

# 14 | Reanimated: 如何让动画变得更流畅?

2022-04-29 蒋宏伟

《React Native 新架构实战课》

课程介绍 >



讲述：蒋宏伟

时长 25:52 大小 23.70M



你好，我是蒋宏伟。

今天我们来聊一聊 **React Native** 中动画的原理。在开始之前，我想请你思考一下：动画的本质到底是什么？

你可能知道，与真实世界中连续运动的事物不同，我们在手机、电脑、电影院的屏幕中看到的动画，实际是由一张张快速切换图片组成的。看动画时，我们的眼睛接收到的是一张张并不连续的静态图片，但我们的大脑把这些不连续的图片“想象”成了一系列连续事件，这就是动画的基本原理。

而手机动画要想流畅，一般而言需要保证每 1 秒渲染 60 帧的速度。这里的每一帧都是一张静态图片，也就是说 1 秒钟需要渲染出 60 个静态图片。这也意味着手机处理每一帧动画的耗时，需要保证在 16.6ms ( $=1000/60$ ) 以内，如果处理一帧的耗时超过 16.6ms，就会掉帧。掉帧多了，我们的大脑就会感觉到动画中的不连续性，也就是常说的卡顿。

动画对渲染性能的要求很高。理论上，你可以使用 `setInterval` 每 16.6ms 执行一次 `setState` 改变状态，渲染新的视图，来实现动画。但实际上，`setState` 是一种耗时比较长的更新页面的方法，特别是在复杂页面、复杂交互的情况下，`setInterval + setState` 的方案并不适合用来实现动画。

所以，为了保障动画的流畅性，在涉及动画的业务场景中，**我们还需要引入动画库。**

在上一讲中，我给你介绍了 **React Native** 中常用的三种动画工具，包括：适合轻量级动画场景使用的 **React Native** 自带的 [🔗 Animated](#) 动画；适合无交互场景的、能找 UI 设计师帮忙自动生成的 [🔗 Lottie](#) 动画；以及我今天重点要和你聊的、适用于可交互场景的 [🔗 Reanimated](#) 动画。

## 初学 Reanimated

**Reanimated** 的名字来源于它的那句口号：

React Native's Animated library reimplemented.

**Reanimated** 名字中的 **Re** 就是 **Reimplemented** 重新实现的意思，**Animated** 代表就是 **React Native** 自带的动画库 **Animated**，加起来就是重新实现的 **Animated** 动画库的意思。它的潜台词好像就是：如果你觉得 **React Native** 自带的 **Animated** 动画库不好用，就来试试我吧，我把 **React Native** 官方的动画库给重新实现了。

我们先用“**切换宽度的动画**”的例子，看看 **Reanimated** 到底应该如何使用。

这个例子是这样的：现在你有一个视图和一个按钮，视图的高度是固定的，视图的宽度可以用动画来控制，你每点一下切换宽度的文字，视图宽度都会随机改变。示意图如下：



你可以看到，在页面中蓝色视图初始化的宽度是 10 像素，当你点击切换宽度的文字后，蓝色视图的宽度会在 0~350 像素之间随机变化。因为是动画，所以蓝色视图的宽度并不是一下就变宽的，而是连续改变的。在 1s 内，先增长几像素，然后再增长几像素，依次类推直到目标长度，因为刷新的帧率很快，因此人的肉眼看起来就是宽度就是连续变化的。

那我们怎么用 Reanimated 实现这个动画效果呢？

别急，为了帮你更好地吃透这个新知识，我们引入之前学过的 **State**，对比着来学习。那么，使用 **Reanimated** 实现视图“动画”和使用 **State** 实现视图“变化”有什么相似之处呢？你可以看下它们的更新步骤的对比：

	State (旧知识)	Reanimated (新知识)
状态 vs 共享值	<code>[状态, setState] = useState(10);</code>	<code>共享值 = useSharedValue(10);</code>
状态衍生值 vs 衍生样式值	<code>状态衍生值 = {width: randomWidth}</code>	<code>衍生样式值 = useAnimatedStyle(() =&gt;({     width: 共享值.value, }));</code>
组件 vs 动画组件	<code>&lt;View style={状态衍生值} /&gt;</code>	<code>&lt;Animated.View style={衍生样式值} /&gt;</code>
更新函数 vs 直接赋值	<code>更新函数(Math.random() * 350)</code>	<code>共享值.value = Math.random() * 350;</code>

