**View动画**

Android 动画分为三种：View动画，帧动画，属性动画。

https://www.ibm.com/developerworks/cn/opensource/os-cn-android-anmt1/

* **View动画http://www.cnblogs.com/yc-755909659/p/4290114.html**

Veiw动画的作用对象是View，它支持四种动画效果对应着Animation的四个子类：

|  |  |  |
| --- | --- | --- |
| 名称 | Xml标签 | Animation子类 |
| 平移动画 | <translate> | TranslateAnimation |
| 缩放动画 | <scale> | ScaleAnimation |
| 旋转动画 | <rotate> | RotateAnimation |
| 透明度动画 | <alpha> | AlphaAnimation |

|  |  |  |
| --- | --- | --- |
| 动画集合 | <set> | AnimationSet |

**在XML中的使用**，首先动画的xml文件一定存在于res/anim/，文件下：

*<?***xml version="1.0" encoding="utf-8"***?>*<**set xmlns:android="http://schemas.android.com/apk/res/android"**>  
 <!-- 平移动画  
  
 整型数值：  
  
 fromXDelta：起始点x坐标  
 fromYDelta：起始点y坐标  
 toXDelta：结束点x坐标  
 toYDelta：结束点y坐标  
  
 注意：以上fromXDelta，fromYDelta，toXDelta，toYDelta这四个值都是相对于控件的左上角的坐标（x,y）而言的，假如我们只关注运动距离，那么可以把控件左上角坐标视为（0,0）：

0：代表控件的原点。

起始位置 x = x + fromXDelta; y = y + fromYDelta;

终止位置 x = x + fromXDelta + toXDelta; y = y + fromYDelta + toYDelta;

x轴上小于0，表明在原点的左侧，大于0则在原点右侧。

y轴上小于0，表明在原点的上方，大于0则在原点下方。

没有指定fromXType，fromYType，toXType,toYTypes时候，默认是以自己为相对参照物。  
 -->  
<**translate  
 android:fromXDelta="100"  
 android:fromYDelta="100"  
 android:toXDelta="300"  
 android:toYDelta="300"** />  
  
  
 <!-- 缩放动画  
  
 浮点型数值：  
 fromXScale：水平方向上缩放的起始值  
 fromYScale：竖直方向上缩放的起始值  
 toXScale：水平方向上缩放的结束值  
 toYScale：竖直方向上缩放的结束值  
  
 说明：以上四种值  
  
 0.0表示收缩到没有  
 1.0表示正常无伸缩  
 值小于1.0表示收缩  
 值大于1.0表示放大  
  
 pivotX：动画相对于物件的x坐标的开始位置  
 pivotY：动画相对于物件的y坐标的开始位置  
  
 说明：

缩放起点X轴坐标，可以是数值、百分数、百分数p 三种样式，比如 50、

50%、50%p，当为数值时，表示在当前View的左上角，即原点处加上50px， 做为起始缩放点；如果是50%，表示在当前控件的左上角加上自己宽度

的50%做为起始点；如果是50%p，那么就是表示在当前的左上角加上父

控件宽度的50%做为起始点x轴坐标

*-->* <**scale  
 android:fromXScale="0.5"  
 android:fromYScale="0.5"  
 android:pivotX="50%"  
 android:pivotY="50%"  
 android:toXScale="1.5"  
 android:toYScale="1.5"** />  
  
  
 <!-- 旋转动画  
  
 浮点型数值：  
 fromDegrees：动画起始时物件的角度  
 toDegrees：动画结束时物件的角度  
  
 说明：  
 当toDegrees角度为正数时——表示顺时针旋转  
 当toDegrees角度为负数时——表示逆时针旋转  
 (负数from——to正数:顺时针旋转)  
 (负数from——to负数:逆时针旋转)  
 (正数from——to正数:顺时针旋转)  
 (正数from——to负数:逆时针旋转)  
  
 pivotX 属性为动画相对于物件的X坐标的开始位置  
 pivotY 属性为动画相对于物件的Y坐标的开始位置  
  
 说明: 以上两个属性取值同缩放动画pivotX和pivotY

-->  
<**rotate  
 android:fromDegrees="10.0"  
 android:pivotX="50%"  
 android:pivotY="50%"  
 android:toDegrees="370.5"** />  
  
   
 <!-- 透明度动画  
  
 浮点型数值：  
 fromAlpha：动画起始时的透明度  
 toAlpha：动画结束时的透明度  
  
 说明：以上取值范围 0.0—1.0 之间的浮点型数值  
 0.0 表示完全透明  
 1.0 表示完全不透明  
 -->  
<**alpha  
 android:fromAlpha="0.2"  
 android:toAlpha="0.8"** />

<!-- 嵌套其他动画集合-->

<**set**>

...

