

# Android 动画

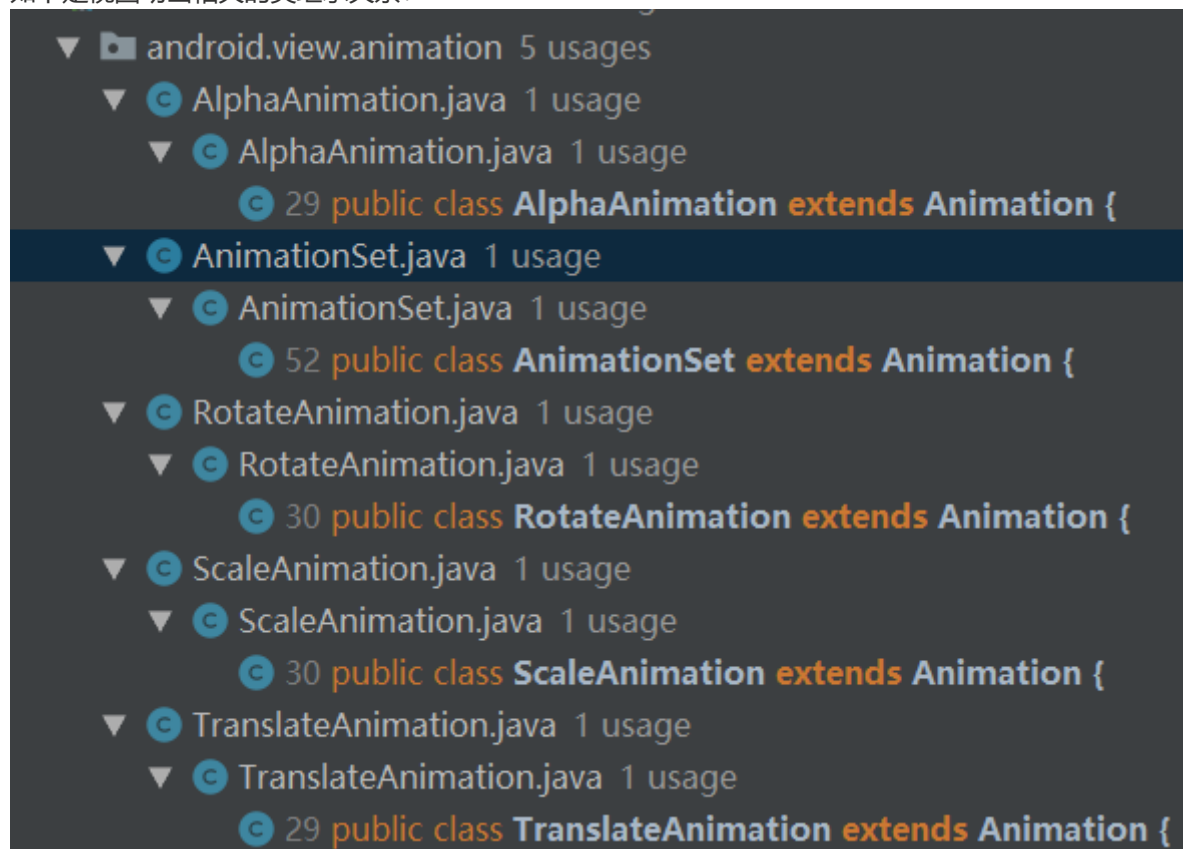
## View Animation（视图动画）

### 概述:

视图动画，也叫 Tween（补间）动画可以在一个视图容器内执行一系列简单变换（位置、大小、旋转、透明度）。譬如，如果你有一个 TextView 对象，您可以移动、旋转、缩放、透明度设置其文本，当然，如果它有一个背景图像，背景图像会随着文本变化。

补间动画通过 XML 或 Android 代码定义，建议使用 XML 文件定义，因为它更具可读性、可重用性

如下是视图动画相关的类继承关系：



java 类名	xml 关键字	描述信息
AlphaAnimation	<alpha> 放置在 res/anim/ 目录下	渐变透明度动画效果
RotateAnimation	<rotate> 放置在 res/anim/ 目录下	画面转移旋转动画效果
ScaleAnimation	<scale> 放置在 res/anim/ 目录下	渐变尺寸伸缩动画效果
TranslateAnimation	<translate> 放置在 res/anim/ 目录下	画面转换位置移动动画效果
AnimationSet	<set> 放置在 res/anim/ 目录下	一个持有其它动画元素 alpha、scale、translate、rotate 或者其它 set 元素的容器

通过上图和上表可以直观的看出来补间动画的关系及种类了，接下来我们就详细一个一个的介绍一下各种补间动画。

## 2. 视图动画详细说明

可以看出来 Animation 抽象类是所有补间动画类的基类，所以基类会提供一些通用的动画属性方法，如下我们就来详细看看这些属性，。

### 2-1. Animation 属性详解

xml 属性	java 方法	解释
android:detachWallpaper	setDetachWallpaper(boolean)	是否在壁纸上运行
android:duration	setDuration(long)	动画持续时间，毫秒为单位
android:fillAfter	setFillAfter(boolean)	控件动画结束时是否保持动画最后的状态
android:fillBefore	setFillBefore(boolean)	控件动画结束时是否还原到开始动画前的状态
android:fillEnabled	setFillEnabled(boolean)	与android:fillBefore效果相同
android:interpolator	setInterpolator(Interpolator)	设定插值器（指定的动画效果，譬如回弹等）
android:repeatCount	setRepeatCount(int)	重复次数
android:repeatMode	setRepeatMode(int)	重复类型有两个值，reverse表示倒序回放，restart表示从头播放
android:startOffset	setStartOffset(long)	调用start函数之后等待开始运行的时间，单位为毫秒
android:zAdjustment	setZAdjustment(int)	表示被设置动画的内容运行时在Z轴上的位置（top/bottom/normal），默认为normal

也就是说，无论我们补间动画的哪一种都已经具备了这种属性，也都可以设置使用这些属性中的一个或多个。

那接下来我们就看看每种补间动画特有的一些属性说明吧。

## 2-2. Alpha 属性详解

xml 属性	java方法	解释
android:fromAlpha	AlphaAnimation(float fromAlpha, ...)	动画开始的透明度 (0.0到1.0, 0.0是全透明, 1.0是不透明)
android:toAlpha	AlphaAnimation(..., float toAlpha)	动画结束的透明度, 同上

## 2-3. Rotate 属性详解

xml 属性	java方法	解释
android:fromDegrees	RotateAnimation(float fromDegrees, ...)	旋转开始角度, 正代表顺时针度数, 负代表逆时针度数
android:toDegrees	RotateAnimation(..., float toDegrees, ...)	旋转结束角度, 正代表顺时针度数, 负代表逆时针度数
android:pivotX	RotateAnimation(..., float pivotX, ...)	缩放起点X坐标 (数值、百分数、百分数 p, 譬如50表示以当前View左上角坐标加 50px为初始点、50%表示以当前View的左上角加上当前View宽高的50%做为初始点、50%p表示以当前View的左上角加上父控件宽高的50%做为初始点)
android:pivotY	RotateAnimation(..., float pivotY)	缩放起点Y坐标, 同上规律

## 2-4. Scale 属性详解

xml 属性	java方法	解释
android:fromXScale	ScaleAnimation(float fromX, ...)	初始X轴缩放比例，1.0表示无变化
android:toXScale	ScaleAnimation(..., float toX, ...)	结束X轴缩放比例
android:fromYScale	ScaleAnimation(..., float fromY, ...)	初始Y轴缩放比例
android:toYScale	ScaleAnimation(..., float toY, ...)	结束Y轴缩放比例
android:pivotX	ScaleAnimation(..., float pivotX, ...)	缩放起点X轴坐标（数值、百分数、百分数 p，譬如50表示以当前View左上角坐标加 50px为初始点、50%表示以当前View的左上角加上当前View宽高的50%做为初始点、50%p表示以当前View的左上角加上父控件宽高的50%做为初始点）
android:pivotY	ScaleAnimation(..., float pivotY)	缩放起点Y轴坐标，同上规律

## 2-5. Translate 属性详解

xml 属性	java方法	解释
android:fromXDelta	TranslateAnimation(float fromXDelta, ...)	起始点X轴坐标（数值、百分数、百分数 p，譬如50表示以当前View左上角坐标加 50px为初始点、50%表示以当前View的左上角加上当前View宽高的50%做为初始点、50%p表示以当前View的左上角加上父控件宽高的50%做为初始点）
android:fromYDelta	TranslateAnimation(..., float fromYDelta, ...)	起始点Y轴坐标，同上规律
android:toXDelta	TranslateAnimation(..., float toXDelta, ...)	结束点X轴坐标，同上规律
android:toYDelta	TranslateAnimation(..., float toYDelta)	结束点Y轴坐标，同上规律

## 2-6. AnimationSet 详解

AnimationSet 继承自 Animation，是上面四种的组合容器管理类，没有自己特有的属性，他的属性继承自 Animation，所以特别注意，当我们对 set 标签使用 Animation 的属性时会对该标签下的所有子控件都产生影响。

## 3. 视图动画使用方法

通过上面对于动画的属性介绍之后我们来看看在 Android 中这些动画如何使用（PS：这里直接演示 xml 方式，至于 java 方式太简单了就不说了），如下：

```

<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@[package:]anim/interpolator_resource"
    android:shareInterpolator=["true" | "false"] >
    <alpha
        android:fromAlpha="float"
        android:toAlpha="float" />
    <scale
        android:fromXScale="float"
        android:toXScale="float"
        android:fromYScale="float"
        android:toYScale="float"
        android:pivotX="float"
        android:pivotY="float" />
    <translate
        android:fromXDelta="float"
        android:toXDelta="float"
        android:fromYDelta="float"
        android:toYDelta="float" />
    <rotate
        android:fromDegrees="float"
        android:toDegrees="float"
        android:pivotX="float"
        android:pivotY="float" />
    <set>
        ...
    </set>
</set>

```

使用：

```

ImageView spaceshipImage = (ImageView) findViewById(R.id.spaceshipImage);
Animation hyperspaceJumpAnimation = AnimationUtils.loadAnimation(this,
R.anim.hyperspace_jump);
spaceshipImage.startAnimation(hyperspaceJumpAnimation);

```

上面就是一个标准的使用我们定义的补间动画的模板。至于补间动画的使用，Animation 还有如下一些比较实用的方法介绍：

Animation 类的方法	解释
reset()	重置 Animation 的初始化
cancel()	取消 Animation 动画
start()	开始 Animation 动画
setAnimationListener(AnimationListener listener)	给当前 Animation 设置动画监听
hasStarted()	判断当前 Animation 是否开始
hasEnded()	判断当前 Animation 是否结束

既然补间动画只能给 View 使用，那就来看看 View 中和动画相关的几个常用方法吧，如下：

View类的常用动画操作方法	解释
startAnimation(Animation animation)	对当前 View 开始设置的 Animation 动画
clearAnimation()	取消当 View 在执行的 Animation 动画

