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。

|  |  |  |
| --- | --- | --- |
| 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 |

#### ****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 |

#### ****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 |

#### ****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 |

#### ****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 |

#### ****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轴从标，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 |

#### ****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。

### ****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

<?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>

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。

### ****视图动画注意事项****

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。

### ****视图动画Interpolator插值器详解****

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。

#### ****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>

#### ****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。如下我们来说说。

先看看XML自定义插值器的步骤：

* 在res/anim/目录下创建filename.xml文件。
* 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"

/>

* 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：

<accelerateDecelerateInterpolator>

无可自定义的attribute。

<accelerateInterpolator>

android:factor 浮点值，加速速率（默认值为1）。

<anticipateInterploator>

android:tension 浮点值，起始点后拉的张力数（默认值为2）。

<anticipateOvershootInterpolator>

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）。

<bounceInterpolator>

无可自定义的attribute。

<cycleInterplolator>

android:cycles 整形，循环的个数（默认为1）。

<decelerateInterpolator>

android:factor 浮点值，减速的速率（默认为1）。

<linearInterpolator>

无可自定义的attribute。

<overshootInterpolator>

android:tension 浮点值，超出终点后的张力（默认为2）。

再来看看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;

}

......

}

## ****Drawable Animation（Drawable动画）使用详解****

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。

AnimationDrawable实际的真实父类就是Drawable。

### ****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。

### ****3-3 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>

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。

### ****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.

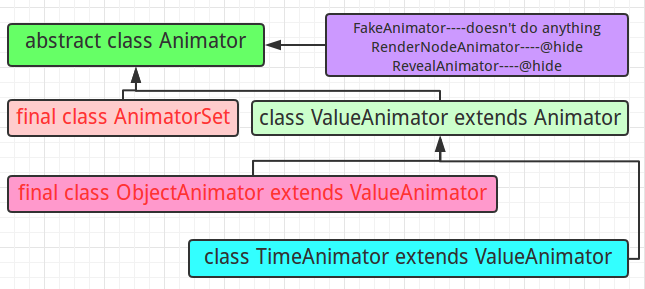
\*/public abstract class Animator implements Cloneable {

......

}

所有的属性动画的抽象基类就是它。

属性动画的实现有7个类，好几个是hide的类，而其他可用的类继承关系又如下：



| **java类名** | **xml关键字** | **描述信息** |
| --- | --- | --- |
| ValueAnimator | <animator> 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 | 时序监听回调工具 |
| ObjectAnimator | <objectAnimator> 放置在res/animator/目录下 | 一个对象的一个属性动画 |
| AnimatorSet | <set> 放置在res/animator/目录下 | 动画集合 |

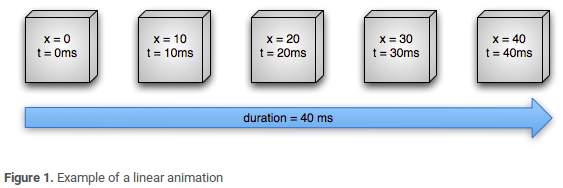
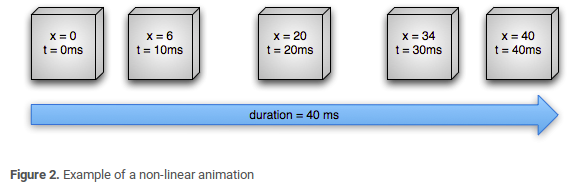
所以可以看见，我们平时使用属性动画的重点就在于AnimatorSet、ObjectAnimator、TimeAnimator、ValueAnimator。所以接下来我们就来依次说说如何使用。

### ****属性动画详细说明****

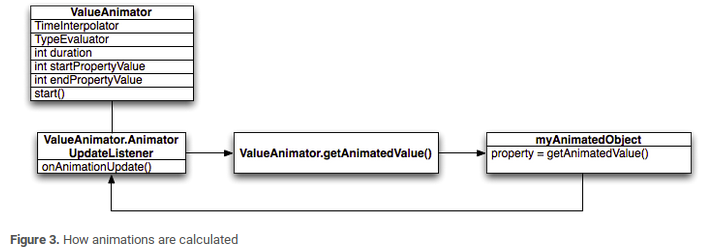
Android属性动画提供了以下属性：

* 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 refreash 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 Country and behavoir：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）：

   
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=10pixel。   
   
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。



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 ms，t = 10 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 = 10ms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=10ms/40ms=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。

#### ****XML方式属性动画****

在xml中可直接用的属性动画节点有ValueAnimator、ObjectAnimator、AnimatorSet。如下是官方的一个例子和解释：

<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），辟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.animtor.property\_animator);set.setTarget(myObject);

set.start();

#### ****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这个三个常用的方法，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，那就可以用到此类，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();

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

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：**

**AccelerateDecelerateInterolator：**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。

**CycleIinterpolator：**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。

**OvershottInterpolator：**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。

#### ****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();

#### ****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。

## ****Android动画总结****

到此Android动画基本已经描述OK了，也就这么三大类，尤其是属性动画更加一筹。但是特别说一句，上面基本都没有提及到各种动画的 Listener接口，原因是这个玩意太简单，所以不提了，相信你会监听View的onClickListener就一定会触类旁通动画的 Listener方法的。有了这些基础相信无论是自定义控件时还是自定义动画时都会起到直接的指导参考作用。其实对于Android的动画实现远远不止现 在提到的这些，但是这些又是基础，所以后面还会写文章说说Android提供的其他动画参考工具类的。

现在我们继续沿用官方的对比，翻译一下这些动画的区别，具体如下：

**View动画：**

View动画只能够为View添加动画，如果想为非View对象添加动画须自己实现；且View动画支持的种类很少；尤其是他改变的是View的绘制效果，View的属性没有改变，其位置与大小都不变； View动画代码量少，使用简单方便。

**Property动画：**

弥补了View动画的缺陷，你可以为一个对象的任意属性添加动画，对象自己的属性会被真的改变；当对象的属性变化的时候，属性动画会自动刷新屏幕；属性动画改变的是对象的真实属性，而且属性动画不止用于View，还可以用于任何对象。