</**set**>  
</**set**>

**除了以上的属性还有一些公共的属性：**

android:duration="1000"：动画的执行时间，单位毫秒，假如<set>标签和具体动画都设定了这个属

性那么以<set> 标签设置的为主。

android:repeatCount="3"：动画重复次数。”-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=""：动画重复模式，restart重复播放，reverse:倒叙播放。

注意：repeatCount和repeatMode，只有设置在具体的动画上才有效果。

android:startOffset：动画运行前，延迟的毫秒数。

注意：比如设置缩放动画，从0.3到1.5的缩放，设置的延迟时间为3000ms,

运行前是指动画准备好初始状态之后，即view先缩小到0.3这个状态之

后计时3000ms的延迟，在开始动画。 假如是个集合动画就有意思了，

比如集合里有三个动画，第一个动画延时1s，第二个延时2s，第三个延

时3s。那么当执行view.startAnimation(animSet)后，view会将自己的

状态同时变成这三个动画的初始值，然后计时，1s后执行第一个动画，

计时到了第2s会执行第二个动画，计时到了第3s执行第三个动画。

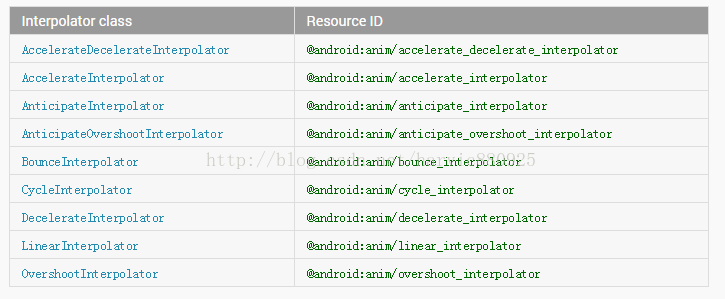
android:fillAfter：动画执行后是否保持动画结束后的状态。true:是，false：不是。这个属性只能 在<set></set>标签中设置才会生效。

android:fillBefore：如果设置为true,控件动画结束时，还原到开始动画前的状态。

android:fillEnabled：与android:fillBefore 相同，都是在动画结束时，将控件还原到初始状态。

android:interpolator: 动画的插值器，默认@android:anim/accelerate\_decelerate\_interpolator,

即匀速加速插值器。



# **常用继承类：**

AccelerateDecelerateInterpolator：动画开始与结束的地方速率改变比较慢，在中间的时候

加速。

AccelerateInterpolator：动画开始的地方速率改变比较慢，然后开始加速。

AnticipateInterpolator ：开始的时候向后然后向前甩。

AnticipateOvershootInterpolator：开始的时候向后然后向前甩一定值后返回最后的值。

BounceInterpolator：动画结束的时候弹起。

CycleInterpolator：动画循环播放特定的次数，速率改变沿着正弦曲线。

DecelerateInterpolator：在动画开始的地方快然后慢。

LinearInterpolator：以常量速率改变。

OvershootInterpolator：向前甩一定值后再回到原来位置。

PathInterpolator：就是可以定义路径坐标，然后可以按照路径坐标来跑动；注意其坐标并不

是 XY，而是单方向，也就是我可以从0~1，然后弹回0.5 然后又弹到0.7

有到0.3，直到最后时间结束。

android:shareInterpolator:表示集合中动画是否和集合共享一个插值器。

View动画可以是一系列动画，也可以是单个动画，但是必须在<set></set>标签内，<set>标签标示动画集合，对应着AnimationSet类，她可以包含若干个动画，并且它的内部也可以嵌套其他的动画。

* **使用XML中的动画效果**

Animation animation = AnimationUtils.*loadAnimation*(**this**, R.anim.***my\_anim***);  
 **view**.startAnimation(animation);

* **View动画监听**

animation.setAnimationListener(**new** Animation.AnimationListener() {  
 @Override  
 **public void** onAnimationStart(Animation animation) {  
 }  
  
 @Override  
 **public void** onAnimationEnd(Animation animation) {  
 }  
  
 @Override  
 **public void** onAnimationRepeat(Animation animation) {  
 }  
 });

* **自定义View动画**

自定义动画是一种既简单又复杂的事情，说它简单，是因为派生一个新的动画只需要继承Animation这个抽象类，重写initialize和applyTransformation方法，在initialize方法中做一些初始化的工作，在applyTransformation方法中进行相应的矩阵变换即可，很多时候需要采用Camera来简化矩阵变换过程。说他复杂是因为自定义View动画过程主要是矩阵变换的过程，而矩阵变换是数学上的概念，如果对这方面的只是不熟悉的话，就会觉得过程比较复杂。