你可以看到，无论是用 **State** 更新页面，还是用 **Reanimated** 更新动画，都需要 4 步。具体它们有什么相似之处呢？我们直接来分析 **Reanimated** 中的这 4 个概念：

第一个概念：共享值（**SharedValue**）。**Reanimated** 中共享值这个概念类似于 **React** 中的状态 **State**，我们简单对比下它们各自的代码，先看看 **State** 的：

 复制代码

```
1 // State 示例代码
2 import { useState } from 'react';
3 const [randomWidth, setRandomWidth] = useState(10);
4 // randomWidth === 10
```

在 **State** 示例代码中，驱动视图变化的最初因子是状态。用于初始化状态的钩子函数 **useState** 是从 **react** 中引入的，然后在组件中使用 **useState** 创建出一个随机宽度状态 **randomWidth**，以及一个改变该状态的函数 **setRandomWidth**。其中，初始化出来的 **randomWidth** 是一个默认赋值数字 **10**。

接着我们看看 **Reanimated** 的代码：

 复制代码

```
1 // Reanimated 示例代码
2 import { useSharedValue } from 'react-native-reanimated';
3 const randomWidth = useSharedValue(10);
4 // randomWidth.value === 10
```

在 **Reanimated** 的示例代码中，驱动动画的最初因子是共享值（**ShareValue**）。用于初始化共享值的钩子函数 **useSharedValue** 是从 **react-native-reanimated** 中引入的。然后使用 **useSharedValue** 创建出一个对象 **randomWidth**，**randomWidth** 的 **value** 属性是一个默认赋值数字 **10**。

第二个概念：衍生值（**DerivedValue**）。**Reanimated** 的衍生值（**DerivedValue**）这个概念类似于 **React** 中的状态衍生值。我们同样先来看 **State** 的示例代码：

 复制代码

```
1 // State 示例代码
2 const style = {
```

```
3   width: randomWidth
4 }
```

在 **State** 示例代码中，你可以在组件函数中的任意位置直接使用状态 `randomWidth`，或者将状态 `randomWidth` 封装到样式对象 `style` 中。

然后是 **Reanimated** 的示例代码：

 复制代码

```
1 // Reanimated 示例代码
2 import { useAnimatedStyle } from 'react-native-reanimated';
3
4 // 错误示范
5 const style = {
6   width: randomWidth.value
7 }
8
9 // 正确示范
10 const style = useAnimatedStyle(() => {
11   return {
12     width: randomWidth.value,
13   };
14 });
```

但在 **Reanimated** 示例代码中，如果你直接将 `const style = {width: randomWidth.value}` 组成的样式对象赋值给 **JSX** 元素，控制视图宽度改变的动画是不生效的。这是 **Reanimated** 驱动动画和 **State** 驱动视图的机制不一样导致的。

这时你需要从 **react-native-reanimated** 中引入钩子函数 `useAnimatedStyle`，这个钩子函数是专门用来处理动画样式的衍生值的，它的第一个入参函数的返回值就是动画组件的样式值。

第三个概念，动画组件（**AnimatedComponent**）。**Reanimated** 的动画组件和 **React/React Native** 中的组件（**Component**）概念是类似的。我们同样先看 **State** 示例代码：

 复制代码

```
1 // State 示例代码
2 import { View } from 'react-native';
3
4 <View style={[{ width: 100}, style]} />
```

在 **State** 示例代码中，你要从 **react-native** 库中引入 **View** 组件，并将组件 **View** 实例化为 **JSX** 元素。

然后再看 **Reanimated** 的示例代码：

 复制代码

```
1 // Reanimated 示例代码
2 import Animated from 'react-native-reanimated';
3
4 <Animated.View style={[{ width: 100}, style]} />
```

你可以看到，在 **Reanimated** 示例代码中，你需要从 **react-native-reanimated** 引入 **Animated** 对象，在该对象上挂了常用的 **react-native** 组件，比如示例代码中的 **Animated.View**，还有 **Animated.Text**、**Animated.FlatList** 等等。

这些由 **Reanimated** 包装好的动画组件，比如 **Animated.View** 等等，使用方式和 **View** 基本类似。不同的是共享值（**ShareValue**）和衍生值（**DerivedValue**）是专门给动画组件（**AnimatedComponent**）用的，普通组件（**Component**）用不了。

第四个概念，更新共享值。**Reanimated** 的更新共享值的方式和 **React/React Native** 更新状态的方式方式是不一样的。

**State** 的示例代码如下：

 复制代码

```
1 // State 示例代码
2 const [randomWidth, setRandomWidth] = useState(10);
3 setRandomWidth(Math.random() * 350)
```