到此整个 Android 的补间动画常用详细属性及方法全部介绍完毕，如有特殊的属性需求可以访问 Android Developer 查阅即可。如下我们就来个综合大演练。

## 4. 视图动画注意事项

关于视图动画（补间动画）的例子我就不介绍了，网上简直多的都泛滥了。只是强调在使用补间动画时注意如下一点即可：

**特别特别注意：**补间动画执行之后并未改变 View 的真实布局属性值。切记这一点，譬如我们在 Activity 中有一个 Button 在屏幕上方，我们设置了平移动画移动到屏幕下方然后保持动画最后执行状态呆在屏幕下方，这时如果点击屏幕下方动画执行之后的 Button 是没有任何反应的，而点击原来屏幕上方没有 Button 的地方却响应的是点击 Button 的事件。

## 5. 视图动画 Interpolator 插值器详解

```

AccelerateDecelerateInterpolator (android.view.animation)
AccelerateInterpolator (android.view.animation)
AnticipateInterpolator (android.view.animation)
AnticipateOvershootInterpolator (android.view.animation)
BaseInterpolator (android.view.animation)
BounceInterpolator (android.view.animation)
CycleInterpolator (android.view.animation)
DecelerateInterpolator (android.view.animation)
FastOutLinearInInterpolator (android.support.v4.view.animation)
FastOutSlowInInterpolator (android.support.v4.view.animation)
LinearInterpolator (android.view.animation)
LinearOutSlowInInterpolator (android.support.v4.view.animation)
LookupTableInterpolator (android.support.v4.view.animation)

```

可以看见其实各种插值器都是实现了 Interpolator 接口而已，同时可以看见系统提供了许多已经实现 OK 的插值器，具体如下：

java 类	xml id值	描述
AccelerateDecelerateInterpolator	@android:anim/accelerate_decelerate_interpolator	动画始末速率较慢，中间加速
AccelerateInterpolator	@android:anim/accelerate_interpolator	动画开始速率较慢，之后慢慢加速
AnticipateInterpolator	@android:anim/anticipate_interpolator	开始的时候从后向前甩
AnticipateOvershootInterpolator	@android:anim/anticipate_overshoot_interpolator	类似上面 AnticipateInterpolator
BounceInterpolator	@android:anim/bounce_interpolator	动画结束时弹起
CycleInterpolator	@android:anim/cycle_interpolator	循环播放速率改变为正弦曲线
DecelerateInterpolator	@android:anim/decelerate_interpolator	动画开始快然后慢
LinearInterpolator	@android:anim/linear_interpolator	动画匀速改变
OvershootInterpolator	@android:anim/overshoot_interpolator	向前弹出一定值之后回到原来位置
PathInterpolator		新增，定义路径坐标后按照路径坐标来跑。

如上就是系统提供的一些插值器，下面我们来看看怎么使用他们。

## 5-2 插值器使用方法

插值器的使用比较简答，如下：

```
<set android:interpolator="@android:anim/accelerate_interpolator">
    ...
</set>
```

## 5-3 插值器的自定义

有时候你会发现系统提供的插值器不够用，可能就像 View 一样需要自定义。所以接下来我们来看看插值器的自定义，关于插值器的自定义分为两种实现方式，xml 自定义实现（其实就是对现有的插值器的一些属性修改）或者 java 代码实现方式。如下我们来说。

先看看 XML 自定义插值器的步骤：

1. 在 res/anim/ 目录下创建 filename.xml 文件。
2. 修改你准备自定义的插值器如下：

```
<?xml version="1.0" encoding="utf-8"?>
<InterpolatorName
xmlns:android="http://schemas.android.com/apk/res/android"
    android:attribute_name="value"
/>
```

3. 在你的补间动画文件中引用该文件即可。

可以看见上面第二步修改的是现有插值器的一些属性，但是有些插值器却不具备修改属性，具体如下：

无可自定义的 attribute。

android:factor 浮点值，加速速率（默认值为1）。

android:tension 浮点值，起始点后拉的张力数（默认值为2）。

android:tension 浮点值，起始点后拉的张力数（默认值为2）。 android:extraTension 浮点值，拉力的倍数（默认值为1.5）。

无可自定义的 attribute。

android:cycles 整形，循环的个数（默认为1）。

android:factor 浮点值，减速的速率（默认为1）。

无可自定义的 attribute。

android:tension 浮点值，超出终点后的张力（默认为2）。

再看看 Java 自定义插值器的（Java 自定义插值器其实是 xml 自定义的升级，也就是说如果我们修改 xml 的属性还不能满足需求，那就可以选择通过 Java 来实现）方式。

可以看见上面所有的 Interpolator 都实现了 Interpolator 接口，而 Interpolator 接口又继承自 TimeInterpolator，TimeInterpolator 接口定义了一个 float getInterpolation(float input); 方法，这个方法是由系统调用的，其中的参数 input 代表动画的时间，在 0 和 1 之间，也就是开始和结束之间。

如下就是一个动画始末速率较慢、中间加速的 AccelerateDecelerateInterpolator 插值器：

```
public class AccelerateDecelerateInterpolator extends BaseInterpolator
    implements NativeInterpolatorFactory {
    .....
    public float getInterpolation(float input) {
        return (float)(Math.cos((input + 1) * Math.PI) / 2.0f) + 0.5f;
    }
    .....
}
```

到此整个补间动画与补间动画的插值器都分析完毕了。

## Drawable Animation (Drawable动画)

### 1. Drawable 动画概述

Drawable 动画其实就是 Frame 动画（帧动画），它允许你实现像播放幻灯片一样的效果，这种动画的实质其实是 Drawable，所以这种动画的 XML 定义方式文件一般放在 `res/drawable/` 目录下。具体关于帧动画的 xml 使用方式翻墙点击我查看，java 方式翻墙点击我查看。

如下图就是帧动画的源码文件：

可以看见实际的真实父类就是 Drawable。

### 2. Drawable 动画详细说明

我们依旧可以使用 xml 或者 java 方式实现帧动画。但是依旧推荐使用 xml，具体如下：

`<animation-list>` 必须是根节点，包含一个或者多个 `<item>` 元素，属性有：

- `android:oneshot` true 代表只执行一次，false 循环执行。
- `<item>` 类似一帧的动画资源。

`<item>` animation-list 的子项，包含属性如下：

- `android:drawable` 一个 frame 的 Drawable 资源。
- `android:duration` 一个 frame 显示多长时间。

### 3. Drawable 动画实例演示

关于帧动画相对来说比较简单，这里给出一个常规使用框架，如下：



```
<!-- 注意: rocket.xml文件位于res/drawable/目录下 -->
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot=["true" | "false"] >
    <item
        android:drawable="@[package:]drawable/drawable_resource_name"
        android:duration="integer" />
</animation-list>
```

使用:

```
ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
rocketImage.setBackgroundResource(R.drawable.rocket_thrust);

rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
rocketAnimation.start();
```

特别注意, AnimationDrawable 的 start() 方法不能在 Activity 的 onCreate 方法中调运, 因为 AnimationDrawable 还未完全附着到 window 上, 所以最好的调运时机是 onWindowFocusChanged() 方法中。

## Property Animation (属性动画) 使用详解

在使用属性动画之前先来看几个常用的 View 属性成员:

- translationX, translationY: 控制View的位置, 值是相对于View容器左上角坐标的偏移。
- rotationX, rotationY: 控制相对于轴心旋转。
- x, y: 控制View在容器中的位置, 即左上角坐标加上translationX和translationY的值。
- alpha: 控制View对象的alpha透明度值。

这几个常用的属性相信大家都很熟悉, 接下来的属性动画我们就从这里展开。

### 1. 属性动画概述

Android 3.0 以后引入了属性动画, 属性动画可以轻而易举的实现许多 View 动画做不到的事, 上面也看见了, View动画无非也就做那几种事情, 别的也搞不定, 而属性动画就可以的, 譬如3D旋转一张图片。其实说白了, 你记住一点就行, 属性动画实现原理就是修改控件的属性值实现的动画。

具体先看下类关系:

```
/**
 * This is the superclass for classes which provide basic support for animations
 * which can be
 * started, ended, and have <code>AnimatorListeners</code> added to them.
 */
public abstract class Animator implements Cloneable {
    .....
}
```

所有的属性动画的抽象基类就是他。我们看下他的实现子类:

```
AnimatorSet (android.animation) < An
ObjectAnimator (android.animation) < An
TimeAnimator (android.animation) < An
ValueAnimator (android.animation) < An
```

java 类名	xml 关键字	描述信息
ValueAnimator	<code>&lt;animator&gt;</code> 放置在 res/animator/ 目录下	在一个特定的时间里执行一个动画
TimeAnimator	不支持/点我查看原因	时序监听回调工具
ObjectAnimator	<code>&lt;objectAnimator&gt;</code> 放置在 res/animator/ 目录下	一个对象的一个属性动画
AnimatorSet	<code>&lt;set&gt;</code> 放置在 res/animator/ 目录下	动画集合

所以可以看见，我们平时使用属性动画的重点就在于 AnimatorSet、ObjectAnimator、TimeAnimator、ValueAnimator。所以接下来我们就来依次说说如何使用。

## 2. 属性动画详细说明

### 2-1 属性动画计算原理

Android 属性动画（注意最低兼容版本，不过可以使用开源项目来替代低版本问题）提供了以下属性：

- Duration：动画的持续时间；
- TimeInterpolation：定义动画变化速率的接口，所有插值器都必须实现此接口，如线性、非线性插值器；
- TypeEvaluator：用于定义属性值计算方式的接口，有 int、float、color 类型，根据属性的起始、结束值和插值一起计算出当前时间的属性值；
- Animation sets：动画集合，即可以同时对一个对象应用多个动画，这些动画可以同时播放也可以对不同动画设置不同的延迟；
- Frame refresh delay：多少时间刷新一次，即每隔多少时间计算一次属性值，默认为 10ms，最终刷新时间还受系统进程调度与硬件的影响；
- Repeat Count and behavior：重复次数与方式，如播放3次、5次、无限循环，可以让此动画一直重复，或播放完时向反向播放；

接下来先来看官方为了解释原理给出的两幅图（其实就是初中物理题，不解释）：

[![Example of a linear animation]

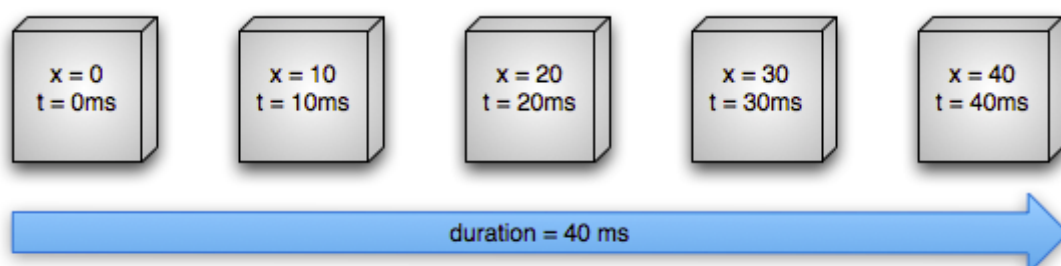


Figure 1. Example of a linear animation

上面就是一个线性匀速动画，描述了一个 Object 的 X 属性运动动画，该对象的X坐标在 40ms 内从 0 移动到 40，每 10ms 刷新一次，移动 4 次，每次移动为  $40/4=10\text{pixel}$ 。