* **帧动画**

帧动画就是播放预先准本好的一组图片。

在XML中使用首先动画的xml文件一定存在于res/drawable/，文件下：

*<?***xml version="1.0" encoding="utf-8"***?>*<**animation-list xmlns:android="http://schemas.android.com/apk/res/android"  
 android:oneshot="false"**>  
 <!--  
 根标签为animation-list，其中oneshot代表着是否只展示一遍，设置为false会不停的循环播放动

画  
 根标签下，通过item标签对动画中的每一个图片进行声明  
 android:duration 表示展示所用的该图片的时间长度  
 -->  
<**item  
 android:drawable="@drawable/ic\_launcher"  
 android:duration="150"**></**item**>  
 <**item  
 android:drawable="@drawable/ic\_launcher2"  
 android:duration="150"**></**item**>  
 <**item  
 android:drawable="@drawable/ic\_launcher3"  
 android:duration="150"**></**item**>  
 <**item  
 android:drawable="@drawable/ic\_launcher4"  
 android:duration="150"**></**item**>  
</**animation-list**>

**代码中使用：**由AnimationDrawable控制开始和停止

**img** = (ImageView) findViewById(R.id.***text***);  
**img**.setImageResource(R.drawable.***anim\_list***);  
//给动画资源赋值AnimationDrawable ad = (AnimationDrawable) **img**.getDrawable();  
ad.start();  
//ad.stop();

、

## **ViewGroup动画LayoutAnimation**

LayoutAnimation作用与ViewGroup，为ViewGroup指定一个动画，这样当它的子元素出场时都会具有这种效果（也可以应用在RecycleView等等），**这里注意，只有在第一次加载的时候容器中的item才会有动画，其他时候没有，要想每次显示都有效果，需要使用代码的方式设置LayoutAnimation动画，然后每次在View显示的时候在代码中设置view.startLayoutAnimation(controller);**，他也是存在于res/anim/，文件夹下，anim\_layout.xml中定义：

*<?***xml version="1.0" encoding="utf-8"***?>*<**layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"  
 android:animation="@anim/anim\_item"  
 android:animationOrder="normal"  
 android:delay="50%"**>  
 <!--  
 **android:delay** 表示每个子元素动画播放的延时，取值是android:animation所指定动画时长的 倍数，即设置的android:duration值的倍数，既可以是百分比，也可以是float小数。比如 子元素设置的duration为1000毫秒，设置0.5 或者50% ，表示每个子元素都要延时500毫秒 才能播放入场动画。  
 **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。  
 --></**layoutAnimation**>

在布局文件中ViewGroup下设置属性**android:layoutAnimation="@anim/anim\_layout"**这样加载布局文件的时候相应的ViewGroup控件的子元素就自动播放动画。

**在代码中设置LayoutAnimation:**

LinearLayout linear = (LinearLayout) findViewById(R.id.***linear***);  
 Animation animation = AnimationUtils.*loadAnimation*(**this**, R.anim.***anim\_layout***);  
 LayoutAnimationController controller = **new** LayoutAnimationController(animation);  
 controller.setDelay(0.3f);  
 linear.setLayoutAnimation(controller);  
 linear.startLayoutAnimation();

**或**  
 LayoutAnimationController controller= AnimationUtils.*loadLayoutAnimation*(**this**,R.anim.***anim\_layout***);

linear.setLayoutAnimation(controller);  
 linear.startLayoutAnimation();

* **Activity的切换效果**

Activity有默认的效果，但是我们是可以自定义的，主要用到：

overridePendingTransition(int enterAnim,int exitAnim);

这个方法必须在startActivity（Intent）或者finish()方法之后调用才能生效。

enterAnim：Activity被打开时的动画资源id。

exitAnim：Activity被暂停时的动画，资源id。

**示例：**

Intent intent =**new** Intent(**this**,OtherActivity.**class**);  
 **this**.startActivity(intent);  
 overridePendingTransition(R.anim.enterAnim,R.anim.exitAnim);

和

@Override  
 **public void** finish() {  
 **super**.finish();  
 overridePendingTransition(R.anim.enterAnim,R.anim.exitAnim);  
 }

对于Fragment也可以添加切换动画，由于Fragment是API 11中新引入的类，因此为了兼容性，我们需要使用support-4这个兼容包，这种情况下我们通过FragmentTransaction中的setCustomAnimations()方法来添加切换动画 。

* **属性动画**

属性动画是Android系统3.0新加入的特性，它作用于任何对像，而不是局限于View对象。它的实质是：在一个时间间隔内把一个对象的属性从一个值改变到另一个值。因此属性动画无所不能，只要对象有这个属性，他都能实现动画效果。对于Android 3.0系统以下可以使用开原动画库nineoldandroids来兼容。