在 **State** 示例代码中，你通过钩子函数 **useState** 生成的状态更新函数 **setRandomWidth** 来更新状态的。

然后是 **Reanimated** 的示例代码：

 复制代码

```
1 // Reanimated 示例代码
2
3 const randomWidth = useSharedValue(10);
4 // 不带动画的更新
5 randomWidth.value = Math.random() * 350;
6 // 带动画的更新
7 randomWidth.value = withTiming(Math.random() * 350);
```

我们可以看到，在 **Reanimated** 示例代码中，没有共享值的更新函数，它只生成了一个共享值对象，其真正的值是挂在 **value** 属性下的。你可以直接通过等号 **=** 把最新的视图宽度 **Math.random() \* 350** 赋值给 **randomWidth.value**。

事实上，**Reanimated** 有两种更新方式，一种是不带动画曲线的更新方式，另一种是带**动画曲线**的更新方式。

你直接把视图宽度 **Math.random() \* 350** 赋值给 **randomWidth.value**，就是通过指定一个最终共享值的方式进行更新的，比如从 **10** 像素宽度直接变为 **100** 像素宽度。这种更新方式是一步到位的，没有动画曲线。

真正的动画是从 **10** 像素宽度，增长到 **11** 像素，然后增长到 **12** 像素，以此类推，通过连续的方式增长到 **100** 像素宽度的。具体地说，控制每一帧增长多少像素、减少多少像素，是通过类似 **withTiming** 的动画曲线实现的。**withTiming** 动画曲线的意思是，启动一个基于时间的动画，在每个单位时间内增长或减少的像素是相等的。

使用 **withTiming(100)** 更新共享值时，就会启动基于时间的动画曲线，其默认的持续时间是 **300ms**。理论上，在这 **300ms** 内，视图的宽度会从 **10** 像素开始，以每一帧增加一个固定的宽度的速度，增加到 **100** 像素。

切换宽度动画的完整示例代码如下：

```
1 import Animated, {
2   useSharedValue,
3   withTiming,
4   useAnimatedStyle,
5   Easing,
6 } from 'react-native-reanimated';
7 import { View, Button } from 'react-native';
8 import React from 'react';
9
```

 复制代码



```

10 function AnimatedStyleUpdateExample(): React.ReactElement {
11   const randomWidth = useSharedValue(10);
12
13   const style = useAnimatedStyle(() => {
14     console.log('==Animated==')
15     return {
16       width: withTiming(randomWidth.value),
17     };
18   });
19
20   console.log('==render==')
21
22   return (
23     <View>
24       <Animated.View
25         style={[
26           { width: 100, height: 30, backgroundColor: 'cornflowerblue' },
27           style,
28         ]}
29       />
30       <Button
31         title="切换宽度"
32         onPress={() => {
33           randomWidth.value = Math.random() * 350;
34         }}
35       />
36     </View>
37   );
38 }

```

从上面的代码你可以看出，使用 **Reanimated** 更新动画的方式和使用 **State** 更新页面的方式，有很多相似之处。

其中，还有一点是你需要注意的，就是 **Reanimated** 和 **State** 的更新机制并不一样。

在使用 **Reanimated** 改变共享值触发动画更新时，只会触发示例代码中 **useAnimatedStyle** 的入参函数的执行，而不会触发 **AnimatedStyleUpdateExample** 组件函数的执行。也就是说，动画更新是不会打印“==render==”日志的，只会打印“==Animated==”日志。

但整体上讲，二者都是通过数据来驱动视图变化。**Reanimated** 是专门用来处理动画形式的视图更新的，而 **State** 是专门用来处理组件、页面渲染的视图更新的。

## Reanimated 的原理



关于 **Reanimated** 的入门概念，我们先介绍到这里，相信通过 **Reanimated** 和 **State** 的类比学习，你已经能把 **Reanimated** 用起来了。

但这种学习方式难免可能让你对概念掌握得不够准确，甚至出现一些理解偏差。所以接下来我们要再进一步，了解 **Reanimated** 工作原理，把其中的基础概念弄扎实了，把一些理解有误的地方纠正回来。

在开发过程中，我们的动画代码和状态代码都是用 **JavaScript** 写在同一个文件中的，你可能会认为你写的动画部分的 **JavaScript** 和状态部分的 **JavaScript** 都是运行在同一个线程中的，但其实并不是这样的。