[![Example of a non-linear animation]

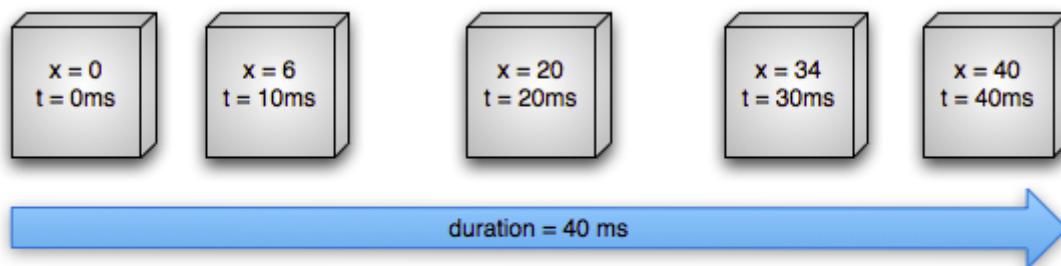


Figure 2. Example of a non-linear animation

上面是一个非匀速动画，描述了一个 Object 的 X 属性运动动画，该对象的 X 坐标在 40ms 内从 0 移动到 40，每 10ms 刷新一次，移动4次，但是速率不同，开始和结束的速度要比中间部分慢，即先加速后减速。

接下来我们来详细的看一下，属性动画系统的重要组成部分是如何计算动画值的，下图描述了如上面所示动画的实现作用过程。

[![How animation are calculated]

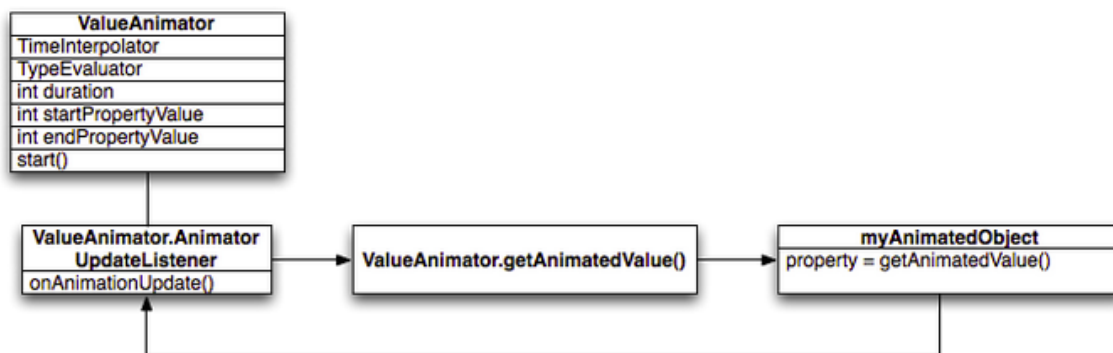


Figure 3. How animations are calculated

其中的 ValueAnimator 是动画的执行类，跟踪了当前动画的执行时间和当前时间下的属性值；ValueAnimator 封装了动画的 TimeInterpolator 时间插值器和一个 TypeEvaluator 类型估值，用于设置动画属性的值，就像上面图2非线性动画里，TimeInterpolator 使用了 AccelerateDecelerateInterpolator、TypeEvaluator 使用了 IntEvaluator。

为了执行一个动画，你需要创建一个 ValueAnimator，并且指定目标对象属性的开始、结束值和持续时间。在调用 start 后，整个动画过程中，ValueAnimator 会根据已经完成的动画时间计算得到一个 0 到 1 之间的分数，代表该动画的已完成动画百分比。0 表示 0%，1 表示 100%，譬如上面图一线性匀速动画中总时间  $t = 40\text{ ms}$ ， $t = 10\text{ ms}$  的时候是 0.25。

当 ValueAnimator 计算完已完成动画分数后，它会调用当前设置的 TimeInterpolator，去计算得到一个 interpolated（插值）分数，在计算过程中，已完成动画百分比会被加入到新的插值计算中。如上图 2 非线性动画中，因为动画的运动是缓慢加速的，它的插值分数大约是 0.15，小于  $t = 10\text{ms}$  时的已完成动画分数 0.25。而在上图1中，这个插值分数一直和已完成动画分数是相同的。

当插值分数计算完成后，ValueAnimator 会根据插值分数调用合适的 TypeEvaluator 去计算运动中的属性值。

好了，现在我们来看下代码就明白这段话了，上面图2非线性动画里，TimeInterpolator 使用了 AccelerateDecelerateInterpolator、TypeEvaluator 使用了 IntEvaluator。所以这些类都是标准的 API，我们来看下标准API就能类比自己写了，如下：

首先计算已完成动画时间分数（以 10ms 为例）： $t=10\text{ms}/40\text{ms}=0.25$ 。

接着看如下源码如何实现计算差值分数的：

```
public class AccelerateDecelerateInterpolator extends BaseInterpolator
    implements NativeInterpolatorFactory {
    public AccelerateDecelerateInterpolator() {
    }
    .....
    //这是我们关注重点，可以发现如下计算公式计算后（input即为时间因子）插值大约为0.15。
    public float getInterpolation(float input) {
        return (float)(Math.cos((input + 1) * Math.PI) / 2.0f) + 0.5f;
    }
    .....
}
```

其实 AccelerateDecelerateInterpolator 的基类接口就是 TimeInterpolator，如下，他只有 getInterpolation 方法，也就是上面我们关注的方法。

```
public interface TimeInterpolator {
    float getInterpolation(float input);
}
```

接着 ValueAnimator 会根据插值分数调用合适的 TypeEvaluator (IntEvaluator) 去计算运动中的属性值，如下，因为 startValue = 0，所以属性值：0+0.15\* (40-0) = 6。

```
public class IntEvaluator implements TypeEvaluator<Integer> {
    public Integer evaluate(float fraction, Integer startValue, Integer
endValue) {
        int startInt = startValue;
        return (int)(startInt + fraction * (endValue - startInt));
    }
}
```

这就是官方给的一个关于属性动画实现的过程及基本原理解释，相信你看到这里是会有些迷糊的，没关系，你先有个大致概念就行，接下来我们会慢慢进入实战，因为 Android 的属性动画相对于其他动画来说涉及的知识点本来就比较复杂，所以我们慢慢来。

## 2-2 XML 方式属性动画

在 xml 中(放在 res/animator/filename.xml )可直接用的属性动画节点有 ValueAnimator、ObjectAnimator、AnimatorSet。

```
<set
    android:ordering=["together" | "sequentially"]>

    <objectAnimator
        android:propertyName="string"
        android:duration="int"
        android:valueFrom="float | int | color"
        android:valueTo="float | int | color"
        android:startOffset="int"
        android:repeatCount="int"
        android:repeatMode=["repeat" | "reverse"]
        android:valueType=["intType" | "floatType"]/>

    <animator
```

```

        android:duration="int"
        android:valueFrom="float | int | color"
        android:valueTo="float | int | color"
        android:startOffset="int"
        android:repeatCount="int"
        android:repeatMode=["repeat" | "reverse"]
        android:valueType=["intType" | "floatType"]/>

<set>
    ...
</set>
</set>

```

<set> 属性解释:

xml 属性	解释
android:ordering	控制子动画启动方式是先后有序的还是同时进行。sequentially: 动画按照先后顺序；together(默认): 动画同时启动；

<objectAnimator> 属性解释:

xml属性	解释
android:propertyName	String 类型，必须要设置的节点属性，代表要执行动画的属性（通过名字引用），譬如你可以指定了一个View的“alpha”或者“backgroundColor”，这个objectAnimator元素没有对外说明target属性，所以你不能在XML中设置执行这个动画，必须通过调用loadAnimator()方法加载你的XML动画资源，然后调用setTarget()应用到具备这个属性的目标对象上（譬如TextView）。
android:valueTo	float、int 或者 color 类型，必须要设置的节点属性，表明动画结束的点；如果是颜色的话，由 6 位十六进制的数字表示。
android:valueFrom	相对应 valueTo，动画的起始点，如果没有指定，系统会通过属性的get方法获取，颜色也是 6 位十六进制的数字表示。
android:duration	动画的时长，int 类型，以毫秒为单位，默认为 300 毫秒。
android:startOffset	动画延迟的时间，从调用 start 方法后开始计算，int 型，毫秒为单位。
android:repeatCount	一个动画的重复次数，int 型，“-1”表示无限循环，“1”表示动画在第一次执行完成后重复执行一次，也就是两次，默认为0，不重复执行。
android:repeatMode	重复模式：int 型，当一个动画执行完的时候应该如何处理。该值必须是正数或者是-1，“reverse”会使得按照动画向相反的方向执行，可实现类似钟摆效果。“repeat”会使得动画每次都从头开始循环。
android:valueType	关键参数，如果该 value 是一个颜色，那么就不需要指定，因为动画框架会自动的处理颜色值。有 intType 和 floatType（默认）两种：分别说明动画值为 int 和 float 型。

<objectAnimator> 属性解释: 同上 <objectAnimator> 属性，不多介绍。

XML 属性动画使用方法：

```
AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(myContext,
    R.animator.property_animator);
set.setTarget(myObject);
set.start();
```

## 2-3 Java 方式属性动画

**1、ObjectAnimator：** 继承自 ValueAnimator，允许你指定要进行动画的对象以及该对象的一个属性。该类会根据计算得到的新值自动更新属性。大多数的情况使用 ObjectAnimator 就足够了，因为它使得目标对象动画值的处理过程变得足够简单，不用像 ValueAnimator 那样自己写动画更新的逻辑，但是 ObjectAnimator 有一定的限制，比如它需要目标对象的属性提供指定的处理方法（譬如提供 getXXX，setXXX 方法），这时候你就需要根据自己的需求在 ObjectAnimator 和 ValueAnimator 中看哪种实现更方便了。

ObjectAnimator 类提供了 ofInt、ofFloat、ofObject 这三个常用的方法，这些方法都是设置动画作用的元素、属性、开始、结束等任意属性值。当属性值（上面方法的参数）只设置一个时就把通过 getXXX 反射获取的值作为起点，设置的值作为终点；如果设置两个（参数），那么一个是开始、另一个是结束。

