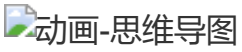
转载请注明链接： https://blog.csdn.net/feather_wch/article/details/81503233

涉及视图动画(补间动画、逐帧动画)、属性动画三种动画的使用方法，包括XML形式和代码形式，解析动画的底层原理和注意点。

Android 动画详解-思维导图版(82题)

版本:2018/08/17-1(7:19)



动画的分类(3题)

1、Android动画分为两种：

> 1. View动画(Animation)
> 2. 属性动画

2、View动画和属性动画的优缺点

> 1. View动画缺点显著：不具备交互性，只能做普通的动画效果
> 2. View动画优点：效率高、使用方便
> 3. 属性动画优点：具有交互性

3、View动画和属性动画区别

> 1. View动画并不支持对控件宽高做动画, 即使进行放大，本质控件的文字等也会被拉伸
> 2. 属性动画就可以给任意属性做动画

视图动画(21题)

1、View动画

> - 提供AlphaAnimation、RotateAnimation、TranslateAnimation、ScaleAnimation四种动画方式
> - 提供动画合集AnimationSet，用于动画混合
> - 缺点显著：不具备交互性，只能做普通的动画效果
> - 优点：效率高、使用方便

视图动画分类

2、视图动画分为两种

> 1. 补间动画
> 2. 逐帧动画

3、 View动画的使用

> 1. XML定义动画
> 2. 代码动态创建
> 3. XML形式的View动画，需要在 `res/anim/` 目录下创建XML文件 `custom.xml`

补间动画

4、什么是补间动画

> 1. 通过确定 开始和结束的视图样式 ， 中间动画变化过程由系统补全。

5、 补间动画的分类

分类	XML标签	效果
TranslateAnimation	translate	移动View
ScaleAnimation	scale	放大或缩小View
RotateAnimation	rotate	旋转View
AlphaAnimation	alpha	改变透明度

XML形式

6、 View动画如何通过XML定义？各属性要的作用?

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:shareInterpolator="true"
    android:duration="2000" //持续时间
    android:fillAfter="true"> //动画后是否停留在结束位置
    <alpha
        android:fillAfter="true"
        android:dutaion="1000"  //动画持续时间
        android:fromAlpha="0.1" //初始透明度，1为不透明，0为完全透明
        android:toAlpha="1"/>

    <scale
        android:fromXScale="0.5" //水平方向缩放，从0.5放大至1.2
        android:toXScale="1.2"
        android:fromYScale="1.1" //垂直方向缩放，从1.1缩小至0.3
        android:toYScale="0.3"
        android:pivotX="0.5"     //轴点(X,Y)
        android:pivotY="0.6"/>

    <translate
        android:fromXDelta="10" //x的初始值
        android:toXDelta="120"  //x的结束值
        android:fromYDelta="0"
        android:toYDelta="100"/>

    <rotate
        android:dutaion="1000"
        android:fromDegrees="0" //旋转开始的角度
        android:toDegrees="180" //旋转结束的角度
        android:pivotY="0"   //根据轴点进行旋转
        android:pivotX="0"/>

</set>
```

1. `android:duration` 表示持续时间，set有duration属性，内部动画的 `duration` 全部以set的为准
2. `android:fillAfter` 动画结束后，是否留在结束位置
3. `set标签` 没有duration时，内部的各种动画标签均以自身的 `duration` 为准
4. `scale` 中的 `(pivotX,pivotY)` 是以该点坐标为中心进行缩放。无论坐标超过View本身的范围。
5. `rotate` 中的 `(pivotX,pivotY)` 是旋转的中心坐标，以此点进行旋转。

7、如何使用XML定义的动画

```
val imageview = findViewById<ImageView>(R.id.imaview)
val animation = AnimationUtils.loadAnimation(this, R.anim.custom_animation)
imageview.startAnimation(animation)
```

代码形式

透明度动画

8、透明度动画

```
//透明度
AlphaAnimation alphaAnimation = new AlphaAnimation(0, 1);
alphaAnimation.setDuration(2000);
imageView1.setAnimation(alphaAnimation);
```

旋转动画

9、旋转动画

```
//旋转
RotateAnimation rotateAnimation = new RotateAnimation(0, 10);
rotateAnimation.setDuration(2000);
imageView2.setAnimation(rotateAnimation);
```

缩放动画

10、缩放动画

```
//缩放
ScaleAnimation scaleAnimation = new ScaleAnimation(0.5f, 1, 0.5f, 1);
scaleAnimation.setDuration(2000);
imageView3.setAnimation(scaleAnimation);
```

平移动画

11、平移动画

```
//平移
TranslateAnimation translateAnimation = new TranslateAnimation(0, 50, 0, 50);
translateAnimation.setDuration(2000);
imageView4.setAnimation(translateAnimation);
```

动画集合(AnimationSet)

12、动画集合

```
/*
*  动画集合，可以混合多种动画效果。
* */
AnimationSet animationSet = new AnimationSet(true);
animationSet.setDuration(3000);
animationSet.addAnimation(alphaAnimation);
animationSet.addAnimation(rotateAnimation);
imageView.setAnimation(animationSet);
```

逐帧动画

13、 逐帧动画是什么？注意点?

> 1. 逐帧动画与补间动画区别在于，需要定义一帧帧的动画内容。
> 2. 补间动画对应类是 `xxxAnimation` ， 而逐帧动画对应类是 `AnimationDrawable`
> 3. 帧动画采用的标签 `animation-list` ， 内部是 `item` 标签。
> 4. 帧动画有可能出现 `OOM` ,因此图片不能太大

自定义视图动画

14、如何自定义View动画

> 1. 自定义动画继承 `Animation`
> 2. 重写 `initialize` -做一些初始化操作
> 3. 重写 `applyTransformation` -进行一定的矩阵变换即可，通常通过 `Camera` 简化矩阵转换过程

特殊使用场景

Acitivity切换动画

15、Acitivity切换动画

> 1. 在Acitivty中调用 `overridePendingTransition`
> 2. `overridePendingTransition(R.anim.enter_anim, R.anim.exit_anim)` 第一个参数，为新Acitivty进入时动画。第二个参数，为旧acitivty退出时动画。
> 3. 必须紧挨着 `startActivity()` 或者 `finish()` 函数之后调用

Fragment切换动画

16、Fragment的切换动画

> 1. 通过 `FragmentTransaction` 的 `setCustomAnimations()` 方法设置
> 2. 必须是 `View` 动画

布局动画

17、什么是布局动画

> 1. 通过给ViewGroup设置布局动画，达到View逐渐呈现的过渡效果。
> 2. `android:animateLayoutChanges="true"` 者可以给布局添加系统默认的效果。
> 3. 如果要自定义过渡效果，需要通过 `LayoutAnimationController` 类来自定义，本质是给布局一个视图动画，在View出现时产生过渡效果。

代码形式

18、布局动画实例