听到这个结论，你可能会很惊讶：**为什么动画代码和状态代码都放在同一个 JavaScript 文件中，但动画部分代码却由另一个线程来执行呢？**

答案就是：把动画代码放到 **UI** 主线程来执行性能会更好，动画不容易卡顿。

你可能知道，**React Native** 有两个常用的线程：一个是 **React Native** 的 **JavaScript** 线程，另一个是 **UI** 主线程。

一方面，**JavaScript** 线程和 **UI** 主线程是异步通信的，这也意味着，如果是由 **JavaScript** 线程发起动画的执行，**UI** 线程并不能同步地收到该命令并且立刻执行，**UI** 线程至少要处理完成当前一帧的渲染任务后，才会执行 **JavaScript** 线程的动画命令。也就是说异步通讯会导致动画至少延迟 **1** 帧。

另一方面，**JavaScript** 线程处理的事件很多，包括所有的业务逻辑、**React Diff**、事件响应等等，容易抢占动画的执行资源。

**正因为 JavaScript 线程非常繁忙，所以如果我们把动画代码交由 JavaScript 线程执行，它就会更加繁忙。**前面我们也讲过，处理 **1** 帧动画的耗时需要控制在 **16.6ms** 以内，如果超过 **16.6ms** 就会导致动画掉帧，掉帧严重的时候，用户就会感觉到卡顿。

好，既然动画部分的 **JavaScript** 代码放在 **JavaScript** 线程中执行，存在至少 **1** 帧的延迟，并且容易导致卡顿。那我们的解决方案是什么呢？

有两种思路。

**第一种思路就是 React Native 自带的 Animated 动画库用的思路。**它是在组件初始化时，把动画的初始值、动画的形式、动画的结束值等配置都传给 UI 主线程。开发者有个开启 UI 主线程执行动画任务的开关 `useNativeDriver`，当开发者开启 `useNativeDriver` 这个开关后，动画就是在 UI 主线程执行了。

但是 **Animated** 动画库的缺陷也很明显，它传给 UI 主线程的是动画配置。配置只是单纯的数据，它不具备图灵完备的特性，不能配置复杂的逻辑。所以 **React Native** 官方也指出了：**Animated** 不能用来改变元素宽度、高度等布局属性，不能处理除了 **ScrollView** 组件的 `onScroll` 事件外的其他手势事件。

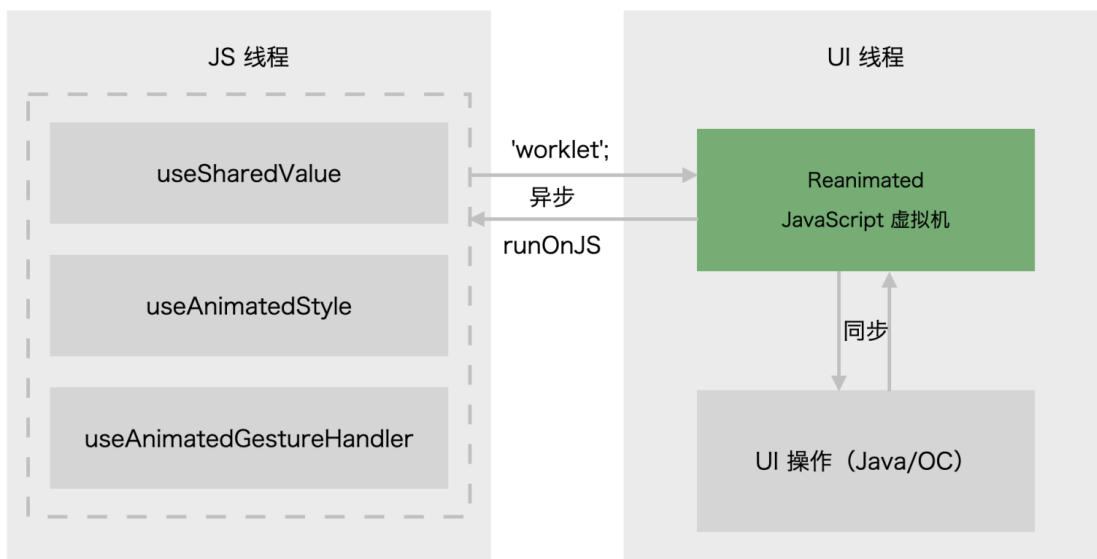
换句话说，在保障性能的前提下，简单的动画、无交互的动画，我们可以用自带的 **Animated** 动画库来处理；如果是逻辑稍微复杂点的、带交互的动画，自带的 **Animated** 动画库就干不了。

