## 为什么要引入属性动画？

Android之前的补间动画机制其实还算是比较健全的，在android.view.animation包下面有好多的类可以供我们操作，来完成一系列的动画效果，比如说对View进行移动、缩放、旋转和淡入淡出，并且我们还可以借助AnimationSet来将这些动画效果组合起来使用，除此之外还可以通过配置Interpolator来控制动画的播放速度等。

其实上面所谓的健全都是相对的，如果你的需求中只需要对View进行移动、缩放、旋转和淡入淡出操作，那么补间动画确实已经足够健全了。但是很显然，这些功能是不足以覆盖所有的场景的，一旦我们的需求超出了移动、缩放、旋转和淡入淡出这四种对View的操作，那么补间动画就不能再帮我们忙了，也就是说它在功能和可扩展方面都有相当大的局限性。

补间动画是只能够作用在View上的。也就是说，我们可以对一个Button、TextView、甚至是LinearLayout、或者其它任何继承自View的组件进行动画操作，但是如果我们想要对一个非View的对象进行动画操作，抱歉，补间动画就帮不上忙了。比如说我们有一个自定义的View，在这个View当中有一个Point对象用于管理坐标，然后在onDraw()方法当中就是根据这个Point对象的坐标值来进行绘制的。也就是说，如果我们可以对Point对象进行动画操作，那么整个自定义View的动画效果就有了。显然，补间动画是不具备这个功能的，这是它的第一个缺陷。

然后补间动画还有一个缺陷，就是它只能够实现移动、缩放、旋转和淡入淡出这四种动画操作，那如果我们希望可以对View的背景色进行动态地改变呢？很遗憾，我们只能靠自己去实现了。说白了，之前的补间动画机制就是使用硬编码的方式来完成的，功能限定死就是这些，基本上没有任何扩展性可言。

最后，补间动画还有一个致命的缺陷，就是它只是改变了View的显示效果而已，而不会真正去改变View的属性。比如说，现在屏幕的左上角有一个按钮，然后我们通过补间动画将它移动到了屏幕的右下角，现在你可以去尝试点击一下这个按钮，点击事件是绝对不会触发的，因为实际上这个按钮还是停留在屏幕的左上角，只不过补间动画将这个按钮绘制到了屏幕的右下角而已。

新引入的属性动画机制已经不再是针对于View来设计的了，也不限定于只能实现移动、缩放、旋转和淡入淡出这几种动画操作，同时也不再只是一种视觉上的动画效果了。它实际上是一种不断地对值进行操作的机制，并将值赋值到指定对象的指定属性上，可以是任意对象的任意属性。所以我们仍然可以将一个View进行移动或者缩放，但同时也可以对自定义View中的Point对象进行动画操作了。我们只需要告诉系统动画的运行时长，需要执行哪种类型的动画，以及动画的初始值和结束值，剩下的工作就可以全部交给系统去完成了。

既然属性动画的实现机制是通过对目标对象进行赋值并修改其属性来实现的，那么之前所说的按钮显示的问题也就不复存在了，如果我们通过属性动画来移动一个按钮，那么这个按钮就是真正的移动了，而不再是仅仅在另外一个位置绘制了而已。

## ValueAnimator

ValueAnimator是整个属性动画机制当中最核心的一个类，前面我们已经提到了，属性动画的运行机制是通过不断地对值进行操作来实现的，而初始值和结束值之间的动画过渡就是由ValueAnimator这个类来负责计算的。它的内部使用一种时间循环的机制来计算值与值之间的动画过渡，我们只需要将初始值和结束值提供给ValueAnimator，并且告诉它动画所需运行的时长，那么ValueAnimator就会自动帮我们完成从初始值平滑地过渡到结束值这样的效果。除此之外，ValueAnimator还负责管理动画的播放次数、播放模式、以及对动画设置监听器等。

但是ValueAnimator的用法却一点都不复杂，我们先从最简单的功能看起吧，比如说想要将一个值从0平滑过渡到1，时长300毫秒，就可以这样写：

1. ValueAnimator anim = ValueAnimator.ofFloat(0f, 1f);
2. anim.setDuration(300);
3. anim.start();

调用ValueAnimator的ofFloat()方法就可以构建出一个ValueAnimator的实例，ofFloat()方法当中允许传入多个float类型的参数，这里传入0和1就表示将值从0平滑过渡到1，然后调用ValueAnimator的setDuration()方法来设置动画运行的时长，最后调用start()方法启动动画。

可是这只是一个将值从0过渡到1的动画，又看不到任何界面效果，我们怎样才能知道这个动画是不是已经真正运行了呢？这就需要借助监听器来实现了，如下所示：

demo1：

1. ValueAnimator anim = ValueAnimator.ofFloat(0f, 1f);
2. anim.setDuration(300);
3. anim.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {
4. @Override
5. **public** **void** onAnimationUpdate(ValueAnimator animation) {
6. **float** currentValue = (**float**) animation.getAnimatedValue();
7. Log.d("TAG", "cuurent value is " + currentValue);
8. }
9. });
10. anim.start();

可以看到，这里我们通过addUpdateListener()方法来添加一个动画的监听器，在动画执行的过程中会不断地进行回调，我们只需要在回调方法当中将当前的值取出并打印出来，就可以知道动画有没有真正运行了。运行上述代码，控制台打印如下所示：

A description...

从打印日志的值我们就可以看出，ValueAnimator确实已经在正常工作了，值在300毫秒的时间内从0平滑过渡到了1，而这个计算工作就是由ValueAnimator帮助我们完成的。另外ofFloat()方法当中是可以传入任意多个参数的，因此我们还可以构建出更加复杂的动画逻辑，比如说将一个值在5秒内从0过渡到5，再过渡到3，再过渡到10，就可以这样写：

1. ValueAnimator anim = ValueAnimator.ofFloat(0f, 5f, 3f, 10f);
2. anim.setDuration(5000);
3. anim.start();

当然也许你并不需要小数位数的动画过渡，可能你只是希望将一个整数值从0平滑地过渡到100，那么也很简单，只需要调用ValueAnimator的ofInt()方法就可以了，如下所示：

1. ValueAnimator anim = ValueAnimator.ofInt(0, 100);

那么除此之外，我们还可以调用setStartDelay()方法来设置动画延迟播放的时间，调用setRepeatCount()和setRepeatMode()方法来设置动画循环播放的次数以及循环播放的模式，循环模式包括RESTART和REVERSE两种，分别表示重新播放和倒序播放的意思。