**特别注意：** ObjectAnimator 的动画原理是不停的调用 setXXX 方法更新属性值，所有使用 ObjectAnimator 更新属性时的前提是 Object 必须声明有 getXXX 和 setXXX 方法。我们通常使用 ObjectAnimator 设置 View 已知的属性来生成动画，而一般 View 已知属性变化时都会主动触发重绘图操作，所以动画会自动实现；但是也有特殊情况，譬如作用 Object 不是 View，或者作用的属性没有触发重绘，或者我们在重绘时需要做自己的操作，那都可以通过如下方法手动设置：

```
ObjectAnimator mObjectAnimator= ObjectAnimator.ofInt(view,
"customerDefineAnythingName", 0, 1).setDuration(2000);
mObjectAnimator.addUpdateListener(new AnimatorUpdateListener()
{
    @Override
    public void onAnimationUpdate(ValueAnimator animation)
    {
        //int value = animation.getAnimatedValue();  可以获取当前属性值
        //view.postInvalidate();  可以主动刷新
        //view.setXXX(value);
        //view.setXXX(value);
        //.....可以批量修改属性
    }
});
```

如下是一个我在项目中的Y轴3D旋转动画实现实例：

```
ObjectAnimator.ofFloat(view, "rotationY", 0.0f,
360.0f).setDuration(1000).start();
```

**2、PropertyValuesHolder：** 多属性动画同时工作管理类。有时候我们需要同时修改多个属性，那就用到此类，具体如下：

```
PropertyValuesHolder a1 = PropertyValuesHolder.ofFloat("alpha", 0f, 1f);
PropertyValuesHolder a2 = PropertyValuesHolder.ofFloat("translationY", 0,
viewwidth);
.....
ObjectAnimator.ofPropertyValuesHolder(view, a1, a2,
.....).setDuration(1000).start();
```

如上代码就可以实现同时修改多个属性的动画啦。

**3、ValueAnimator：**属性动画中的时间驱动，管理着动画时间的开始、结束属性值，相应时间属性值计算方法等。包含所有计算动画值的核心函数以及每一个动画时间节点上的信息、一个动画是否重复、是否监听更新事件等，并且还可以设置自定义的计算类型。

特别注意：ValueAnimator 只是动画计算管理驱动，设置了作用目标，但没有设置属性，需要通过 updateListener 里设置属性才会生效。

```
ValueAnimator animator = ValueAnimator.ofFloat(0, mContentHeight); //定义动画
animator.setTarget(view); //设置作用目标
animator.setDuration(5000).start();
animator.addUpdateListener(new AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator animation){
        float value = (float) animation.getAnimatedValue();
        view.setXXX(value); //必须通过这里设置属性值才有效
        view.mXXX = value; //不需要setXXX属性方法
    }
});
```

大眼看上去可以发现和 ObjectAnimator 没啥区别，实际上正是由于 ValueAnimator 不直接操作属性值，所以要操作对象的属性可以不需要 setXXX 与 getXXX 方法，你完全可以通过当前动画的计算去修改任何属性。

**4、AnimationSet：**动画集合，提供把多个动画组合成一个组合的机制，并可设置动画的时序关系，如同时播放、顺序播放或延迟播放。具体使用方法比较简单，如下：

```
ObjectAnimator a1 = ObjectAnimator.ofFloat(view, "alpha", 1.0f, 0f);
ObjectAnimator a2 = ObjectAnimator.ofFloat(view, "translationY", 0f, viewwidth);

.....
AnimatorSet animSet = new AnimatorSet();
animSet.setDuration(5000);
animSet.setInterpolator(new LinearInterpolator());
//animSet.playTogether(a1, a2, ...); //两个动画同时执行
animSet.play(a1).after(a2); //先后执行
.....//其他组合方式
animSet.start();
```

**5、Evaluators 相关类解释：**Evaluators 就是属性动画系统如何去计算一个属性值。它们通过 Animator 提供的动画的起始和结束值去计算一个动画的属性值。

- IntEvaluator：整数属性值。
- FloatEvaluator：浮点数属性值。
- ArgbEvaluator：十六进制color属性值。
- TypeEvaluator：用户自定义属性值接口，譬如对象属性值类型不是 int、float、color 类型，你必须实现这个接口去定义自己的数据类型。



既然说到这了，那就来个例子吧，譬如我们需要实现一个自定义属性类型和计算规则的属性动画，如下类型 float[]:

```
ValueAnimator valueAnimator = new ValueAnimator();
valueAnimator.setDuration(5000);
valueAnimator.setObjectValues(new float[2]); //设置属性值类型
valueAnimator.setInterpolator(new LinearInterpolator());
valueAnimator.setEvaluator(new TypeEvaluator<float[]>()
{
    @Override
    public float[] evaluate(float fraction, float[] startValue,
        float[] endValue)
    {
        //实现自定义规则计算的float[]类型的属性值
        float[] temp = new float[2];
        temp[0] = fraction * 2;
        temp[1] = (float)Math.random() * 10 * fraction;
        return temp;
    }
});

valueAnimator.start();
valueAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener()
{
    @Override
    public void onAnimationUpdate(ValueAnimator animation)
    {
        float[] xyPos = (float[]) animation.getAnimatedValue();
        view.setHeight(xyPos[0]); //通过属性值设置View属性动画
        view.setWidth(xyPos[1]); //通过属性值设置View属性动画
    }
});
```

## 6、Interpolators 相关类解释:

- AccelerateDecelerateInterpolator: 先加速后减速。
- AccelerateInterpolator: 加速。
- DecelerateInterpolator: 减速。
- AnticipateInterpolator: 先向相反方向改变一段再加速播放。
- AnticipateOvershootInterpolator: 先向相反方向改变，再加速播放，会超出目标值然后缓慢移动至目标值，类似于弹簧回弹。
- BounceInterpolator: 快到目标值时值会跳跃。
- CycleInterpolator: 动画循环一定次数，值的改变为一正弦函数: `Math.sin(2 * mCycles * Math.PI * input)`。
- LinearInterpolator: 线性均匀改变。
- OvershootInterpolator: 最后超出目标值然后缓慢改变到目标值。
- TimeInterpolator: 一个允许自定义 Interpolator 的接口，以上都实现了该接口。

举个例子，就像系统提供的标准 API 一样，如下就是加速插值器的实现代码，我们自定义时也可以类似实现:

```
//开始很慢然后不断加速的插值器。
public class AccelerateInterpolator implements Interpolator {
    private final float mFactor;
    private final double mDoubleFactor;
```



```

public AccelerateInterpolator() {
    mFactor = 1.0f;
    mDoubleFactor = 2.0;
}

.....

//input 0到1.0。表示动画当前点的值，0表示开头，1表示结尾。
//return 插值。值可以大于1超出目标值，也可以小于0突破低值。
@Override
public float getInterpolation(float input) {
    //实现核心代码块
    if (mFactor == 1.0f) {
        return input * input;
    } else {
        return (float)Math.pow(input, mDoubleFactor);
    }
}
}

```

综上所述可以发现，我们可以使用现有系统提供标准的东东实现属性动画，也可以通过自定义继承相关接口实现自己的动画，只要实现上面提到的那些主要方法即可。

## 2-4 Java 属性动画拓展之 ViewPropertyAnimator 动画

在 Android API 12 时，View 中添加了 animate 方法，具体如下：

```

public class View implements Drawable.Callback, KeyEvent.Callback,
    AccessibilityEventSource {
    .....
    /**
     * This method returns a ViewPropertyAnimator object, which can be used to
    animate
     * specific properties on this View.
     *
     * @return ViewPropertyAnimator The ViewPropertyAnimator associated with
    this View.
     */
    public ViewPropertyAnimator animate() {
        if (mAnimator == null) {
            mAnimator = new ViewPropertyAnimator(this);
        }
        return mAnimator;
    }
    .....
}

```

可以看见通过 View 的 animate() 方法可以得到一个 ViewPropertyAnimator 的属性动画（有人说他没有继承 Animator 类，是的，他是成员关系，不是之前那种继承关系）。

ViewPropertyAnimator 提供了一种非常方便的方法为 View 的部分属性设置动画（切记，是部分属性），它可以直接使用一个 Animator 对象设置多个属性的动画；在多属性设置动画时，它比上面的 ObjectAnimator 更加牛逼、高效，因为他会管理多个属性的 invalidate 方法统一调运触发，而不像上面分别调用，所以还会有一些性能优化。如下就是一个例子：

```
myView.animate().x(0f).y(100f).start();
```

## 2-5 Java 属性动画拓展之 LayoutAnimator 容器布局动画

Property 动画系统还提供了对 ViewGroup 中 View 添加时的动画功能，我们可以用 LayoutTransition 对 ViewGroup 中的 View 进行动画设置显示。LayoutTransition 的动画效果都是设置给 ViewGroup，然后当被设置动画的 ViewGroup 中添加删除 View 时体现出来。该类用于当前布局容器中有 View 添加、删除、隐藏、显示等时候定义布局容器自身的动画和 View 的动画，也就是说当在一个 LinerLayout 中隐藏一个 View 的时候，我们可以自定义 整个由于 LinerLayout 隐藏 View 而改变的动画， 同时还可以自定义被隐藏的 View 自己消失时候的动画等。

我们可以发现 LayoutTransition 类中主要有五种容器转换动画类型，具体如下：

- LayoutTransition.APPEARING：当 View 出现或者添加的时候 View 出现的动画。
- LayoutTransition.CHANGE\_APPEARING：当添加 View 导致布局容器改变的时候整个布局容器的动画。
- LayoutTransition.DISAPPEARING：当View消失或者隐藏的时候 View 消失的动画。
- LayoutTransition.CHANGE\_DISAPPEARING：当删除或者隐藏 View 导致布局容器改变的时候整个布局容器的动画。
- LayoutTransition.CHANGE：当不是由于 View 出现或消失造成对其他 View 位置造成改变的时候整个布局容器的动画。

### XML 方式使用系统提供的默认 LayoutTransition 动画：

我们可以通过如下方式使用系统提供的默认 ViewGroup 的 LayoutTransition 动画：

```
android:animateLayoutChanges="true"
```

在 ViewGroup 添加如上 xml 属性默认是没有任何动画效果的，因为前面说了，该动画针对于 ViewGroup 内部东东发生改变时才有效， 所以当我们设置如上属性然后调运 ViewGroup 的 addView、removeView 方法时就能看见系统默认的动画效果了。

还有一种就是通过如下方式设置：

```
android:layoutAnimation="@anim/customer_anim"
```

通过这种方式就能实现很多吊炸天的动画。

Java 方式使用系统提供的默认 LayoutTransition 动画：

在使用 LayoutTransition 时，你可以自定义这几种事件类型的动画，也可以使用默认的动画， 总之最终都是通过 setLayoutTransition(LayoutTransition lt) 方法把这些动画以一个 LayoutTransition 对象设置给一个 ViewGroup。

譬如实现如上 Xml 方式的默认系统 LayoutTransition 动画如下：

```
mTransitioner = new LayoutTransition();
mViewGroup.setLayoutTransition(mTransitioner);
```

