HenCoder Plus 第 13 课 讲义

自定义尺寸和内部布局、手写 TagLayout

布局过程

- 确定每个 View 的位置和尺寸
- 作用：为绘制和触摸范围做支持
  - 绘制：知道往哪里绘制
  - 触摸反馈：知道用户点的是哪里

流程

- 从整体看：
  - 测量流程：从根 View 递归调用每一级子 View 的 measure() 方法，对它们进行测量
  - 布局流程：从根 View 递归调用每一级子 View 的 layout() 方法，把测量过程得出的子 View 的位置和尺寸传给子 View，子 View 保存
  - 为什么要分两个流程?
- 从个体看，对于每个 View：
  1. 运行前，开发者在 xml 文件里写入对 View 的布局要求 layout_xxx
  2. 父 View 在自己的 onMeasure() 中，根据开发者在 xml 中写的对子 View 的要求，和自己的可用空间，得出对子 View 的具体尺寸要求
  3. 子 View 在自己的 onMeasure() 中，根据自己的特性算出自己的期望尺寸
     - 如果是 ViewGroup，还会在这里调用每个子 View 的 measure() 进行测量
  4. 父 View 在子 View 计算出期望尺寸后，得出子 View 的实际尺寸和位置
  5. 子 View 在自己的 layout() 方法中，将父 View 传进来的自己的实际尺寸和位置保存
     - 如果是 ViewGroup，还会在 onLayout() 里调用每个字 View 的 layout() 把它们的尺寸位置传给它们

具体开发

- 继承已有的 View，简单改写它们的尺寸：重写 onMeasure()：SquareImageView
  1. 重写 onMeasure()
  2. 用 getMeasuredWidth() 和 getMeasuredSize() 获取到测量出的尺寸
  3. 计算出最终要的尺寸
  4. 用 setMeasuredDimension(width, height) 把结果保存

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);

    int width = getMeasuredWidth();
    int height = getMeasuredHeight();
    int size = Math.min(width, height);

    setMeasuredDimension(size, size);
}
```

- 对自定义 View 完全进行自定义尺寸计算：重写 onMeasure()：CircleView
    1. 重写 onMeasure()
    2. 计算出自己的尺寸
    3. 用 resolveSize() 或者 resolveSizeAndState() 修正结果
        - resolveSize() / resolveSizeAndState() 内部实现（一定读一下代码，这个极少需要自己写，但面试时很多时候会考）：
            - 首先用 MeasureSpec.getMode(measureSpec) 和 MeasureSpec.getSize(measureSpec) 取出父 对自己的尺寸限制类型和具体限制尺寸；
            - 如果 measure spec 的 mode 是 EXACTLY，表示父 View 对子 View 的尺寸做出了精确限制，所以就放弃计算出的 size，直接选用 measure spec 的 size；
            - 如果 measure spec 的 mode 是 AT_MOST，表示父 View 对子 View 的尺寸只限制了上限，需要看情况：
                - 如果计算出的 size 不大于 spec 中限制的 size，表示尺寸没有超出限制，所以选用计算出的 size；
                - 而如果计算出的 size 大于 spec 中限制的 size，表示尺寸超限了，所以选用 spec 的 size，并且在 resolveSizeAndState() 中会添加标志 MEASURED_STATE_TOO_SMALL（这个标志可以辅助父 View 做测量和布局的计算；
            - 如果 measure spec 的 mode 是 UNSPECIFIED，表示父 View 对子 View 没有任何尺寸限制，所以直接选用计算出的 size，忽略 spec 中的 size。
    4. 使用 setMeasuredDimension(width, height) 保存结果

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    int width = (int) ((PADDING + RADIUS) * 2);
    int height = (int) ((PADDING + RADIUS) * 2);

    setMeasuredDimension(resolveSizeAndState(width, widthMeasureSpec,
0),
            resolveSizeAndState(height, heightMeasureSpec, 0));
}
```

- 自定义 Layout：重写 onMeasure() 和 onLayout()：TagLayout
    1. 重写 onMeasure()

