链滴

Kotlin 进阶 | 动画代码太丑，用 DSL 动画库拯救

作者：lzlyy

原文链接：https://ld246.com/article/1594089709016

来源网站：链滴

Android 构建动画的代码语法啰嗦，可读性差。若能构建一套可读性更强的接口就能提高动画的发效率。本文尝试用 Kotlin 的 DSL 重写了整套构建动画的 API ，使得构建动画的代码量锐减，语义一目了然。另外，Android 提供了转动画的接口，但只有在 API level 26 以上才能使用，本文尝试突破这个限制。
<h2 id="原生动画代码">原生动画代码</h2>
假设需求如下：“缩放 textView 的同时平移 button ，然后拉长 imageView，动画结束后 toast 提示”。用系统原生接口构建如下：
<pre><code class="highlight-chroma">PropertyValuesHolder scaleX = PropertyValuesHolder.ofFloat("scaleX", 1.0f, 1.3f);
PropertyValuesHoder scaleY = PropertyValuesHolder.ofFloat("scaleY", 1.0f, 1.3f);
ObjectAnimator tAnimator = ObjectAnimator.ofPropertyValuesHolder(textView, scaleX, scaleY);
tvAnimator.setDuation(300);
tvAnimator.setInterpolator(new LinearInterpolator());

PropertyValuesHoder translationX = PropertyValuesHolder.ofFloat("translationX", 0f, 100f);
ObjectAnimator bnAnimator = ObjectAnimator.ofPropertyValuesHolder(button, translationX);
btnAnimator.setDration(300);
btnAnimator.setInterpolator(new LinearInterpolator());

ValueAnimator rightAnimator = ValueAnimator.ofInt(ivRight, screenWidth);
rightAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onnimationUpdate(ValueAnimator animation) {
        int right = ((it) animation.getAnimatedValue());
        imageView.seRight(right);
    }
});
rightAnimator.seturation(400);
rightAnimator.setnterpolator(new LinearInterpolator());

AnimatorSet animtorSet = new AnimatorSet();
animatorSet.play(vAnimator).with(btnAnimator);
animatorSet.play(vAnimator).before(rightAnimator);
animatorSet.addListener(new Animator.AnimatorListener() {