```
// 1. 过渡动画
ScaleAnimation sa = new ScaleAnimation(0,1,0,1);
 sa.setDuration(1000);
// 2. 设置布局动画的显示属性（第一个参数，是需要作用的动画，而第二个参数，则是每个子View显示的delay时
LayoutAnimationController lac = new LayoutAnimationController(sa,0.5f);
// 3. 子View的显示顺序(delay > 0)
lac.setOrder(LayoutAnimationController.ORDER_NORMAL);
// 4. 为ViewGroup设置布局动画
LinearLayout mLinear  = (LinearLayout) findViewById(R.id.mLinear);
mLinear.setLayoutAnimation(lac);
```

19、布局动画的子View显示顺序

> 当delay的时间不为0时，可以设置子View显示的顺序：
>
> 1. LayoutAnimationController.ORDER_NORMAL——顺序
> 2. LayoutAnimationController.ORDER_RANDOM——随机
> 3. LayoutAnimationController.ORDER_REVERSE——反序

XML形式

20、XML形式的LayoutAnimation使用步骤

> 1-定义布局动画，使用 `layoutAnimation` 标签, 并引用item动画

```
//布局动画：res/anim/layout_animation
<?xml version="1.0" encoding="utf-8"?>
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
    android:delay="0.3"
    android:animationOrder="normal"
    android:animation="@anim/item_animation"/>
```

> 2-定义item动画(和一般View动画一样定义)

```
//item动画：res/anim/item_animation
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:shareInterpolator="true"
    android:duration="300"
    android:fillBefore="false">
    <alpha
        android:fromAlpha="0"
        android:toAlpha="1"/>
    <translate
        android:fromXDelta="500"
        android:toXDelta="0"/>
</set>
```

> 3. ViewGroup的对象使用 布局动画- `android:layoutAnimation="@anim/layout_animation"`

```
//ListView中使用
<ListView
    android:id="@+id/listview"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layoutAnimation="@anim/layout_animation">
</ListView>
```

21、XML形式LayoutAnimation要点

> 1. `android:delay="0.3"` 是指子元素动画延时开始的间隔。比如item动画时间为200ms，则每个子元素动画开始的间隔就是60ms.
> 2. `android:animationOrder="normal"` 动画的顺序：顺序、逆序和随机
> 3. `android:animation="@anim/item_animation"` 引用子元素所采用的动画

属性动画(47题)

1、属性动画是什么?

> 1. API11提出的新特性
> 2. 通过对 属性 操作能对任何对象做动画。
> 3. 支持更多的动画效果
> 4. 属性动画包括 `ObjectAnimator、AnimatorSet、ValueAnimator、Interpolator`
> 5. ObjectAnimator控制一个对象的一个属性
> 6. AnimatorSet是将多个ObjectAnimator组合并形成动画。
>
> 5，`ObjectAnimator` 继承自 `ValueAnimator`

2、属性值有哪些?

属性值	作用
translationX、translationY	控制View从左上角偏移的位置
rotation、rotationX、rotationY	控制View围绕支点做2D和3D旋转
scaleX、scaleY	围绕支点2D缩放
pivotX、pivotY	控制支点位置，默认为View中心
x、y	描述View的最终位置
alpha	透明度，默认1不透明，0为完全透明

ObjectAnimator

3、ObjectAnimator的使用步骤

> 1. 如果是自定义控件，需要添加 `setter / getter` 方法；
> 2. 用 `ObjectAnimator.ofXXX()` 创建 ObjectAnimator 对象；
> 3. 用 `start()` 方法执行动画。

```
public class SportsView extends View {

    float progress = 0;
    // 创建 getter 方法
    public float getProgress() {
        return progress;
    }
    // 创建 setter 方法
    public void setProgress(float progress) {
        this.progress = progress;
        invalidate();
    }
    @Override
    public void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        canvas.drawArc(arcRectF, 135, progress * 2.7f, false, paint);
    }
}
// 创建 ObjectAnimator 对象
ObjectAnimator animator = ObjectAnimator.ofFloat(view, "progress", 0, 65);
// 执行动画
animator.start();
```

实例：移动

4、属性动画实例：移动

```
//X轴平移一定距离
ObjectAnimator.ofFloat(imageView, "translationX", 100f).start()
```

实例：背景颜色变化

5、属性动画实例：背景颜色变化

```
val colorAnim = ObjectAnimator.ofInt(imageview, "backgroundColor", -0x7f80, -0x7f7f01)
colorAnim.setDuration(1000)
colorAnim.setEvaluator(ArgbEvaluator())
colorAnim.repeatCount = ValueAnimator.INFINITE
colorAnim.repeatMode = ValueAnimator.REVERSE
colorAnim.start()
```

ObjectAnimator的API

6、ObjectAnimator的通用方法

