }
    override fun onDraw(canvas: Canvas) {
        // 把下面的绘制代码移到 super.onDraw() 的上面，就可以让原主体内容盖住你的绘制代码了
        // （或者你也可以把 super.onDraw() 移到这段代码的下面）
        val layout = layout
        bounds.left = layout.getLineLeft(1)
        bounds.right = layout.getLineRight(1)
        bounds.top = layout.getLineTop(1).toFloat()
        bounds.bottom = layout.getLineBottom(1).toFloat()
        //绘制方形背景
        canvas.drawRect(bounds, paint)
        super.onDraw(canvas)
    }
}
```

- 这里会涉及到画笔 Paint()、画布 canvas、路径 Path、绘画顺序等的一些知识点，后面再详细说明

1.3.3 直接继承 **View**

- 这种就是类似 TextView 等，不需要去轮询子 View，只需要根据自己的需求重写 onMeasure()、onLayout()、onDraw() 等方法便可以，要注意点就是记得 Padding 等值要记得加入运算

```
private int getCalculateSize(int defaultSize, int measureSpec) {
    int finallSize = defaultSize;
    int mode = MeasureSpec.getMode(measureSpec);
    int size = MeasureSpec.getSize(measureSpec);
    //  根据模式对
    switch (mode) {
        case MeasureSpec.EXACTLY:
            break;
        case MeasureSpec.AT_MOST:
            break;
        case MeasureSpec.UNSPECIFIED:
            break;
    }
    return finallSize;
}
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int width = getCalculateSize(120, widthMeasureSpec);
    int height = getCalculateSize(120, heightMeasureSpec);
    setMeasuredDimension(width, height);
}
@Override
protected void onDraw(Canvas canvas) { // 画一个圆
    // 调用父 View 的 onDraw 函数，因为 View 这个类帮我们实现了一些基本的而绘制功能，比如绘制背景颜色、背景图片等
    super.onDraw(canvas);
    int r = getMeasuredWidth() / 2;
    // 圆心的横坐标为当前的 View 的左边起始位置 + 半径
    int centerX = getLeft() + r;
    // 圆心的纵坐标为当前的 View 的顶部起始位置 + 半径
    int centerY = getTop() + r;
    Paint paint = new Paint();
    paint.setColor(Color.RED);
    canvas.drawCircle(centerX, centerY, r, paint);
}
```

1.3.4 直接继承 **ViewGroup**

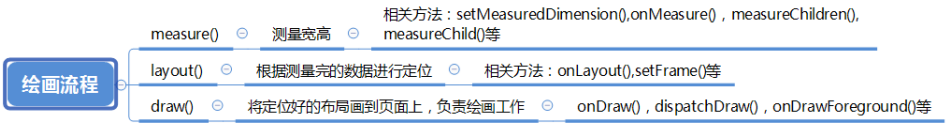
- 类似实现 LinearLayout 等，可以去看那一下 LinearLayout 的实现基本的你可能要重写 onMeasure()、onLayout()、onDraw() 方法, 这块很多问题要处理包括轮训 childView 的测量值以及模式进行大小逻辑计算等，这个篇幅过大后期加多个文章写详细的
- 这里写个简单的需求，模仿 LinearLayout 的垂直布局

