1. 完整绘制流程

我们先看一个完整绘制的调用栈，（这里完整绘制是指没有触发重入的情况，整个 ViewTree 上每个 View 都会完整的走一遍绘制流程）

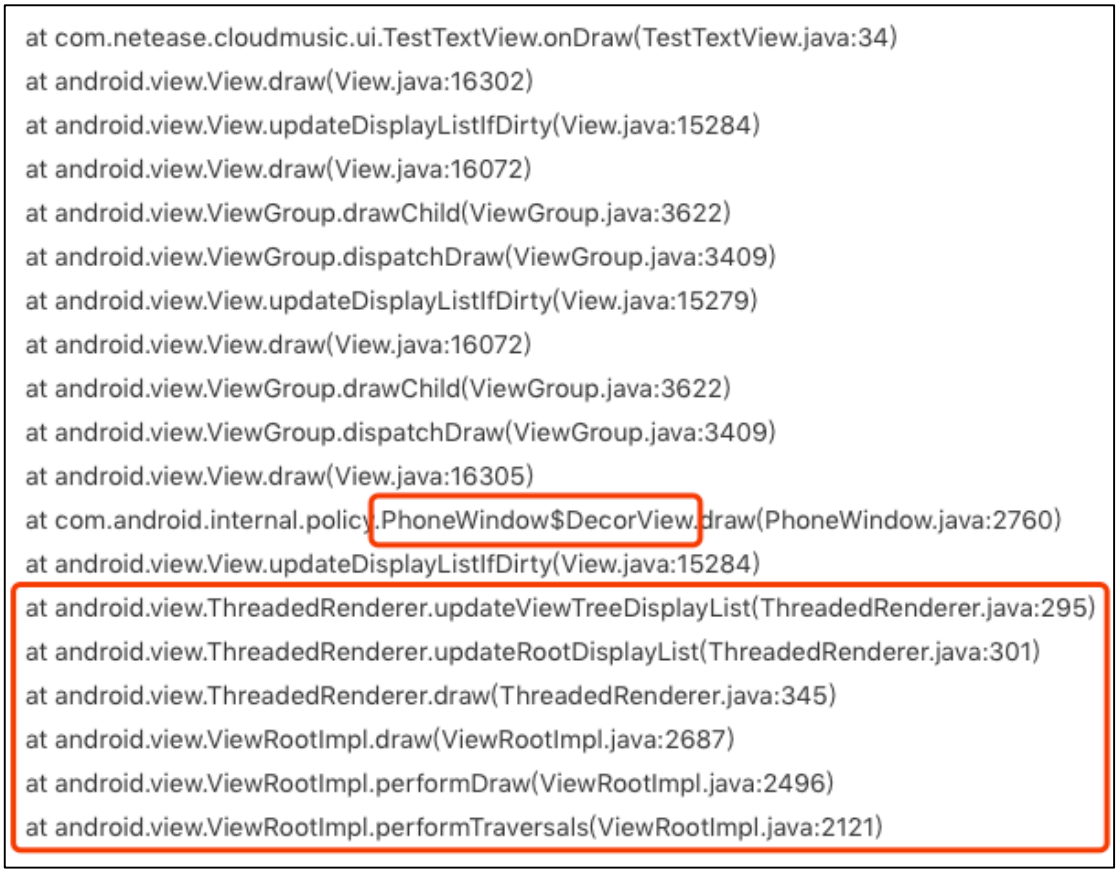

图 1 一次绘制的调用栈

从上面的调用栈可以看到，红框中的部分实际是 framework 比较固定的逻辑，全部绘制逻辑都是从上述逻辑触发的。其中 DecorView 作为所有窗口中 ViewRootImpl 中的第一级子 View，是严格意义上的根 View（虽然理论上 ViewRootImpl 是根 View，但是 ViewRootImpl 只是 implement 了 ViewParent 接口，并不是一个实质的 View；而 DecorView 是继承自 FrameLayout 的，是一个实实在在的 View）。

所以这里我们可以先不管红框中的内容，直接看从 DecorView 开始向下递归的逻辑。可以看到这里有几个固定的函数一直在互相迭代，包括 updateDisplayListIfDirty、draw、dispatchDraw、drawChild 和最终的 onDraw。其中 dispatchDraw 和 drawChild 的调用路径是固定的，dispatchDraw 只会调用 drawChild，所以这里我们把这两个函数作为一组，统称为 dispatchDraw ;另外，从堆栈上也可以看出，这里的 draw 函数实际有两个（同名不同参），为了方便区分，我们把其中接受 3 个入参的称为 draw(3)，接受 1 个入参的称为 draw(1)。

实际上，绘制也可以理解成跟 Input 一样，是 framework 分发的一种事件。所以下面，我们主要分析一下绘制事件在 ViewTree 中流向是怎样的——即 updateDisplayListIfDirty、draw(3)、draw(1)、dispatchDraw、onDraw 这几组逻辑是怎么互相调用的。

1）updateDisplayListIfDirty

updateDisplayListIfDirty 函数是整个绘制的核心，完整绘制和缓存、硬件加速和软件绘制的逻辑，都是在这个函数里控制的，但是其内部的逻辑实际非常简洁；同时，这个函数也

可以理解成是整个 ViewTree 绘制的起点，ThreadedRenderer 就是简单的调用了一下 DecorView 的 updateDisplayListIfDirty 函数，然后递归遍历就开始了。

这里我们先看硬件加速下，完整绘制的情况，缓存的逻辑我们等会再分析。

逻辑如下，（顺便可以观摩一下 Scroll 的逻辑，为啥在 computeScroll 不断调 scrollTo 和 invlidate 就可以实现位移，就是图 2 里那两行代码决定的）

```
} else {
    computeScroll();

    canvas.translate(-mScrollX, -mScrollY);
    mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
    mPrivateFlags &= ~PFLAG_DIRTY_MASK;

    // Fast path for layouts with no backgrounds
    if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
        dispatchDraw(canvas);
        if (mOverlay != null && !mOverlay.isEmpty()) {
            mOverlay.getOverlayView().draw(canvas);
        }
    } else {
        draw(canvas);
    }
```

图 2 updateDisplayListIfDirty 中的逻辑

可以看到，这里以 PFLAG_SKIP_DRAW 为标志做了区分。

PFLAG_SKIP_DRAW 由两个因素共同决定：一个是有没有设置 willNotDraw；另一个是当前 View 有没有背景和前景。

```
if ((mViewFlags & WILL_NOT_DRAW) != 0) {
    if (mBackground != null
            || (mForegroundInfo != null && mForegroundInfo.mDrawable != null)) {
        mPrivateFlags &= ~PFLAG_SKIP_DRAW;
    } else {
        mPrivateFlags |= PFLAG_SKIP_DRAW;
    }
} else {
    mPrivateFlags &= ~PFLAG_SKIP_DRAW;
}
```

图 3 PFLAG_SKIP_DRAW 所对应的逻辑

所以基于图 3 的逻辑，图 2 对应的逻辑可以理解成，对于自身不需要参与绘制的 ViewGroup 和 View，updateDisplayListIfDirty 中会直接调用 dispatchDraw；而需要绘制的 ViewGroup 和普通 View，会调用 draw(1)。

2）dispatchDraw

dispatchDraw 本身在 View 里是个空实现（因为理论上 View 是不需要向下分发绘制事件的）；ViewGroup 重载了这个函数，而在 ViewGroup 的 dispatchDraw 函数里，主要做了下面几件事，

```
for (int i = 0; i < childrenCount; i++) {
    while (transientIndex >= 0 && mTransientIndices.get(transientIndex) == i) {
        final View transientChild = mTransientViews.get(transientIndex);
        if ((transientChild.mViewFlags & VISIBILITY_MASK) == VISIBLE ||
                transientChild.getAnimation() != null) {
            more |= drawChild(canvas, transientChild, drawingTime);
        }
        transientIndex++;
        if (transientIndex >= transientCount) {
            transientIndex = -1;
        }
    }

    final int childIndex = getAndVerifyPreorderedIndex(childrenCount, i, customOrder);
    final View child = getAndVerifyPreorderedView(preorderedList, children, childIndex);
    if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE || child.getAnimation() != null) {
        more |= drawChild(canvas, child, drawingTime);
    }
}
```

图 4 dispatchDraw 的逻辑

a. 处理 layout 动画；
b. 处理 clipToPadding 的逻辑（这里如果 clipToPadding 为 true，这里就会限制 canvas 的绘制范围了）；
c. 按照 getChildDrawingOrder 的顺序，依次调用 drawChild 函数，而 drawChild 默认是直接调用了 View 的 draw(3)函数。

所以，除去不相干的逻辑，在默认情况下，dispatchDraw 的逻辑就是 ViewGroup 遍历调用子 View 的 draw (3)函数。

3）draw(1)

draw(1)和 draw(3)是主要绘制逻辑所在的函数，先看 draw(1)，draw(1)的逻辑实际很简单（图 5），这里依次会绘制 background、onDraw、dispatchDraw 和 foreground。

对于普通 View 来说，draw(1)就是绘制的终点了，这里回调完了 onDraw 函数，由于普通 View 的 dispatchDraw 是个空实现，所以相当于是 ViewTree 中的一个叶子节点，递归和遍历就在这里停止了；但对于参与绘制的 ViewGroup 来说，dispatchDraw 就跟前面分析的一样，会继续向下遍历自己的子 View，调用子 View 的 draw(3)函数。

```
// 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);

    // 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);

    // we're done...
    return;
}
```

图 5 draw(1)的逻辑

4）draw(3)

从前面的分析我们可以看到，在 dispatchDraw 里，并没有区分 View 和 ViewGroup，都会直接调用子 View 的 draw(3)函数，所以这里 draw(3)相当于是，除了 updateDisplayListIfDirty 之外，ViewTree 上的每一个 View（除了 DecorView，DecorView 相当于根节点，没有 parent，没有经过 dispatchDraw，直接是从 updateDisplayListIfDirty 开始的）都会被调用到的函数。

```
if (drawingWithRenderNode) {
    // Delay getting the display list until animation-driven alpha values are
    // set up and possibly passed on to the view
    renderNode = updateDisplayListIfDirty();
    if (!renderNode.isValid()) {
        // Uncommon, but possible. If a view is removed from the hierarchy during the call
        // to getDisplayList(), the display list will be marked invalid and we should not
        // try to use it again.
        renderNode = null;
        drawingWithRenderNode = false;
    }
}
```