## ObjectAnimator

相比于ValueAnimator，ObjectAnimator可能才是我们最常接触到的类，因为ValueAnimator只不过是对值进行了一个平滑的动画过渡，但我们实际使用到这种功能的场景好像并不多。而ObjectAnimator则就不同了，它是可以直接对任意对象的任意属性进行动画操作的，比如说View的alpha属性。

不过虽说ObjectAnimator会更加常用一些，但是它其实是继承自ValueAnimator的，底层的动画实现机制也是基于ValueAnimator来完成的，因此ValueAnimator仍然是整个属性动画当中最核心的一个类。那么既然是继承关系，说明ValueAnimator中可以使用的方法在ObjectAnimator中也是可以正常使用的，它们的用法也非常类似，这里如果我们想要将一个TextView在5秒中内从常规变换成全透明，再从全透明变换成常规，就可以这样写：

demo2：

1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f, 1f);
2. animator.setDuration(5000);
3. animator.start();

可以看到，我们还是调用了ofFloat()方法来去创建一个ObjectAnimator的实例，只不过ofFloat()方法当中接收的参数有点变化了。这里第一个参数要求传入一个object对象，我们想要对哪个对象进行动画操作就传入什么，这里我传入了一个textview。第二个参数是想要对该对象的哪个属性进行动画操作，由于我们想要改变TextView的不透明度，因此这里传入"alpha"。后面的参数就是不固定长度了，想要完成什么样的动画就传入什么值，这里传入的值就表示将TextView从常规变换成全透明，再从全透明变换成常规。之后调用setDuration()方法来设置动画的时长，然后调用start()方法启动动画。

比如说我们想要将TextView进行一次360度的旋转，就可以这样写：

demo3：

1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "rotation", 0f, 360f);
2. animator.setDuration(5000);
3. animator.start();

可以看到，这里我们将第二个参数改成了"rotation"，然后将动画的初始值和结束值分别设置成0和360。

那么如果想要将TextView先向左移出屏幕，然后再移动回来，就可以这样写：

demo4：

1. **float** curTranslationX = textview.getTranslationX();
2. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "translationX", curTranslationX, -500f, curTranslationX);
3. animator.setDuration(5000);
4. animator.start();

这里我们先是调用了TextView的getTranslationX()方法来获取到当前TextView的translationX的位置，然后ofFloat()方法的第二个参数传入"translationX"，紧接着后面三个参数用于告诉系统TextView应该怎么移动。

然后我们还可以TextView进行缩放操作，比如说将TextView在垂直方向上放大3倍再还原，就可以这样写：

demo5：

1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "scaleY", 1f, 3f, 1f);
2. animator.setDuration(5000);
3. animator.start();

这里将ofFloat()方法的第二个参数改成了"scaleY"，表示在垂直方向上进行缩放。

我们可以传入任意的值到ofFloat()方法的第二个参数当中。因为ObjectAnimator在设计的时候就没有针对于View来进行设计，而是针对于任意对象的，它所负责的工作就是不断地向某个对象中的某个属性进行赋值，然后对象根据属性值的改变再来决定如何展现出来。

1. ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f);

其实这段代码的意思就是ObjectAnimator会帮我们不断地改变textview对象中alpha属性的值，从1f变化到0f。然后textview对象需要根据alpha属性值的改变来不断刷新界面的显示，从而让用户可以看出淡入淡出的动画效果。

那么textview对象中是不是有alpha属性这个值呢？没有，不仅textview没有这个属性，连它所有的父类也是没有这个属性的！这就奇怪了，textview当中并没有alpha这个属性，ObjectAnimator是如何进行操作的呢？其实ObjectAnimator内部的工作机制并不是直接对我们传入的属性名进行操作的，而是会去寻找这个属性名对应的get和set方法，因此alpha属性所对应的get和set方法应该就是：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/43536355) [copy](http://blog.csdn.net/guolin_blog/article/details/43536355) A description...A description...

1. **public** **void** setAlpha(**float** value);
2. **public** **float** getAlpha();

那么textview对象中是否有这两个方法呢？确实有，并且这两个方法是由View对象提供的，也就是说不仅TextView可以使用这个属性来进行淡入淡出动画操作，任何继承自View的对象都可以的。

## 组合动画

实现组合动画功能主要需要借助AnimatorSet这个类，这个类提供了一个play()方法，如果我们向这个方法中传入一个Animator对象(ValueAnimator或ObjectAnimator)将会返回一个AnimatorSet.Builder的实例，AnimatorSet.Builder中包括以下四个方法：

* after(Animator anim)   将现有动画插入到传入的动画之后执行
* after(long delay)   将现有动画延迟指定毫秒后执行
* before(Animator anim)   将现有动画插入到传入的动画之前执行
* with(Animator anim)   将现有动画和传入的动画同时执行

比如说我们想要让TextView先从屏幕外移动进屏幕，然后开始旋转360度，旋转的同时进行淡入淡出操作，就可以这样写：

demo6：

1. ObjectAnimator moveIn = ObjectAnimator.ofFloat(textview, "translationX", -500f, 0f);
2. ObjectAnimator rotate = ObjectAnimator.ofFloat(textview, "rotation", 0f, 360f);
3. ObjectAnimator fadeInOut = ObjectAnimator.ofFloat(textview, "alpha", 1f, 0f, 1f);
4. AnimatorSet animSet = **new** AnimatorSet();
5. animSet.play(rotate).with(fadeInOut).after(moveIn);
6. animSet.setDuration(5000);
7. animSet.start();

可以看到，这里我们先是把三个动画的对象全部创建出来，然后new出一个AnimatorSet对象之后将这三个动画对象进行播放排序，让旋转和淡入淡出动画同时进行，并把它们插入到了平移动画的后面，最后是设置动画时长以及启动动画。

## Animator监听器

在很多时候，我们希望可以监听到动画的各种事件，比如动画何时开始，何时结束，然后在开始或者结束的时候去执行一些逻辑处理。这个功能是完全可以实现的，Animator类当中提供了一个addListener()方法，这个方法接收一个AnimatorListener，我们只需要去实现这个AnimatorListener就可以监听动画的各种事件了。

ObjectAnimator是继承自ValueAnimator的，而ValueAnimator又是继承自Animator的，因此不管是ValueAnimator还是ObjectAnimator都是可以使用addListener()这个方法的。另外AnimatorSet也是继承自Animator的，因此addListener()这个方法算是个通用的方法。

