Android属性动画

原创 xyzso1z 2020-06-30 20:11:31 35 收藏 编辑 版权
分类专栏： Android

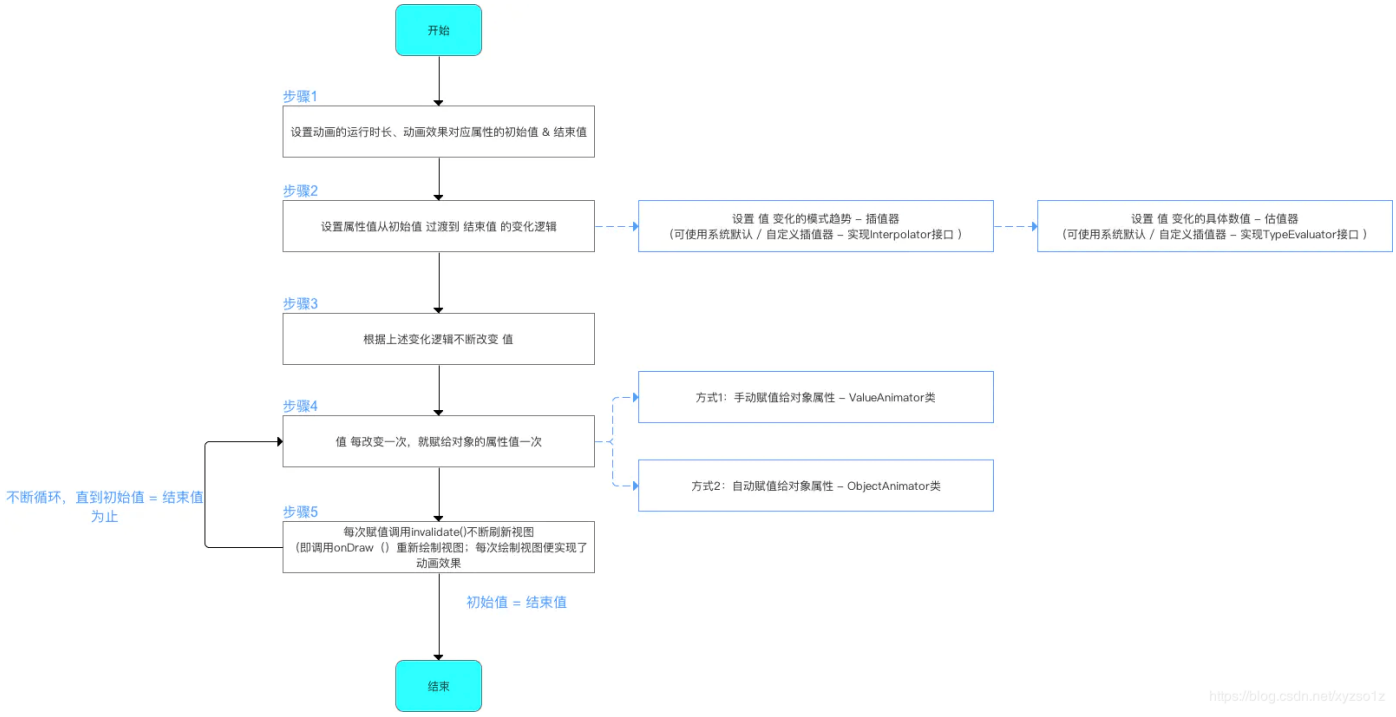
属性动画

> 特点：

- 作用对象不只是 `View` 对象，甚至没有对象也可以。
- 动画效果不只是4中基本变换，还有其他动画效果。
- 作用领域:API11 后引入。

> 工作原理

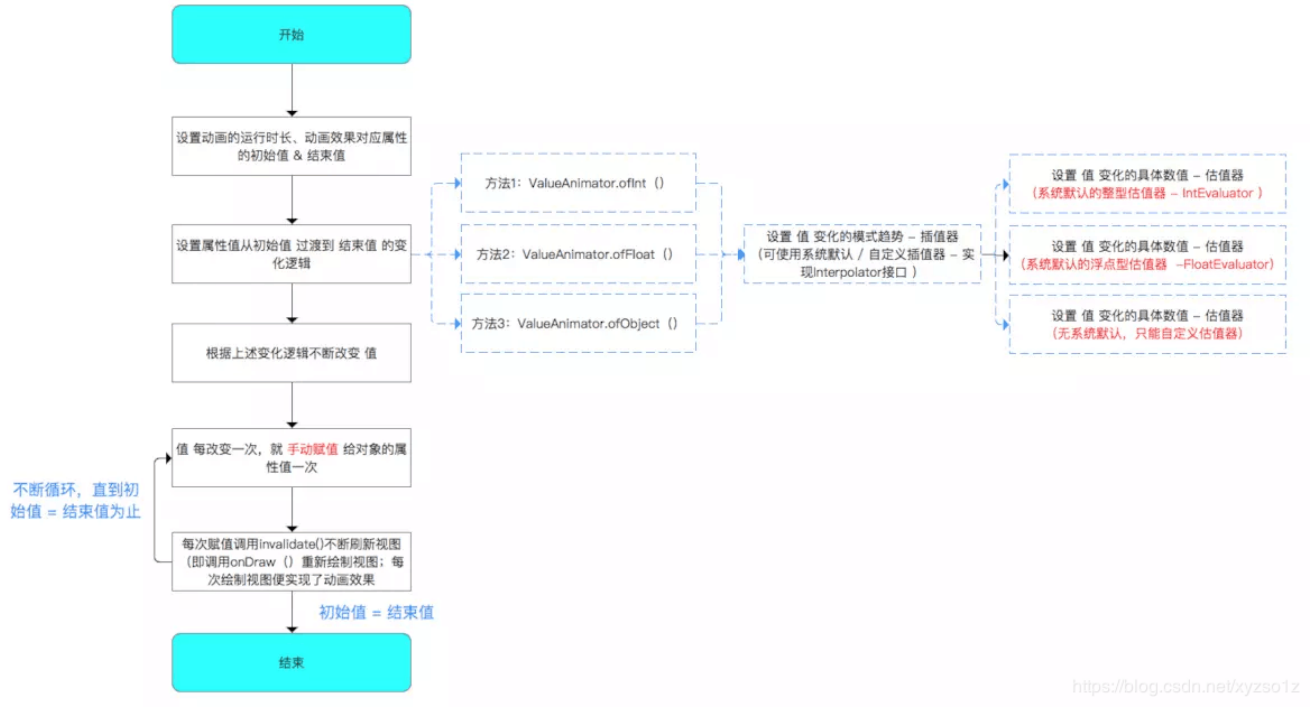
在一定时间间隔内，通过不断对值进行改变，并不断将该值赋给对象的属性，从而实现该对象在该属性上的动画效果。



从上述工作原理可以看出属性动画有两个非常重要的类：`ValueAnimator`类和`ObjectAnimator`类。

一、`ValueAnimator`类

- 通过不断控制 值 的变化，再不断 手动 赋给对象的属性，从而实现动画效果。
- 具体如图：



从上面原理可以看出：`ValueAnimator`类中有4个重要方法:

1. `ValueAnimator.ofInt(int values）`
2. `ValueAnimator.ofFloat(float values）`
3. `ValueAnimator.ofObject(int values)`
4. `ValueAnimator.ofArgb(int values)`

下面逐一介绍：

1.1、`ValueAnimator.ofInt()`方法

- 作用：将初始值以整型数值的形式 过渡到结束值，即估值器是整型估值器-`IntEvaluator`
- 使用：
  因为`ValueAnimator`本质只是一种值得操作机制，所以下面的介绍先是展示如何改变一个值得过程（下面的实例主要讲解：如何将一个值从0平滑地过渡到3）

Java代码实现

实际开发中，建议使用Java代码实现属性动画：因为很多时候属性的起始值是无法提前确定的（无法使用`xml`设置），这就需要在 Java 代码里动态获取。

```
// 步骤1：设置动画属性的初始值 & 结束值
ValueAnimator anim = ValueAnimator.ofInt(0, 3);
      // ofInt（）作用有两个
      // 1. 创建动画实例
      // 2. 将传入的多个Int参数进行平滑过渡:此处传入0和1,表示将值从0平滑过渡到1
      // 如果传入了3个Int参数 a,b,c ,则是先从a平滑过渡到b,再从b平滑过渡到C，以此类推
      // ValueAnimator.ofInt()内置了整型估值器,直接采用默认的.不需要设置，即默认设置了如何从初始值 过渡到 结束值
      // 关于自定义插值器我将在下节进行讲解
      // 下面看看ofInt()的源码分析 ->>关注1

// 步骤2：设置动画的播放各种属性