图 6 draw(3)的逻辑

draw(3)函数中，实际大部分逻辑都是在处理软件绘制的逻辑，软件绘制中，translate、alpha、scale 和 rotate 的逻辑，都是在这里通过对 canvas 提前做相应操作，然后再调用 dispatchDraw 和 draw(1)实现的（这里 google 的处理实际很巧妙，硬件加速和软件加速共用了大部分代码，绘制流程也是一样的，只是在 draw(3)里做了区分）。

所以只考虑硬件绘制的话，draw(3)里的逻辑实际非常简单，主要有两个，

a. 一个是处理 View 动画的逻辑（这部分逻辑是跟软件绘制共用的，后面会具体分析）；
b. 另一个就是调用自己的 updateDisplayListIfDirty 函数。

所以对于硬件绘制来说，实际上 draw(3)就是又辗转调用了 updateDisplayListIfDirty 而已。

不过这里有个问题可以提前关注一下，我们知道前面讨论的 dispatchDraw、draw(1)和 draw(3)作为绘制的分发函数，有一个共同点——它们都在传递 canvas。但可能跟想象的不同，绘制时并不是从 DecorView 开始，使用同一块 canvas 不断向下传递，让子 View 把内容绘制在我们的 canvas 上。这样做显然缺乏隔离性；同时很多时候，我们的 APP 大部分内容都是空白，如果我们传递一块跟窗口大小一样的 canvas，属于浪费内存。

所以，在绘制时，每个 View 都生成了一块对应自己大小的 Canvas 用来绘制自身的内容。而子 View 的内容要怎么绘制到父 View 的 canvas 上呢？这个跨层级的逻辑就发生在 draw(3)中。

从图 6 可以看到，这里 updateDisplayListIfDirty 的返回值实际是这个 View 的 RenderNode，后面我们会分析到，子 View 所绘制的内容都会保存在 RenderNode 中，在这里通过 updateDisplayListIfDirty 返回值的形式返回。然后我们要怎么把 RenderNode 中的内容，绘在我们父 View 传给我们的 canvas 上呢？

```
if (!drawingWithDrawingCache) {
    if (drawingWithRenderNode) {
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;
        ((DisplayListCanvas) canvas).drawRenderNode(renderNode);
    } else {
        // Fast path for layouts with no backgrounds
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
            mPrivateFlags &= ~PFLAG_DIRTY_MASK;
            dispatchDraw(canvas);
        } else {
            draw(canvas);
        }
    }
```

图 6.1 draw(3)的逻辑

是通过 DisplayListCanvas 的 drawRenderNode 函数，通过这种方式，就实现了跨 View 层级的绘制传递。（这里有点模糊，后面再详细分析）

5）一个例子

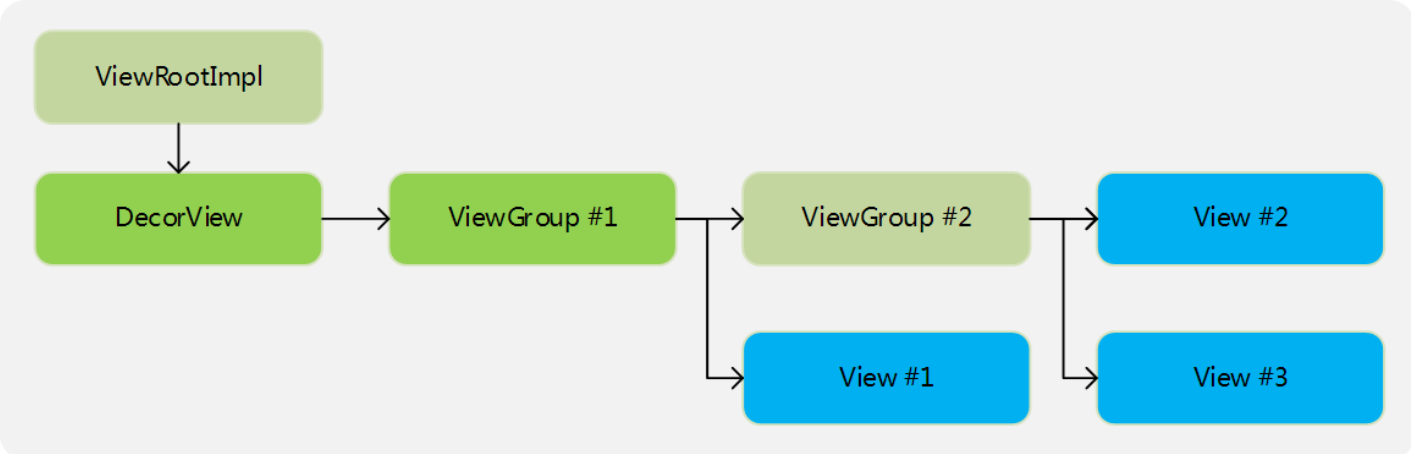

图 7 例子中的布局层级

到这，每一个绘制函数互相之间的调用关系，我们就看完了，下面看一个完整的流程图，会有更好的认识。

首先，我们先想象一个布局（图 7），它的根布局设置了 willNotDraw 为 false，同时，它包含两个子 View：一个普通 View，一个 ViewGroup；而这个子 ViewGroup，又包含了两个子 View，这两个 View 都是普通 View。它对应的绘制流向如下图，

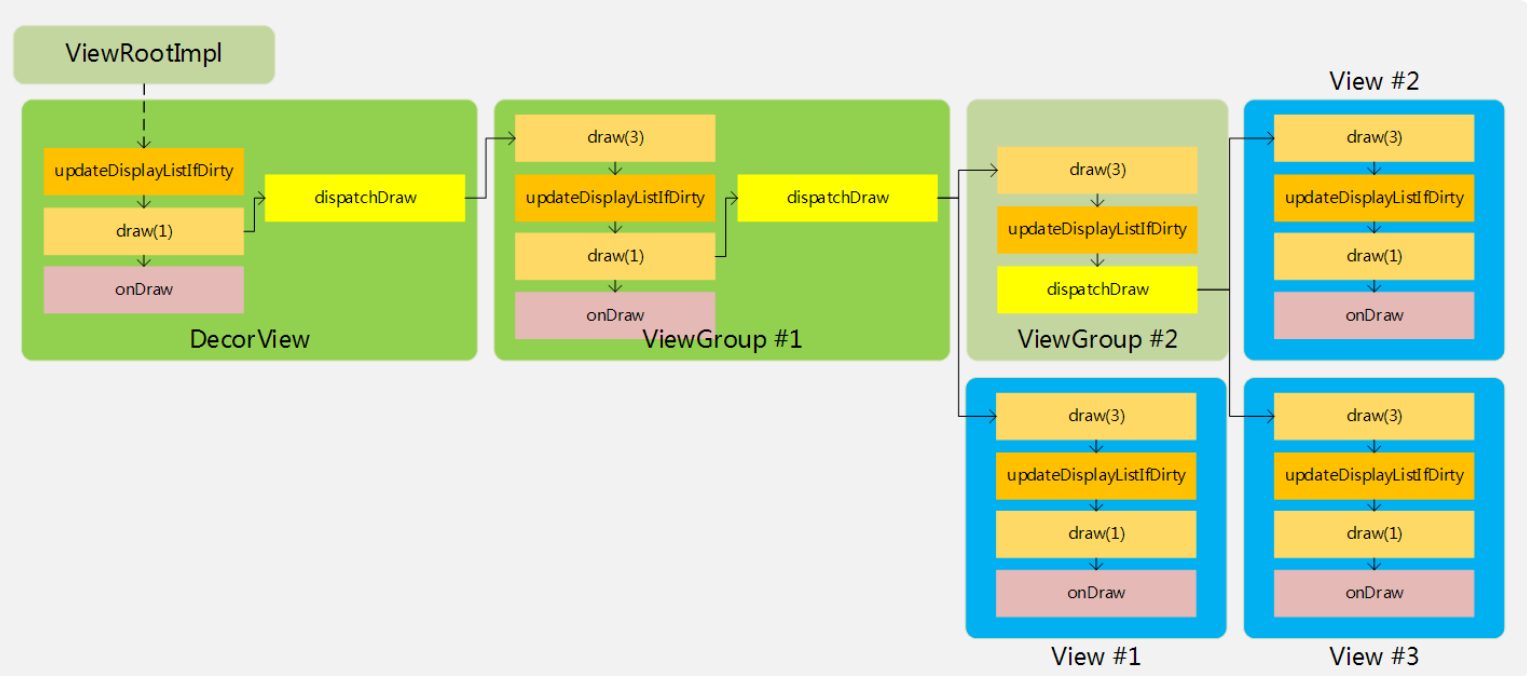

图 8 例子中布局绘制时函数的调用关系

从图上可以看出，基本的流向是比较固定的；另外，Android 里的 ViewTree 遍历都是深度优先的，绘制流程也不例外。

2. 缓存流程

我们知道，在进行过一次绘制之后，如果界面发生变化，为了提高绘制效率，实际我们只需要把“脏区”重绘，其他部分保持缓存不变就可以了。那么这里“脏区”是如何定义的，重绘又是怎样触发的呢？

我们继续来分析 updateDisplayListIfDirty 中的逻辑，核心的逻辑如下，

```
if ((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0
        || !renderNode.isValid()
        || (mRecreateDisplayList)) {
    // Don't need to recreate the display list, just need to tell our
    // children to restore/recreate theirs
    if (renderNode.isValid()
            && !mRecreateDisplayList) {
        mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;
        dispatchGetDisplayList();

        return renderNode; // no work needed
    }
```

图 9 updateDisplayListIfDirty 中重绘的逻辑

这部分逻辑实际在图 2（也就是图 10）中逻辑的前面，也就是说，如果图 9 这里，第一个 if 满足，且没有进入第二个 if 提前返回，那么就会走入完整绘制的流程。

从图 9 这个逻辑可以看出，一共会有三种逻辑分支，对应的就是绘制的三个分支。所以，这里的几个判断条件至关重要——其中主要涉及了 3 个条件，分别是 PFLAG_DRAWING_CACHE_VALID、renderNode.isValid()和 mRecreateDisplayList，我们分别来研究一下这 3 个变量。

