自定义View之Draw

> 前言

查看Android总结专题

自定义View总结：

- View基础
- measure方法
- layout方法
- draw方法
- Path类
- Canvas类

1. draw过程详解

类似`measure`过程、`layout`过程，`draw`过程根据`View`的类型分为2种情况：

View类型	Draw过程
单一View	仅绘制视图View本身
ViewGroup (含子View)	除了绘制自身View外，还需绘制其所有子View

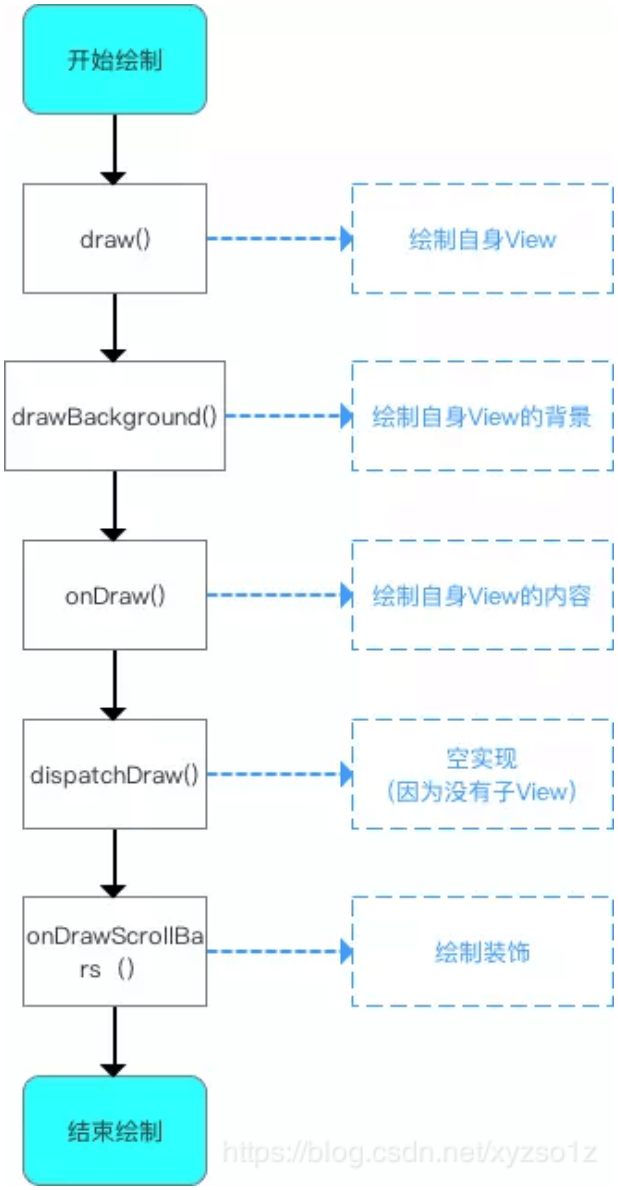
接下来，详细分析这2种情况下单`draw`过程

3.1 单一View的draw过程

- 原理（步骤）

1. View绘制自身（含背景、内容）；
2. 绘制装饰（滚动指示器、滚动条和前景）

- 具体流程



下面我将一个个方法进行详细分析：`draw`过程的入口=`draw()`

```
/**
  * 源码分析：draw（）
  * 作用：根据给定的 Canvas 自动渲染 View（包括其所有子 View）。
  * 绘制过程：
  *   1. 绘制view背景
  *   2. 绘制view内容
  *   3. 绘制子View
  *   4. 绘制装饰（渐变框，滑动条等等）
  * 注：
  *    a. 在调用该方法之前必须要完成 layout 过程
  *    b. 所有的视图最终都是调用 View 的 draw （）绘制视图（ ViewGroup 没有复写此方法）
  *    c. 在自定义View时，不应该复写该方法，而是复写 onDraw(Canvas) 方法进行绘制
  *    d. 若自定义的视图确实要复写该方法，那么需先调用 super.draw(canvas)完成系统的
  *       绘制，然后再进行自定义的绘制
  */
  public void draw(Canvas canvas) {

      // 仅贴出关键代码
```

```
    ...// 仅贴出关键代码

    int saveCount;

    // 步骤1： 绘制本身View背景
        if (!dirtyOpaque) {
            drawBackground(canvas);
        }

    // 若有必要，则保存图层（还有一个复原图层）
    // 优化技巧：当不需绘制 Layer 时，“保存图层“和“复原图层“这两步会跳过
    // 因此在绘制时，节省 Layer 可以提高绘制效率
    final int viewFlags = mViewFlags;
    if (!verticalEdges && !horizontalEdges) {

    // 步骤2：绘制本身View内容
        if (!dirtyOpaque)
            onDraw(canvas);
        // View 中：默认为空实现，需复写
        // ViewGroup中：需复写

    // 步骤3：绘制子View
    // 由于单一View无子View，故View 中：默认为空实现
    // ViewGroup中：系统已经复写好对其子视图进行绘制我们不需要复写
        dispatchDraw(canvas);

    // 步骤4：绘制装饰，如滑动条、前景色等等
        onDrawScrollBars(canvas);

        return;
    }
    ...
}
```

下面，我们继续分析在`draw()`中4个步骤调用的`drawBackground()`、`onDraw()`、`onDrawScrollBars()`

```

/**
  * 步骤1：drawBackground(canvas)
  * 作用：绘制View本身的背景
  */
  private void drawBackground(Canvas canvas) {
        // 获取背景 drawable
        final Drawable background = mBackground;
        if (background == null) {
            return;
        }
        // 根据在 layout 过程中获取的 View 的位置参数，来设置背景的边界
        setBackgroundBounds();