* **在XML中的使用**

属性动画的xml文件存放在res/animator，文件夹下：

<[set](https://developer.android.com/guide/topics/resources/animation-resource.html" \l "animator-set-element)  
  android:ordering=["together" | "sequentially"]>  
  
    <[objectAnimator](https://developer.android.com/guide/topics/resources/animation-resource.html" \l "obj-animator-element)  
        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](https://developer.android.com/guide/topics/resources/animation-resource.html" \l "val-animator-element)  
        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](https://developer.android.com/guide/topics/resources/animation-resource.html" \l "animator-set-element)>  
        ...  
    </set>  
</set>

或，不使用动画集合，单独使用一个动画

<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"  
    android:duration="1000"  
    android:valueTo="200"  
    android:valueType="floatType"  
    android:propertyName="y"  
    android:repeatCount="1"  
    android:repeatMode="reverse"/>

注意：在API 23（系统6.0之后）还可以组合<propertyValuesHolder/>和<keyframe/>两个标签一起使用：

<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"  
                android:duration="1000"  
                android:repeatCount="1"  
                android:repeatMode="reverse">  
    <propertyValuesHolder android:propertyName="x" android:valueTo="400"/>  
    <propertyValuesHolder android:propertyName="y" android:valueTo="200"/>

<propertyValuesHolder android:propertyName="x" >  
    <keyframe android:fraction="0" android:value="800" />  
    <keyframe android:fraction=".2"  
              android:interpolator="@android:anim/accelerate\_interpolator"  
              android:value="1000" />  
    <keyframe android:fraction="1"  
              android:interpolator="@android:anim/accelerate\_interpolator"  
              android:value="400" />  
 </propertyValuesHolder>  
</objectAnimator>

上面示例列出了三个标签<set>、<objectAnimator>、<animator>分别对应AnimatorSet、ObjectAnimator和ValueAnimator三个类。

* **标签内属性：**

**<set>：**动画集合。

ordering:定义动画集合内的动画的执行顺序，默认together ：

Sequentially 按顺序执行

together  同时执行

**<objectAnimator>：**针对于对象的动画。

propertyName：即ObjectAnimator需要通过反射查找对应属性的setProperty()函数

的那个property字符串。这里注意ObjectAnimator操作的对象不一定 存在property这个属性，但是一定要存在相应的setProperty()方法， 如果动画没有指定初始值，那么还需要提供getProperty()方法。

例如：动画设置propertyName为＂abc＂，那么ObjectAnimator

操作的对象里不一定要有abc这个属性，但是一定要存在setAbc()这个

方法，如果动画没有设置初始值，那么一定要设置getAbc()这个方法。

valueFrom:动画的起始值。类型：float | int | color

valueTo:动画的结束值。类型：float | int | color

valueType：操作相应值的类型，默认是float类型,如果是颜色类型那么不需要指定，

动画框架自动处理颜色类型。

intType 整数型 floatType 浮点型

在api19 里面添加了 android:propertyXName android:propertyYName 俩个更加细分的属性。其他属性都和View动画是一样的效果。

在代码中调用：

AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(myContext,  
    R.anim.property\_animator);  
set.setTarget(myObject);  
set.start();

* **相应类解读**

属性动画原理：属性动画要求动画作用的对象提供相应属性的get和set方法（我们知道没有相应属性，只有这两个方法也是可以的），属性动画根据外界提供的初始值和最终值，以动画的效果多次去调用set方法，每次传递给set方法的值都不一样，确切的说随着时间的推移，所传递的值越来越接近最终值。

动画时间：默认的动画时间是300毫秒。

动画刷新：默认10ms/帧。但是最终的刷新速度取决于你的系统是否繁忙，以及手机的 性能，可以设置。

属性动画作用于View，可以作用的属性有：

* translationX 、translationY、rotation、 rotationX、rotationY、scaleX、scaleY、pivotX 、pivotY、x 、y、alpha。
* **ValueAnimator**

ValueAnimator不作用于任何对像，直接使用没有任何效果。它可以对一个值做动画，然后我们监听其动画过程，在动画过程中修改对象的值，这样也相当于给对象做了动画。它并不需要get和set方法。我们可以使用ValueAnimator的工厂方法，ofInt()、ofFloat()、ofObject()方法来获取一个动画对象。例如：

ValueAnimator animation = ValueAnimator.ofFloat(0f, 100f);  
animation.setDuration(1000);  
animation.start();

