



HenCoder Android 自定义 View

1-6

属性动画 Property Animation (上手篇)



简介

前几期发布后，经常在回复里看到有人问我什么时候讲动画。本来我是不打算讲动画的，因为动画其实不算是自定义 View 的内容。但后来考虑了一下，动画在自定义 View 的开发中也起着很重要的作用，有的时候你对动画的了解不够，就难以实现一些自定义 View 的效果。

于是决定：加两期，讲动画！

不过并不是所有的动画都讲，我要讲的是属性动画。Android 里动画是有一些分类的：动画可以分为两类：Animation 和 Transition；其中 Animation 又可以再分为 View Animation 和 Property Animation 两类：View Animation 是纯粹基于 framework 的绘制转变，比较简单，如果你有兴趣的话可以上网搜一下它的用法；Property Animation，属性动画，这是在 Android 3.0 开始引入的新的动画形式，不过说它新只是相对的，它已经有好几年的历史了，而且现在的项目中的动画 99% 都是用的它，极少再用到 View Animation 了。属性动画不仅可以使使用自带的 API 来实现最常用的动画，而且通过自定义 View 的方式来做出定制化的动画。除了这两种 Animation，还有一类动画是 Transition。Transition 这个词的本意是转换，在 Android 里指的是切换界面时的动画效果，这个在逻辑上要复杂一点，不过它的重点是在于切换而不是动画，所以它也不是这次要讨论的内容。这次的内容只专注于一点：

Property Animation（属性动画）。在这一期我就基于前面几期讲过的自定义绘制，这一个自定义 View 的分支，来说一下属性动画的原理以及使用。

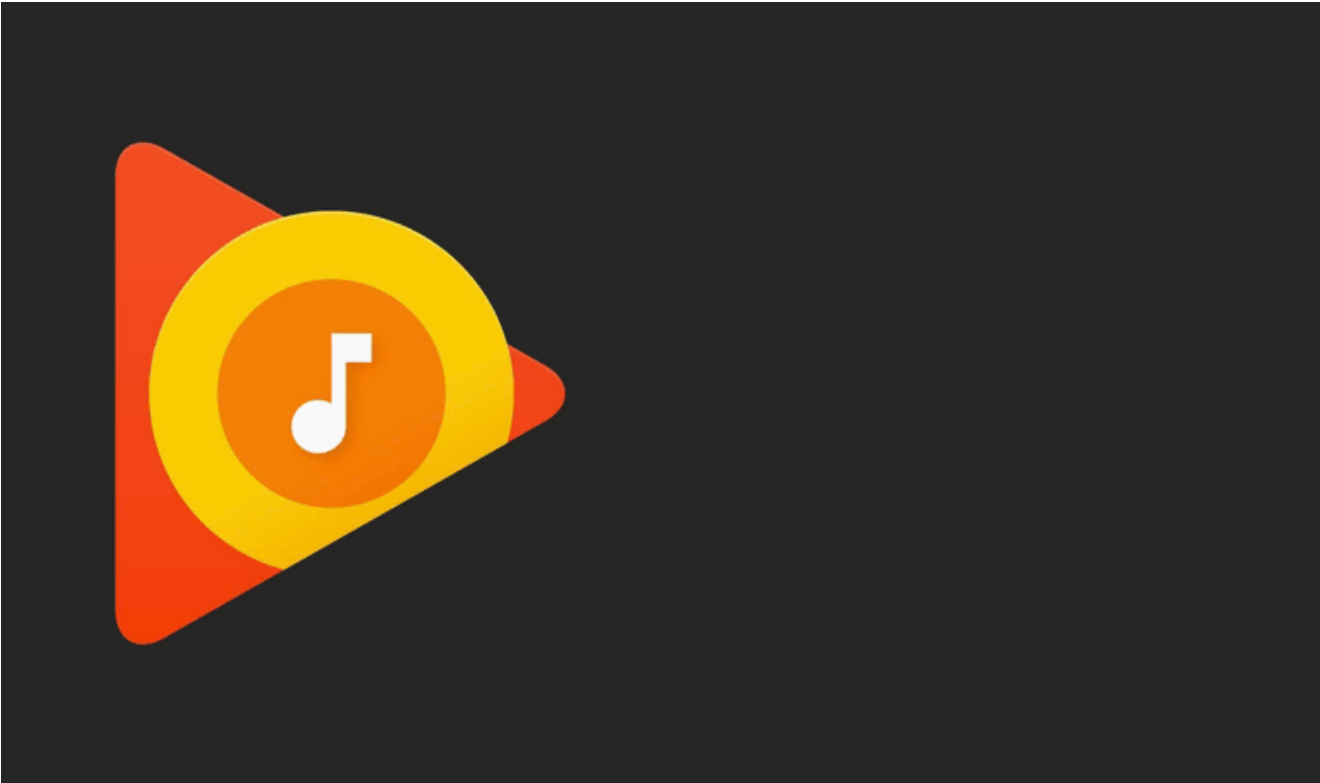
讲解

复杂的东西用文字很难讲清楚，所以每次遇到难讲的内容我都会选择上视频，这期也不例外。

ViewPropertyAnimator

使用方式：`View.animate()` 后跟 `translationX()` 等方法，动画会自动执行。

```
view.animate().translationX(500);
```



具体可以跟的方法以及方法所对应的 view 中的实际操作的方法如下图所示：

View 中的方法	功能	对应的 ViewPropertyAnimator 中的方法	
setTranslationX()	设置 x 轴偏移	translationX()	translationXBy()
setTranslationY()	设置 y 轴偏移	translationY()	translationYBy()
setTranslationZ()	设置 z 轴偏移	translationZ()	translationZBy()
setX()	设置 x 轴绝对位置	x()	xBy()
setY()	设置 y 轴绝对位置	y()	yBy()
setZ()	设置 z 轴绝对位置	z()	zBy()
setRotation()	设置平面旋转	rotation()	rotationBy()
setRotationX()	设置沿 x 轴旋转	rotationX()	rotationXBy()
setRotationY()	设置沿 y 轴旋转	rotationZ()	rotationZBy()
setScaleX()	设置横向放缩	scaleX()	scaleXBy()
setScaleY()	设置纵向放缩	scaleY()	scaleYBy()
setAlpha()	设置透明度	alpha()	alphaBy()

从图中可以看到，view 的每个方法都对应了 ViewPropertyAnimator 的两个方法，其中一个是带有 -By 后缀的，例如，View.setTranslationX() 对应了 ViewPropertyAnimator.translationX() 和 ViewPropertyAnimator.translationXBy() 这两个方法。其中带有 -By() 后缀的是增量版本的方法，例如，translationX(100) 表示用动画把 view 的 translationX 值渐变为 100，而 translationXBy(100) 则表示用动画把 view 的 translationX 值渐变地增加 100。

这些方法的效果都简单易懂，而且视频里也有简单的演示，所以就不放示例图了。如果你想看，可以去下面的练习项目。（最好顺便也练一下代码）

ObjectAnimator

使用方式：

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",
// 执行动画
animator.start();
```



0%

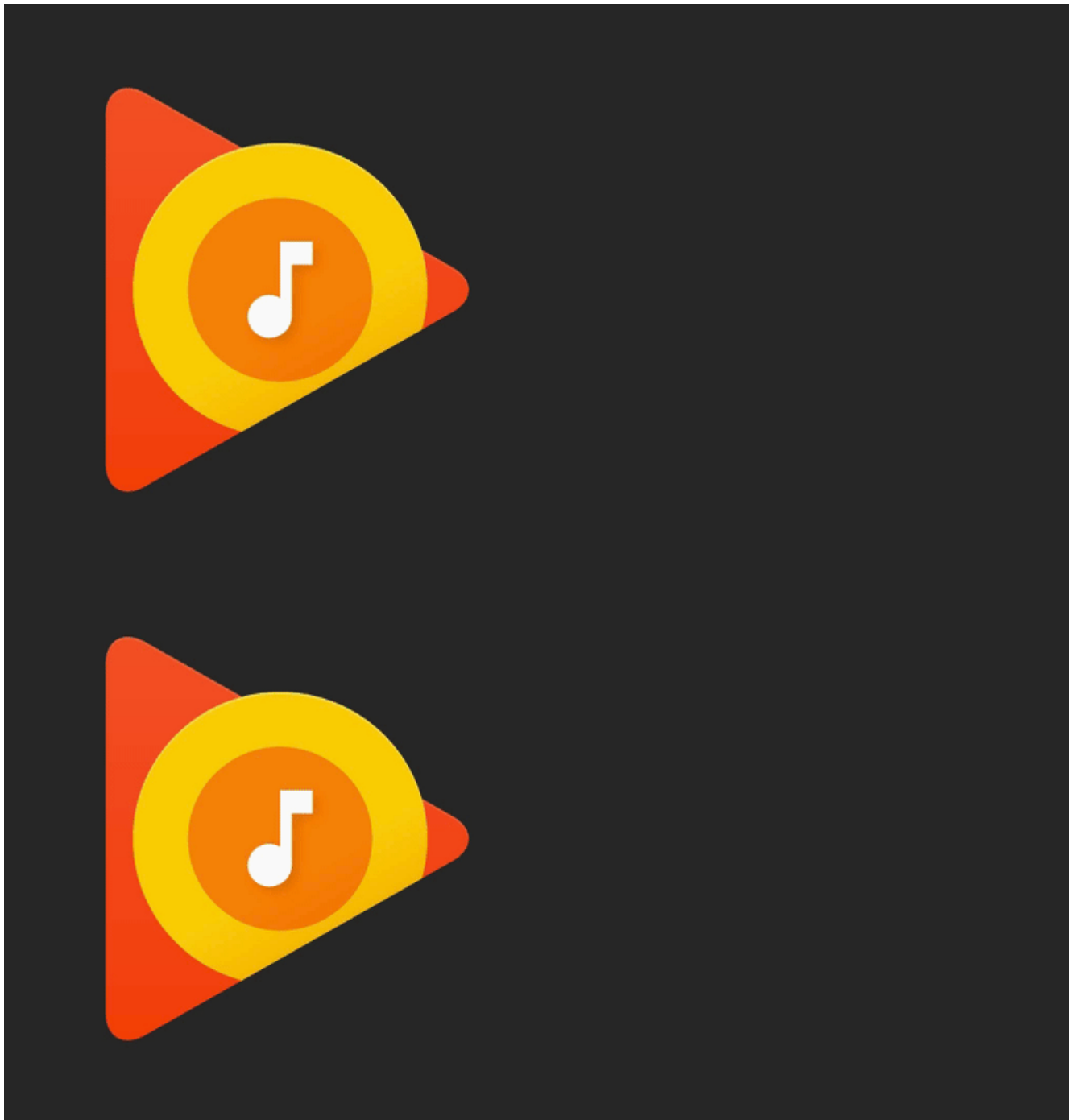
通用功能

1. `setDuration(int duration)` 设置动画时长

单位是毫秒。

```
// imageView1: 500 毫秒
imageView1.animate()
    .translationX(500)
    .setDuration(500);