```

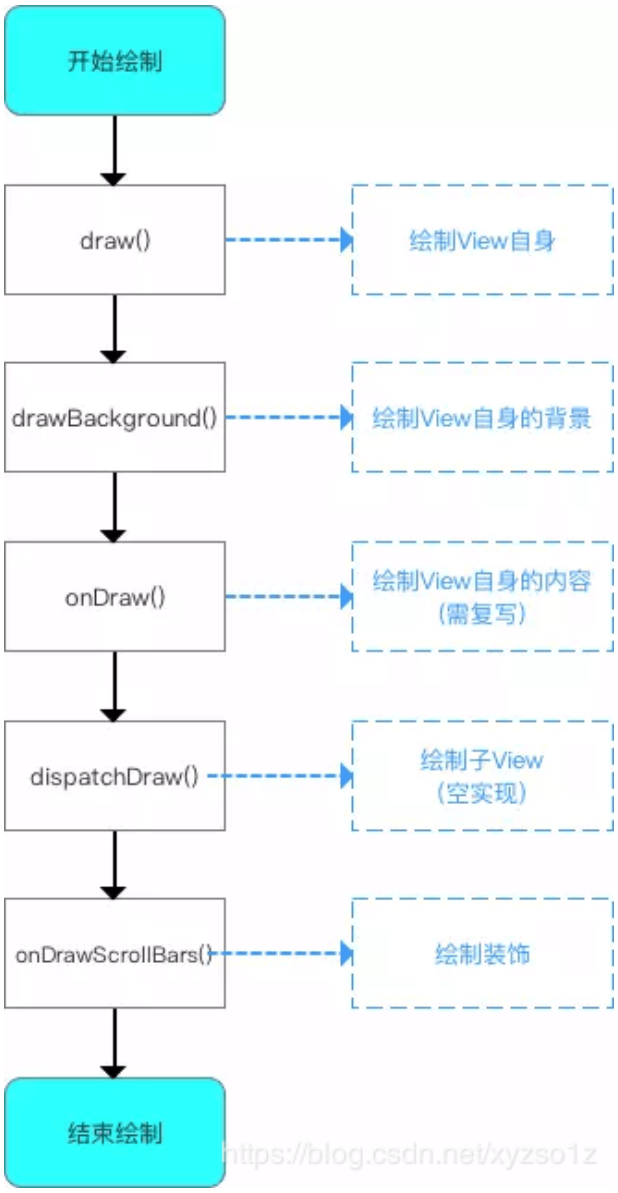
```
        .....

        // 获取 mScrollX 和 mScrollY值
        final int scrollX = mScrollX;
        final int scrollY = mScrollY;
        if ((scrollX | scrollY) == 0) {
            background.draw(canvas);
        } else {
            // 若 mScrollX 和 mScrollY 有值，则对 canvas 的坐标进行偏移
            canvas.translate(scrollX, scrollY);


            // 调用 Drawable 的 draw 方法绘制背景
            background.draw(canvas);
            canvas.translate(-scrollX, -scrollY);
        }
    }

/**
  * 步骤2：onDraw(canvas)
  * 作用：绘制View本身的内容
  * 注：
  *   a. 由于 View 的内容各不相同，所以该方法是一个空实现
  *   b. 在自定义绘制过程中，需由子类去实现复写该方法，从而绘制自身的内容
  *   c. 谨记：自定义View中 必须 且 只需复写onDraw（）
  */
  protected void onDraw(Canvas canvas) {

        ... // 复写从而实现绘制逻辑


  }

/**
  * 步骤3： dispatchDraw(canvas)
  * 作用：绘制子View
  * 注：由于单一View中无子View，故为空实现
  */
  protected void dispatchDraw(Canvas canvas) {

        ... // 空实现


  }

/**
  * 步骤4： onDrawScrollBars(canvas)
  * 作用：绘制装饰，如 滚动指示器、滚动条、和前景等
  */
  public void onDrawForeground(Canvas canvas) {
        onDrawScrollIndicators(canvas);
        onDrawScrollBars(canvas);

        final Drawable foreground = mForegroundInfo != null ? mForegroundInfo.mDrawable
```

```
        if (foreground != null) {
            if (mForegroundInfo.mBoundsChanged) {
                mForegroundInfo.mBoundsChanged = false;
                final Rect selfBounds = mForegroundInfo.mSelfBounds;
                final Rect overlayBounds = mForegroundInfo.mOverlayBounds;

                if (mForegroundInfo.mInsidePadding) {
                    selfBounds.set(0, 0, getWidth(), getHeight());
                } else {
                    selfBounds.set(getPaddingLeft(), getPaddingTop(),
                            getWidth() - getPaddingRight(), getHeight() - getPaddingBot
                }

                final int ld = getLayoutDirection();
                Gravity.apply(mForegroundInfo.mGravity, foreground.getIntrinsicWidth(),
                        foreground.getIntrinsicHeight(), selfBounds, overlayBounds, ld)
                foreground.setBounds(overlayBounds);
            }

            foreground.draw(canvas);
        }
    }
```