      anim.setDuration(500);
      // 设置动画运行的时长

      anim.setStartDelay(500);
      // 设置动画延迟播放时间

      anim.setRepeatCount(0);
      // 设置动画重复播放次数 = 重放次数+1
```

```
        // 动画播放次数 = infinite时,动画无限重复

        anim.setRepeatMode(ValueAnimator.RESTART);
        // 设置重复播放动画模式
        // ValueAnimator.RESTART(默认):正序重放
        // ValueAnimator.REVERSE:倒序回放

    // 步骤3：将改变的值手动赋值给对象的属性值：通过动画的更新监听器
        // 设置 值的更新监听器
        // 即：值每次改变、变化一次,该方法就会被调用一次
        anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animation) {

                int currentValue = (Integer) animation.getAnimatedValue();
                // 获得改变后的值

                System.out.println(currentValue);
                // 输出改变后的值

     // 步骤4：将改变后的值赋给对象的属性值，下面会详细说明
                View.setproperty（currentValue）;

      // 步骤5：刷新视图，即重新绘制，从而实现动画效果
                View.requestLayout();


            }
        });

        anim.start();
        // 启动动画
    }

    // 关注1：ofInt（）源码分析
    public static ValueAnimator ofInt(int... values) {
        // 允许传入一个或多个Int参数
        // 1. 输入一个的情况（如a）：从0过渡到a；
        // 2. 输入多个的情况（如a，b，c）：先从a平滑过渡到b，再从b平滑过渡到c

        ValueAnimator anim = new ValueAnimator();
        // 创建动画对象
        anim.setIntValues(values);
        // 将传入的值赋值给动画对象
        return anim;
    }
```

