

HenCoder Android 自定义 View

1-7: 属性动画 Property Animation (进阶篇)



简介

上期的内容，对于大多数简单的属性动画场景已经够用了。这期的内容主要针对两个方面：

1. 针对特殊类型的属性来做属性动画；
2. 针对复杂的属性关系来做属性动画。

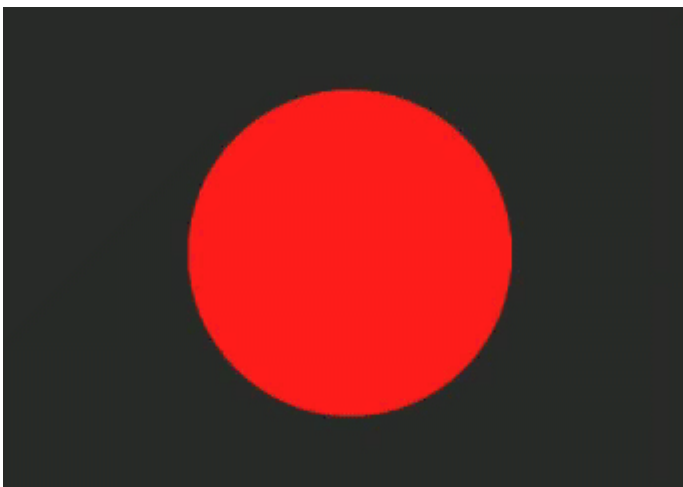
TypeEvaluator

关于 `ObjectAnimator`，上期讲到可以用 `ofInt()` 来做整数的属性动画和用 `ofFloat()` 来做小数的属性动画。这两种属性类型是属性动画最常用的两种，不过在实际的开发中，可以做属性动画的类型还是有其他的一些类型。当需要对其他类型来做属性动画的时候，就需要用到 `TypeEvaluator` 了。

ArgbEvaluator

如视频中的例子，`TypeEvaluator` 最经典的用法是使用 `ArgbEvaluator` 来做颜色渐变的动画。

```
ObjectAnimator animator = ObjectAnimator.ofInt(view, "color", 0xffff);
animator.setEvaluator(new ArgbEvaluator());
animator.start();
```



另外，在 Android 5.0（API 21）加入了新的方法 `ofArgb()`，所以如果你的 `minSdk` 大于或者等于 21（哈哈哈哈哈），你可以直接用下面这种方式：

```
ObjectAnimator animator = ObjectAnimator.ofArgb(view, "color", 0xff
animator.start();
```

自定义 Evaluator

如果你对 `ArgbEvaluator` 的效果不满意，或者你由于别的什么原因希望写一个自定义的 `TypeEvaluator`，你可以这样写：

```
// 自定义 HsvEvaluator
private class HsvEvaluator implements TypeEvaluator<Integer> {
    float[] startHsv = new float[3];
    float[] endHsv = new float[3];
    float[] outHsv = new float[3];

    @Override
    public Integer evaluate(float fraction, Integer startValue, Integer endValue) {
        // 把 ARGB 转换成 HSV
        Color.colorToHSV(startValue, startHsv);
        Color.colorToHSV(endValue, endHsv);

        // 计算当前动画完成度 (fraction) 所对应的颜色值
        if (endHsv[0] - startHsv[0] > 180) {
            endHsv[0] -= 360;
        } else if (endHsv[0] - startHsv[0] < -180) {
            endHsv[0] += 360;
        }
        outHsv[0] = startHsv[0] + (endHsv[0] - startHsv[0]) * fraction;
        if (outHsv[0] > 360) {
            outHsv[0] -= 360;
        } else if (outHsv[0] < 0) {
            outHsv[0] += 360;
        }
        outHsv[1] = startHsv[1] + (endHsv[1] - startHsv[1]) * fraction;
        outHsv[2] = startHsv[2] + (endHsv[2] - startHsv[2]) * fraction;
    }
}
```