// imageView2: 2 秒
ObjectAnimator animator = ObjectAnimator.ofFloat(
    imageView2, "translationX", 500);
animator.setDuration(2000);
animator.start();
```

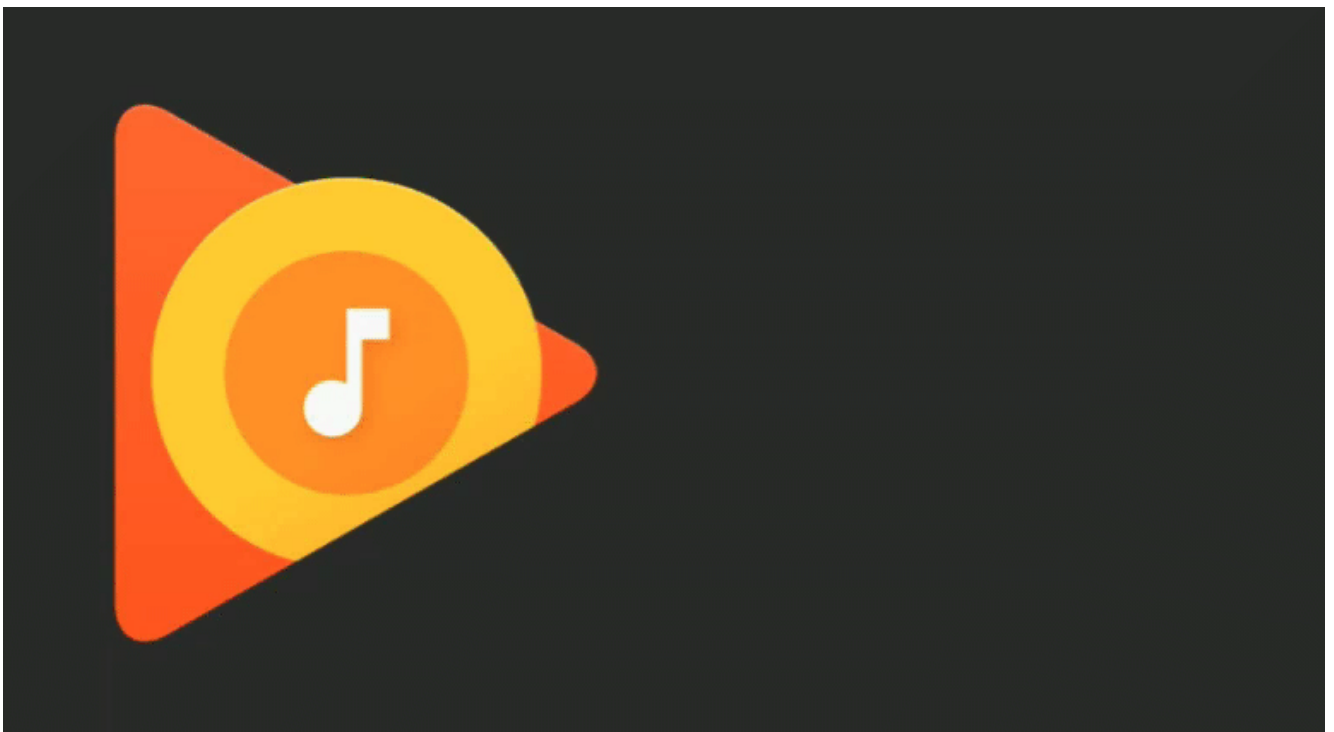
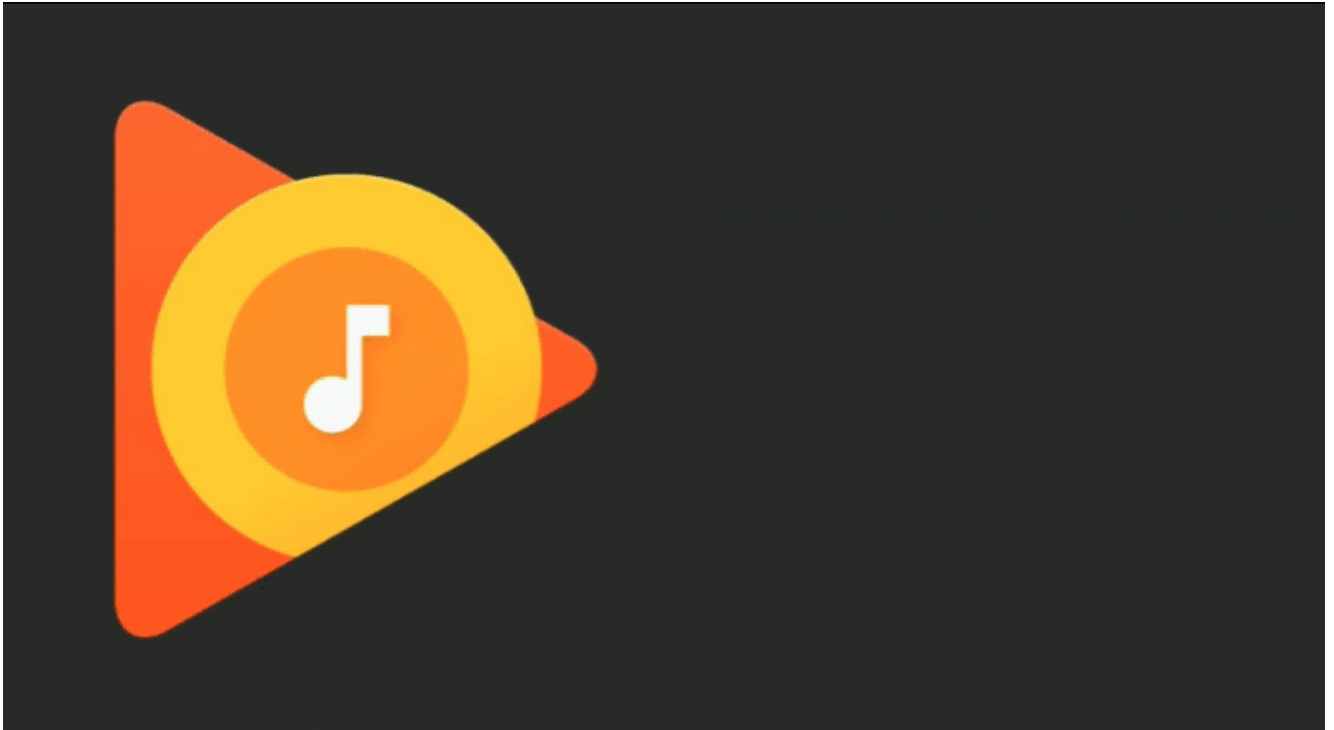


2. setInterpolator(Interpolator interpolator) 设置 Interpolator

视频里已经说了， `Interpolator` 其实就是速度设置器。你在参数里填入不同的 `Interpolator` ，动画就会以不同的速度模型来执行。

```
// imageView1: 线性 Interpolator, 匀速
imageView1.animate()
    .translationX(500)
    .setInterpolator(new LinearInterpolator());
```