效果图：
值 从初始值 过渡到 结束值 的过程如下：

```
04-18 03:30:26.928 15753-15753/? I/System.out: 0
04-18 03:30:27.144 15753-15753/? I/System.out: 2
04-18 03:30:27.216 15753-15753/? I/System.out: 2
04-18 03:30:27.224 15753-15753/? I/System.out: 2
04-18 03:30:27.244 15753-15753/? I/System.out: 2
04-18 03:30:27.256 15753-15753/? I/System.out: 2
04-18 03:30:27.276 15753-15753/? I/System.out: 2
04-18 03:30:27.292 15753-15753/? I/System.out: 3
```


xml 实现

具有重用性，即将通用的动画写到 `xml` 里，可在各个界面中重用它。

- 步骤1： 在路径 `res/animator` 的文件里创建相应的动画 `.xml` 文件(`set_animator.xml`)
- 步骤2： 设置动画参数：

```
 // ValueAnimator采用<animator>  标签
<animator xmlns:android="http://schemas.android.com/apk/res/android"
 android:valueFrom="0"   // 初始值
 android:valueTo="100"  // 结束值
 android:valueType="intType" // 变化值类型 ：floatType & intType

 android:duration="3000" // 动画持续时间（ms），必须设置，动画才有效果
 android:startOffset ="1000" // 动画延迟开始时间（ms）
 android:fillBefore = “true” // 动画播放完后，视图是否会停留在动画开始的状态，默认为true
 android:fillAfter = “false” // 动画播放完后，视图是否会停留在动画结束的状态，优先于fillBefore值，默认为false
 android:fillEnabled= “true” // 是否应用fillBefore值，对fillAfter值无影响，默认为true
 android:repeatMode= “restart” // 选择重复播放动画模式，restart代表正序重放，reverse代表倒序回放，默认为restart|
 android:repeatCount = “0” // 重放次数（所以动画的播放次数=重放次数+1），为infinite时无限重复
 android:interpolator = @[package:]anim/interpolator_resource // 插值器，即影响动画的播放速度,下面会详细讲
/>
```

- 步骤3： 在 `Java` 代码中启动动画

```
 // 载入XML动画
 Animator animator = AnimatorInflater.loadAnimator(context, R.animator.set_animation);

// 设置动画对象
animator.setTarget(view);

// 启动动画
animator.start();
```

1.2、 `ValueAnimator.oFloat()`方法

- 作用：将初始值 以浮点型数值的形式 过渡到结束值
- 用法：同 `ValueAnimator.ofInt()` 方法。

1.3、 `ValueAnimator.ofObject()`方法

- 作用： 将初始值 以对象的形式过渡到结束值(即通过操作对象 实现动画效果)
- 用法：

```
// 创建初始动画时的对象  & 结束动画时的对象
myObject object1 = new myObject();
myObject object2 = new myObject();

ValueAnimator anim = ValueAnimator.ofObject(new myObjectEvaluator(),  object1, object2);
// 创建动画对象 & 设置参数
// 参数说明
// 参数1：自定义的估值器对象（TypeEvaluator 类型参数） - 下面会详细介绍
// 参数2：初始动画的对象
// 参数3：结束动画的对象
anim.setDuration(5000);
anim.start();
```

在这里要讲解一下估值器(`TypeEvalator`)

估值器（ `TypeEvaluator` ）

- 作用：设置动画 如何从初始值过渡到结束值的逻辑

> 1. 插值器(`Interpolator`)决定值得变化模式（匀速、加速）
> 2. 估值器(`TypeEvaluator`)决定值得具体变化数值

从上面可知：

- `ValueAnimator.ofFloat()` 实现了将初始值 以浮点型的形式 过渡到结束值的逻辑，那么这个过渡逻辑具体是怎么样的呢？
  答：系统内部内置了一个 `FloatEvaluator` 估值器，内部实现了初始值与结束值 以浮点型的过渡逻辑。下面是 `FloatEvaluator` 的代码实现：

```
public class FloatEvaluator implements TypeEvaluator {
// FloatEvaluator实现了TypeEvaluator接口

// 重写evaluate()
public Object evaluate(float fraction, Object startValue, Object endValue) {
// 参数说明
// fraction：表示动画完成度（根据它来计算当前动画的值）
// startValue、endValue：动画的初始值和结束值
float startFloat = ((Number) startValue).floatValue();

 return startFloat + fraction * (((Number) endValue).floatValue() - startFloat);
  // 初始值 过渡 到结束值 的算法是：
  // 1. 用结束值减去初始值，算出它们之间的差值
  // 2. 用上述差值乘以fraction系数
  // 3. 再加上初始值，就得到当前动画的值
}
}
```