1）renderNode.isValid()

```
final DisplayListCanvas canvas = renderNode.start(width, height);
canvas.setHighContrastText(mAttachInfo.mHighContrastText);

try {
    if (layerType == LAYER_TYPE_SOFTWARE) {
        buildDrawingCache(true);
        Bitmap cache = getDrawingCache(true);
        if (cache != null) {
            canvas.drawBitmap(cache, 0, 0, mLayerPaint);
        }
    } else {
        computeScroll();

        canvas.translate(-mScrollX, -mScrollY);
        mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;

        // Fast path for layouts with no backgrounds
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
            dispatchDraw(canvas);
            if (mOverlay != null && !mOverlay.isEmpty()) {
                mOverlay.getOverlayView().draw(canvas);
            }
        } else {
            draw(canvas);
        }
    }
} finally {
    renderNode.end(canvas);
    setDisplayListProperties(renderNode);
}
```

图 10 updateDisplayListIfDirty 的相关逻辑

我们知道，在每个 View 初始化时，在 View 的构造函数里都创建了一个 RenderNode 对象，这个对象持有了 Native 层同名变量的指针，而重绘时，会先生成一个 DisplayListCanvas，这个 DisplayListCanvas 就作为上面几个绘制函数互相传递的 canvas 变量，发生绘制时，具体的操作就在 DisplayListCanvas 上进行（具体的绘制过程我们等会再分析）。

等上述绘制结束后，会调用 RenderNode.end(canvas)方法，会把绘制的参数传递到底层保存，这个过程后面我们会进一步分析。而当 end 方法结束后，就会把对应的 RenderNode 置为 valid。

也就是说，renderNode.isValid()这个判断可以理解成——对应的这个 View（RenderNode）有没有发生过绘制，绘制的内容有没有保存在 RenderNode 中。如果 View 第一次创建，还没有经过绘制，那么 renderNode.isValid()就是 false；另外在 onDetachedFromWindow 中，也会清除底层数据，重置此变量。

2）mRecreateDisplayList

```
if (hardwareAcceleratedCanvas) {
    // Clear INVALIDATED flag to allow invalidation to occur during rendering, but
    // retain the flag's value temporarily in the mRecreateDisplayList flag
    mRecreateDisplayList = (mPrivateFlags & PFLAG_INVALIDATED) != 0;
    mPrivateFlags &= ~PFLAG_INVALIDATED;
}
```

图 11 mRecreateDisplayList 相关的逻辑

mRecreateDisplayList 这个变量比较简单，这个变量是 PFLAG_INVALIDATE 在调用 updateDisplayListIfDirty 之前的缓存。

为什么要这样设计呢？注释上有解释，因为 updateDisplayListIfDirty 就代表绘制的开始，而在绘制过程中，我们随时可能调用 invalidate()，比如可以通过在 onDraw 中不断调用 invalidate()来实现动画效果。所以为了保证在绘制过程中 PFLAG_INVALIDATE 这个 flag 是可用的，在调用 updateDisplayListIfDirty 之前我们要先把它缓存起来，就缓存在 mRecreateDisplayList 这个变量中。

3）PFLAG_DRAWING_CACHE_VALID

这个 flag 不太好理解，需要结合图 9 缓存的逻辑一块分析。从图 9 的逻辑可以看到，当 renderNode.isValid()为 true，mRecreateDisplayList 为 false，而(mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0 时，也就是当前 View 是绘制过的，而且当前 View 也不是 invalidate 的，才会走入 dispatchGetDisplayList()这个缓存的分支。

这涉及到 invalidate 的流程，在 invalidate 的流程中，实际把重绘分裂成两个级别了，分别对应 PFLAG_INVALIDATE 和 PFLAG_DRAWING_CACHE_VALID——PFLAG_INVALIDATE 代表强制重绘，即只要设置了这个 flag 的 View 就一定会被重绘；而 PFLAG_DRAWING_CACHE_VALID 只是表示当前缓存是有效的，所以当这个 flag 被置空时，代表此时我们自身不需要重绘，但是需要检查缓存是否发生改变，也就是我们的子 View（如果有的话）是否需要重绘，相比于 PFLAG_INVALIDATE，这是一种程度更轻的重绘。

通过上面的分析，我们就可以看出哪些条件可以进入缓存逻辑，这也是绘制流程对应的全部 3 种分支，

1.如果当前 View 既没有 PFLAG_INVALIDATE 也没有 PFLAG_DRAWING_CACHE_VALID，同时 RenderNode 也是有效的，此时将直接跳过这个 View 的绘制，即保持上次的绘制结果不变，可以理解成是完全缓存的逻辑；
2. 如果只是 PFLAG_DRAWING_CACHE_VALID，则走部分缓存（也可以叫部分重绘）的逻辑；
3. 否则上述两个分支不满足，则走完全重绘的逻辑。

但需要注意两个问题，一个是，上述 3 个分支都是发生在 updateDisplayListIfDirty 这个函数当中，也就是说是以 View 为单位，按层级向下递归的；另外需要注意的是，如果一个 View 处于完全缓存状态，会直接返回，那么向下的递归绘制就不会发生了（也就是保持它的子 View 内容不变）。

下面，我们看一下部分缓存的分支里，dispatchGetDisplayList 这个函数里做了什么操作。

4）dispatchGetDisplayList

在 View 中，这个函数是空实现；而在 ViewGroup 中，参考图 12，这个函数跟 dispatchDraw 的逻辑有点类似，也是遍历子 View 依次调用 recreateChildDisplayList 函数，而在 recreateChildDisplayList 函数，直接会调用 updateDisplayListIfDirty 函数（也是子 View 的对

应函数）。然后子 View 的 updateDisplayListIfDirty 中还是有三种分支，根据子 View 的标志位继续选择要走的分支，并依次反复，从而实现了对 ViewTree 的遍历。

另外，值得注意的是，在 recreateChildDisplayList 里，依然是先把 PFLAG_INVALIDATE 缓存在 mRecreateDisplayList 变量里，然后再调用 updateDisplayListIfDirty。

这个部分缓存的逻辑存在的意义是什么呢？

其实不难理解，这种逻辑可以实现更高效的重绘。比如当一个 ViewGroup 中有多个子 View 时，当其中有一个 View 需要 invalidate 时，我们可以把这个 ViewGroup 及其以上的所有 View 设为 PFLAG_DRAWING_CACHE_VALID,把需要重绘的 View 设为 PFLAG_INVALIDATE。这样下一帧绘制时，不需要重绘整个 ViewGroup 及其以外的 View（因为它们处于部分缓存的状态），但仍可以保证对 ViewTree 的遍历发生，从而可以找到那个已经 invalidate 的子 View 并触发重绘。（前面说过，如果 View 处于完全缓存状态，递归就提前结束了，那么我们就无法递归找到那个需要重绘的 View 了）

```
@Override
protected void dispatchGetDisplayList() {
    final int count = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < count; i++) {
        final View child = children[i];
        if (((child.mViewFlags & VISIBILITY_MASK) == VISIBLE || child.getAnimation() != null)) {
            recreateChildDisplayList(child);
        }
    }
    if (mOverlay != null) {
        View overlayView = mOverlay.getOverlayView();
        recreateChildDisplayList(overlayView);
    }
    if (mDisappearingChildren != null) {
        final ArrayList<View> disappearingChildren = mDisappearingChildren;
        final int disappearingCount = disappearingChildren.size();
        for (int i = 0; i < disappearingCount; ++i) {
            final View child = disappearingChildren.get(i);
            recreateChildDisplayList(child);
        }
    }
}

private void recreateChildDisplayList(View child) {
    child.mRecreateDisplayList = (child.mPrivateFlags & PFLAG_INVALIDATED) != 0;
    child.mPrivateFlags &= ~PFLAG_INVALIDATED;
    child.updateDisplayListIfDirty();
    child.mRecreateDisplayList = false;
}
```

图 12 dispatchGetDisplayList 中的逻辑

图 13 是一次子 View invalidate 所触发的堆栈，可以看到，位于 ViewRootImp 到这个脏 View 路径上的所有 View，都设置了 PFLAG_DRAWING_CACHE_VALID 处于部分缓存状态，这样在 performTraversals 的过程中，我们就可以不断向下递归，顺利找到这个“脏” View 了。

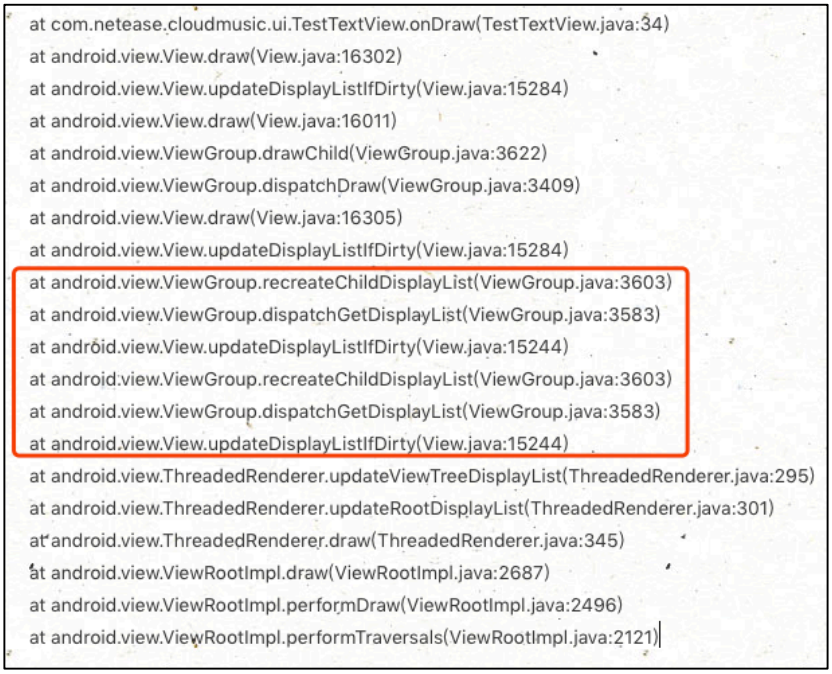

图 13 一次部分缓存的堆栈

3. View 动画 v.s. 属性动画

铺垫了这么久，我们来看看最初要讨论的问题，View 动画和属性动画在绘制效率上的差异究竟在哪。

1）属性动画

首先看看属性动画，我们知道属性动画本质也是调用 View 对应属性的 set 接口而已，这里以 setRotation 为例，对应逻辑如下，

```
public void setRotation(float rotation) {
    if (rotation != getRotation()) {
        // Double-invalidation is necessary to capture view's old and new areas
        invalidateViewProperty(true, false);
        mRenderNode.setRotation(rotation);
        invalidateViewProperty(false, true);

        invalidateParentIfNeededAndWasQuickRejected();
        notifySubtreeAccessibilityStateChangedIfNeeded();
    }
}
```