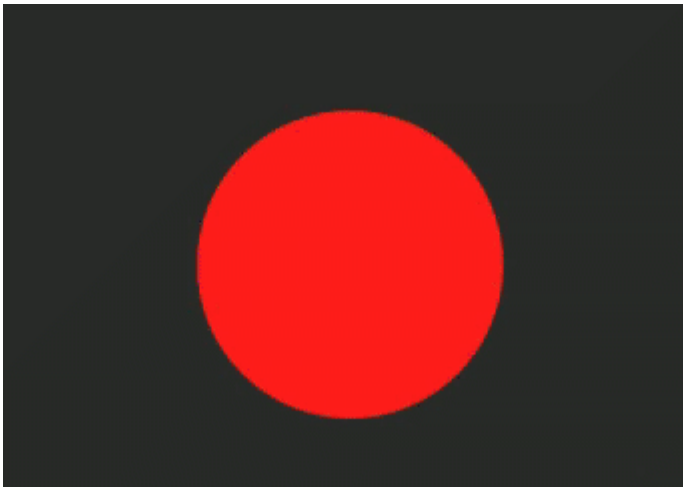
```

        // 计算当前动画完成度 (fraction) 所对应的透明度
        int alpha = startValue >> 24 + (int) ((endValue >> 24 - startValue) * fraction);

        // 把 HSV 转换回 ARGB 返回
        return Color.HSVToColor(alpha, outHsv);
    }
}

ObjectAnimator animator = ObjectAnimator.ofInt(view, "color", 0xff0000);
// 使用自定义的 HslEvaluator
animator.setEvaluator(new HsvEvaluator());
animator.start();

```



ofObject()

借助于 `TypeEvaluator`，属性动画就可以通过 `ofObject()` 来对不限定类型的属性做动画了。方式很简单：

1. 为目标属性写一个自定义的 `TypeEvaluator`
2. 使用 `ofObject()` 来创建 `Animator`，并把自定义的 `TypeEvaluator` 作为参数填入

```

private class PointFEvaluator implements TypeEvaluator<PointF> {
    PointF newPoint = new PointF();

    @Override

```

```
public PointF evaluate(float fraction, PointF startValue, PointF
    float x = startValue.x + (fraction * (endValue.x - startValue.x));
    float y = startValue.y + (fraction * (endValue.y - startValue.y));

    newPoint.set(x, y);

    return newPoint;
}
}

ObjectAnimator animator = ObjectAnimator.ofObject(view, "position",
    new PointFEvaluator(), new PointF(0, 0), new PointF(1, 1));
animator.start();
```



另外在 API 21 中，已经自带了 `PointFEvaluator` 这个类，所以如果你的 `minSdk` 大于或者等于 21（哈哈哈哈哈哈哈哈），上面这个类你就不用写了，直接用就行了。





ofMultiInt() ofMultiFloat()

在 API 引入的新的方法还有 `ofMultiInt()` 和 `ofMultiFloat()` 等，用法也很简单，不过实用性就低了一些。你有兴趣的话可以去做一下了解，这里不在多做介绍。


以上这些就是对 `TypeEvaluator` 的介绍。它的作用是让你可以对同样的属性有不同的解析方式，对本来无法解析的属性也可以打造出你需要的解析方式。有了 `TypeEvaluator`，你的属性动画就有了更大的灵活性，从而有了无限的可能。

`TypeEvaluator` 是本期的第一部分内容：针对特殊的属性来做属性动画，它可以让你「做到本来做不到的动画」。接下来是本期的第二部分内容：针对复杂的属性关系来做动画，它可以让你「能做到的动画做起来更简单」。

PropertyValuesHolder 同一个动画中改变多个属性

很多时候，你在同一个动画中会需要改变多个属性，例如在改变透明度的同时改变尺寸。如果使用 `ViewPropertyAnimator`，你可以直接用连写的方式来在一个动画中同时改变多个属性：

```
view.animate()  
    .scaleX(1)  
    .scaleY(1)  
    .alpha(1);
```



