简介

精心设计的动画会让用户界面感觉更直观、流畅，能改善用户体验。 Flutter可以轻松实现各种动画类型，对于许多widget，特别是Material Design widgets，都带有在其设计规范中定义的标准动画效果(但也可以自定义这些效果)。本章将详细介绍Flutter的动画系统，并会通过几个小实例来演示，以帮助开发者可以迅速理解并掌握动画的开发流程与原理。

本章目录

动画

在任何系统的UI框架中，动画实现的原理都是相同的，即：在一段时间内，快速地多次改变UI外观，由于人眼会产生视觉暂留，最终看到的就是一个“连续”的动画，这和电影的原理是一样的，而UI的一次改变称为一个动画帧，对应一次屏幕刷新，而决定动画流畅度的一个重要指标就是帧率FPS（Frame Per Second），指每秒的动画帧数。很明显，帧率越高则动画就会越流畅。一般情况下，对于人眼来说，动画帧率超过16FPS，就比较流畅了，超过32FPS就会非常的细腻平滑，而超过32FPS基本就感受不到差别了。由于动画的每一帧都是要改变UI输出，所以在一个时间段内连续的改变UI输出是比较耗资源的，对设备的软硬件系统要求都较高，所以在UI系统中，动画的平均帧率是重要的性能指标，而在Flutter中，理想情况下是可以实现60FPS的，这和原生应用动画基本是持平的。

Flutter中动画抽象

为了方便开发者创建动画，不同的UI系统对动画都进行了一些抽象，比如在Android中可以通过XML来描述一个动画然后设置给View。Flutter中也对动画进行了抽象，主要涉及Tween、Animation、Curve、Controller这些角色。

Animation

Animation对象本身和UI渲染没有任何关系。Animation是一个抽象类，它用于保存动画的插值和状态；其中一个比较常用的Animation类是Animation<double>。Animation对象是一个在一段时间内依次生成一个区间(Tween)之间值的类。Animation对象的输出值可以是线性的、曲线的、一个步进函数或者任何其他曲线函数。 根据Animation对象的控制方式，动画可以反向运行，甚至可以在中间切换方向。Animation还可以生成除double之外的其他类型值，如：Animation<Color> 或 Animation<Size>。可以通过Animation对象的`value`属性获取动画的当前值。

动画通知

我们可以通过Animation来监听动画的帧和状态变化：

1. `addListener()`可以给Animation添加帧监听器，在每一帧都会被调用。帧监听器中最常见的行为是改变状态后调用setState()来触发UI重建。
2. `addStatusListener()`可以给Animation添加“动画状态改变”监听器；动画开始、结束、正向或反向（见AnimationStatus定义）时会调用StatusListener。

在后面的章节中我们将会举例说明。

Curve

动画过程可以是匀速的、加速的或者先加速后减速等。Flutter中通过Curve（曲线）来描述动画过程，Curve可以是线性的(Curves.linear)，也可以是非线性的。

CurvedAnimation 将动画过程定义为一个非线性曲线.

```
final CurvedAnimation curve =
    new CurvedAnimation(parent: controller, curve: Curves.easeIn);
```

注: Curves 类类定义了许多常用的曲线，也可以创建自己的，例如：

```
class ShakeCurve extends Curve {
  @override
  double transform(double t) {
    return math.sin(t * math.PI * 2);
  }
}
```

CurvedAnimation和AnimationController（下面介绍）都是Animation<double>类型。CurvedAnimation可以通过包装AnimationController和Curve生成一个新的动画对象 。

AnimationController

AnimationController用于控制动画，它包含动画的启动`forward()`、停止`stop()`、反向播放`reverse()`等方法。AnimationController会在动画的每一帧，就会生成一个新的值。默认情况下，AnimationController在给定的时间段内线性的生成从0.0到1.0（默认区间）的数字。 例如，下面代码创建一个Animation对象，但不会启动它运行：

```
final AnimationController controller = new AnimationController(
    duration: const Duration(milliseconds: 2000), vsync: this);
```

AnimationController生成数字的区间可以通过`lowerBound`和`upperBound`来指定，如：

```
final AnimationController controller = new AnimationController(
 duration: const Duration(milliseconds: 2000),
 lowerBound: 10.0,
 upperBound: 20.0,
 vsync: this
);
```