这是ValueAnimator最基本的用法了，它表示，在1000ms内将0f增加到100f。我们可以添加一个监听器，来监听使用这个变化的值：

animation.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {  
    @Override  
    public void onAnimationUpdate(ValueAnimator updatedAnimation) {  
        // You can use the animated value in a property that uses the  
        // same type as the animation. In this case, you can use the  
        // float value in the translationX property.  
        float animatedValue = (float)updatedAnimation.getAnimatedValue();  
        textView.setTranslationX(animatedValue);  
    }  
});

* **ObjectAnimator**

首先它继承自ValueAnimator类。

ObjectAnimator animation = ObjectAnimator.ofFloat(textView, "translationX", 0f,100f );  
animation.setDuration(1000);  
animation.start();

上面示例表示将对象textview的translationX的值在1秒内从0f变化到100f，显示出来的效果就是这个textview在一秒内从左向右运动了100像素。

我们知道ObjectAnimator作用的对象必须提供set方法，假如这个对象没有提供相应的set方法，官方给出以下解决方案：

1. 如果你有权限，你可以给该类添加set方法。
2. 我们可以使用一个包装类，包装这个对象，然后在包装类里面提供set方法，最后

让属性动画作用于这个包装类，在这个包装类的set方法里面操作实际对象呈现动

画效果。

1. 使用ValueAnimator代替ObjectAnimator。

另外假如通过ObjectAnimator.ofXXX方法只传递了一个值，那么这个值会被认为是结束值，这时候我们需要给出get方法提供初始值，并且get方法和set方法操作的值必须是同一类型。

## **AnimatorSet**

和ValueAnimator一样继承自Animator。它可以添加一组动画，并可以指定这一组动画的执行方式：一起执行还是顺序执行，亦或是在某个动画之前或之后插播另一个动画，并且还可以指定两个动画执行之间的延迟时间等等。

AnimatorSet bouncer = new AnimatorSet();  
bouncer.play(bounceAnim).before(squashAnim1);  
bouncer.play(squashAnim1).with(squashAnim2);  
bouncer.play(squashAnim1).with(stretchAnim1);  
bouncer.play(squashAnim1).with(stretchAnim2);  
bouncer.play(bounceBackAnim).after(stretchAnim2);  
ValueAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);  
fadeAnim.setDuration(250);  
AnimatorSet animatorSet = new AnimatorSet();  
animatorSet.play(bouncer).before(fadeAnim);  
animatorSet.start();

* **动画监听**

1. [ValueAnimator.AnimatorUpdateListener](https://developer.android.com/reference/android/animation/ValueAnimator.AnimatorUpdateListener.html)比较特殊，它会监听整个动画的过程，动画是又许多帧组成，每播放一帧onAnimationUpdate就会被调用一次。

valueAnimator.addUpdateListener(**new** ValueAnimator.AnimatorUpdateListener() {  
 @Override  
 **public void** onAnimationUpdate(ValueAnimator animation) {  
 **text**.setText(animation.getAnimatedValue() + **""**);  
 }  
 });

1. [Animator.AnimatorListener](https://developer.android.com/reference/android/animation/Animator.AnimatorListener.html)

**objectAnimator**.addListener(**new** Animator.AnimatorListener() {  
 @Override  
 **public void** onAnimationStart(Animator animation) {  
 }  
  
 @Override  
 **public void** onAnimationEnd(Animator animation) {  
 }  
  
 @Override  
 **public void** onAnimationCancel(Animator animation) {  
 }  
  
 @Override  
 **public void** onAnimationRepeat(Animator animation) {  
 }  
});

从定义可以看到AnimatorListener监听了动画的开始、结束、取消、重复播放。但有时我们并不需要实现全部的方法，所以系统还提供了AnimatorListenerAdapter这个类，它是AnimatorListener的适配类，我们可以有选择的实现上面的4个方法。

* **时间插值器TimeInterpolator**

TimeInterpolator时间插值器，它的作用是根据时间流逝的百分比计算出当前属性值改变的百分比。

**valueAnimator**.setInterpolator(**new** TimeInterpolator() {  
 @Override  
 **public float** getInterpolation(**float** input) {  
 **return** 0;  
 }  
 });

TimeInterpolator实际上是一个接口，自定义插值器需要实现getInterpolation(input)方法：

input:指时间流逝的百分比，取值范围[0-1]。input=（当前时间-开始时间）/总时长

返回值：表示根据时间流逝计算出的动画进度，取值范围最好也是[0-1]，如果返回小于0或者大于1 的值，那么动画将跳出你所设置的范围，出现异常的效果。

系统也提供了很多插值器，例如线性插值器：LinearInterpolator，减速插值器：DecelerateInterpolator等。

另外贴上可视化插值器网站http://inloop.github.io/interpolator/

* **估值器TypeEvaluator**

TypeEvaluator估值器，它的作用是根据属性改变的百分比计算出属性的值。

**valueAnimator**.setEvaluator(**new** TypeEvaluator() {  
 @Override  
 **public** Object evaluate(**float** fraction, Object startValue, Object endValue) {  
 **return null**;  
 }  
 });

fraction：属性改变的百分比，也就是动画的进度。

startValue：开始值

endValue：结束值

返回值：开始值+动画完成百分比\*(结束值-开始值)