```
// imageView: 带施法前摇和回弹的 Interpolator  
ObjectAnimator animator = ObjectAnimator.ofFloat(  
    imageView2, "translationX", 500);  
animator.setInterpolator(new AnticipateOvershootInterpolator());  
animator.start();
```

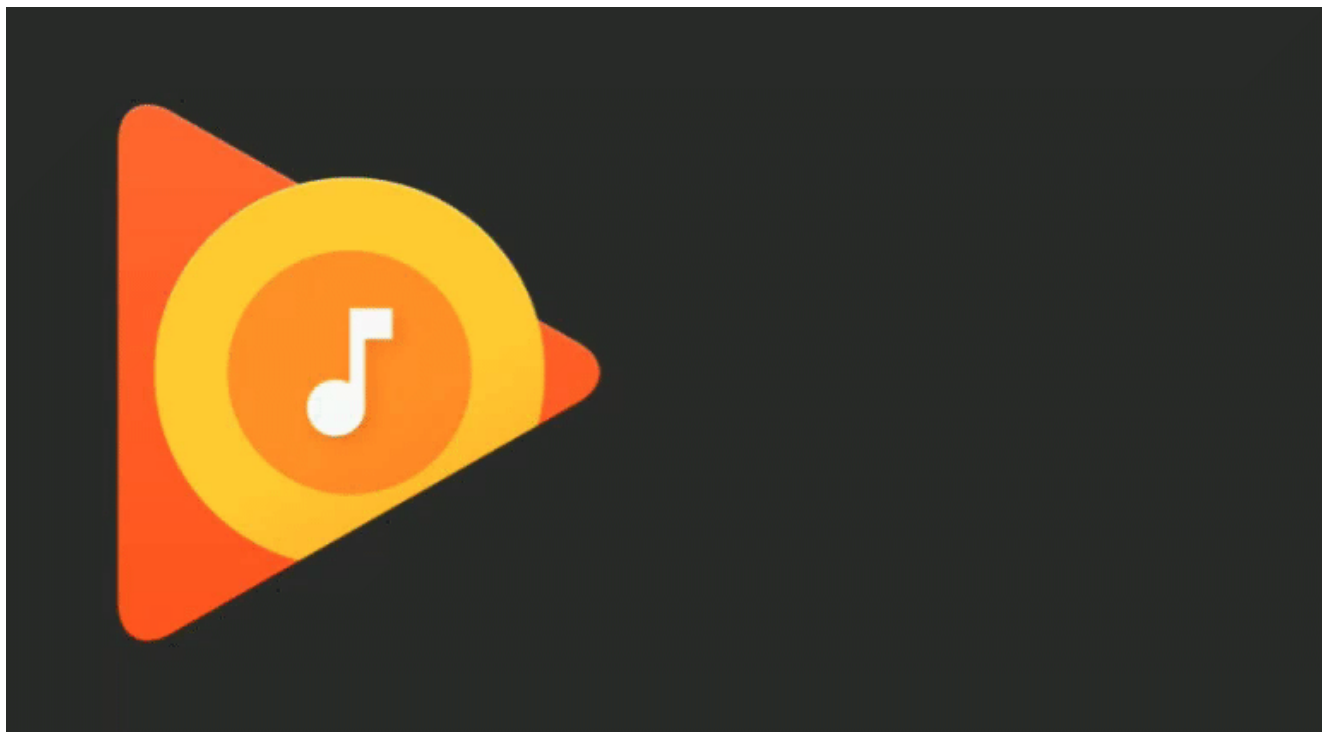


简单介绍一下每一个 Interpolator。

AccelerateDecelerateInterpolator

先加速再减速。这是默认的 `Interpolator`，也就是说如果你不设置的话，那么动画将会使用这个 `Interpolator`。

视频里已经说过了，这个是一种最符合现实中物体运动的 `Interpolator`，它的动画效果看起来就像是物体从速度为 0 开始逐渐加速，然后再逐渐减速直到 0 的运动。它的速度 / 时间曲线以及动画完成度 / 时间曲线都是一条正弦 / 余弦曲线（这句话看完就忘掉就行，没用）。具体的效果如下：



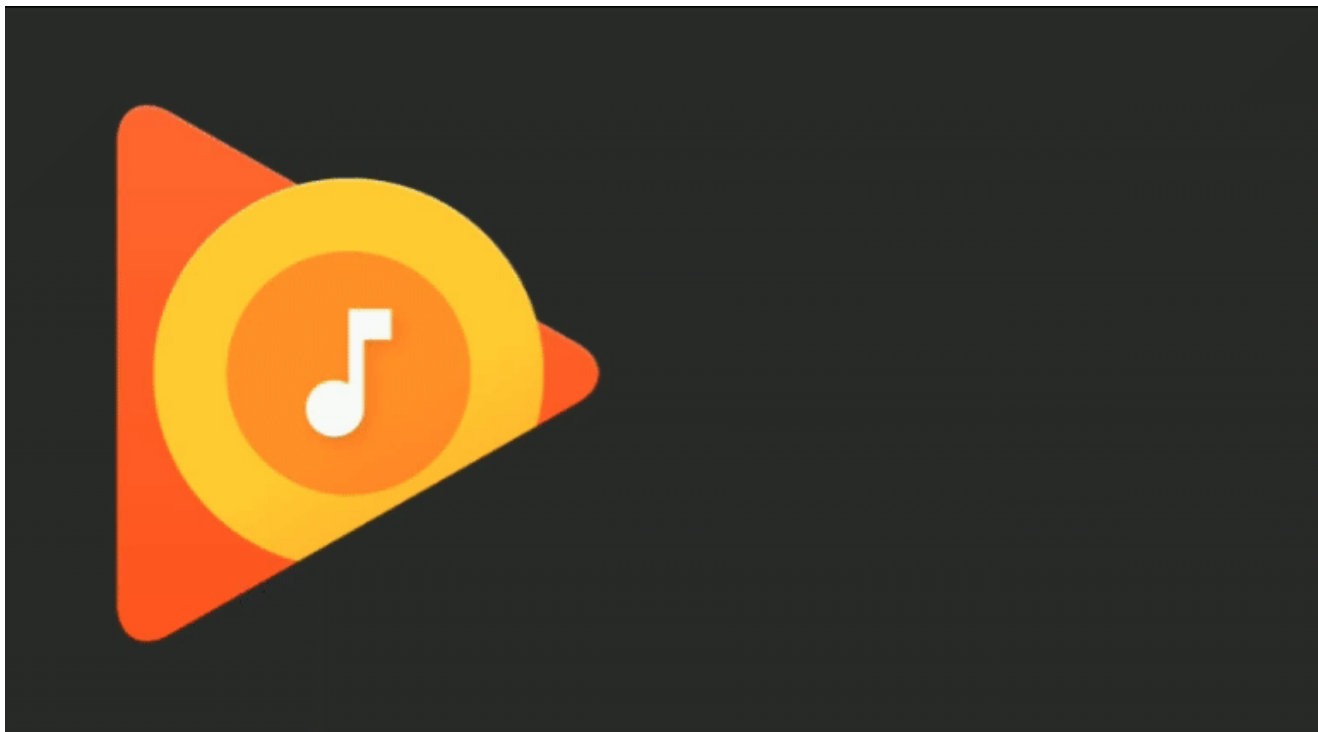
好像不太看得出来加速减速过程？你就将就着看吧，毕竟 gif 不是视频，要啥自行车啊。

用途：就像上面说的，它是一种最符合物理世界的模型，所以如果你要做的是最简单的状态变化（位移、放缩、旋转等等），那么一般不用设置 `Interpolator`，就用这个默认的最好。

LinearInterpolator

匀速。

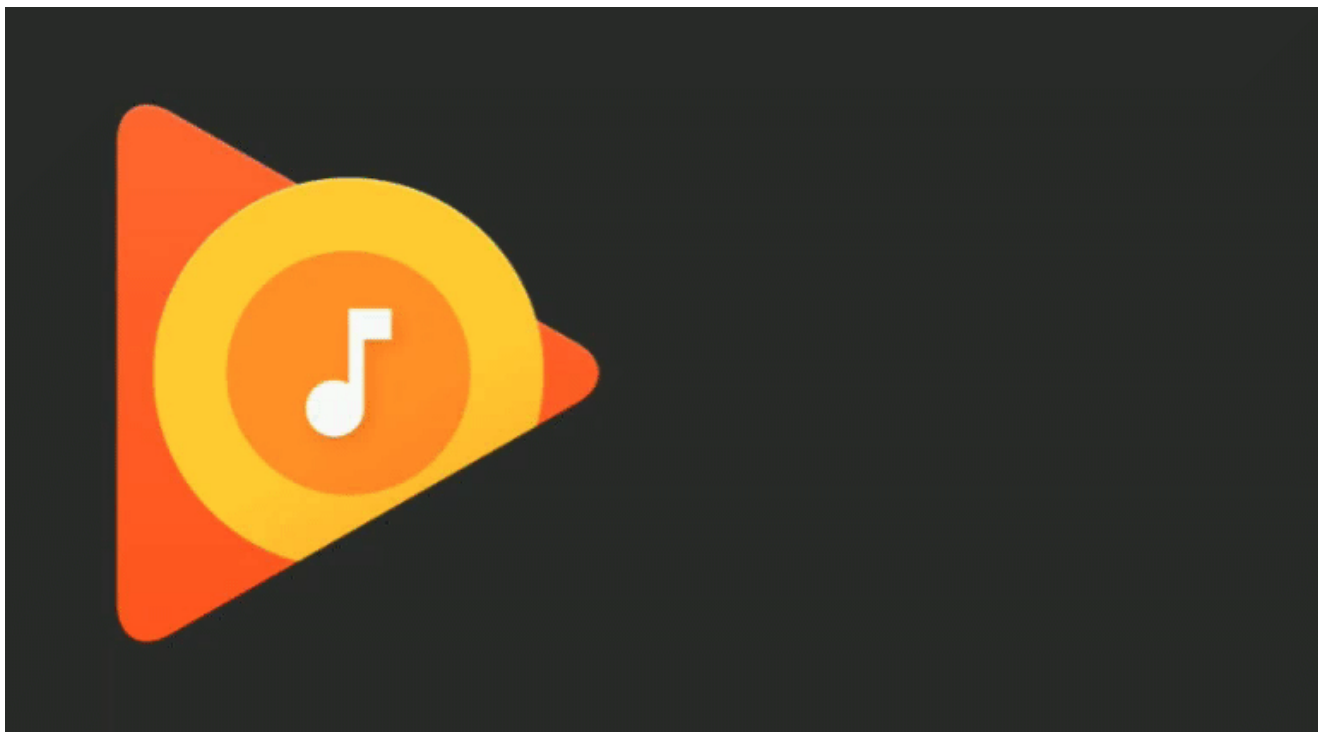
匀速就不用解释了吧？直接上效果：



AccelerateInterpolator

持续加速。

在整个动画过程中，一直在加速，直到动画结束的一瞬间，直接停止。

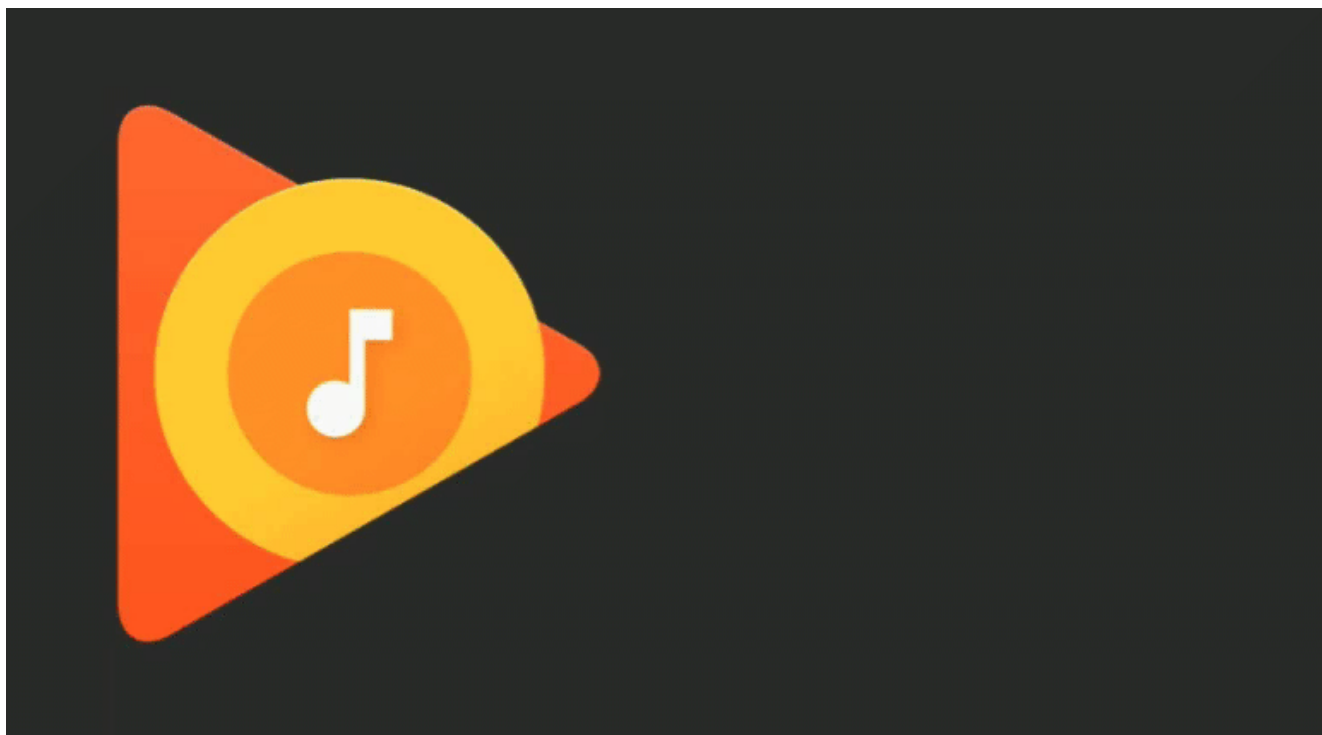


别看见它加速骤停就觉得这是个神经病模型哦，它很有用的。它主要用在离场效果中，比如某个物体从界面中飞离，就可以用这种效果。它给人的感觉就会是「这货从零起步，加速飞走了」。到了最后动画骤停的时候，物体已经飞出用户视野，看不到了，所以他们是并不会察觉到这个骤停的。

DecelerateInterpolator

持续减速直到 0。

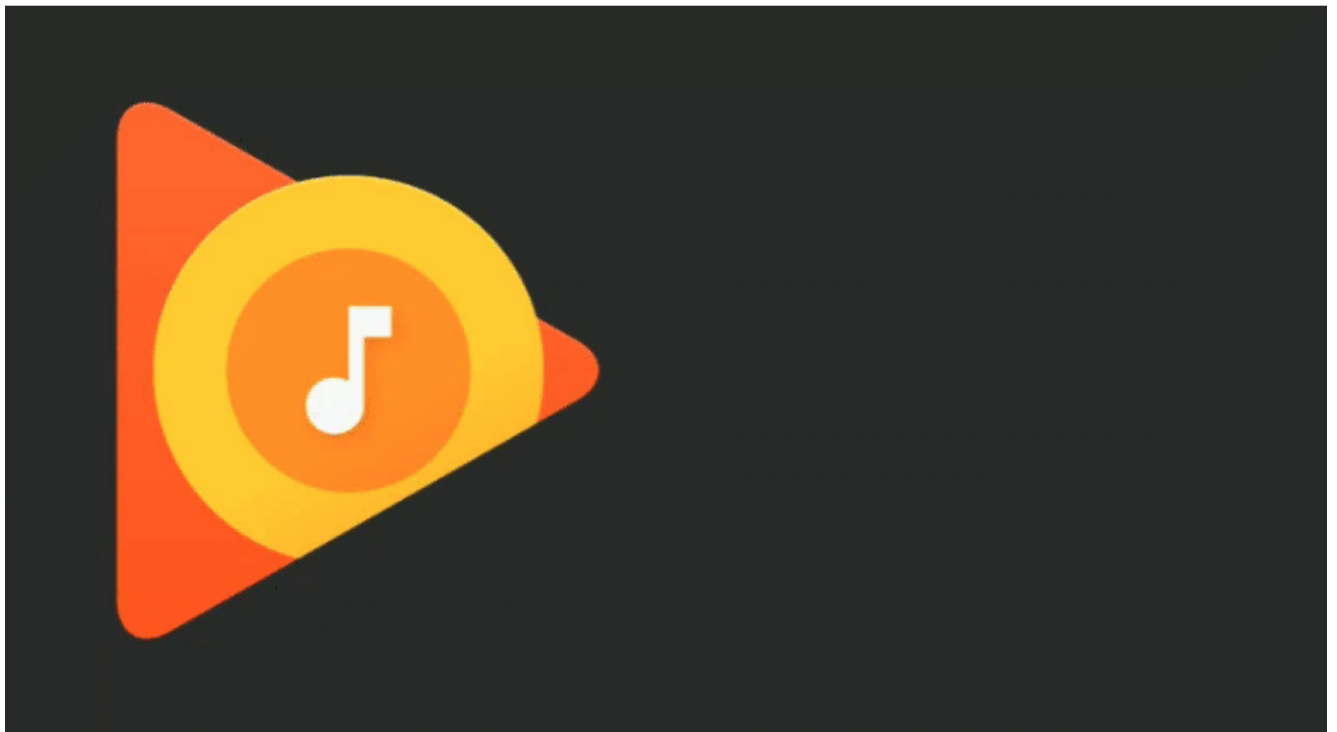
动画开始的时候是最高速度，然后在动画过程中逐渐减速，直到动画结束的时候恰好减速到 0。