AnimationController派生自Animation<double>，因此可以在需要Animation对象的任何地方使用。 但是，AnimationController具有控制动画的其他方法，例如`forward()`方法可以启动动画。数字的产生与屏幕刷新有关，因此每秒钟通常会产生60个数字(即60fps)，在动画的每一帧，生成新的数字后，每个Animation对象会调用其Listener对象回调，等动画状态发生改变时（如动画结束）会调用StatusListeners监听器。

duration表示动画执行的时长，通过它我们可以控制动画的速度。

> **注意**： 在某些情况下，动画值可能会超出AnimationController的0.0-1.0的范围。例如，`fling()`函数允许您提供速度(velocity)、力量(force)等，因此可以在0.0到1.0范围之外。 CurvedAnimation生成的值也可以超出0.0到1.0的范围。根据选择的曲线，CurvedAnimation的输出可以具有比输入更大的范围。例如，Curves.elasticIn等弹性曲线会生成大于或小于默认范围的值。

Ticker

当创建一个AnimationController时，需要传递一个`vsync`参数，它接收一个TickerProvider类型的对象，它的主要职责是创建Ticker，定义如下：

```
abstract class TickerProvider {
  //通过一个回调创建一个Ticker
  Ticker createTicker(TickerCallback onTick);
}
```

Flutter应用在启动时都会绑定一个SchedulerBinding，通过SchedulerBinding可以给每一次屏幕刷新添加回调，而Ticker就是通过SchedulerBinding来添加屏幕刷新回调，这样一来，每次屏幕刷新都会调用TickerCallback。使用Ticker(而不是Timer)来驱动动画会防止屏幕外动画（动画的UI不在当前屏幕时，如锁屏时）消耗不必要的资源，因为Flutter中屏幕刷新时会通知到绑定的SchedulerBinding，而Ticker是受SchedulerBinding驱动的，由于锁屏后屏幕会停止刷新，所以Ticker就不会再触发。

通过将SingleTickerProviderStateMixin添加到State的定义中，然后将State对象作为`vsync`的值，这在后面的例子中可以见到。

Tween

默认情况下，AnimationController对象值的范围是0.0到1.0。如果我们需要不同的范围或不同的数据类型，则可以使用Tween来配置动画以生成不同的范围或数据类型的值。例如，像下面示例，Tween生成从-200.0到0.0的值：

```
final Tween doubleTween = new Tween<double>(begin: -200.0, end:
0.0);
```

Tween构造函数需要`begin`和`end`两个参数。Tween的唯一职责就是定义从输入范围到输出范围的映射。输入范围通常为0.0到1.0，但这不是必须的，我们可以自定义需要的范围。

Tween继承自Animatable<T>，而不是继承自Animation<T>。Animatable与Animation相似，不是必须输出double值。例如，ColorTween指定两种颜色之间的过渡。

```
final Tween colorTween =
    new ColorTween(begin: Colors.transparent, end: Colors.black54);
```

Tween对象不存储任何状态，相反，它提供了`evaluate(Animation<double> animation)`方法，它可以获取动画当前值。 Animation对象的当前值可以通过`value()`方法取到。`evaluate`函数还执行一些其它处理，例如分别确保在动画值为0.0和1.0时返回开始和结束状态。

Tween.animate

要使用Tween对象，需要调用其`animate()`方法，然后传入一个控制器对象。例如，以下代码在500毫秒内生成从0到255的整数值。

```
final AnimationController controller = new AnimationController(
    duration: const Duration(milliseconds: 500), vsync: this);
Animation<int> alpha = new IntTween(begin: 0, end:
255).animate(controller);
```

注意`animate()`返回的是一个Animation，而不是一个Animatable。

以下示例构建了一个控制器、一条曲线和一个Tween：

```
final AnimationController controller = new AnimationController(
    duration: const Duration(milliseconds: 500), vsync: this);
final Animation curve =
    new CurvedAnimation(parent: controller, curve: Curves.easeOut);
Animation<int> alpha = new IntTween(begin: 0, end:
255).animate(curve);
```