```
setDuration(int duration) //设置动画时长-单位是毫秒。

ObjectAnimator animator = ObjectAnimator.ofFloat(imageView2, "translationX", 500);
//1. 设置动画时长，单位毫秒。
animator.setDuration(2000);
//2. 设置插值器(动画的速度和表现形式)
animator.setInterpolator(new AccelerateDecelerateInterpolator()); //先加速，再减速。
animator.setInterpolator(new LinearInterpolator()); //匀速
animator.setInterpolator(new AccelerateInterpolator()); //加速
animator.setInterpolator(new DecelerateInterpolator()); //减速
animator.setInterpolator(new AnticipateInterpolator()); //先回拉再进行正常动画(如放大的会先缩小在
animator.setInterpolator(new OvershootInterpolator()); //会超过目标值，然后回到目标值。
animator.setInterpolator(new AnticipateOvershootInterpolator()); //先回拉，正常动画，会超过目标值
animator.setInterpolator(new BounceInterpolator()); //目标处弹动
animator.setInterpolator(new CycleInterpolator(0.5f)); //一个正弦/余弦曲线，可以自定义曲线的周期，
/**
 * 自定义动画完成度 / 时间完成度曲线。
 * 1. path-必须连续不能间断，也不能重叠
 * https://ws4.sinaimg.cn/large/006tKfTcly1fj8jmom7kaj30cd0ay74f.jpg
 * */
Path interpolatorPath = new Path();
// 先以「动画完成度 : 时间完成度 = 1 : 1」的速度匀速运行 25%
interpolatorPath.lineTo(0.25f, 0.25f);
// 然后瞬间跳跃到 150% 的动画完成度
interpolatorPath.moveTo(0.25f, 1.5f);
// 再匀速倒车，返回到目标点
interpolatorPath.lineTo(1, 1);
animator.setInterpolator(new PathInterpolator(interpolatorPath));

animator.setInterpolator(new FastOutLinearInInterpolator()); //加速运动(贝塞尔曲线)
animator.setInterpolator(new FastOutSlowInInterpolator());  //先加速再减速
animator.setInterpolator(new LinearOutSlowInInterpolator()); //持续减速
animator.start();
```

属性动画要点

7、属性动画要点

> 1. 属性动画要求该属性必须要有 `set/get` 方法
> 2. 插值器和估值器都可以自定义
> 3. 插值器自定义需要实现 `Interpolator` 或者 `TimeInterpolator`
> 4. 估值器自定义需要实现 `TypeEvaluator` 接口
> 5. `int/float/Color` 以外的类型必须要自定义 类型估值算法

8、属性动画想要生效，必须满足两个条件

> 1. 该属性需要有 `set` 和 `get` 方法

2. `set` 方法所做出的属性改变必须能通过UI等改变反映出来(`Button` 的setWidth方法本质就不能改变空间的高度)

9、TextView/Button改变宽高的动画为什么不能生效?

1. TextView以及子类的确有 `getWidth/setWidth` 方法，满足条件1，不满足条件2
2. 源码中 `getWidth=mRight-mLeft` 的确是View的高度 `android:layout_width` ， 该条满足 条件1
3. 而 `setWidth` 设置的是TextView的最大宽度和最小宽度，对应着 `android:width` 属性，并不是设置View的宽度，因此不满足 条件2

10、官方针对属性动画生效的条件问题，提供三种解决办法

1. 有权限的情况下，给对象加上 `get/set` 方法————一般难以做到，因为无权给SDK内部实现添加方法
2. 使用 装饰者模式 包装原始对象，间接为其提供 `get/set` 方法
3. 采用 `ValueAnimator` ， 监听动画过程，自己实现属性的改变

11、装饰者模式获得 `get、set`

1-实现包装类

```
private class WrapperView{
    private View view;
    public WrapperView(View view){
        this.view = view;
    }
    public int getWidth(){
        return view.getLayoutParams().width;
    }
    public void setWidth(int width){
        view.getLayoutParams().width = width;
        view.requestLayout();
    }
}
```

2-使用包装类实现属性动画

```
WrapperView wrapperView = new WrapperView(imageView);
ObjectAnimator.ofInt(wrapperView, "width", 500).setDuration(2000).start();
```

XML形式

12、通过xml使用ObjectAnimator

```
// res/animator/scalex.xml(在X轴上缩放，从1.0>0.5)
<?xml version="1.0" encoding="utf-8"?>
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:propertyName="scaleX"
    android:valueFrom="1.0"
    android:valueTo="0.5"
    android:valueType="floatType">

</objectAnimator>
```

13、Java中使用该动画

```
        Animator animator = AnimatorInflater.loadAnimator(this, R.animator.scalex);
        animator.setTarget(imageView);
        animator.start();
```

监听器

14、ViewPropertyAnimator/ObjectAnimator设置AnimatorListener监听器

```
//1-设置方法
view.animate().setListener(xxx);
objectAnimator.addListener(xxx);
//2-监听器的回调
//监听全部步骤
new Animator.AnimatorListener() {
    @Override
    public void onAnimationStart(Animator animation) {
        //动画开始执行时调用
    }

    @Override
    public void onAnimationEnd(Animator animation) {
        //动画结束时调用
    }

    @Override
    public void onAnimationCancel(Animator animation) {
        //1-动画通过cancel取消时，调用
        //2-cancel()之后onAnimationEnd()依旧会调用
    }

    @Override
    public void onAnimationRepeat(Animator animation) {
        //ViewPropertyAnimator不支持重复，因此该方法无效
        //ObjectAnimator通过setRepeatMode()和setRepeatCount()或者repeat()重复执行时,会调用
    }
}

//选择性监听
objectAnimator.addListener(new AnimatorListenerAdapter() {
            @Override
            public void onAnimationEnd(Animator animation) {
                super.onAnimationEnd(animation);
            }
        });
```

15、ViewPropertyAnimator/ObjectAnimator设置AnimatorUpdateListener

```
//1-设置更新监听器
objectAnimator.addUpdateListener(xxx);
view.animate().setUpdateListener(xxx);
//2-回调方法
new ValueAnimator.AnimatorUpdateListener(){
    @Override
    public void onAnimationUpdate(ValueAnimator animation) {
        //1-当动画的属性更新时，就会调用
        //2-参数的ValueAnimator是ObjectAnimator的父类，也是ViewPropertyAnimator的内部实现
        //3-ValueAnimator具有很多方法(查看当前的动画完成度、当前属性等)
    }
}
```

16、ViewPropertyAnimator/ObjectAnimator设置AnimatorPauseListener