至此，单一`View`的`draw`过程已分析完毕。

总结

单一`View`的`draw`过程解析如下：



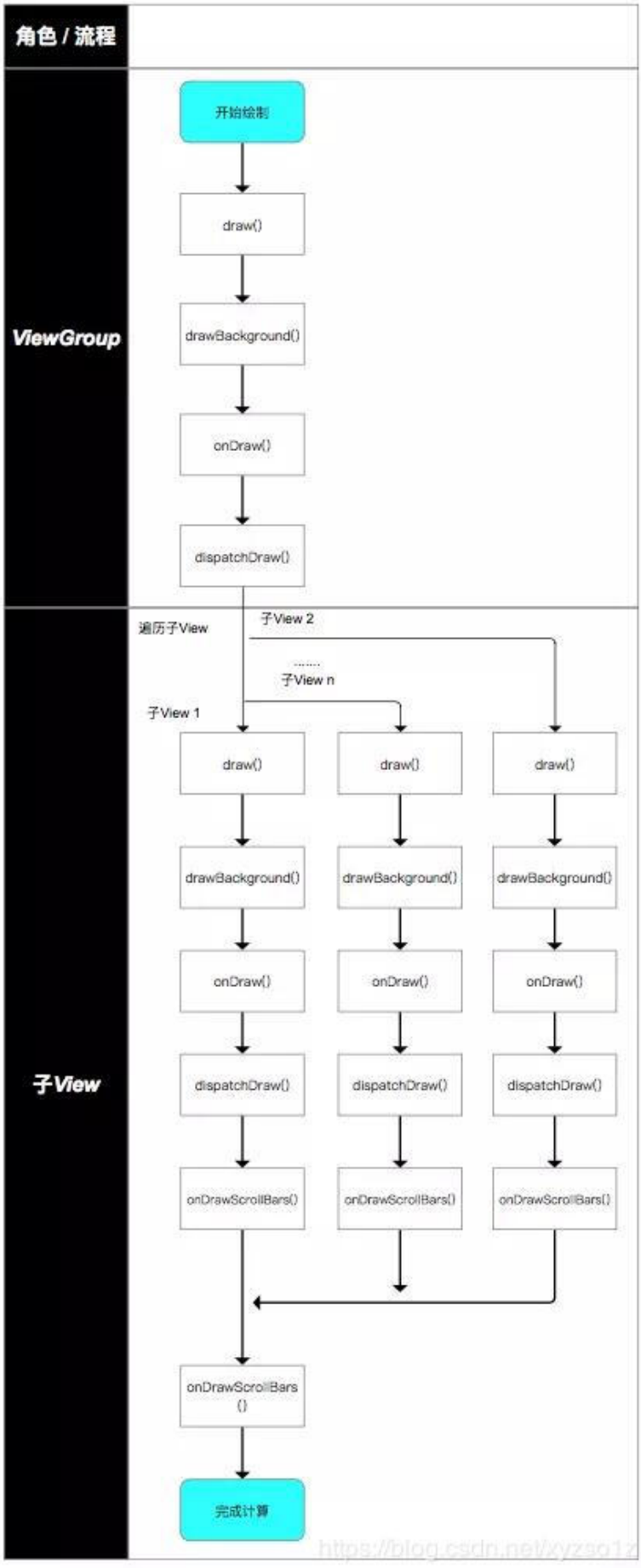
3.2 ViewGroup的draw过程

- 原理（步骤）

1. `ViewGroup` 绘制自身（含背景、内容）
2. `ViewGroup` 遍历子 `View` 和绘制其所有子 `View`
3. `ViewGroup` 绘制装饰（滚动指示器、滚动条和前景）

> 自上而下、一层层地传递下去，直到完成整个 `View` 树的 `draw` 过程

- 具体流程



下面将对每个步骤和方法进行详细分析：`draw`过程的入口=`draw()`

```
/**
 * 源码分析：draw（）
 * 与单一View的draw（）流程类似
 * 作用：根据给定的 Canvas 自动渲染 View（包括其所有子 View）
 * 绘制过程：
 *   1. 绘制view背景
 *   2. 绘制view内容
 *   3. 绘制子View
 *   4. 绘制装饰（渐变框，滑动条等等）
 * 注：
 *    a. 在调用该方法之前必须要完成 layout 过程
 *    b. 所有的视图最终都是调用 View 的 draw （）绘制视图（ ViewGroup 没有复写此方法）
 *    c. 在自定义View时，不应该复写该方法，而是复写 onDraw(Canvas) 方法进行绘制
 *    d. 若自定义的视图确实要复写该方法，那么需先调用 super.draw(canvas)完成系统的
 *       绘制，然后再进行自定义的绘制
 */
  public void draw(Canvas canvas) {

    ...// 仅贴出关键代码

    int saveCount;

    // 步骤1： 绘制本身View背景
        if (!dirtyOpaque) {
            drawBackground(canvas);
        }

    // 若有必要，则保存图层（还有一个复原图层）
    // 优化技巧：当不需绘制 Layer 时，“保存图层“和“复原图层“这两步会跳过
    // 因此在绘制时，节省 Layer 可以提高绘制效率
    final int viewFlags = mViewFlags;
    if (!verticalEdges && !horizontalEdges) {

    // 步骤2：绘制本身View内容
        if (!dirtyOpaque)
            onDraw(canvas);
        // View 中：默认为空实现，需复写
        // ViewGroup中：需复写

    // 步骤3：绘制子View
    // ViewGroup中：系统已复写好对其子视图进行绘制，不需复写
        dispatchDraw(canvas);

    // 步骤4：绘制装饰，如滑动条、前景色等等
        onDrawScrollBars(canvas);

        return;
    }
    ...
}
```

