

22 | 自定义组件：如何满足业务的个性化需求？

2022-05-18 况众文 朴惠姝

《React Native 新架构实战课》

课程介绍 >



讲述：蒋宏伟

时长 23:41 大小 21.70M



你好，我是众文，这一讲还是由我和惠姝来讲解。

上一讲，我们讲了如何构建混合应用。当环境配置、载体页、调试打包都 OK 后，我们就要开始复杂业务的开发了。在实际开发中，除了负责 React Native 框架本身的维护迭代外，另一个重要的工作就是配合前端业务，开发对应的 Native 组件。

那么什么时候用这些自定义的 Native 组件呢？

比如，有时候 App 需要访问平台 API，但 React Native 可能还没有相应的模块包装；或者你需要复用公司内的一些用 Java/OC 写的通用组件，而不是用 JavaScript 重新实现一遍；又或者你需要实现某些高性能的、多线程的代码，譬如图片处理、数据库，或者各种高级扩展等。

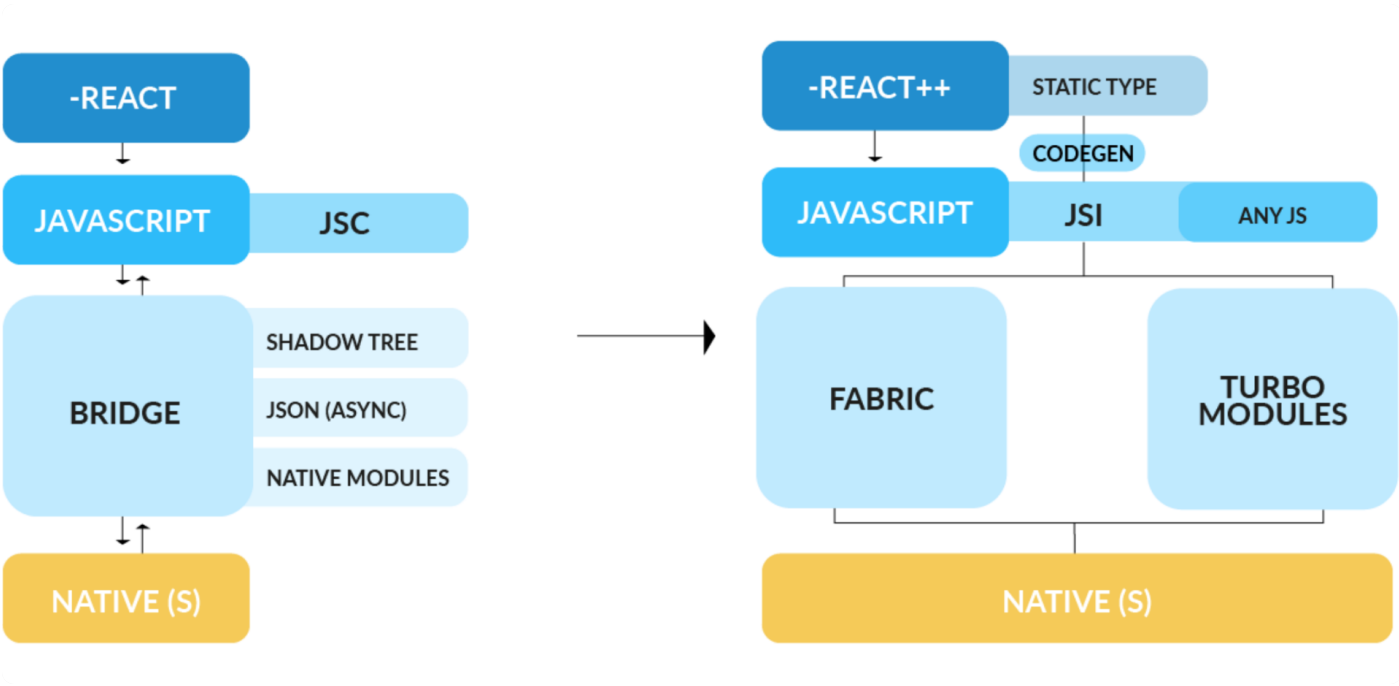
当然，你可以通过官方文档（[🔗 Android](#)/[🔗 iOS](#)），快速访问你的原生模块。但官方文档提供的主要是简单的 Demo 和步骤，在实际开发中，你可能还需要自定义组件的方方面面，包括