</code></pre>

```
</span></span><span class="highlight-line"><span class="highlight-cl">    @Override
</span></span><span class="highlight-line"><span class="highlight-cl">    public void on
nimationStart(Animator animation) {}
</span></span><span class="highlight-line"><span class="highlight-cl">    @Override
</span></span><span class="highlight-line"><span class="highlight-cl">    public void on
nimationEnd(Animator animation) {
</span></span><span class="highlight-line"><span class="highlight-cl">        Toast.makeT
xt(activity,"animation end" ,Toast.LENGTH_SHORT).show();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    @Override
</span></span><span class="highlight-line"><span class="highlight-cl">    public void on
nimationCancel(Animator animation) {}
</span></span><span class="highlight-line"><span class="highlight-cl">    @Override
</span></span><span class="highlight-line"><span class="highlight-cl">    public void on
nimationRepeat(Animator animation) {}
</span></span><span class="highlight-line"><span class="highlight-cl">});
</span></span><span class="highlight-line"><span class="highlight-cl">animatorSet.start()

</span></span></code></pre>
<p>啰嗦！而且乍一看不知道在做啥，只能一行一行的细看，待看完整段代码后，才能在脑海中构建整个需求的样子。</p>
<p>但逐行看也很费劲，不信就试着从第一行开始读：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">创建一个横向缩放属性
</span></span><span class="highlight-line"><span class="highlight-cl">创建一个纵向缩放
性
</span></span><span class="highlight-line"><span class="highlight-cl">创建一个动画，这
动画施加在 textView 上，并且包含缩放和透明度属性
</span></span><span class="highlight-line"><span class="highlight-cl">动画时长300毫秒
</span></span><span class="highlight-line"><span class="highlight-cl">动画使用线性插值

</span></span></code></pre>
<p>原生 API 将“缩放 textView ”这短短的一句话拆分成一个个零散的逻辑单元，并以一种不符合然语言的顺序排列，所以不得不读完所有单元，才能拼凑出整个语义。</p>
<p><em><strong>如果有一种更符合自然语言的 API，就能更省力地构建动画，更快速地理解代。</strong></em></p>
<h2 id="用-Kotlin-预定义扩展函数简化代码">用 Kotlin 预定义扩展函数简化代码</h2>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">AnimatorSet().apply {
</span></span><span class="highlight-line"><span class="highlight-cl">    ObjectAnimator
ofPropertyValuesHolder(
</span></span><span class="highlight-line"><span class="highlight-cl">        textView,
</span></span><span class="highlight-line"><span class="highlight-cl">        PropertyVa
uesHolder.ofFloat("scaleX", 1.0f, 1.3f),
</span></span><span class="highlight-line"><span class="highlight-cl">        PropertyVa
uesHolder.ofFloat("scaleY", 1.0f, 1.3f)
</span></span><span class="highlight-line"><span class="highlight-cl">    ).apply {
</span></span><span class="highlight-line"><span class="highlight-cl">        duration = 3
0L
</span></span><span class="highlight-line"><span class="highlight-cl">        interpolator
 LinearInterpolator()
</span></span><span class="highlight-line"><span class="highlight-cl">    }.let {
</span></span><span class="highlight-line"><span class="highlight-cl">        play(it).with(
</span></span><span class="highlight-line"><span class="highlight-cl">            ObjectA
```

```
imator.ofPropertyValuesHolder(
                but
on,
                Pr
pertyValuesHolder.ofFloat("translationX", 0f, 100f)
            ).apply {
                durat
on = 300L
                inter
olator = LinearInterpolator()
            }
        )
        play(it).befor
(
            ValueAn
mator.ofInt(ivRight,screenWidth).apply {
                addU
dateListener { animation -&gt; imageView.right= animation.animatedValue as Int }
                durat
on = 400L
                inter
olator = LinearInterpolator()
            }
        )
    }
    addListener(obj
ct : Animator.AnimatorListener {
        override fun
nAnimationRepeat(animation: Animator?) {}
        override fun
nAnimationEnd(animation: Animator?) {
            Toast.mak
Text(activity,"animation end",Toast.LENGTH_SHORT).show()
        }
        override fun
nAnimationCancel(animation: Animator?) {}
        override fun
nAnimationStart(animation: Animator?) {}
    })
    start()
}
```

使用 <code>apply()</code> 和 <code>let()</code> 避免了重复对象名，缩减了代码量。重要的是 Kotlin 的代码有一种结构，这种结构让代码更符合自然语言。试着读一下：

```
构建动画集，它包含{
    动画1
    将动画1和动画
一起播放
    将动画3在动画
之后播放
    。。。
}
```

虽然在语义上已经比较清晰，但结构还是显得啰嗦，此起彼伏的缩进看着有点乱。
<h2 id="用-DSL-进一步简化代码">用 DSL 进一步简化代码</h2>
如果使用自定义的 DSL，就可以做的更好！
直接上代码：
<pre><code class="highlight-chroma">animSet {
    objectAnim {
        target = textview
        scaleX = floaArrayOf(1.0f,1.3f)
        scaleY = scalX
        duration = 30L
        interpolator = LinearInterpolator()
    } with objectAnim {
        target = butn
        translationX = floatArrayOf(0f,100f)
        duration = 30
        interpolator = LinearInterpolator()
    } before anim {
        values = intArrayOf(ivRight,screenWidth)
        action = { value -> imageView.right = value as Int }
        duration = 40
        interpolator = LinearInterpolator()
    }
    onEnd = ToastmakeText(activity,"animation end",Toast.LENGTH_SHORT).show()
    start()
}
</code></pre>
一目了然的语义和清晰的结构，就好像是一篇英语文章。
这里运用了多个 Kotlin 语言特性，包括扩展函数、带接收者的 lambda、顶层函数、抽象属性属性访问器、中缀表示法、函数类型变量、apply()、also()、let()。
逐个讲解 Kotlin 语法知识点后，再分析整套 DSL 的实现方案。
<h2 id="带接收者的-lambda">带接收者的 lambda</h2>
代码中 <code>animSet()</code>、<code>objectAnim()</code>、<code>anim()</code> 都是带有一个参数的函数，这个参数是 <code>带接受者的 lambda</code>。<code>animSet()</code> 代码如下：
<pre><code class="highlight-chroma">fun animSet(creation: AnimSet.() -> Unit) = AnimSet().apply { creation() }.also { it.build() }
</code></pre>