另外系统还预设置有估值器，FloatEvaluator，IntEvaluator，PointFEvaluator等等：

**public class** FloatEvaluator **implements** TypeEvaluator<Number> {**public** Float evaluate(**float** fraction, Number startValue, Number endValue) {  
 **float** startFloat = startValue.floatValue();  
 **return** startFloat + fraction \* (endValue.floatValue() - startFloat);  
 }  
}

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

**public class** PointFEvaluator **implements** TypeEvaluator<PointF> {  
**private** PointF **mPoint**;**public** PointFEvaluator(PointF reuse) {  
 **mPoint** = reuse;  
 }@Override  
 **public** PointF evaluate(**float** fraction, PointF startValue, PointF endValue) {  
 **float** x = startValue.**x** + (fraction \* (endValue.**x** - startValue.**x**));  
 **float** y = startValue.**y** + (fraction \* (endValue.**y** - startValue.**y**));  
 **if** (**mPoint** != **null**) {  
 **mPoint**.set(x, y);  
 **return mPoint**;  
 } **else** {  
 **return new** PointF(x, y);  
 }  
 }  
}

这样属性动画的大体流程就弄明白了，首先我们通过ofXXX（...）方法设置一个值的取值范围，然后通过TimeInterpolator计算动画进度的百分比，TypeEvaluator通过这个百分比计算出这个这个属性值具体是多少。如果想要获取这个具体指，则可以通过添加AnimatorUpdateListener来获取。

### **ViewPropertyAnimator**

它保存了动画过程中所需要操作的属性和对应的值。也就是说我们可以通过这个类同时操作一个对象的多个属性。

PropertyValuesHolder propertyY = PropertyValuesHolder.*ofFloat*(**"translationY"**, 0f, 500f);  
 PropertyValuesHolder propertyX = PropertyValuesHolder.*ofFloat*(**"translationX"**, 0f, 500f);  
 **objectAnimator** = ObjectAnimator.*ofPropertyValuesHolder*(**text**, propertyY,propertyX);

## **Keyframes**

关键帧，是一个时间/值对，它可以定义在一个特定时间上的特定状态，而且在两个关键帧之间可以定义不同的Interpolator，就好像多个动画的拼接，第一个动画结束点是第二个动画的起点，Keyframes是个抽象类，需要通过ofInt()，ofFloat()，ofObject来获取适当的Keyframes，最后使用PropertyValuesHolder.ofKeyframe获得PropertyValuesHolder对象：

Keyframe keyframe0 = Keyframe.*ofFloat*(0f, -300f);  
 Keyframe keyframe1 = Keyframe.*ofFloat*(0.4f, 300f);  
 Keyframe keyframe2 = Keyframe.*ofFloat*(0.8f, -300f);  
 Keyframe keyframe3 = Keyframe.*ofFloat*(1f, 300f);  
 PropertyValuesHolder propertyX = PropertyValuesHolder.*ofKeyframe*(**"translationX"**,  
 keyframe0, keyframe1, keyframe2, keyframe3);  
 **objectAnimator** = ObjectAnimator.*ofPropertyValuesHolder*(**text**, propertyX);  
 **objectAnimator**.setDuration(1000);

第一个参数为时间百分比，即从插值器中返回的值。

第二个参数为属性值。

上面代码意思：text开始0秒时起点为-300，

0—0.4秒text从-300横向运动到300，

0.4—0.8秒text从300横向运动到-300，

0.8—1秒（结束）text从-300横向运动到300。

当然我们可以添加插值器keyframe3.setInterpolator()，这里当keyframe3添加插值器后，他代表的是从keyframe2到keyframe3这个时间段内会使用这个插值器，即给当前帧添加的插值器会作用于上一帧到当前帧这个时间段。

### **ViewPropertyAnimator**

只作用于View，可以控制View的多个属性同时执行动画，在系统3.1时被加入：

**ViewPropertyAnimator**

myView.animate().x(50f).y(100f);

****Multiple ObjectAnimator objects****

ObjectAnimator animX = ObjectAnimator.ofFloat(myView, "x", 50f);  
ObjectAnimator animY = ObjectAnimator.ofFloat(myView, "y", 100f);  
AnimatorSet animSetXY = new AnimatorSet();  
animSetXY.playTogether(animX, animY);  
animSetXY.start();