```
class CustomViewGroup :ViewGroup{
    constructor(context:Context):super(context)
    constructor(context: Context,attrs:AttributeSet):super(context,attrs){
        // 可获取自定义的属性等
    }
    override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec)
        // 将所有的子 View 进行测量，这会触发每个子 View 的 onMeasure 函数
        measureChildren(widthMeasureSpec, heightMeasureSpec)
        val widthMode = MeasureSpec.getMode(widthMeasureSpec)
        val widthSize = MeasureSpec.getSize(widthMeasureSpec)
        val heightMode = MeasureSpec.getMode(heightMeasureSpec)
        val heightSize = MeasureSpec.getSize(heightMeasureSpec)
        val childCount = childCount
        if (childCount == 0) {
            // 没有子 View 的情况
            setMeasuredDimension(0, 0)
        } else {
            // 如果宽高都是包裹内容
            if (widthMode == MeasureSpec.AT_MOST && heightMode == MeasureSpec.AT_MOST) {
                // 我们将高度设置为所有子 View 的高度相加，宽度设为子 View 中最大的宽度
                val height = getTotalHeight()
                val width = getMaxChildWidth()
                setMeasuredDimension(width, height)
            } else if (heightMode == MeasureSpec.AT_MOST) {
                // 如果只有高度是包裹内容
                // 宽度设置为 ViewGroup 自己的测量宽度，高度设置为所有子 View 的高度总和
                setMeasuredDimension(widthSize, getTotalHeight())
            } else if (widthMode == MeasureSpec.AT_MOST) {// 如果只有宽度是包裹内容
            // 宽度设置为子 View 中宽度最大的值，高度设置为 ViewGroup 自己的测量值
            setMeasuredDimension(getMaxChildWidth(), heightSize)
        }
    }
    // 获取子 View 中宽度最大的值
    private fun getMaxChildWidth(): Int {
        val childCount = childCount
        var maxWidth = 0
        for (i in 0 until childCount) {
            val childView = getChildAt(i)
            if (childView.measuredWidth > maxWidth)
            maxWidth = childView.measuredWidth
        }
        return maxWidth
    }
    // 将所有子 View 的高度相加
    private fun getTotalHeight(): Int {
        val childCount = childCount
        var height = 0
        for (i in 0 until childCount) {
            val childView = getChildAt(i)
            height += childView.measuredHeight
        }
        return height
    }
}
override fun onLayout(changed: Boolean, l: Int, t: Int, r: Int, b: Int) {
    val count = childCount
    var currentHeight = t
    for (i in 0 until count) {
        val child = getChildAt(i)
        val h = child.measuredHeight
        val w = child.measuredWidth
        child.layout(l, currentHeight, l + w, currentHeight + h) // 摆放子 view
        currentHeight += h
```

```
    }
}
```

- 主要两点先 measureChildren() 轮训遍历子 View 获取宽高, 并根据测量模式逻辑计算最后所有的控件的所需宽高，最后 setMeasuredDimension() 保存一下 ### 四、View 的绘制流程相关最基本的三个相关函数 measure() ->layout()->draw()

1.4 四、onMeasure() 相关的知识点



1.4.1 1. MeasureSpec

- MeasureSpec 是 View 的内部类，它封装了一个 View 的尺寸，在 onMeasure() 当中会根据这个 MeasureSpec 的值来确定 View 的宽高。
- MeasureSpec 的数据是 int 类型，有 32 位。高两位表示模式，后面 30 位表示大小 size。
- 则 MeasureSpec = mode+size 三种模式分别为：EXACTLY, AT$_{MOST}$, UNSPECIFIED
  - EXACTLY: （match$_{parent}$ 精确数据值）精确模式，对应的数值就是 MeasureSpec 当中的 size
  - AT$_{MOST}$:(wrap$_{content}$) 最大值模式，View 的尺寸有一个最大值，View 不超过 MeasureSpec 当中的 Size 值
  - UNSPECIFIED:（一般系统使用）无限制模式，View 设置多大就给他多大

```
// 获取测量模式
val widthMode = MeasureSpec.getMode(widthMeasureSpec)
// 获取测量大小
val widthSize = MeasureSpec.getSize(widthMeasureSpec)
// 通过 Mode 和 Size 构造 MeasureSpec
val measureSpec = MeasureSpec.makeMeasureSpec(size, mode);
```

1.4.2 2. View #onMeasure() 源码

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),
                         getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
}
protected int getSuggestedMinimumWidth() {
    return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.getMinimumWidth());
}
protected final void setMeasuredDimension(int measuredWidth, int measuredHeight) {
    boolean optical = isLayoutModeOptical(this);
    if (optical != isLayoutModeOptical(mParent)) {
        Insets insets = getOpticalInsets();
        int opticalWidth  = insets.left + insets.right;
        int opticalHeight = insets.top  + insets.bottom;
        measuredWidth  += optical ? opticalWidth  : -opticalWidth;
        measuredHeight += optical ? opticalHeight : -opticalHeight;
    }
    setMeasuredDimensionRaw(measuredWidth, measuredHeight);
}
```

```
public static int getDefaultSize(int size, int measureSpec) {
    int result = size;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);
    switch (specMode) {
    case MeasureSpec.UNSPECIFIED:
        result = size;
        break;
    case MeasureSpec.AT_MOST:
    case MeasureSpec.EXACTLY:
        result = specSize;
        break;
    }
    return result;
}
private void setMeasuredDimensionRaw(int measuredWidth, int measuredHeight) {
    mMeasuredWidth = measuredWidth;
    mMeasuredHeight = measuredHeight;
    mPrivateFlags |= PFLAG_MEASURED_DIMENSION_SET;
}
```