它是一个顶层函数，定义在类体外，即它不隶属于任何类。这样定义的目是可以在任何地方调用 <code>animSet()</code> 来构造动画集。
它的参数类型是一个带接收者的 lambda <code>AnimSet.() -> Unit</code>，接收者是 <ode>AnimSet</code> 类，它表示动画集（类似 <code>AnimatorSet</code>）。这样定义的处是，可以在传入 <code>animSet()</code> 的 lambda 中访问 <code>AnimSet</code> 中的私有成员，若把构建单个动画的方法 <code>objectAnim()</code> 和 <code>anim()</code> 义在 <code>AnimSet()</code> 中，就可以像写 HTML 一样使用结构化的语法构建动画。所以参数 <code>creation</code> 描述的是在动画集中构建动画的过程。
<code>animSet()</code> 在函数体中，创建了一个动画集 <code>AnimSet</code> 实例并将构建子动画的方法应用在此实例上。
关于 <code>带接收者的lambda</code> 和 <code>apply()</code>、<code>also()</cod>、<code>let()</code> 更详细的讲解可以点击这里。
构建动画的方法定义如下：
<pre><code class="highlight-chroma">class AnimSet {
    //'构建ValueAnm'
    fun anim(animCeation: ValueAnim.() -> Unit): Anim = ValueAnim().apply(animCreation).also { anims.add(it)

    //'构建ObjectAnm'
    fun objectAnim animCreation: ObjectAnim.() -> Unit): Anim = ObjectAnim().apply(animCreation).also { it.sePropertyValueHolder() }.also { anims.add(it) }
}
</code></pre>
这两个函数和构建动画集的函数非常相似，都使用了 <code>带接收者的lambda</code> 作参数，它定义了如何构建动画。<code>ValueAnim</code> 和 <code>ObjectAnim</code> 分对应于原生的 <code>ValueAnimator</code> 和 <code>ObjectAnimator</code>。它们有一共同的基类 <code>Anim</code> 对应于原生的 <code>Animator</code>：
<pre><code class="highlight-chroma">abstract class Anim {
    //'原生动画实例'
    abstract var animator: ValueAnimator
    //'动画时长'
    var duration
        get() = 300L
        set(value) {
            animator.uration = value
        }
    //'插值器'
    var interpolator
        get() = LinearInterpolator() as Interpolator
        set(value) {
            animator.interpolator = value
        }