图 14 setRotation 的逻辑

首先这里有防重入判断；然后只是把旋转的角度设置到了 RenderNode 中。当发生渲染时，Render 线程就会根据 RenderNode 中的旋转角度把对应 RenderNode 旋转。然后我们看一下这里的 invalidateViewProperty 函数，

```
void invalidateViewProperty(boolean invalidateParent, boolean forceRedraw) {
    if (!isHardwareAccelerated()
            || !mRenderNode.isValid()
            || (mPrivateFlags & PFLAG_DRAW_ANIMATION) != 0) {
        if (invalidateParent) {
            invalidateParentCaches();
        }
        if (forceRedraw) {
            mPrivateFlags |= PFLAG_DRAWN; // force another invalidation with the new orientation
        }
        invalidate(false);
    } else {
        damageInParent();
    }
    if (isHardwareAccelerated() && invalidateParent && getZ() != 0) {
        damageShadowReceiver();
    }
}
```

图 15 invalidateViewProperty 的逻辑

这个函数，无论怎么传参，对于硬件加速的 APP 来说，也只是调用了 damageInParent 而已，而 damageInParent 相关的逻辑如下，

```
/**
 * Quick invalidation method called by View.invalidateViewProperty. This doesn't set the
 * DRAWN flags and doesn't handle the Animation logic that the default invalidation methods
 * do; all we want to do here is schedule a traversal with the appropriate dirty rect.
 *
 * @hide
 */
public void damageChild(View child, final Rect dirty) {
    if (damageChildDeferred(child)) {
        return;
    }
```

图 16 damageInParent 相关的逻辑

也就是说，这里只是触发了一次 scheduleTraversal 并设置了脏区而已，并没有走任何 invalidate 的逻辑。并且这里之所以设置脏区，是为了让 ViewRootImpl 调用 performDraw 函数时能进入 draw 流程（这里具体的逻辑就不贴了，就是绘制之前会判断一下脏区是不是为空）。

那么，在这种状态下，如果进入绘制流程，会走前面分析的哪一种分支呢？由于这里没有设置任何绘制的标记，所以从前面对 updateDisplayListIfDirty 函数的分析可以看出，属性动画走的绘制分支是完全缓存，也就是完全不触发 CPU 绘制。

可以想象，在这种状态下，绘制耗时为 0，只有渲染耗时，不愧是效率最高的动画；而且即使不应用于动画，只是通过设置属性改变一个 View 的位置，也完全不需要触发重绘，相当高效而神奇。

2）View 动画

前面提到过，View 动画的实现在 draw(3)中，实际是一种为早期软件绘制的 Android 版本设计的动画效果，我们可以看下具体的代码，

```
if (a != null) {
    more = applyLegacyAnimation(parent, drawingTime, a, scalingRequired);
    concatMatrix = a.willChangeTransformationMatrix();
    if (concatMatrix) {
        mPrivateFlags3 |= PFLAG3_VIEW_IS_ANIMATING_TRANSFORM;
    }
    transformToApply = parent.getChildTransformation();
```

图 17 View 动画相关的逻辑（1）

首先，每一帧时都需要调用 applyLegacyAnimation 函数，同时获取当前 animation 的变化矩阵 concatMatrix，我们直接看 applyLegacyAnimation 函数。

```
if (more) {
    if (!a.willChangeBounds()) {
        if ((flags & (ViewGroup.FLAG_OPTIMIZE_INVALIDATE | ViewGroup.FLAG_ANIMATION_DONE)) ==
                ViewGroup.FLAG_OPTIMIZE_INVALIDATE) {
            parent.mGroupFlags |= ViewGroup.FLAG_INVALIDATE_REQUIRED;
        } else if ((flags & ViewGroup.FLAG_INVALIDATE_REQUIRED) == 0) {
            // The child need to draw an animation, potentially offscreen, so
            // make sure we do not cancel invalidate requests
            parent.mPrivateFlags |= PFLAG_DRAW_ANIMATION;
            parent.invalidate(mLeft, mTop, mRight, mBottom);
        }
```

图 18 View 动画相关逻辑（2）

applyLegacyAnimation 中，前面的逻辑主要是根据当前帧时间计算当前动画的进度，之后会走到图 18 这里，然后直接调用了 parent 的 invalidate 函数。好吧，我们分析了半天，原来 View 动画直接调用了 invalidate。但还是有一个问题：我们实际测试时，在做 View 动画时，实际并没有回调 View 的 onDraw 函数，这又是怎么实现的呢？

首先，在整个 View 动画的逻辑里，我们都没有修改自身的绘制 flag，只是调用了 parent.invalidate 函数；而在 parent.invalidate 中，实际是把自己的父 View 设置为完整重绘了，然后又调用了父 View 的 parent（就是比我们高两级的那个 View）的 invalidateChild，

```
if (invalidateCache) {
    mPrivateFlags |= PFLAG_INVALIDATED;
    mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
}

// Propagate the damage rectangle to the parent view.
final AttachInfo ai = mAttachInfo;
final ViewParent p = mParent;
if (p != null && ai != null && l < r && t < b) {
    final Rect damage = ai.mTmpInvalRect;
    damage.set(l, t, r, b);
    p.invalidateChild(this, damage);
}
```

图 19 parent.invalidate 函数的相关逻辑

```
if (child.mLayerType != LAYER_TYPE_NONE) {
    mPrivateFlags |= PFLAG_INVALIDATED;
    mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;
}
```

图 20 p.invalidateChild 函数中的逻辑

在 invalidateChild 中，有一个很重要的逻辑（图 20），可以看到这里对 LayerType 进行了判断，只有当 LayerType 为 None 时，才不会修改 PFLAG_INVALIDATE 和 PFLAG_DRAWING_CACHE_VALID（View 默认的 LayerType 是 None，无论是在硬件加速还是软件加速下）。接着这个函数还会调用 invalidateChildInParent 函数，在这个函数中的逻辑也是类似的（图 21），

```
mPrivateFlags &= ~PFLAG_DRAWING_CACHE_VALID;

location[CHILD_LEFT_INDEX] = left;
location[CHILD_TOP_INDEX] = top;

if (mLayerType != LAYER_TYPE_NONE) {
    mPrivateFlags |= PFLAG_INVALIDATED;
}
```

图 21 invalidateChildInParent 函数中的逻辑

所以我们重新理顺一下 View 动画调用 parent.invalidate 之后都发生了什么：

- 首先，parent.invalidate 函数相当于把自己父 View 设为了完整重绘；
- 然后，通过不断向上递归调用 invalidateChild 函数，把父 View 和 ViewRootImpl 之间的 View 都去掉了 PFLAG_DRAWING_CACHE_VALID flag，设为部分缓存的状态；
- 同时我们也可以看到，整个过程中，对这个 View 本事没有设置任何绘制 flag，也就是自身处于完全缓存状态。

所以最终的结果如用下面的堆栈，刚好是 3 种绘制状态结合的情况，

```
at com.netease.cloudmusic.theme.ui.CustomThemeConstraintLayout.invalidate(CustomThemeConstraintLayout.java:89)
at android.view.View.applyLegacyAnimation(View.java:15931)                    完全缓存
at android.view.View.draw(View.java:16011)
at android.view.ViewGroup.drawChild(ViewGroup.java:3622)
at android.view.ViewGroup.dispatchDraw(ViewGroup.java:3409)                   完整重绘
at android.view.View.draw(View.java:16305)
at android.view.View.updateDisplayListIfDirty(View.java:15284)
at android.view.ViewGroup.recreateChildDisplayList(ViewGroup.java:3603)
at android.view.ViewGroup.dispatchGetDisplayList(ViewGroup.java:3583)
at android.view.View.updateDisplayListIfDirty(View.java:15244)
at android.view.ViewGroup.recreateChildDisplayList(ViewGroup.java:3603)       部分缓存
at android.view.ViewGroup.dispatchGetDisplayList(ViewGroup.java:3583)
at android.view.View.updateDisplayListIfDirty(View.java:15244)
at android.view.ViewGroup.recreateChildDisplayList(ViewGroup.java:3603)
at android.view.ViewGroup.dispatchGetDisplayList(ViewGroup.java:3583)
at android.view.View.updateDisplayListIfDirty(View.java:15244)
```

图 22 View 动画时的调用堆栈

为什么要这样处理呢？我们知道在没有硬件加速的 Android 版本上，View 实现旋转、平移等操作，都是通过——先对 canvas 做相应操作，然后再把 View 绘制在处理完的 canvas 上实现的。这也正是 View 动画实现的方式（图 23），

```
if (transformToApply != null) {
    if (concatMatrix) {
        if (drawingWithRenderNode) {
            renderNode.setAnimationMatrix(transformToApply.getMatrix());
        } else {
            // Undo the scroll translation, apply the transformation matrix,
            // then redo the scroll translate to get the correct result.
            canvas.translate(-transX, -transY);
            canvas.concat(transformToApply.getMatrix());
            canvas.translate(transX, transY);
        }
        parent.mGroupFlags |= ViewGroup.FLAG_CLEAR_TRANSFORMATION;
    }
```

图 23 View 动画的实现逻辑（3）

这里以旋转为例，由于发生旋转的 View 本身内容没变，所以不需要触发 View 自身的重绘；但是需要想办法触发自己父 View 的重绘，这样父 View 在重绘时，就可以根据动画先把对应的旋转角度设置在 canvas 上，再把子 View（的缓存）绘制在 canvas 上；而父 View 以上的 View 的内容也没有变化，但是仍需要它们把绘制的流程传递下来，因此就保持部分缓存状态（从这就可以看出，部分缓存实际是一种查询状态）。

上述流程，可以说是最小化的遍历和更新 ViewTree 来实现的 View 动画，这在软件绘制的年代，对性能还是有优化的，因为不需要对动画的 View 进行 onDraw 重绘，整体性能还是略优于 invalidate + draw 这种方式的。