动画基本结构

我们通过实现一个图片逐渐放大的示例来演示一下Flutter中动画的基本结构：

```
class ScaleAnimationRoute extends StatefulWidget {
  @override
  _ScaleAnimationRouteState createState() => new
_ScaleAnimationRouteState();
}

//需要继承TickerProvider，如果有多个AnimationController，则应该使用
TickerProviderStateMixin。
class _ScaleAnimationRouteState extends State<ScaleAnimationRoute>
with SingleTickerProviderStateMixin{

  Animation<double> animation;
  AnimationController controller;

  initState() {
    super.initState();
    controller = new AnimationController(
        duration: const Duration(seconds: 3), vsync: this);
    //图片宽高从0变到300
    animation = new Tween(begin: 0.0, end:
300.0).animate(controller)
      ..addListener(() {
        setState(()=>{});
      });
    //启动动画(正向执行)
    controller.forward();
  }

  @override
  Widget build(BuildContext context) {
    return new Center(
       child: Image.asset("images/avatar.png",
          width: animation.value,
          height: animation.value
      ),
    );
  }

  dispose() {
    //路由销毁时需要释放动画资源
    controller.dispose();
    super.dispose();
  }
}
```

上面代码中`addListener()`函数调用了`setState()`，所以每次动画生成一个新的数字时，当前帧被标记为脏(dirty)，这会导致widget的`build()`方法再次被调用，而在`build()`中，改变Image的宽高，因为它的高度和宽度现在使用的是`animation.value`，所以就会逐渐放大。值得注意的是动画完成时要释放控制器(调用`dispose()`方法)以防止内存泄漏。

上面的例子中并没有指定Curve，所以放大的过程是线性的（匀速），下面我们指定一个Curve，来实现一个类似于弹簧效果的动画过程，我们只需要将`initState`中的代码改为下面这样即可：

```
  initState() {
    super.initState();
    controller = new AnimationController(
        duration: const Duration(seconds: 3), vsync: this);
    //使用弹性曲线
    animation=CurvedAnimation(parent: controller, curve:
Curves.bounceIn);
    //图片宽高从0变到300
    animation = new Tween(begin: 0.0, end:
300.0).animate(animation)
      ..addListener(() {
        setState(() {
        });
      });
    //启动动画
    controller.forward();
  }
```

使用AnimatedWidget简化

细心的读者可能已经发现上面示例中通过`addListener()`和`setState()`来更新UI这一步其实是通用的，如果每个动画中都加这么一句是比较繁琐的。AnimatedWidget类封装了调用`setState()`的细节，并允许我们将Widget分离出来，重构后的代码如下：

```
class AnimatedImage extends AnimatedWidget {
  AnimatedImage({Key key, Animation<double> animation})
      : super(key: key, listenable: animation);

  Widget build(BuildContext context) {
    final Animation<double> animation = listenable;
    return new Center(
```

```
      child: Image.asset("images/avatar.png",
          width: animation.value,
          height: animation.value
      ),
    );
  }
}


class ScaleAnimationRoute extends StatefulWidget {
  @override
  _ScaleAnimationRouteState createState() => new
_ScaleAnimationRouteState();
}

class _ScaleAnimationRouteState extends State<ScaleAnimationRoute>
    with SingleTickerProviderStateMixin {

  Animation<double> animation;
  AnimationController controller;

  initState() {
    super.initState();
    controller = new AnimationController(
        duration: const Duration(seconds: 3), vsync: this);
    //图片宽高从0变到300
    animation = new Tween(begin: 0.0, end:
300.0).animate(controller);
    //启动动画
    controller.forward();
  }

  @override
  Widget build(BuildContext context) {
    return AnimatedImage(animation: animation,);
  }

  dispose() {
    //路由销毁时需要释放动画资源
    controller.dispose();
    super.dispose();
  }
}
```

用AnimatedBuilder重构

用AnimatedWidget可以从动画中分离出widget，而动画的渲染过程（即设置宽高）仍然在AnimatedWidget中，假设如果我们再添加一个widget透明度变化的动画，那么我们需要再实现一个AnimatedWidget，这样不是很优雅，如果我们能把渲染过程也抽象出来，那就会好很多，而AnimatedBuilder正是将渲染逻辑分离出来, 上面的build方法中的代码可以改为：

```
@override
Widget build(BuildContext context) {
  //return AnimatedImage(animation: animation,);
    return AnimatedBuilder(
      animation: animation,
      child: Image.asset("images/avatar.png"),
      builder: (BuildContext ctx, Widget child) {
        return new Center(
          child: Container(
              height: animation.value,
              width: animation.value,
              child: child,
          ),
        );
      },
    );
}
```