它的效果和上面这个 `AccelerateInterpolator` 相反，适用场景也和它相反：它主要用于入场效果，比如某个物体从界面的外部飞入界面后停在某处。它给人的感觉会是「咦飞进来个东西，让我仔细看看，哦原来是 XXX」。

AnticipateInterpolator

先回拉一下再进行正常动画轨迹。效果看起来有点像投掷物体或跳跃等动作前的蓄力。

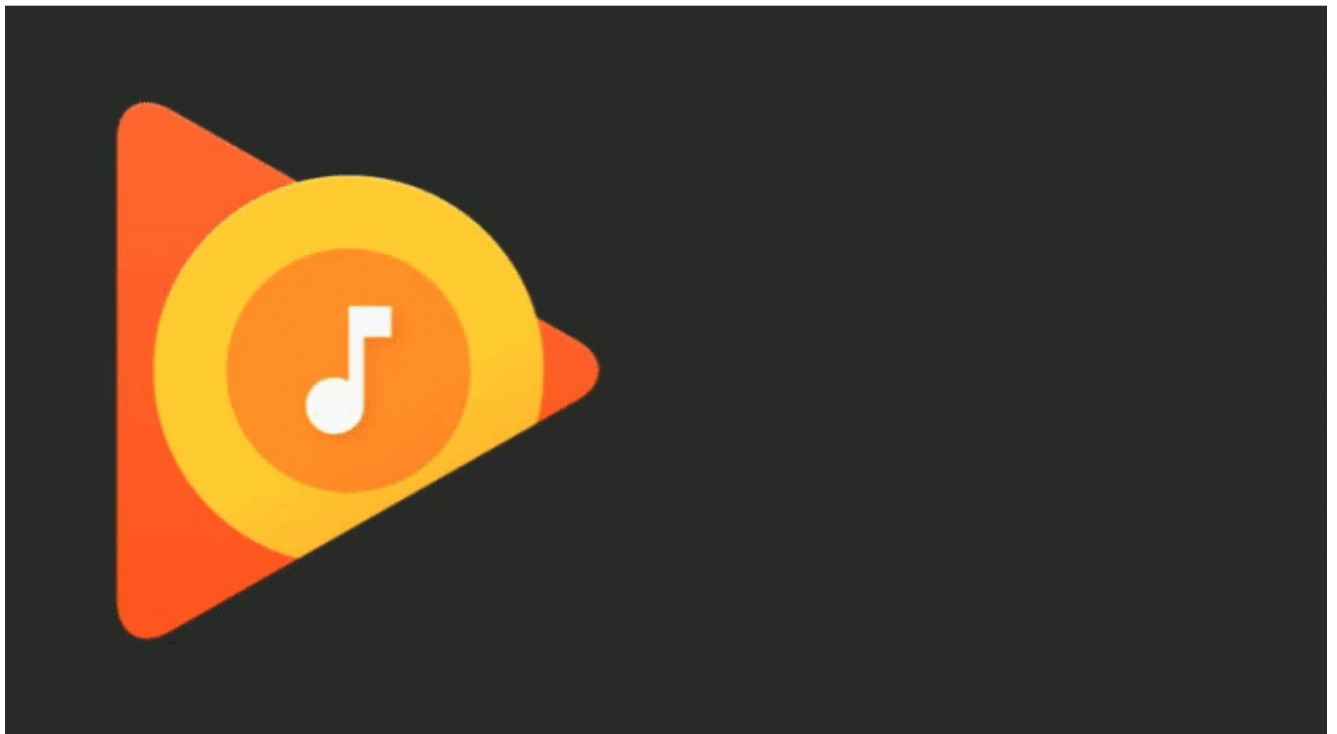


如果是图中这样的平移动画，那么就是位置上的回拉；如果是放大动画，那么就是先缩小一下再放大；其他类型的动画同理。

这个 `Interpolator` 就有点耍花样了。没有通用的适用场景，根据具体需求和设计师的偏好而定。

OvershootInterpolator

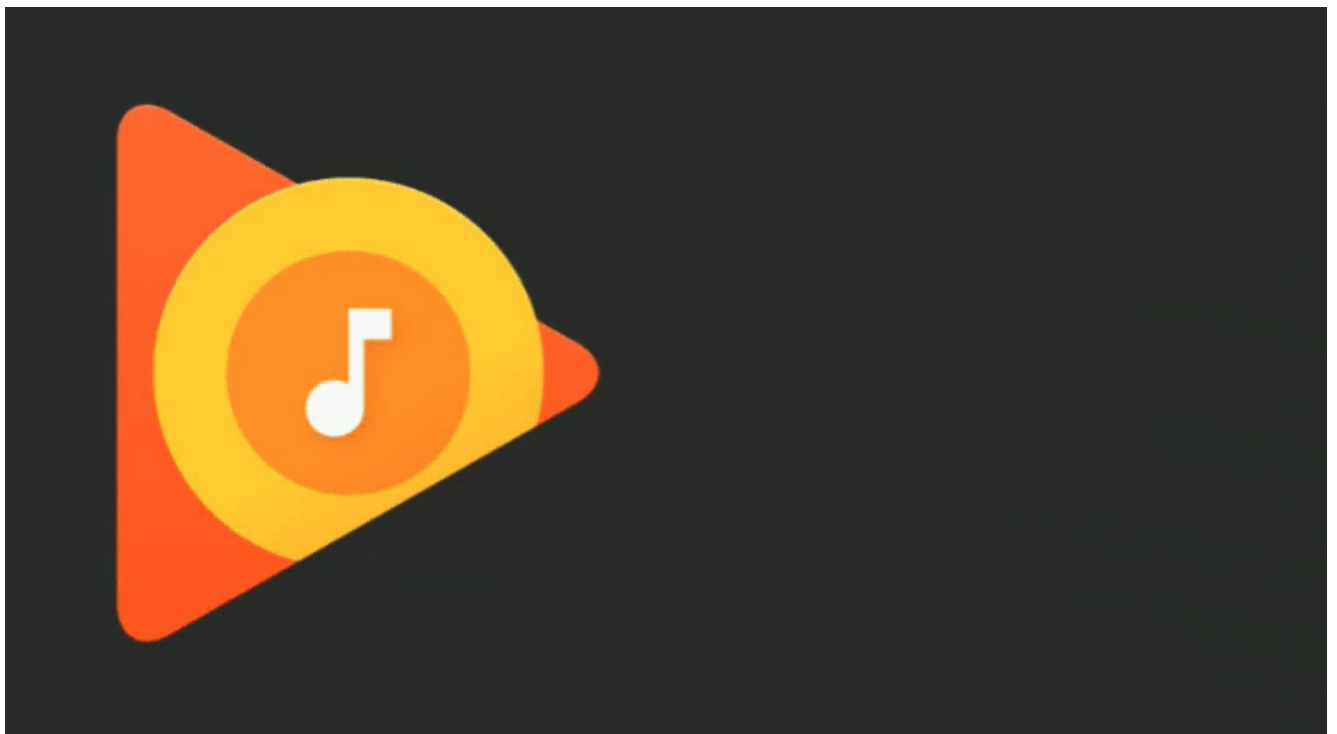
动画会超过目标值一些，然后再弹回来。效果看起来有点像你一屁股坐在沙发上后又被弹起来一点的感觉。



和 `AnticipateInterpolator` 一样，这是个耍花样的 `Interpolator`，没有通用的适用场景。

`AnticipateOvershootInterpolator`

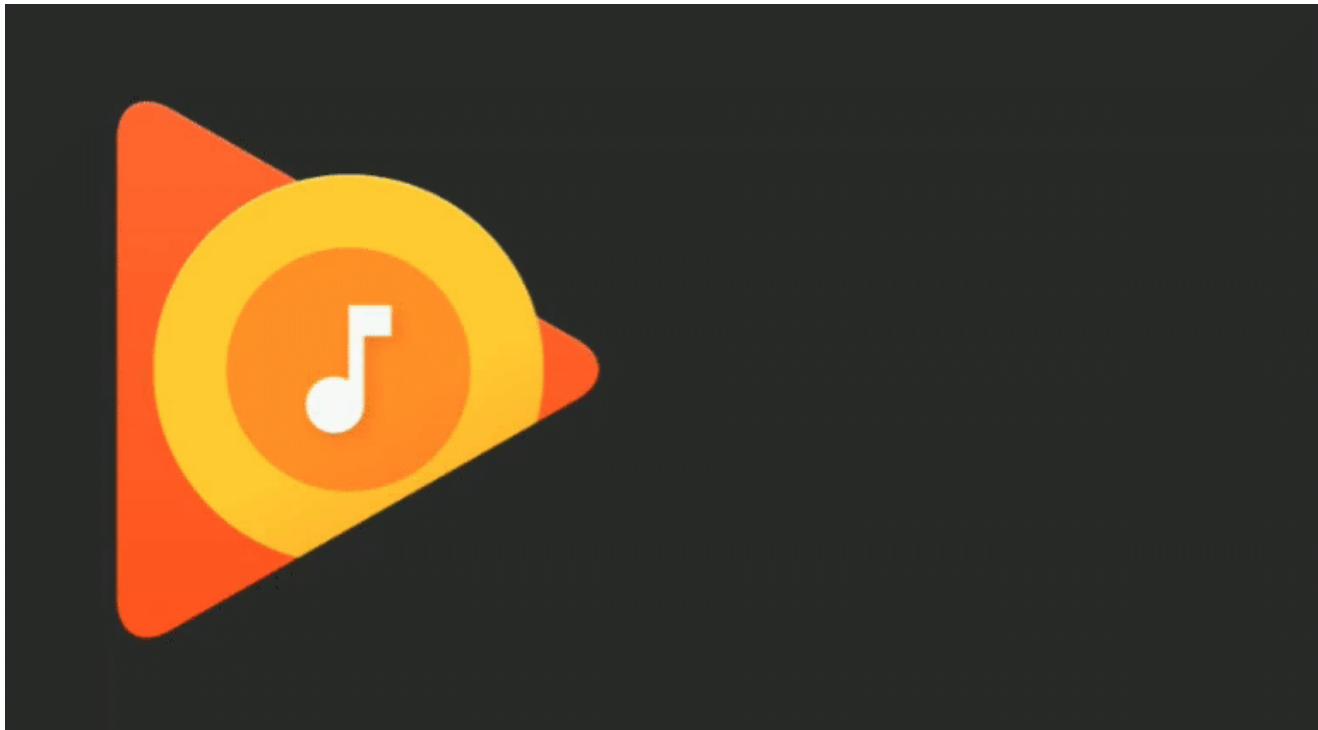
上面这两个的结合版：开始前回拉，最后超过一些然后回弹。



依然耍花样，不多解释。

BounceInterpolator

在目标值处弹跳。有点像玻璃球掉在地板上的效果。

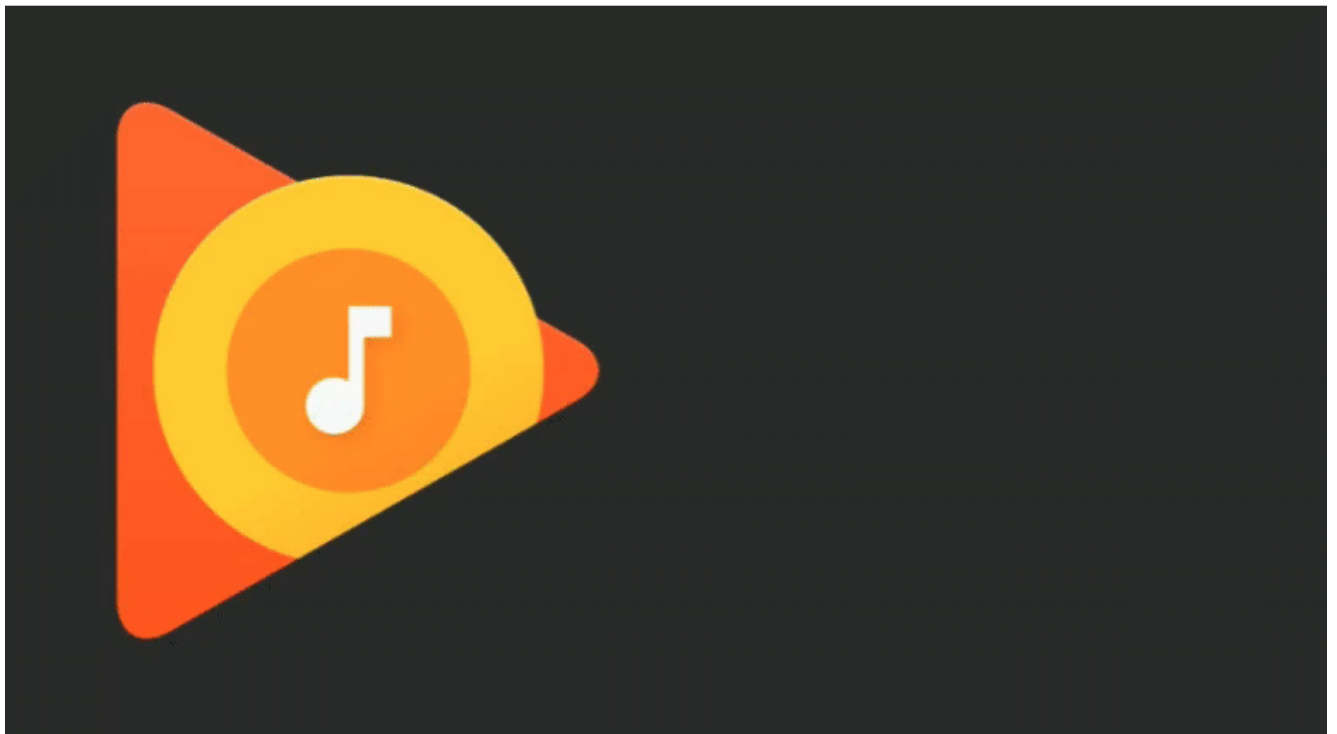


耍花样 +1。

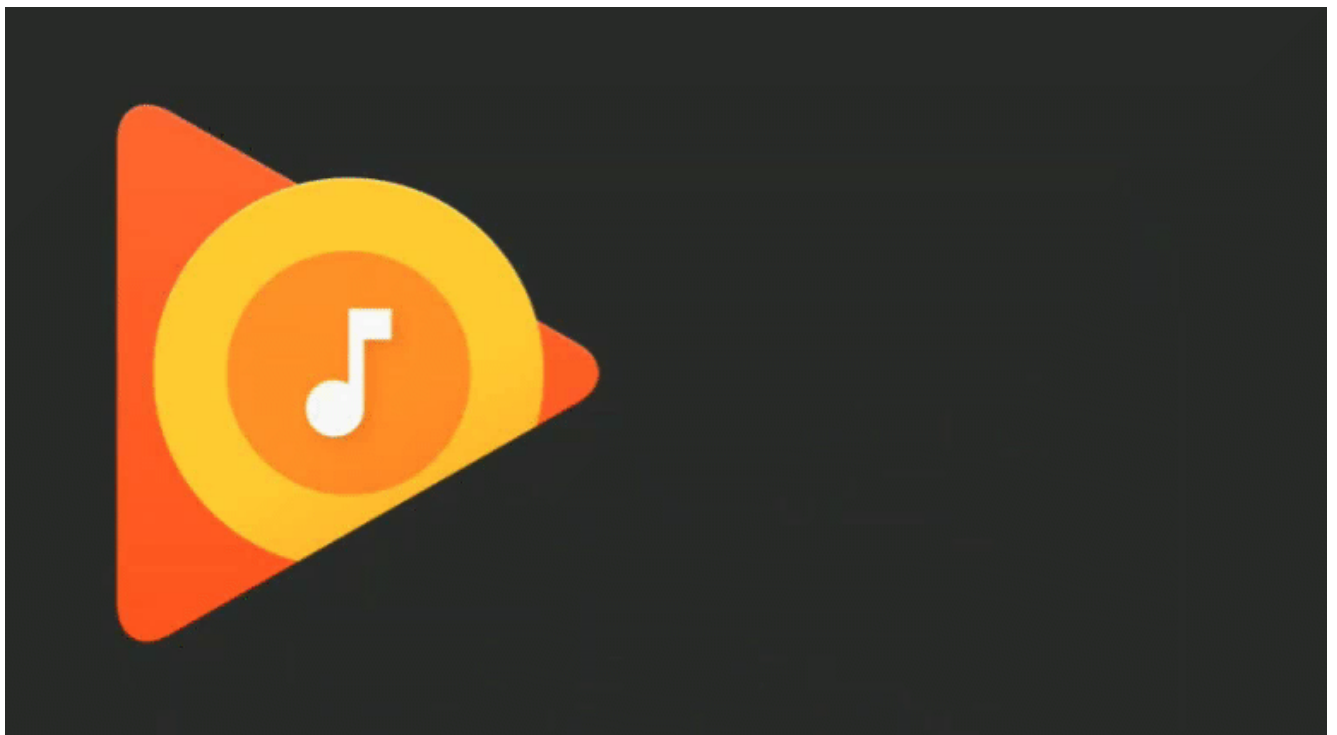
CycleInterpolator

这个也是一个正弦 / 余弦曲线，不过它和 `AccelerateDecelerateInterpolator` 的区别是，它可以自定义曲线的周期，所以动画可以不到终点就结束，也可以到达终点后回弹，回弹的次数由曲线的周期决定，曲线的周期由 `CycleInterpolator()` 构造方法的参数决定。

参数为 0.5f:



参数为 2f:



PathInterpolator

自定义动画完成度 / 时间完成度曲线。

用这个 `Interpolator` 你可以定制出任何你想要的速度模型。定制的方式是使用一个 `Path` 对象来绘制出你要的动画完成度 / 时间完成度曲线。例如：

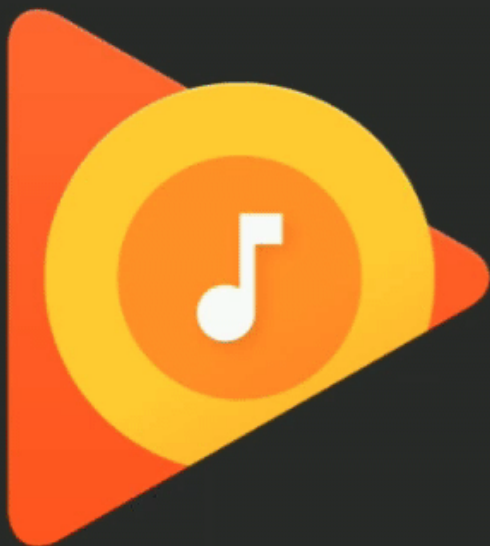
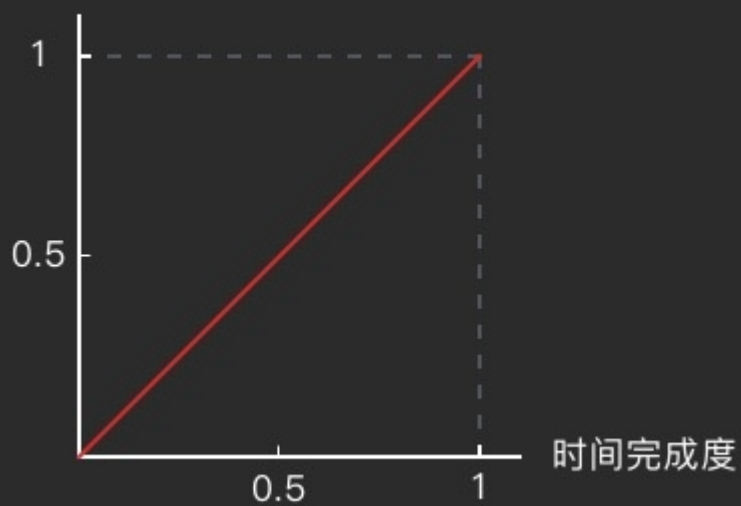
```
Path interpolatorPath = new Path();
```