3）离屏缓冲区

另外，从上面对 invalidate 的分析可以看出来，LayerType 实际跟 APP 是硬件加速还是软件加速没有关系，LayerType 实际是指 View 的离屏（off-screen）缓冲区的类型。

a. 默认 View 的 LayerType 为 None，代表 View 没有离屏缓冲（但这也不影响 RenderNode 提供的帧缓冲区的使用）；
b. LAYER_TYPE_HARDWARE 对应了硬件缓冲，是一个保存在 GPU 中的 texture；
c. LAYER_TYPE_SOFTWARE 对应了软件缓冲，在硬件加速状态下也可以使用，是一个保存在 ViewTree 上的 bitmap

这里对于离屏缓冲区需要注意的几点是：

- 本身软件缓冲更多是为了向下兼容（硬件加速不支持全部的绘制指令），一旦开启软件缓冲，RenderNode 里的缓存就不会再用了，所以还是会影响性能；
- 另外，从前面 invalidate 的分析我们可以看出，缓冲区对 invalidate 更敏感，如果一个 ViewGroup 设置了缓冲区，那么它的任意子 View 申请 invalidate，都会触发 ViewGroup 本身发生重绘，以此来更新缓存（跟 View 动画的逻辑有点类似，子 View 的缓存更新是发生在 draw(3)函数里，所以子 View 重绘必须也要触发父 View 重绘才能实现缓存更新）。
- 为了解决缓存区 invalidate 更敏感的问题，可以在开发者选项中打开 “显示硬件层更新” 来调试缓存区更新过于频繁的问题。

4. DisplayList

前面我们讨论了绘制是如何从 ViewRootImpl 向整个 ViewTree 分发的；同时我们还讨论了绘制的三个不同分支；最后讨论了 View 动画的实现和离屏缓冲区的概率。

可以发现，这部分关于 CPU 绘制分发的逻辑，并没有涉及 DisplayList 相关的逻辑，那么究竟什么是 DisplayList 呢？下面我们就继续来讨论一下，具体的绘制是如何进行的，DisplayList 又代表了什么。

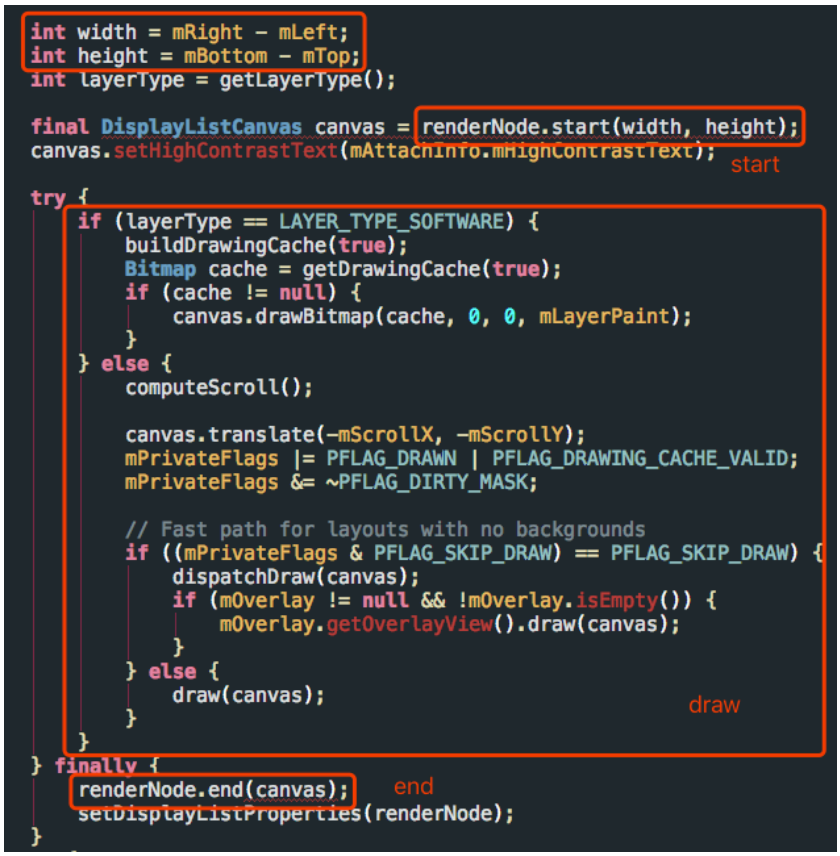

图 24 绘制的三个步骤

前面(图 10)我们已经分析过了，在 updateDisplayListIfDirty 中的 dispatchDraw 和 draw(1)本身是向下遍历绘制下一级 ViewTree 的过程，本质上，就是不断触发 View 向自己的 Canvas 上绘制内容（可以想象，子 View 依次对着自己的 canvas 调用 drawBitmap、drawText、drawCircle 等方法），这部分逻辑可以理解成是在画布上绘制的过程。

所以，从图 24（跟图 10 的代码一样，这里再近距离看一下）可以把绘制行为拆分成三步——start、draw 和 end，我们依次来分析一下。

1）RenderNode.start

前面我们已经提到过了，每个 View 都持有了一个 RenderNode 对象，其对应了 Native 层的一个同名变量。Native 层的 RenderNode 类管理了绘制相关的两个主要逻辑——RenderProperties 和 DisplayList，具体逻辑等会会分析，先来看 RenderNode.start 方法。

```
public DisplayListCanvas start(int width, int height) {
    return DisplayListCanvas.obtain(this, width, height);
}
```

```
private DisplayListCanvas(int width, int height) {
    super(nCreateDisplayListCanvas(width, height));
    mDensity = 0; // disable bitmap density scaling
}
```

图 25 RenderNode.start 相关的逻辑

RenderNode.start 的逻辑不复杂，就是从对象池中获取了一个 DisplayListCanvas 对象，DisplayListCanvas 类本身是继承自 Canvas 的。（另外从图 24 的逻辑，可以看到，这块画布的大小就是 View 的大小）

DisplayListCanvas 的构造函数也是在 Native 层创建了一个同名类的对象。

除了初始化 DisplayListCanvas 对象，start 函数这里也没有其他处理了。

2）在 DisplayListCanvas 上绘制

第二步就是在 canvas 上具体的绘制行为了，我们先来看两个典型的操作：canvas.scale 和 canvas.drawBitmap，两者分别对应画布操作（StateOp）和具体的绘制行为（DrawOp）。

a）canvas.scale

DisplayListCanvas.java 本身是没有实现 scale 方法的，所以这里直接调用的是 Canvas.java 的 scale 方法，然后通过 JNI 辗转调用到了 DisplayListCanvas.cpp 的对应方法，逻辑如下，

```
void DisplayListCanvas::scale(float sx, float sy) {
    if (sx == 1.0f && sy == 1.0f) return;

    addStateOp(new (alloc()) ScaleOp(sx, sy));
    mState.scale(sx, sy);
}
```

图 26 DisplayListCanvas.cpp 的 scale 函数

这里的逻辑比较清晰，第一步是参数检查，对于 scale 参数为 1 的时候不做处理（不过这里建议还是直接在 JAVA 层检查，毕竟 JNI 调用也是有成本的）；第二步是 addStateOp，相关逻辑如下，

```
class ScaleOp : public StateOp {
public:
    ScaleOp(float sx, float sy)
            : mSx(sx), mSy(sy) {}

    virtual void applyState(OpenGLRenderer& renderer, int saveCount) const override {
        renderer.scale(mSx, mSy);
    }

    virtual void output(int level, uint32_t logFlags) const override {
        OP_LOG("Scale by %f %f", mSx, mSy);
    }

    virtual const char* name() override { return "Scale"; }
```

```
size_t DisplayListCanvas::addOpAndUpdateChunk(DisplayListOp* op) {
    int insertIndex = mDisplayList->ops.size();
#if HWUI_NEW_OPS
    LOG_ALWAYS_FATAL("unsupported");
#else
    mDisplayList->ops.push_back(op);
#endif
```

图 27 addStateOp 相关的逻辑

在图 26 中，这里先 alloc 了一个 ScaleOp 对象，而从图 27（上）可以看到，ScaleOp 这个类是继承自 StateOp 的，StateOp 又继承自 DisplayListOp（公共基类），从这种继承关系和命名规则就大概可以猜想到，这里 DisplayListCanvas.cpp 把对 canvas 的操作抽象成了 operation 的概念；另外可以看到，在 ScaleOp 中实现了一个 applyState 函数，里面调用的是 renderer.scale，这里是对真正的底层 OpenGL 节点进行操作（不过暂时还没有用到这个调用，后面我们会看到具体调用的位置），也就是说我们把真正的绘制指令分装在了一个 operation 对象中。

继续来看图 27（下），addStateOp 后续会辗转调用到 mDisplayList->ops.push_back(op)。mDisplayList 这个成员变量，定义在 DisplayList.cpp 中，内部没有什么逻辑，ops 是内部的

一个 Vector，所以 DisplayList.cpp 可以理解成 DisplayListCanvas 中一个用来保存数据的类。综上，这里的操作就是把刚才创建的 operation 加入到 Vector 中。

```
void CanvasState::scale(float sx, float sy) {
    mSnapshot->transform->scale(sx, sy, 1.0f);
}
```

图 28 mState.scale(sx, sy)相关逻辑

然后我们再来看图 26 中剩下的一句调用 mState.scale(sx, sy)，这里 mState 是一个 CanvasState 对象，其内部 mSnapshot->transform 是一个矩阵，所以这里相当于对这个矩阵做了 scale 变换。

CanvasState 相关的逻辑也比较好理解，前面我们看到，scaleOp 入栈之后实际对当前这个 DisplayListCanvas.cpp 没有产生任何影响，我们要怎么获取当前 canvas 的状态呢？显然不可能从 mDisplayList 依次取出 operation 再叠加计算一遍，这样效率太低。因此，在 DisplayListCanvas.cpp 里就还需要一个缓存当前 canvas 状态的变量，提供给 JAVA 层，用以查询（比如 getClipBounds、quickRejectRect 等需要查询 canvas 状态的函数），这就是 mState（CanvasState）这个变量的作用。

从上面 canvas.scale 的逻辑，我们已经可以大概看出 DisplayList 的概念和设计了，我们继续看一下 canvas.drawBitmap。

b）canvas.drawBitmap

```
void DisplayListCanvas::drawBitmap(const SkBitmap* bitmap, const SkPaint* paint) {
    bitmap = refBitmap(*bitmap);
    paint = refPaint(paint);

    addDrawOp(new (alloc()) DrawBitmapOp(bitmap, paint));
}
```