添加一个监听器的代码如下所示：

1. anim.addListener(**new** AnimatorListener() {
2. @Override
3. **public** **void** onAnimationStart(Animator animation) {
4. }
6. @Override
7. **public** **void** onAnimationRepeat(Animator animation) {
8. }
10. @Override
11. **public** **void** onAnimationEnd(Animator animation) {
12. }
14. @Override
15. **public** **void** onAnimationCancel(Animator animation) {
16. }
17. });

可以看到，我们需要实现接口中的四个方法，onAnimationStart()方法会在动画开始的时候调用，onAnimationRepeat()方法会在动画重复执行的时候调用，onAnimationEnd()方法会在动画结束的时候调用，onAnimationCancel()方法会在动画被取消的时候调用。

但是也许很多时候我们并不想要监听那么多个事件，可能我只想要监听动画结束这一个事件，那么每次都要将四个接口全部实现一遍就显得非常繁琐。没关系，为此Android提供了一个适配器类，叫作AnimatorListenerAdapter，使用这个类就可以解决掉实现接口繁琐的问题了，如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/43536355) [copy](http://blog.csdn.net/guolin_blog/article/details/43536355) A description...A description...

1. anim.addListener(**new** AnimatorListenerAdapter() {
2. });

这里我们向addListener()方法中传入这个适配器对象，由于AnimatorListenerAdapter中已经将每个接口都实现好了，所以这里不用实现任何一个方法也不会报错。那么如果我想监听动画结束这个事件，就只需要单独重写这一个方法就可以了，如下所示：

1. anim.addListener(**new** AnimatorListenerAdapter() {
2. @Override
3. **public** **void** onAnimationEnd(Animator animation) {
4. }
5. });

## 

## 使用XML编写动画

通过XML来编写动画可能会比通过代码来编写动画要慢一些，但是在重用方面将会变得非常轻松，比如某个将通用的动画编写到XML里面，我们就可以在各个界面当中轻松去重用它。

如果想要使用XML来编写动画，首先要在res目录下面新建一个animator文件夹，所有属性动画的XML文件都应该存放在这个文件夹当中。然后在XML文件中我们一共可以使用如下三种标签：

* <animator>  对应代码中的ValueAnimator
* <objectAnimator>  对应代码中的ObjectAnimator
* <set>  对应代码中的AnimatorSet

那么比如说我们想要实现一个从0到100平滑过渡的动画，在XML当中就可以这样写：

demo7\_1：

1. **<animator** xmlns:android="http://schemas.android.com/apk/res/android"
2. android:valueFrom="0"
3. android:valueTo="100"
4. android:valueType="intType"**/>**

而如果我们想将一个视图的alpha属性从1变成0，就可以这样写：

demo7\_2：

1. **<objectAnimator** xmlns:android="http://schemas.android.com/apk/res/android"
2. android:valueFrom="1"
3. android:valueTo="0"
4. android:valueType="floatType"
5. android:propertyName="alpha"**/>**

另外，我们也可以使用XML来完成复杂的组合动画操作，比如将一个视图先从屏幕外移动进屏幕，然后开始旋转360度，旋转的同时进行淡入淡出操作，就可以这样写：

1. **<set** xmlns:android="http://schemas.android.com/apk/res/android"
2. android:ordering="sequentially" **>**
4. **<objectAnimator**
5. android:duration="2000"
6. android:propertyName="translationX"
7. android:valueFrom="-500"
8. android:valueTo="0"
9. android:valueType="floatType" **>**
10. **</objectAnimator>**
12. **<set** android:ordering="together" **>**
13. **<objectAnimator**
14. android:duration="3000"
15. android:propertyName="rotation"
16. android:valueFrom="0"
17. android:valueTo="360"
18. android:valueType="floatType" **>**
19. **</objectAnimator>**
21. **<set** android:ordering="sequentially" **>**
22. **<objectAnimator**
23. android:duration="1500"
24. android:propertyName="alpha"
25. android:valueFrom="1"
26. android:valueTo="0"
27. android:valueType="floatType" **>**
28. **</objectAnimator>**
29. **<objectAnimator**
30. android:duration="1500"
31. android:propertyName="alpha"
32. android:valueFrom="0"
33. android:valueTo="1"
34. android:valueType="floatType" **>**
35. **</objectAnimator>**
36. **</set>**
37. **</set>**
39. **</set>**

最后XML文件是编写好了，那么我们如何在代码中把文件加载进来并将动画启动呢？只需调用如下代码即可：

1. Animator animator = AnimatorInflater.loadAnimator(context, R.animator.anim\_file);
2. animator.setTarget(view);
3. animator.start();

调用AnimatorInflater的loadAnimator来将XML动画文件加载进来，然后再调用setTarget()方法将这个动画设置到某一个对象上面，最后再调用start()方法启动动画就可以了。

## TypeEvaluator

（demo8）

简单来说，TypeEvaluator就是告诉动画系统如何从初始值过度到结束值。ValueAnimator.ofFloat()方法实现了初始值与结束值之间的平滑过度，那么这个平滑过度是怎么做到的呢？其实就是系统内置了一个FloatEvaluator，它通过计算告知动画系统如何从初始值过度到结束值，我们来看一下FloatEvaluator的代码实现：

1. **public** **class** FloatEvaluator **implements** TypeEvaluator {
2. **public** Object evaluate(**float** fraction, Object startValue, Object endValue) {
3. **float** startFloat = ((Number) startValue).floatValue();
4. **return** startFloat + fraction \* (((Number) endValue).floatValue() - startFloat);
5. }
6. }

可以看到，FloatEvaluator实现了TypeEvaluator接口，然后重写evaluate()方法。evaluate()方法当中传入了三个参数，第一个参数fraction非常重要，这个参数用于表示动画的完成度的，我们应该根据它来计算当前动画的值应该是多少，第二第三个参数分别表示动画的初始值和结束值。那么上述代码的逻辑就比较清晰了，用结束值减去初始值，算出它们之间的差值，然后乘以fraction这个系数，再加上初始值，那么就得到当前动画的值了。

好的，那FloatEvaluator是系统内置好的功能，并不需要我们自己去编写，但介绍它的实现方法是要为我们后面的功能铺路的。前面我们使用过了ValueAnimator的ofFloat()和ofInt()方法，分别用于对浮点型和整型的数据进行动画操作的，但实际上ValueAnimator中还有一个ofObject()方法，是用于对任意对象进行动画操作的。但是相比于浮点型或整型数据，对象的动画操作明显要更复杂一些，因为系统将完全无法知道如何从初始对象过度到结束对象，因此这个时候我们就需要实现一个自己的TypeEvaluator来告知系统如何进行过度。