```
...
```

```
// 匀速
```

```
interpolatorPath.lineTo(1, 1);
```

动画完成度

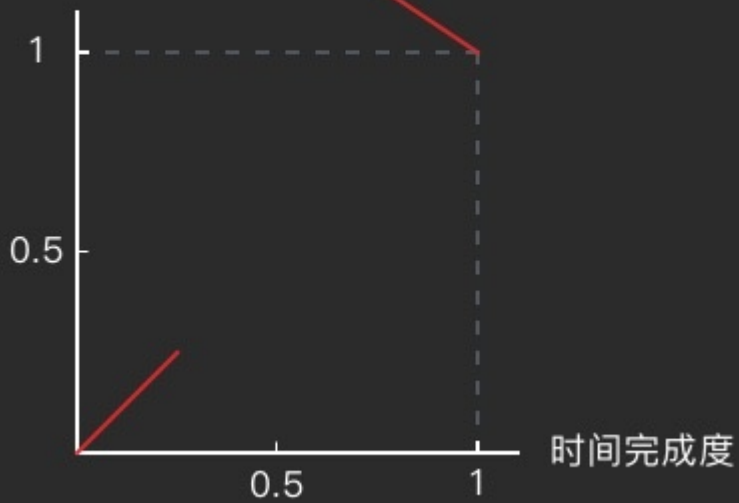


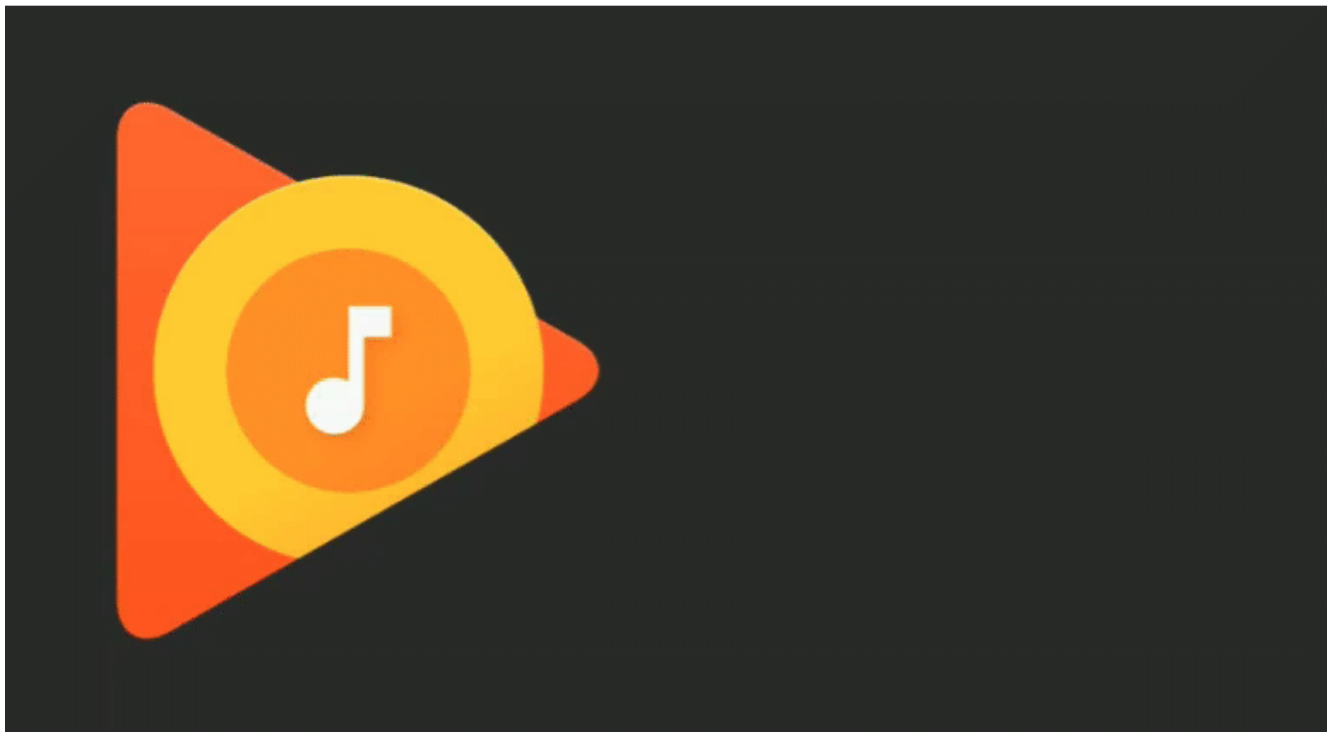
```
Path interpolatorPath = new Path();
```


...