图 29 canvas.drawBitmap 相关的逻辑

从图 29 可以看到，这里的逻辑跟 scale 差不多，还是抽象成一个 operation，这里对应的是 DrawBitmapOp，依次继承自 DrawBoundedOp -> DrawOp-> DisplayListOp。

```
class DrawBitmapOp : public DrawBoundedOp {
public:
    DrawBitmapOp(const SkBitmap* bitmap, const SkPaint* paint)
            : DrawBoundedOp(0, 0, bitmap->width(), bitmap->height(), paint)
            , mBitmap(bitmap)
            , mEntryValid(false), mEntry(nullptr) {
    }

    virtual void applyDraw(OpenGLRenderer& renderer, Rect& dirty) override {
        renderer.drawBitmap(mBitmap, mPaint);
    }

    AssetAtlas::Entry* getAtlasEntry(OpenGLRenderer& renderer) {
        if (!mEntryValid) {
            mEntryValid = true;
            mEntry = renderer.renderState().assetAtlas().getEntry(mBitmap->pixelRef());
        }
        return mEntry;
    }
```

图 30 DrawBitmapOp 相关的逻辑

可以看到，DrawBoundedOp 这层主要保存了当前 bitmap 的大小和 paint，而 DrawBitmapOp 这层保存了 bitmap，结构很清晰；同样这里也有 appleDraw 函数，同样也暂时没有调用。所以，这里具体的绘制操作，还是抽象成了一个 operation。

相比于把操作抽象成 operation，我们更关心另一个问题，这里的 bitmap 是怎么管理的，我们继续来看图 29 中 refBitmap 相关的逻辑。

```
inline const SkBitmap* refBitmap(const SkBitmap& bitmap) {
    // Note that this assumes the bitmap is immutable. There are cases this won't handle
    // correctly, such as creating the bitmap from scratch, drawing with it, changing its
    // contents, and drawing again. The only fix would be to always copy it the first time,
    // which doesn't seem worth the extra cycles for this unlikely case.
    SkBitmap* localBitmap = alloc().create<SkBitmap>(bitmap);
    mDisplayList->bitmapResources.push_back(localBitmap);
    return localBitmap;
}
```

```
template<class T, typename... Params>
T* create(Params&&... params) {
    T* ret = new (allocImpl(sizeof(T))) T(std::forward<Params>(params)...);
    if (!std::is_trivially_destructible<T>::value) {
        auto dtor = [](void* ret) { ((T*)ret)->~T(); };
        addToDestructionList(dtor, ret);
    }
    return ret;
}
```

```
/**
 *  Copy the settings from the src into this bitmap. If the src has pixels
 *  allocated, they will be shared, not copied, so that the two bitmaps will
 *  reference the same memory for the pixels. If a deep copy is needed,
 *  where the new bitmap has its own separate copy of the pixels, use
 *  deepCopyTo().
 */
SkBitmap(const SkBitmap& src);
```

图 31 refBitmap 相关的逻辑

这里 create 了一个 SkBitmap，但是跟踪代码可以看到，这里毫无疑问是浅拷贝，即跟之前的 bitmap 共享 pixel，这里只拷贝 settings（方便理解，这里可以想象成 Drawable.java，逻辑是一样的）。从这里的逻辑可以看到，DisplayList 里只是保存了 bitmap 的一个浅拷贝，所以注释中也提到了，如果需要复用同一个 bitmap，来多次绘制不同内容，是无法实现的，需要自己创建深拷贝。

同时，这里把创建出的浅拷贝，保存进了 DisplayList.cpp 的 bitmapResources 这个 Vector。

总结一下，从上面两种绘制的逻辑我们基本可以看出 DisplayList 的概念了，这里实际是 Android 采用了延迟绘制的设计，即把绘制的逻辑和实际绘制的时机次错开，当我们在 UI 线程对 canvas 做各种绘制操作时，只是把对应的操作抽象成一个 operation，并按顺序保存起来；当我们真正需要绘制的时候，再依次取出这些 operation。这样最直接的好处就是，上层的绘制过程可以跟底层异步处理了，从而解放了 UI 线程。

其他的 canvas 的绘制操作也都是类似的实现方式，这里我们就不再赘述了。

那么，这些绘制指令真正发挥作用是在什么时候呢？是在 renderNode.end(canvas)中么？我们继续分析。

3）RenderNode.end

```java
public void end(DisplayListCanvas canvas) {
    long displayList = canvas.finishRecording();
    nSetDisplayList(mNativeRenderNode, displayList);
    canvas.recycle();
    mValid = true;
}
```

图 32 RenderNode.end 中的逻辑

通过变量命名（图 32）大概可以看出，这里先通过 DisplayListCanvas.finishRecording 取出 displayList 设置给 RenderNode；之后马上把 DisplayListCanvas 回收了；最后设置了标志位（前面说过 RenderNode.isValid 的判断，就是根据这个标志位）。我们大概过一下其他的逻辑，

```java
DisplayList* DisplayListCanvas::finishRecording() {
    flushRestoreToCount();
    flushTranslate();

    mPaintMap.clear();
    mRegionMap.clear();
    mPathMap.clear();
    DisplayList* displayList = mDisplayList;
    mDisplayList = nullptr;
    mSkiaCanvasProxy.reset(nullptr);
    return displayList;
}
```

图 33 finishRecord 相关的逻辑

finishRecording 的逻辑很简单，主要是清除了大部分变量用以下一次复用（不要忘了上层的 DisplayListCanvas.java 里有个对象池）,然后把 mDisplayList 的指针传回来了。（前面 flushRestoreToCount 和 flushTranslate 函数，主要是 canvas.save + restore 相关的逻辑，也被抽象成 operation 操作了，这里就暂时先不分析了）

nSetDisplayList 的逻辑如下图，

```java
static void android_view_RenderNode_setDisplayList(JNIEnv* env,
        jobject clazz, jlong renderNodePtr, jlong displayListPtr) {
    class RemovedObserver : public TreeObserver {
    public:
        virtual void onMaybeRemovedFromTree(RenderNode* node) override {
            maybeRemovedNodes.insert(sp<RenderNode>(node));
        }
        std::set< sp<RenderNode> > maybeRemovedNodes;
    };

    RenderNode* renderNode = reinterpret_cast<RenderNode*>(renderNodePtr);
    DisplayList* newData = reinterpret_cast<DisplayList*>(displayListPtr);
    RemovedObserver observer;
    renderNode->setStagingDisplayList(newData, &observer);
    for (auto& node : observer.maybeRemovedNodes) {
        if (node->hasParents()) continue;
        onRenderNodeRemoved(env, node.get());
    }
}
```

```java
void RenderNode::setStagingDisplayList(DisplayList* displayList, TreeObserver* observer) {
    mNeedsDisplayListSync = true;
    delete mStagingDisplayList;
    mStagingDisplayList = displayList;
    // If mParentCount == 0 we are the sole reference to this RenderNode,
    // so immediately free the old display list
    if (!mParentCount && !mStagingDisplayList) {
        deleteDisplayList(observer);
    }
}
```

图 34 nSetDisplayList 的相关逻辑

这里主要有两部分逻辑，一部分是把前面从 DisplayListCanvas 中取出来的 displayList 保存到自己的 mStagingDisplayList 变量中，并设置了 mNeedsDisplayListSync 标志位；另一部（就是用了观察者模式的那一部分代码）是判断自己是否还有 parent 的逻辑，会把已经不连通的 RenderNode 节点移除掉。

可以看到，在 RenderNode.end 方法中，实际就是把 DisplayListCanvas 中收集的 DisplayList 自己保存了起来，暂时也还没有用到。

4）DisplayList 在 View 之间传递

我们知道，RenderNode 是跟 View 一一对应的，也就是说上面分析的绘制的三个步骤，都是发生在同一个 View 中的——在这个过程中，我们通过对着一块 DisplayListCanvas 一阵操作，生成了一个 View 绘制所需要的 DisplayList，并保存在自己的 RenderNode 中。

从前面对绘制分发流程的分析我们可以看到，绘制是沿着 parent.dispatchDraw -> child.draw(3) -> child.updateDisplayListIfDirty 这条路径向下传递的，现在我们有了自己 RenderNode 对应的 DisplayList，要怎么传递给 parent 来实习递归遍历呢？

```
if (!drawingWithDrawingCache) {
    if (drawingWithRenderNode) {
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;
        ((DisplayListCanvas) canvas).drawRenderNode(renderNode);
    } else {
        // Fast path for layouts with no backgrounds
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
            mPrivateFlags &= ~PFLAG_DIRTY_MASK;
            dispatchDraw(canvas);
        } else {
            draw(canvas);
        }
    }
```

图 35 RenderNode 向上传递

前面（图 6.1）我们已经讨论过了，updateDisplayListIfDirty 的返回值，最终通过 canvas.drawRenderNode 绘制到了上一层级的 canvas 中了，通过这种方式，建立起了一个树状结构，每一层的 RenderNode 中，除了保存有本层的 DisplayList，还保存了下一层 ViewRenderNode 中的信息。

具体保存了哪些信息呢？我们来看看 drawRenderNode 的具体实现，

```
void DisplayListCanvas::drawRenderNode(RenderNode* renderNode) {
    LOG_ALWAYS_FATAL_IF(!renderNode, "missing rendernode");
    DrawRenderNodeOp* op = new (alloc()) DrawRenderNodeOp(
            renderNode,
            *mState.currentTransform(),
            mState.clipIsSimple());
    addRenderNodeOp(op);
}
```

图 36 drawRenderNode 的逻辑

```
public:
    DrawRenderNodeOp(RenderNode* renderNode, const mat4& transformFromParent, bool clipIsSimple)
            : DrawBoundedOp(0, 0,
                    renderNode->stagingProperties().getWidth(),
                    renderNode->stagingProperties().getHeight(),
                    nullptr)
            , renderNode(renderNode)
            , mRecordedWithPotentialStencilClip(!clipIsSimple || !transformFromParent.isSimple())
            , localMatrix(transformFromParent)
            , skipInOrderDraw(false) {}

    virtual void defer(DeferStateStruct& deferStruct, int saveCount, int level,
            bool useQuickReject) override {
        if (renderNode->isRenderable() && !skipInOrderDraw) {
            renderNode->defer(deferStruct, level + 1);
        }
    }

    virtual void replay(ReplayStateStruct& replayStruct, int saveCount, int level,
            bool useQuickReject) override {
        if (renderNode->isRenderable() && !skipInOrderDraw) {
            renderNode->replay(replayStruct, level + 1);
        }
    }
```