</code></pre>

```
</span></span><span class="highlight-line"><span class="highlight-cl">    //'动画与动画间的连机器'
</span></span><span class="highlight-line"><span class="highlight-cl">    var builder:AnimatorSet.Builder? = null
</span></span><span class="highlight-line"><span class="highlight-cl">    //'反转动画'
</span></span><span class="highlight-line"><span class="highlight-cl">    abstract fun reverseValues()
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h2 id="抽象属性">抽象属性</h2>
<p>动画基类 <code>Anim</code> 是抽象类，因为 <code>animator</code> 属性和 <code>reverseValues()</code> 方法是抽象的。</p>
<p><code>animator</code> 属性对于 <code>ValueAnim</code> 来说是 <code>ValueAnimator</code> 实例，对于 <code>ObjectAnim</code> 来说是 <code>ObjectAnimator</code> 实例：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">class ObjectAnim : Anim() {
</span></span><span class="highlight-line"><span class="highlight-cl">    override var animator: ValueAnimator = ObjectAnimator()
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">class ValueAnim : Anim() {
</span></span><span class="highlight-line"><span class="highlight-cl">    override var animator: ValueAnimator = ValueAnimator()
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>关于抽象属性更详细的介绍可以点击<a href="https://ld246.com/forward?goto=https%3A%F%2Fwww.lzlyy.top%2Farticles%2F2020%2F06%2F24%2F1592988757513.html" target="_blank" rel="nofollow ugc">这里</a></p>
<p>反转动画的算法对于 <code>ValueAnim</code> 和 <code>ObjectAnim</code> 有所不同将反转算法作为抽象函数放在基类的好处时，在动画集 <code>AnimSet</code> 中可以无需关心法细节而是直接调用 <code>reverseValues()</code> 实现反转动画：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">class AnimSet {
</span></span><span class="highlight-line"><span class="highlight-cl">    //'动画集中包的所有子动画'
</span></span><span class="highlight-line"><span class="highlight-cl">    private val anims by lazy { mutableListOf&lt;Anim&gt;() }
</span></span><span class="highlight-line"><span class="highlight-cl">    fun reverse() {
</span></span><span class="highlight-line"><span class="highlight-cl">        if (animatorSet.isRunning) return
</span></span><span class="highlight-line"><span class="highlight-cl">        //'遍历所有画并让其反转'
</span></span><span class="highlight-line"><span class="highlight-cl">        anims.takeIf { !isReverse }?.forEach { anim -&gt; anim.reverseValues() }
</span></span><span class="highlight-line"><span class="highlight-cl">        animatorSet.start()
</span></span><span class="highlight-line"><span class="highlight-cl">        isReverse = true
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>反转动画的算法会在下面分析，先来看下一个用到的 Kotlin 特性。</p>
```

<h2 id="属性访问器">属性访问器</h2>
<pre><code class="highlight-chroma">var duration
    get() = 300L
    set(value) {
        animator.duration = value
    }
</code></pre>
在类属性的下面实现 <code>set()</code> 和 <code>get()</code> 方法，这样的语法叫属性访问器。当定义了访问器的属性被赋值时，<code>set()</code> 函数会执行属性被读取时，<code>get()</code> 函数会执行，所以访问器定义了属性值的读写算法。
访问器在这里的好处是提供了默认值并隐藏了赋值细节，如果在构建动画时没有提供 duration 则默认为 300ms，为 <code>Anim</code> 实例设置 duration 时，其实就是调用了原生的 <code>ValueAnimator.setDuration()</code> 方法，属性访问器隐藏了这一细节，使得可以使用如下这简洁的语法构建动画：
<pre><code class="highlight-chroma">anim{
    values = intArrayOf(ivRight,screenWidth)
    action = { value -> imageView.right = value as Int }
    duration = 400 /'为动画设置时长'
    interpolator = LinearInterpolator()
}
</code></pre>
<h2 id="函数类型">函数类型</h2>
构建单个动画进行了 4 个属性赋值操作。其中 <code>action</code> 属性表示“如何将动画的序列应用到 View 上”：
<pre><code class="highlight-chroma">class ValueAnim : Anim() {
    override var animator: ValueAnimator = ValueAnimator()
    var action: ((Any -> Unit)? = null
        set(value) {
            field = value
            animator.addUpdateListener { valueAnimator ->
                valueAnimator.animatedValue.let { value?.invoke(it) }
            }
        }
}
</code></pre>
Kotlin 中可以将函数保存在一个变量中，这种变量的类型叫做 <code>函数类型</code>，<code>action</code> 的类型就是 <code>函数类型</code>，用 <code>((Any) -> Unit)?</code> 描述，意思是这个函数接收一个 <code>Any</code> 类型的参数但什么也不返回。
这个属性也用到了访问器，当 <code>action</code> 被赋值时就会为原生动画设置 <code>AnimatorUpdateListener</code>，并将属性值变化的序列作为参数传递给存放在 <code>action</code>