```
// 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);
```

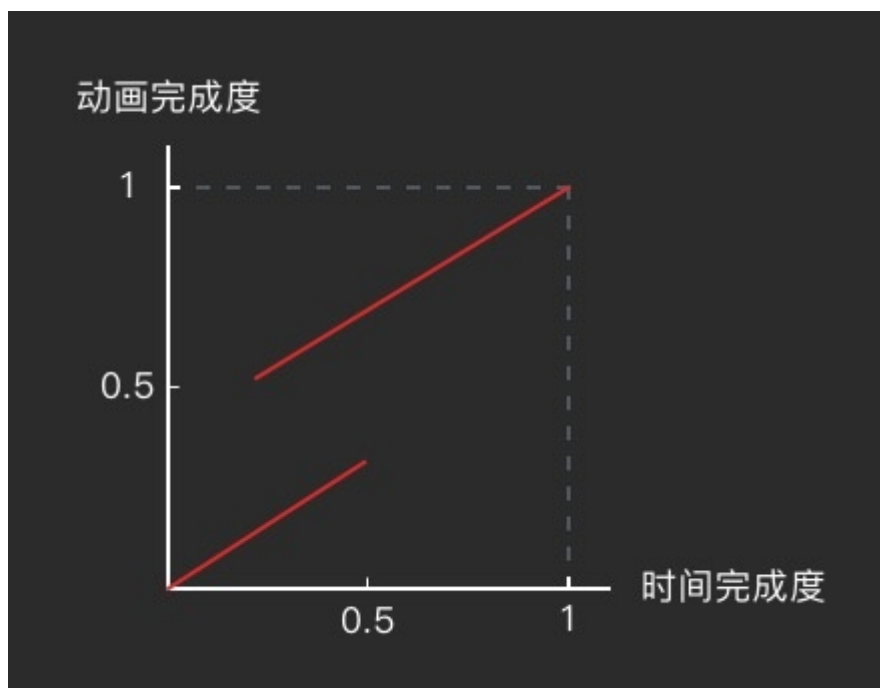
动画完成度



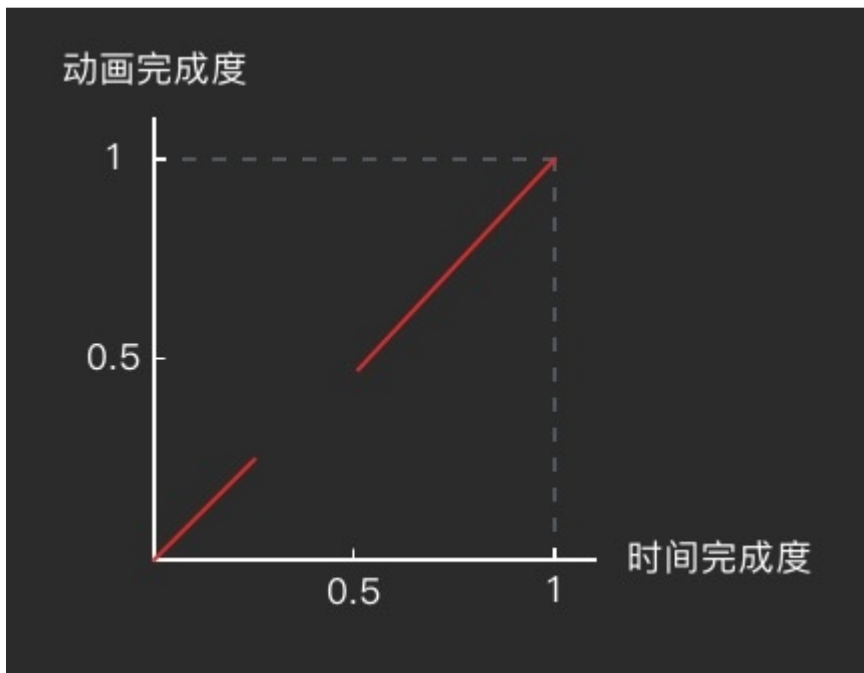


你根据需求，绘制出自己需要的 `Path`，就能定制出你要的速度模型。

不过要注意，这条 `Path` 描述的其实是一个 $y = f(x)$ ($0 \leq x \leq 1$) (y 为动画完成度， x 为时间完成度) 的曲线，所以同一段时间完成度上不能有两段不同的动画完成度（这个好理解吧？因为内容不能出现分身术呀），而且每一个时间完成度的点上都必须要有对应的动画完成度（因为内容不能在某时间段内消失呀）。所以，下面这样的 `Path` 是非法的，会导致程序 FC：



出现重复的动画完成度，即动画内容出现「分身」——程序 FC



有一段时间完成度没有对应的动画完成度，即动画出现「中断」——程序 FC

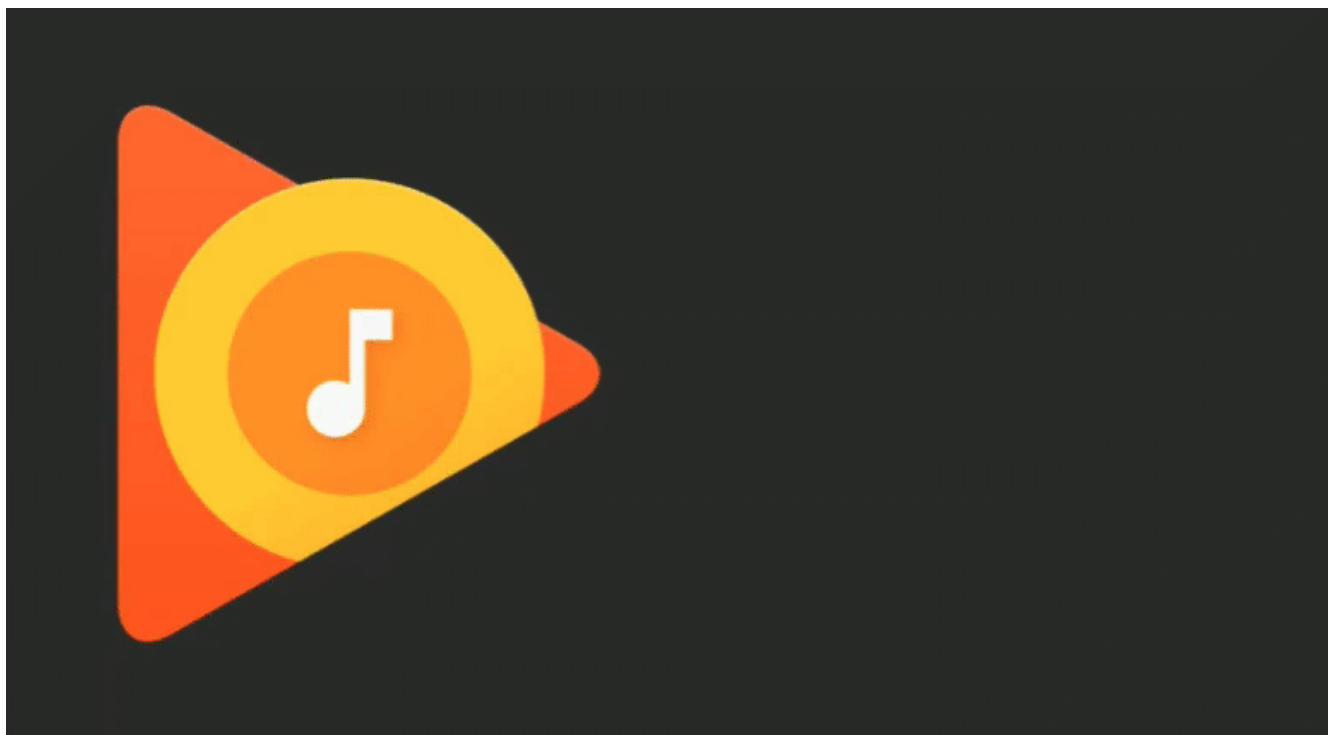
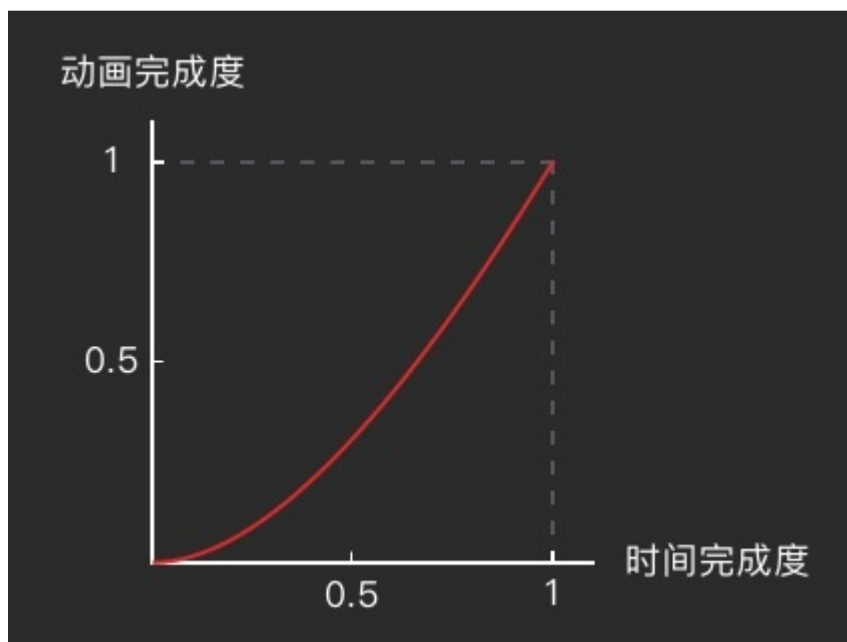
除了上面的这些，Android 5.0 (API 21) 引入了三个新的 `Interpolator` 模型，并把它们加入了 `support v4` 包中。这三个新的 `Interpolator` 每个都和之前的某个已有的 `Interpolator` 规则相似，只有略微的区别。

FastOutLinearInInterpolator

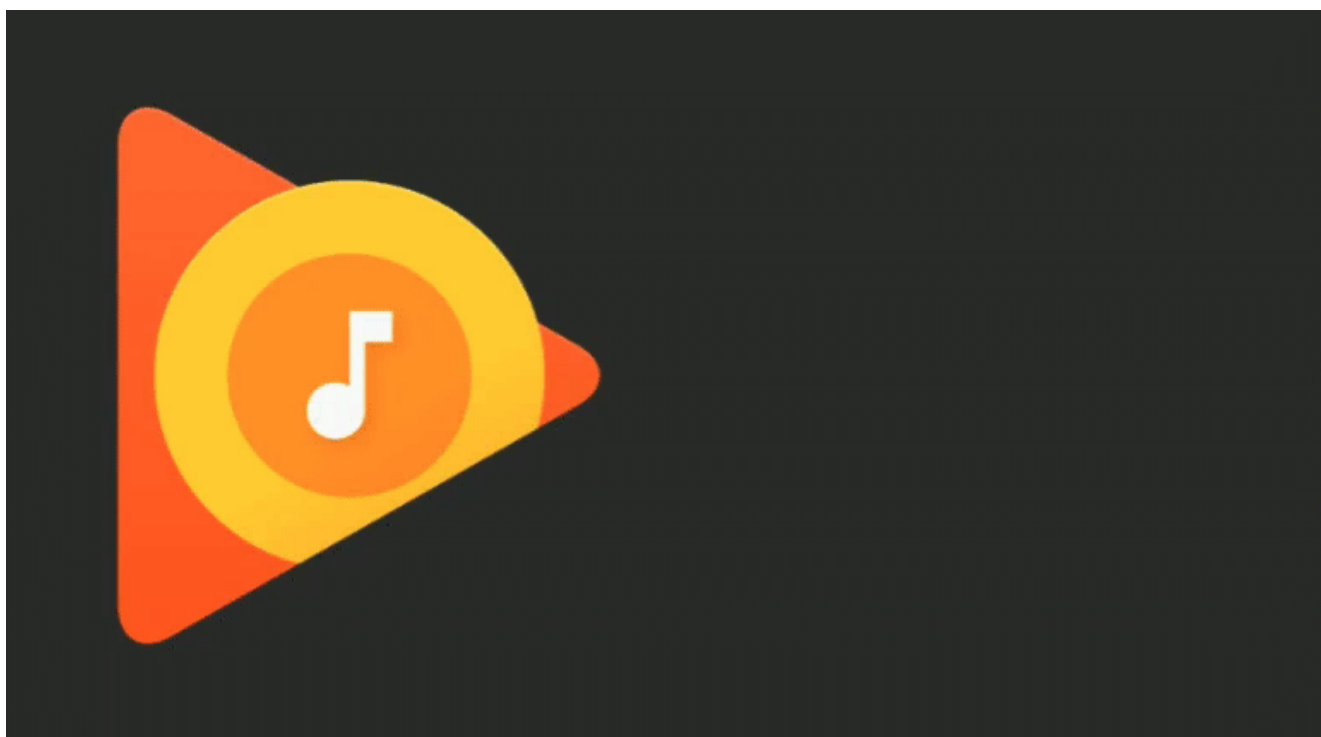
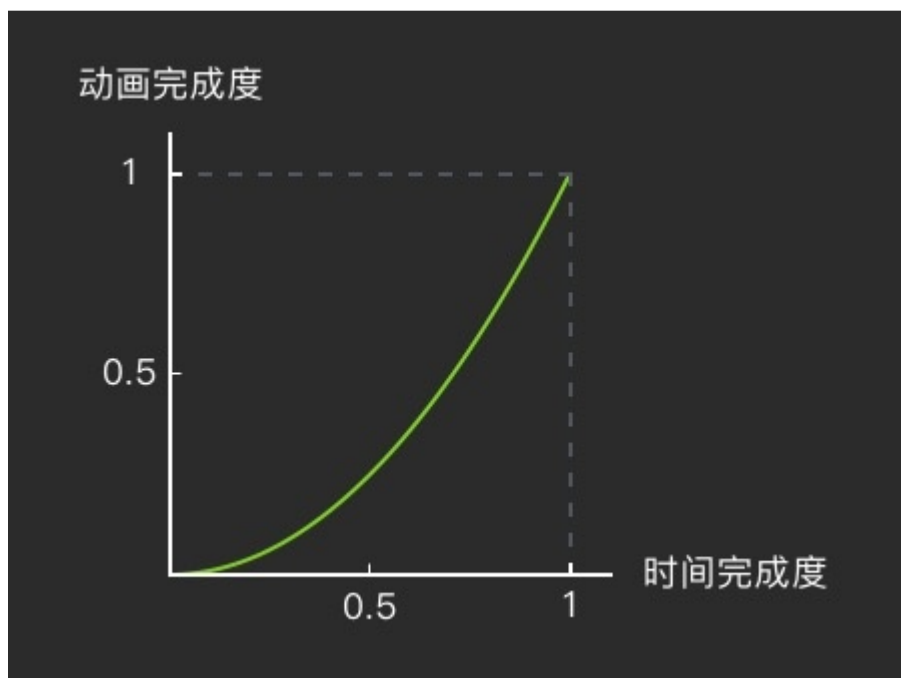
加速运动。

这个 `Interpolator` 的作用你不能看它的名字，一会儿 `fast` 一会儿 `linear` 的，完全看不懂。其实它和 `AccelerateInterpolator` 一样，都是一个持续加速的运动路线。只不过 `FastOutLinearInInterpolator` 的曲线公式是用的贝塞尔曲线，而 `AccelerateInterpolator` 用的是指数曲线。具体来说，它俩最主要的区别是 `FastOutLinearInInterpolator` 的初始阶段加速度比 `AccelerateInterpolator` 要快一些。

`FastOutLinearInInterpolator`：

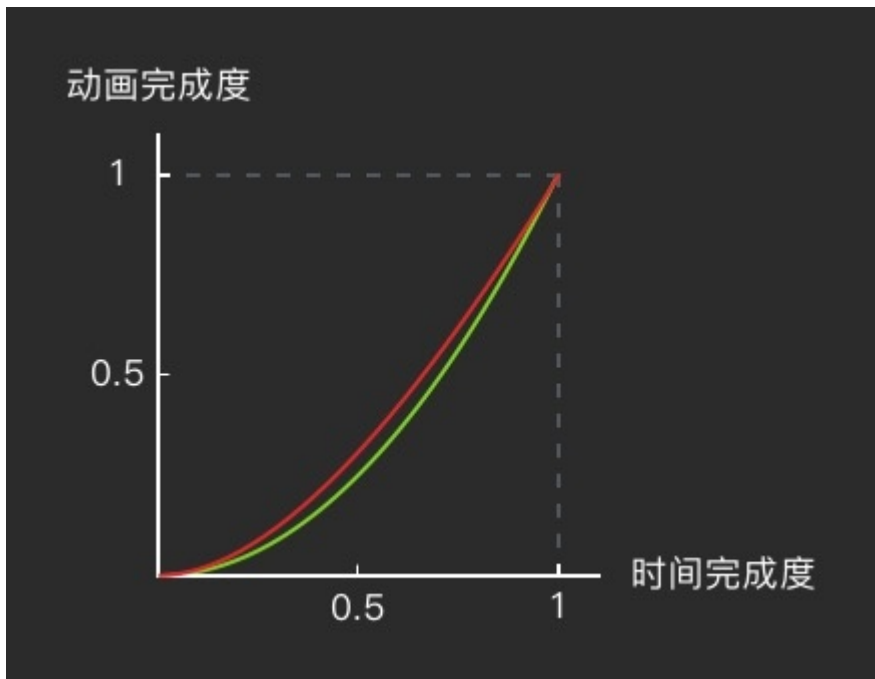


AccelerateInterpolator:



能看出它俩的区别吗？

能看出来就怪了。这俩的速度模型几乎就是一样的，不信我把它们的动画完成度 / 时间完成度曲线放在一起给你看：



看到了吗？两条线几乎是一致的，只是红线比绿线更早地到达了较高的斜率，这说明在初始阶段，`FastOutLinearInInterpolator` 的加速度比 `AccelerateInterpolator` 更高。

那么这意味着什么呢？

意味个毛。实际上，这点区别，在实际应用中用户根本察觉不出来。而且，`AccelerateInterpolator` 还可以在构造方法中调节变速系数，分分钟调节到和 `FastOutLinearInInterpolator`（几乎）一模一样。所以你在使用加速模型的时候，这两个选哪个都一样，没区别的。

那么既然都一样，我做这么多对比，讲这么些干什么呢？

因为我得让你了解。它俩虽然「用起来没区别」，但这是基于我对它足够了解所做出的判断，可我如果直接甩给你一句「它俩没区别，想用谁用谁，少废话别问那么多」，你心里肯定会有一大堆疑问，在开发时用到它们的时候也会畏畏缩缩心里打鼓的，对吧？

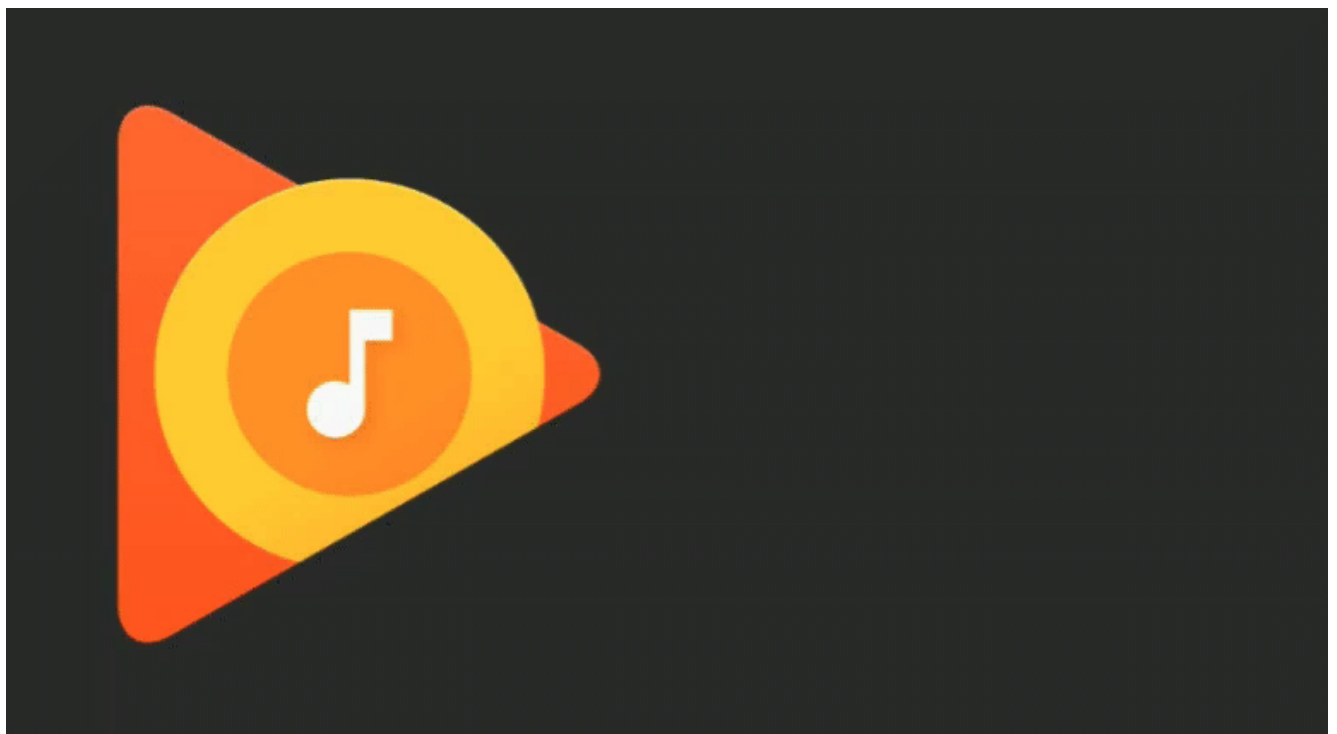
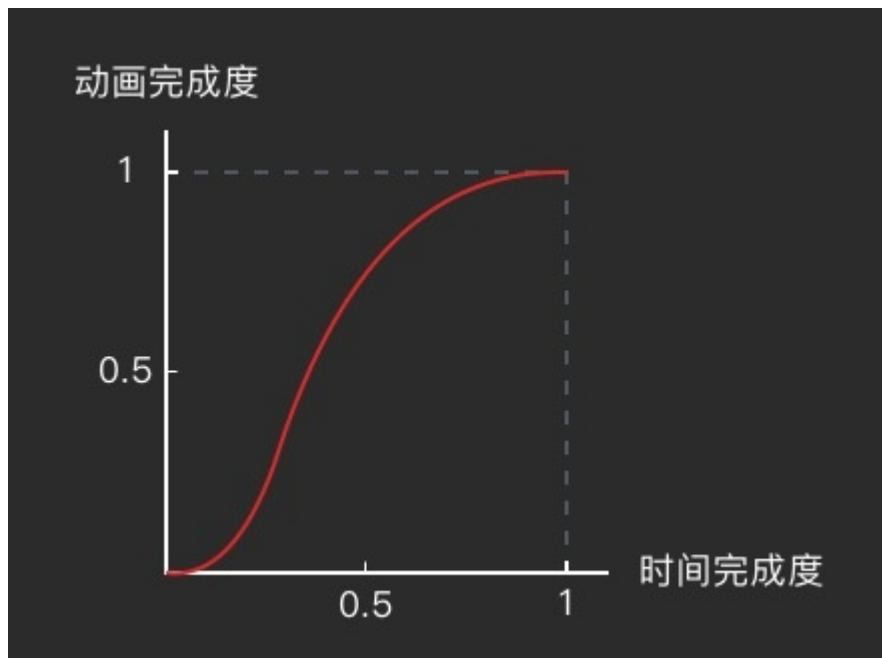
FastOutSlowInInterpolator

先加速再减速。

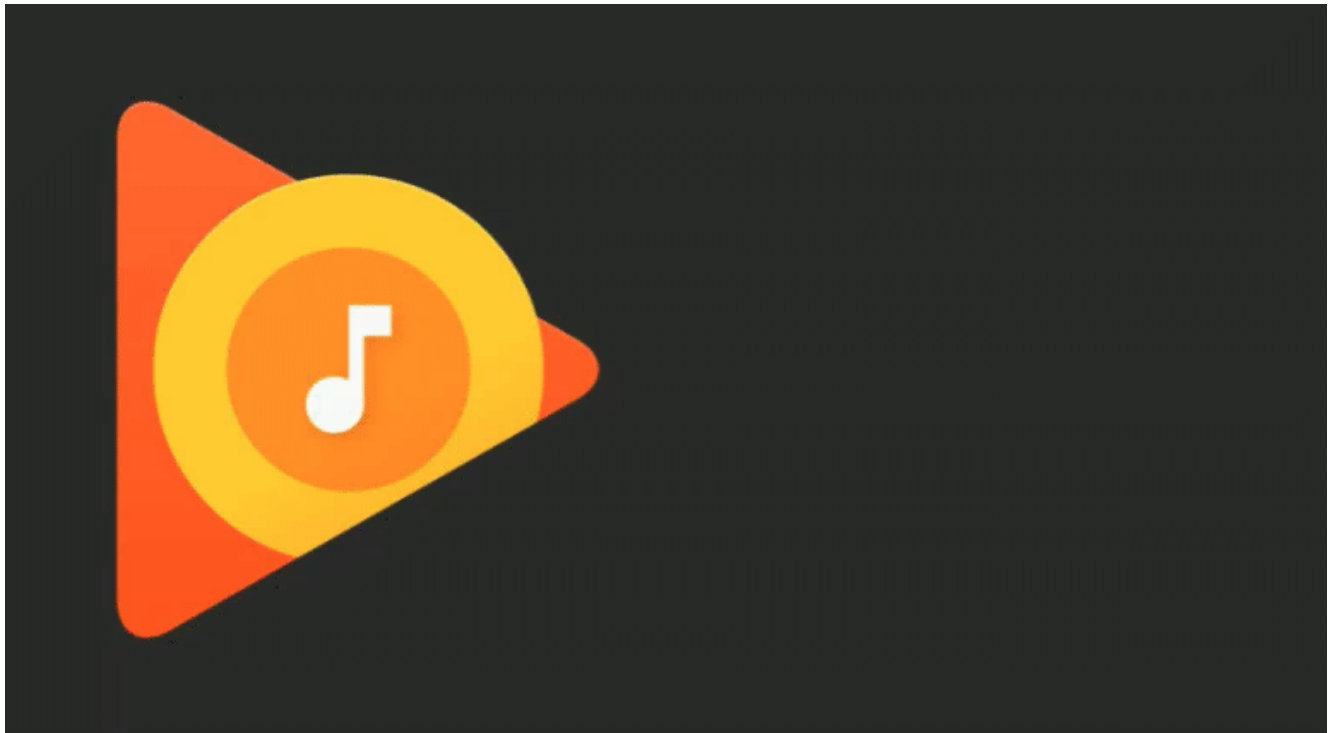
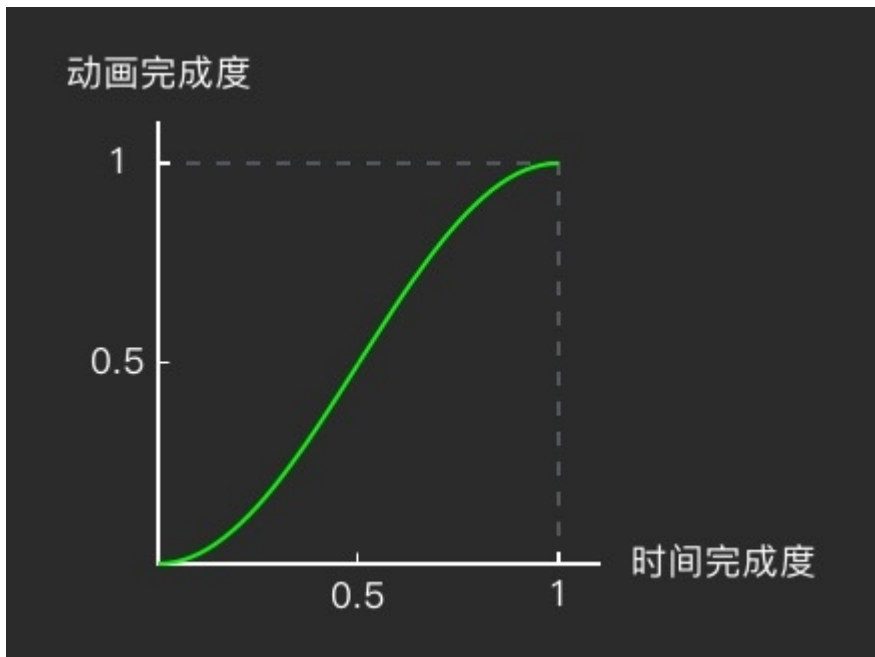
同样也是先加速再减速的还有前面说过的

`AccelerateDecelerateInterpolator`，不过它们的效果是明显不一样的。`FastOutSlowInInterpolator` 用的是贝塞尔曲线，`AccelerateDecelerateInterpolator` 用的是正弦 / 余弦曲线。具体来讲，`FastOutSlowInInterpolator` 的前期加速度要快得多。