- setMeasuredDimension(int measuredWidth, int measuredHeight)：用来设置 View 的宽高，在我们自定义 View 保存宽高也会要用到。
- getSuggestedMinimumWidth()：当 View 没有设置背景时，默认大小就是 mMinWidth，这个值对应 Android:minWidth 属性，如果没有设置时默认为 0. 如果有设置背景，则默认大小为 mMinWidth 和 mBackground.getMinimumWidth() 当中的较大值。
- getDefaultSize(int size, int measureSpec)：用来获取 View 默认的宽高，在 getDefaultSize() 中对 MeasureSpec.AT$_{MOST}$,MeasureSpec.EXACTLY 两个的处理是一样的，我们自定义 View 的时候要对两种模式进行处理。

1.4.3 3. ViewGroup 中并没有 measure() 也没有 onMeasure()

- 因为 ViewGroup 除了测量自身的宽高，还需要测量各个子 View 的宽高，不同的布局测量方式不同 (例如 LinearLayout 跟 RelativeLayout 等布局），所以直接交由继承者根据自己的需要去复写。但是里面因为子 View 的测量是相对固定的，所以里面已经提供了基本的 measureChildren() 以及 measureChild() 来帮助我们对子 View 进行测量这个可以看一下我另一篇文章：LinearLayout # onMeasure()LinearLayout onMeasure 源码阅读