上面的代码中有一个迷惑的问题是，`child`看起来像被指定了两次。但实际发生的事情是：将外部引用child传递给AnimatedBuilder后AnimatedBuilder再将其传递给匿名构造器， 然后将该对象用作其子对象。最终的结果是AnimatedBuilder返回的对象插入到Widget树中。

也许你会说这和我们刚开始的示例差不了多少，其实它会带来三个好处：

1. 不用显式的去添加帧监听器，然后再调用`setState()`了，这个好处和AnimatedWidget是一样的。

2. 动画构建的范围缩小了，如果没有builder，setState()将会在父widget上下文调用，这将会导致父widget的build方法重新调用，而有了builder之后，只会导致动画widget的build重新调用，这在复杂布局下性能会提高。

3. 通过AnimatedBuilder可以封装常见的过渡效果来复用动画。下面我们通过封装一个GrowTransition来说明，它可以对子widget实现放大动画：

```
class GrowTransition extends StatelessWidget {
  GrowTransition({this.child, this.animation});

  final Widget child;
  final Animation<double> animation;

  Widget build(BuildContext context) {
    return new Center(
      child: new AnimatedBuilder(
          animation: animation,
          builder: (BuildContext context, Widget child) {
            return new Container(
                height: animation.value,
                width: animation.value,
                child: child
            );
          },
          child: child
      ),
    );
  }
}
```

这样，最初的示例就可以改为：

```
...
Widget build(BuildContext context) {
    return GrowTransition(
    child: Image.asset("images/avatar.png"),
    animation: animation,
    );
}
```

Flutter中正是通过这种方式封装了很多动画，如：FadeTransition、ScaleTransition、SizeTransition、FractionalTranslation等，很多时候都可以复用这些预置的过渡类。

动画状态监听

上面说过，我们可以通过Animation的`addStatusListener()`方法来添加动画状态改变监听器。Flutter中，有四种动画状态，在AnimationStatus枚举类中定义，下面我们逐个说明：

枚举值	含义
`dismissed`	动画在起始点停止
`forward`	动画正在正向执行
`reverse`	动画正在反向执行
`completed`	动画在终点停止

示例

我们将上面图片放大的示例改为先放大再缩小再放大……这样的循环动画。要实现这种效果，我们只需要监听动画状态的改变即可，即：在动画正向执行结束时反转动画，在动画反向执行结束时再正向执行动画。代码如下：

```
  initState() {
    super.initState();
    controller = new AnimationController(
        duration: const Duration(seconds: 1), vsync: this);
    //图片宽高从0变到300
    animation = new Tween(begin: 0.0, end:
300.0).animate(controller);
    animation.addStatusListener((status) {
      if (status == AnimationStatus.completed) {
        //动画执行结束时反向执行动画
        controller.reverse();
      } else if (status == AnimationStatus.dismissed) {
        //动画恢复到初始状态时执行动画（正向）
        controller.forward();
      }
    });

    //启动动画（正向）
    controller.forward();
  }
```

自定义路由切换动画

Material库中提供了一个MaterialPageRoute，它可以使用和平台风格一致的路由切换动画，如在iOS上会左右滑动切换，而在Android上会上下滑动切换。如果在Android上也想使用左右切换风格，可以直接使用CupertinoPageRoute, 如：

```
Navigator.push(context, CupertinoPageRoute(
           builder: (context){
             return PageB(); //路由B
           }
));
```

如果想自定义路由切换动画，可以使用PageRouteBuilder，例如我们想以渐隐渐入动画来实现路由过渡：

```
  Navigator.push(context, PageRouteBuilder(
      transitionDuration: Duration(milliseconds: 500), //动画时间为
500毫秒
      pageBuilder: (BuildContext context, Animation animation,
          Animation secondaryAnimation) {
        return new FadeTransition( //使用渐隐渐入过渡,
          opacity: animation,
          child: PageB(); //路由B
        );
      }));
}),
```

我们可以看到`pageBuilder`有一个animation参数，这是Flutter路由管理器提供的，在路由切换时`pageBuilder`在每个动画帧都会被回调，因此我们可以通过animation对象来自定义过渡动画。

无论是MaterialPageRoute、CupertinoPageRoute，还是PageRouteBuilder，它们都继承自PageRoute类，而PageRouteBuilder其实只是PageRoute的一个包装，我们可以直接继承PageRoute类来实现自定义路由，上面的例子可以通过如下方式实现：