`FastOutSlowInInterpolator`：

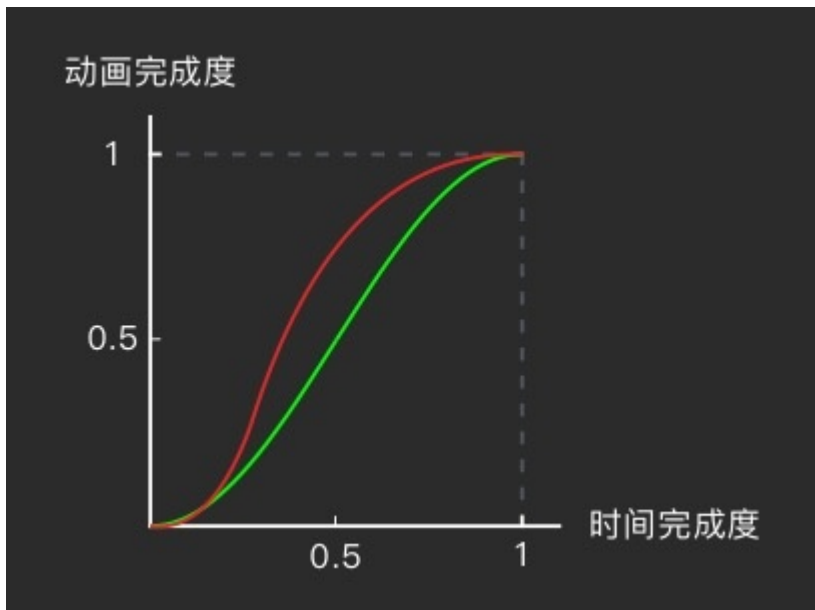


`AccelerateDecelerateInterpolator`：



不论是从动图还是从曲线都可以看出，这二者比起来，`FastOutSlowInInterpolator` 的前期加速更猛一些，后期的减速过程的也减得更迅速。用更直观一点的表达就是，`AccelerateDecelerateInterpolator` 像是物体的自我移动，而 `FastOutSlowInInterpolator` 则看起来像有一股强大的外力「推」着它加速，在接近目标值之后又「拽」着它减速。总之，`FastOutSlowInInterpolator` 看起来有一点「着急」的感觉。

二者曲线对比图：

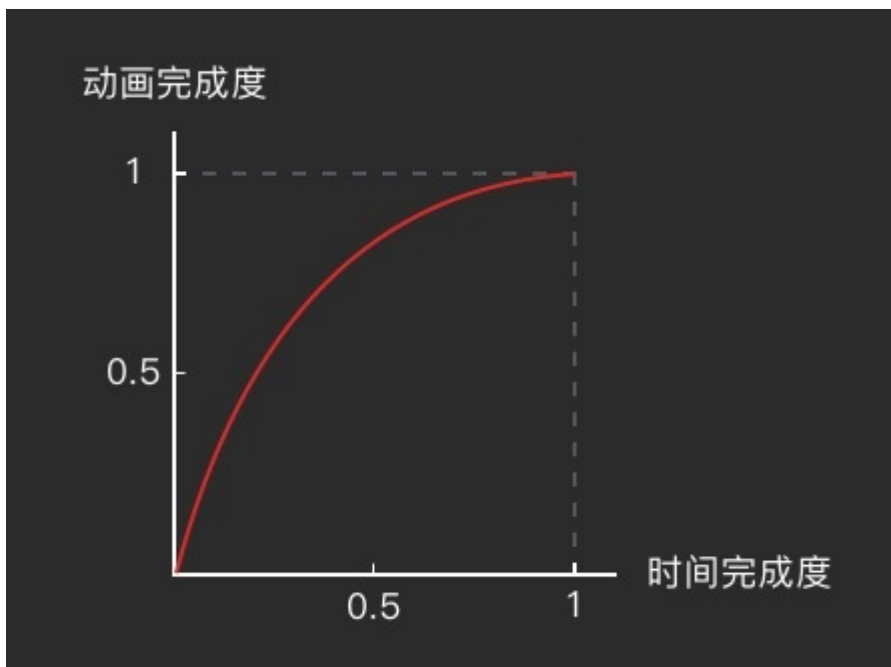


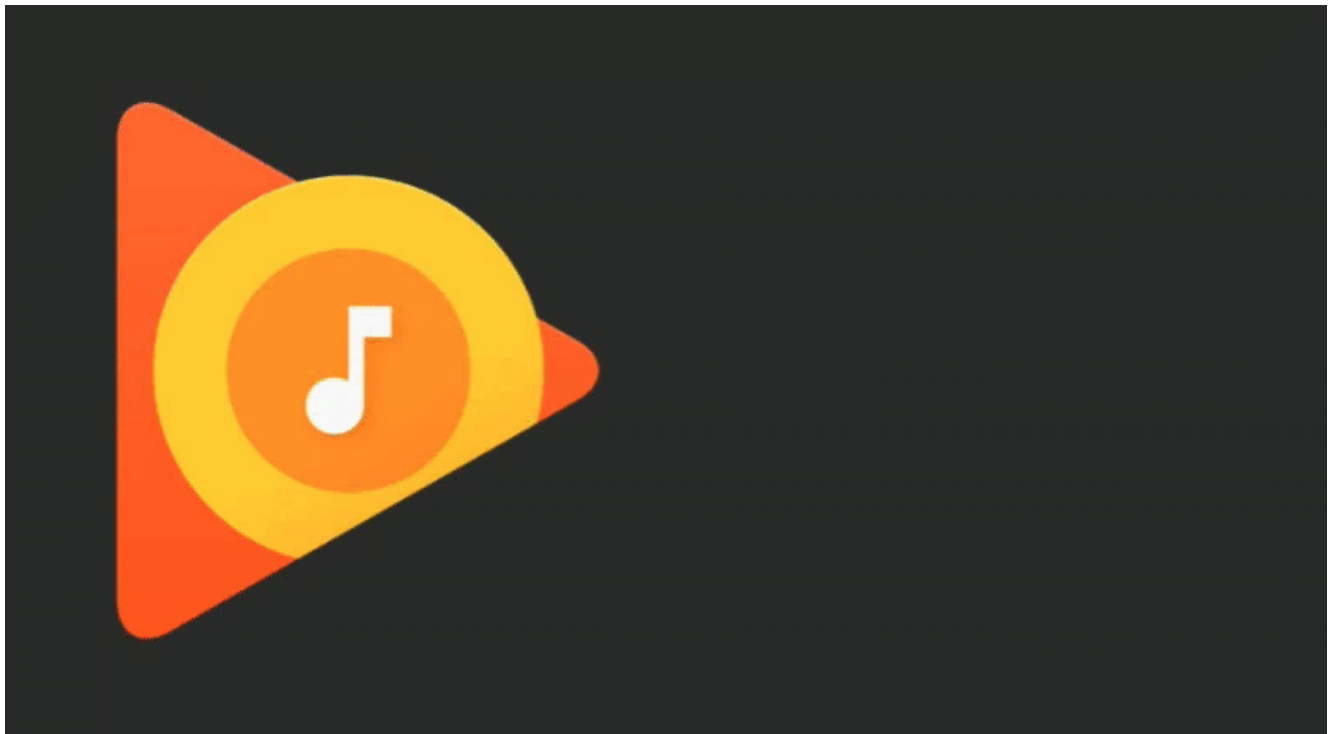
LinearOutSlowInInterpolator

持续减速。

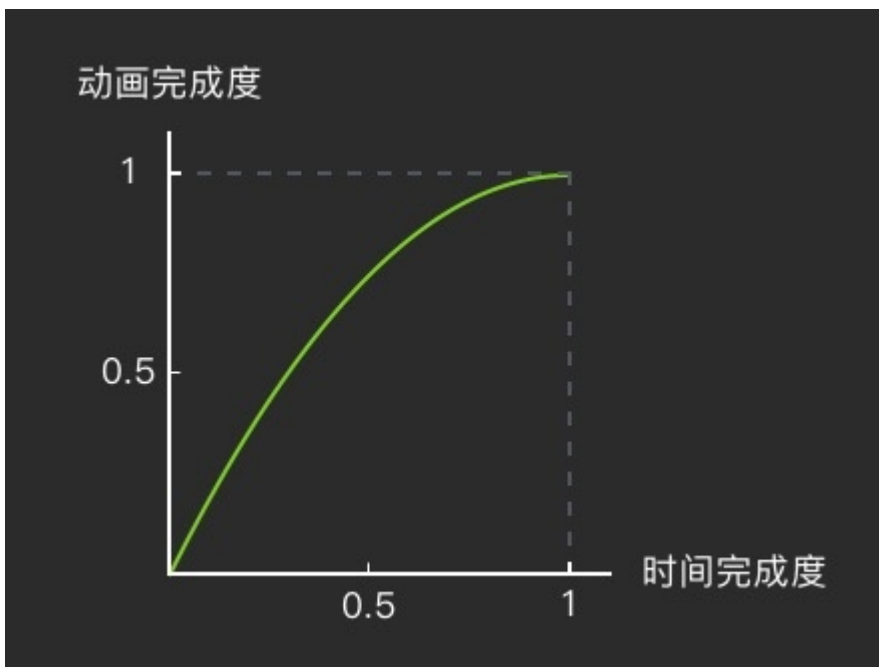
它和 `DecelerateInterpolator` 比起来，同为减速曲线，主要区别在于 `LinearOutSlowInInterpolator` 的初始速度更高。对于人眼的实际感觉，区别其实也不大，不过还是能看出来一些的。

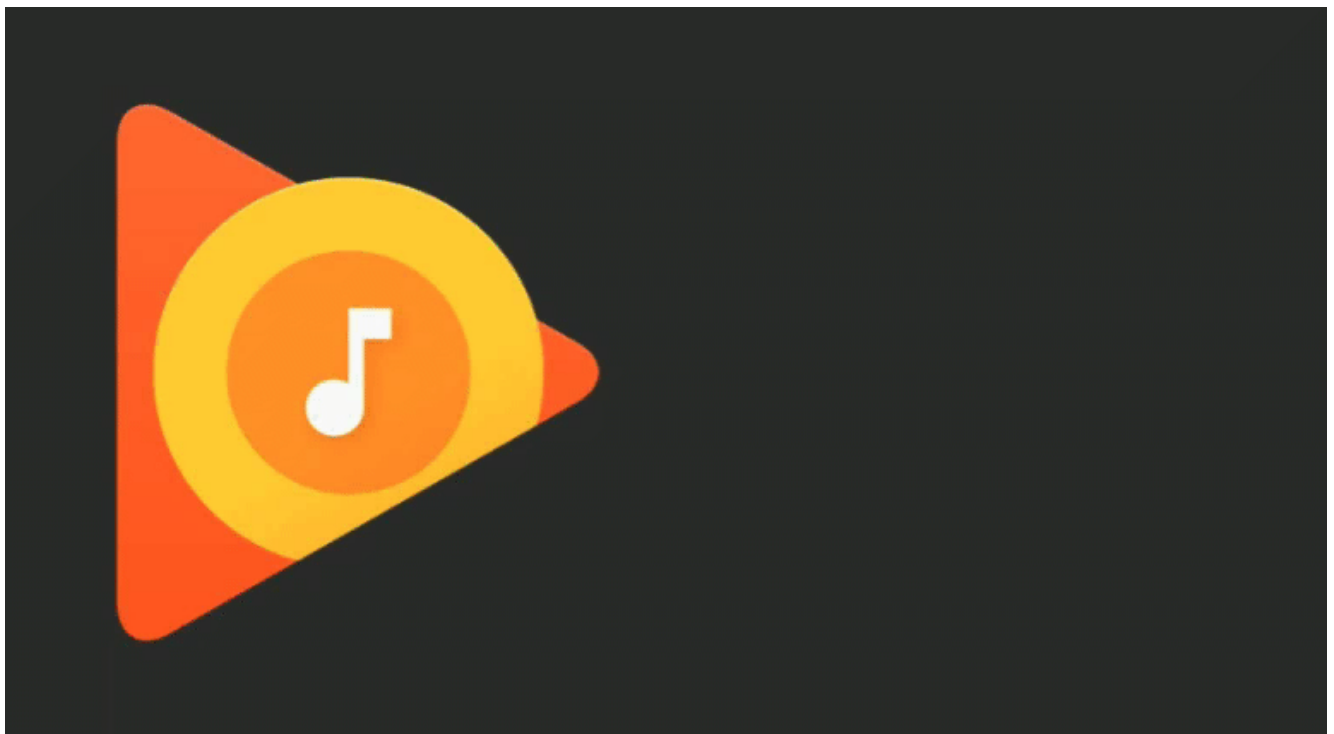
`LinearOutSlowInInterpolator`：



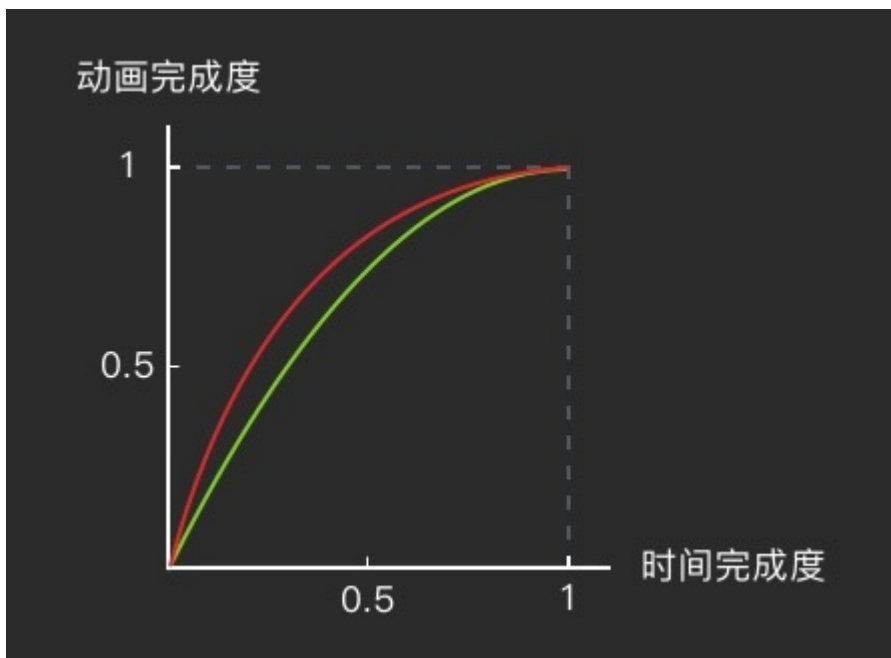


DecelerateInterpolator :





二者曲线对比：



对于所有 Interpolator 的介绍就到这里。这些 Interpolator，有的较为常用且有通用的使用场景，有的需要你自己来根据情况而定。把它们了解清楚了，对于制作出观感舒服的动画很有好处。

3. 设置监听器

给动画设置监听器，可以在关键时刻得到反馈，从而及时做出合适的操作，例如在动画的属性更新时同步更新其他数据，或者在动画结束后回收资源等。

设置监听器的方法，`ViewPropertyAnimator` 和 `ObjectAnimator` 略微不一样：`ViewPropertyAnimator` 用的是 `setListener()` 和 `setUpdateListener()` 方法，可以设置一个监听器，要移除监听器时通过 `set[Update]Listener(null)` 填 `null` 值来移除；而 `ObjectAnimator` 则是用 `addListener()` 和 `addUpdateListener()` 来添加一个或多个监听器，移除监听器则是通过 `remove[Update]Listener()` 来指定移除对象。

另外，由于 `ObjectAnimator` 支持使用 `pause()` 方法暂停，所以它还多了一个 `addPauseListener()` / `removePauseListener()` 的支持；而 `ViewPropertyAnimator` 则独有 `withStartAction()` 和 `withEndAction()` 方法，可以设置一次性的动画开始或结束的监听。

3.1 ViewPropertyAnimator.setListener() / ObjectAnimator.addListener()

这两个方法的名称不一样，可以设置的监听器数量也不一样，但它们的参数类型都是 `AnimatorListener`，所以本质上其实都是一样的。`AnimatorListener` 共有 4 个回调方法：

3.1.1 onAnimationStart(Animator animation)

当动画开始执行时，这个方法被调用。

3.1.2 onAnimationEnd(Animator animation)

当动画结束时，这个方法被调用。

3.1.3 onAnimationCancel(Animator animation)

当动画被通过 `cancel()` 方法取消时，这个方法被调用。