```
//1-设置暂停监听器
View.animate().setUpdateListener(xxx);
objectAnimator.addPauseListener(new Animator.AnimatorPauseListener() {
    @Override
    public void onAnimationPause(Animator animation) {

    }

    @Override
    public void onAnimationResume(Animator animation) {

    }
});
```

AnimatorSet

17、AnimatorSet的作用和使用

> 1. 将动画融合(类似PropertyValuesHolder)
> 2. 在此基础上还可以控制动画的顺序。

```
ObjectAnimator objectAnimator1 = ObjectAnimator.ofFloat(imageView,"translationY", 300);
ObjectAnimator objectAnimator2 = ObjectAnimator.ofFloat(imageView,"scaleX", 1f, 0, 1f);
AnimatorSet animatorSet = new AnimatorSet();

animatorSet.playTogether(objectAnimator1, objectAnimator2);
animatorSet.setDuration(2000);
animatorSet.start();
```

XML形式

18、XML中定义属性动画集

> 需要在res文件夹中创建animator文件夹, 并创建XML文件

```xml
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:ordering="together" //表明动画集合中子动画是同时播放还是顺序播放
    >

    <objectAnimator  //对应ObjectAnimator
        android:propertyName="translationX" //属性名称
        android:duration="1000"         //持续时间
        android:valueFrom="200"         //属性起始值
        android:valueTo="500"           //属性结束值
        android:startOffset="10"        //动画的延迟时间，动画开始后，需要多少ms才真正播放动画
        android:repeatCount="10"        //动画的重复次数，默认0，-1为无限循环
        android:repeatMode="restart"    //动画的重复模式
        android:valueType="intType"     //表示propertyName所指属性的类型，但当属性表示颜色时不需要
        />

    <animator //对应ValueAnimator
          //相比于objectAnimator缺少一个android:propertyName
        />

    <set> //对应set
        ...
    </set>
</set>
```

19、代码中使用XML中定义的属性动画(包括AnimatorSet)

```kotlin
val set = AnimatorInflater.loadAnimator(this, R.animator.animator) as AnimatorSet
set.setTarget(listView)
set.start()
```

PropertyValuesHolder

20、PropertyValueHolder的作用

> 1. `ViewPropertyAnimator` 中通过 链式调用 就可以 同时改变多个属性

```
PropertyValuesHolder holder1 = PropertyValuesHolder.ofFloat("scaleX", 1);
PropertyValuesHolder holder2 = PropertyValuesHolder.ofFloat("scaleY", 1);
PropertyValuesHolder holder3 = PropertyValuesHolder.ofFloat("alpha", 1);

ObjectAnimator animator = ObjectAnimator.ofPropertyValuesHolder(view, holder1, holder2, holder3
animator.start();
```

21、PropertyValuesHolder实现动画效果

> 类似于AnimationSet的作用，将多种效果共同作用于对象。

```
PropertyValuesHolder pvh1 = PropertyValuesHolder.ofFloat("translationY", 200);
PropertyValuesHolder pvh2 = PropertyValuesHolder.ofFloat("scaleX", 1f, 0, 1f);
PropertyValuesHolder pvh3 = PropertyValuesHolder.ofFloat("scaleY", 1f, 0, 1f);
ObjectAnimator.ofPropertyValuesHolder(imageView, pvh1, pvh2, pvh3).setDuration(1000).start();
```

22、ObjectAnimator.ofPropertyValuesHolder()解析

```
public static ObjectAnimator ofPropertyValuesHolder(Object target,
        PropertyValuesHolder... values) {
    //1. 本质是创建ObjectAnimator对象，并将`PropertyValuesHolder`存入
    ObjectAnimator anim = new ObjectAnimator();
    anim.setTarget(target);
    anim.setValues(values);
    return anim;
}
```

> 1. 本质是创建ObjectAnimator对象，并将 PropertyValuesHolder 存入
> 2. ObjectAnimator.start()方法最底层本质就是通过 PropertyValuesHolder 的 setupValue 调用 get 方法， setAnimatedValue方法 去 set 属性值

ofKeyframe

23、PropertyValuesHolders.ofKeyframe()

> 1. 把一个属性进行 拆分

```
// 1、在 0% 处开始
Keyframe keyframe1 = Keyframe.ofFloat(0, 0);
// 2、时间经过 50% 的时候，动画完成度 100%
Keyframe keyframe2 = Keyframe.ofFloat(0.5f, 100);
// 3、时间见过 100% 的时候，动画完成度倒退到 80%，即反弹 20%
Keyframe keyframe3 = Keyframe.ofFloat(1, 80);
// 4、合成
PropertyValuesHolder holder = PropertyValuesHolder.ofKeyframe("progress", keyframe1, keyframe2,
// 5、使用
ObjectAnimator animator = ObjectAnimator.ofPropertyValuesHolder(view, holder);
animator.start();
```

ViewPropertyAnimator

24、ViewPropertyAnimator的由来

> - 属性动画 提供的 ValueAnimator类和ObjectAnimator类 本质不是针对 View对象 而设计的，而是一种 对数值不断操作 的过程，但大部分情况下还是对 View 进行动画操作的。因此 Google官方 在 3.1 中推出了 ViewPropertyAnimator 。

25、ViewPropertyAnimator的特点?

1. 专门针对View对象动画而操作的类。
2. 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。
3. 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）。
4. 每个属性提供 两种类型方法 设置(直接设置和 By 的形式)。
5. 该类只能通过 View的animate() 获取其实例对象的引用

26、ViewPropertyAnimator和ObjectAnimator的区别

1. 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)
2. 多个属性动画，也就是n个ObjectAnimator就会进行n次属性变化，就有n次invalidate。

27、ViewPropertyAnimator的使用方法

1-只能通过 view.animate()方法 实例化
2- view.animate().translationX(500);

3- 图中大部分方法 都具有 xxxBy() 版本，如 view.animate().translationXBy(10); 会在当前基础上 +10

28、View的animate()实现动画效果

1. 是属性动画的一种简写形式。

