第 3 章　自定义 View

本章导读

View 表现为显示在屏幕上的各种视图。View 类是 Android 中各种组件的基类，如 View 就是 ViewGroup 基类。Android 中的 UI 组件都是由 View 和 ViewGroup 组成的。自定义 View 对于一个 Android 开发者来说是必须掌握的知识点，也是 Android 开发进阶的必经之路。为什么要自定义 View 呢？主要是 Android 系统内置的 View 无法完全满足业务需求，因而需要针对业务需求编制 View。本章主要内容：（1）自定义 View 的分类；（2）自定义 View 的构造函数；（3）View 的绘制流程；（4）自定义 View 示例。

3.1　自定义 View 的分类

Android 自定义 View 主要分为两大类：自定义 View 和自定义 ViewGroup。

1．自定义 View

（1）继承特定 View。

一般用于扩展已有（特定）View 的功能，如 TextView、Button、EditText、ImageView 等。这种方法比较常见和容易实现，不需要自己支持 wrap_content 和 padding 等属性。

（2）继承 View。

主要用于实现一些不规则的效果，这种效果不方便通过布局的组合方式达到，往往需要静态或动态地显示一些不规则的图形，它需要通过绘制的方式来实现，即重写 onDraw()方法。采用这种方式需要自己支持 wrap_content 属性和 padding 属性，但不需要支持 margin 属性，因为 margin 属性是由父容器决定的。

2．自定义 ViewGroup

（1）继承特定的 ViewGroup。

拓展某种布局的方式。在原有 ViewGroup 基础上组合，可较容易实现。与自定义 ViewGroup 相比更加简单，但自由度不高。

（2）继承 ViewGroup。

主要用于实现自定义的布局，即除了 LinearLayout、RelativeLayout、FrameLayout 等系统布局，再重新定义一种新的布局。当某种效果看起来很像几种 View 组合在一起时，可以采用这种方式实现，但需要在处理 ViewGroup 的测量（measure）和布局（layout）这两个过程时，同时处理子元素（子 View）的测量（measure）和布局（layout），过程较为复杂。这就需要自定义支持 wrap_content 属性、padding 属性和 margin 属性，可更加接近 View 的底层。

3.2 自定义 View 的构造函数

以自定义 RoundRectView 为例，这里 RoundRectView 类继承自 View 类。

```
public class RoundRectView extends View { }
```

View 有四个构造器，其区别在于参数的数目和类型不同，一般使用前三个构造器。

1．只有一个 Context 参数的构造方法

```
public RoundRectView(Context context) {
    super(context);
    init();
}
```

如果 View 是在 Java 代码里面通过 new 关键字创建的，则调用第一个构造函数。通常在通过代码初始化控件时使用，即当在 Java 代码中直接通过 new 关键字创建这个控件时，就会调用这个方法。

2．两个参数（Context 上下文和 AttributeSet 属性集）的构造方法

```
public RoundRectView(Context context,  @Nullable AttributeSet attrs) {
    this(context, attrs, 0);
}
```

如果 View 是在.xml 中声明的，则会调用第二个构造函数。自定义属性是从 AttributeSet 参数传进来的。两个参数的构造方法通常对应布局文件中控件被映射成对象时调用（需要解析属性），即当需要在自定义控件中获取属性时，就默认调用这个构造方法。

AttributeSet 对象就是这个控件中定义的所有属性。可以通过 AttributeSet 对象的 getAttributeCount()方法获取属性的个数，通过 getAttributeName()方法获取到某条属性的名称，并通过 getAttributeValue()方法获取到某条属性的值。

注意，不管有没有使用自定义属性，都会默认调用这个构造方法。

3．三个参数（Context 上下文、AttributeSet 属性集和 defStyleAttr 自定义属性）的构造方法

```
public RoundRectView(Context context, @Nullable AttributeSet attrs, int defStyle
Attr) {
        super(context, attrs, defStyleAttr);
        TypedArray a = context.obtainStyledAttributes(attrs, R.styleable. RoundRect
View);
        mColor = a.getColor(R.styleable.RoundRectView_roundrect_color, Color.RED);
        a.recycle();
        init();
    }
```