新架构定义组件的全流程，以及实际业务中的踩坑指南等。

今天这一讲，我们会先带你补齐组件的相关基础知识，包括组件的生命周期、组件传输数据类型，并以新架构的 TurboModule 和 Fabric 为案例，带你了解自定义组件的方方面面。你也能借此对 React Native 新架构建立起初步认识。接下来让我们先了解下期待已久的 React Native 新架构。

新架构介绍

我们现在先通过下面这张图简单对比下新老架构：



新架构变更点主要在下面这几个方面：

#	说明
通信	JSI 替换 bridge，性能大幅提升，JSI 无需异步序列化与反序列化过程，可实现同步的 JavaScript 和原生 Java 或者 OC 的通讯。同时原有 bridge 数据通讯采用 JSON 和基础类型数据，而 JSI 还支持 JSI Object
JavaScript 引擎	JSI 作为 JSC 之上的抽象，JSI 用来屏蔽 JavaScript 引擎的差异，允许换用不同的 JS 引擎（JSC、V8、Hermes）
Fabric	对标旧框架的 UIManager，FabricUIManager 可以和 C++ 层直接进行通讯，解除了原有的 UIManager 依赖单个 bridge 的问题，有了 JSI 后，以前批量依赖 bridge 的 UI 操作，都可以同步的执行到 C++ 层，性能得到大幅提升，特别是在列表快速滑动、复杂动画交互方面
TurboModules	旧框架 Native Modules 需要在启动时全部初始化，影响启动速度，TurboModules 允许按需加载 Native 模块，并在模块初始化之后直接持有其引用，不再依靠消息通信来调用模块功能
CodeGen	新架构 UI 增加了 C++ 层的 shadow、component 层，而且大部分组件都是基于 JSI，因而开发 UI 组件和 API 的流程更复杂了，要求开发者具有 C++、JNI 的编程能力，为了方便开发者快速开发 Facebook 也提供了 codegen 工具，帮助生成一些自动化的代码



前面也说了，我们一讲将以 TurboModule 和 Fabric 为案例对自定义组件进行讲解。所以你现在需要对新架构有一个初步的认识，特别是要注意，TurboModule 和 Fabric 对比旧版的 Native Module 和 UIManager 有哪些差异和优势。

如果你还想了解新架构的更多信息，你可以参考[🔗 官方文档](#)。这里包括了新架构介绍、如何在 Android、iOS 上开启新架构、如何在 Android、iOS 上开启使用 TurboModule 和 Fabric 等。你也可以根据官方文档试着编译运行新架构。

而且，[🔗 react-native-screens](#)、[🔗 react-native-gesture-handler](#) 等知名 React Native 库的新版都已适配了新架构，感兴趣的话，你可以去了解下。

好的，现在我们进入这一讲的正题，先来了解一下组件的生命周期。

组件的生命周期

组件的生命周期，指的是在组件创建、更新、销毁的过程中伴随的各种各样的函数执行。这些在组件特定的时期被触发的函数，统称为组件的生命周期。

让组件拥有生命周期，我们就可以更好地管理组件的状态、内存，跟随载体页的生命周期做相应的处理。

Android

在 Android 端，一个 Module 组件的生命周期包括：

复制代码

```
1 构造 -> 初始化 -> onHostResume() -> onHostPause() -> onHostDestroy()
```

这几个生命周期的意思是：

构造	组件创建。这里需要注意：旧版本的 NativeModule 是在载体页初始化 React Native 环境时，全部进行初始化创建的；而 TurboModule 则不同，TurboModule 采用懒加载模式，在 JavaScript 运行时第一次 import 该TurboModule 时进行创建
initialize	初始化
onHostResume()	载体页 onResume() 时触发，一般是指载体页与用户能进行交互时被执行
onHostPause()	载体页 onPause() 时触发，一般是载体页被弹窗遮挡，或者即将离开前也会触发
onHostDestroy()	载体页销毁时