稍微再高端一点吧，我们来自定义这几类事件的动画，分别实现他们，那么你可以像下面这么处理：

```
mTransitioner = new LayoutTransition();
.....
ObjectAnimator anim = ObjectAnimator.ofFloat(this, "scalex", 0, 1);
.....//设置更多动画
mTransition.setAnimator(LayoutTransition.APPEARING, anim);
.....//设置更多类型的动画
mViewGroup.setLayoutTransition(mTransitioner);
```

到此通过 LayoutTransition 你就能实现类似小米手机计算器切换普通型和科学型的炫酷动画了。

## 3. ViewPropertyAnimator 动画(属性动画拓展)

### 3-1 ViewPropertyAnimator 概述

通过上面的学习，我们应该明白了属性动画的推出已不再是针对于 View 而进行设计的了，而是一种对数值不断操作的过程，我们可以将属性动画对数值的操作过程设置到指定对象的属性上来，从而形成一种动画的效果。虽然属性动画给我们提供了 ValueAnimator 类和 ObjectAnimator 类，在正常情况下，基本都能满足我们对动画操作的需求，但 ValueAnimator 类和 ObjectAnimator 类本身并不是针对 View 对象的而设计的，而我们在大多数情况下主要都还是对 View 进行动画操作的，因此 Google 官方在 Android 3.1 系统中补充了 ViewPropertyAnimator 类，这个类便是专门为 View 动画而设计的。当然这个类不仅仅是为提供 View 而简单设计的，它存在以下优点：

- 专门针对View对象动画而操作的类。
- 提供了更简洁的链式调用设置多个属性动画，这些动画可以同时进行的。
- 拥有更好的性能，多个属性动画是一次同时变化，只执行一次UI刷新（也就是只调用一次 invalidate ,而 n 个 ObjectAnimator 就会 进行 n 次属性变化，就有 n 次 invalidate）。
- 每个属性提供两种类型方法设置。
- 该类只能通过 View 的 animate() 获取其实例对象的引用

好~，下面我们来了解一下 ViewPropertyAnimator 常规使用

### 3-2 ViewPropertyAnimator 常规使用

之前我们要设置一个View控件旋转 360 的代码是这样：

```
ObjectAnimator.ofFloat(btn, "rotation", 360).setDuration(200).start();
```

而现在我们使用 ViewPropertyAnimator 后是这样：

```
btn.animate().rotation(360).setDuration(200);
```

代码是不是特简洁？这里我们来解析一下，首先必须用 View#animate() 方法来获取一个 ViewPropertyAnimator 的对象实例，前面我们说过 ViewPropertyAnimator 支持链式操作，所以这里直接通过 rotation 方法设置旋转角度，再设置时间即可，有没有发现连动画的启动都不用我们去操作！是的，ViewPropertyAnimator 内部会自动去调用。对于 View#animate() 方法，这里再说明一下，animate() 方法是在 Android 3.1 系统上新增的一个方法，其作用就是返回 ViewPropertyAnimator 的实例对象，其源码如下，一目了然：

```
/**
 * This method returns a ViewPropertyAnimator object, which can be used to
 * animate
 * specific properties on this view.
 *
 * @return ViewPropertyAnimator The ViewPropertyAnimator associated with this
 * View.
 */
public ViewPropertyAnimator animate() {
    if (mAnimator == null) {
        mAnimator = new ViewPropertyAnimator(this);
    }
    return mAnimator;
}
```

可以看见通过 View 的 animate() 方法可以得到一个 ViewPropertyAnimator 的属性动画（有人说他没有继承 Animator 类，是的，他是成员关系，不是之前那种继承关系）。

接着我们再来试试别的方法，同时设置一组动画集合如下：

```
AnimatorSet set = new AnimatorSet();
set.playTogether( ObjectAnimator.ofFloat(btn, "alpha", 0.5f),
    ObjectAnimator.ofFloat(btn, "rotation", 360),
    ObjectAnimator.ofFloat(btn, "scaleX", 1.5f),
    ObjectAnimator.ofFloat(btn, "scaleY", 1.5f),
    ObjectAnimator.ofFloat(btn, "translationX", 0, 50),
    ObjectAnimator.ofFloat(btn, "translationY", 0, 50)
);
set.setDuration(5000).start();
```

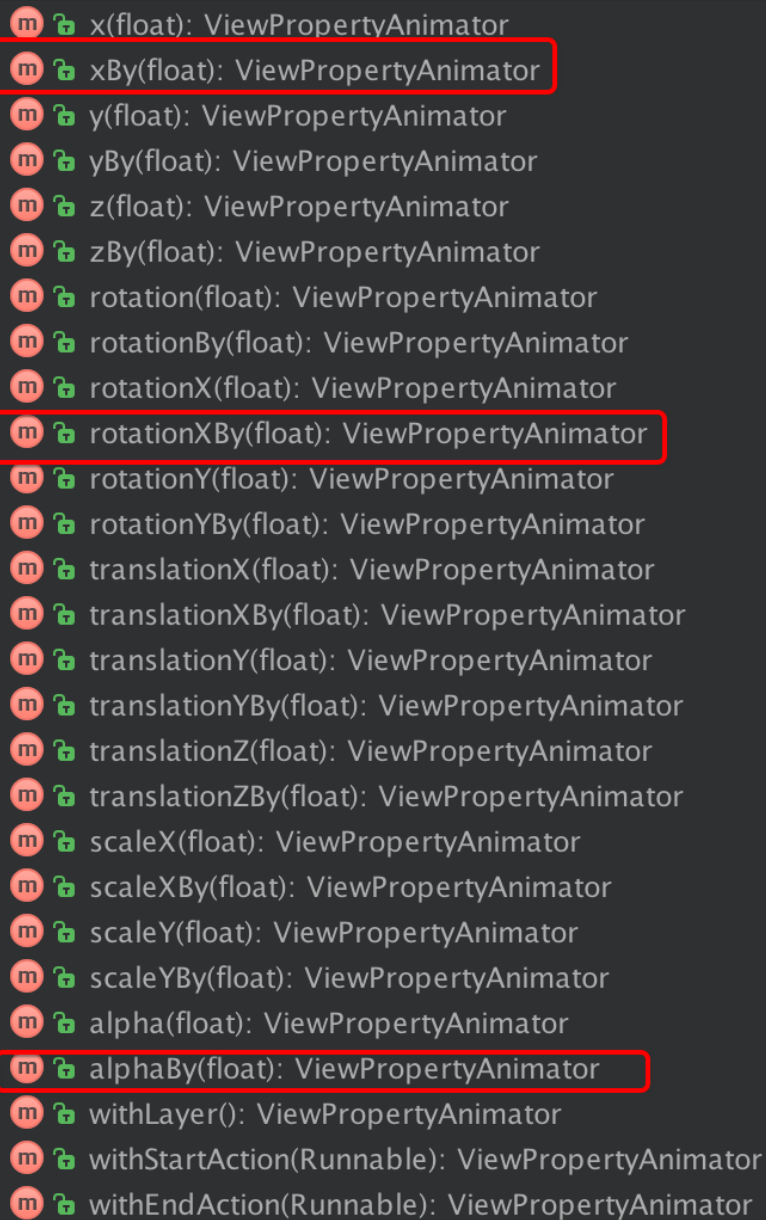
使用 ViewPropertyAnimator 设置代码如下：

```
btn.animate().alpha(0.5f).rotation(360).scaleX(1.5f).scaleY(1.5f)
    .translationX(50).translationY(50).setDuration(5000);
```

是不是已经深深地爱上 ViewPropertyAnimator？真的太简洁了！都快感动地哭出来了.....先去厕所哭会.....好吧，ViewPropertyAnimator 简单用法讲完了，这里小结一下 ViewPropertyAnimator 的常用方法：

Method	Discription
alpha(float value)	设置透明度，value表示变化到多少，1不透明，0全透明。
scaleY(float value)	设置Y轴方向的缩放大小，value表示缩放到多少。1表示正常规格。小于1代表缩小，大于1代表放大。
scaleX(float value)	设置X轴方向的缩放大小，value表示缩放到多少。1表示正常规格。小于1代表缩小，大于1代表放大。
translationY(float value)	设置Y轴方向的移动值，作为增量来控制View对象相对于它父容器的左上角坐标偏移的位置，即移动到哪里。
translationX(float value)	设置X轴方向的移动值，作为增量来控制View对象相对于它父容器的左上角坐标偏移的位置。
rotation(float value)	控制View对象围绕支点进行旋转，rotation针对2D旋转
rotationX (float value)	控制View对象围绕X支点进行旋转，rotationX针对3D旋转
rotationY(float value)	控制View对象围绕Y支点进行旋转，rotationY针对3D旋转
x(float value)	控制View对象相对于它父容器的左上角坐标在X轴方向的最终位置。
y(float value)	控制View对象相对于它父容器的左上角坐标在Y轴方向的最终位置
void cancel()	取消当前正在执行的动画
addListener(Animator.AnimatorListener listener)	设置监听器，监听动画的开始，结束，取消，重复播放
setUpdateListener(ValueAnimator.AnimatorUpdateListener listener)	设置监听器，监听动画的每一帧的播放
setInterpolator(TimeInterpolator interpolator)	设置插值器
setStartDelay(long startDelay)	设置动画延长开始的时间
setDuration(long duration)	设置动画执行的时间
withLayer()	设置是否开启硬件加速
withStartAction(Runnable runnable)	设置用于动画监听开始 (Animator.AnimatorListener) 时运行的Runnable任务对象
withEndAction(Runnable runnable)	设置用于动画监听结束 (Animator.AnimatorListener) 时运行的Runnable任务对象

以上便是 ViewPropertyAnimator 一些操作方法，其实上面很多属性设置方法都对应着一个By结尾的方法，其变量则代表的是变化量，如下：



- m x(float): ViewPropertyAnimator
- m xBy(float): ViewPropertyAnimator
- m y(float): ViewPropertyAnimator
- m yBy(float): ViewPropertyAnimator
- m z(float): ViewPropertyAnimator
- m zBy(float): ViewPropertyAnimator
- m rotation(float): ViewPropertyAnimator
- m rotationBy(float): ViewPropertyAnimator
- m rotationX(float): ViewPropertyAnimator
- m rotationXBy(float): ViewPropertyAnimator
- m rotationY(float): ViewPropertyAnimator
- m rotationYBy(float): ViewPropertyAnimator
- m translationX(float): ViewPropertyAnimator
- m translationXBy(float): ViewPropertyAnimator
- m translationY(float): ViewPropertyAnimator
- m translationYBy(float): ViewPropertyAnimator
- m translationZ(float): ViewPropertyAnimator
- m translationZBy(float): ViewPropertyAnimator
- m scaleX(float): ViewPropertyAnimator
- m scaleXBy(float): ViewPropertyAnimator
- m scaleY(float): ViewPropertyAnimator
- m scaleYBy(float): ViewPropertyAnimator
- m alpha(float): ViewPropertyAnimator
- m alphaBy(float): ViewPropertyAnimator
- m withLayer(): ViewPropertyAnimator
- m withStartAction(Runnable): ViewPropertyAnimator
- m withEndAction(Runnable): ViewPropertyAnimator