这个构造方法不会默认调用，必须通过手动调用，它和两个参数的构造方法的唯一区别就是，这个构造方法传入了一个默认属性集。defStyleAttr 指向的是自定义属性的<declare-styleable>标签中定义的自定义属性集，在创建 TypedArray 对象时需要用到 defStyleAttr。

如果在 Code 中实例化一个 View 就会调用第一个构造函数，如果在 xml 中定义就会调用第二个构造函数，而第三个函数系统是不调用的，要由 View（自定义的或系统预定义的 View）显式调用，如这里在第二个构造函数中调用了第三个构造函数。

第三个参数的意义就如同它的名字，即默认的 Style。

3.3 View 的绘制流程

当一个应用启动时，会启动一个主 Activity，Android 系统会根据 Activity 的布局来对它进行绘制。绘制会从根视图 ViewRoot 的 performTraversals()方法开始，从上到下遍历整个视图树，每个 View 控件负责绘制自己，而 ViewGroup 还需要负责通知自己的子 View 进行绘制操作。视图绘制的过程可以分为三个步骤，分别是测量（Measure）、布局（Layout）和绘制（Draw）。View 的绘制基本由 measure()、layout()、draw()这个三个函数完成。

（1）measure()：用来测量 View 的宽和高，相关方法有 measure()、setMeasuredDimension()和 onMeasure()。

（2）layout()：用来确定 View 在父容器中的布局位置，相关方法有 layout()、onLayout()和 setFrame()。

（3）draw()：负责将 View 绘制在屏幕上，相关方法有 draw()和 onDraw()。

移动设备一般定义屏幕左上角为坐标原点，向右为 x 轴的增大方向，向下为 y 轴的增大方向。

3.3.1 Measure 过程

1. MeasureSpec

MeasureSpec 是 View 的内部类，它封装了一个 View 的尺寸，在 onMeasure()中会根据这个 MeasureSpec 值来确定 View 的宽和高。MeasureSpec 值保存在一个 32 位的 int 值中，前两位则表示模式 SpecMode，后 30 位则表示大小 SpecSize。

即 MeasureSpec = SpecMode + SpecSize。

在 MeasureSpec 当中一共存在三种 SpecMode：EXACTLY、AT_MOST 和 UNSPECIFIED。对于 View 来说，MeasureSpec 的 SpecMode 和 SpecSize 有如下意义。

（1）EXACTLY：精确测量模式。父容器已经检测出 View 所需要的精确大小，此时 View 的最终大小就是 SpecSize 所指定的值。它对应于 LayoutParams 中的 match_parent 和具体数值这两种模式。

（2）AT_MOST：最大值模式。View 的尺寸有一个最大值，View 不可以超过 MeasureSpec 当中的 SpecSize 值。它对应于 LayoutParams 中的 wrap_content。

（3）UNSPECIFIED：不指定测量模式。父容器不对 View 有任何限制，子视图可以是想要的任何尺寸，通常用于系统内部，但在应用开发中很少使用到。

获取测量模式（SpecMode）的示例代码如下：

```
int specMode = MeasureSpec.getMode(measureSpec);
```

获取测量大小（SpecModeSize）的示例代码如下：

```
int specSize = MeasureSpec.getSize(measureSpec);
```

通过 SpecModeMode 和 SpecModeSize 生成新的 MeasureSpec，其示例代码如下：

```
int measureSpec=MeasureSpec.makeMeasureSpec(size, mode);
```

