# 5.1 动画

动画是现代用户体验中非常重要的一个部分，`Animated`库就是用来创造流畅、强大、并且易于构建

和维护的动画。

最简单的工作流程就是创建一个`Animated.Value`，把它绑定到组件的一个或多个样式属性上。然后

可以通过动画驱动它，譬如`Animated.timing`，或者通过`Animated.event`把它关联到一个手势上，譬

如拖动或者滑动操作。除了样式，`Animated.value`还可以绑定到 props 上，并且一样可以被插值。这

里有一个简单的例子，一个容器视图会在加载的时候淡入显示：

```
class FadeInView extends React.Component {
   constructor(props) {
     super(props);
     this.state = {
       fadeAnim: new Animated.Value(0), // init opacity 0
     };
   }
   componentDidMount() {
     Animated.timing(          // Uses easing functions
       this.state.fadeAnim,    // The value to drive
       {toValue: 1}            // Configuration
     ).start();                // Don't forget start!
```

```
  }
  render() {
    return (
      <Animated.View          // Special animatable View
        style={{opacity: this.state.fadeAnim}}> // Binds
        {this.props.children}
      </Animated.View>
    );
  }
}
```

注意只有声明为可动画化的组件才能被关联动画。`View`、`Text`，还有`Image`都是可动画化的。如果你

想让自定义组件可动画化，可以用`createAnimatedComponent`。这些特殊的组件里面用了一些黑魔

法，来把动画数值绑定到属性上，然后在每帧去执行原生更新，来避免每次 render 和同步过程的开

销。他们还处理了在节点卸载时的清理工作以确保使用安全。

动画具备很强的可配置性。自定义或者预定义的过渡函数、延迟、时间、衰减比例、刚度等等。取

决于动画类型的不同，你还可以配置更多的参数。

一个`Animated.Value`可以驱动任意数量的属性，并且每个属性可以配置一个不同的插值函数。插值

函数把一个输入的范围映射到输出的范围，通常我们用线性插值，不过你也可以使用其他的过渡函

数。默认情况下，当输入超出范围时，它也会对应的进行转换，不过你也可以把输出约束到范围之

内。

举个例子，你可能希望你的 `Animated.Value` 从 0 变化到 1 时，把组件的位置从 150px 移动到 0px，不透

明度从 0 到 1。可以通过以下的方法修改 style 属性来实现：

```
  style={{
    opacity: this.state.fadeAnim, // Binds directly
    opacity: transform: [{
      translateY: this.state.fadeAnim.interpolate({
        inputRange: [0, 1],
        outputRange: [150, 0] // 0 : 150, 0.5 : 75, 1 : 0
      }),
    }],
  }}>
```

动画还可以被更复杂地组合，通过一些辅助函数例如 `sequence` 或者 `parallel`（它们分别用于先后执

行多个动画和同时执行多个动画），而且还可以通过把 `toValue` 设置为另一个 `Animated.Value` 来产生

一个动画序列。

`Animated.ValueXY` 则用来处理一些 2D 动画（如滑动）。还有一些辅助功能如 `setOffset` 和 `getLayout`

来帮助实现一些常见的交互效果，譬如拖放操作(Drag and drop)。

你可以在 `AnimationExample.js` 中找到更复杂的例子。你还可以看看 `Gratuitous Animation App`,

以及左侧导航栏的动画指南文档。

注意 `Animated` 模块被设计为可完全序列化的，这样动画可以脱离 JavaScript 事件循环，以一种高性

能的方式运行。这可能会导致 API 看起来比较难懂，与一个完全同步的动画系统相比稍微有一些奇

怪。`Animated.Value.addListener` 可以帮助你解决一些相关限制，不过使用它的时候需要小心，因

为将来的版本中它可能会牵扯到性能问题。

## 方法

static **decay**(value, config)

推动一个值以一个初始的速度和一个衰减系数逐渐变为 0。

static **timing**(value, config)

推动一个值按照一个过渡曲线而随时间变化。Easing 模块定义了一大堆曲线，你也可以使用你自己

的函数。

static **spring**(value, config)

产生一个基于 Rebound 和 Origami 实现的 Spring 动画。它会在 toValue 值更新的同时跟踪当前的速度

状态，以确保动画连贯。可以链式调用。

static **add**(a, b)

将两个动画值相加计算，创建一个新的动画值。

static **divide**(a, b)

将两个动画值相除计算(a/b)，创建一个新的动画值。

static **multiply**(a, b)

将两个动画值相乘计算，创建一个新的动画值。

static **modulo**(a, modulus)

创建一个新的动画值，它是所提供的动画值的（非负数）模数

static **diffClamp**(a, min, max)

创建一个新的动画值，它是最大值和最小值之间的值，使用和上次值之间的差值，即使这个值远离

边界，该值开始再次接近时，它也会开始改变。`value = clamp(value + diff, min, max)`

这在滚动事件中是非常有用的，举个例子：当向上滚动显示 navbar，向下滚动隐藏。

static **delay**(time)

延迟多长时间后开始动画

static **sequence**(animations)

按顺序执行一个动画数组里的动画，等待一个完成后再执行下一个。如果当前的动画被中止，后面

的动画则不会继续执行。

static **parallel**(animations, config?)