那 **Reanimated** 动画库能够处理复杂动画、有交互的动画吗？它是怎么做到的呢？

可以。**Reanimated** 动画库采用了另一种思路，它把动画相关的 **JavaScript** 函数及其上下文传给了 UI 主线程。不过，UI 主线程并没有能运行 **JavaScript** 函数的虚拟机，于是 **Reanimated** 又创建了一个 **JavaScript** 虚拟机来运行传过来的 **JavaScript** 函数。

换句话说，在使用 **Reanimated** 之前，**React Native** 只会在 **JavaScript** 线程创建一个 **JavaScript** 虚拟机，来运行 **JavaScript** 代码。而使用 **Reanimated** 之后，**Reanimated** 会在 UI 主线程中创建另一个 **JavaScript** 虚拟机来运行动画部分相关的代码。

我给你画了一张 **Reanimated** 的原理图，你可以看下，加深一下理解：



在这张原理图中，你会看到有两个线程：**JavaScript 线程**和 **UI 线程**。

在 **JavaScript 线程**中包括了三个动画相关的函数或值，[🔗 useSharedValue](#)（其底层会调用 [🔗 cancelAnimation](#)）、[🔗 useAnimatedStyle](#) 和 [🔗 useAnimatedGestureHandler](#)。这三部分的代码会在其底层，将相关的回调函数标记为“worklet”，被标记的“worklet”函数或值会被放在一个由 **Reanimated** 创建的 **JavaScript 虚拟机**中执行。

而这个由 **Reanimated** 创建的 **JavaScript 虚拟机**，会在 **UI 线程**中执行传过来的“worklet”函数，并且执行的函数还可以同步地操作 **UI**。

**那 JavaScript 线程和 UI 主线程是怎么配合工作的呢？**

我们结合宽度切换动画的代码示例，来看下原理图中的各个部分是怎么运行的：

复制代码

```
1 const randomWidth = useSharedValue(10);
2 const style = useAnimatedStyle(()=>({width: randomWidth.value}));
3
4 // 运行在 JS 线程的点击事件
5 const handlePress = () => {
6   randomWidth.value = Math.random() * 350;
7 }
```

在 JavaScript 代码初始化时，先执行的是 `useSharedValue` 函数，它会生成一个共享值对象 `randomWidth`。`randomWidth` 对象上挂了一个 `value` 属性。这个 `value` 属性既可以在 JavaScript 线程获取和修改，也可以在 UI 线程中的 JavaScript 虚拟机中获取和修改。

然后执行的是 `useAnimatedStyle` 函数。`useAnimatedStyle` 的入参也是一个函数，`Reanimated` 会将 `useAnimatedStyle` 的入参函数和与它相关的上下文都放到 UI 线程中的 JavaScript 虚拟机中。

最后就是处理点击事件了。但由于这一讲，我们还没有接触到手势 `React Native Gesture Handler`，所以我先用普通的事件处理函数 `handlePress` 来处理点击事件了。使用普通的事件处理函数有一个弊端：它是在 JavaScript 线程中触发的。所以，如果要生成的是某种对事件的响应速度有要求的动画，比如拖拽类的动画，就容易导致卡顿。

解决方案就是把处理点击事件的函数，也放到 UI 线程中，让独立的 JavaScript 虚拟机来执行。这时候，我们就需要用 `useAnimatedGestureHandler` 将其包装起来，示例代码如下：

 复制代码

```
1 // 运行 UI 线程的点击事件
2 const handleAnimatedPress = useAnimatedGestureHandler({
3   onEnd: (_) => {
4     randomWidth.value = Math.random() * 350;
5   },
6 })
```

不过，代码中的动画手势处理函数 `handleAnimatedPress`，需要结合 `React Native Gesture Handler` 的手势组件一起使用，具体如何结合使用，我们下一讲再讲。

在这一讲，你只需要知道，我们使用 `Reanimated` 生成动画的时候，只有在 JavaScript 代码初始化时，相关的动画代码会在 JavaScript 线程执行。初始化完成后，`useSharedValue` 生成的共享值是在 JavaScript 线程和 UI 线程的 JavaScript 虚拟机中共享的。

并且，在初始化完成后，`useAnimatedStyle` 的样式入参函数和 `useAnimatedGestureHandler` 的手势函数，以及相关的上下文都会放到 UI 线程中的 JavaScript 虚拟机中去。