我们看看其中 scaleY 与 scaleYBy 的实现：

```
public ViewPropertyAnimator scaleY(float value) {  
    animateProperty(SCALE_Y, value);  
    return this;  
}  
  
public ViewPropertyAnimator scaleYBy(float value) {  
    animatePropertyBy(SCALE_Y, value);  
    return this;  
}
```

再看看 animateProperty() 与 animatePropertyBy()

```

private void animateProperty(int constantName, float toValue) {
    float fromValue = getValue(constantName);
    float deltaValue = toValue - fromValue;
    animatePropertyBy(constantName, fromValue, deltaValue);
}

private void animatePropertyBy(int constantName, float byValue) {
    float fromValue = getValue(constantName);
    animatePropertyBy(constantName, fromValue, byValue);
}

```

看了源码现在应该很清楚有By结尾（代表变化量的大小）和没By结尾（代表变化到多少）的方法的区别了吧。好~，再来看看监听器，实际上我们可以通过 `setListener(Animator.AnimatorListener listener)` 和 `setUpdateListener(ValueAnimator.AnimatorUpdateListener listener)` 设置自定义监听器，而在 `ViewPropertyAnimator` 内部也有自己实现的监听器，同样我们可以看一下其实现源码：

```

private class AnimatorEventListener
    implements Animator.AnimatorListener,
    ValueAnimator.AnimatorUpdateListener {
    @Override
    public void onAnimationStart(Animator animation) {
        //调用了设置硬件加速的Runnable
        if (mAnimatorSetupMap != null) {
            Runnable r = mAnimatorSetupMap.get(animation);
            if (r != null) {
                r.run();
            }
            mAnimatorSetupMap.remove(animation);
        }
        if (mAnimatorOnStartMap != null) {
            //调用我们通过withStartAction(Runnable runnable)方法设置的Runnable
            Runnable r = mAnimatorOnStartMap.get(animation);
            if (r != null) {
                r.run();
            }
            mAnimatorOnStartMap.remove(animation);
        }
        if (mListener != null) {
            //调用我们自定义的监听器方法
            mListener.onAnimationStart(animation);
        }
    }

    @Override
    public void onAnimationCancel(Animator animation) {
        if (mListener != null) {
            //调用我们自定义的监听器方法
            mListener.onAnimationCancel(animation);
        }
        if (mAnimatorOnEndMap != null) {
            mAnimatorOnEndMap.remove(animation);
        }
    }

    @Override

```

```

public void onAnimationRepeat(Animator animation) {
    if (mListener != null) {
        //调用我们自定义的监听器方法
        mListener.onAnimationRepeat(animation);
    }
}

@Override
public void onAnimationEnd(Animator animation) {
    mView.setHasTransientState(false);
    if (mListener != null) {
        //调用我们自定义的监听器方法
        mListener.onAnimationEnd(animation);
    }
    if (mAnimatorOnEndMap != null) {
        //调用我们通过withEndAction(Runnable runnable)方法设置的runnable
        Runnable r = mAnimatorOnEndMap.get(animation);
        if (r != null) {
            r.run();
        }
        mAnimatorOnEndMap.remove(animation);
    }
    if (mAnimatorCleanupMap != null) {
        //移除硬件加速
        Runnable r = mAnimatorCleanupMap.get(animation);
        if (r != null) {
            r.run();
        }
        mAnimatorCleanupMap.remove(animation);
    }
    mAnimatorMap.remove(animation);
}
}

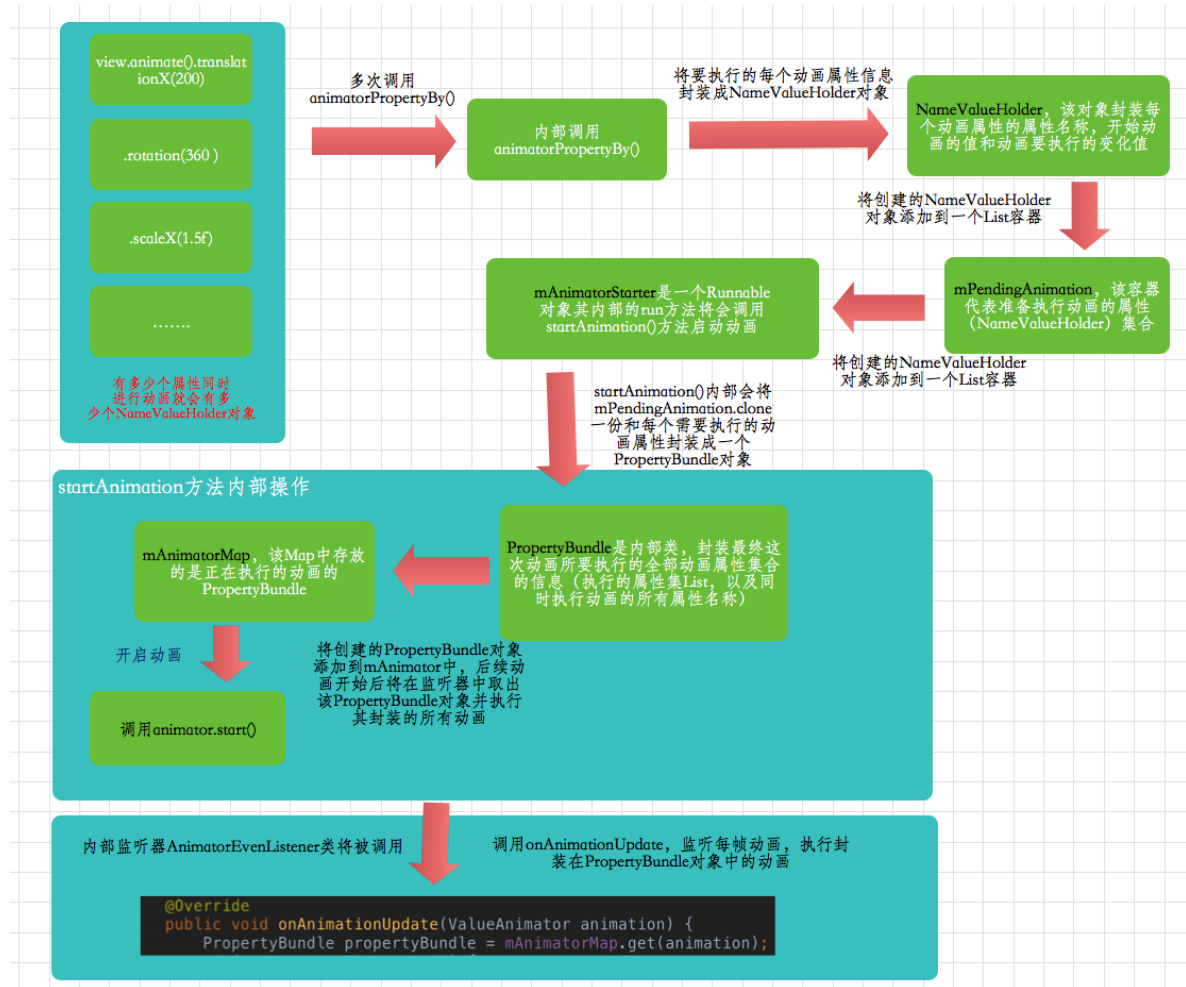
```

由源码我们知道当监听器仅需要监听动画的开始和结束时，我们可以通过 `withStartAction(Runnable runnable)` 和 `withEndAction(Runnable runnable)` 方法来设置一些特殊的监听操作。在 `AnimatorEventListener` 中的开始事件还会判断是否开启硬件加速，当然在动画结束时也会去关闭硬件加速。我们可以通过 `ViewPropertyAnimator #withLayer()` 方法开启硬件加速功能。到此对于 `ViewPropertyAnimator` 的常规使用方式 已很清晰了。

### 3-3 ViewPropertyAnimator 原理解析



ViewPropertyAnimator 原理解析 我们先通过一副图来大概了解一下 ViewPropertyAnimator 内部的整体运行工作原理



## ViewPropertyAnimator

我们这里先给出整体执行流程（有个整体的概念就行哈，不理解也没有关系，看完下面的分析，再回来看看也是可以），然后再详细分析：

通过 imageView.animate() 获取 ViewPropertyAnimator 对象。

调用 alpha、translationX 等方法，返回当前 ViewPropertyAnimator 对象，可以继续链式调用

alpha、translationX 等方法内部最终调用 animatePropertyBy(int constantName, float startValue, float byValue)方法

在 animatePropertyBy 方法中则会将 alpha、translationX 等方法的封装成 NameVauleHolder，并将每个 NameValueHolder 对象添加到准备列表 mPendingAnimations 中。

animatePropertyBy 方法启动 mAnimationStarter，调用startAnimation，开始动画。

startAnimation 方法中会创建一个 ValueAnimator 对象设置内部监听器 AnimatorEventListener，并将 mPendingAnimations 和 要进行动画的属性名称封装成一个 PropertyBundle 对象，最后 mAnimatorMap 保存当前 Animator 和对应的PropertyBundle 对象。该Map将会在 animatePropertyBy 方法和 Animator 监听器 mAnimatorEventListener 中使用，启动动画。

在动画的监听器的 onAnimationUpdate 方法中设置所有属性的变化值，并通过RenderNode类优化绘制性能，最后刷新界面。有了整体概念后，现在我们沿着该工作流程图的路线来分析

ViewPropertyAnimator内部执行过程，从上图可以看出，通过 View#animate() 获取到 ViewPropertyAnimator 实例后，可以通过 ViewPropertyAnimator 提供的多种方法来设置动画，如 translationX()、scaleX() 等等，而当调用完这些方法后，其内部最终则会通过多次调用 animatorPropertyBy()，我们先看看 animatePropertyBy 方法源码：

```

/**
 * Utility function, called by animateProperty() and animatePropertyBy(), which
 handles the
 * details of adding a pending animation and posting the request to start the
 animation.
 *
 * @param constantName The specifier for the property being animated
 * @param startValue The starting value of the property
 * @param byValue The amount by which the property will change
 */
private void animatePropertyBy(int constantName, float startValue, float
byValue) {
    // First, cancel any existing animations on this property
    //判断该属性上是否存在运行的动画，存在则结束。
    if (mAnimatorMap.size() > 0) {
        Animator animatorToCancel = null;
        Set<Animator> animatorSet = mAnimatorMap.keySet();
        for (Animator runningAnim : animatorSet) {
            PropertyBundle bundle = mAnimatorMap.get(runningAnim);
            if (bundle.cancel(constantName)) { // 结束对应属性动画
                // property was canceled - cancel the animation if it's now
empty
                // Note that it's safe to break out here because every new
animation
                // on a property will cancel a previous animation on that
property, so
                // there can only ever be one such animation running.
                if (bundle.mPropertyMask == NONE) { //判断是否还有其他属性
                    // the animation is no longer changing anything - cancel it
                    animatorToCancel = runningAnim;
                    break;
                }
            }
        }
        if (animatorToCancel != null) {
            animatorToCancel.cancel();
        }
    }
    //将要执行的属性的名称，开始值，变化值封装成NameValuesHolder对象
    NameValuesHolder nameValuePair = new NameValuesHolder(constantName,
startValue, byValue);
    //添加到准备列表中
    mPendingAnimations.add(nameValuePair);
    mView.removeCallbacks(mAnimationStarter);
    mView.postOnAnimation(mAnimationStarter);
}

