ardwar

HenCoder Android
自定义 View 1-8 硬件加速

elerat

硬件加速这个词每当被提及，很多人都会感兴趣。这个词给大部分人的概念大致有两个：快速、不稳定。对很多人来说，硬件加速似乎是一个只可远观而不可亵玩的高端科技：是，我听说它很牛逼，但我不敢「乱」用，因为我怕 hold 不住。

今天我试着就把硬件加速的外衣脱掉（并没有），聊一聊它的原理和应用：

1. 硬件加速的本质和原理；
2. 硬件加速在 Android 中的应用；
3. 硬件加速在 Android 中的限制。

> 本篇是 「HenCoder Android 开发进阶」自定义 View 部分的最后一篇：硬件加速。

概念

在正式开始之前需要说明一下，作为绘制部分的最后一期，本期内容只是为了内容的完整性做一个补充，因为之前好几期的内容里都有涉及硬件加速的技术点，而一些读者因为不了解硬件加速而产生了一些疑问。所以仅仅从难度上来讲，这期的内容并不难，并且本期的大部分内容你都可以从这两个页面中找到：

1. Hardware Acceleration | Android Developers
2. Google I/O 2011: Accelerated Android Rendering

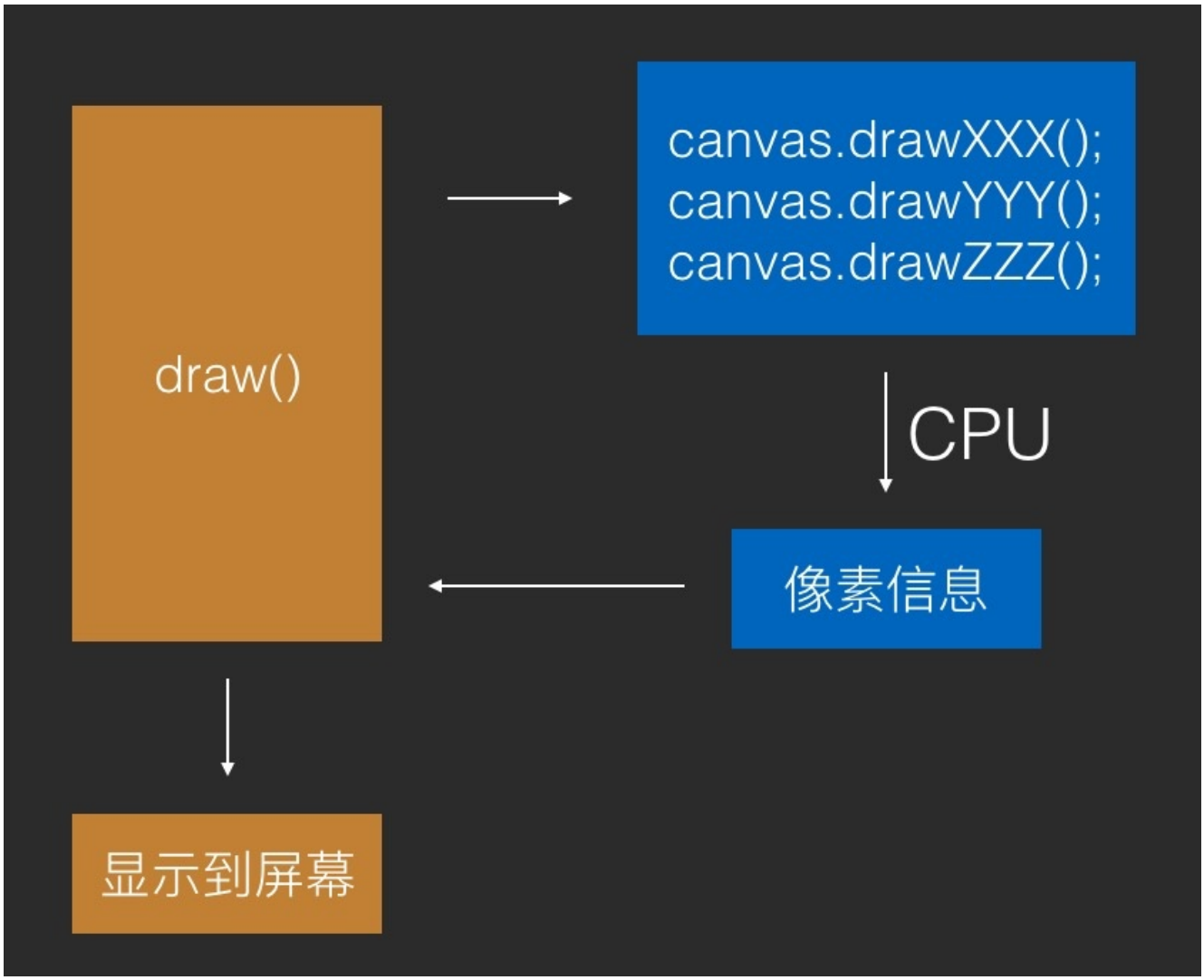
下面进入正题。

所谓硬件加速，指的是把某些计算工作交给专门的硬件来做，而不是和普通的计算工作一样交给 CPU 来处理。这样不仅减轻了 CPU 的压力，而且由于有了「专人」的处理，这份计算工作的速度也被加快了。这就是「硬件加速」。

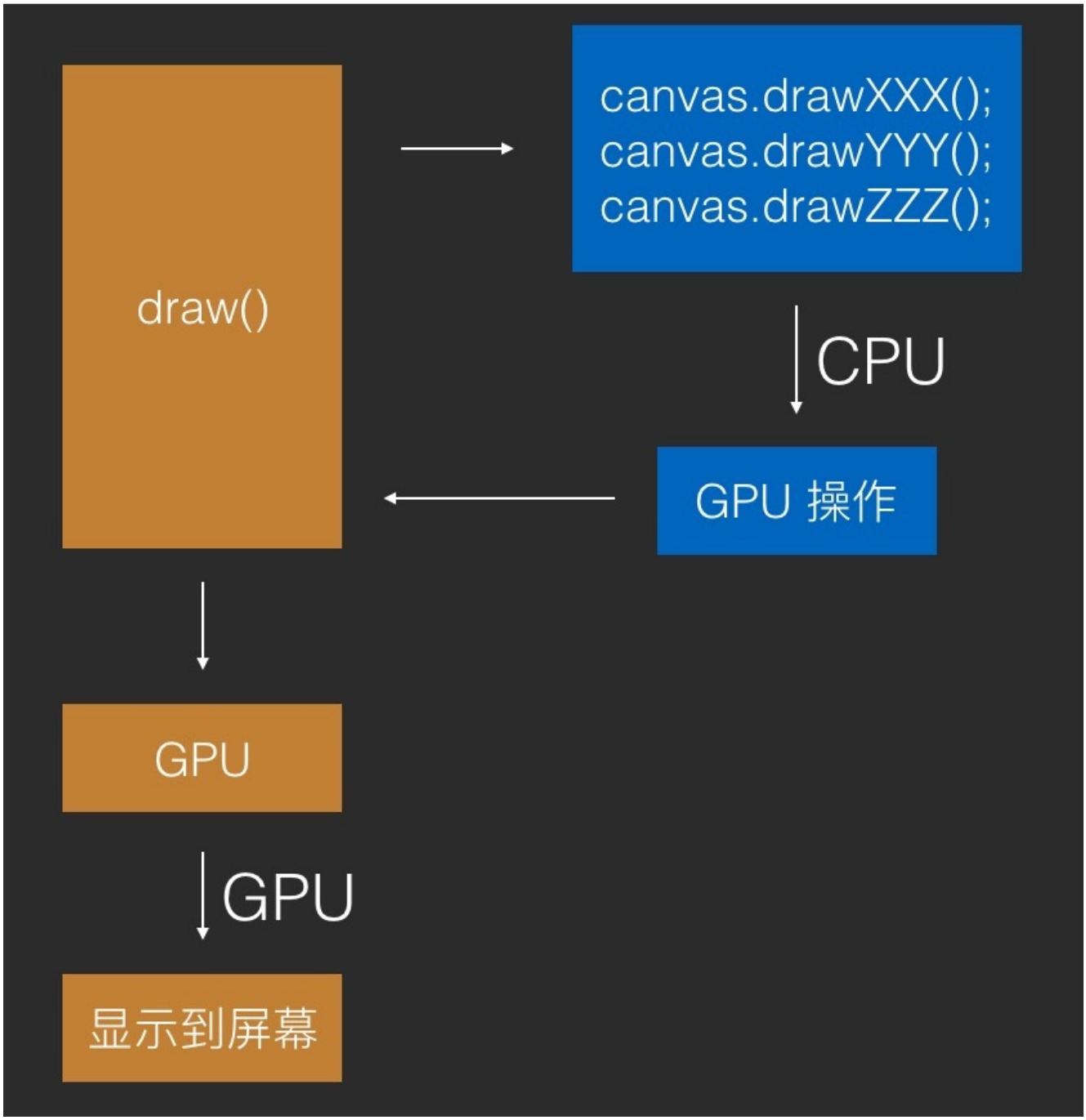
而对于 Android 来说，硬件加速有它专属的意思：在 Android 里，硬件加速专指把 View 中绘制的计算工作交给 GPU 来处理。进一步地再明确一下，这个「绘制的计算工作」指的就是把绘制方法中的那些 `Canvas.drawXXX()` 变成实际的像素这件事。