```
imageView.animate() //获得animator
        .alpha(0)
        .y(300)
        .setDuration(3000)
        .withStartAction(new Runnable() {
            @Override
            public void run() {
            }
        })
        .withEndAction(new Runnable() {
            @Override
            public void run() {
                //结束动作后，在UI线程操作
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                    }
                });
            }
        })
        .start(); //开始
```

29、 ViewPropertyAnimator.withStartAction/EndAction()

> 这两个方法是 ViewPropertyAnimator 的独有方法。

30、withStartAction/EndAction()和set/addListener()中onAnimationStart() / onAnimationEnd() 的区别

> 1. `withStartAction() / withEndAction()` 是一次性的，在动画执行结束后会自动弃掉，之后再重用 ViewPropertyAnimator 来做别的动画，用它们设置的回调也不会再被调用。
> 2. `set/addListener()` 所设置的 `AnimatorListener` 是 持续有效 的，当动画重复执行时，回调总会被调用。
> 3. `withEndAction()` 设置的回调只有在动画正常结束时才会被调用，而在动画被取消时不会被执行。这点和 AnimatorListener.onAnimationEnd() 的行为不一致。

ValueAnimator

31、ValueAnimator的作用

> 1. `ObjectAnimator` 的父类，
> 2. 提供数值变化和监听，本身不完成动画，通过得到的数值可以去进行一定变换。

32、ValueAnimator的使用: Int类型属性

> 1、Int属性(直接获取数值 or 通过插值器和估值器获得最终数据)

```
//起始值和终止值是0~100
ValueAnimatorvalueAnimator = ValueAnimator.ofInt(0, 100);
valueAnimator.addUpdateListener(
        new ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator animator) {

        //通过getAnimatedValue()来获取计算出的值
        //因为上面是ofInt，所以这里可以强转为Integer
        Integer animatedValue = (Integer) animator.getAnimatedValue();
        ((TextView) view).setText("$ " + animatedValue);
    }
});
valueAnimator.setDuration(3000);
valueAnimator.start();

//等同于
ValueAnimatorvalueAnimator = ValueAnimator.ofInt(0, 100);
valueAnimator.addUpdateListener(
        new ValueAnimator.AnimatorUpdateListener() {

            private IntEvaluator mEvaluator=new IntEvaluator();

            @Override
            public void onAnimationUpdate(ValueAnimator animator) {

                //先得到插值器返回的数据变化的百分比
                float animatedFraction =animator.getAnimatedFraction();

                //使用估值器，通过上面的 数据变化的百分比，得到改变后的数据
                Integer evaluate = mEvaluator.evaluate(animatedFraction, 0, 100);
                ((TextView) view).setText("$ " + evaluate);
            }
        });
valueAnimator.setDuration(3000);
valueAnimator.start();
```

33、ValueAnimator的使用: 颜色类型属性

```
//Argb传递的值也是int
ValueAnimatorvalueAnimator = ValueAnimator.ofArgb(/*RED*/0xFFFF8080, /*BLUE*/0xFF8080FF);
valueAnimator.addUpdateListener(
        new ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animator) {

                int animatedValue = (int) animator.getAnimatedValue();
                view.setBackgroundColor(animatedValue);

            }
        });
valueAnimator.setDuration(3000);
valueAnimator.start();

//等同于
ValueAnimatorvalueAnimator = ValueAnimator.ofArgb(/*RED*/0xFFFF8080, /*BLUE*/0xFF8080FF);
valueAnimator.addUpdateListener(
        new ValueAnimator.AnimatorUpdateListener() {

            private ArgbEvaluator mEvaluator=new ArgbEvaluator();

            @Override
            public void onAnimationUpdate(ValueAnimator animator) {
                float animatedFraction =animator.getAnimatedFraction();
                Object evaluate = mEvaluator.evaluate(animatedFraction, /*RED*/0xFFFF8080, /*BLU
                view.setBackgroundColor((int)evaluate);

            }
        });
valueAnimator.setDuration(3000);
valueAnimator.start();
```

xml形式

34、xml形式使用ValueAnimator

```
// res/animator/anim_test.xml
<?xml version="1.0" encoding="utf-8"?>
<animator xmlns:android="http://schemas.android.com/apk/res/android"
    android:valueType="intType"
    android:valueFrom="0"
    android:valueTo="100"
    android:repeatCount="1"
    android:repeatMode="restart"
    android:startOffset="1000"
    android:duration="3000">
</animator>
```

```
ValueAnimator valueAnimator= (ValueAnimator) AnimatorInflater.loadAnimator(this, R.animator.ani
valueAnimator.addUpdateListener(
        new ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animator) {
                int animatedValue = (int) animator.getAnimatedValue();
                ((TextView) view).setText("$ " + animatedValue);
            }
        });
valueAnimator.start();
```

监听器

35、监听器的使用

```
valueAnimator.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) {
    }
});
```

Interpolator

3、Interpolator作用

> 1. 插值器可以定义动画的变换速率（根据时间流逝的百分比来计算出当前属性值改变的百分比）
> 2. `TimeInterpolator`： 插值器，时间插值器 ， 用于决定 动画运动的变化曲线 ， 可以实现如加速、减速、弹性动画等效果。
> 3. 系统预置了：LinearInterpolator(匀速动画)、AccelerateDecelerateInterpolator(动画两头慢中间快)、DecelerateInterpolator(动画越来越慢)等等

37、自定义Interpolator

> 1、自定义一个弹性插值器

```
public class SpringInterpolator implements Interpolator{
    //弹性因数
    private float factor;

    public SpringInterpolator(float factor) {
        this.factor = factor;
    }

    @Override
    public float getInterpolation(float input) {
        return (float) (Math.pow(2, -10 * input) * Math.sin((input - factor / 4) * (2 * Math.PI
    }
}
```

> 2、使用