- 对于 `ValueAnimator.ofObject()` ，系统并没有默认实现，因为对对象的动画操作复杂且多样，系统无法知道如何从初始对象过渡到结束对象。因此我们需要自定义估值器（ `TypeEvaluator` ）来告知系统如何进行从初始对象过渡到结束对象的逻辑。

```
 // 实现TypeEvaluator接口
 public class ObjectEvaluator implements TypeEvaluator{

   // 复写evaluate（）
   // 在evaluate（）里写入对象动画过渡的逻辑
   @Override
   public Object evaluate(float fraction, Object startValue, Object endValue) {
    // 参数说明
    // fraction：表示动画完成度（根据它来计算当前动画的值）
    // startValue、endValue：动画的初始值和结束值

   ...
   // 写入对象动画过渡的逻辑

    return value;
   // 返回对象动画过渡的逻辑计算后的值
  }
}
```

实例说明

实现动画效果：一个圆点从一个点移动到另一个点

步骤1：定义对象类

- 因为 `ValueAnimator.ofObject()` 是面向对象操作的，所以需要自定义对象类。
- 本利需要操作的对象是：圆的点坐标（ `Point.java` ）

```
public class Point {

    // 设置两个变量用于记录坐标的位置
    private float x;
    private float y;

    // 构造方法用于设置坐标
    public Point(float x, float y) {
        this.x = x;
        this.y = y;
    }

    // get方法用于获取坐标
    public float getX() {
        return x;
    }

    public float getY() {
        return y;
    }
}
```

步骤2：根据需求实现 `TypeEvaluator` 接口

- 实现 `TypeEvaluator` 接口的目的是自定义如何从 初始点坐标 过渡到结束点坐标。
- 本例实现的是一个从左上角到右下角的坐标过渡逻辑

`PointEvaluator.java`

```
// 实现TypeEvaluator接口
public class PointEvaluator implements TypeEvaluator {

    // 复写evaluate（）
    // 在evaluate（）里写入对象动画过渡的逻辑
    @Override
    public Object evaluate(float fraction, Object startValue, Object endValue) {

        // 将动画初始值startValue、和 动画结束值endValue 强制类型转换成Point对象
```

```
        // 将动画初始值startValue 和 动画结束值endValue 强制类型转换成Point对象
        Point startPoint = (Point) startValue;
        Point endPoint = (Point) endValue;

        // 根据fraction来计算当前动画的x和y的值
        float x = startPoint.getX() + fraction * (endPoint.getX() - startPoint.getX());
        float y = startPoint.getY() + fraction * fraction * (endPoint.getY() - startPoint.getY());

        // 将计算后的坐标封装到一个新的Point对象中并返回
        Point point = new Point(x, y);
        return point;
    }

}
```

- 上面步骤是根据需求自定义 `TypeEvaluator` 的实现
- 下面将讲解如何通过对 `Point` 对象进行动画操作，从而实现整个自定义 `View` 的动画效果