**One ObjectAnimator**

PropertyValuesHolder pvhX = PropertyValuesHolder.ofFloat("x", 50f);  
PropertyValuesHolder pvhY = PropertyValuesHolder.ofFloat("y", 100f);  
ObjectAnimator.ofPropertyValuesHolder(myView, pvhX, pvyY).start();

为什么会有这种调用方式？下面是优点：

1. 由上可见大大的简化了代码量和可读性。
2. 在ViewProperTyAnimator会在每一个动画帧，计算变化的属性值并设置给属性，然后只调用一次

invalidate进行重绘，代替了每一个属性单独计算，单独重绘。性能大大提高。

注意：

1. ViewProperTyAnimator动画可以隐式启动，并不需要调用start()方法,这里的隐式启动并不是设置完了立刻启动。当然我们也可以使用start()方法显示启动，官方解释说start()方法启动挂起的动画。
2. 允许函数链式调用。

### **LayoutTransition**

## android:animateLayoutChanges属性：

在API 11之后，在布局文件中ViewGroup的派生容器中设置这个属性，那么在它添加或删除子View时，子View会拥有默认的动画过渡效果。

1. LayoutTransition

上面设置的属性只是默认的过渡效果，为了能够自定义ViewGroup的派生容器中的子View在显示、消失等变化时的动画效果，我们可以使用LayoutTransition这个类。当LayoutTransition中发生如下状态变化时我们可以定义子View的动画：

APPEARING - 子View添加到容器中时的过渡动画效果。

CHANGE\_APPEARING - 子View添加到容器中时，其他子View位置改变的过渡动画。

DISAPPEARING - 子View从容器中移除时的过渡动画效果。

CHANGE\_DISAPPEARING - 子View从容器中移除时，其它子view位置改变的过渡动画。

CHANGING - 子View在容器中位置改变时的过渡动画，不涉及删除或者添加操作。

**如下使用：**

**private int i** = 0;  
**private** LinearLayout **linearLayout**;  
**private** LayoutTransition **mTransitioner**;  
  
@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 **linearLayout** = (LinearLayout) findViewById(R.id.***linear***);  
 **mTransitioner** = **new** LayoutTransition();  
 **linearLayout**.setLayoutTransition(**mTransitioner**);  
 setTransition();  
}  
  
**private void** setTransition() {  
 */\*\*  
 \* 添加View时过渡动画效果  
 \*/* ObjectAnimator addAnimator = ObjectAnimator.*ofFloat*(**null**, **"rotationY"**, 0, 90, 0).  
 setDuration(**mTransitioner**.getDuration(LayoutTransition.***APPEARING***));  
 **mTransitioner**.setAnimator(LayoutTransition.***APPEARING***, addAnimator);  
  
 */\*\*  
 \* 移除View时过渡动画效果  
 \*/* ObjectAnimator removeAnimator = ObjectAnimator.*ofFloat*(**null**, **"rotationX"**, 0, -90, 0).  
 setDuration(**mTransitioner**.getDuration(LayoutTransition.***DISAPPEARING***));  
 **mTransitioner**.setAnimator(LayoutTransition.***DISAPPEARING***, removeAnimator);  
  
 */\*\*  
 \* view 动画改变时，布局中的每个子view动画的时间间隔  
 \*/* **mTransitioner**.setStagger(LayoutTransition.***CHANGE\_APPEARING***, 30);  
 **mTransitioner**.setStagger(LayoutTransition.***CHANGE\_DISAPPEARING***, 30);  
  
  
 */\*\*  
 \*LayoutTransition.CHANGE\_APPEARING和LayoutTransition.CHANGE\_DISAPPEARING的过渡动画效果  
 \* 必须使用PropertyValuesHolder所构造的动画才会有效果，不然无效！使用ObjectAnimator是行不通的,  
 \* 发现这点时真特么恶心,但没想到更恶心的在后面,在测试效果时发现在构造动画时，”left”、”top”、”bottom”、”right”属性的  
 \* 变动是必须设置的,至少设置两个,不然动画无效,问题是我们即使这些属性不想变动!!!也得设置!!!  
 \* 我就问您恶不恶心!,因为这里不想变动,所以设置为(0,0)  
 \*  
 \*/* PropertyValuesHolder pvhLeft = PropertyValuesHolder.*ofInt*(**"left"**, 0, 0);  
 PropertyValuesHolder pvhTop = PropertyValuesHolder.*ofInt*(**"top"**, 0, 0);  
 PropertyValuesHolder pvhRight = PropertyValuesHolder.*ofInt*(**"right"**, 0, 0);  
 PropertyValuesHolder pvhBottom = PropertyValuesHolder.*ofInt*(**"bottom"**, 0, 0);  
  