下面来先定义一个Point类，如下所示：

1. **public** **class** Point {
3. **private** **float** x;
5. **private** **float** y;
7. **public** Point(**float** x, **float** y) {
8. **this**.x = x;
9. **this**.y = y;
10. }
12. **public** **float** getX() {
13. **return** x;
14. }
16. **public** **float** getY() {
17. **return** y;
18. }
20. }

Point类非常简单，只有x和y两个变量用于记录坐标的位置，并提供了构造方法来设置坐标，以及get方法来获取坐标。接下来定义PointEvaluator，如下所示：

1. **public** **class** PointEvaluator **implements** TypeEvaluator{
3. @Override
4. **public** Object evaluate(**float** fraction, Object startValue, Object endValue) {
5. Point startPoint = (Point) startValue;
6. Point endPoint = (Point) endValue;
7. **float** x = startPoint.getX() + fraction \* (endPoint.getX() - startPoint.getX());
8. **float** y = startPoint.getY() + fraction \* (endPoint.getY() - startPoint.getY());
9. Point point = **new** Point(x, y);
10. **return** point;
11. }
13. }

可以看到，PointEvaluator同样实现了TypeEvaluator接口并重写了evaluate()方法。其实evaluate()方法中的逻辑还是非常简单的，先是将startValue和endValue强转成Point对象，然后同样根据fraction来计算当前动画的x和y的值，最后组装到一个新的Point对象当中并返回。

这样我们就将PointEvaluator编写完成了，接下来我们就可以非常轻松地对Point对象进行动画操作了，比如说我们有两个Point对象，现在需要将Point1通过动画平滑过度到Point2，就可以这样写：

1. Point point1 = **new** Point(0, 0);
2. Point point2 = **new** Point(300, 300);
3. ValueAnimator anim = ValueAnimator.ofObject(**new** PointEvaluator(), point1, point2);
4. anim.setDuration(5000);
5. anim.start();

代码很简单，这里我们先是new出了两个Point对象，并在构造函数中分别设置了它们的坐标点。然后调用ValueAnimator的ofObject()方法来构建ValueAnimator的实例，这里需要注意的是，ofObject()方法要求多传入一个TypeEvaluator参数，这里我们只需要传入刚才定义好的PointEvaluator的实例就可以了。

新建一个MyAnimView继承自View，代码如下所示：

1. **public** **class** MyAnimView **extends** View {
3. **public** **static** **final** **float** RADIUS = 50f;
5. **private** Point currentPoint;
7. **private** Paint mPaint;
9. **public** MyAnimView(Context context, AttributeSet attrs) {
10. **super**(context, attrs);
11. mPaint = **new** Paint(Paint.ANTI\_ALIAS\_FLAG);
12. mPaint.setColor(Color.BLUE);
13. }
15. @Override
16. **protected** **void** onDraw(Canvas canvas) {
17. **if** (currentPoint == **null**) {
18. currentPoint = **new** Point(RADIUS, RADIUS);
19. drawCircle(canvas);
20. startAnimation();
21. } **else** {
22. drawCircle(canvas);
23. }
24. }
26. **private** **void** drawCircle(Canvas canvas) {
27. **float** x = currentPoint.getX();
28. **float** y = currentPoint.getY();
29. canvas.drawCircle(x, y, RADIUS, mPaint);
30. }
32. **private** **void** startAnimation() {
33. Point startPoint = **new** Point(RADIUS, RADIUS);
34. Point endPoint = **new** Point(getWidth() - RADIUS, getHeight() - RADIUS);
35. ValueAnimator anim = ValueAnimator.ofObject(**new** PointEvaluator(), startPoint, endPoint);
36. anim.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {
37. @Override
38. **public** **void** onAnimationUpdate(ValueAnimator animation) {
39. currentPoint = (Point) animation.getAnimatedValue();
40. invalidate();
41. }
42. });
43. anim.setDuration(5000);
44. anim.start();
45. }
47. }

首先在自定义View的构造方法当中初始化了一个Paint对象作为画笔，并将画笔颜色设置为蓝色，接着在onDraw()方法当中进行绘制。这里我们绘制的逻辑是由currentPoint这个对象控制的，如果currentPoint对象不等于空，那么就调用drawCircle()方法在currentPoint的坐标位置画出一个半径为50的圆，如果currentPoint对象是空，那么就调用startAnimation()方法来启动动画。

那么我们来观察一下startAnimation()方法中的代码，就是对Point对象进行了一个动画操作而已。这里我们定义了一个startPoint和一个endPoint，坐标分别是View的左上角和右下角，并将动画的时长设为5秒。有一点大家注意的是，我们通过监听器对动画的过程进行了监听，每当Point值有改变的时候都会回调onAnimationUpdate()方法。在这个方法当中，我们对currentPoint对象进行了重新赋值，并调用了invalidate()方法，这样的话onDraw()方法就会重新调用，并且由于currentPoint对象的坐标已经改变了，那么绘制的位置也会改变，于是一个平移的动画效果也就实现了。

下面我们只需要在布局文件当中引入这个自定义控件：

1. **<RelativeLayout** xmlns:android="http://schemas.android.com/apk/res/android"
2. android:layout\_width="match\_parent"
3. android:layout\_height="match\_parent"
4. **>**
6. **<com.example.tony.myapplication.MyAnimView**
7. android:layout\_width="match\_parent"
8. android:layout\_height="match\_parent" **/>**
10. **</RelativeLayout>**

## ObjectAnimator扩展

（demo 9）

ObjectAnimator内部的工作机制是通过寻找特定属性的get和set方法，然后通过方法不断地对值进行改变，从而实现动画效果的。因此我们就需要在MyAnimView中定义一个color属性，并提供它的get和set方法。这里我们可以将color属性设置为字符串类型，使用#RRGGBB这种格式来表示颜色值，代码如下所示：

1. **public** **class** MyAnimView **extends** View {
3. ...
5. **private** String color;
7. **public** String getColor() {
8. **return** color;
9. }
11. **public** **void** setColor(String color) {
12. **this**.color = color;
13. mPaint.setColor(Color.parseColor(color));
14. invalidate();
15. }
17. ...
19. }

注意在setColor()方法当中，我们编写了一个非常简单的逻辑，就是将画笔的颜色设置成方法参数传入的颜色，然后调用了invalidate()方法。这段代码虽然只有三行，但是却执行了一个非常核心的功能，就是在改变了画笔颜色之后立即刷新视图，然后onDraw()方法就会调用。在onDraw()方法当中会根据当前画笔的颜色来进行绘制，这样颜色也就会动态进行改变了。