而对于 `ObjectAnimator`，是不能这么用的。不过你可以使用 `PropertyValuesHolder` 来同时在一个动画中改变多个属性。

```
PropertyValuesHolder holder1 = PropertyValuesHolder.ofFloat("scaleX", 0.5f);
PropertyValuesHolder holder2 = PropertyValuesHolder.ofFloat("scaleY", 0.5f);
PropertyValuesHolder holder3 = PropertyValuesHolder.ofFloat("alpha", 0.5f);

ObjectAnimator animator = ObjectAnimator.ofPropertyValuesHolder(view, holder1, holder2, holder3);
animator.start();
```

`PropertyValuesHolder` 的意思从名字可以看出来，它是一个属性值的批量存放地。所以你如果有多个属性需要修改，可以把它们放在不同的 `PropertyValuesHolder` 中，然后使用 `ofPropertyValuesHolder()` 统一放进 `Animator`。这样你就不用为每个属性单独创建一个 `Animator` 分别执行了。

AnimatorSet 多个动画配合执行

有的时候，你不止需要在一个动画中改变多个属性，还会需要多个动画配合工作，比如，在内容的大小从 0 放大到 100% 大小后开始移动。这种情况使用 `PropertyValuesHolder` 是不行的，因为这些属性如果放在同一个动画中，需要共

享动画的开始时间、结束时间、Interpolator 等等一系列的设定，这样就不能有先后次序地执行动画了。

这就需要用到 AnimatorSet 了。

```
ObjectAnimator animator1 = ObjectAnimator.ofFloat(...);
animator1.setInterpolator(new LinearInterpolator());
ObjectAnimator animator2 = ObjectAnimator.ofInt(...);
animator2.setInterpolator(new DecelerateInterpolator());

AnimatorSet animatorSet = new AnimatorSet();
// 两个动画依次执行
animatorSet.playSequentially(animator1, animator2);
animatorSet.start();
```

使用 playSequentially()，就可以让两个动画依次播放，而不用为它们设置监听器来手动为他们监管协作。

AnimatorSet 还可以这么用：

```
// 两个动画同时执行
animatorSet.playTogether(animator1, animator2);
animatorSet.start();
```



```
animatorSet.start();
```

以及这么用：

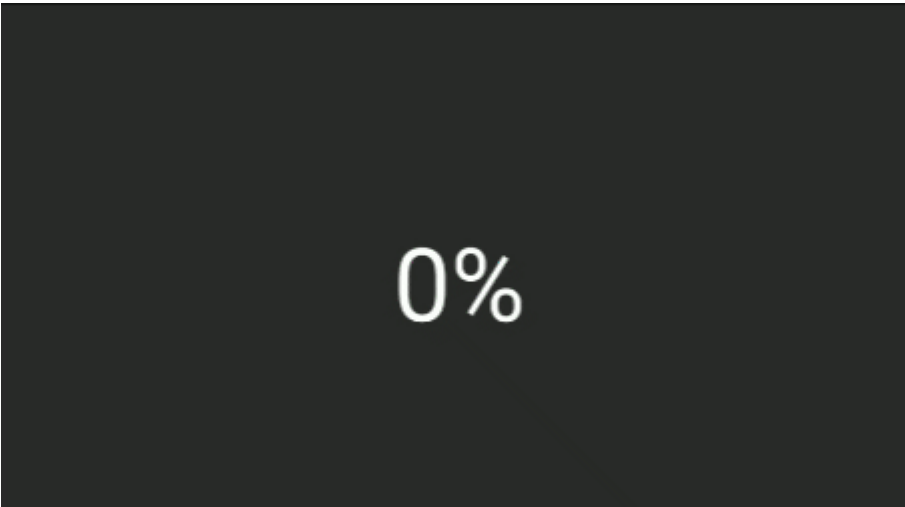
```
// 使用 AnimatorSet.play(animatorA).with/before/after(animatorB)
// 的方式来精确配置各个 Animator 之间的关系
animatorSet.play(animator1).with(animator2);
animatorSet.play(animator1).before(animator2);
animatorSet.play(animator1).after(animator2);
animatorSet.start();
```