```
valueAnimator.setInterpolator(new SpringInterpolator(0.4f));
```

PathInterpolator

38、PathInterpolator的构造方法

方法	作用
PathInterpolator(Path path)	用Path创建Interpolator
PathInterpolator(float controlX1, float controlY1, float controlX2, float controlY2)	用三次贝塞尔曲线创建Interpolator。
PathInterpolator(float controlX, float controlY)	用二次贝塞尔曲线创建Interpolator。

39、二次贝塞尔曲线构造PathInterpolator

> 起点和终点是指定的(0f, 0f)和(1f, 1f), 唯一能指定的就是 控制点

```
PathInterpolator pathInterpolator
= new PathInterpolator(0.5f, 1f);
```

40、三次贝塞尔曲线构造PathInterpolator

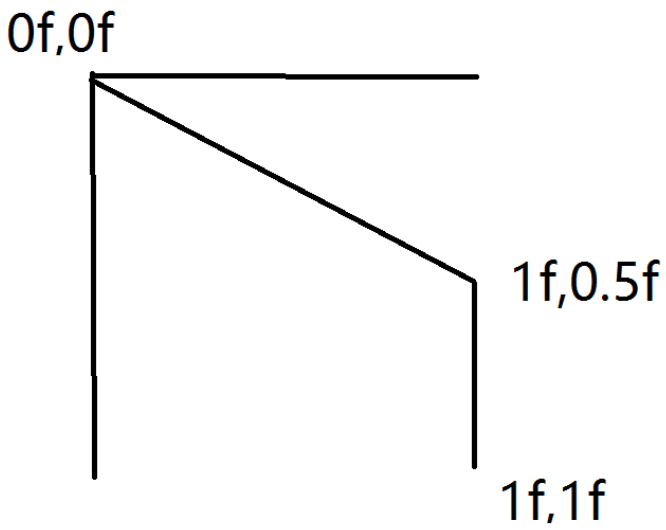
> 起点和终点是指定的(0f, 0f)和(1f, 1f), 能指定 两个控制点

```
PathInterpolator pathInterpolator = new PathInterpolator(0.5f, 1f, 2f, 2f);
```

41、Path构造PathInterpolator

1. 用 path 构造插值器，能通过 贝塞尔曲线 等方法进行构造。
2. 注意：禁止x值一定时，有两个y值

```
Path path = new Path();
path.moveTo(0f,0f);
path.lineTo(1f,0.5f);
path.lineTo(1f,1f);
```



TypeEvaluator

42、TypeEvaluator的作用

1. TypeEvaluator：类型估值算法，也称为估值器
2. 作用: 根据当前属性改变的百分比来计算改变后的属性值
3. 系统预置：IntEvaluator(针对整型属性)、FloatEvaluator(针对浮点型)、ArgbEvaluator(针对Color属性)
4. TimeInterpolator和TypeEvaluator是实现非匀速动画的重要手段。

43、ArgbEvaluator颜色估值器

```
//1、颜色的渐变
animator = ObjectAnimator.ofInt(this, "color", 0xffff0000, 0xff00ff00);
animator.setEvaluator(new ArgbEvaluator());
//2、颜色渐变(API >= 21)
animator = ObjectAnimator.ofArgb(this, "color", 0xffff0000, 0xff00ff00);

animator.start();
```

44、TypeEvaluator使用实例：自定义点估值器

1、自定义估值器

```
public class PointEvaluator implements TypeEvaluator<Point>{
    @Override
    public Point evaluate(float fraction, Point startValue, Point endValue) {
        float x = startValue.x + fraction * (endValue.x - startValue.x);
        float y = startValue.y + fraction * (endValue.y - startValue.y);
        Point point = new Point((int)x, (int)y);
        return point;
    }
}
```

2、ValueAnimator的ofObject进行动画

```
ValueAnimatorvalueAnimator=ValueAnimator.ofObject(new PointEvaluator(), new Point(0, 0), new Po
valueAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator animation) {
       Point point = (Point) animation.getAnimatedValue();
       xxx获得到变化后的点xxx
    }
});

valueAnimator.setDuration(3000);
valueAnimator.start();
```

45、TypeEvaluator使用实例：自定义颜色估值器

```
private class HsvEvaluator implements TypeEvaluator<Integer> {
    float[] startHsv = new float[3];
    float[] endHsv = new float[3];
    float[] outHsv = new float[3];

    @Override
    public Integer evaluate(float fraction, Integer startValue, Integer endValue) {
        // 把 ARGB 转换成 HSV
        Color.colorToHSV(startValue, startHsv);
        Color.colorToHSV(endValue, endHsv);

        // 计算当前动画完成度（fraction）所对应的颜色值
        if (endHsv[0] - startHsv[0] > 180) {
            endHsv[0] -= 360;
        } else if (endHsv[0] - startHsv[0] < -180) {
            endHsv[0] += 360;
        }
        outHsv[0] = startHsv[0] + (endHsv[0] - startHsv[0]) * fraction;
        if (outHsv[0] > 360) {
            outHsv[0] -= 360;
        } else if (outHsv[0] < 0) {
            outHsv[0] += 360;
        }
        outHsv[1] = startHsv[1] + (endHsv[1] - startHsv[1]) * fraction;
        outHsv[2] = startHsv[2] + (endHsv[2] - startHsv[2]) * fraction;

        // 计算当前动画完成度（fraction）所对应的透明度
        //右移动24位， ARGB 一共32位，每8位代表一个属性，依次代表透明度(alpha)、红色(red)、绿色(gre
        int alpha = startValue >> 24 + (int) ((endValue >> 24 - startValue >> 24) * fraction);

        // 把 HSV 转换回 ARGB 返回
        return Color.HSVToColor(alpha, outHsv);
    }
}

//使用一：
ObjectAnimator animator = ObjectAnimator.ofInt(view, "color", 0xffff0000, 0xff00ff00);
// 使用二：
ObjectAnimator animator = ObjectAnimator.ofObject(view, "color",
new HsvEvaluator(), 0xffff0000, 0xff00ff00);
// 使用自定义的 HslEvaluator
animator.setEvaluator(new HsvEvaluator());
animator.start();
```

46、ofObject()的使用

> 1. 属性动画 可以借助 自定义`TypeEvaluator` 通过 `ofObject()` 来对不限定类的属性做动画