简而言之，`Reanimated` 动画性能好的原因就在于，`React Native` 的 JavaScript 线程是性能瓶颈点，而 UI 线程不是，在 `Reanimated` 真正执行动画时，你已经把所有与动画相关

JavaScript 函数都放到了 UI 线程中独立的 JavaScript 虚拟机中了，并不会和 JavaScript 线程抢占硬件资源，因此 Reanimated 执行动画的性能会更好。

## 附加材料

1. 官方的几个入门视频都是 1 个小时以上的，我为你选了一个 17 分钟入门的视频 [🔗](#)  
《Introduction to React Native Reanimated 2》，可以帮你快速入门。
2. 入门之后，建议你再看看官方的文章，它会帮你更好的理解 Reanimated2 的 [🔗原理](#)。
3. 除了 [🔗useAnimatedStyle](#)、[🔗useAnimatedStyle](#) 和 [🔗useAnimatedGestureHandler](#) 之外，还有一些钩子函数也可以把它的入参函数放到 UI 线程中执行，包括 [🔗useDerivedValue](#)、[🔗useAnimatedScrollHandler](#)、[🔗useAnimatedReaction](#) 和 [🔗useAnimatedProps](#)，当然你也可以通过把 [🔗“worklet”](#) 字面量放到函数顶部，这样 Reanimated 就会把该函数放到 UI 线程中执行了。
4. 常见的动画曲线有 [🔗withTiming](#)、[🔗withSpring](#)、[🔗withDecay](#) 和 [🔗withDelay](#)，甚至你还可以使用 [🔗Easing.bezier](#) 自定义动画曲线。
5. 这节课中的 Demo，我也放到了 [🔗GitHub](#) 上。

## 总结

对于交互类的动画，我们有两种选择，一种是 React Native 自带的 Animated 动画库，另一种是社区的 Reanimated 动画库。

为了实现流畅的动画效果，二者都把原本来 JavaScript 线程执行的动画任务，放到了 UI 线程中来执行。不同的是，官方动画库采用的是传递动画配置的形式，社区动画库采用的是传递 JavaScript 函数的形式，因此 Reanimated 动画库的应用场景更加广泛。

在今天这一讲中，我也帮你也搭起了两座知识的桥：一座桥是连接的是 Reanimated 和 State 两个知识点，这座桥的目的是帮你快速学习 Reanimated；另一座桥连接的是 Reanimated 和 React Native 架构，这座桥的目的是帮你弄清楚 Reanimated 的底层原理。

除了学习 Reanimated 本身知识外，我也希望你能掌握这种“搭桥修路”式的学习方法。你掌握的知识点越多，你搭的桥、修的路就越多，下次你碰到新的知识时，你学习的速度也就越快，进步也就越快。


## 作业

- 这节课，我们通过点击事件来设置视图的宽度。请你使用 `Animated.ScrollView` 和 `useAnimatedScrollHandler` 实现通过滚动控制视图大小的动画。
- 能说说你在工作中的哪些场景中用到了动画吗？希望你能和我们分享一下你使用动画的心得。

欢迎在留言区写下你的想法。我是蒋宏伟，咱们下节课见。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 生态：React Native Awesome

下一篇 15 | Gesture（上）：如何实现一个拖拽动效？

## 精选留言 (1)

 写留言



worm

2022-05-11

老师您好，Reanimated 是如何把 JS 动画代码放到 UI 主线程的 JS 虚拟机中的呢？这部分是 C++ 实现的吧？有没有这部分的讲解材料或者实现的源码位置？

作者回复: 简单的讲，执行 JavaScript 代码（字符串）依赖的是 JavaScript 引擎，而 JavaScript 函数调用时，可以把 aWorklet 函数和其上下文告诉 JavaScript 引擎，然后再在 UI 线程单独开一个同步的上下文执行。

官方视频如下：<https://swmansion.com/academy/webinars/krzysztof-magiera-reanimated-2/>

官方伪代码如下：

---JavaScript---

```
const CHANCE = 0.6;
```

```
function notAWorklet(hotDogOrNot){  
  lemmeUpdateSomeReeduxx({ isHotDog: hotDogOrNot });  
}
```

```
function aWorklet(thing){  
  'worklet';  
  const decide = Math.random() > CHANCE; notAWorklet(decide);  
}
```

---C++---

```
aWorklet.asString ='function aWorklet(thing){ const decide = Math.random() >0'  
aWorklet.closure ={ CHANCE,notAWorklet }
```