步骤3：将属性动画作用到自定义 `View` 当中

`PointView.java`

```
public class PointView extends View {
// 设置需要用到的变量
public static final float RADIUS = 70f;// 圆的半径 = 70
private Point currentPoint;// 当前点坐标
 private Paint mPaint;// 绘图画笔

 // 构造方法(初始化画笔)
 public PointView(Context context, AttributeSet attrs) {
     super(context, attrs);
     // 初始化画笔
     mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
     mPaint.setColor(Color.BLUE);
 }

// 复写onDraw()从而实现绘制逻辑
 // 绘制逻辑:先在初始点画圆,通过监听当前坐标值(currentPoint)的变化,每次变化都调用onDraw()重新绘制圆,从而实现圆的平移动画效果
 @Override
 protected void onDraw(Canvas canvas) {
     // 如果当前点坐标为空(即第一次)
     if (currentPoint == null) {
         currentPoint = new Point(RADIUS, RADIUS);
         // 创建一个点对象(坐标是(70,70))

         // 在该点画一个圆:圆心 = (70,70),半径 = 70
         float x = currentPoint.getX();
         float y = currentPoint.getY();
         canvas.drawCircle(x, y, RADIUS, mPaint);


         // (重点关注)将属性动画作用到View中
         // 步骤1:创建初始动画时的对象点  & 结束动画时的对象点
         Point startPoint = new Point(RADIUS, RADIUS);// 初始点为圆心(70,70)
         Point endPoint = new Point(1300, 2800);// 结束点为(700,1000)

         // 步骤2:创建动画对象 & 设置初始值 和 结束值
         ValueAnimator anim = ValueAnimator.ofObject(new PointEvaluator(), startPoint, endPoint);
         // 参数说明
         // 参数1:TypeEvaluator 类型参数 - 使用自定义的PointEvaluator(实现了TypeEvaluator接口)
         // 参数2:初始动画的对象点
         // 参数3:结束动画的对象点

         // 步骤3:设置动画参数
         anim.setDuration(3000);
         // 设置动画时长

         anim.setRepeatCount(-1);
         anim.setRepeatMode(ValueAnimator.REVERSE);
```

```
            // 步骤3：通过 值 的更新监听器，将改变的对象手动赋值给当前对象
            // 此处是将 改变后的坐标值对象 赋给 当前的坐标值对象
            // 设置 值的更新监听器
            // 即每当坐标值（Point对象）更新一次，该方法就会被调用一次
            anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animation) {
                currentPoint = (Point) animation.getAnimatedValue();
                // 将每次变化后的坐标值（估值器PointEvaluator中evaluate（）返回的Piont对象值）到当前坐标值对象（currentPoint）
                // 从而更新当前坐标值（currentPoint）

                // 步骤4：每次赋值后就重新绘制，从而实现动画效果
                invalidate();
                // 调用invalidate()后，就会刷新View，即才能看到重新绘制的界面，即onDraw()会被重新调用一次
                // 所以坐标值每改变一次，就会调用onDraw()一次
              }
          });

           anim.start();
           // 启动动画


       } else {
            // 如果坐标值不为0，则画圆
            // 所以坐标值每改变一次，就会调用onDraw()一次，就会画一次圆，从而实现动画效果

            // 在该点画一个圆：圆心 = (30,30)，半径 = 30
            float x = currentPoint.getX();
            float y = currentPoint.getY();
            canvas.drawCircle(x, y, RADIUS, mPaint);
        }
    }

}
```

步骤4：在布局文件中加入自定义View控件

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent">

<com.example.animationtest.valueanimation.ofobject.PointView
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
</LinearLayout>
```

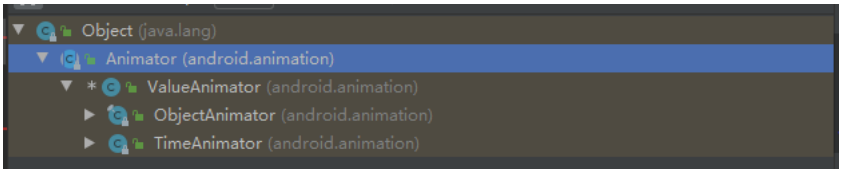
步骤5：在需要的界面显示视图