如果你不熟悉 Android Activity/Fragment 生命周期，那你可以看下 [这篇文章](#)加深下理解。

那么如何让 Native Module 具备生命周期呢？

React Native 给我们提供了一个接口：`com.facebook.react.bridge.LifecycleEventListener`。我们只需要在组件中添加这个接口的注册和取消注册，就可以让组件具备生命周期了。这里要注意，不要忘记在 `onHostDestroy()` 中移除注册，否则会造成内存泄漏。示例代码如下：

```

1 public class TestJavaModule extends ReactContextBaseJavaModule
2 implements LifecycleEventListener, ReactModuleWithSpec, TurboModule {
3     public TestJavaModule(ReactApplicationContext reactContext) {
4         super(reactContext.real());
5         reactContext.addLifecycleEventListener(this);
6     }
7
8     @Override
9     public String getName() {
10         return getClass().getSimpleName();
11     }
12
13     @Override
14     public void onHostResume() {
15     }
16
17     @Override
18     public void onHostPause() {
19     }
20
21     @Override
22     public void onHostDestroy() {
23         getReactApplicationContext().removeLifecycleEventListener(this);
24     }
25 }

```

其实组件的生命周期原理很简单，就是观察者模式。当载体页触发自身的生命周期回调时，调用 `ReactInstanceManager` 的 `onHostXXX()` 方法，`ReactInstanceManager` 进而调用 `ReactContext` 的对应回调。

比如载体页调用 `onResume()` 时，最终会调用 `ReactContext` 的 `onHostResume()`，内部会遍历注册的事件进行回调：

```

1 public void onHostResume(@Nullable Activity activity) {
2     Iterator iterator = this.mLifecycleEventListeners.iterator();
3     while(iterator.hasNext()) {
4         LifecycleEventListener listener = (LifecycleEventListener) iterator.next();
5         // 观察者模式，载体页时调用已注册组件的生命周期回调
6         listener.onHostResume();
7     }
8 }

```

以上就是 Android 端组件生命周期的讲解，我们再来看看 iOS 端。

iOS

iOS 中 NativeModules 组件的创建销毁时机，与 bridge 的创建销毁时机完全一致：

- `alloc`：创建当前组件；
- `dealloc`：销毁当前组件。

创建一个组件 `TestNativeModule`，通过 `RCT_EXPORT_MODULE()` 声明组件，默认会根据类名声明组件名，当然也可以通过在参数中传入其他字符串作为组件的名。

 复制代码

```
1 @implementation TestNativeModule
2 RCT_EXPORT_MODULE()
3
4 - (instancetype)init{
5     self = [super init];
6     return self;
7 }
8 - (void)dealloc{
9     NSLog(@"dealloc");
10 }
```

而 **TurboModule** 组件的生命周期却与 **NativeModule** 不同。**TurboModule** 采用懒加载模式，在 Bridge 创建后页面中第一次 import 当前 **TurboModule**，也就是 JavaScript 端通过 **TurboModuleRegistry.getEnforcing** 方法加载组件时，Native 会创建对应的 **TurboModule** 并进行缓存。如果 JS 端没有加载当前自定义组件，该组件就不会进行初始化。

JS 端加载组件方式如下：

 复制代码

```
1 export default (TurboModuleRegistry.getEnforcing<Spec>(  
2     'TestTurboModule')  
3     : Spec);
```

而 TurboModule 的销毁时机与 Bridge 的销毁时机一致。Bridge 进行销毁时会发送一个 RCTBridgeDidInvalidateModulesNotification 通知，TurboModuleManager 会监听该事件，依次对所有已创建的 TurboModule 进行销毁。示例代码如下：