de> 中的 lambda，这样在构建动画时，就可以用一个简单的 lambda 定义做什么样的动画，比如下就是在做向右平移动画：
<pre><code class="highlight-chroma">anim{
    values = floatAr ayOf(0f,100f)
    action = { value -> imageView.translationX = value as Float }
    duration = 400
    interpolator = L nearInterpolator()
}
</code></pre>
其中的 <code>values</code> 属性表示动画值序列：
<pre><code class="highlight-chroma">class ValueAnim : Anim() {
    var values: Any? = null
        set(value) {
            field = val e
            value?.let {
                //'构建V lueAnimator对象'
                when (it) {
                    is Floa Array -> animator.setFloatValues(*it)
                    is Int rray -> animator.setIntValues(*it)
                    else > throw IllegalArgumentException("unsupported value type")
                }
            }
        }
}
</code></pre>
<code>values</code> 属性也使用了访问器，将根据类型调用 <code>ValueAnimator.setXX Value()</code> 细节隐藏。
<h2 id="中缀表示法">中缀表示法</h2>
Kotlin 中，当函数调用只有一个参数时，可以省略包括参数的 <code>()</code>，以让代码更洁，更符合自然语言，这种表示法叫中缀表示法。上述代码中用于连接多个动的 <code>before()</code> 函数就使用了中缀表示法：
<pre><code class="highlight-chroma">infix fun Anim.before(anim: Anim): Anim {
    animatorSet.pla (animator).before(anim.animator).let { this.builder = it }
    return anim
}
</code></pre>
中缀表示的方法必须以关键词 <code>infix</code> 开头，且函数只能有一个参数。同时这也一个 <code>Anim</code> 类的扩展函数。这个函数的调用者、参数、返回值都是一个 <code>An m</code> 实例。所以可以像 <code>a1 with a2 with a3</code> 这样将多个 <code>Anim</code> 连接起来。（连接动画的原理会在下面分析。）