那么接下来的问题就是怎样让setColor()方法得到调用了，毫无疑问，当然是要借助ObjectAnimator类，但是在使用ObjectAnimator之前我们还要完成一个非常重要的工作，就是编写一个用于告知系统如何进行颜色过度的TypeEvaluator。创建ColorEvaluator并实现TypeEvaluator接口，代码如下所示：

1. **public** **class** ColorEvaluator **implements** TypeEvaluator {
3. **private** **int** mCurrentRed = -1;
5. **private** **int** mCurrentGreen = -1;
7. **private** **int** mCurrentBlue = -1;
9. @Override
10. **public** Object evaluate(**float** fraction, Object startValue, Object endValue) {
11. String startColor = (String) startValue;
12. String endColor = (String) endValue;
13. **int** startRed = Integer.parseInt(startColor.substring(1, 3), 16);
14. **int** startGreen = Integer.parseInt(startColor.substring(3, 5), 16);
15. **int** startBlue = Integer.parseInt(startColor.substring(5, 7), 16);
16. **int** endRed = Integer.parseInt(endColor.substring(1, 3), 16);
17. **int** endGreen = Integer.parseInt(endColor.substring(3, 5), 16);
18. **int** endBlue = Integer.parseInt(endColor.substring(5, 7), 16);
19. // 初始化颜色的值
20. **if** (mCurrentRed == -1) {
21. mCurrentRed = startRed;
22. }
23. **if** (mCurrentGreen == -1) {
24. mCurrentGreen = startGreen;
25. }
26. **if** (mCurrentBlue == -1) {
27. mCurrentBlue = startBlue;
28. }
29. // 计算初始颜色和结束颜色之间的差值
30. **int** redDiff = Math.abs(startRed - endRed);
31. **int** greenDiff = Math.abs(startGreen - endGreen);
32. **int** blueDiff = Math.abs(startBlue - endBlue);
33. **int** colorDiff = redDiff + greenDiff + blueDiff;
34. **if** (mCurrentRed != endRed) {
35. mCurrentRed = getCurrentColor(startRed, endRed, colorDiff, 0,
36. fraction);
37. } **else** **if** (mCurrentGreen != endGreen) {
38. mCurrentGreen = getCurrentColor(startGreen, endGreen, colorDiff,
39. redDiff, fraction);
40. } **else** **if** (mCurrentBlue != endBlue) {
41. mCurrentBlue = getCurrentColor(startBlue, endBlue, colorDiff,
42. redDiff + greenDiff, fraction);
43. }
44. // 将计算出的当前颜色的值组装返回
45. String currentColor = "#" + getHexString(mCurrentRed)
46. + getHexString(mCurrentGreen) + getHexString(mCurrentBlue);
47. **return** currentColor;
48. }
50. /\*\*
51. \* 根据fraction值来计算当前的颜色。
52. \*/
53. **private** **int** getCurrentColor(**int** startColor, **int** endColor, **int** colorDiff,
54. **int** offset, **float** fraction) {
55. **int** currentColor;
56. **if** (startColor > endColor) {
57. currentColor = (**int**) (startColor - (fraction \* colorDiff - offset));
58. **if** (currentColor < endColor) {
59. currentColor = endColor;
60. }
61. } **else** {
62. currentColor = (**int**) (startColor + (fraction \* colorDiff - offset));
63. **if** (currentColor > endColor) {
64. currentColor = endColor;
65. }
66. }
67. **return** currentColor;
68. }
70. /\*\*
71. \* 将10进制颜色值转换成16进制。
72. \*/
73. **private** String getHexString(**int** value) {
74. String hexString = Integer.toHexString(value);
75. **if** (hexString.length() == 1) {
76. hexString = "0" + hexString;
77. }
78. **return** hexString;
79. }
81. }

首先在evaluate()方法当中获取到颜色的初始值和结束值，并通过字符串截取的方式将颜色分为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。接下来计算一下初始颜色值到结束颜色值之间的差值，这个差值很重要，决定着颜色变化的快慢，如果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，而如果颜色值相差很大，比如说从黑到白，那么就要经历255\*3这个幅度的颜色过度，变化就会非常快。

那么控制颜色变化的速度是通过getCurrentColor()这个方法来实现的，这个方法会根据当前的fraction值来计算目前应该过度到什么颜色，并且这里会根据初始和结束的颜色差值来控制变化速度，最终将计算出的颜色进行返回。

最后，由于我们计算出的颜色是十进制数字，这里还需要调用一下getHexString()方法把它们转换成十六进制字符串，再将RGB颜色拼装起来之后作为最终的结果返回。

比如说我们想要实现从蓝色到红色的动画过度，历时5秒，就可以这样写：

1. ObjectAnimator anim = ObjectAnimator.ofObject(myAnimView, "color", **new** ColorEvaluator(),
2. "#0000FF", "#FF0000");
3. anim.setDuration(5000);
4. anim.start();

接下来我们需要将上面一段代码移到MyAnimView类当中，让它和刚才的Point移动动画可以结合到一起播放。修改MyAnimView中的代码，如下所示：

1. **public** **class** MyAnimView **extends** View {
3. ...
5. **private** **void** startAnimation() {
6. Point startPoint = **new** Point(RADIUS, RADIUS);
7. Point endPoint = **new** Point(getWidth() - RADIUS, getHeight() - RADIUS);
8. ValueAnimator anim = ValueAnimator.ofObject(**new** PointEvaluator(), startPoint, endPoint);
9. anim.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {
10. @Override
11. **public** **void** onAnimationUpdate(ValueAnimator animation) {
12. currentPoint = (Point) animation.getAnimatedValue();
13. invalidate();
14. }
15. });
16. ObjectAnimator anim2 = ObjectAnimator.ofObject(**this**, "color", **new** ColorEvaluator(),
17. "#0000FF", "#FF0000");
18. AnimatorSet animSet = **new** AnimatorSet();
19. animSet.play(anim).with(anim2);
20. animSet.setDuration(5000);
21. animSet.start();
22. }
24. }

可以看到，我们并没有改动太多的代码，重点只是修改了startAnimation()方法中的部分内容。这里先是将颜色过度的代码逻辑移动到了startAnimation()方法当中。接着我们又创建了一个AnimatorSet，并把两个动画设置成同时播放，动画时长为五秒，最后启动动画。