 复制代码

```
1 - (void)bridgeDidInvalidateModules:(NSNotification *)notification
2 {
3     RCTBridge *bridge = notification.userInfo[@"bridge"];
4     if (bridge != _bridge) {
5         return;
6     }
7     [self _invalidateModules]; // 销毁所有 TurboModules
8 }
```

了解完了组件的生命周期，我们再来看下组件的传输数据类型。在组件运行过程中，Native 与 JavaScript 不可避免地需要进行数据交互，如 JavaScript 调用组件方法传入数据，Native 向 JavaScript 回传结果，而 React Native 也帮我们封装好了对应的数据类型。

组件传输数据类型

在 Native 与 JavaScript 通信的过程中，组件需要获取输入参数、回传结果，对此 React Native 给我们包装了相应的数据类型，方便快速操作，我们通过一个 Demo 来简单了解下。

现在，我们让 JavaScript 端调用 TestModule 的 testMethod 方法，传入参数 type 和 message，接收 native 回传数据：

 复制代码

```
1 NativeModules.TestModule.testMethod({type: 1, message: "fromJS"}, (result)=>{
2     console.info(result);
3 })
4 );
```

然后我们来看 TestModule 在 Android、iOS 侧的实现。先来看 Android 端是怎么做的。

不过，在实现 TestModule 之前，我们需要先了解下 Android 端的组件传输数据类型：

JavaScript 端	Android 端
Bool类型	boolean
double类型	double
int类型	int
String类型	String
Array类型	Array
Object类型	Map



这里你要注意，数字类型有点特殊。因为 JavaScript 不支持 long 64 位长类型，只支持 int (32) 和 double，所以对于长数字，JavaScript 端统一用 double 表示。那么 Android 端如何转换成自己需要的数据类型呢？

我们以 long 为例，可以这样参考 [官方 issue](#) 这样处理：


```

1 double value = readableMap.getDouble(key);
2 try {
3     // 判断是否为 long 的范围：超过了 int 的最大值且为整数
4     if (value > Integer.MAX_VALUE && value % 1 == 0) {
5         long cv = (long) value;
6         // 转换成 long 型返回
7     }
8 } catch (Exception e) {
9     // 异常时，仍使用 double
10 }

```

这段代码中，我们先将 JavaScript 传入的数值统一以双精度浮点数 `double` 来获取。获取完后，判断这个值是否超出了整数的最大值且不为小数，条件符合就将它转换成长整数 `long`，否则还是以 `double` 来返回。

了解完 Android 端的组件数据传输类型后，我们就可以来实现上文中的 `TestModule` 了：

```

1 public class TestModule extends ReactContextBaseJavaModule implements ReactModu
2     public TestModule(ReactApplicationContext reactContext) {
3         super(reactContext.react());
4     }
5
6     @Override
7     public String getName() {
8         return getClass().getSimpleName();
9     }
10
11     @ReactMethod
12     public void testMethod(ReadableMap data, Callback callback) {
13         // 获取 JS 的调用输入参数
14         int type = data.getInt("type");
15         String message = data.getString("message");
16         // 回传数据给 JS
17         WritableMap resultMap = new WritableNativeMap();
18         map.putInt("code", 1);
19         map.putString("message", "success");
20         callback.invoke(resultMap);
21     }
22 }

```

上面代码中，我们定义了 Native 组件 `TestModule`，内部实现了 JavaScript 需要调用的 `testMethod` 方法。此方法包含两个参数：`ReadableMap` 和 `Callback`。`ReadableMap` 为

JavaScript 传入参数的字典，我们可以通过对应的 **key** 获取到 JavaScript 的入参值，而 **Callback** 是在 **Native** 回传数据时需要使用的，后面我们还有对通信方式这部分的讲解，这里我们只需要了解一下就好。

具体实现上，我们是首先获取 JavaScript 调用传入的 **type** 和 **message**，然后再通过 **WritableMap** 写入数据，最后通过 **callback** 回传给 JavaScript。