子元素的 MeasureSpec 创建与父容器的 MeasureSpec 和子元素本身的 LayoutParams 有关。ViewGroup 的 getChildMeasureSpec()方法的主要代码如下：

```
    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) {
           //当父 View 要求一个精确值时，为子 View 赋值
```

```
            case MeasureSpec.EXACTLY:
                //如果子View有自己的尺寸，则使用自己的尺寸
                if (childDimension >= 0) {
                    resultSize = childDimension;
                    resultMode = MeasureSpec.EXACTLY;
                    //当子View是match_parent，将父View的大小赋值给子View
                } else if (childDimension == LayoutParams.MATCH_PARENT) {
                    resultSize = size;
                    resultMode = MeasureSpec.EXACTLY;
                    //如果子View是wrap_content，设置子View的最大尺寸为父View
                } else if (childDimension == LayoutParams.WRAP_CONTENT) {
                    resultSize = size;
                    resultMode = MeasureSpec.AT_MOST;
                }
                break;

            //父布局给子View一个最大界限
            case MeasureSpec.AT_MOST:
                if (childDimension >= 0) {
                    //如果子View有自己的尺寸，则使用自己的尺寸
                    resultSize = childDimension;
                    resultMode = MeasureSpec.EXACTLY;
                } else if (childDimension == LayoutParams.MATCH_PARENT) {
                    //父View的尺寸为子View的最大尺寸
                    resultSize = size;
                    resultMode = MeasureSpec.AT_MOST;
                } else if (childDimension == LayoutParams.WRAP_CONTENT) {
                    //父View的尺寸为子View的最大尺寸
                    resultSize = size;
                    resultMode = MeasureSpec.AT_MOST;
                }
                break;

            //父布局对子View没有做任何限制
            case MeasureSpec.UNSPECIFIED:
                if (childDimension >= 0) {
                    //如果子View有自己的尺寸，则使用自己的尺寸
                    resultSize = childDimension;
                    resultMode = MeasureSpec.EXACTLY;
                } else if (childDimension == LayoutParams.MATCH_PARENT) {
                    //因父布局没有对子View做出限制，当子View为MATCH_PARENT时则大小为0
                    resultSize = View.sUseZeroUnspecifiedMeasureSpec ? 0 : size;
                    resultMode = MeasureSpec.UNSPECIFIED;
                } else if (childDimension == LayoutParams.WRAP_CONTENT) {
                    //因父布局没有对子View做出限制，当子View为WRAP_CONTENT时则大小为0
                    resultSize = View.sUseZeroUnspecifiedMeasureSpec ? 0 : size;
                    resultMode = MeasureSpec.UNSPECIFIED;
                }
                break;
        }
        return MeasureSpec.makeMeasureSpec(resultSize, resultMode);
    }
```

小贴士：针对不同的父容器和 View 本身不同的 LayoutParams，View 可以有多种 MeasureSpec。

（1）当 View 采用固定宽和高时，不管父容器的 MeasureSpec 是什么，View 的 MeasureSpec 都是精确模式，并且其大小遵循 LayoutParams 中的大小。

（2）当 View 的宽和高是 match_parent 时，如果父容器的模式是精确模式，那么 View 也是精确模式，并且其大小是父容器的剩余空间；如果父容器的模式是最大值模式，那么 View 也是最大值模式并且其大小不会超过父容器的剩余空间。

（3）当 View 的宽和高是 wrap_content 时，不管父容器的模式是精确还是最大值，View 的模式总是最大值模式，并且其大小不能超过父容器的剩余空间。

2．onMeasure()

整个测量过程的入口位于 View 的 Measure 方法当中，该方法做了一些参数的初始化之后调用了 onMeasure 方法（Measure 方法是一个 final 类型的方法，子类不能重写此方法）。

```
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
       setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(),
widthMeasureSpec),
             getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
    }
```

其中，

（1）setMeasuredDimension(int measuredWidth, int measuredHeight)，该方法用来设置 View 的宽和高，在自定义 View 时会经常用到。

（2）getDefaultSize(int size, int measureSpec)，该方法用来获取 View 默认的宽和高，可结合源码来进行分析。

直接继承 View 的自定义控件需要重写 onMeasure 方法，并设置 wrap_content 时的自身大小，否则在布局中使用 wrap_content 就相当于使用 match_parent。

Measure 过程会因为布局的不同或者需求的不同而呈现不同的形式，使用时还是要根据业务场景来具体分析。

小贴士：ViewGroup 的测量（Measure）过程与 View 有些不同，其本身是继承自 View，它没有对 View 的 Measure 方法及 onMeasure 方法进行重写。ViewGroup 除了要测量自身的宽和高，还需要测量各个子 View 的大小，不同的布局，测量方式也都不同（可参考 LinearLayout 及 FrameLayout），所以没有办法统一设置。因此它提供了测量子 View 的 measureChildren()和 measureChild()的方法，以帮助对子 View 进行测量。通过阅读 measureChildren()及 measureChild()的源码可以知道，其大致流程就是遍历所有的子 View，然后再调用 View 的 measure()方法，让子 View 测量自身大小。