```
public class ValueAnimationOfObjectActivity extends BaseActivity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_value_animation_of_object);
    }
}
```

二、`ObjectAnimator`类

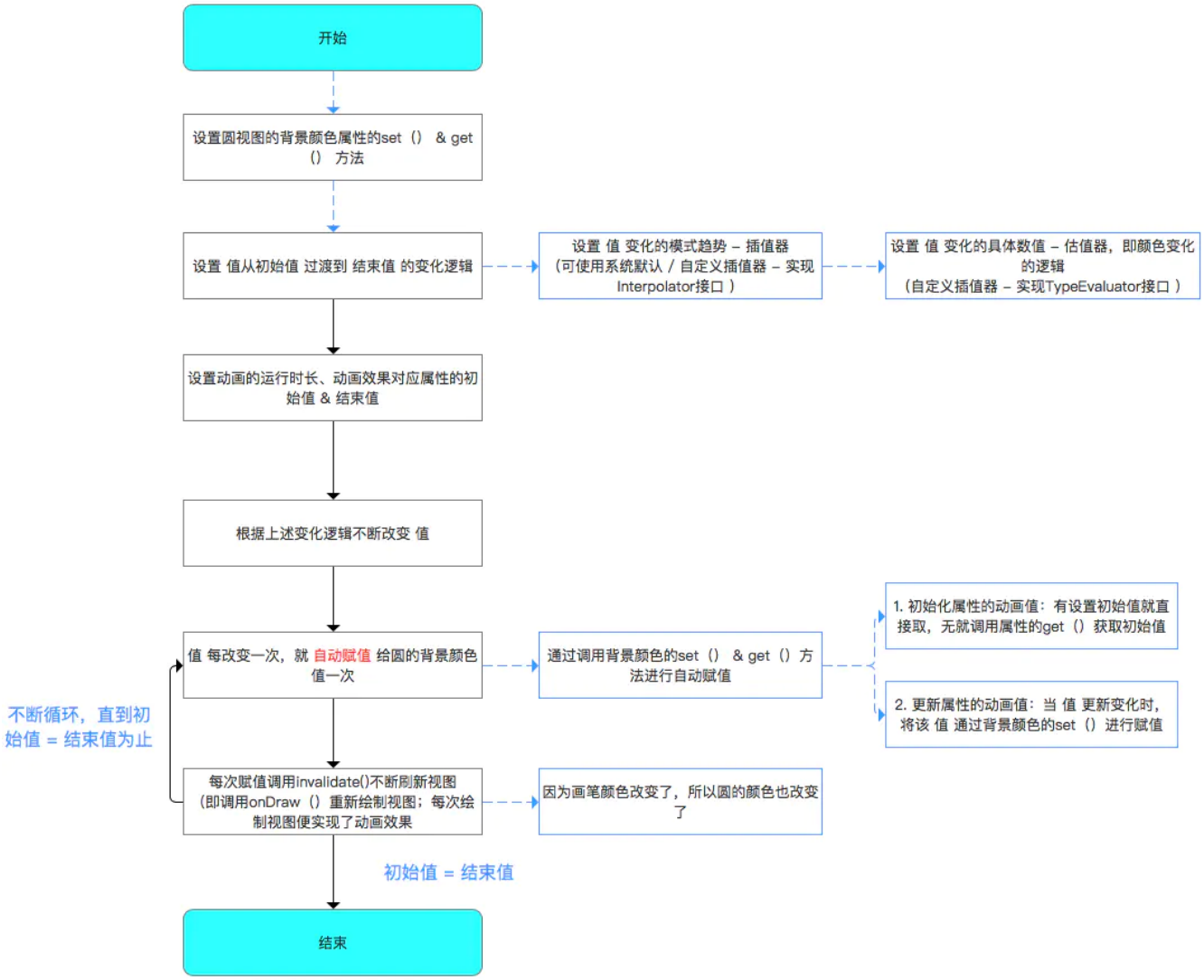
2.1 简介

- 类图：



继承自 `ValueAnimator` 类,即底层的动画实现机制是基于`ValueAnimator` 类。

- 原理：
  直接对对象的属性值进行改变，从而实现动画效果(如直接改变View的alpha属性从而实现透明度的动画效果)



从上图可看出 `ObjectAnimator` 与 `ValueAnimator` 类的区别：

- `ValueAnimator` 类是先改变值，然后**手动**赋值给对象的属性从而实现动画；是**间接**对对象属性进行操作
- `ObjectAnimator` 类是先改变值，然后**自动**赋值给对象的属性从而实现动画；是**直接**对对象属性进行操作。

2.2 使用

因为继承了 `ValueAnimator` 类，所以使用的方式基本相同： `Java` 设置\ `xml` 设置。

2.2.1 Java代码设置

```
ObjectAnimator animator = ObjectAnimator.ofFloat(Object object, String property, float ....values);

// ofFloat()作用有两个
// 1. 创建动画实例
// 2. 参数设置：参数说明如下
// Object object：需要操作的对象
// String property：需要操作的对象的属性
// float ....values：动画初始值 & 结束值（不固定长度）
// 若是两个参数a,b，则动画效果则是从属性的a值到b值
```

```
// 若是三个参数a,b,c，则则动画效果则是从属性的a值到b值再到c值
// 以此类推
// 至于如何从初始值 过渡到 结束值，同样是由估值器决定，此处ObjectAnimator.ofFloat（）是有系统内置的浮点型估值器FloatEvaluator，同Valu

anim.setDuration(500);
        // 设置动画运行的时长

        anim.setStartDelay(500);
        // 设置动画延迟播放时间

        anim.setRepeatCount(0);
        // 设置动画重复播放次数 = 重放次数+1
        // 动画播放次数 = infinite时,动画无限重复

        anim.setRepeatMode(ValueAnimator.RESTART);
        // 设置重复播放动画模式
        // ValueAnimator.RESTART(默认):正序重放
        // ValueAnimator.REVERSE:倒序回放

animator.start();
// 启动动画
```

2.2.2 在xml代码中设置

- 步骤一：在路径 `res/animator` 的文件里创建动画效果 `.xml` 文件
- 步骤二：设置动画参数

```
 // ObjectAnimator 采用<animator>  标签
 <objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
     android:valueFrom="1"   // 初始值
     android:valueTo="0"  // 结束值
     android:valueType="floatType"  // 变化值类型 ：floatType & intType
     android:propertyName="alpha" // 对象变化的属性名称
/>
```

- 步骤三： 在Java代码中启动动画

```
   // 载入XML动画
Animator animator = AnimatorInflater.loadAnimator(context,   R.animator.view_animation);

// 设置动画对象
animator.setTarget(view);

// 启动动画
animator.start();
```

2.3 实例

四种基本变换：平移、旋转、缩放 & 透明度

- 透明度（ `alpha` ）

```
  ObjectAnimator animator = ObjectAnimator.ofFloat(textView, "alpha", 1f, 0f, 1f);
    // 表示的是:
    // 动画作用对象是mButton
    // 动画作用的对象的属性是透明度alpha
    // 动画效果是:常规 - 全透明 - 常规
    animator.setDuration(5000);
    animator.start();
```

- 旋转（ `rotation` ）

```
   ObjectAnimator animator = ObjectAnimator.ofFloat(textView, "rotation", 0f, 360f);