<h2 id="实现方案">实现方案</h2>
将从“如何构建 Object 动画”、“如何反转动画”、“如何连接动画”这三个方面来分析整套 SL 的实现方法，关于 DSL 更详细的解释可以点击这里。
<h3 id="构建ObjectAnim">构建 ObjectAnim</h3>
整套 DSL 并不是实现一个全新的动画框架。而是将原生动画提供的接口通过 DSL 封装成结构化的 API 以减少代码量并增加可读性。
<code>ObjectAnim</code> 中定义了属性用于存放动画值序列：
<pre><code class="highlight-chroma">class ObjectAnim : Anim() {
    //'构建空ObjectAnimator对象'
    override var animator: ValueAnimator = ObjectAnimator()
    //'各个属性值列'
    var translationX: FloatArray? = null
    var translationY: FloatArray? = null
    var scaleX: FloaArray? = null
    var scaleY: FloaArray? = null
    var alpha: Floatrray? = null
    //'用数组存放空的属性值序列'
    private val valusHolder = mutableListOf<PropertyValuesHolder>()
</code></pre>
当调用如下代码时，属性被赋值：
<pre><code class="highlight-chroma">objectAnim {
    target = textView
    scaleX = floatArrayOf(1.0f,1.3f)
    scaleY = scaleX
    duration = 300L
    interpolator = LinearInterpolator()
}
</code></pre>
因为并不知道，每个动画会为哪些属性赋值，所以不能调用 <code>ObjectAnimator.ofPropertValuesHolder(textView, scaleX, scaleY);</code> 来构建 <code>ObjectAnimator</code> 对象而只能用一个数组存放所有被赋值的属性，并且通过遍历数组调用 <code>ObjectAnimator.setValus()</code> 异步构建 <code>ObjectAnimator</code> 对象：
<pre><code class="highlight-chroma">class AnimSet {
    fun objectAnimaction: ObjectAnim.() -> Unit): Anim = ObjectAnim().apply(action).also { it.setPropertyValuHolder() }.also { anims.add(it) }

```
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">class ObjectAnim : Anim() {
</span></span><span class="highlight-line"><span class="highlight-cl">    fun setPropertyalueHolder() {
</span></span><span class="highlight-line"><span class="highlight-cl">        //'遍历所有性序列，如果非空则构建PropertyValuesHolder并将其加入到集合中'
</span></span><span class="highlight-line"><span class="highlight-cl">        translationX?.et { PropertyValuesHolder.ofFloat(TRANSLATION_X, *it) }?.let { valuesHolder.add(it) }
</span></span><span class="highlight-line"><span class="highlight-cl">        translationY?.et { PropertyValuesHolder.ofFloat(TRANSLATION_Y, *it) }?.let { valuesHolder.add(it) }
</span></span><span class="highlight-line"><span class="highlight-cl">        scaleX?.let { PopertyValuesHolder.ofFloat(SCALE_X, *it) }?.let { valuesHolder.add(it) }
</span></span><span class="highlight-line"><span class="highlight-cl">        scaleY?.let { PopertyValuesHolder.ofFloat(SCALE_Y, *it) }?.let { valuesHolder.add(it) }
</span></span><span class="highlight-line"><span class="highlight-cl">        alpha?.let { PopertyValuesHolder.ofFloat(ALPHA, *it) }?.let { valuesHolder.add(it) }
</span></span><span class="highlight-line"><span class="highlight-cl">        animator.setalues(*valuesHolder.toTypedArray())
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h3 id="反转动画">反转动画</h3>
<p>反转动画的思路是：“将动画值序列倒序并重新播放动画”。动画基类 <code>AnimSet</cod> 中定义了反转算法的抽象方法：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlightcl">abstract class Anim {
</span></span><span class="highlight-line"><span class="highlight-cl">    abstract fun reverseValues()
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><code>ValueAnimator</code> 重写如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlightcl">class ValueAnim : Anim() {
</span></span><span class="highlight-line"><span class="highlight-cl">    override var animator: ValueAnimator = ValueAnimator()
</span></span><span class="highlight-line"><span class="highlight-cl">    //'属性值序列它是ValueAnim必须的属性'
</span></span><span class="highlight-line"><span class="highlight-cl">    var values: Any? = null
</span></span><span class="highlight-line"><span class="highlight-cl">        set(value) {
</span></span><span class="highlight-line"><span class="highlight-cl">            field = value
</span></span><span class="highlight-line"><span class="highlight-cl">            value?.let {
</span></span><span class="highlight-line"><span class="highlight-cl">                //'根据型将属性值序列设置给ValueAnimator'
</span></span><span class="highlight-line"><span class="highlight-cl">                when (it) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    is FloaArray -&amp;gt; animator.setFloatValues(*it)
</span></span><span class="highlight-line"><span class="highlight-cl">                    is Intrray -&amp;gt; animator.setIntValues(*it)
</span></span><span class="highlight-line"><span class="highlight-cl">                    else
```

```
&gt; throw IllegalArgumentException(’ unsupported value type’ )
            }
        }
    }

    override fun reverseValues() {
        values?.let {
            //'将属性序列原地翻转并重新应用到ValueAnimator上'
            when (it) {
                is FloatArray -&gt; {
                    it.reverse()
                    animator.setFloatValues(*it)
                }
                is IntArray -&gt; {
                    it.reverse()
                    animator.setIntValues(*it)
                }
                else -&gt; throw IllegalArgumentException("unsupported type of value")
            }
        }
    }
}
```

<code>AnimSet</code> 提供反转动画对的外接口：

```
class AnimSet {
    //'动画集所有动画'
    private val anim by lazy { mutableListOf&lt;Anim&gt;() }
    //'反转动画中有子动画'
    fun reverse() {
        if (animatorSt.isRunning) return
        //'逐个调用Anm.reverseValues()'
        anims.takeIf { !isReverse }?.forEach { anim -&gt; anim.reverseValues() }
        animatorSet.start()
        isReverse = true
    }
}
```