1. 遍历每个子 View，用 measureChildWidthMargins() 测量子 View
    - 需要重写 generateLayoutParams() 并返回 MarginLayoutParams 才能使用 measureChildWithMargins() 方法
    - 有些子 View 可能需要重新测量（比如换行处）
    - 测量完成后，得出子 View 的实际位置和尺寸，并暂时保存

```
protected void onMeasure(int widthMeasureSpec, int
heightMeasureSpec) {
    for (int i = 0; i < getChildCount(); i++) {
        View child = getChildAt(i);
        Rect childBounds = childrenBounds[i];
        // 测量子 View
        measureChildWithMargins(child, widthMeasureSpec,
widthUsed,
                heightMeasureSpec, heightUsed);
        // 保存子 View 的位置和尺寸
        childBounds.set(childlLeft, childTop, childLeft
                + child.getMeasuredWidth(), chiltTop
                + child.getMeasuredHeight());
        ......
    }

    // 计算自己的尺寸，并保存
    int width = ...;
    int height = ...;
    setMeasuredDimension(resolveSizeAndState(width,
widthMeasureSpec, 0),
            resolveSizeAndState(height,  heightMeasureSpec, 0));
}
```

    - measureChildWidthMargins() 的内部实现（一定读一下代码，这个极少需要自己写，但面试时很多时候会考）：
        - 通过 getChildMeasureSpec(int spec, int padding, int childDimension) 方法计算出子 View 的 widthMeasureSpec 和 heightMeasureSpec，然后调用 child.measure() 方法来让子 View 自我测量；

```
// ViewGroup.measureChildWithMargins() 源码
protected void measureChildWithMargins(View child,
    int parentWidthMeasureSpec, int widthUsed,
        int parentHeightMeasureSpec, int heightUsed) {
    final MarginLayoutParams lp =
        (MarginLayoutParams) child.getLayoutParams();

    final int childWidthMeasureSpec =
        getChildMeasureSpec(parentWidthMeasureSpec,
        mPaddingLeft + mPaddingRight + lp.leftMargin
        + lp.rightMargin + widthUsed, lp.width);
```

```
        final int childHeightMeasureSpec =
            getChildMeasureSpec(parentHeightMeasureSpec,
            mPaddingTop + mPaddingBottom + lp.topMargin
            + lp.bottomMargin + heightUsed, lp.height);

        child.measure(childWidthMeasureSpec,
    childHeightMeasureSpec);
    }
```

getChildMeasureSpec(int spec, int padding, int childDimension) 方法的内部实现是，结合开发者设置的 LayoutParams 中的 width 和 height 与父 View 自己的剩余可用空间，综合得出子 View 的尺寸限制，并使用 MeasureSpec.makeMeasureSpec(size, mode) 来求得结果：

```
// ViewGroup.getChildMeasureSpec() 源码
public static int getChildMeasureSpec(int spec, int padding,
        int childDimension) {
    int specMode = MeasureSpec.getMode(spec);
    int specSize = MeasureSpec.getSize(spec);

    int size = Math.max(0, specSize - padding);

    int resultSize = 0;
    int resultMode = 0;

    switch (specMode) {
    // Parent has imposed an exact size on us
    case MeasureSpec.EXACTLY:
        if (childDimension >= 0) {
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension ==
    LayoutParams.MATCH_PARENT) {
            // Child wants to be our size. So be it.
            resultSize = size;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension ==
    LayoutParams.WRAP_CONTENT) {
            // Child wants to determine its own size. It
    can't be
            // bigger than us.
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        }
        break;

    // Parent has imposed a maximum size on us
    case MeasureSpec.AT_MOST:
        if (childDimension >= 0) {
```

```
            // Child wants a specific size... so be it
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension ==
LayoutParams.MATCH_PARENT) {
            // Child wants to be our size, but our size is
not fixed.
            // Constrain child to not be bigger than us.
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        } else if (childDimension ==
LayoutParams.WRAP_CONTENT) {
            // Child wants to determine its own size. It
can't be
            // bigger than us.
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        }
        break;

    // Parent asked to see how big we want to be
    case MeasureSpec.UNSPECIFIED:
        if (childDimension >= 0) {
            // Child wants a specific size... let him have
it
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension ==
LayoutParams.MATCH_PARENT) {
            // Child wants to be our size... find out how
big it should
            // be
            resultSize = View.sUseZeroUnspecifiedMeasureSpec
? 0 : size;
            resultMode = MeasureSpec.UNSPECIFIED;
        } else if (childDimension ==
LayoutParams.WRAP_CONTENT) {
            // Child wants to determine its own size....
find out how
            // big it should be
            resultSize = View.sUseZeroUnspecifiedMeasureSpec
? 0 : size;
            resultMode = MeasureSpec.UNSPECIFIED;
        }
        break;
    }
    //noinspection ResourceType
    return MeasureSpec.makeMeasureSpec(resultSize,
resultMode);
```

```
}
```