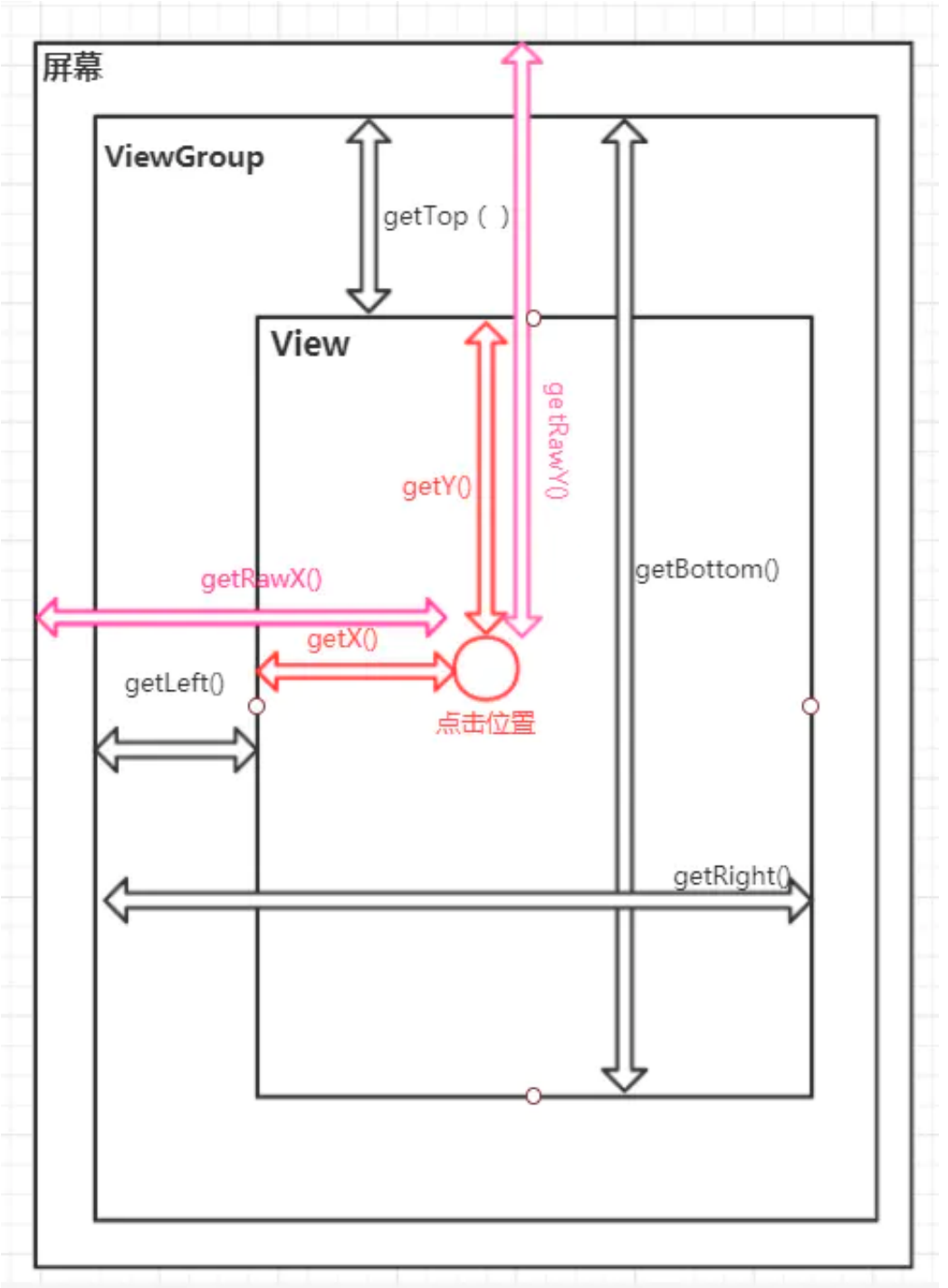
1.5 五、onLayout() 相关

View.java 的 onLayout 方法是空实现: 因为子 View 的位置，是由其父控件的 onLayout 方法来确定的。onLayout(int l, int t, int r, int b) 中的参数 l、t、r、b 都是相对于其父控件的位置。自身的 mLeft, mTop, mRight, mBottom 都是相对于父控件的位置。

1.5.1 1. Android 坐标系

- 安卓屏幕的左上角为坐标原点，向右为 X 轴正向，向下为 Y 轴正向

1.5.2 2. 内部 View 坐标系跟点击坐标



1.5.3 3. 看一下 View#layout(int l, int t, int r, int b) 源码

```
public void layout(int l, int t, int r, int b) {
    if ((mPrivateFlags3 & PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT) != 0) {
        onMeasure(mOldWidthMeasureSpec, mOldHeightMeasureSpec);
        mPrivateFlags3 &= ~PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
```

```
    }
    int oldL = mLeft;
    int oldT = mTop;
    int oldB = mBottom;
    int oldR = mRight;
    boolean changed = isLayoutModeOptical(mParent) ?
        setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);
    if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PFLAG_LAYOUT_REQUIRED) {
        onLayout(changed, l, t, r, b);
        //   .... 省略其它部分
    }
}
private boolean setOpticalFrame(int left, int top, int right, int bottom) {
    Insets parentInsets = mParent instanceof View ?
        ((View) mParent).getOpticalInsets() : Insets.NONE;
    Insets childInsets = getOpticalInsets();
    return setFrame(
        left   + parentInsets.left - childInsets.left,
        top    + parentInsets.top  - childInsets.top,
        right  + parentInsets.left + childInsets.right,
        bottom + parentInsets.top  + childInsets.bottom);
}
protected boolean setFrame(int left, int top, int right, int bottom) {
    boolean changed = false;
    // .... 省略其它部分
    if (mLeft != left || mRight != right || mTop != top || mBottom != bottom) {
        changed = true;
        int drawn = mPrivateFlags & PFLAG_DRAWN;
        int oldWidth = mRight - mLeft;
        int oldHeight = mBottom - mTop;
        int newWidth = right - left;
        int newHeight = bottom - top;
        boolean sizeChanged = (newWidth != oldWidth) || (newHeight != oldHeight);
        invalidate(sizeChanged);
        mLeft = left;
        mTop = top;
        mRight = right;
        mBottom = bottom;
        mRenderNode.setLeftTopRightBottom(mLeft, mTop, mRight, mBottom);
        mPrivateFlags |= PFLAG_HAS_BOUNDS;
        if (sizeChanged)
            sizeChange(newWidth, newHeight, oldWidth, oldHeight);
        if ((mViewFlags & VISIBILITY_MASK) == VISIBLE || mGhostView != null) {
            mPrivateFlags |= PFLAG_DRAWN;
            invalidate(sizeChanged);
            invalidateParentCaches();
        }
        mPrivateFlags |= drawn;
        mBackgroundSizeChanged = true;
        mDefaultFocusHighlightSizeChanged = true;
        if (mForegroundInfo != null)
            mForegroundInfo.mBoundsChanged = true;
        notifySubtreeAccessibilityStateChangedIfNeeded();
    }
    return changed;
}
```