3.3.2　Layout 过程

布局（Layout）过程对于 View 来说，就是用来计算 View 的位置参数，而对于 ViewGroup 来说，除了要测量自身位置，还需要测量子 View 的位置。

layout()方法是整个 Layout 流程的入口。

```
    //这里的四个参数 l、t、r、b 分别代表 View 的左、上、右、下四个边界相对于其父 View 的距离。
    public void layout(int l, int t, int r, int b) {
       …
       int oldL = mLeft;
       int oldT = mTop;
       int oldB = mBottom;
       int oldR = mRight;
       //通过 setFrame 或 setOpticalFrame 方法确定 View 在父容器当中的位置。
```

```
        boolean changed = isLayoutModeOptical(mParent) ? setOpticalFrame(l, t, r, b) :
setFrame(l, t, r, b);
        //调用 onLayout 方法。onLayout 方法是一个空实现，不同的布局会有不同的实现。
        if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PFLAG_LAYOUT_
REQUIRED) {
            onLayout(changed, l, t, r, b);
        }
    }
```

由上述内容可知，在 layout()方法中已经通过 setOpticalFrame(l, t, r, b)或 setFrame(l, t, r, b)方法对 View 自身的位置进行了设置。所以，onLayout(changed, l, t, r, b)方法主要是 ViewGroup 对子 View 的位置进行计算。

3.3.3 Draw 过程

Draw 流程也就是 View 绘制到屏幕上的过程，整个流程的入口在 View 的 draw()方法之中，其过程可以分为 6 个步骤：

① 如果需要，绘制背景（drawBackground(canvas)）；

② 如果有必要，保存当前 canvas；

③ 绘制 View 的内容（onDraw(canvas)）；

④ 绘制子 View（dispatchDraw(canvas)）；

⑤ 如果有必要，绘制边缘、阴影等效果；

⑥ 绘制装饰，如滚动条等。

onDraw()方法用于绘制自己（自身 View），dispatchDraw()方法用于绘制子 View，它们的使用说明如下：

（1）protected void onDraw(Canvas canvas)。

绘制 View 的内容。该方法是一个空的实现，在各个业务中自行处理。

（2）protected void dispatchDraw(Canvas canvas)。

绘制子 View。该方法在 View 当中是一个空的实现，在各个业务中自行处理。

在 ViewGroup 中对 dispatchDraw 方法做了实现，主要是遍历子 View，并调用子类的 draw 方法，一般不需要重写该方法。

3.4 自定义 View 示例

3.4.1 实现一个基本的自定义 View

【示例】 实现一个基本的自定义 View。

启动 Android Studio，在 Ch3 工程中创建 RoundRectDemo 模块，并创建 MainActivity.java、activity_main.xml，以及自定义 View：RoundRectView.java。

RoundRectDemo 模块的 RoundRectView.java 主要代码如下：

```
 public class RoundRectView extends View {
    private int mColor = Color.RED;
    private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  //设置无锯齿
    public RoundRectView(Context context) {
        super(context);
        init();
    }
```

