00.来源

- CircleImageView用于显示圆形的头像以及头像按压效果，是自定义View。
- 要想理解CircleImageView，我们需要理解自定义View的过程, 自定义控件你需要以下的步骤。
    - 创建自定义View，并获取自定义属性。
    - 内部处理View的测量
    - 绘制View
    - 与用户进行交互
    - 优化已定义的View
- 上面列出的五项就是android官方给出的自定义控件的步骤。根据你的需要，某些步骤可以省略。
- ok，接下来我们来具体介绍。

01.创建自定义View，并获取自定义属性。

- 分为两个部分，代码侧和xml侧。
- 首先代码侧：我们要自定义一个View的代码，它要继承自某个已有的View父类，他需要有成员变量来描述自身的自定义属性，同时在代码中可以解析xml文件去设置自身的属性。最好还可以暴露某些属性让外部可以在代码运行过程中动态的访问和修改它的属性。
- 然后xml侧：我们要定义xml中可以配置它的哪些自定义属性，接着我们在xml中配置它的自定义属性。

1.1 创建自定义View类（代码侧）

- 继承自已有的View类，覆写构造方法。接收context和AttributeSet参数，内部调用init设置自身属性。

```
class CircleImageView : AppCompatImageView {
    constructor(context: Context) : super(context) {
        init(context, null)
    }

    constructor(context: Context, attrs: AttributeSet?) : super(context,
attrs) {
        init(context, attrs)
    }

    constructor(context: Context, attrs: AttributeSet?, defStyleAttr:
Int) : super(context, attrs, defStyleAttr) {
        init(context, attrs)
    }
}
```

- 它要有某些属性值，比如说：shapeType形状。

```
// rectangle or round, 1 is circle, 2 is rectangle
private var shapeType = 0
```

- 然后我们要有方法在类的创建过程中获取自定义属性
  - 这里是init方法，也就是构造函数中调用的，其传入context和attrs。
  - 然后在init方法中，首先可以为属性直接设置初始值，也就是没在xml文件中配置的时候给出默认值。然后可以调用context.obtainStyledAttributes()方法，**从 attrs，XML 属性集合中提取对应的CircleImageView的属性值容器array。**
  - 再通过array.getColor()等方法获取对应的属性设置给自身属性成员变量
  - ```
    在init方法中
    //init the value
        borderWidth = 0
        borderColor = -0x22000001
        pressAlpha = 0x42
        pressColor = 0x42000000
        radius = 16
        shapeType = 0

    // get attribute of EaseImageView
        if (attrs != null) {
            val array = context.obtainStyledAttributes(attrs,
    R.styleable.CircleImageView)
            borderColor =
    array.getColor(R.styleable.CircleImageView_ease_border_color,
    borderColor)
            borderWidth =
    array.getDimensionPixelOffset(R.styleable.CircleImageView_ease_border_wi
    dth, borderWidth)
            pressAlpha =
    array.getInteger(R.styleable.CircleImageView_ease_press_alpha,
    pressAlpha)
            pressColor =
    array.getColor(R.styleable.CircleImageView_ease_press_color, pressColor)
    ```
    ```

```
            radius =
array.getDimensionPixelOffset(R.styleable.CircleImageView_ease_radius,
radius)
            shapeType =
array.getInteger(R.styleable.CircleImageView_es_shape_type, shapeType)
            array.recycle()
        }
```

- getter，setter方法暴露某些属性，让外部可以在代码运行过程中动态的访问和修改它的属性。
    - ```
      fun setBorderColor(borderColor: Int) {
          this.borderColor = borderColor
          invalidate()
      }
      ```

1.2 定义自定义属性（xml侧）

- 我们要定义xml中可以配置它的哪些自定义属性，自定义属性通常写在在res/values/attrs.xml文件中 下面是自定义属性的标准写法
    - 首先设置declare-styleable name，这对应自定义View类的类名。表示这个类有哪些可以自定义的属性。
    - 然后设置attr name，属性，以及属性的format格式。
    - ```
      <!--圆角或者圆形图片组件-->
          <declare-styleable name="CircleImageView">
              <attr name="ease_border_color" format="color" />
              <attr name="ease_border_width" format="dimension" />
              <attr name="ease_press_alpha" format="integer" />
              <attr name="ease_press_color" format="color" />
              <attr name="ease_radius" format="dimension" />
              <attr name="es_shape_type" format="enum">
                  <enum name="none" value="0" />
                  <enum name="round" value="1" />
                  <enum name="rectangle" value="2" />
              </attr>
          </declare-styleable>
      ```

- 在xml中使用这些自定义属性。