- 四个参数 l、t、r、b 分别代表 View 的左、上、右、下四个边界相对于其父 View 的距离。在调用 onLayout(changed, l, t, r, b); 之前都会调用到 setFrame() 确定 View 在父容器当中的位置，赋值给 mLeft,mTop,mRight,mBottom。在 ViewGroup#onLayout() 跟 View#onLayout() 都是空实现，交给继承者根据自身需求去定位
- 部分零散知识点：
  - getMeasureWidth() 与 getWidth() getMeasureWidth() 返回的是 mMeasuredWidth，而该值是在 setMeasureDimension() 中的 setMeasureDimensionRaw() 中设置的。因此 onMeasure() 后的所有方法都能获取到这个值。getWidth 返回的是 mRight-mLeft，这两个值，是在 layout() 中的 setFrame() 中设置的. getMeasureWidthAndState 中有一句：This should be used during measurement and layout calculations only. Use {@link #getWidth()} to see how wide a view is after layout.

- 总结：只有在测量过程中和布局计算时，才用 getMeasuredWidth()。在 layout 之后，用 getWidth() 来获取宽度

1.6 六、draw() 绘画过程

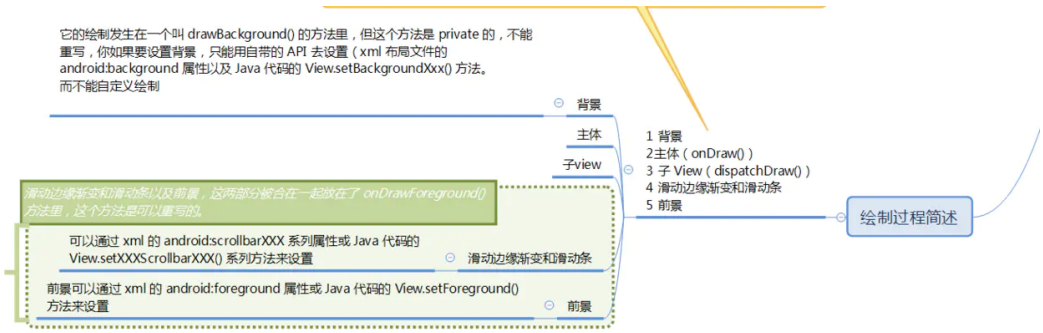
```
/*
         * Draw traversal performs several drawing steps which must be executed
         * in the appropriate order:
         *
         *      1\. Draw the background
         *      2\. If necessary, save the canvas' layers to prepare for fading
         *      3\. Draw view's content
     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

- 上面是 draw() 里面写的绘画顺序。
  - 绘制背景。
  - 如果必要的话, 保存当前 canvas
  - 绘制 View 的内容
  - 绘制子 View
  - 如果必要的话, 绘画边缘重新保存图层
  - 画装饰 (例如滚动条)

1.6.1 1. 看一下 View#draw() 源码的实现