```

从源码可以看出，animatePropertyBy 方法主要干了以下几件事：

首先会去当前属性是否还有在动画在执行，如果有则先结束该属性上的动画，保证该属性上只有一个 Animator 在进行动画操作。将本次动画需要执行的动画属性封装成一个NameValueHolder对象 将每个 NameValuesHolder 对象添加到 mPendingAnimations 的准备列表中 NameValuesHolder对象是一个内部类，其相关信息如下：NameValueHolder： 内部类，封装每个要进行动画属性值开始值和变化值，比如 translationX(200)，那么这个动画的属性值、开始值和变化值将被封装成一个NameValueHolder，其源码也非常简单：

```

static class NameValuesHolder {
    int mNameConstant; //要进行动画的属性名称
    float mFromValue; //开始值
    float mDeltaValue; //变化值
    NameValuesHolder(int nameConstant, float fromValue, float deltaValue) {
        mNameConstant = nameConstant;
        mFromValue = fromValue;
        mDeltaValue = deltaValue;
    }
}

```

而 mPendingAnimations 的相关信息如下：mPendingAnimations：装载的是准备进行动画的属性值（NameValueHolder）所有列表，也就是每次要同时进行动画的全部属性的集合

ArrayList mPendingAnimations = new ArrayList(); 当添加完每个要运行的属性动画后，则会通过 mAnimationStarter 对象去调用 startAnimation()，启动动画。

Runnable mAnimationStarter：用来执行动画的 Runnable。它会执行 startAnimation 方法，而在 startAnimation 方法中会通过 animator.start() 启动动画，源码非常简洁：

```

private Runnable mAnimationStarter = new Runnable() {
    @Override
    public void run() {
        startAnimation();
    }
};

```

接着我们看看 startAnimation() 的源码：

```

/**
 * Starts the underlying Animator for a set of properties. We use a single
 * animator that
 * simply runs from 0 to 1, and then use that fractional value to set each
 * property
 * value accordingly.
 */
private void startAnimation() {
    if (mRTBackend != null && mRTBackend.startAnimation(this)) {
        return;
    }
    mView.setHasTransientState(true);
    //创建ValueAnimator
    ValueAnimator animator = ValueAnimator.ofFloat(1.0f);
    //clone一份mPendingAnimations赋值给nameValueList
    ArrayList<NameValuesHolder> nameValueList =
        (ArrayList<NameValuesHolder>) mPendingAnimations.clone();
    //赋值完后清空
    mPendingAnimations.clear();
    //用于标识要执行动画的属性
    int propertyMask = 0;
    int propertyCount = nameValueList.size();
    //遍历所有nameValuesHolder，取出其属性名称mNameConstant，
    //执行"|"操作并最终赋值propertyMask
    for (int i = 0; i < propertyCount; ++i) {
        NameValuesHolder nameValuesHolder = nameValueList.get(i);
        propertyMask |= nameValuesHolder.mNameConstant;
    }
}

```

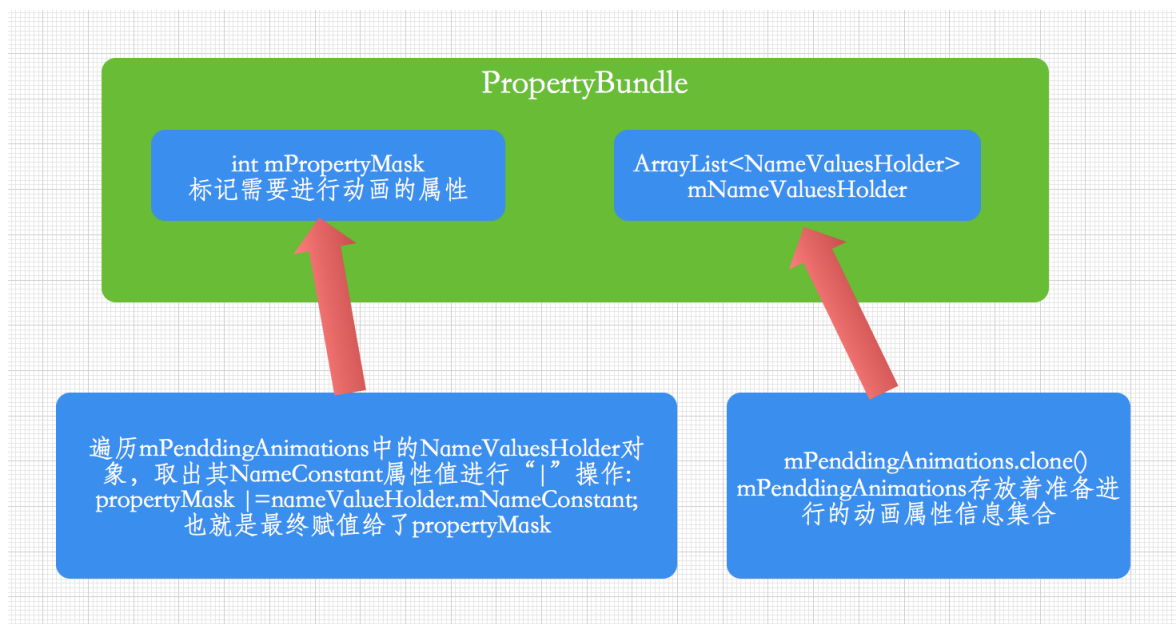
```

}
//创建PropertyBundle, 并添加到mAnimatorMap中
mAnimatorMap.put(animator, new PropertyBundle(propertyMask, nameValueList));
if (mPendingSetupAction != null) {
    //设置硬件加速
    mAnimatorSetupMap.put(animator, mPendingSetupAction);
    mPendingSetupAction = null;
}
if (mPendingCleanupAction != null) {
    //移除硬件加速
    mAnimatorCleanupMap.put(animator, mPendingCleanupAction);
    mPendingCleanupAction = null;
}
if (mPendingOnStartAction != null) {
    //设置开始的动画（监听器的开始方法中调用）
    mAnimatorOnStartMap.put(animator, mPendingOnStartAction);
    mPendingOnStartAction = null;
}
if (mPendingOnEndAction != null) {
    //设置结束后要进行的下一个动画（监听器的结束方法中调用）
    mAnimatorOnEndMap.put(animator, mPendingOnEndAction);
    mPendingOnEndAction = null;
}
//添加内部监听器
animator.addUpdateListener(mAnimatorEventListener);
animator.addListener(mAnimatorEventListener);
//判断是否延长开始
if (mStartDelaySet) {
    animator.setStartDelay(mStartDelay);
}
//执行动画的实现
if (mDurationSet) {
    animator.setDuration(mDuration);
}
//设置插值器
if (mInterpolatorSet) {
    animator.setInterpolator(mInterpolator);
}
//开始执行动画
animator.start();
}

```

我们上面的注释非常全面，这里 startAnimation 主要做下面几件事：

创建 Animator,变化值从 0 到 1，设置内部监听器 mAnimatorEventListener。clone 一份 mPendingAnimations 列表，并计算属性值标记 propertyMask，封装成 PropertyBundle 对象。使用 mAnimatorMap 保存当前 Animator 和对应的 PropertyBundle 对象。该Map 将会在 animatePropertyBy 方法和 Animator 监听器 mAnimatorEventListener 中使用。启动 animator 动画。关于PropertyBundle的分析如下：PropertyBundle：内部类，存放着将要执行的动画的属性集合信息，每次调用 animator.start(); 前，都会将存放在 mPendingAnimations 的 clone 一份存入 PropertyBundle 的内部变量 mNameValuesHolder 中，然后再将遍历 mPendingAnimations 中的 NameValueHolder 类，取出要执行的属性进行“|”操作,最后记录成一个 mPropertyMask 的变量，存放在 PropertyBundle 中，PropertyBundle 就是最终要执行动画的全部属性的封装类，其内部结构如下图



## PropertyBundle

AnimatorEventListener: ViewPropertyAnimator 内部的监听器。这个类实现了 Animator.AnimatorListener, ValueAnimator.AnimatorUpdateListener 接口。我们前面已经分享过它的部分源码，这个类还有一个 onAnimationUpdate() 的监听方法，这个方法我们放在后面解析，它是动画执行的关键所在。

HashMap mAnimatorMap: 存放 PropertyBundle 类的 Map。这个 Map 中存放的是正在执行的动画的 PropertyBundle，这个 PropertyBundle 包含这本次动画的所有属性的信息。最终在 AnimatorEventListener 的 onAnimationUpdate() 方法中会通过这个 map 获取相应的属性，然后不断更新每帧的属性值以达到动画效果。通过前面对 animatePropertyBy 方法的分析，我们可以知道该 Map 会保证当前只有一个 Animator 对象对该 View 的属性进行操作，不会存在两个 Animator 在操作同一个属性，其声明如下：

```
private HashMap<Animator, PropertyBundle> mAnimatorMap =  
    new HashMap<Animator, PropertyBundle>();
```

最后我们看看动画是在哪里执行的，根据我们前面的原理图，内部监听器的 onAnimationUpdate() 方法将会被调用（当然内部监听器 AnimatorEventListener 实现了两个动画监听接口，其开始，结束，重复，取消4个方法也会被调用，这个我们前面已分析过）。

```
@Override  
public void onAnimationUpdate(ValueAnimator animation) {  
    //取出当前Animator对应用propertyBundle对象  
    PropertyBundle propertyBundle = mAnimatorMap.get(animation);  
    if (propertyBundle == null) {  
        // Shouldn't happen, but just to play it safe  
        return;  
    }  
    //是否开启了硬件加速  
    boolean hardwareAccelerated = mView.isHardwareAccelerated();  
  
    // alpha requires slightly different treatment than the other (transform)  
    properties.  
    // The logic in setAlpha() is not simply setting mAlpha, plus the  
    invalidation  
    // logic is dependent on how the view handles an internal call to  
    onSetAlpha().
```

```

        // We track what kinds of properties are set, and how alpha is handled when
        it is
        // set, and perform the invalidation steps appropriately.
        boolean alphaHandled = false;
        if (!hardwareAccelerated) {
            mView.invalidateParentCaches();
        }
        //取出当前的估算值（插值器计算值）
        float fraction = animation.getAnimatedFraction();
        int propertyMask = propertyBundle.mPropertyMask;
        if ((propertyMask & TRANSFORM_MASK) != 0) {
            mView.invalidateViewProperty(hardwareAccelerated, false);
        }
        //取出所有要执行的属性动画的封装对象NameValuesHolder
        ArrayList<NameValuesHolder> valueList = propertyBundle.mNameValuesHolder;
        if (valueList != null) {
            int count = valueList.size();
            //遍历所有NameValuesHolder，计算变化值，并设置给对应的属性
            for (int i = 0; i < count; ++i) {
                NameValuesHolder values = valueList.get(i);
                float value = values.mFromValue + fraction * values.mDeltaValue;
                if (values.mNameConstant == ALPHA) {
                    alphaHandled = mView.setAlphaNoInvalidation(value);
                } else {
                    setValue(values.mNameConstant, value);
                }
            }
        }
        if ((propertyMask & TRANSFORM_MASK) != 0) {
            if (!hardwareAccelerated) {
                mView.mPrivateFlags |= View.PFLAG_DRAWN; // force another
            }
            invalidation
        }
        // invalidate(false) in all cases except if alphaHandled gets set to true
        // via the call to setAlphaNoInvalidation(), above
        if (alphaHandled) {
            mView.invalidate(true);
        } else {
            mView.invalidateViewProperty(false, false);
        }
        if (mUpdateListener != null) {
            mUpdateListener.onAnimationUpdate(animation);
        }
    }
}