以上就是 **Android** 端的 **React Native** 读写数据类型，我们再来看下 **iOS** 端。

iOS 端也是一样，在实现前面这个 **demo** 前，我们需要先看下 **iOS** 端支持的传入数据类型：

JavaScript 端	iOS 端
Bool类型	NSNumber对象
double类型	NSNumber对象
String类型	NSString对象
Array类型	NSArray对象
Object类型	NSDictionary对象



那么，iOS 端中是如何实现上文中的 TestModule 的呢？我们可以在 Module 中进行 callback，然后通过 NSArray 来返回，如下：

复制代码

```
1 RCT_EXPORT_METHOD(getValueWithCallback : (RCTResponseSenderBlock)callback){
```

```

2   if (!callback) {
3       return;
4   }
5   callback([ @ "value from callback!" ]);
6 }
7

```

不过，我们这个组件案例，只是演示了 Native 可以通过 callback 向 JavaScript 回传数据。那么除了 callback，React Native 与原生还有什么通信方式呢？

React Native 与原生的通信方式

总体来说，native 向 JavaScript 传递数据的方式分成以下三种：

- **Callback**：由 JavaScript 主导触发，Native 进行回传，一次触发只能传递一次；
- **Promise**：由 JavaScript 主导触发，Native 进行回传，一次触发只能传递一次。Promise 是 ES6 的新特性，类似 RXJava 的链式调用。Promise 有三种状态，分别是 pending (进行时)、resolve (已完成)、reject (已失败)；
- **发送事件**：由 Native 主导触发，可传递多次，类似 Android 的广播和 iOS 的通知中心。

Callback 在上面的例子中已经出现过了，我们通过 `callback.invoke(xx)` 就可以将数据回传给 JavaScript，使用起来比较简单，这边我们就不再赘述了。现在我们主要来看下 Promise 和发送事件的示例，以便更好地了解 React Native 和原生之间是如何进行通信的。

首先我们来看下 Promise 示例，我们从 JavaScript 如何触发、Native 如何回传数据两方面来进行讲解。

首先，JavaScript 端调用客户端定义的 `SystemPropsModule` 的 `getSystemModel` 来获取手机的设备类型，获取结果的方式使用 Promise 方式（`then... catch...`）：

```

1 NativeModules.SystemPropsModule.getSystemModel().then(result=> {
2   console.log(result);
3 }).catch(error=> {
4   console.log(error);
5 });

```

 复制代码

然后，Native 端定义 SystemPropsModule，实现 getSystemModel 方法，内部使用 promise 获取手机的 model 数据。使用 promise.resolve(xx) 为成功，promise.reject(xx) 为失败：

 复制代码

```
1 SystemPropsModule:
2 ...
3 @ReactMethod
4 public void getSystemModel(Promise promise) {
5     // 回传成功，使用 resolve
6     promise.resolve(Build.MODEL);
7 }
8 ...
```

接下来看发送事件示例，我们从 JavaScript 如何监听 Native 事件、Native 如何发送事件这两方面来进行讲解。

首先，JavaScript 端使用 EventEmitterManager 来注册 Native 的事件监听。通过 NativeModules 获取 EventEmitterManager，随后使用它构建出 NativeEventEmitter，最后通过 NativeEventEmitter 注册监听：

 复制代码

```
1 componentWillMount(){
2     // 拿到原生模块
3     var eventEmitterManager = NativeModules.EventEmitterManager;
4     const nativeEventEmitter = new NativeEventEmitter(eventEmitterManager);
5     const eventEmitterManagerEvent = EventEmitterManager.EventEmitterManagerEvent
6     // 监听 Native 发送的通知
7     this.listener = nativeEventEmitter.addListener(eventEmitterManagerEvent, (data) => {
8         console.log("Receive native event: " + data);
9     });
10 }
11
12 componentWillUnmount(){
13     // 移除监听
14     this.listener.remove();
15 }
```

在 Native 端的使用则很简单。我们获取 RCTDeviceEventEmitter 这个 JSModule，使用 emit 方法就可以向 JavaScript 发送事件了：

```
1 reactContext.getJSModule(DeviceEventManagerModule.RCTDeviceEventEmitter.class).emit("msg", "say hello");
2 .emit("msg", "say hello");
```