```
<com.bytedance.tiktok.view.CircleImageView
    android:id="@+id/ivHead"
    android:layout_width="52dp"
    android:layout_height="52dp"
    app:es_shape_type="round"
    android:scaleType="centerCrop"/>
```

02.OnMeasure

- OnMeasure我们没有复写。因为我们是固定大小的，直接使用父类的测量逻辑足够了，getSize(MeasureSpec)，然后SetMeasureDimmision足够了，不用我们去根据padding去获取我们的desiredSize了。

03.OnDraw

3.1 整体流程

- 在onDraw阶段绘制图像、边框和按压效果
- 首先是获取并判断形状，如果是原始形状，采用系统绘制

```
// rectangle or round, 1 is circle, 2 is rectangle
    private var shapeType = 0

//获取用户定义的shapeType
    private fun init(context: Context, attrs: AttributeSet?)
    val array = context.obtainStyledAttributes(attrs,
R.styleable.CircleImageView)
    shapeType =
array.getInteger(R.styleable.CircleImageView_es_shape_type, shapeType)


/**
 * 用户没有提要求，采用原始形状，那么直接使用系统绘制
 */
    if (shapeType == 0) {
        super.onDraw(canvas)
        return
    }
```

- 不是原始形状，我们要自定义绘制了，自定义绘制我们要先将传递进来的drawable对象通过getBitmapFromDrawable转换为bitmap

```
val drawable = drawable ?: return

val bitmap = getBitmapFromDrawable(drawable)
```

> 为什么是把drawable转化为位图，drawable是什么?
>
> Drawable 是 Android 中表示“可绘制图形”的抽象类，这里的drawable是输入的用户头像图片，也就是ImageView要绘制的图片。
> 他有多种来源，包括XML（如矢量图、形状）和本地图片（如PNG、JPEG）。我们在这里将各种类型的Drawable统一转换为Bitmap对象，方便后续操作，同时对比特位图的操作更高效，也支持无损缩放。

- 接着进行绘制三部曲：主内容绘制，交互态绘制，边框装饰绘制。也就是
  - `drawDrawable(canvas, bitmap)`进行主内容的绘制，也就是将用户头像绘制到画布上
  - drawPress(canvas)，进行交互态按压效果的绘制
  - drawBorder(canvas)，进行边框的绘制。

3.2 getBitmapFromDrawable

- 该方法目的是将多种来源类型的drawable转为统一的Bitmap对象
- 前置处理
  - 如果drawable为空，或者是bitmap的实例，那么直接返回。
  - ```
    //drawable为null，返回null
    if (drawable == null) {
        return null
    }
    /**
     * 如果drawable是BitmapDrawable的实例，那么直接返回其实例
     */
    if (drawable is BitmapDrawable) {
        return drawable.bitmap
    }
    ```
- 创建一个空白位图
  - 通过Bitmap.createBitmap创建一个空的bitmap
  - ```
    //通过Bitmap.createBitmap创建一个空白位图
    bitmap = Bitmap.createBitmap(width, height, BITMAP_CONFIG)
    ```
- 将位图绑定到画布上。`val canvas = Canvas(bitmap)`
- 设置 Drawable 的绘制区域为 Bitmap 的整个范围，然后调用drawable.draw，将自身数据绘制到Canvas上（即写入 Bitmap）
  - ```
    // 设置 Drawable 的绘制区域为 Bitmap 的整个范围
    drawable.setBounds(0, 0, canvas.width, canvas.height)

    // 将 Drawable 绘制到 Canvas（即写入 Bitmap）
    drawable.draw(canvas)
    ```
- 返回绘制好的位图 `return bitmap`

> 这里怎么将drawable内容绘制到bitmap上的?
>
> 通过 `Canvas(bitmap)` 将 `Bitmap` 绑定到画布上，后续对画布的绘制会最终修改 `Bitmap` 的像素数据。
>
> 调用 `Drawable` 的绘制方法，将其内容渲染到 `Bitmap`。

3.3 drawDrawable

- 调用`drawDrawable(canvas, bitmap)`进行主内容的绘制，最终会得到一个圆形的头像，但是我们传递进行的ImageView不是圆形的啊，它的bitmap也不是，所以我们要在里面实现裁剪。然后绘制。形状裁剪通过 **离屏缓冲（Offscreen Layer）** 与 **混合模式（Xfermode）** 实现，核心步骤分解如下：
  - 调用canvas.saveLayer，新建一个透明的临时画布（离屏缓冲），所有后续绘制操作在此进行。这是为了避免混合模式直接修改主画布内容，确保裁剪效果是独立的进行的。
  - 调用canvas.drawCircle，绘制一个白色圆形，用于当作蒙版，作为后续裁剪的“模具”
    ```