     // 表示的是:
     // 动画作用对象是mButton
     // 动画作用的对象的属性是旋转alpha
```

```
// 动画效果是:0 - 360
animator.setDuration(5000);
animator.setRepeatCount(-1);
animator.setRepeatMode(ValueAnimator.REVERSE);
animator.start();
```

- 平移(`translationX`)

```
float curTranslationX = textView.getTranslationX();
// 获得当前按钮的位置
ObjectAnimator animator = ObjectAnimator.ofFloat(textView, "translationX", curTranslationX, 500, curTranslationX, curT


// 表示的是:
// 动画作用对象是mButton
// 动画作用的对象的属性是X轴平移(在Y轴上平移同理,采用属性"translationY"
// 动画效果是:从当前位置平移到 x=1500 再平移到初始位置
animator.setDuration(5000);
animator.setRepeatCount(-1);
animator.setRepeatMode(ValueAnimator.RESTART);
animator.start();
```

- 缩放(`scaleX`)

```
ObjectAnimator animator = ObjectAnimator.ofFloat(textView, "scaleX", 1f, 1.5f, 1f, 1.5f, 1);
// 表示的是:
// 动画作用对象是mButton
// 动画作用的对象的属性是X轴缩放
// 动画效果是:放大到3倍,再缩小到初始大小
animator.setDuration(5000);
animator.setRepeatCount(-1);
animator.setRepeatMode(ValueAnimator.RESTART);
animator.start();
```

效果

2.4 通过自定义对象属性实现动画

上面我们使用了属性动画最基本的四种动画效果：透明度、平移、旋转和缩放。即在 `ObjectAnimator.ofFloat()` 的第二个参数 `String property` 传入 `alpha` 、 `rotation` 、 `translationX` 和 `scaleY` 等。

属性	作用
Alpha	控制View的透明度
TranslationX	控制X方向的位移
TranslationY	控制Y方向的位移
ScaleX	控制X方向的缩放倍数
ScaleY	控制Y方向的缩放倍数
Rotation	控制以屏幕方向为轴的旋转度数
RotationX	控制以X轴为轴的旋转度数
RotationY	控制以Y轴为轴的旋转度数

`ofFloat()` 的第二个参数还可以传入任意属性值

- `ObjectAnimator` 类 对 对象属性值 进行改变从而实现动画效果的本质是：通过不断控制 值 的变化，再不断 自动 赋给对象的属性，从而实现动画。
- 自动赋值给对象的属性的本质是调用该对象属性的set()和get()方法进行赋值。
- `ObjectAnimator.ofFloat(Object object , String property , float... values )` 的第二个参数传入值的作用是：让 `ObjectAnimator` 类根据传入的属性名 找到该对象属性名的 `set()` 和 `get()` 方法，从而进行对象属性值的赋值。

2.5 实例

对于属性动画，其优势在于：不局限于系统限定的动画，可以自定义动画，即自定义对象的属性，并通过操作自定义的属性从而实现动画。那么，该如何自定义属性呢？本质上就是：

- 为对象设置需要操作属性的 `set()` 和 `get()` 方法
- 通过实现 `TypeEvaluator` 类从而定义属性变化的逻辑

下面，用一个实例来说明如何通过自定义属性实现属性动画效果。

- 效果：一个圆的颜色渐变
- 实现：
  步骤一：设置对象类属性的 `set()` 和 `get()` 方法
  `DIYView.java`

```
public class DIYView extends View {
  // 设置需要用到的变量
    public static final float RADIUS = 100f;// 圆的半径 = 100
    private Paint mPaint;// 绘图画笔

    private String color;
    // 设置背景颜色属性

  // 设置背景颜色的get() & set()方法
  public String getColor() {
       return color;
  }

    public void setColor(String color) {
        this.color = color;
        mPaint.setColor(Color.parseColor(color));
        // 将画笔的颜色设置成方法参数传入的颜色
        invalidate();
       // 调用了invalidate()方法，即画笔颜色每次改变都会刷新视图，然后调用onDraw()方法重新绘制圆
       // 而因为每次调用onDraw()方法时画笔的颜色都会改变，所以圆的颜色也会改变
     }


    // 构造方法(初始化画笔)
    public DIYView(Context context, AttributeSet attrs) {
        super(context, attrs);
        // 初始化画笔
        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        mPaint.setColor(Color.BLUE);
    }

    // 复写onDraw()从而实现绘制逻辑
    // 绘制逻辑：先在初始点画圆，通过监听当前坐标值(currentPoint)的变化，每次变化都调用onDraw()重新绘制圆，从而实现圆的平移动画效果
    @Override
     protected void onDraw(Canvas canvas) {
         canvas.drawCircle(500, 500, RADIUS, mPaint);
     }
}
```

步骤二：在布局文件加入自定义 `View` 控件

```
  <?xml version="1.0" encoding="utf-8"?>
  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="match_parent"
      android:layout_height="match_parent">