```

onAnimationUpdate方法主要做了以下几件事：

取出当前 Animator 对应用 propertyBundle 对象并获取当前的估算值（插值器计算值），用于后续动画属性值的计算 从 propertyBundle 取出要进行动画的属性列表 ArrayList valueList 遍历所有 NameValuesHolder，计算变化值，并通过 setValue 设置给对应的属性，如果是 ALPHA，则会特殊处理一下，最终形成动画效果 setValue方法源码：

```

private void setValue(int propertyConstant, float value) {
    final View.TransformationInfo info = mView.mTransformationInfo;
    final RenderNode renderNode = mView.mRenderNode;
    switch (propertyConstant) {

```

```

        case TRANSLATION_X:
            renderNode.setTranslationX(value);
            break;
        case TRANSLATION_Y:
            renderNode.setTranslationY(value);
            break;
        case TRANSLATION_Z:
            renderNode.setTranslationZ(value);
            break;
        case ROTATION:
            renderNode.setRotation(value);
            break;
        case ROTATION_X:
            renderNode.setRotationX(value);
            break;
        case ROTATION_Y:
            renderNode.setRotationY(value);
            break;
        case SCALE_X:
            renderNode.setScaleX(value);
            break;
        case SCALE_Y:
            renderNode.setScaleY(value);
            break;
        case X:
            renderNode.setTranslationX(value - mView.mLeft);
            break;
        case Y:
            renderNode.setTranslationY(value - mView.mTop);
            break;
        case Z:
            renderNode.setTranslationZ(value - renderNode.getElevation());
            break;
        case ALPHA:
            info.mAlpha = value;
            renderNode.setAlpha(value);
            break;
    }
}

```

从源码可以看出实际上都会把属性值的改变设置到 renderNode 对象中，而 RenderNode 类则是一个可以优化绘制流程和绘制动画的类，该类可以提升优化绘制的性能，其内部操作最终会去调用到 Native 层方法，这里我们就不深追了。最后这里我们再回忆一下前面给出的整体流程说明：

通过imageView.animate()获取ViewPropertyAnimator对象。调用alpha、translationX等方法，返回当前ViewPropertyAnimator对象，可以继续链式调用 alpha、translationX等方法内部最终调用 animatePropertyBy(int constantName, float startValue, float byValue)方法 在animatePropertyBy方法中则会将alpha、translationX等方法的封装成NameVauleHolder，并将每个 NameValueHolder对象添加到准备列表mPendingAnimations中。animatePropertyBy方法启动 mAnimationStarter，调用startAnimation，开始动画。startAnimation方法中会创建一个 ValueAnimator对象设置内部监听器AnimatorEventListener，并将mPendingAnimations和要进行动画的属性名称封装成一个PropertyBundle对象，最后mAnimatorMap保存当前Animator和对应的PropertyBundle对象。该Map将会在animatePropertyBy方法和Animator监听器 mAnimatorEventListener中使用，启动动画。在动画的监听器的onAnimationUpdate方法中设置所有属性的变化值，并通过RenderNode类优化绘制性能，最后刷新界面。



## 4. Java 属性动画拓展之 LayoutAnimator 容器布局动画

Property 动画系统还提供了对 ViewGroup 中 View 添加时的动画功能，我们可以用 LayoutTransition 对 ViewGroup 中的 View 进行动画设置显示。LayoutTransition 的动画效果都是设置给 ViewGroup，然后当被设置动画的 ViewGroup 中添加删除 View 时体现出来。该类用于当前布局容器中有 View 添加、删除、隐藏、显示等时候定义布局容器自身的动画和 View 的动画，也就是说当在一个 LinerLayout 中隐藏一个 View 的时候，我们可以自定义 整个由于 LinerLayout 隐藏 View 而改变的动画，同时还可以自定义被隐藏的 View 自己消失时候的动画等。

我们可以发现 LayoutTransition 类中主要有五种容器转换动画类型，具体如下：

- LayoutTransition.APPEARING：当View出现或者添加的时候View出现的动画。
- LayoutTransition.CHANGE\_APPEARING：当添加View导致布局容器改变的时候整个布局容器的动画。
- LayoutTransition.DISAPPEARING：当View消失或者隐藏的时候View消失的动画。
- LayoutTransition.CHANGE\_DISAPPEARING：当删除或者隐藏View导致布局容器改变的时候整个布局容器的动画。
- LayoutTransition.CHANGE：当不是由于View出现或消失造成对其他View位置造成改变的时候整个布局容器的动画。

### 4-1 XML 方式使用系统提供的默认 LayoutTransition 动画

我们可以通过如下方式使用系统提供的默认ViewGroup的LayoutTransition动画：

```
android:animateLayoutChanges="true"
```

在 ViewGroup 添加如上 xml 属性默认是没有任何动画效果的，因为前面说了，该动画针对于 ViewGroup 内部东东发生改变时才有效，所以当我们设置如上属性然后调运 ViewGroup 的 addView、removeView 方法时就能看见系统默认的动画效果了。

还有一种就是通过如下方式设置：

```
android:layoutAnimation="@anim/customer_anim"
```

通过这种方式就能实现很多吊炸天的动画,并在加载布局的时候就会自动播放 layout-animtion。其中设置的动画位于 res/anim 目录下的动画资源（如下）：

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
    android:delay="30%"
    android:animationOrder="reverse"
    android:animation="@anim/slide_right"/>
```

每个属性的作用：

- `android:delay` 表示动画播放的延时，既可以是百分比，也可以是 float 小数。
- `android:animationOrder` 表示动画的播放顺序，有三个取值 normal(顺序)、reverse(反序)、random(随机)。
- `android:animation` 指向了子控件所要播放的动画。

### 4-2 Java 方式使用系统提供的默认 LayoutTransition 动画

在使用LayoutTransition时，你可以自定义这几种事件类型的动画，也可以使用默认的动画，总之最终都是通过 `setLayoutTransition(LayoutTransition lt)` 方法把这些动画以一个 LayoutTransition 对象设置给一个 ViewGroup。



譬如实现如上 Xml 方式的默认系统 LayoutTransition 动画如下：

```
mTransitioner = new LayoutTransition();
mViewGroup.setLayoutTransition(mTransitioner);
```

如果在xml中文件已经写好 LayoutAnimation，可以使用 AnimationUtils 直接加载：

```
AnimationUtils.loadLayoutAnimation(context, id)
```

另外还可以手动java代码编写，如：

```
//通过加载XML动画设置文件来创建一个Animation对象；
Animation animation=AnimationUtils.loadAnimation(this, R.anim.slide_right); //
得到一个LayoutAnimationController对象；
LayoutAnimationController controller = new LayoutAnimationController(animation);
//设置控件显示的顺序；
controller.setOrder(LayoutAnimationController.ORDER_REVERSE); //设置控件显示间隔
时间；
controller.setDelay(0.3); //为ListView设置LayoutAnimationController属性；
listview.setLayoutAnimation(controller);
listview.startLayoutAnimation();
```

## 4-3 LayoutTransition 的用法

稍微再高端一点吧，我们来自定义这几类事件的动画，分别实现他们，那么你可以像下面这么处理：

```
mTransitioner = new LayoutTransition();
.....
ObjectAnimator anim = ObjectAnimator.ofFloat(this, "scaleX", 0, 1);
.....//设置更多动画
mTransition.setAnimator(LayoutTransition.APPEARING, anim);
.....//设置更多类型的动画
mViewGroup.setLayoutTransition(mTransitioner);
```

到此通过 LayoutTransition 你就能实现类似小米手机计算器切换普通型和科学型的炫酷动画了。

# 动画动画占用大量内存，如何优化？

## 帧动画优化

当我们使用 `AnimationDrawable` 用于实现帧动画。在动画开始之前，所有帧的图片都被解析并占用内存，一旦动画较复杂帧数较多，在低配置手机上容易发生 OOM。即使不发生 OOM，也会对内存造成不小的压力，那么如何来优化呢？

有没有什么办法让帧动画的数据逐帧加载，而不是一次性全部加载到内存？`SurfaceView` 就提供了这种能力。

## SurfaceView

屏幕的显示机制和帧动画类似，也是一帧一帧的连环画，只不过刷新频率很高，感觉像连续的。为了显示一帧，需要经历计算和渲染两个过程，CPU 先计算出这一帧的图像数据并写入内存，然后调用 OpenGL 命令将内存中数据渲染成图像存放在 GPU Buffer 中，显示设备每隔一定时间从 Buffer 中获取图像并显示。

上述过程中的计算，对于 `view` 来说，就好比在主线程遍历 View 树 以决定视图画多大（measure），画在哪（layout），画些啥（draw），计算结果存放在内存中，SurfaceFlinger 会调用 OpenGL 命令将内存中的数据渲染成图像存放在 GPU Buffer 中。每隔16.6ms，显示器从 Buffer 中取出帧并显示。所以自定义 View 可以通过重载 `onMeasure()`、`onLayout()`、`onDraw()` 来定义帧内容，但不能定义帧刷新频率。

`surfaceview` 可以突破这个限制。而且它可以将计算帧数据放到独立的线程中进行

- 利用Bitmap **绘制一帧解析一张**
- 将 Bitmap 的解析参数 `inBitmap` 设置为已经成功解析的 Bitmap 对象以实现复用

## 自定义动画优化

- 移动图片资源至大分辨率目录下，比如xxxhdpi
- 动画完成且不再循环展示的部分，相关bitmap释放
- 无用对象释放，非透明背景图片采用RGB\_565颜色格式，并且将图片的inSampleSize设置为2
- 能在初始化中做的事，坚决不在onDraw中做
- 能用硬件加速，就别关闭它
- 优化算数运算，并尽量从ondraw中移除算数运算