## Interpolator的用法

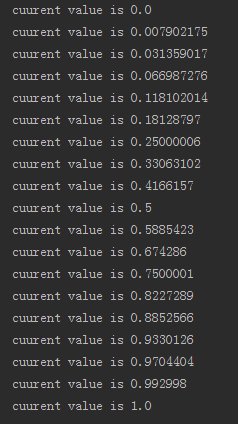
Interpolator直译过来是补间器的意思，它的主要作用是可以控制动画的变化速率，比如去实现一种非线性运动的动画效果。那么什么叫做非线性运动的动画效果呢？就是说动画改变的速率不是一成不变的，像加速运动以及减速运动都属于非线性运动。

不过Interpolator并不是属性动画中新增的技术，实际上从Android 1.0版本开始就一直存在Interpolator接口了，而之前的补间动画当然也是支持这个功能的。只不过在属性动画中新增了一个TimeInterpolator接口，这个接口是用于兼容之前的Interpolator的，这使得所有过去的Interpolator实现类都可以直接拿过来放到属性动画当中使用，那么我们来看一下现在TimeInterpolator接口的所有实现类，如下图所示：

A description...

可以看到，TimeInterpolator接口已经有非常多的实现类了，这些都是Android系统内置好的并且我们可以直接使用的Interpolator。每个Interpolator都有它各自的实现效果，比如说AccelerateInterpolator就是一个加速运动的Interpolator，而DecelerateInterpolator就是一个减速运动的Interpolator。

在前面使用ValueAnimator所打印的值如下所示：



可以看到，一开始的值变化速度明显比较慢，仅0.0开头的就打印了4次，之后开始加速，最后阶段又开始减速，因此我们可以很明显地看出这一个先加速后减速的Interpolator。

使用属性动画时，系统默认的Interpolator其实就是一个先加速后减速的Interpolator，对应的实现类就是AccelerateDecelerateInterpolator。

当然，我们也可以很轻松地修改这一默认属性，将它替换成任意一个系统内置好的Interpolator。MyAnimView中的startAnimation()方法是开启动画效果的入口，这里我们对Point对象的坐标稍做一下修改，让它变成一种垂直掉落的效果，代码如下所示：

demo10：

1. **private** **void** startAnimation() {
2. Point startPoint = **new** Point(getWidth() / 2, RADIUS);
3. Point endPoint = **new** Point(getWidth() / 2, getHeight() - RADIUS);
4. ValueAnimator anim = ValueAnimator.ofObject(**new** PointEvaluator(), startPoint, endPoint);
5. anim.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {
6. @Override
7. **public** **void** onAnimationUpdate(ValueAnimator animation) {
8. currentPoint = (Point) animation.getAnimatedValue();
9. invalidate();
10. }
11. });
12. anim.setDuration(2000);
13. anim.start();
14. }

这里主要是对Point构造函数中的坐标值进行了一下改动，那么现在小球运动的动画效果应该是从屏幕正中央的顶部掉落到底部。但是现在默认情况下小球的下降速度肯定是先加速后减速的，这不符合物理的常识规律，如果把小球视为一个自由落体的话，那么下降的速度应该是越来越快的。只需要调用Animator的setInterpolator()方法就可以了，这个方法要求传入一个实现TimeInterpolator接口的实例，那么比如说我们想要实现小球下降越来越快的效果，就可以使用AccelerateInterpolator，代码如下所示：

1. **private** **void** startAnimation() {
2. Point startPoint = **new** Point(getWidth() / 2, RADIUS);
3. Point endPoint = **new** Point(getWidth() / 2, getHeight() - RADIUS);
4. ValueAnimator anim = ValueAnimator.ofObject(**new** PointEvaluator(), startPoint, endPoint);
5. anim.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {
6. @Override
7. **public** **void** onAnimationUpdate(ValueAnimator animation) {
8. currentPoint = (Point) animation.getAnimatedValue();
9. invalidate();
10. }
11. });
12. anim.setInterpolator(**new** AccelerateInterpolator(2f));
13. anim.setDuration(2500);
14. anim.start();
15. }

代码很简单，这里调用了setInterpolator()方法，然后传入了一个AccelerateInterpolator的实例，注意AccelerateInterpolator的构建函数可以接收一个float类型的参数，这个参数是用于控制加速度的。

但是现在的动画效果看上去仍然是怪怪的，因为一个小球从很高的地方掉落到地面上直接就静止了，这也是不符合物理规律的，小球撞击到地面之后应该要反弹起来，然后再次落下，接着再反弹起来，又再次落下，以此反复，最后静止。这个功能我们当然可以自己去写，只不过比较复杂，所幸的是，Android系统中已经提供好了这样一种Interpolator，我们只需要简单地替换一下就可以完成上面的描述的效果，代码如下所示：

1. **private** **void** startAnimation() {
2. Point startPoint = **new** Point(getWidth() / 2, RADIUS);
3. Point endPoint = **new** Point(getWidth() / 2, getHeight() - RADIUS);
4. ValueAnimator anim = ValueAnimator.ofObject(**new** PointEvaluator(), startPoint, endPoint);
5. anim.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {
6. @Override
7. **public** **void** onAnimationUpdate(ValueAnimator animation) {
8. currentPoint = (Point) animation.getAnimatedValue();
9. invalidate();
10. }
11. });
12. anim.setInterpolator(**new** BounceInterpolator());
13. anim.setDuration(3000);
14. anim.start();
15. }

可以看到，我们只是将设置的Interpolator换成了BounceInterpolator的实例，而BounceInterpolator就是一种可以模拟物理规律，实现反复弹起效果的Interpolator。另外还将整体的动画时间稍微延长了一点，因为小球反复弹起需要比之前更长的时间。

下面我们来看一下Interpolator的内部实现机制是什么样的，并且来尝试写一个自定义的Interpolator。

首先看一下TimeInterpolator的接口定义，代码如下所示：

1. /\*\*
2. \* A time interpolator defines the rate of change of an animation. This allows animations
3. \* to have non-linear motion, such as acceleration and deceleration.
4. \*/
5. **public** **interface** TimeInterpolator {
7. /\*\*
8. \* Maps a value representing the elapsed fraction of an animation to a value that represents
9. \* the interpolated fraction. This interpolated value is then multiplied by the change in
10. \* value of an animation to derive the animated value at the current elapsed animation time.
11. \*
12. \* @param input A value between 0 and 1.0 indicating our current point
13. \*        in the animation where 0 represents the start and 1.0 represents
14. \*        the end
15. \* @return The interpolation value. This value can be more than 1.0 for
16. \*         interpolators which overshoot their targets, or less than 0 for
17. \*         interpolators that undershoot their targets.
18. \*/
19. **float** getInterpolation(**float** input);
20. }