```
public void draw(Canvas canvas) {
    // 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);
        return;
    }
}
```

- 由上面可以看到先调用 drawBackground(canvas) ->onDraw(canvas)->dispatchDraw(canvas)->onDrawForeground(canvas) 越是后面绘画的越是覆盖在最上层。
- drawBackground(canvas): 画背景，不可重写
- onDraw(canvas)：画主体
  - 代码写在 super.onDraw() 前：会被父类的 onDraw 覆盖
  - 代码写在 super.onDraw() 后：不会被父类的 onDraw 覆盖
- dispatchDraw() ：绘制子 View 的方法
  - 代码写在 super.dispatchDraw(canvas) 前：把绘制代码写在 super.dispatchDraw() 的上面，这段绘制就会在 onDraw() 之后、super.dispatchDraw() 之前发生，也就是绘制内容会出现在主体内容和子 View 之间。而这个……其实和重写 onDraw() 并把绘制代码写在 super.onDraw() 之后的做法，效果是一样的。
  - 代码写在 super.dispatchDraw(canvas) 后：只要重写 dispatchDraw()，并在 super.dispatchDraw(的下面写上你的绘制代码，这段绘制代码就会发生在子 View 的绘制之后，从而让绘制内容盖住子 View 了。
- onDrawForeground(canvas)：包含了滑动边缘渐变和滑动条跟前景
  - 一般来说，一个 View（或 ViewGroup）的绘制不会这几项全都包含，但必然逃不出这几项，并且一定会严格遵守这个顺序。例如通常一个 LinearLayout 只有背景和子 View，那么它会先绘制背景再绘制子 View；一个 ImageView 有主体，有可能会再加上一层半透明的前景作为遮罩，那么它的前景也会在主体之后进行绘制。需要注意，前景的支持是在 Android 6.0（也就是 API 23）才加入的；之前其实也有，不过只支持 FrameLayout，而直到 6.0 才把这个支持放进了 View 类里。



1.6.2 2. 注意事项

1. 2.1 在 ViewGroup 的子类中重写除 dispatchDraw() 以外的绘制方法时，可能需要调用 setWillNotDraw(false)：
   - 出于效率的考虑，ViewGroup 默认会绕过 draw() 方法，换而直接执行 dispatchDraw()，以此来简化绘制流程。所以如果你自定义了某个 ViewGroup 的子类（比如 LinearLayout）并且需要在它的除 dispatchDraw() 以外的任何一个绘制方法内绘制内容，你可能会需要调用 View.setWillNotDraw(false) 这行代码来切换到完整的绘制流程（是「可能」而不是「必须」的原因是，有些 ViewGroup 是已经调用过 setWillNotDraw(false) 了的，例如 ScrollView）。
2. 2.2 在重写的方法有多个选择时，优先选择 onDraw()

- 一段绘制代码写在不同的绘制方法中效果是一样的，这时你可以选一个自己喜欢或者习惯的绘制方法来重写。但有一个例外：如果绘制代码既可以写在 onDraw() 里，也可以写在其他绘制方法里，那么优先写在 onDraw() ，因为 Android 有相关的优化，可以在不需要重绘的时候自动跳过 onDraw() 的重复执行，以提升开发效率。享受这种优化的只有 onDraw() 一个方法。

1.7 七、在 **Activity** 中获取 **View** 的宽高的几种方式

- Activity 获取 view 的宽高，在 onCreate , onResume 等方法中获取到的都是 0，因为 View 的测量过程并不是和 Activity 的声明周期同步执行的

1.7.1 1. view.post

- post 可以将一个 runnable 投递到消息队列的尾部，然后等待 Looper 调用此 runnable 的时候，View 也已经初始化好了

```
view.post(new Runnable() {
    @Override
    public void run() {
        int width = view.getMeasuredWidth();
        int height = view.getMeasuredHeight();
    }
});
```

1.7.2 2. ViewTreeObserver

- 使用 addOnGlobalLayoutListener 接口，当 view 树的状态发生改变或者 View 树内部的 view 的可见性发生改变时，onGlobalLayout() 都会被调用，需要注意的是，onGlobalLayout 方法可能被调用多次，代码如下：