由于步骤2：`drawBackground()`、步骤3:`onDraw()`、步骤5： `onDrawForegound()`，与单个View的draw过程类似，不作过多描述。

下面对步骤4：`dispatchDraw()`

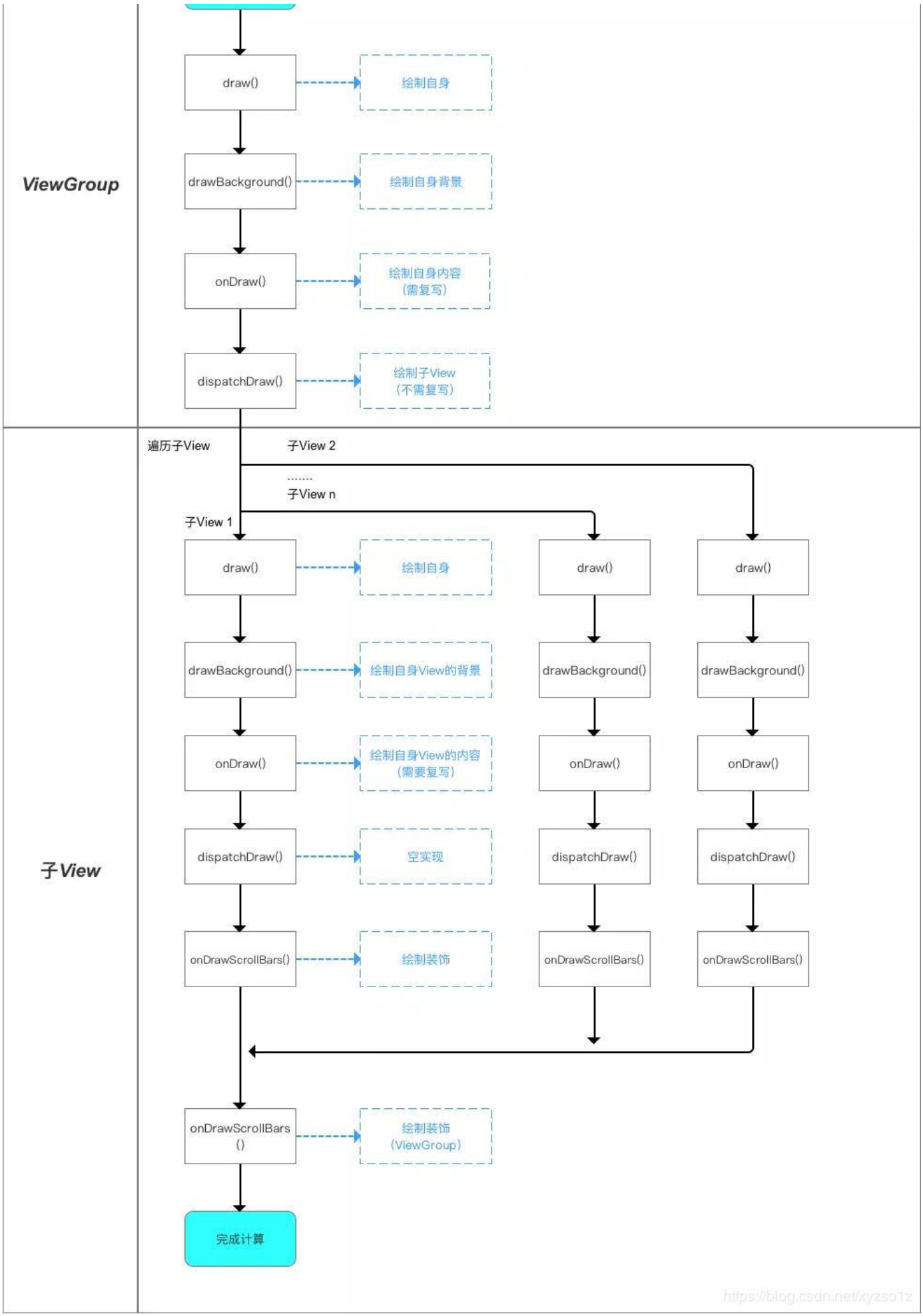
```
/**
  * 源码分析：dispatchDraw（）
  * 作用：遍历子View & 绘制子View
  * 注：
  *   a. ViewGroup中：由于系统为我们实现了该方法，故不需重写该方法
  *   b. View中默认为空实现（因为没有子View可以去绘制）
  */
    protected void dispatchDraw(Canvas canvas) {
        ......

         // 1. 遍历子View
        final int childrenCount = mChildrenCount;
        ......

        for (int i = 0; i < childrenCount; i++) {
                ......
                if ((transientChild.mViewFlags & VISIBILITY_MASK) == VISIBLE ||
                        transientChild.getAnimation() != null) {
                  // 2. 绘制子View视图 ->>分析1
                    more |= drawChild(canvas, transientChild, drawingTime);
                }
                ....
        }
    }

/**
  * 分析1：drawChild（）
  * 作用：绘制子View
  */
    protected boolean drawChild(Canvas canvas, View child, long drawingTime) {
        // 最终还是调用了子 View 的 draw （）进行子View的绘制
        return child.draw(canvas, this, drawingTime);
    }
```