有了 `AnimatorSet`，你就可以对多个 `Animator` 进行统一规划和管理，让它们按照要求的顺序来工作。它的使用比较简单，具体的用法我写在讲义里，你可以看一下。

PropertyValuesHolders.ofKeyframe() 把同一个属性拆分

除了合并多个属性和调配多个动画，你还可以在 `PropertyValuesHolder` 的基础上更进一步，通过设置 `Keyframe`（关键帧），把同一个动画属性拆分成多个阶段。例如，你可以让一个进度增加到 100% 后再「反弹」回来。

```
// 在 0% 处开始
Keyframe keyframe1 = Keyframe.ofFloat(0, 0);
// 时间经过 50% 的时候，动画完成度 100%
Keyframe keyframe2 = Keyframe.ofFloat(0.5f, 100);
// 时间见过 100% 的时候，动画完成度倒退到 80%，即反弹 20%
Keyframe keyframe3 = Keyframe.ofFloat(1, 80);
PropertyValuesHolder holder = PropertyValuesHolder.ofKeyframe("progress",
    keyframe1, keyframe2, keyframe3);
ObjectAnimator animator = ObjectAnimator.ofPropertyValuesHolder(view,
    holder);
animator.start();
```



0%

第二部分，「关于复杂的属性关系来做动画」，就这么三种：

1. 使用 `PropertyValuesHolder` 来对多个属性同时做动画；
2. 使用 `AnimatorSet` 来同时管理调配多个动画；
3. `PropertyValuesHolder` 的进阶使用：使用 `PropertyValuesHolder.ofKeyframe()` 来把一个属性拆分成多段，执行更加精细的属性动画。

ValueAnimator 最基本的轮子

额外简单说一下 `ValueAnimator`。很多时候，你用不到它，只是在你使用一些第三方库的控件，而你想要做动画的属性却没有 `setter / getter` 方法的时候，会需要用到它。

除了 `ViewPropertyAnimator` 和 `ObjectAnimator`，还有第三个选择是 `ValueAnimator`。`ValueAnimator` 并不常用，因为它的功能太基础了。`ValueAnimator` 是 `ObjectAnimator` 的父类，实际上，`ValueAnimator` 就是一个不能指定目标对象版本的 `ObjectAnimator`。`ObjectAnimator` 是自动调用目标对象的 `setter` 方法来更新目标属性的值，以及很多的时候还会以此来改变目标对象的 UI，而 `ValueAnimator` 只是通过渐变的方式来改变一个独立的数据，这个数据不是属于某个对象的，至于在数据更新后要做什么事，全都由你来定，你可以依然是去调用某个对象的 `setter` 方法（别这么为难自己），也可以做其他的事，不管要做什么，都是要你自己来写的，`ValueAnimator` 不会帮你做。功能最少、最不方便，但有时也是束缚最少、最灵活。比如有的时候，你要给一个第三方控件做动画，你需要更新的那个属性没有 `setter` 方法，只能直接修改，这样的话 `ObjectAnimator` 就不灵了

啊。怎么办？这个时候你就可以用 `ValueAnimator`，在它的 `onUpdate()` 里面更新这个属性的值，并且手动调用 `invalidate()`。

所以你看，`ViewPropertyAnimator`、`ObjectAnimator`、`ValueAnimator` 这三种 `Animator`，它们其实是一种递进的关系：从左到右依次变得更加难用，也更加灵活。但我要说明一下，它们的性能是一样的，因为 `ViewPropertyAnimator` 和 `ObjectAnimator` 的内部实现其实都是 `ValueAnimator`，`ObjectAnimator` 更是本来就是 `ValueAnimator` 的子类，它们三个的性能并没有差别。它们的差别只是使用的便捷性以及功能的灵活性。所以在实际使用时候的选择，只要遵循一个原则就行：尽量用简单的。能用 `View.animate()` 实现就不用 `ObjectAnimator`，能用 `ObjectAnimator` 就不用 `ValueAnimator`。

练习项目

为了避免转头就忘，强烈建议你趁热打铁，做一下这个练习项目：

[HenCoderPracticeDraw7](#)
