01.View布局介绍
02. Layout流程分析
2.1 视图树内部Layout
2.2 视图树内部OnLayout流程
2.3 View的布局流程图
05.FrameLayout中onLayout解析
其他介绍
01.关于我的博客

提问：

1. View和ViewGroup的onLayout关系是什么样的?
2. layout和onlayout关系是什么样的?

01.View布局介绍

- 请问布局的本质是什么，**View是一块矩形区域，布局就是确定这个矩形区域四个顶点的坐标，也就是View中的mLeft,mTop,mBottom,mRight变量**，在我们确定矩形区域的位置后，View才能在绘制（onDraw）的时候知道应该在哪个矩形区域内绘制控件。
- 布局分为两个方法，一个是layout，一个是onlayout。其中，Layout方法是用来设置坐标，确定这个矩形区域位置的方法。其在内部 为这几个成员变量赋值，那onLayout()的作用是什么呢?
- **onLayout()都是由ViewGroup类别实现的，他的作用就是确定ViewGroup中每个子View的位置。onLayout()方法会遍历容器中所有的子控件，然后计算他们左上右下的坐标值，并调用child.layout()方法为子控件设置坐标**；然后由于子View的layout()方法中又调用了onLayout()方法，如果子控件child也是一个容器，就会继续为该子View的子控件计算坐标，如果child不是容器，onLayout()方法将什么也不做。

02. Layout流程分析

- 它的整体流程和Measure一样，所以让我们直接进入到它的View树中的流程吧。

2.1 视图树内部Layout

- 首先是ViewGroup的layout()源码如下所示
  - 从源码中可以看出实际上调用的还是View的layout()方法。

```
@Override
public final void layout(int l, int t, int r, int b) {
    if (!mSuppressLayout && (mTransition == null ||
!mTransition.isChangingLayout())) {
        if (mTransition != null) {
            mTransition.layoutChange(this);
        }
        //内部实际上是调用View的layout()
        super.layout(l, t, r, b);
    } else {
        // record the fact that we noop'd it; request layout when transition
finishes
        mLayoutCalledWhileSuppressed = true;
    }
}
```

- 那么View的layout()：
  - 获取上下左右四个顶点坐标系的值并最终通过setFrame进行设置来确定自身位置。接着调用OnLayout进行子View的布局流程。
  - 对比测量过程，**ViewGroup先在layout()中确定自己的布局，然后在onLayout()方法中再调用子View的layout()方法，让子View布局。在Measure过程中，ViewGroup一般是先测量子View的大小，然后再确定自身的大小。**

```
public void layout(int l, int t, int r, int b) {

    // 当前视图的四个顶点
    int oldL = mLeft;
    int oldT = mTop;
    int oldB = mBottom;
    int oldR = mRight;

    // 确定视图的四个顶点
    boolean changed = isLayoutModeOptical(mParent) ?
            setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);

    // 调用 onLayout()测量子视图
    if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) ==
PFLAG_LAYOUT_REQUIRED) {
        onLayout(changed, l, t, r, b);
    }
  ...

}
```