[复制代码](#)

自定义组件相关的知识点，我们先介绍到这里。接下来进入实战阶段，我们将分别以一个数据存取 TurboModule 和视频播放 Fabric component 的案例，加深你对自定义组件的理解。我们先来看 TurboModule。

TurboModule：数据存取

TurboModule 采用懒加载模式，在运行时第一次 import 该 TurboModule 时，Native 会创建对应的 TurboModule 并进行缓存。而旧版本的 NativeModule 都是在创建环境时统一进行构造的，会对 React Native 的启动性能有比较大的影响。

接下来我们以一个实际的案例来带你了解 TurboModule。当你需要使用 native 数据存取相关能力，如跨进程存取、偏好存取、加密存取等，而 React Native 自带的数据存储 module 满足不了你的需求，你可以通过自定义数据存储的 TurboModule 来实现。我们先来看下 JavaScript 侧。

JavaScript

在 Spec 中定义方法，定义好存和取的方法后，再导出 StorageModule：

```
1 export interface Spec extends TurboModule {
2   +save: (key: string, value: string, callback: (value: Object) => void) => void;
3   +get: (key: string, callback: (value: Object) => void) => void;
4 }
5 // 导出 StorageModule
6 export default (TurboModuleRegistry.getEnforcing<Spec>(
7   'StorageModule',
8 ): Spec);
```

[复制代码](#)

调用：

```
1 NativeModules.StorageModule.save("testKey", "testValue", (result)=>{
2   console.info(result);
3 })
```

[复制代码](#)

```
4  );  
5  NativeModules.StorageModule.get("testKey", (result)=>{  
6      console.info(result);  
7  }  
8  );
```

接下来我们再看看 Android 端和 iOS 端的实现。

Android

在实现 `StorageModule` 之前，我们需要在混合工程中，将 `React Native` 新架构的运行配置搭建好，这套配置可以运行 `TurboModule`、`Fabric`，后面的 `Fabric` 案例也是基于此配置来运行的。

而且，上一讲在我们已经讲解了如何基于 `React Native` 最新版本（`0.68.0`）搭建混合应用，我们在这个基础上开启新架构配置就好了。

第一步，我们要做些准备工作，也就是获取 `newarchitecture` 的模版代码。

我们以 `0.68.0` 版本创建一个 `React Native` 工程，来获取新架构的模版代码，我们把这个工程名叫做 `ReactNativeNewArch`：

```
1 npx react-native init ReactNativeNewArch --version 0.68.0
```

 复制代码

创建好后，你将看到如下工程代码，包含 `Java` 和 `C++`：



这些工程代码主要是新架构的 JSI、Fabric、TurboModule 的注册和加载代码，内部逻辑非常复杂，这一讲我们就不做过多分析了，先专注于实操部分。

如果我们想基于这个 Demo 去运行新架构，可以做如下操作：


```
1 1. 修改 android 目录下的 gradle.properties:
2 # 开启新架构
3 newArchEnabled=true
4 # 配置 java home 为 JDK 11
5 org.gradle.java.home=/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/
6
7 2. 运行
8 yarn react-native run-android
```