原理

在硬件加速关闭的时候， `Canvas` 绘制的工作方式是：把要绘制的内容写进一个 `Bitmap`，然后在之后的渲染过程中，这个 `Bitmap` 的像素内容被直接用于渲染到屏幕。这种绘制方式的主要计算工作在于把绘制操作转换为像素的过程（例如由一句 `Canvas.drawCircle()` 来获得一个具体的圆的像素信息），这个过程的计算是由 CPU 来完成的。大致就像这样：



而在硬件加速开启时， `Canvas` 的工作方式改变了：它只是把绘制的内容转换为 GPU 的操作保存了下来，然后就把它交给 GPU，最终由 GPU 来完成实际的显示工作。大致是这样：



> 如图，在硬件加速开启时，CPU 做的事只是把绘制工作转换成 GPU 的操作，这个工作量相对来说是非常小的。

怎么就「加速」了?

从上面的图中可以看出，硬件加速开启后，绘制的计算工作由 CPU 转交给了 GPU。不过这怎么就能起到「加速」作用，让绘制变快了呢?

硬件加速能够让绘制变快，主要有三个原因:

1. 本来由 CPU 自己来做的事，分摊给了 GPU 一部分，自然可以提高效率；
2. 相对于 CPU 来说，GPU 自身的设计本来就对于很多常见类型内容的计算（例如简单的圆形、简单的方形）具有优势；
3. 由于绘制流程的不同，硬件加速在界面内容发生重绘的时候绘制流程可以得到优化，避免了一些重复操作，从而大幅提升绘制效率。

其中前两点可以总结为一句：**用了 GPU，绘制就是快。**原因很直观，不再多说。

关于第三点，它的原理我大致说一下：

前面说到，在硬件加速关闭时，绘制内容会被 CPU 转换成实际的像素，然后直接渲染到屏幕。具体来说，这个「实际的像素」，它是由 `Bitmap` 来承载的。在界面中的某个 View 由于内容发生改变而调用 `invalidate()` 方法时，如果没有开启硬件加速，那么为了正确计算 `Bitmap` 的像素，这个 `View` 的父 View、父 View 的父 View 乃至一直向上直到最顶级 View，以及所有和它相交的兄弟 `View`，都需要被调用 `invalidate()` 来重绘。一个 View 的改变使得大半个界面甚至整个界面都重绘一遍，这个工作量是非常大的。

而在硬件加速开启时，前面说过，绘制的内容会被转换成 GPU 的操作保存下来（承载的形式称为 display list，对应的类也叫做 `DisplayList`），再转交给 GPU。由于所有的绘制内容都没有变成最终的像素，所以它们之间是相互独立的，那么在界面内容发生改变的时候，只要把发生了改变的 View 调用 `invalidate()` 方法以更新它所对应的 GPU 操作就好，至于它的父 View 和兄弟 View，只需要保持原样。那么这个工作量就很小了。