1. 定义一个路由类FadeRoute

```
class FadeRoute extends PageRoute {
  FadeRoute({
    @required this.builder,
    this.transitionDuration = const Duration(milliseconds: 300),
    this.opaque = true,
    this.barrierDismissible = false,
    this.barrierColor,
    this.barrierLabel,
    this.maintainState = true,
  });

  final WidgetBuilder builder;

  @override
  final Duration transitionDuration;

  @override
  final bool opaque;

  @override
  final bool barrierDismissible;

  @override
  final Color barrierColor;

  @override
  final String barrierLabel;

  @override
  final bool maintainState;

  @override
  Widget buildPage(BuildContext context, Animation<double>
animation,
      Animation<double> secondaryAnimation) => builder(context);

  @override
  Widget buildTransitions(BuildContext context,
Animation<double> animation,
      Animation<double> secondaryAnimation, Widget child) {
     return FadeTransition(
       opacity: animation,
       child: builder(context),
     );
  }
}
```

2. 使用FadeRoute

```
Navigator.push(context, FadeRoute(builder: (context) {
  return PageB();
}));
```

虽然上面的两种方法都可以实现自定义切换动画，但实际使用时应考虑优先使用PageRouteBuilder，这样无需定义一个新的路由类，使用起来会比较方便。但是有些时候PageRouteBuilder是不能满足需求的，例如在应用过渡动画时我们需要读取当前路由的一些属性，这时就只能通过继承PageRoute的方式了，举个例子，假如我们只想在打开新路由时应用动画，而在返回时不使用动画，那么我们在构建过渡动画时就必须判断当前路由`isActive`属性是否为true，代码如下：

```
@override
Widget buildTransitions(BuildContext context, Animation<double>
animation,
    Animation<double> secondaryAnimation, Widget child) {
 //当前路由被激活，是打开新路由
 if(isActive) {
   return FadeTransition(
     opacity: animation,
     child: builder(context),
   );
 }else{
   //是返回，则不应用过渡动画
   return Padding(padding: EdgeInsets.zero);
 }
}
```

关于路由参数的详细信息读者可以自行查阅API文档，比较简单，不再赘述。

Hero动画

Hero指的是可以在路由(页面)之间“飞行”的widget，简单来说Hero动画就是在路由切换时，有一个共享的Widget可以在新旧路由间切换，由于共享的Widget在新旧路由页面上的位置、外观可能有所差异，所以在路由切换时会逐渐过渡，这样就会产生一个Hero动画。

你可能多次看到过 hero 动画。例如，一个路由中显示待售商品的缩略图列表，选择一个条目会将其跳转到一个新路由，新路由中包含该商品的详细信息和“购买”按钮。 在Flutter中将图片从一个路由“飞”到另一个路由称为**hero动画**，尽管相同的动作有时也称为 **共享元素转换**。下面我们通过一个示例来体验一下hero 动画。

示例

假设有两个路由A和B，他们的内容交互如下：

A：包含一个用户头像，圆形，点击后跳到B路由，可以查看大图。

B：显示用户头像原图，矩形；

在AB两个路由之间跳转的时候，用户头像会逐渐过渡到目标路由页的头像上，接下来我们先看看代码，然后再解析：

```
// 路由A
class HeroAnimationRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      alignment: Alignment.topCenter,
      child: InkWell(
        child: Hero(
          tag: "avatar", //唯一标记，前后两个路由页Hero的tag必须相同
          child: ClipOval(
            child: Image.asset("images/avatar.png",
              width: 50.0,
            ),
          ),
        ),
        onTap: () {
          //打开B路由
          Navigator.push(context, PageRouteBuilder(
              pageBuilder: (BuildContext context, Animation
animation,
                  Animation secondaryAnimation) {
                return new FadeTransition(
                  opacity: animation,
                  child: PageScaffold(
                    title: "原图",
                    body: HeroAnimationRouteB(),
                  ),
                );
              })
          );
        },
      ),
    );
  }
}
```

路由B:

```
class HeroAnimationRouteB extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Hero(
          tag: "avatar", //唯一标记，前后两个路由页Hero的tag必须相同
          child: Image.asset("images/avatar.png"),
      )
```

```
      ),
    );

  }
}
```