    <com.example.animationtest.valueanimation.ofobject.DIYView
        android:id="@+id/diy_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

步骤3：根据需求实现 `TypeEvaluator` 接口

此处实现估值器的本质是：实现 颜色过渡的逻辑。

```
public class ColorEvaluator implements TypeEvaluator {
     // 实现TypeEvaluator接口

  private int mCurrentRed;

private int mCurrentGreen ;

private int mCurrentBlue ;

// 复写evaluate（）
// 在evaluate（）里写入对象动画过渡的逻辑：此处是写颜色过渡的逻辑
@Override
public Object evaluate(float fraction, Object startValue, Object endValue) {

    // 获取到颜色的初始值和结束值
    String startColor = (String) startValue;
    String endColor = (String) endValue;

    // 通过字符串截取的方式将初始化颜色分为RGB三个部分，并将RGB的值转换成十进制数字
    // 那么每个颜色的取值范围就是0-255
    int startRed = Integer.parseInt(startColor.substring(1, 3), 16);
    int startGreen = Integer.parseInt(startColor.substring(3, 5), 16);
    int startBlue = Integer.parseInt(startColor.substring(5, 7), 16);

    int endRed = Integer.parseInt(endColor.substring(1, 3), 16);
    int endGreen = Integer.parseInt(endColor.substring(3, 5), 16);
    int endBlue = Integer.parseInt(endColor.substring(5, 7), 16);

    // 将初始化颜色的值定义为当前需要操作的颜色值
    mCurrentRed = startRed;
    mCurrentGreen = startGreen;
    mCurrentBlue = startBlue;


    // 计算初始颜色和结束颜色之间的差值
    // 该差值决定着颜色变化的快慢：初始颜色值和结束颜色值很相近，那么颜色变化就会比较缓慢；否则，变化则很快
    // 具体如何根据差值来决定颜色变化快慢的逻辑写在getCurrentColor()里.
    int redDiff = Math.abs(startRed - endRed);
    int greenDiff = Math.abs(startGreen - endGreen);
    int blueDiff = Math.abs(startBlue - endBlue);
    int colorDiff = redDiff + greenDiff + blueDiff;
    if (mCurrentRed != endRed) {
        mCurrentRed = getCurrentColor(startRed, endRed, colorDiff, 0,
                fraction);
        // getCurrentColor()决定如何根据差值来决定颜色变化的快慢 ->>关注1
    } else if (mCurrentGreen != endGreen) {
        mCurrentGreen = getCurrentColor(startGreen, endGreen, colorDiff,
                redDiff, fraction);
    } else if (mCurrentBlue != endBlue) {
        mCurrentBlue = getCurrentColor(startBlue, endBlue, colorDiff,
                redDiff + greenDiff, fraction);
    }
    // 将计算出的当前颜色的值组装返回
    String currentColor = "#" + getHexString(mCurrentRed)
            + getHexString(mCurrentGreen) + getHexString(mCurrentBlue);

    // 由于我们计算出的颜色是十进制数字，所以需要转换成十六进制字符串：调用getHexString()->>关注2
    // 最终将RGB颜色拼装起来，并作为最终的结果返回
    return currentColor;
}


// 关注1：getCurrentColor()
// 具体是根据fraction值来计算当前的颜色。

private int getCurrentColor(int startColor, int endColor, int colorDiff,
                            int offset, float fraction) {
```

```
    int currentColor;
    if (startColor > endColor) {
        currentColor = (int) (startColor - (fraction * colorDiff - offset));
        if (currentColor < endColor) {
            currentColor = endColor;
        }
    } else {
        currentColor = (int) (startColor + (fraction * colorDiff - offset));
        if (currentColor > endColor) {
            currentColor = endColor;
        }
    }
    return currentColor;
}

// 关注2:将10进制颜色值转换成16进制。
private String getHexString(int value) {
    String hexString = Integer.toHexString(value);
    if (hexString.length() == 1) {
        hexString = "0" + hexString;
    }
    return hexString;
}


}
```

步骤4：调用 `ObjectAnimator.ofObject()` 方法

```
    diyView = findViewById(R.id.diy_view);
    ObjectAnimator anim = ObjectAnimator.ofObject(diyView, "color", new ColorEvaluator(),
            "#0000FF", "#FF0000");
    // 设置自定义View对象、背景颜色属性值 & 颜色估值器
    // 本质逻辑：
    // 步骤1：根据颜色估值器不断 改变 值
    // 步骤2：调用set（）设置背景颜色的属性值（实际上是通过画笔进行颜色设置）
    // 步骤3：调用invalidate()刷新视图，即调用onDraw（）重新绘制，从而实现动画效果

    anim.setDuration(8000);
    anim.start();
```

效果

三、资源

- 本文部分转自 Carson_Ho 的 这份属性动画的核心使用类ValueAnimator学习指南请收好
- 本文 源码
- 本文 效果