正是由于上面的原因，硬件加速不仅是由于 GPU 的引入而提高了绘制效率，还**由于绘制机制的改变，而极大地提高了界面内容改变时的刷新效率。**

所以把上面的三条压缩总结一下，硬件加速更快的原因有两条：

1. 用了 GPU，绘制变快了；
2. 绘制机制的改变，导致界面内容改变时的刷新效率极大提高。

限制

如果仅仅是这样，硬件加速只有好处没有缺陷，那大家都不必关心硬件加速了，这篇文章也不会出现：既然是好东西就用呗，关心那么多原理干吗？

可事实就是，硬件加速不只是好处，也有它的限制：受到 GPU 绘制方式的限制， `Canvas` 的有些方法在硬件加速开启式会失效或无法正常工作。比如，在硬件加速开启时， `clipPath()` 在 API 18 及以上的系统中才有效。具体的 API 限制和 API 版本的关系如下图：

	First supported API level
Canvas	
drawBitmapMesh() (colors array)	18
drawPicture()	23
drawPosText()	16
drawTextOnPath()	16
drawVertices()	✗
setDrawFilter()	16
clipPath()	18
clipRegion()	18
clipRect(Region.Op.XOR)	18
clipRect(Region.Op.Difference)	18
clipRect(Region.Op.ReverseDifference)	18
clipRect() with rotation/perspective	18
Paint	
setAntiAlias() (for text)	18
setAntiAlias() (for lines)	16
setFilterBitmap()	17
setLinearText()	✗
setMaskFilter()	✗
setPathEffect() (for lines)	✗
setRasterizer()	✗
setShadowLayer() (other than text)	✗
setStrokeCap() (for lines)	18
setStrokeCap() (for points)	19
setSubpixelText()	✗
Xfermode	
PorterDuff.Mode.DARKEN (framebuffer)	✗
PorterDuff.Mode.LIGHTEN (framebuffer)	✗
PorterDuff.Mode.OVERLAY (framebuffer)	✗
Shader	
ComposeShader inside ComposeShader	✗
Same type shaders inside ComposeShader	✗
Local matrix on ComposeShader	18

所以，如果你的自定义控件中有自定义绘制的内容，最好参照一下这份表格，确保你的绘制操作可以正确地在所有用户的手机里能够正常显示，而不是只在你的运行了最新版本 Android 系统的 Nexus 或 Pixel 里测试一遍没问题就发布了。小心被祭天。

不过有一点可以放心的是，所有的原生自带控件，都没有用到 API 版本不兼容的绘制操作，可以放心使用。所以你只要检查你写的自定义绘制就好。

View Layer

在之前几期的内容里我提到过几次，如果你的绘制操作不支持硬件加速，你需要手动关闭硬件加速来绘制界面，关闭的方式是通过这行代码：

```
view.setLayerType(LAYER_TYPE_SOFTWARE, null);
```

有不少人都有过疑问：什么是 layer type？如果这个方法是硬件加速的开关，那么它的参数为什么不是一个 `LAYER_TYPE_SOFTWARE` 来关闭硬件加速以及一个 `LAYER_TYPE_HARDWARE` 来打开硬件加速这么两个参数，而是三个参数，在 `SOFTWARE` 和 `HARDWARE` 之外还有一个 `LAYER_TYPE_NONE`？难道还能既不用软件绘制，也不用硬件绘制吗？

事实上，这个方法的本来作用并不是用来开关硬件加速的，只是当它的参数为 `LAYER_TYPE_SOFTWARE` 的时候，可以「顺便」把硬件加速关掉而已；并且除了这个方法之外，Android 并没有提供专门的 View 级别的硬件加速开关，所以它就「顺便」成了一个开关硬件加速的方法。

`setLayerType()` 这个方法，它的作用其实就是名字里的意思：设置 View Layer 的类型。所谓 View Layer，又称为离屏缓冲（Off-screen Buffer），它的作用是单独启用一块地方来绘制这个 View，而不是使用软件绘制的 Bitmap 或者通过硬件加速的 GPU。这块「地方」可能是一块单独的 `Bitmap`，也可能是一块 OpenGL 的纹理（texture，OpenGL 的纹理可以简单理解为图像的意思），具体取决于硬件加速是否开启。采用什么来绘制 View 不是关键，关键在于当设置了 View Layer 的时候，它的绘制会被缓存下来，而且缓存的是最终的绘制结果，而不是像硬件加速那样只是把 GPU 的操作保存下来再交给 GPU 去计算。通过这样更进一步的缓存方式，View 的重绘效率进一步提高了：只要绘制的内容没有变，那么不论是 CPU 绘制还是 GPU 绘制，它们都不用重新计算，而只要只用之前缓存的绘制结果就可以了。