我们可以看到，实现Hero动画只需要用Hero Widget将要共享的Widget包装起来，并提供一个相同的tag即可，中间的过渡帧都是Flutter Framework自动完成的。必须要注意，前后路由页的共享Hero的tag必须是相同的，Flutter Framework内部正式通过tag来对应新旧路由页Widget的对应关系的。

Hero动画的原理比较简单，Flutter Framework知道新旧路由页中共享元素的位置和大小，所以根据这两个端点，在动画执行过程中求出过渡时的插值即可，幸运的是，这些事情Flutter已经帮我们做了。

交错动画

有些时候我们可能会需要一些复杂的动画，这些动画可能由一个动画序列或重叠的动画组成，比如：有一个柱状图，需要在高度增长的同改变颜色，等到增长到最大高度后，我们需要在X轴上平移一段距离。这时我们就需要使用交错动画（Stagger Animation）。交错动画需要注意以下几点：

1. 要创建交错动画，需要使用多个动画对象
2. 一个AnimationController控制所有动画
3. 给每一个动画对象指定间隔（Interval）

所有动画都由同一个[AnimationController](https://docs.flutter.io/flutter/animation/AnimationController-class.html)驱动，无论动画实时持续多长时间，控制器的值必须介于0.0和1.0之间，而每个动画的间隔（Interval）介于0.0和1.0之间。对于在间隔中设置动画的每个属性，请创建一个[Tween](https://docs.flutter.io/flutter/animation/Tween-class.html)。 Tween指定该属性的开始值和结束值。也就是说0.0到1.0代表整个动画过程，我们可以给不同动画指定起始点和终止点来决定它们的开始时间和终止时间。

示例

下面我们看一个例子，实现一个柱状图增长的动画：

1. 开始时高度从0增长到300像素，同时颜色由绿色渐变为红色；这个过程占据整个动画时间的60%。
2. 高度增长到300后，开始沿X轴向右平移100像素；这个过程占用整个动画时间的40%。

2. 高度增长到300后，开始沿X轴向右平移100像素，这个过程占用整个动画时间的40%。

我们将执行动画的Widget分离出来：

```dart
class StaggerAnimation extends StatelessWidget {
  StaggerAnimation({ Key key, this.controller }): super(key: key){
    //高度动画
    height = Tween<double>(
      begin:.0 ,
      end: 300.0,
    ).animate(
      CurvedAnimation(
        parent: controller,
        curve: Interval(
          0.0, 0.6, //间隔，前60%的动画时间
          curve: Curves.ease,
        ),
      ),
    );

    color = ColorTween(
      begin:Colors.green ,
      end:Colors.red,
    ).animate(
      CurvedAnimation(
        parent: controller,
        curve: Interval(
          0.0, 0.6,//间隔，前60%的动画时间
          curve: Curves.ease,
        ),
      ),
    );

    padding = Tween<EdgeInsets>(
      begin:EdgeInsets.only(left: .0),
      end:EdgeInsets.only(left: 100.0),
    ).animate(
      CurvedAnimation(
        parent: controller,
        curve: Interval(
          0.6, 1.0, //间隔，后40%的动画时间
          curve: Curves.ease,
        ),
      ),
    );
  }
```

```
  final Animation<double> controller;
  Animation<double> height;
  Animation<EdgeInsets> padding;
  Animation<Color> color;

  Widget _buildAnimation(BuildContext context, Widget child) {
    return Container(
      alignment: Alignment.bottomCenter,
      padding:padding.value ,
      child: Container(
        color: color.value,
        width: 50.0,
        height: height.value,
      ),
    );
  }

  @override
  Widget build(BuildContext context) {
    return AnimatedBuilder(
      builder: _buildAnimation,
      animation: controller,
    );
  }
}
```

