转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81503742

思维导图之后补全。。

View工作原理-思维导图版

版本：2018/4/25-1

1-ViewRoot

1、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。

1.1三大绘制流程

2、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

3、View三大流程的作用

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

- `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 ! = 实际值`

2-DecorView

4、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
4. 获取content： `ViewGroup content = findViewById(R.android.id.content)`
5. 获取设置的View： `content.getChidlAt(0)`

2.1ViewRoot和DecorView的关系

5、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);
```

6、 `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

3-MeasureSpec

5、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)

6、MeasureSpec要点解析

> 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

3.1.SpecMode

7、测量模式SpecMode的类型

> 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)

3.2.MeasureSpec和LayoutParams

8、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

3.3.MeasureSpec的创建规则

9、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。

4-View的工作流程(三大流程)

10、View的工作流程以及具体的功能

> 1. measure：测量——确定View的测量宽/高

> 2. layout：布局——确定View的最终宽/高和四个顶点的位置
> 3. draw：绘制——将View绘制到屏幕上

4.2-measure过程

View

12、View的measure过程及要点

> 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

13、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

14、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
        case MeasureSpec.AT_MOST:
        case MeasureSpec.EXACTLY:
            result = specSize;
            break;
        }
        return result;
    }
```

15、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)

wrap_content和match_parent效果一致的原因

16、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。

自定义View：需要重写onMeasure方法

17、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

18、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，然后调用子元素View的measure方法进行测量

19、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，需要指定默认的宽/高

LinearLayout的onMeasure()

20、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

4.3-layout过程

View

25、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

26、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。

LinearLayout

27、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)

测量宽高和最终宽高的区别

28、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

4.4-draw过程

29、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)

30、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。

4.5-获取View的宽高

如何获取测量宽高

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

Activity启动时获取宽高的四种方法

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

21、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();
        }
    }
```

22、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();
            }
        });
```

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

```
    val observer = imageView.viewTreeObserver
        //1. 使用ViewTreeObserver的接口，可以再View树状态改变或者View树内部View的可见性改变时，onGlobalLayout
    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
            }
        })
```

24、view.measure()获得View的宽/高(Kotlin)

> 1. mathc_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
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
```

4.1.Activity启动到加载ViewRoot的流程

11、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。

5-自定义View

31、自定义View的分类

分类	实现方法	备注
1.继承View	重写onDraw()方法	需要支持 `wrap_content` 和 `padding`
2.继承ViewGroup	需要处理ViewGroup的 测量 和 布局	需要处理子元素的 测量 和 布局 过程
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	___

32、自定义View要点

> 1. View需要支持wrap_content
> 2. View需要支持padding
> 3. 尽量不要再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

5.1-直接继承View

33、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, defStyleAti
    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

34、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>
```

5.2-直接继承ViewGroup

35、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margin和padding)
> 2. 必须实现onLayout方法，并且处理margin和padding属性
> 3. 要支持margin功能，需要重写LayoutParmas相关方法

```
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, defStyleAt
    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
                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 + childLa
                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 + childLa
                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. 要支持Margin功能，必须要重写方法，并实现自己LayoutParams
     */
    override fun generateDefaultLayoutParams() = MyLayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.WR
    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)
    }
}
```

滑动冲突

36、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。