|  |
| --- |
| 自定义控件步骤：1）定义declare-styleable，添加到attr.xml；使用TypedArray获取自定义属性：TypedArray typedArray = context.obtainStyledAttributes(attrs, R.styleable.CircularAttrsView);…………typedArray.recycle();//一个同步的对象池，注意回收。2）构造函数有三个参数的构造函数中第三个参数是默认的Style，调用init();初始化画笔、画布等.a:代码中直接实例化自定义控件，会调用第一个构造函数;b:xml布局使用自定义控件,会调用第二个构造函数;public MView(Context context) {this(context,null);}public MView(Context context, @Nullable AttributeSet attrs) {this(context, attrs,0);}3）onMeasure(int widthMeasureSpec, int heightMeasureSpec);计算绘制的尺寸、位置4）onLayout()设置视图在屏幕中显示的位置，onLayout一般只会在自定义ViewGroup中才会使用5）onDraw(Canvas canvas);进行绘制过程。如果自定义ViewGroup这个函数则不需要重载。6）计算一些自定义控件需要的值 onSizeChange()onSizeChange() 方法在view第一次被指定了大小值、或者view的大小发生改变时会被调用。所以一般用来计算一些位置和与view的size有关的值。7）优化你的自定义View，下面是官网给出的优化建议：a.避免不必要的代码;b.在onDraw()方法中不应该有会导致垃圾回收的代码。c.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。、 8）自定义控件一般分类：  （1）继承view：自绘view、含动画view，一般是没有类似的实现： 百分比的控件；  继承已有的view，进行扩展、重写： 比如显示头像的圆形、矩形圆角imageView；  （2）容器控件子类：重写触摸事件、分发、拦截、处理触摸或滑动冲突等；  （3）组合控件，用于复用，标题栏、可清除的editText等。 |

# 1.自定义控件步骤

## 1.1定义declare-styleable，添加到attr.xml；

使用TypedArray获取自定义属性：

TypedArray typedArray = context.obtainStyledAttributes(attrs, R.styleable.CircularAttrsView);

…………

typedArray.recycle();

## 1.2构造函数

有三个参数的构造函数中第三个参数是默认的Style，

调用init();

初始化画笔、画布等

## 1.3 onMeasure(int widthMeasureSpec, int heightMeasureSpec);

计算绘制的尺寸、位置

## 1.4 onLayout()

设置视图在屏幕中显示的位置，onLayout一般只会在自定义ViewGroup中才会使用

## 1.5 onDraw(Canvas canvas);

进行绘制过程。如果自定义ViewGroup这个函数则不需要重载。

onDraw方法包含了一个Canvas叫做画布的参数，onDraw()简单来说就两点：

Canvas决定要去画什么

Paint决定怎么画

比如，Canvas提供了画线方法，Paint就来决定线的颜色。Canvas提供了画矩形，Paint又可以决定让矩形是空心还是实心。

在onDraw方法中开始绘制之前，你应该让画笔Paint对象的信息初始化完毕。这是因为View的重新绘制是比较频繁的，这就可能多次调用onDraw，所以初始化的代码不应该放在onDraw方法里。

## 1.6 计算一些自定义控件需要的值 onSizeChange()

onSizeChange() 方法在view第一次被指定了大小值、或者view的大小发生改变时会被调用。所以一般用来计算一些位置和与view的size有关的值。

## 1.7 优化你的自定义View

下面是官网给出的优化建议：

1、避免不必要的代码;

2、在onDraw()方法中不应该有会导致垃圾回收的代码。

3、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。

# 2.自定义属性

<resource>

<declare-styleable name="自定义属性名称">

<attr name="属性名称" format="属性种类"/>

......

</declare-styleable>

</resource>

format属性值：

reference:引用资源

string:字符串

Color：颜色

boolean：布尔值

dimension：尺寸值

float：浮点型

integer：整型

fraction：百分数

enum：枚举类型

flag：位或运算

# 3.组合控件

组合控件定义一些通用的布局，有时也会使用include标签。

1）先定义一个布局文件，这里为何要使用merge标签，自定义组合控件时会继承RelativeLayout、LinearLayout等控件，这样导致布局的层级无形中增加了一层。

2）继承至现有控件，在构造函数中 LayoutInflater.from(context).inflate(); 将xml填充到当前的容器控件内。

# 自定义控件常用的三个构造方法

public MView(Context context) {

super(context);

}

public MView(Context context, @Nullable AttributeSet attrs) {

super(context, attrs);

}

public MView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {

super(context, attrs, defStyleAttr);

}

三个方法的区别主要

a:代码中直接实例化自定义控件，会调用第一个构造函数

b:xml布局使用自定义控件,会调用第二个构造函数

c:xml布局使用自定义控件,并且控件含有自定义属性,仍然调用的是第二个构造函数.

3.总结：

系统默认只会调用自定义控件的前两个构造函数，第三个构造函数的调用,通常是我们自己在构造函数中主动调用的（例如,在第二个构造函数中调用第三个构造函数） 。通常，自定义控件时需要初始化一些数据，比如画布，画笔等，如果按照默认的构造方法，最好每个构造方法内都要进行初始化操作 ，因此，对构造方法进行修改。

public MView(Context context) {

this(context,null);

}

public MView(Context context, @Nullable AttributeSet attrs) {

this(context, attrs,0);

}

public MView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {

super(context, attrs, defStyleAttr);

// 初始化数据

init();

}

修改前两个方法，把super 给成this，不管控件怎样产生，都去调用三个参数的构造方法，在里面进行初始化操作。

# recycle() 方法将 TypedArray 回收

在 Android 自定义 View 的时候，需要使用 TypedArray 来获取 XML layout 中的属性值，使用完之后，需要调用 recycle() 方法将 TypedArray 回收。

TypedArray不是我们new出来的，而是调用了 obtainStyledAttributes 方法得到的对象。

管理一个对象池，获取对象，释放对象。

从Pool的实现当中可以看到，Pool是一个final类，本身不可变，不允许继承。而内部类SynchronizedPool通过继承SimplePool，并添加对象锁，实现了一个同步的对象池，保证了线程安全，同时也继承了对象池的优点，避免对象重复创建和销毁，减少了系统开销。

从上述源码可以看到，framework层维护了一个同步栈结构的对象池，从而避免在程序运行期间频繁创建属性值TypedArray对象，维护TypedArray对象池的大小默认为5，使用时记得调用recyle()方法将不用的对象返回至对象池来达到重用的目的。

更确切来讲，TypedArray的使用场景之一，就是上述的自定义View，会随着 Activity的每一次Create而Create，因此，需要系统频繁的创建array，对内存和性能是一个不小的开销，如果不使用池模式，每次都让GC来回收，很可能就会造成OutOfMemory。不得不让人Google程序猿大神们的构思和精妙的设计，也为我们平常编写程序优化性能提供了依据和参考。

# invalidate()；

/\*\*

\* Invalidate the whole view. If the view is visible,

\* {@link #onDraw(android.graphics.Canvas)} will be called at some point in

\* the future.

\* <p>

\* This must be called from a UI thread. To call from a non-UI thread, call

\* {@link #postInvalidate()}.

\*/

public void invalidate() {

invalidate(true);

}