接口还是非常简单的，只有一个getInterpolation()方法。getInterpolation()方法中接收一个input参数，这个参数的值会随着动画的运行而不断变化，不过它的变化是非常有规律的，就是根据设定的动画时长匀速增加，变化范围是0到1。也就是说当动画一开始的时候input的值是0，到动画结束的时候input的值是1，而中间的值则是随着动画运行的时长在0到1之间变化的。

说到这个input的值，我们想起前面使用过的fraction值。那么这里的input和fraction有什么关系或者区别呢？答案很简单，input的值决定了fraction的值。input的值是由系统经过计算后传入到getInterpolation()方法中的，然后我们可以自己实现getInterpolation()方法中的[**算法**](http://lib.csdn.net/base/datastructure)，根据input的值来计算出一个返回值，而这个返回值就是fraction了。

因此，最简单的情况就是input值和fraction值是相同的，这种情况由于input值是匀速增加的，因而fraction的值也是匀速增加的，所以动画的运动情况也是匀速的。系统中内置的LinearInterpolator就是一种匀速运动的Interpolator，那么我们来看一下它的源码是怎么实现的：

1. /\*\*
2. \* An interpolator where the rate of change is constant
3. \*/
4. @HasNativeInterpolator
5. **public** **class** LinearInterpolator **extends** BaseInterpolator **implements** NativeInterpolatorFactory {
7. **public** LinearInterpolator() {
8. }
10. **public** LinearInterpolator(Context context, AttributeSet attrs) {
11. }
13. **public** **float** getInterpolation(**float** input) {
14. **return** input;
15. }
17. /\*\* @hide \*/
18. @Override
19. **public** **long** createNativeInterpolator() {
20. **return** NativeInterpolatorFactoryHelper.createLinearInterpolator();
21. }
22. }

这里我们只看getInterpolation()方法，这个方法没有任何逻辑，就是把参数中传递的input值直接返回了，因此fraction的值就是等于input的值的，这就是匀速运动的Interpolator的实现方式。

当然这是最简单的一种Interpolator的实现了，我们再来看一个稍微复杂一点的。既然现在大家都知道了系统在默认情况下使用的是AccelerateDecelerateInterpolator，那我们就来看一下它的源码吧，如下所示：

1. /\*\*
2. \* An interpolator where the rate of change starts and ends slowly but
3. \* accelerates through the middle.
4. \*
5. \*/
6. @HasNativeInterpolator
7. **public** **class** AccelerateDecelerateInterpolator **implements** Interpolator, NativeInterpolatorFactory {
8. **public** AccelerateDecelerateInterpolator() {
9. }
11. @SuppressWarnings({"UnusedDeclaration"})
12. **public** AccelerateDecelerateInterpolator(Context context, AttributeSet attrs) {
13. }
15. **public** **float** getInterpolation(**float** input) {
16. **return** (**float**)(Math.cos((input + 1) \* Math.PI) / 2.0f) + 0.5f;
17. }
19. /\*\* @hide \*/
20. @Override
21. **public** **long** createNativeInterpolator() {
22. **return** NativeInterpolatorFactoryHelper.createAccelerateDecelerateInterpolator();
23. }
24. }

代码虽然没有变长很多，但是getInterpolation()方法中的逻辑已经明显变复杂了，不再是简单地将参数中的input进行返回，而是进行了一个较为复杂的数学运算。那这里我们来分析一下它的算法实现，可以看到，算法中主要使用了余弦函数，由于input的取值范围是0到1，那么cos函数中的取值范围就是π到2π。而cos(π)的结果是-1，cos(2π)的结果是1，那么这个值再除以2加上0.5之后，getInterpolation()方法最终返回的结果值还是在0到1之间。只不过经过了余弦运算之后，最终的结果不再是匀速增加的了，而是经历了一个先加速后减速的过程。我们可以将这个算法的执行情况通过曲线图的方式绘制出来，结果如下图所示：

A description...

可以看到，这是一个S型的曲线图，当横坐标从0变化到0.2的时候，纵坐标的变化幅度很小，但是之后就开始明显加速，最后横坐标从0.8变化到1的时候，纵坐标的变化幅度又变得很小。

通过分析LinearInterpolator和AccelerateDecelerateInterpolator的源码，我们已经对Interpolator的内部实现机制有了比较清楚的认识了，那么接下来我们就开始尝试编写一个自定义的Interpolator。

编写自定义Interpolator最主要的难度都是在于数学计算方面的。既然属性动画默认的Interpolator是先加速后减速的一种方式，这里我们就对它进行一个简单的修改，让它变成先减速后加速的方式。新建DecelerateAccelerateInterpolator类，让它实现TimeInterpolator接口，代码如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/44171115) [copy](http://blog.csdn.net/guolin_blog/article/details/44171115) A description...A description...

1. **public** **class** DecelerateAccelerateInterpolator **implements** TimeInterpolator{
3. @Override
4. **public** **float** getInterpolation(**float** input) {
5. **float** result;
6. **if** (input <= 0.5) {
7. result = (**float**) (Math.sin(Math.PI \* input)) / 2;
8. } **else** {
9. result = (**float**) (2 - Math.sin(Math.PI \* input)) / 2;
10. }
11. **return** result;
12. }
14. }

这段代码是使用正弦函数来实现先减速后加速的功能的，因为正弦函数初始弧度的变化值非常大，刚好和余弦函数是相反的，而随着弧度的增加，正弦函数的变化值也会逐渐变小，这样也就实现了减速的效果。当弧度大于π/2之后，整个过程相反了过来，现在正弦函数的弧度变化值非常小，渐渐随着弧度继续增加，变化值越来越大，弧度到π时结束，这样从0过度到π，也就实现了先减速后加速的效果。

同样我们可以将这个算法的执行情况通过曲线图的方式绘制出来，结果如下图所示：

A description...