需要说明一下的是，就算动画被取消，`onAnimationEnd()` 也会被调用。所以当动画被取消时，如果设置了 `AnimatorListener`，那么 `onAnimationCancel()` 和 `onAnimationEnd()` 都会被调用。`onAnimationCancel()` 会先于 `onAnimationEnd()` 被调用。

3.1.4 `onAnimationRepeat(Animator animation)`

当动画通过 `setRepeatMode()` / `setRepeatCount()` 或 `repeat()` 方法重复执行时，这个方法被调用。

由于 `ViewPropertyAnimator` 不支持重复，所以这个方法对 `ViewPropertyAnimator` 相当于无效。

3.2 `ViewPropertyAnimator.setUpdateListener()` / `ObjectAnimator.addUpdateListener()`

和上面 3.1 的两个方法一样，这两个方法虽然名称和可设置的监听器数量不一样，但本质其实都一样的，它们的参数都是 `AnimatorUpdateListener`。它只有一个回调方法：`onAnimationUpdate(ValueAnimator animation)`。

3.2.1 `onAnimationUpdate(ValueAnimator animation)`

当动画的属性更新时（不严谨的说，即每过 10 毫秒，动画的完成度更新时），这个方法被调用。

方法的参数是一个 `ValueAnimator`，`ValueAnimator` 是 `ObjectAnimator` 的父类，也是 `ViewPropertyAnimator` 的内部实现，所以这个参数其实就是 `ViewPropertyAnimator` 内部的那个 `ValueAnimator`，或者对于 `ObjectAnimator` 来说就是它自己本身。

`ValueAnimator` 有很多方法可以用，它可以查看当前的动画完成度、当前的属性值等等。不过 `ValueAnimator` 是下一期才讲的内容，所以这期就不多说了。

3.3 `ObjectAnimator.addPauseListener()`

由于 `ObjectAnimator.pause()` 是下期的内容，所以这个方法在这期就不讲了。当然，如果你有兴趣的话，现在就了解一下也可以。

3.3

ViewPropertyAnimator.withStartAction/EndAction()

这两个方法是 `ViewPropertyAnimator` 的独有方法。它们和 `set/addListener()` 中回调的 `onAnimationStart()` / `onAnimationEnd()` 相比起来的不同主要有两点：

1. `withStartAction()` / `withEndAction()` 是一次性的，在动画执行结束后就自动弃掉了，就算之后再重用 `ViewPropertyAnimator` 来做别的动画，用它们设置的回调也不会再被调用。而 `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。
2. `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()` 的行为是不一致的。

关于监听器，就说到这里。本期内容的讲义部分也到此结束。

练习项目

为了避免转头就忘，强烈建议你趁热打铁，做一下这个练习项目：

[HenCoderPracticeDraw](#)

HenCoder 绘制 6

translationX/Y/Z()


rotation/X/Y()

scaleX/Y()


alpha()

ViewPropertyAnimator-多属性

setDuration()



ANIMATE



ANIMATE

HenCoder 绘制 6

translationX/Y/Z()


rotation/X/Y()

scaleX/Y()


alpha()

ViewPropertyAnimator-多属性

setDuration()



ANIMATE



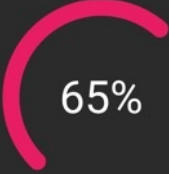
ANIMATE

HenCoder 绘制 6

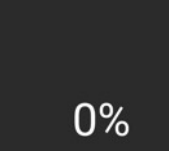
setDuration()

setInterpolator()

ObjectAnimator



ANIMATE



ANIMATE