2.2 视图树内部OnLayout流程

- View的onLayout()：查看源码如下所示
  - 由于继承View的自定义控件不包含子控件，所以View的onLayout()方法是空实现。

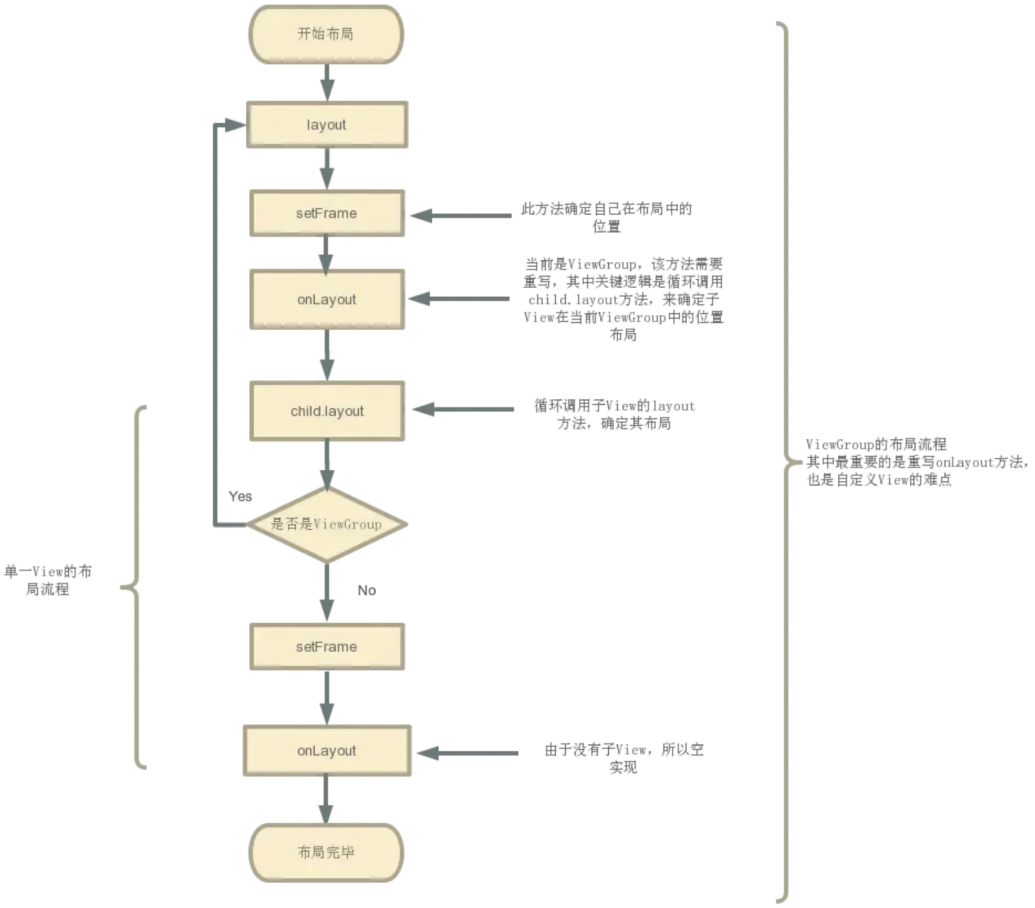
```
/**
  * onLayout（）
  * 注：对于单一View的laytou过程
  *     a. 由于单一View是没有子View的，故onLayout（）是一个空实现
  *     b. 由于在layout（）中已经对自身View进行了位置计算，所以单一View的layout过程在
layout（）后就已完成了
  */
protected void onLayout(boolean changed, int left, int top, int right, int
bottom) {
}
```

- ViewGroup的onLayout()： 查看源码如下所示
  - ViewGroup中的onLayout方法是抽象方法，需要我们自己去实现，它需要去遍历子View，根据子View的宽高，布局参数，以及自身的布局方向，计算出子View的四个顶点的坐标，然后调用子View的Layout为其确定布局。

```
/**
* 作用：计算该ViewGroup包含所有的子View在父容器的位置（）
* 注：
*     a. 定义为抽象方法，需重写，因：子View的确定位置与具体布局有关，所以onLayout（）在
ViewGroup没有实现
*     b. 在自定义ViewGroup时必须复写onLayout（）！！！！！
*     c. 复写原理：遍历子View 、计算当前子View的四个位置值 & 确定自身子View的位置（调用子
View layout（））
* */
@Override
protected abstract void onLayout(boolean changed,int l, int t, int r, int b);
```