可以看到，这也是一个S型的曲线图，只不过曲线的方向和刚才是相反的。从上图中我们可以很清楚地看出来，一开始纵坐标的变化幅度很大，然后逐渐变小，横坐标到0.5的时候纵坐标变化幅度趋近于零，之后随着横坐标继续增加纵坐标的变化幅度又开始变大，的确是先减速后加速的效果。

那么现在我们将DecelerateAccelerateInterpolator在代码中进行替换，如下所示：

1. **private** **void** startAnimation() {
2. Point startPoint = **new** Point(getWidth() / 2, RADIUS);
3. Point endPoint = **new** Point(getWidth() / 2, getHeight() - RADIUS);
4. ValueAnimator anim = ValueAnimator.ofObject(**new** PointEvaluator(), startPoint, endPoint);
5. anim.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {
6. @Override
7. **public** **void** onAnimationUpdate(ValueAnimator animation) {
8. currentPoint = (Point) animation.getAnimatedValue();
9. invalidate();
10. }
11. });
12. anim.setInterpolator(**new** DecelerateAccelerateInterpolator());
13. anim.setDuration(3000);
14. anim.start();
15. }

## ViewPropertyAnimator的用法

ViewPropertyAnimator的用法格外的简单，只不过和前面所学的所有属性动画的知识不同，它并不是在3.0系统当中引入的，而是在3.1系统当中附增的一个新的功能。

我们都知道，属性动画的机制已经不是再针对于View而进行设计的了，而是一种不断地对值进行操作的机制，它可以将值赋值到指定对象的指定属性上。但是，在绝大多数情况下，我相信大家主要都还是对View进行动画操作的。Android开发团队也是意识到了这一点，没有为View的动画操作提供一种更加便捷的用法确实是有点太不人性化了，于是在Android 3.1系统当中补充了ViewPropertyAnimator这个机制。

那我们先来回顾一下之前的用法吧，比如我们想要让一个TextView从常规状态变成透明状态，就可以这样写：

1. ObjectAnimator animator = ObjectAnimator.ofFloat(textview, "alpha", 0f);
2. animator.start();

看上去复杂吗？好像也不怎么复杂，但确实也不怎么容易理解。我们要将操作的view、属性、变化的值都一起传入到ObjectAnimator.ofFloat()方法当中，虽然看上去也没写几行代码，但这不太像是我们平时使用的面向对象的思维。

那么下面我们就来看一下如何使用ViewPropertyAnimator来实现同样的效果，ViewPropertyAnimator提供了更加易懂、更加面向对象的API，如下所示：

1. textview.animate().alpha(0f);

果然非常简单！不过textview.animate()这个方法是怎么回事呢？animate()方法就是在Android 3.1系统上新增的一个方法，这个方法的返回值是一个ViewPropertyAnimator对象，也就是说拿到这个对象之后我们就可以调用它的各种方法来实现动画效果了，这里我们调用了alpha()方法并转入0，表示将当前的textview变成透明状态。

除此之外，ViewPropertyAnimator还可以很轻松地将多个动画组合到一起，比如我们想要让textview运动到500,500这个坐标点上，就可以这样写：

1. textview.animate().x(500).y(500);

可以看出，ViewPropertyAnimator是支持连缀用法的，我们想让textview移动到横坐标500这个位置上时调用了x(500)这个方法，然后让textview移动到纵坐标500这个位置上时调用了y(500)这个方法，将所有想要组合的动画通过这种连缀的方式拼接起来，这样全部动画就都会一起被执行。

那么怎样去设定动画的运行时长呢？很简单，也是通过连缀的方式设定即可，比如我们想要让动画运行5秒钟，就可以这样写：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/44171115) [copy](http://blog.csdn.net/guolin_blog/article/details/44171115) A description...A description...

1. textview.animate().x(500).y(500).setDuration(5000);

除此之外，Interpolator技术我们也可以应用在ViewPropertyAnimator上面，如下所示：

1. textview.animate().x(500).y(500).setDuration(5000)
2. .setInterpolator(**new** BounceInterpolator());

用法很简单，同样也是使用连缀的方式。ViewPropertyAnimator其实并没有什么太多的技巧可言，用法基本都是大同小异的，需要用到什么功能就连缀一下，因此更多的用法大家只需要去查阅一下文档，看看还支持哪些功能，有哪些接口可以调用就可以了。

那么除了用法之外，关于ViewPropertyAnimator有几个细节还是值得大家注意一下的：

* 整个ViewPropertyAnimator的功能都是建立在View类新增的animate()方法之上的，这个方法会创建并返回一个ViewPropertyAnimator的实例，之后的调用的所有方法，设置的所有属性都是通过这个实例完成的。
* 大家注意到，在使用ViewPropertyAnimator时，我们自始至终没有调用过start()方法，这是因为新的接口中使用了隐式启动动画的功能，只要我们将动画定义完成之后，动画就会自动启动。并且这个机制对于组合动画也同样有效，只要我们不断地连缀新的方法，那么动画就不会立刻执行，等到所有在ViewPropertyAnimator上设置的方法都执行完毕后，动画就会自动启动。当然如果不想使用这一默认机制的话，我们也可以显式地调用start()方法来启动动画。
* ViewPropertyAnimator的所有接口都是使用连缀的语法来设计的，每个方法的返回值都是它自身的实例，因此调用完一个方法之后可以直接连缀调用它的另一个方法，这样把所有的功能都串接起来，我们甚至可以仅通过一行代码就完成任意复杂度的动画功能。

###### 布局动画（Layout Animations）

主要使用LayoutTransition为布局的容器设置动画，当容器中的视图层次发生变化时存在过渡的动画效果。

基本代码为：

1. LayoutTransition transition = **new** LayoutTransition();
2. transition.setAnimator(LayoutTransition.CHANGE\_APPEARING,
3. transition.getAnimator(LayoutTransition.CHANGE\_APPEARING));
4. transition.setAnimator(LayoutTransition.APPEARING,
5. **null**);
6. transition.setAnimator(LayoutTransition.DISAPPEARING,
7. **null**);
8. transition.setAnimator(LayoutTransition.CHANGE\_DISAPPEARING,
9. **null**);
10. mGridLayout.setLayoutTransition(transition);

过渡的类型一共有四种：

LayoutTransition.APPEARING 当一个View在ViewGroup中出现时，对此View设置的动画

LayoutTransition.CHANGE\_APPEARING 当一个View在ViewGroup中出现时，对此View对其他View位置造成影响，对其他View设置的动画

LayoutTransition.DISAPPEARING  当一个View在ViewGroup中消失时，对此View设置的动画

LayoutTransition.CHANGE\_DISAPPEARING 当一个View在ViewGroup中消失时，对此View对其他View位置造成影响，对其他View设置的动画

LayoutTransition.CHANGE 不是由于View出现或消失造成对其他View位置造成影响，然后对其他View设置的动画。

注意动画到底设置在谁身上，此View还是其他View。