> 注意：源码中的分类方式是先比较自己的 MeasureSpec 中的 mode，再比较开发者设置的 layout_width 和 layout_height，而我给出的判断方式（下面的这几段内容）是先比较 layout_width 和 layout_height，再比较自己 MeasureSpec 中的 mode。两种分类方法都能得出正确的结果，但源码中的分类方法在逻辑上可能不够直观，如果你读源码理解困难，可以尝试用我上面的这种方法来理解。

1. 如果开发者写了具体值（例如 layout_width="24dp"），就不用再考虑父 View 的剩余空间了，直接用 LayoutParams.width / height 来作为子 View 的限制 size，而限制 mode 为 EXACTLY（为什么？课堂上说过，因为软件的直接开发者——即 xml 布局文件的编写者——的意见最重要，发生冲突的时候应该以开发者的意见为准。换个角度说，如果真的由于冲突导致界面不正确，开发者可以通过修改 xml 文件来解决啊，所以开发者的意见是第一位，但你如果设计成冲突时开发者的意见不在第一位，就会导致软件的可配置性严重降低）；
2. 如果开发者写的是 MATCH_PARENT，即要求填满父控件的可用空间，那么由于自己的可用空间和自己的两个 MeasureSpec 有关，所以需要根据自己的 widthMeasureSpec 或 heightMeasureSpec 中的 mode 来分情况判断：
    1. 如果自己的 spec 中的 mode 是 EXACTLY 或者 AT_MOST，说明自己的尺寸有上限，那么把 spec 中的 size 减去自己的已用宽度或高度，就是自己可以给子 View 的 size；至于 mode，就用 EXACTLY（注意：就算自己的 mode 是 AT_MOST，传给子 View 的也是 EXACTLY，想不通的话好好琢磨一下）；
    2. 如果自己的 spec 中的 mode 是 UNSPECIFIED，说明自己的尺寸没有上限，那么让子 View 填满自己的可用空间就无从说起，因此选用退让方案：给子 View 限制的 mode 就设置为 UNSPECIFIED，size 写 0 就好；
3. 如果开发者写的是 WRAP_CONTENT，即要求子 View 在**不超限制的前提下**自我测量，那么同样由于自己的可用空间和自己的两个 MeasureSpec 有关，所以也需要根据自己的 widthMeasureSpec 和 heightMeasureSpec 中的 mode 来分情况判断：
    1. 如果自己的 spec 中的 mode 是 EXACTLY 或者 AT_MOST，说明自己的尺寸有上限，那么把 spec 中的 size 减去自己的已用宽度或高度，就是自己可以给子 View 的尺寸上限；至于 mode，就用 AT_MOST（注意，就算自己的 mode 是 EXACTLY，传给子 View 的也是 AT_MOST，想不通的话好好琢磨一下；
    2. 如果自己的 spec 中的 mode 是 UNSPECIFIED，说明自己的尺寸没有上限，那么也就不必限制子 View 的上限，因此给子 View 限制的 mode 就设置为 UNSPECIFIED，size 写 0 就好。

2. 测量出所有子 View 的位置和尺寸后，计算出自己的尺寸，并用 setMeasuredDimension(width, height) 保存

2. 重写 onLayout()
    - 遍历每个子 View，调用它们的 layout() 方法来将位置和尺寸传给它们

```
protected void onLayout(boolean changed, int l, int t, int r,
int b) {
    for (int i = 0; i < getChildCount(); i++) {
        View child = getChildAt(i);
        Rect childBounds = childrenBounds[i];
        // 将每个子 View 的位置和尺寸传递给它
        child.layout(childBounds.left, childBounds.top,
                childBounds.right, childBounds.bottom);
    }
}
```