图 37 DrawRenderNodeOp 的逻辑

还是抽象成了 operation 操作，DrawRenderNodeOp 本身没什么特别的，也是继承自 DrawBoundedOp。这里实际就是对 RenderNode 做了一层 wrapper，RenderNode 的指针会保存在这里。再来看下 addRenderNodeOp 的相关逻辑，

```
size_t DisplayListCanvas::addRenderNodeOp(DrawRenderNodeOp* op) {
    int opIndex = addDrawOp(op);
#if !HWUI_NEW_OPS
    int childIndex = mDisplayList->addChild(op);

    // update the chunk's child indices
    DisplayList::Chunk& chunk = mDisplayList->chunks.back();
    chunk.endChildIndex = childIndex + 1;

    if (op->renderNode->stagingProperties().isProjectionReceiver()) {
        // use staging property, since recording on UI thread
        mDisplayList->projectionReceiveIndex = opIndex;
    }
#endif
    return opIndex;
}
```

```
size_t DisplayList::addChild(NodeOpType* op) {
    referenceHolders.push_back(op->renderNode);
    size_t index = children.size();
    children.push_back(op);
    return index;
}
```

图 38 addRenderNodeOp 相关的逻辑

也很简单，首先是把 DrawRenderNodeOp 也保存进了 DisplayList 的 Vector 中，然后通过 addChild 保存了一下 child 信息。从这里也可以看出，JAVA 层和 Native 层，对于 parent 和 child 的关系保存也是不同步的——JAVA 层 removeChild 之后，必须要触发重绘，才能把底层 child 的 RenderNode 移除。(这就是为什么在 RenderNode.end 方法中，必须要检查并移除不连通的 RenderNode，因为触发重绘的 View 必然会在 updateDisplayListIfDirty 中调用 RenderNode.end 方法)

综上， RenderNode 对于 DisplayListCanvas 来说，也是一种特殊的绘制 operation，同样保存在 DisplayList 中。可以想象，对于一个 ViewTree，在绘制时，会从每一个 View 节点利用 DisplayListCanvas 向下收集绘制指令，保存在 RenderNode 中，然后再利用 canvas.drawRenderNode 的方式，向上回传，最终全部汇总保存在 DecorView 的 RenderNode 中，在 DecorView 的 RenderNode 中形成一个巨大的 DisplayList。

这就是整个绘制过程的本质——我们层层递归，把全部的绘制指令以抽象为 operation 的方式，按顺序形成了一个 DisplayList，保存在 DecorView 的 RenderNode 中。

5）HWUI_NEW_OPS

实际上，前面分析的 DisplayList 生成逻辑（也就是 View 绘制逻辑）是 Android N 之前的逻辑，在 Android N 之后，通过 HWUI_NEW_OPS 宏，提供了一种新的绘制逻辑。但是，这部分新逻辑，对于绘制阶段来说，跟之前并没有较大的差异，主要差异在渲染阶段。（关于渲染的逻辑，有机会再分析）

在绘制阶段，比较显著的一个差异，就是新的绘制 operation 去除了对于画布的操作，还是以 canvas.scale 和 canvas.drawBitmap 为例，逻辑如下，

```
void RecordingCanvas::scale(float sx, float sy) {
    if (sx == 1 && sy == 1) return;

    mState.scale(sx, sy);
}
```

```
void RecordingCanvas::drawBitmap(const SkBitmap* bitmap, const SkPaint* paint) {
    addOp(alloc().create_trivial<BitmapOp>(
            Rect(bitmap->width(), bitmap->height()),
            *(mState.currentSnapshot()->transform),
            getRecordedClip(),
            refPaint(paint), refBitmap(*bitmap)));
}
```

图 39 canvas.scale 和 canvas.drawBitmap 相关的逻辑

这个逻辑本身也比较好理解，因为本身 canvas 的属性操作是作用与绘制操作的，也就是说属性操作没有单独存在的意义，所以对于新的 operation，属性操作只修改 mState 的状态，然后对于具体的绘制操作，会把 mState 当前的矩阵作为参数传递进去。

```
int CanvasState::saveSnapshot(int flags) {
    mSnapshot = allocSnapshot(mSnapshot, flags);
    return mSaveCount++;
}

int CanvasState::save(int flags) {
    return saveSnapshot(flags);
}
```

```
Snapshot* CanvasState::allocSnapshot(Snapshot* previous, int savecount) {
    void* memory;
    if (mSnapshotPool) {
        memory = mSnapshotPool;
        mSnapshotPool = mSnapshotPool->previous;
        mSnapshotPoolCount--;
    } else {
        memory = malloc(sizeof(Snapshot));
    }
    return new (memory) Snapshot(previous, savecount);
}
```

图 40 canvas.save 的逻辑

按照这种逻辑，实际 save 和 restore 也不需要作为一个 operation 存在了（之前 save 和 restore 的存在，主要是区分画布操作的作用域，现在画布操作直接跟绘制操作绑定了，那也不需要区分作用域了）。所以可以看到新的 canvas.save 逻辑中（图 40），实际就是把 mState.mSnapshot 深拷贝了一份。

所以对比之下的好处是，可以减少很多不必要的 operation，只留下了绘制操作，逻辑更聚焦；不过由于每个绘制 operation 都要独自保存当前 canvas 的 matrix，所以略微增加了内存消耗。

5. 软件绘制

前面我们分析了绘制的流程和 DisplayList 的概念，但是这些讨论都是基于硬件加速打开的情况下，下面我们看下非硬件加速的情况下逻辑是怎样的。

1）软件绘制

了解了硬件加速情况下的绘制流程，再来看非硬件加速的情况实际比较简单，软件绘制的入口函数在 ViewRootImpl.drawSoftware 函数，主要逻辑如下，

```
try {
    canvas.translate(-xoff, -yoff);
    if (mTranslator != null) {
        mTranslator.translateCanvas(canvas);
    }
    canvas.setScreenDensity(scalingRequired ? mNoncompatDensity : 0);
    attachInfo.mSetIgnoreDirtyState = false;

    mView.draw(canvas);

    drawAccessibilityFocusedDrawableIfNeeded(canvas);
} finally {
    if (!attachInfo.mSetIgnoreDirtyState) {
        // Only clear the flag if it was not set during the mView.draw() call
        attachInfo.mIgnoreDirtyState = false;
    }
}
finally {
  try {
    surface.unlockCanvasAndPost(canvas);
  } catch (IllegalArgumentException e) {
    Log.e(mTag, "Could not unlock surface", e);
    mLayoutRequested = true;    // ask wm for a new surface next time.
    //noinspection ReturnInsideFinallyBlock
    return false;
  }
```

图 41 ViewRootImpl.drawSoftware 函数的相关逻辑

可以看到，这里不是借助 RenderThread 对 surface 操作，而是直接从 Surface 上获取 canvas，然后在 canvas 上完成绘制和渲染，然后直接提交到 Surface 上。

类似的，我们先来看下绘制分发的逻辑，如图 41，这里直接调用的是 DecorView（mView）的 draw(1)函数，从前面的分析可以看到，draw(1)会调 dispatchDraw 和 onDraw，然后 dispatchDraw 又会调子 View 的 draw(3)函数。前面，我们也说过，硬件加速和非硬件加速的逻辑就汇聚在 draw(3)函数里，所以我们简单看下 draw(3)里跟非硬件加速有关的逻辑，

```
Bitmap cache = null;
int layerType = getLayerType(); // TODO: signify cache state with just 'cache' local
if (layerType == LAYER_TYPE_SOFTWARE || !drawingWithRenderNode) {
    if (layerType != LAYER_TYPE_NONE) {
        // If not drawing with RenderNode, treat HW layers as SW
        layerType = LAYER_TYPE_SOFTWARE;
        buildDrawingCache(true);
    }
    cache = getDrawingCache(true);
}
```

图 43 draw(3)中非硬件加速相关的逻辑

这里有一个非常关键的逻辑，就是软件绘制时缓存的逻辑，首先从判断条件可以看出有两个条件可以走进这个逻辑，其中 layerType == LAYER_TYPE_SOFTWARE 代表当前 View 开启了软件缓冲区，而!drawingWithDrawingCache 则表示当前是非硬件加速。(这里要注意非硬件加速和软件缓冲区是两个不同的概念，只是如果在硬件加速下，一个 View 开启了软件缓存区，那么它也会走软件绘制的逻辑，从而可以实现向下兼容)

我们继续来看下 getDrawingCache 中的逻辑，

```
public Bitmap getDrawingCache(boolean autoScale) {
    if ((mViewFlags & WILL_NOT_CACHE_DRAWING) == WILL_NOT_CACHE_DRAWING) {
        return null;
    }
    if ((mViewFlags & DRAWING_CACHE_ENABLED) == DRAWING_CACHE_ENABLED) {
        buildDrawingCache(autoScale);
    }
    return autoScale ? mDrawingCache : mUnscaledDrawingCache;
}
```

图 44 getDrawingCache 中的逻辑

这里有两个标志位，应用可以自行设置是否要使用缓存区，以及是否要开启缓存区（这两个判断默认都是 false）。所以结合图 43 的逻辑，这里可以组合出几种不同的逻辑分支，

类型 \ 状态	默认	WILL_NOT_CACHE_DRAWING	DRAWING_CACHE_ENABLED
非硬件加速	无	无	有
软件缓冲区	有	无	有

图 45 不同逻辑分支下缓存区的有无情况

从上面的表格可以看到在几种不同情况下，View 的缓存区有无情况；同时，也可以看出非硬件加速和软件缓冲区的差别——非硬件加速默认是不带 View 缓存的；而设置了 LAYER_TYPE_SOFTWARE，也就是开启了软件缓冲区，默认带有 View 缓存。(由于软件缓冲区和非硬件加速共用了大部分绘制逻辑，所以这里可以理解成，硬件加速 + 软件缓冲区 = 非硬件加速 + 缓冲区)