- 通过paint.xfermode = PorterDuffXfermode(PorterDuff.Mode.SRC_IN)，设置混合模式。混合模式是指：仅保留 **源图像（Bitmap）** 与 **目标图像（白色蒙版）** 重叠区域的像素，其它的都被裁剪掉。这样调用paint绘画之后，Bitmap 被裁剪为圆形。
- 缩放Bitmap，并进行最终的绘制。
  - 计算缩放比例 martix
  - 通过bitmap = Bitmap.createBitmap，传递bitmap和大小，以及matrix缩放比例，新建一个缩放后的bitmap
  - 调用canvas.drawBitmap(bitmap, 0f, 0f, paint)，将bitmap绘制到离屏图层中。
- 调用canvas.restore()进行图层合并，将离屏缓冲中的最终图像（裁剪后的 Bitmap）合并到主画布。此时离屏图层自动回收，无需手动管理

3.4 drawPress

- 实现按压效果，具体是绘制一个覆盖在原有内容上方的颜色为半透黑色的遮罩，初始为全透明，点击的时候会变为半透明。以此模拟按压效果。
- 我们首先在init中初始化按压效果画笔，设置完全透明，颜色为半透黑色。
- 然后调用canvas.drawCircle，设置圆心x坐标，y坐标，半径，传递按压效果的画笔。绘制
  - ```
    // 定义按压颜色（半透明黑色，alpha 值为 0x42 = 66/255 ≈ 26% 透明度）
    pressColor = 0x42000000

    // 在画布中心绘制圆形按压效果
    canvas.drawCircle(
        (width / 2).toFloat(),   // 圆心 X 坐标（控件水平中心）
        (height / 2).toFloat(),  // 圆心 Y 坐标（控件垂直中心）
        (width / 2 - 1).toFloat(), // 半径（控件宽度的一半减 1px）
        pressPaint!!             // 按压效果的画笔
    )
    ```

3.5 drawBorder

- 绘制圆形头像外围的黑色边框。
- 创建画笔，设置画笔中边框的粗细和颜色。
- 然后调用canvas.drawCircle，设置圆心x坐标，y坐标，半径，传递边框绘制的画笔。绘制
  - ```
    val paint = Paint()
    paint.strokeWidth = borderWidth.toFloat() // 边框粗细
    paint.style = Paint.Style.STROKE          // 仅绘制轮廓
    paint.color = borderColor                 // 边框颜色
    paint.isAntiAlias = true                  // 抗锯齿
    ```
  - ```
    canvas.drawCircle(
        (width / 2).toFloat(),
        (height / 2).toFloat(),
        ((width - borderWidth) / 2).toFloat(),
        paint
    )
    ```
    ```

04.OnTouchEvent

- 实现控件的按压状态反馈，当用户触摸控件时显示颜色为半透黑色的遮罩（pressPaint），松开后隐藏遮罩。
- 具体原理是覆写onTouchEvent，在里面监听点击事件，根据点击事件的类型改变pressPaint的alpha 透明度，然后触发重绘流程。
    - 按压时，让其透明度为1f，完全不透明，显示半透黑色的圆形遮罩。
    - 松下时，让其透明度为0f，也就是全透了。
- ```
  pressAlpha = array.getInteger(R.styleable.CircleImageView_ease_press_alpha, pressAlpha)

  override fun onTouchEvent(event: MotionEvent): Boolean {
          when (event.action) {
              MotionEvent.ACTION_DOWN -> {
                  pressPaint!!.alpha = pressAlpha
                  invalidate()
              }
              MotionEvent.ACTION_UP -> {
                  pressPaint!!.alpha = 0
                  invalidate()
              }
              MotionEvent.ACTION_MOVE -> {
              }
              else -> {
                  pressPaint!!.alpha = 0
                  invalidate()
              }
          }
          return super.onTouchEvent(event)
      }
  ```

05.优化

- 在上面的步骤结束之后，其实一个完善的自定义控件已经出来了。接下来你要做的只是确保自定义控件运行得流畅，官方的说法是：为了避免你的控件看得来迟缓，确保动画始终保持每秒60帧。
- 下面是优化建议：
    - 避免不必要的代码
    - 在onDraw()方法中不应该有会导致垃圾回收的代码。
    - 尽可能少让onDraw()方法调用，大多数onDraw()方法调用都是手动调用了invalidate()的结果，所以如果不是必须，不要调用invalidate()方法。
- 我们怎么优化上面呢？
    - 首先是画笔，我们在OnDraw中多次创建了画笔，我们要将其调整为成员变量而非局部变量。避免其频繁创建导致垃圾回收。
    - 然后是避免不必要的代码，我们绘制边框时，可以添加判断，当边框宽度大于0时，才进行绘制，避免无效的渲染。
- 至此，我们完成了整个的绘制流程。
  ```