至此，`ViewGroup`的`draw`过程已分析完毕。

总结

`ViewGroup`的`draw`过程如下：

角色 / 流程	
	


ViewGroup
draw()
绘制自身
drawBackground()
绘制自身背景
onDraw()
绘制自身内容
(需复写)
dispatchDraw()
绘制子View
(不需复写)
子View
遍历子View
子View 2
.....
子View n
子View 1
draw()
绘制自身
drawBackground()
绘制自身View的背景
onDraw()
绘制自身View的内容
(需要复写)
dispatchDraw()
空实现
onDrawScrollBars()
绘制装饰
onDrawScrollBars
()
绘制装饰
(ViewGroup)
完成计算

2. 其它细节问题：View.setWillNotDraw()

```
/**
  * 源码分析：setWillNotDraw()
  * 定义：View 中的特殊方法
  * 作用：设置 WILL_NOT_DRAW 标记位；
  * 注：
  *   a. 该标记位的作用是：当一个View不需要绘制内容时，系统进行相应优化
  *   b. 默认情况下：View 不启用该标记位（设置为false）；ViewGroup 默认启用（设置为true）
  */

public void setWillNotDraw(boolean willNotDraw) {

    setFlags(willNotDraw ? WILL_NOT_DRAW : 0, DRAW_MASK);

}

// 应用场景
// a. setWillNotDraw参数设置为true：当自定义View继承自 ViewGroup 、且本身并不具备任何绘制时
//    设置为 true 后，系统会进行相应的优化。
// b. setWillNotDraw参数设置为false：当自定义View继承自 ViewGroup 、且需要绘制内容时，
//    那么设置为 false，来关闭 WILL_NOT_DRAW 这个标记位。
```

xyzso1z
原创文章 80　获赞 40　访问量 2万+