通过上面的分析，我们可以把软件绘制的逻辑大致分成两种分支，一种是无缓存的逻辑，一种是有缓存的逻辑。我们分别来看一下。

a. 无缓存绘制

在无缓存的情况下，根据图 43 种的逻辑，cache 为 null，那么之后在 draw(3)的逻辑如图 46，

```
if (!drawingWithDrawingCache) {
    if (drawingWithRenderNode) {
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;
        ((DisplayListCanvas) canvas).drawRenderNode(renderNode);
    } else {
        // Fast path for layouts with no backgrounds
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
            mPrivateFlags &= ~PFLAG_DIRTY_MASK;
            dispatchDraw(canvas);
        } else {
            draw(canvas);
        }
    }
```

图 46 draw(3)中非硬件加速相关的逻辑

可以看到，在 draw(3)中直接又根据是否需要绘制自身而调用了 draw(1)或 dispatchDraw，从而实现了向下的递归。

总结一下，主要跟硬件加速绘制有 3 点差异，

- 首先，没有通过 RenderThread 操作，而是直接往 Surface 上写内容；
- 没有通过 updateDisplayListIfNeeded 的这个函数，也就是说软件绘制默认不开启 View 缓存的情况下，完全没有缓存的概念（也无处可存），每一次绘制都需要完整重绘；
- 最后，前面分析硬件加速的绘制流程时，我们分析过，子 View 向 parent 上绘制内容，主要是通过把自己的 RenderNode 返回给 parent，然后 parent 通过 canvas.drawRenderNode 把子 View 绘制在自己的 canvas 上；而对于软件绘制，从头到尾使用的都是同一块 canvas（从 ViewRootImpl 上传递下来的）。

b. 有缓存绘制

```
public void buildDrawingCache(boolean autoScale) {
    if ((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0 || (autoScale ?
            mDrawingCache == null : mUnscaledDrawingCache == null)) {
        if (Trace.isTagEnabled(Trace.TRACE_TAG_VIEW)) {
            Trace.traceBegin(Trace.TRACE_TAG_VIEW,
                    "buildDrawingCache/SW Layer for " + getClass().getSimpleName());
        }
        try {
            buildDrawingCacheImpl(autoScale);
        } finally {
            Trace.traceEnd(Trace.TRACE_TAG_VIEW);
        }
    }
}
```

图 47 缓存绘制相关逻辑（1）

从图 47 可以看到，缓存逻辑也是由 PFLA_DRAWING_CACHE_VALID 控制的，这个变量在 invalidate 的流程中会修改，前面已经分析过了，这里就不再讨论了。

```
try {
    bitmap = Bitmap.createBitmap(mResources.getDisplayMetrics(),
            width, height, quality);
    bitmap.setDensity(getResources().getDisplayMetrics().densityDpi);
    if (autoScale) {
        mDrawingCache = bitmap;
    } else {
        mUnscaledDrawingCache = bitmap;
    }
    if (opaque && use32BitCache) bitmap.setHasAlpha(false);
Canvas canvas;
if (attachInfo != null) {
    canvas = attachInfo.mCanvas;
    if (canvas == null) {
        canvas = new Canvas();
    }
    canvas.setBitmap(bitmap);
    // Temporarily clobber the cached Canvas in case one of our children
    // is also using a drawing cache. Without this, the children would
    // steal the canvas by attaching their own bitmap to it and bad, bad
    // thing would happen (invisible views, corrupted drawings, etc.)
    attachInfo.mCanvas = null;
} else {
    // This case should hopefully never or seldom happen
    canvas = new Canvas(bitmap);
}
```

```
computeScroll();
final int restoreCount = canvas.save();

if (autoScale && scalingRequired) {
    final float scale = attachInfo.mApplicationScale;
    canvas.scale(scale, scale);
}

canvas.translate(-mScrollX, -mScrollY);

mPrivateFlags |= PFLAG_DRAWN;
if (mAttachInfo == null || !mAttachInfo.mHardwareAccelerated ||
        mLayerType != LAYER_TYPE_NONE) {
    mPrivateFlags |= PFLAG_DRAWING_CACHE_VALID;
}

// Fast path for layouts with no backgrounds
if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
    mPrivateFlags &= ~PFLAG_DIRTY_MASK;
    dispatchDraw(canvas);
    if (mOverlay != null && !mOverlay.isEmpty()) {
        mOverlay.getOverlayView().draw(canvas);
    }
} else {
    draw(canvas);
}
```

图 48 缓存创建的逻辑

创建缓存的逻辑如上，主要就是创建了一个 Bitmap，然后借助 canvas 调用 dispatchDraw 或 draw(1)把 View 绘在这块 Bitmap 上。

```
} else if (cache != null) {
    mPrivateFlags &= ~PFLAG_DIRTY_MASK;
    if (layerType == LAYER_TYPE_NONE || mLayerPaint == null) {
        // no layer paint, use temporary paint to draw bitmap
        Paint cachePaint = parent.mCachePaint;
        if (cachePaint == null) {
            cachePaint = new Paint();
            cachePaint.setDither(false);
            parent.mCachePaint = cachePaint;
        }
        cachePaint.setAlpha((int) (alpha * 255));
        canvas.drawBitmap(cache, 0.0f, 0.0f, cachePaint);
    } else {
        // use layer paint to draw the bitmap, merging the two alphas, but also restore
        int layerPaintAlpha = mLayerPaint.getAlpha();
        if (alpha < 1) {
            mLayerPaint.setAlpha((int) (alpha * layerPaintAlpha));
        }
        canvas.drawBitmap(cache, 0.0f, 0.0f, mLayerPaint);
        if (alpha < 1) {
            mLayerPaint.setAlpha(layerPaintAlpha);
        }
    }
}
```

图 49 缓存绘制相关逻辑（2）

之后的逻辑也比较简单，如果有缓存（如果缓存失效了，在前面的流程里就会直接重建一份缓存，所以走到这里一般缓存都不会是 null）的话，就会把 cache 对应的 Bitmap 绘在 canvas 上。（从图 49 可以看到，是软件缓冲区的话逻辑会有点细微的差异，软件缓冲区会有一个固定的 paint，这里就不细讨论了）

最后，对比纯粹的软件绘制，有缓存的情况主要是在缓存有效（简单说就是没有调用过 invalidate）的情况下，不需要重复对 View 进行绘制，可以直接使用保存在 bitmap 中的数据。

2）绘制操作

然后，我们再来对比一下具体的 canvas 操作，这里还是以 canvas.scale 和

canvas.drawBitmap 为例。在非硬件加速绘制的情况下，上层使用的依然是 Canvas.java 对象，但是底层对应的却是 SkCanvas.cpp 了，绘制操作的逻辑如下，

```
void SkCanvas::scale(SkScalar sx, SkScalar sy) {
    SkMatrix m;
    m.setScale(sx, sy);
    this->concat(m);
}
```

```
void SkCanvas::concat(const SkMatrix& matrix) {
    if (matrix.isIdentity()) {
        return;
    }

    this->checkForDeferredSave();
    fDeviceCMDirty = true;
    fCachedLocalClipBoundsDirty = true;
    fMCRec->fMatrix.preConcat(matrix);

    this->didConcat(matrix);
}
```

图 50 canvas.scale 的相关逻辑

可以看到，这里没有 operation 的概念，而是直接对 canvas 做了操作，这里 didConcat 函数，直接是往 Surface 上写数据了。

```
void SkCanvas::drawBitmap(const SkBitmap& bitmap, SkScalar dx, SkScalar dy, const SkPaint* paint) {
    if (bitmap.drawsNothing()) {
        return;
    }
    this->onDrawBitmap(bitmap, dx, dy, paint);
}
```

```
bool drawAsSprite = bounds && this->canDrawBitmapAsSprite(x, y, bitmap.width(), bitmap.height(),
                                                          *paint);
if (drawAsSprite && paint->getImageFilter()) {
    // Until imagefilters are updated, they cannot handle any src type but N32...
    if (bitmap.info().colorType() != kN32_SkColorType || bitmap.info().isSRGB()) {
        drawAsSprite = false;
    }
}

LOOPER_BEGIN_DRAWBITMAP(*paint, drawAsSprite, bounds)

while (iter.next()) {
    const SkPaint& pnt = looper.paint();
    if (drawAsSprite && pnt.getImageFilter()) {
        SkPoint pt;
        iter.fMatrix->mapXY(x, y, &pt);
        iter.fDevice->drawBitmapAsSprite(iter, bitmap,
                                         SkScalarRoundToInt(pt.fX),
                                         SkScalarRoundToInt(pt.fY), pnt);
    } else {
        iter.fDevice->drawBitmap(iter, bitmap, matrix, looper.paint());
    }
}

LOOPER_END
```

```
void SkBaseDevice::drawBitmapAsSprite(const SkDraw& draw, const SkBitmap& bitmap, int x, int y,
                                      const SkPaint& paint) {
    SkImageFilter* filter = paint.getImageFilter();
    if (filter && !this->canHandleImageFilter(filter)) {
        SkImageFilter::DeviceProxy proxy(this);
        SkBitmap dst;
        SkIPoint offset = SkIPoint::Make(0, 0);
        SkMatrix matrix = *draw.fMatrix;
        matrix.postTranslate(SkIntToScalar(-x), SkIntToScalar(-y));
        const SkIRect clipBounds = draw.fClip->getBounds().makeOffset(-x, -y);
        SkAutoTUnref<SkImageFilter::Cache> cache(this->getImageFilterCache());
        SkImageFilter::Context ctx(matrix, clipBounds, cache.get());
        if (filter->filterImageDeprecated(&proxy, bitmap, ctx, &dst, &offset)) {
            SkPaint tmpUnfiltered(paint);
            tmpUnfiltered.setImageFilter(nullptr);
            this->drawSprite(draw, dst, x + offset.x(), y + offset.y(), tmpUnfiltered);
        }
    } else {
        this->drawSprite(draw, bitmap, x, y, paint);
    }
}
```

图 51 canvas.drawBitmap 的相关逻辑

同样的，也是直接向 Surface 写数据，对比硬件加速，同样有两点差异，

- 没有 operation 的概念，不是异步操作，是直接向底层写数据；
- 没有区分绘制和渲染的概念，是在一起操作的。

通过上述对比，就可以看出硬件加速做了哪些优化。不过这里要注意的是，上述讨论的逻辑特指非硬件加速的情况，对于在硬件加速 + 开启软件缓冲区的情况，这部分逻辑依然跟硬件加速下的逻辑保持一致。