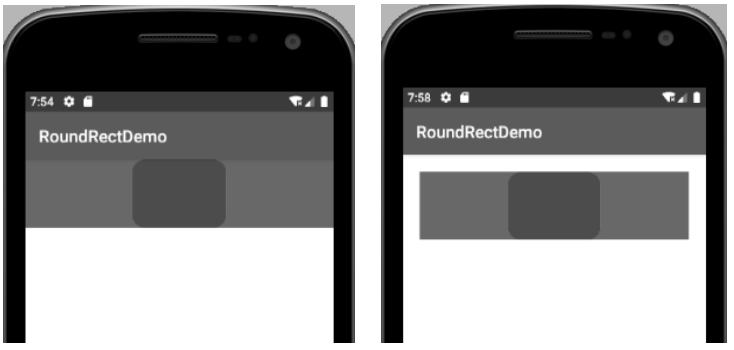
```
        public RoundRectView(Context context, AttributeSet attrs) {
            super(context, attrs);
            init();
        }
        public RoundRectView(Context context, AttributeSet attrs, int defStyleAttr)
{
            super(context, attrs, defStyleAttr);
            init();
        }
        private void init() {
            mPaint.setColor(mColor);
        }
        @Override
        protected void onDraw(Canvas canvas) {
            super.onDraw(canvas);
            int width = getWidth();
            int height = getHeight();
            int rectWidth = 220;
            int rectHeight = 160;
            RectF rectF = new RectF((width - rectWidth) / 2, (height - rectHeight) /
2, (width + rectWidth) / 2,
                    (height + rectHeight) / 2);
            canvas.drawRoundRect(rectF, 30, 30, mPaint);
        }
    }
```

RoundRectDemo 模块的 activity_main.xml 内容如下。

```
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#ffffff"
        android:orientation="vertical">
        <com.mialab.roundrectdemo.ui.RoundRectView
            android:id="@+id/roundRectView1"
            android:layout_width="wrap_content"
            android:layout_height="80dp"
            android:background="#6D6767" />
    </LinearLayout>
```

运行 RoundRectDemo 的界面如图 3-1（a）所示。在 ui.RoundRectView 中添加一行代码如下。

```
     android:layout_margin="20dp"
```



（a）　　（b）

图 3-1　实现一个基本的自定义 View

再重新运行 RoundRectDemo 模块，界面如图 3-1（b）所示。由于 margin 属性是由父容器控制的，所以 margin 属性生效了。

仍在 com.mialab.roundrectdemo.ui.RoundRectView 中添加一行 padding 属性的代码如下：

```
 android:padding = "10dp"
```

重新运行 RoundRectDemo 模块，界面仍如图 3-1（a）所示，没有什么变化。这说明 padding 属性未生效。

无论如图 3-1（a）所示，还是如图 3-1（b）所示，都证明了 wrap_content 属性亦未生效。

3.4.2 支持 wrap_content 属性和 padding 属性

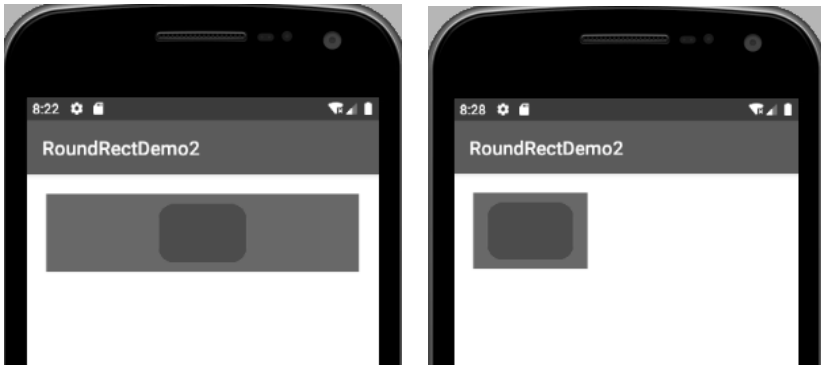
【示例】 支持 wrap_content 属性和 padding 属性。

启动 Android Studio，在 Ch3 工程中创建 RoundRectDemo2 模块，并创建 MainActivity.java、activity_main.xml、RoundRectView.java。

为了支持 padding 属性，RoundRectView.java 的 onDraw()方法代码改变如下。

```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    final int paddingLeft = getPaddingLeft();
    final int paddingRight = getPaddingRight();
    final int paddingTop = getPaddingTop();
    final int paddingBottom = getPaddingBottom();
    int width = getWidth() - paddingLeft - paddingRight;
    int height = getHeight() - paddingTop - paddingBottom;
    int rectWidth = 220;
    int rectHeight = 160;
    RectF rectF = new RectF(paddingLeft + (getWidth() - rectWidth) / 2,
          paddingTop + (getHeight() - rectHeight) / 2,
          (getWidth() + rectWidth) / 2 - paddingRight,
          (getHeight() + rectHeight) / 2 - paddingBottom);
    canvas.drawRoundRect(rectF, 30, 30, mPaint);
}
```