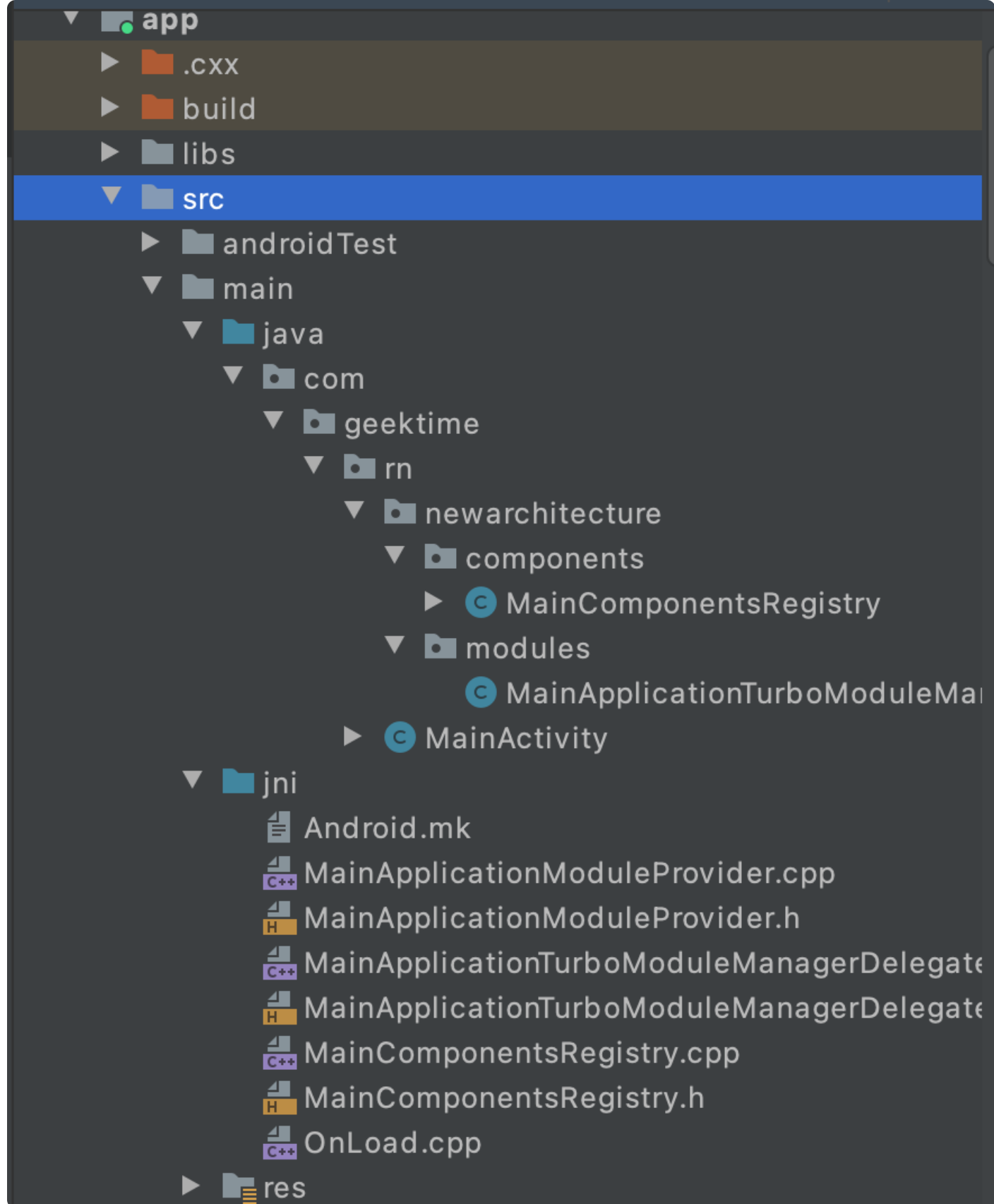
第二步，拷贝 newarchitecture 的模版代码到我们之前的混合工程。

这一步中，我们需要将 Java 层和 C++ 层代码拷贝到混合工程中，需要拷贝的相关代码如下：

```
1 Java 层:
2 - MainComponentsRegistry.java
3 - MainApplicationTurboModuleManagerDelegate.java
4
5 C++ 层:
6 - jni 目录
```

 复制代码

拷贝完的效果是这样的：



第三步是修改拷贝的代码，主要是下面这四点。

1.MainApplicationTurboModuleManagerDelegate.java:

修改 so 库名称为我们自定义名称 `geektime_new_arch`。

```
1 @Override
2 protected synchronized void maybeLoadOtherSoLibraries() {
3     if (!sIsSoLibraryLoaded) {
4         SoLoader.loadLibrary("geektime_new_arch");
5         sIsSoLibraryLoaded = true;
6     }
7 }
```