```
//API21中已经实现，理解思路
private class PointFEvaluator implements TypeEvaluator<PointF> {
    PointF newPoint = new PointF();

    @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));

        newPoint.set(x, y);

        return newPoint;
    }
}

ObjectAnimator animator = ObjectAnimator.ofObject(view, "position",
        new PointFEvaluator(), new PointF(0, 0), new PointF(1, 1));
animator.start();
```

47、新API中新增了ofMultiInt()、ofMultiFloat()等方法

动画原理(8题)

View动画原理

1、View动画(Animation)的原理

> 1. Animation内部通过 `ViewRootImpl` 的 `scheduleTraversals` 来监听下一个屏幕刷新信号
> 2. 当接收到信号时，会从 `DecorView` 开始遍历 `View树` 并进行绘制
> 3. 绘制过程中顺带将 `View` 绑定的动画执行
> 4. 监听 下一个屏幕刷新信号 是通过 `Choreographer` 完成的，可以参考 屏幕刷新机制

属性动画原理

2、属性动画为什么需要get/set方法?

> 1. 属性动画通过传递给 `set` 的值不一样，并且越来越接近最终值，最终实现动画效果
> 2. 如果动画时没有传递初始值，则需要通过 `get` 方法获取属性的初始值
> 3. 如果初始值已经有了，则不需要 `get` 方法

3、ObjectAnimator的start()流程

> 1. `start()` 会先判断：若当前东、等待的动画和延迟的动画中有和当前动画相同的动画，就会取消相同的动画；最终调用父类 `ValueAnimator` 的 `start()`
> 2. `ValueAnimator` 中属性动画需要运行在Looper线程中；最终会调用 `AnimationHandler` 的start方法，此AnimationHandler并不是Handler，而是Runnable

3. 该Runnable中涉及JNI层的交互，最终是进入到ValueAnimatior的 `doAnimationFrame` 方法
4. `doAnimationFrame` 中最后调用 `animationFrame()` 方法，其内部调用 `animateValue()` 方法
5. `animateValue()` 中 `calculateValue()` 用于计算每帧动画所对应的属性值。
6. 初始化时，若属性初始值没有提供，则调用 `get` 方法： `PropertyValuesHolder` 中的 `setupValue` ,通过反射调用的 `get` 方法
7. 当动画下一帧动画到来时， `PropertyValuesHolder` 中的 `setAnimatedValue`方法 会将新的属性值设置给对象，通过反射调用其 `set` 方法

4、属性动画原理要点

1. 属性动画需要运行在Looper线程中
2. 初始化时，若没有提供属性初始值， `PropertyValuesHolder` 的 `setupValue` ,通过反射调用的 `get` 方法
3. 当动画下一帧动画到来时， `PropertyValuesHolder` 的 `setAnimatedValue`方法 会通过反射调用其 `set` 方法，设置新的属性值

ValueAnimator原理

动画注册

5、ValueAnimator的start中的注册流程

```
//ValueAnimator.java
public void start() {
    start(false);
}

//ValueAnimator.java
private void start(boolean playBackwards) {
    // 1. 一些变量的初始化
    mStarted = true;
    mStartedDelay = false;
    mPaused = false;
    /**============================*
     * 2. AnimationHandler
     *  1- 属性动画需要运行在Looper线程池中
     *  2- AnimationHandler本身是Runnable
     *============================*/
    AnimationHandler animationHandler = getOrCreateAnimationHandler();
    animationHandler.mPendingAnimations.add(this);
    if (mStartDelay == 0) {
        //xxx
    }
    // 3. 开始
    animationHandler.start();
}

//ValueAnimator.java的内部类：AnimationHandler的方法
public void start() {
    scheduleAnimation();
}

//ValueAnimator.java的内部类：AnimationHandler的方法
private void scheduleAnimation() {
    if (!mAnimationScheduled) {
        /**=========================
         * 1. 在下一帧刷新信号到来时，执行Runnable mAnimate
         *=========================*/
        mChoreographer.postCallback(Choreographer.CALLBACK_ANIMATION, mAnimate, null);
        mAnimationScheduled = true;
    }
}

//Choreographer.java
public void postCallback(int callbackType, Runnable action, Object token) {
    //在下一帧刷新信号到来时，调用Runnable
    postCallbackDelayed(callbackType, action, token, 0);
}
```

1. start() 方法最终会调用到 AnimationHandler 的 scheduleAnimation()
2. 通过 Choreographer 的 postCallback() 去注册 下一个刷新信号
3. 下一帧刷新信号 到达时， Choreographer 会调用注册的 Runnable

动画处理

6、接收到刷新信号后的动画处理过程

```
/**
 * //ValueAnimator.java的内部类：AnimationHandler中
 *  1. 被Choreographer调用
 */
final Runnable mAnimate = new Runnable() {
    @Override
    public void run() {
        mAnimationScheduled = false;
        doAnimationFrame(mChoreographer.getFrameTime());
    }
};

//ValueAnimator.java的内部类：AnimationHandler
void doAnimationFrame(long frameTime) {
    /**
     * 1、遍历待执行动画队列
     */
    for (int i = 0; i < mPendingAnimations.size(); ++i) {
        ValueAnimator anim = pendingCopy.get(i);
        // 2、无延迟动画直接开始，有延迟的动画添加到延迟队列
        if (anim.mStartDelay == 0) {
            /**
             * 1. 动画的初始化工作
             * 2. 添加到AnimationHandler的mAnimations队列中
             */
            anim.startAnimation(this);
        } else {
            // 加入到延迟队列中
            mDelayedAnims.add(anim);
        }
    }

    // 3、将 Delayed动画队列中的动画添加到 active的动画队列中
    for (int i = 0; i < mDelayedAnims.size(); ++i) {
        mReadyAnims.add(anim);
    }
    // 4、遍历active的动画队列，将其中的动画都执行startAnimation
    for (int i = 0; i < mReadyAnims.size(); ++i) {
        ValueAnimator anim = mReadyAnims.get(i);
        // 添加到AnimationHandler的mAnimations队列中
        anim.startAnimation(this);
    }