运行 RoundRectDemo2 模块的界面如图 3-2（a）所示，这说明 padding 属性生效了。



（a）　　（b）

图 3-2　支持 padding 和 wrap_content 属性

为了支持 wrap_content 属性，重写 RoundRectView.java 的 onMeasure()方法，其代码如下：

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);
```

```
        int widthSpecSize = MeasureSpec.getSize(widthMeasureSpec);
        int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);
        int heightSpecSize = MeasureSpec.getSize(heightMeasureSpec);
        if (widthSpecMode == MeasureSpec.AT_MOST&& heightSpecMode == MeasureSpec.
AT_MOST) {
            setMeasuredDimension(240, 160);
        } else if (widthSpecMode == MeasureSpec.AT_MOST) {
            setMeasuredDimension(240, heightSpecSize);
        } else if (heightSpecMode == MeasureSpec.AT_MOST) {
            setMeasuredDimension(widthSpecSize, 160);
        }
    }
```

重新运行 RoundRectDemo2 模块的界面如图 3-2（b）所示，这说明 wrap_content 属性生效了。

3.4.3 自定义属性

系统自带的 View 可以在 XML 中配置属性，对于写好的自定义 View 同样可以在 XML 中配置属性，为了使自定义的 View 的属性可以在 XML 中配置，一般需要以下 4 个步骤：

- 通过<declare-styleable>为自定义 View 添加属性；
- 在 XML 中为相应的属性声明属性值；
- 在运行时（一般为构造函数）获取属性值；
- 将获取到的属性值应用到 View。

【示例】 为自定义 View 提供自定义属性。

启动 Android Studio，在 Ch3 工程中创建 RoundRectDemo3 模块，并创建 MainActivity.java、activity_main.xml、RoundRectView.java。

（1）在 RoundRectDemo3 模块 res\values 中创建 attrs.xml 文件，其代码如下：

```
    <?xml version="1.0" encoding="utf-8"?>
    <resources>
        <declare-styleable  name = "RoundRectView">
            <attr name="roundrect_color"  format="color" />
        </declare-styleable>
    </resources>
```

（2）在 RoundRectDemo3 模块的 RoundRectView 类的构造方法中须解析自定义属性的值，并做相应处理。

```
    public RoundRectView(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }
    public RoundRectView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        TypedArray  a  =  context.obtainStyledAttributes(attrs,R.styleable.Round
RectView);
        mColor = a.getColor(R.styleable.RoundRectView_roundrect_color, Color.RED);
        a.recycle();
        init();
    }
```

context 通过调用 obtainStyledAttributes()方法来获取一个 TypeArray，然后由该 TypeArray 对属性进行设置。

```
    TypedArray  a  =  context.obtainStyledAttributes(attrs,  R.styleable.RoundRect
View);
```

调用结束后务必要用 recycle()方法释放资源。

（3）在 activity_main.xml 文件中使用自定义属性。

还须在布局文件 activity_main.xml 添加 schemas 声明。

```
xmlns:app = "http://schemas.android.com/apk/res-auto"
```

com.mialab.roundrectdemo3.ui.RoundRectView 中添加对于自定义属性的使用。

```
app:roundrect_color = "@color/color_blue"
```

运行 RoundRectDemo3 模块的界面如图 3-3 所示，这说明自定义属性生效了。

图 3-3　为 View 提供自定义属性

3.5　本章小结

本章主要介绍了自定义 View 的相关内容。首先概述了自定义 View 的分类和自定义 View 的构造函数，然后阐述了 View 的绘制流程，最后给出一个自定义 View 的示例。

习　题　3

1．自定义 View 的分类有哪些？

2．View 的绘制流程是怎样的？

3．如何自定义 View？试编程举例说明。

4．如何自定义 ViewGroup？试编程举例说明。

5．View 的事件传递机制是怎样的？可上网查找相关资料并加以说明。

6．ViewGroup 的事件传递机制是怎样的？可上网查找相关资料并加以说明。