  
 */\*\*  
 \* view被添加时,其他子View的过渡动画效果  
 \*/* PropertyValuesHolder animator = PropertyValuesHolder.*ofFloat*(**"scaleX"**, 1, 1.5f, 1);  
 **final** ObjectAnimator changeIn = ObjectAnimator.*ofPropertyValuesHolder*(**this**, pvhLeft,  
 pvhBottom, animator).  
 setDuration(**mTransitioner**.getDuration(LayoutTransition.***CHANGE\_APPEARING***));  
  
 **mTransitioner**.setAnimator(LayoutTransition.***CHANGE\_APPEARING***, changeIn);  
  
  
 */\*\*  
 \* view移除时，其他子View的过渡动画  
 \*/* PropertyValuesHolder pvhRotation = PropertyValuesHolder.*ofFloat*(**"scaleX"**, 1, 1.5f, 1);  
 **final** ObjectAnimator changeOut = ObjectAnimator.*ofPropertyValuesHolder*(**this**, pvhLeft,  
 pvhBottom, pvhRotation).  
 setDuration(**mTransitioner**.getDuration(LayoutTransition.***CHANGE\_DISAPPEARING***));  
  
 **mTransitioner**.setAnimator(LayoutTransition.***CHANGE\_DISAPPEARING***, changeOut);  
}  
  
  
**public void** addView(View view) {  
 **i**++;  
 Button button = **new** Button(**this**);  
 button.setText(**"布局动画\_"** + **i**);  
 LinearLayout.LayoutParams params = **new** LinearLayout.LayoutParams(ViewGroup.LayoutParams  
 .***WRAP\_CONTENT***, ViewGroup.LayoutParams.***WRAP\_CONTENT***);  
 **linearLayout**.addView(button, Math.*min*(1, **linearLayout**.getChildCount()), params);  
}  
  
**public void** removeView(View view) {  
 **if** (**i** > 0)  
 **linearLayout**.removeViewAt(0);  
}

**关于监听器接口TransitionListener原型如下：**

/\*\*

\* This interface is used for listening to starting and ending events for transitions.

\*/**public** **interface** **TransitionListener** {

/\*\*

\* 监听LayoutTransition当前对应的transitionType类型动画开始

\* @param transition LayoutTransition对象实例

\* @param container LayoutTransition绑定的容器-container

\* @param view 当前在做动画的View对象

\* @param transitionType LayoutTransition类型，取值有：APPEARING、DISAPPEARING、

\* CHANGE\_APPEARING、CHANGE\_DISAPPEARING、CHANGING

\*/

**public** **void** **startTransition**(LayoutTransition transition, ViewGroup container,

View view, **int** transitionType);

/\*\*

\* 监听LayoutTransition当前对应的transitionType类型动画结束

\* @param transition LayoutTransition对象实例

\* @param container LayoutTransition绑定的容器-container

\* @param view 当前在做动画的View对象

\* @param transitionType LayoutTransition类型，取值有：APPEARING、DISAPPEARING、

\* CHANGE\_APPEARING、CHANGE\_DISAPPEARING、CHANGING

\*/

**public** **void** **endTransition**(LayoutTransition transition, ViewGroup container, View view, **int** transitionType);

}

* **View属性动画控制和状态**

参考：<https://www.cnblogs.com/ldq2016/p/6913159.html>

* 控制

属性动画的控制包括开始、停止和取消。在API 19中还可以控制动画的暂停和恢复。通过属性动画的控制，可以改变动画的状态。其控制的方法集合如下：

Animator.start()  ：动画开始播放。

Animator.end()    ：停止播放动画，只有再次调用start方法才会开始播放并且从开始状态播放。调用后动画快进执行

到最后一帧，即动画结束的状态并停止。

Animator.cancel() ：停止播放动画，只有再次调用start方法才会开始播放并且从开始状态播放。调用后动画只是停止

了它的时间轴，动画的的状态会停在一个中间态。

Animator.pause()    ：API 19中添加，暂停当前动画，并停留在中间状态。

Animator.resume()   ：API 19中添加，恢复播放pause()暂停的动画，是从中间状态继续播放。

* 状态

动画的状态判断包括以下三个方法：

boolean isStarted()：API 14。 已经调用了start函数并且还没播放完成也没有被取消掉，那么isStarted方法会返回true。就算是在动画播放延迟中，该方法依然会返回 true。

boolean isRunning()：只会在动画确实在播放并且还没停止的时候返回 true。延迟播放中返回false。

boolean isPaused()： API 19。 isPaused 返回了 true，那么说明当前动画是在暂停状态下即动画调用了pause()方法，但是此时isStarted()和isRunning()也是true。当动画调用了resume()方法，这三个判断状态都为true。