原文链接：Kotlin 进阶 | 动画代码太丑，用 DSL 动画库拯救

</code></pre>
<code>ObjectAnim</code> 的反转算法略有不同：
<pre><code class="highlight-chroma">class ObjectAnim : Anim() {
    //'属性序列'
    var translationX: FloatArray? = null
    var translationY: FloatArray? = null
    var scaleX: FloaArray? = null
    var scaleY: FloaArray? = null
    var alpha: Floatrray? = null
    //'属性序列集合'
    private val valusHolder = mutableListOf<PropertyValuesHolder>()
    //'遍历属性序集合并翻转对应属性序列'
    override fun reerseValues() {
        valuesHolder.orEach { valuesHolder ->
            when (valesHolder.propertyName) {
                TRANSLTION_X -> translationX?.let {
                    it.reverse()
                    valueHolder.setFloatValues(*it)
                }
                TRANSLTION_Y -> translationY?.let {
                    it.reverse()
                    valueHolder.setFloatValues(*it)
                }
                SCALE_X -> scaleX?.let {
                    it.reverse()
                    valueHolder.setFloatValues(*it)
                }
                SCALE_Y -> scaleY?.let {
                    it.reverse()
                    valueHolder.setFloatValues(*it)
                }

```
</span></span><span class="highlight-line"><span class="highlight-cl">            ALPHA
&gt; alpha?.let {
</span></span><span class="highlight-line"><span class="highlight-cl">                it.reve
se()
</span></span><span class="highlight-line"><span class="highlight-cl">                value
Holder.setFloatValues(*it)
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h3 id="连接动画">连接动画</h3>
<p>DSL 中的连接方案抛弃了 <code>AnimatorSet.playTogether()</code> 和 <code>playSequ
ntially()</code>，而是采用更加灵活的 <code>AnimtorSet.Builder</code> 方式。</p>
<p>被加入到 <code>AnimatorSet</code> 的 <code>Animator</code> 会被保存在 <code>
</code> 这个结构中：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public final class AnimatorSet extends Animator {
</span></span><span class="highlight-line"><span class="highlight-cl">    private static cl
ss Node implements Cloneable {
</span></span><span class="highlight-line"><span class="highlight-cl">        Animator m
nimation;
</span></span><span class="highlight-line"><span class="highlight-cl">        //孩子列表
</span></span><span class="highlight-line"><span class="highlight-cl">        ArrayList&lt;
ode&gt; mChildNodes = null;
</span></span><span class="highlight-line"><span class="highlight-cl">        //兄弟列表
</span></span><span class="highlight-line"><span class="highlight-cl">        ArrayList&lt;
ode&gt; mSiblings;
</span></span><span class="highlight-line"><span class="highlight-cl">        //父亲列表
</span></span><span class="highlight-line"><span class="highlight-cl">        ArrayList&lt;
ode&gt; mParents;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><code>Animator</code> 之间的播放顺序关系通过三个列表维护。兄弟列表中的动画会和自
同时播放，孩子列表会晚于自己播放，父亲列表会早于自己播放。</p>
<p>为了向这三个列表填值，系统定义了 <code>Builder</code> 类：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public final class AnimatorSet extends Animator {
</span></span><span class="highlight-line"><span class="highlight-cl">    public class Bui
der {
</span></span><span class="highlight-line"><span class="highlight-cl">        private Node
mCurrentNode;
</span></span><span class="highlight-line"><span class="highlight-cl">        //'为当前动
构建新结点'
</span></span><span class="highlight-line"><span class="highlight-cl">        Builder(Anim
tor anim) {
</span></span><span class="highlight-line"><span class="highlight-cl">            mDepend
ncyDirty = true;
</span></span><span class="highlight-line"><span class="highlight-cl">            mCurrent
ode = getNodeForAnimation(anim);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        //'向当前动
```

```
的兄弟列表中添加动画'
        public Builder with(Animator anim) {
            Node node = getNodeForAnimation(anim);
            mCurrentNode.addSibling(node);
            return this;
        }
        //'向当前动画的孩子列表中添加动画'
        public Builder before(Animator anim) {
            Node node = getNodeForAnimation(anim);
            mCurrentNode.addChild(node);
            return this;
        }
    }
    //'只能通过这方法构建Builder'
    public Builder play(Animator anim) {
        if (anim != null) {
            return new Builder(anim);
        }
        return null;
    }
}
```