同时开始一个动画数组里的全部动画。默认情况下，如果有任何一个动画停止了，其余的也会被停

止。你可以通过 `stopTogether` 选项来改变这个效果。

static **stagger**(time, animations)

一个动画数组，里面的动画有可能会同时执行（重叠），不过会以指定的延迟来开始。用来制作拖

尾效果非常合适。

static **event**(argMapping, config?)

接受一个映射的数组，对应的解开每个值，然后调用所有对应的输出的 `setValue` 方法。例如：

```
 onScroll={Animated.event(
   [{nativeEvent: {contentOffset: {x: this._scrollX}}}]>
   {listener},              // Optional async listener
)
...
onPanResponderMove: Animated.event([
  null,                // raw event arg ignored
  {dx: this._panX},    // gestureState arg
]),
```

static **createAnimatedComponent**(Component)

使得任何一个 React 组件支持动画。用它来创建 `Animated.View` 等等。

## 属性

**Value**： AnimatedValue

表示一个数值的类，用于驱动动画。通常用 `new Animated.Value(0)`; 来初始化。

**ValueXY**： AnimatedValueXY

表示一个 2D 值的类，用来驱动 2D 动画，例如拖动操作等。

# class AnimatedValue

用于驱动动画的标准值。一个 `Animated.Value` 可以用一种同步的方式驱动多个属性，但同时只能被

一个行为所驱动。启用一个新的行为（譬如开始一个新的动画，或者运行 `setValue`）会停止任何之

前的动作。

## 方法

**constructor**(value)

**setValue**(value)

直接设置它的值。这个会停止任何正在进行的动画，然后更新所有绑定的属性

**setOffset**(offset)

设置一个相对值，不论接下来的值是由 `setValue`、一个动画，还是 `Animated.event` 产生的，都会加

上这个值。常用来在拖动操作一开始的时候用来记录一个修正值（譬如当前手指位置和 View 位

置）。

**flattenOffset**(0)

把当前的相对值合并到值里，并且将相对值设为 0。最终输出的值不会变化。常在拖动操作结束后

调用。

**addListener**(callback)

添加一个异步监听函数，这样你就可以监听动画值的变更。这有时候很有用，因为你没办法同步的

读取动画的当前值，因为有时候动画会在原生层次运行。

**removeListener**(id)

**removeAllListeners**(0)

**stopAnimation**(callback？)

停止任何正在运行的动画或跟踪值。`callback`会被调用，参数是动画结束后的最终值，这个值可能

会用于同步更新状态与动画位置。

**interpolate**(config)

在更新属性之前对值进行插值。譬如：把 0-1 映射到 0-10。

**animate**(animation, callback)

一般仅供内部使用。不过有可能一个自定义的动画类会用到此方法。

**stopTracking**(0)

仅供内部使用。

**track**(tracking)

仅供内部使用。

# class AnimatedValueXY

用来驱动 2D 动画的 2D 值，譬如滑动操作等。API 和普通的 `Animated.Value` 几乎一样，只不过是个复

合结构。它实际上包含两个普通的 `Animated.Value`。

```
class DraggableView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      pan: new  Animated.ValueXY(), // inits to zero
    };
      this.state.panResponder = PanResponder.create({
      onStartShouldSetPanResponder: () => true,
      onPanResponderMove: Animated.event([null, {
        dx: this.state.pan.x, // x,y are Animated.Value
        dy: this.state.pan.y,
      }]),
      onPanResponderRelease: () => {
        Animated.spring(
          this.state.pan, // Auto-multiplexed
          {toValue: {x: 0, y: 0}} // Back to zero
        ).start();(
      },
    });
  }
  render() {
    return (
      <Animated.View
        {...this.state.panResponder.panHandlers}
```

```
          style={this.state.pan.getLayout()}>
          {this.props.children}
        </Animated.View>
      );
    }
  }
```

## 方法

**constructor**(valueIn?)

**setValue**(value)

**setOffset**(offset)

**flattenOffset**(0)

**stopAnimation**(callback?)

**addListener**(callback)

**removeListener**(id)

**getLayout**(0)

将一个`{x, y}`组合转换为`{left, top}`以用于样式。例如：

```
style={this.state.anim.getLayout()}
```

**getTranslateTransform**(0)

将一个`{x, y}`组合转换为一个可用的位移变换，例如：

```
style={{
  transform: this.state.anim.getTranslateTransform()
}}
```