    /**
     *  5、临时列表进行存储
     */
    int numAnims = mAnimations.size();
    for (int i = 0; i < numAnims; ++i) {
        mTmpAnimations.add(mAnimations.get(i));
    }
    /**
     *  6、执行所有的active状态的动画
     *      1-doAnimationFrame 执行动画
     *      2-将执行完的动画加入到 mEndingAnims
```

```
     */

    for (int i = 0; i < numAnims; ++i) {
        ValueAnimator anim = mTmpAnimations.get(i);
        if (anim.doAnimationFrame(frameTime)) {
            mEndingAnims.add(anim);
        }
    }
    /**
     * 7、动画进行结束操作
     */
    for (int i = 0; i < mEndingAnims.size(); ++i) {
        mEndingAnims.get(i).endAnimation(this);
    }

    // 8、该帧画面的最后提交
    mChoreographer.postCallback(Choreographer.CALLBACK_COMMIT, mCommit, null);

    // 9、如果还有需要执行的动画和延迟动画，则监听下一帧刷新信号
    if (!mAnimations.isEmpty() || !mDelayedAnims.isEmpty()) {
        scheduleAnimation();
    }
}

//ValueAnimator.java-处理一帧的动画
final boolean doAnimationFrame(long frameTime) {
    // 1、处理第一帧动画的工作
    if (mSeekFraction < 0) {
        mStartTime = frameTime;
    } else {
        long seekTime = (long) (mDuration * mSeekFraction);
        mStartTime = frameTime - seekTime;
        mSeekFraction = -1;
    }
    //2、修正第一帧动画的时间
    final long currentTime = Math.max(frameTime, mStartTime);
    //3、计算出当前时间所对应的属性数值
    return animationFrame(currentTime);
}

//ValueAnimator.java
boolean animationFrame(long currentTime) {
    boolean done = false;
    //1、计算出fraction
    float fraction = mDuration > 0 ? (float)(currentTime - mStartTime) / mDuration : 1f;
    //xxx
    //2、通过插值器计算出最终的fraction和数值，并回调onAnimationUpdate
    animateValue(fraction);
    //3、返回是否处理完成
    return done;
}

//ValueAnimator.java-将动画从active动画列表、延迟列表、待执行列表中移除
protected void endAnimation(AnimationHandler handler) {
    handler.mAnimations.remove(this);
```

```
    handler.mPendingAnimations.remove(this);
    handler.mDelayedAnims.remove(this);
    //xxx
}
```

1. Choreographer 会调用 Runnable 中的方法
2. 会执行 AnimationHandler 的 doAnimationFrame()
3. 根据当前时间计算出 属性值
4. 将执行完的动画移除 动画队列
5. 如果 动画队列 中有未执行完的动画，通过 Choreographer 去注册下一帧刷新信号
6. 循环往复直至 动画队列 中所有动画都执行完毕。

7、第一帧动画的时间矫正

1. 在第一帧动画开始前，会进行 三大流程 ，假如耗时过多会导致前几帧动画的丢失。
2. 如果动画 还未开始 就丢失几帧画面是不合理的， 在doAnimationFrame 处理动画时，会对第一帧动画的开始时间进行校正。
3. 该工作只对 第一帧 有效，防止丢帧。但是如果 动画中途出现丢帧 是无法处理的。

8、ValueAnimator补充点

1. ValueAnimator 本身不涉及 UI操作 ，需要在 回调中进行UI变化

动画的要点总结(2)

1、动画使用的7个注意点

1. OOM：图片数量较多或者图片较大时容易出现OOM，且尽量避免帧动画
2. 内存泄露：属性动画中无限循环动画，需要在Acitivty退出后及时停止，否则会导致Activity无法释放。验证后发现View动画并不存在此问题。
3. 兼容性问题： 3.0以下系统上有兼容问题，需要适配
4. View动画的问题：View动画并不是真正改变View的状态，可能会动画之后View的setVisibility(GONE)失效，需要调用 view.clearAnimation() 清除View动画后才能解决
5. 不要使用px：要使用dp，px会导致不同设备上有不同效果
6. 动画元素的交互：3.0后，属性动画点击事件会跟随View而移动，View动画会停留在原位置
7. 硬件加速: 建议开启硬件加速，会提高动画的流畅性

2、复杂属性动画大致三种方法

1. 使用 PropertyValuesHolder 来对多个属性同时做动画；
2. 使用 AnimatorSet 来同时管理调配多个动画；
3. 使用 PropertyValuesHolder.ofKeyframe() 来把一个属性拆分成多段，执行更加精细的属性动画。

知识储备(1)

1、HSV是什么?

> HSV(Hue, Saturation, Value)是根据颜色的直观特性由A. R. Smith在1978年创建的一种颜色空间, 也称六角锥体模型(Hexcone Model)。这个模型中颜色的参数分别是：色调（H），饱和度（S），明度（V）。
>
> 1. 色调H
>   用角度度量，取值范围为0°～360°，从红色开始按逆时针方向计算，红色为0°，绿色为120°,蓝色为240°。它们的补色是：黄色为60°，青色为180°,品红为300°；
> 2. 饱和度S
>   饱和度S表示颜色接近光谱色的程度。一种颜色，可以看成是某种光谱色与白色混合的结果。其中光谱色所占的比例愈大，颜色接近光谱色的程度就愈高，颜色的饱和度也就愈高。饱和度高，颜色则深而艳。光谱色的白光成分为0，饱和度达到最高。通常取值范围为0%～100%，值越大，颜色越饱和。
> 3. 明度V
>   明度表示颜色明亮的程度，对于光源色，明度值与发光体的光亮度有关；对于物体色，此值和物体的透射比或反射比有关。通常取值范围为0%（黑）到100%（白）。

参考资料

1. 插值器网站
2. 三次贝塞尔曲线-效果查看
3. 弹性动画-3种实现方法
4. 动画3PropertyAnimator ValueAnimator
5. PropertyValuesHolders.ofKeyframe()详解
6. Android 动画：手把手教你使用 补间动画