同时播放 a1,a2,a3 动画，只需要这样调用 java API：

```
AnimatorSet set = new AnimatorSet();
set.play(a1).with(a).with(a3);
```

此时结点间只有一个层级，即 a1 在外层，a2 和 a3 存放在 a1 的兄弟列表中。 将上述 java 代转换成 Kotlin 的中缀表示法如下：

```
class AnimSet {
    private val animatorSet = AnimatorSet()

    infix fun Anim.with(anim: Anim): Anim {
        //'当前动画有Builder，则调用play()构建Builder，否则直接调用with()'
        if (builder == null) builder = animatorSet.play(animator).with(anim.animator)
        else builder?.with(anim.animator)
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">        return anim
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">abstract class Ani
{
</span></span><span class="highlight-line"><span class="highlight-cl">    //'动画对应的Bui
der'
</span></span><span class="highlight-line"><span class="highlight-cl">    var builder:Ani
atorSet.Builder? = null
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>因为同时播放的动画只有一个层级，所以调用链中，只需要第一个动画调用一次 <code>play()<
code> 即可。为 <code>Anim</code> 增加了 <code>builder</code> 属性以判断当前动画是
调用过 <code>play()</code> 来创建结点。</p>
<p>相比之下，顺序播放的代码层级就变多了，如果要先播放 a1，再播放 a2，最后播放 a3，java api
如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">AnimatorSet set = new AnimatorSet();
</span></span><span class="highlight-line"><span class="highlight-cl">set.play(a1).before
a2);
</span></span><span class="highlight-line"><span class="highlight-cl">set.play(a2).before
a3);
</span></span></code></pre>
<p>这个结构有点像树，后续结点是之前结点的孩子。对应的中缀表达式定义如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">class AnimSet {
</span></span><span class="highlight-line"><span class="highlight-cl">    infix fun Anim.
efore(anim: Anim): Anim {
</span></span><span class="highlight-line"><span class="highlight-cl">        animatorSet.
lay(animator).before(anim.animator).let { this.builder = it }
</span></span><span class="highlight-line"><span class="highlight-cl">        return anim
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>每次都为当前动画调用 <code>play()</code> 创建 <code>Builder</code> 并将后续动画
入孩子列表。</p>
<h2 id="推荐阅读">推荐阅读</h2>
<blockquote>
<ol>
<li><a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.lzlyy.top%2Farticles%
F2020%2F05%2F20%2F1589962952055.html" target="_blank" rel="nofollow ugc">Kotlin 基础
 白话文转文言文般的 Kotlin 常识</a></li>
<li><a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.lzlyy.top%2Farticles%
F2020%2F05%2F21%2F1590048659105.html" target="_blank" rel="nofollow ugc">Kotlin 基础
 望文生义的 Kotlin 集合操作</a></li>
<li><a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.lzlyy.top%2Farticles%
F2020%2F06%2F11%2F1591841223656.html" target="_blank" rel="nofollow ugc">Kotlin 实战
 用实战代码更深入地理解预定义扩展函数</a></li>
<li><a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.lzlyy.top%2Farticles%
F2020%2F06%2F24%2F1592988248748.html" target="_blank" rel="nofollow ugc">Kotlin 实战
 使用 DSL 构建结构化 API 去掉冗余的接口方法</a></li>
<li><a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.lzlyy.top%2Farticles%
F2020%2F06%2F24%2F1592988757513.html" target="_blank" rel="nofollow ugc">Kotlin 基础
```

抽象属性的应用场景
Kotlin 进阶 动画代码太丑，用 DSL 动画库拯救

</blockquote>