> 多说一句，其实这个离屏缓冲（Off-screen Buffer），更准确的说应该叫做离屏缓存（Off-screen Cache）会更合适一点。原因在上面这一段里已经说过了，因为它其实是缓存而不是缓冲。（这段话仅代表个人意见）

基于这样的原理，在进行移动、旋转等（无需调用 `invalidate()`）的属性动画的时候开启 Hardware Layer 将会极大地提升动画的效率，因为在动画过程中 View 本身并没有发生改变，只是它的位置或角度改变了，而这种改变是可以由 GPU 通过简单计算就完成的，并不需要重绘整个 View。所以在这种动画的过程中开启 Hardware Layer，可以让本来就依靠硬件加速而变流畅了的动画变得更加流畅。实现方式大概是这样：

```
view.setLayerType(LAYER_TYPE_HARDWARE, null);
ObjectAnimator animator = ObjectAnimator.ofFloat(view, "rotationY",

animator.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        view.setLayerType(LAYER_TYPE_NONE, null);
    }
});

animator.start();
```

或者如果是使用 `ViewPropertyAnimator`，那么更简单：

```
view.animate()
        .rotationY(90)
        .withLayer(); // withLayer() 可以自动完成上面这段代码的复杂操作
```

不过一定要注意，只有你在对 `translationX` `translationY` `rotation` `alpha` 等无需调用 `invalidate()` 的属性做动画的时候，这种方法才适用，因为这种方法本身利用的就是当界面不发生时，缓存未更新所带来的时间的节省。所以简单地说——

这种方式不适用于基于自定义属性绘制的动画。一定记得这句话。

另外，除了用于关闭硬件加速和辅助属性动画这两项功能外，Layer 还可以用于给 View 增加一些绘制效果，例如设置一个 `ColorMatrixColorFilter` 来让 View 变成黑白的：

```
ColorMatrix colorMatrix = new ColorMatrix();
colorMatrix.setSaturation(0);


Paint paint = new Paint();
paint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));

view.setLayerType(LAYER_TYPE_HARDWARE, paint);
```

另外，由于设置了 View Layer 后，View 在初次绘制时以及每次 `invalidate()` 后重绘时，需要进行两次的绘制工作（一次绘制到 Layer，一次从 Layer 绘制到显示屏），所以其实它的每次绘制的效率是被降低了的。所以一定要慎重使用 View Layer，在需要用到它的时候再去使用。

总结

本期内容就到这里，就像开头处我说的，本期只是作为一个完整性的补充，并没有太多重要或高难度的东西，我也没有准备视频或太多的截图或动图来做说明。惯例总结一下：

硬件加速指的是使用 GPU 来完成绘制的计算工作，代替 CPU。它从工作分摊和绘制机制优化这两个角度提升了绘制的速度。

硬件加速可以使用 `setLayerType()` 来关闭硬件加速，但这个方法其实是用来设置 View Layer 的：

1. 参数为 `LAYER_TYPE_SOFTWARE` 时，使用软件来绘制 View Layer，绘制到一个 Bitmap，并顺便关闭硬件加速；
2. 参数为 `LAYER_TYPE_HARDWARE` 时，使用 GPU 来绘制 View Layer，绘制到一个 OpenGL texture（如果硬件加速关闭，那么行为和 `VIEW_TYPE_SOFTWARE` 一致）；
3. 参数为 `LAYER_TYPE_NONE` 时，关闭 View Layer。

View Layer 可以加速无 `invalidate()` 时的刷新效率，但对于需要调用 `invalidate()` 的刷新无法加速。

View Layer 绘制所消耗的实际时间是比不使用 View Layer 时要高的，所以要慎重使用。