```
view.getViewTreeObserver().addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener() {
            @Override
            public void onGlobalLayout() {
                view.getViewTreeObserver().removeOnGlobalLayoutListener(this);
                int width = view.getMeasuredWidth();
                int height = view.getMeasuredHeight();
            }
        });
```

1.7.3 3. onWindowFocusChanged

- 这个方法的含义是 View 已经初始化完毕了，宽高已经准备好了，需要注意的就是这个方法可能会调用多次，在 Activity onResume 和 onPause 的时候都会调用，也会有多次调用的情况

```
@Override
public void onWindowFocusChanged(boolean hasWindowFocus) {
    super.onWindowFocusChanged(hasWindowFocus);
    if (hasWindowFocus){
        int width = view.getMeasuredWidth();
        int height = view.getMeasuredHeight();
    }
}
```

1.8 **View** 工作流程

- 通过 SetContentView()，调用到 PhoneWindow ，后实例 DecorView ，通过 LoadXmlResourceParser() 进行 IO 操作解析 xml 文件通过反射创建出 View，并将 View 绘制在 DecorView

上，这里的绘制则交给了 ViewRootImpl 来完成，通过 performTraversals() 触发绘制流程，performMeasure 方法获取 View 的尺寸，performLayout 方法获取 View 的位置，然后通过 performDraw 方法遍历 View 进行绘制。

1.9 事件分发

- 一个 MotionEvent 产生后，按 Activity -> Window -> DecorView（ViewGroup）-> View 顺序传递，View 传递过程就是事件分发，因为开发过程中存在事件冲突，所以需要熟悉流程：
  - dispatchTouchEvent：用于分发事件，只要接受到点击事件就会被调用，返回结果表示是否消耗了当前事件
  - onInterceptTouchEvent：用于判断是否拦截事件（只有 ViewGroup 中存在），当 ViewGroup 确定要拦截事件后，该事件序列都不会再触发调用此 ViewGroup 的 onIntercept
  - onTouchEvent：用于处理事件，返回结果表示是否处理了当前事件，未处理则传递给父容器处理。（事件顺序是：OnTouchListener -> OnTouchEvent -> OnClick）

1.10 自定义 View!!

准备自定义 View 方面的面试最简单的方法：
就是自己动手实现几个 View（由简单到复杂）；分析一些热门 App 中的自定义 View 的效果是怎么实现的；阿里面试官：自定义 View 跟绘制流程相关知识点？（标准参考解答，值得收藏）

- https://www.cnblogs.com/Android-Alvin/p/12297933.html

2 View 相关

2.1 View 工作流程

- 通过 SetContentView()，调用到 PhoneWindow，后实例 DecorView，通过 LoadXmlResourceParser() 进行 IO 操作解析 xml 文件通过反射创建出 View，并将 View 绘制在 DecorView 上，这里的绘制则交给了 ViewRootImpl 来完成，通过 performTraversals() 触发绘制流程，performMeasure 方法获取 View 的尺寸，performLayout 方法获取 View 的位置，然后通过 performDraw 方法遍历 View 进行绘制。

2.2 事件分发

- 一个 MotionEvent 产生后，按 Activity -> Window -> DecorView（ViewGroup）-> View 顺序传递，View 传递过程就是事件分发，因为开发过程中存在事件冲突，所以需要熟悉流程：
  - dispatchTouchEvent：用于分发事件，只要接受到点击事件就会被调用，返回结果表示是否消耗了当前事件
  - onInterceptTouchEvent：用于判断是否拦截事件（只有 ViewGroup 中存在），当 ViewGroup 确定要拦截事件后，该事件序列都不会再触发调用此 ViewGroup 的 onIntercept
  - onTouchEvent：用于处理事件，返回结果表示是否处理了当前事件，未处理则传递给父容器处理。（事件顺序是：OnTouchListener -> OnTouchEvent -> OnClick）

2.3 自定义 View!!

准备自定义 View 方面的面试最简单的方法：就是自己动手实现几个 View（由简单到复杂）；分析一些热门 App 中的自定义 View 的效果是怎么实现的；阿里面试官：自定义 View 跟绘制流程相关知识点？（标准参考解答，值得收藏）

- https://www.cnblogs.com/Android-Alvin/p/12297933.html