2.3 View的布局流程图

View的布局流程：假如从一个ViewGroup开始。

- 

05.FrameLayout中onLayout解析

- FrameLayout 的 onLayout 方法比较简单，这里直接通过注释的形式进行说明。

```
@Override
//onLayout() 方法只是简单地调用了 layoutChildren()，实际的布局逻辑都在
layoutChildren() 中。
protected void onLayout(boolean changed, int left, int top, int right, int
bottom) {
    layoutChildren(left, top, right, bottom, false /* no force left gravity
*/);
}

void layoutChildren(int left, int top, int right, int bottom,
                            boolean forceLeftGravity) {
    final int count = getChildCount();

//计算父view可使用的上下左右边界
    final int parentLeft = getPaddingLeftWithForeground();
    final int parentRight = right - left - getPaddingRightWithForeground();
    final int parentTop = getPaddingTopWithForeground();
    final int parentBottom = bottom - top - getPaddingBottomWithForeground();

// 遍历每一个 view，设置每一个 view 的位置。
    for (int i = 0; i < count; i++) {
        final View child = getChildAt(i);
        // 只操作可见 view
        if (child.getVisibility() != GONE) {
```

```
            //获取子视图的LayoutParams及其测量好的宽高
            final LayoutParams lp = (LayoutParams) child.getLayoutParams();
            // width 和 height 分别是 Measure 过后的 宽和高
            final int width = child.getMeasuredWidth();
            final int height = child.getMeasuredHeight();

            int childLeft;
            int childTop;

            //从上面获取到的子视图的LayoutParams中解析其gravity属性，其用于 控制视图内容在其自身范围内的对齐方式。简单来说，它决定了视图内的内容应该靠左、靠右、居中还是其他对齐方式。
            int gravity = lp.gravity;
            if (gravity == -1) {
                gravity = DEFAULT_CHILD_GRAVITY;
            }
            /*
                获取当前布局的方向（从左到右或从右到左）。
                    View.LAYOUT_DIRECTION_LTR（0）：从左到右（默认方向）。
                    View.LAYOUT_DIRECTION_RTL（1）：从右到左（阿拉伯语、希伯来语等语言使用的布局方式）。
                将 gravity 属性转换为绝对方向。
                    在布局方向为 RTL（从右到左）时，这一步非常重要，它可以将相对的 gravity（如 LEFT 和 RIGHT）翻译为绝对的方向。
                提取 gravity 属性中的垂直对齐方式（与水平对齐分开处理）。
                    gravity & Gravity.VERTICAL_GRAVITY_MASK 的作用是屏蔽掉水平方向上的标志位，仅保留垂直方向的对齐规则。
            */
            final int layoutDirection = getLayoutDirection();
            final int absoluteGravity = Gravity.getAbsoluteGravity(gravity, layoutDirection);
            final int verticalGravity = gravity & Gravity.VERTICAL_GRAVITY_MASK;

//根据 gravity 计算 left 和 top 坐标
            //水平对齐规则
            switch (absoluteGravity & Gravity.HORIZONTAL_GRAVITY_MASK) {
                case Gravity.CENTER_HORIZONTAL:
                    // gravity 是水平居中的情况
                    // 左坐标的计算可以分为两部分
                    // 1. 可使用的父 view 的左边界范围 + 放置view的中间位置(父view可用范围 减去 view 宽度后的一半)
                    // 2. 移除右 margin 加上 左margin
                    childLeft = parentLeft + (parentRight - parentLeft - width) / 2 +
                    lp.leftMargin - lp.rightMargin;
                    break;
                case Gravity.RIGHT:
                    // 这里主要考虑的是强制从左排列，在开发者选项中可以进行设置。
                    // 这里就先不讨论这个。
                    if (!forceLeftGravity) {
                        childLeft = parentRight - width - lp.rightMargin;
                        break;
                    }
                case Gravity.LEFT:
                default:
```

```
                    // 默认情况，加上左 margin 就行。
                    childLeft = parentLeft + lp.leftMargin;
                }
//垂直对齐规则
                switch (verticalGravity) {
                    case Gravity.TOP:
                        childTop = parentTop + lp.topMargin;
                        break;
                    case Gravity.CENTER_VERTICAL:
                        // 垂直居中的情况，与上面类似，也不重复了。
                        childTop = parentTop + (parentBottom - parentTop -
height) / 2 +
                        lp.topMargin - lp.bottomMargin;
                        break;
                    case Gravity.BOTTOM:
                        childTop = parentBottom - height - lp.bottomMargin;
                        break;
                    default:
                        childTop = parentTop + lp.topMargin;
                }
                // 最重要的地方，将计算得出的四个位置作为参数，设置进去。
                child.layout(childLeft, childTop, childLeft + width, childTop +
height);
            }
        }
}
```

其他介绍

01.关于我的博客

- csdn：http://my.csdn.net/qq_35829566
- 掘金：https://juejin.im/user/499639464759898
- github：https://github.com/jjjjjjava
- 简书：http://www.jianshu.com/u/92a2412be53e
- 邮箱：[934137388@qq.com]