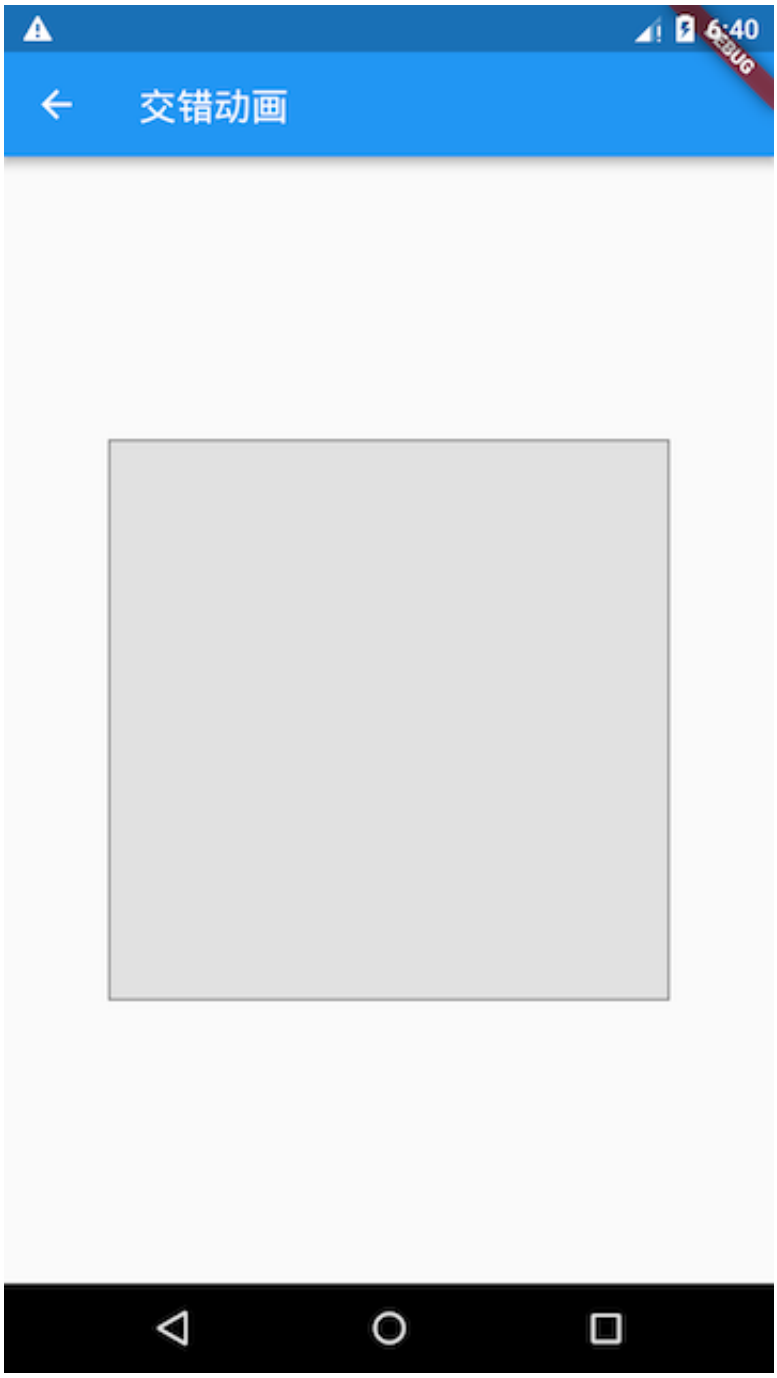
StaggerAnimation中定义了三个动画，分别是对Container的height、color、padding属性设置的动画，然后通过Interval来为每个动画指定在整个动画过程的起始点和终点。

下面我们来实现启动动画的路由：

```
class StaggerDemo extends StatefulWidget {
  @override
  _StaggerDemoState createState() => _StaggerDemoState();
}

class _StaggerDemoState extends State<StaggerDemo> with
TickerProviderStateMixin {
  AnimationController _controller;

  @override
  void initState() {
    super.initState();
```

```
    _controller = AnimationController(
        duration: const Duration(milliseconds: 2000),
        vsync: this
    );
  }


  Future<Null> _playAnimation() async {
    try {
      //先正向执行动画
      await _controller.forward().orCancel;
      //再反向执行动画
      await _controller.reverse().orCancel;
    } on TickerCanceled {
      // the animation got canceled, probably because we were
disposed
    }
  }

  @override
  Widget build(BuildContext context) {
    return  GestureDetector(
      behavior: HitTestBehavior.opaque,
      onTap: () {
        _playAnimation();
      },
      child: Center(
        child: Container(
          width: 300.0,
          height: 300.0,
          decoration: BoxDecoration(
            color: Colors.black.withOpacity(0.1),
            border: Border.all(
              color:  Colors.black.withOpacity(0.5),
            ),
          ),
          //调用我们定义的交错动画Widget
          child: StaggerAnimation(
              controller: _controller
          ),
        ),
      ),
    );
  }
}
```

执行效果如下，点击灰色矩形，就可以看到整个动画效果：


6:40
DEBUG
交错动画