2.jni/Android.mk: 修改 Android.mk 中的 so 库名称为上面的 `geektime_new_arch`。

```
1 # You can customize the name of your application .so file here.
2 LOCAL_MODULE := geektime_new_arch
```

3.jni/MainApplicationTurboModuleManagerDelegate.h: 修改

MainApplicationTurboModuleManagerDelegate 对应的 Java 类路径，截图中拷贝好的 MainApplicationTurboModuleManagerDelegate.java 路径为 `com/reactnativewarch/newarchitecture/modules`。

```
1 static constexpr auto kJavaDescriptor =
2     "Lcom/reactnativewarch/newarchitecture/modules/MainApplicationTurboModule
```

4.jni/MainComponentsRegistry.h: 修改 MainComponentsRegistry 对应的 Java 类路径，截图中拷贝好的 MainComponentsRegistry.java 路径为 `com/reactnativewarch/newarchitecture/components`。

```
1 constexpr static auto kJavaDescriptor =
2     "Lcom/reactnativewarch/newarchitecture/components/MainComponentsRegistry;
```

做完后，最后一步就是修改 React Native 初始化代码了。


```

24         reactApplicationContext,
25         componentFactory,
26         new EmptyReactNativeConfig(),
27         viewManagerRegistry);
28     }
29 });
30 return specs;
31 }
32 };
33 }

```

至此，我们新架构的运行环境就配置好了。由于目前新架构文章非常少，几乎没有混合工程运行新架构的方案，上面这些主要是我们用了大量的时间去调研和测试的结果，你可以参考一下。

好了，回到正题。在混合工程中，新架构的运行环境搭建好后，我们就可以简单快速地来写 TurboModule 和 Fabric 了。

我们继续来进行数据存储的 Demo 在 Java 层定义并实现 StorageModule。Android 端我们使用 SharedPreferences 来实现轻量级偏好存取。这里要注意，你需要继承 ReactModuleWithSpec 和 TurboModule，具体代码如下：

 复制代码

```

1 // 1. 定义 StorageModule，继承 ReactContextBaseJavaModule 类
2 // 实现 ReactModuleWithSpec & TurboModule 接口
3 public class StorageModule extends ReactContextBaseJavaModule
4     implements ReactModuleWithSpec, TurboModule {
5     // native 存储的 sp 文件名称
6     private static final String SP_NAME = "rn_storage";
7     // 返回给 JS 的结果码
8     private static final int CODE_SUCCESS = 1;
9     private static final int CODE_ERROR = 2;
10
11     public StorageModule(ReactApplicationContextWrapper reactContext) {
12         super(reactContext);
13     }
14
15     // 返回 module 名称，一般以类名作为 module 名称
16     @Override
17     public String getName() {
18         return StorageModule.class.getSimpleName();
19     }
20
21     // 定义供 JS 调用的存储数据方法，isBlockingSynchronousMethod 表示是否同步执行
22     @ReactMethod(isBlockingSynchronousMethod = true)

```

```

23     public void save(String key, String value, Callback callback) {
24         WritableMap result = new WritableNativeMap();
25         // 如果 js 传入的 key 为空, 则回传失败码和信息
26         if (TextUtils.isEmpty(key)) {
27             result.putInt("code", CODE_ERROR);
28             result.putString("msg", "key is empty or null");
29             callback.invoke(result);
30             return;
31         }
32         // 调用 native 的 sp 进行数据存储
33         SharedPreferences sp = getReactApplicationContext().getSharedPreferences
34         sp.edit().putString(key, value).apply();
35         result.putInt("code", CODE_SUCCESS);
36         result.putString("msg", "save success");
37         // 回传 js 告知存储成功
38         callback.invoke(result);
39     }
40
41     // 定义供 JS 调用的获取数据方法, isBlockingSynchronousMethod 表示是否同步执行
42     @ReactMethod(isBlockingSynchronousMethod = true)
43     public void get(String key, Callback callback) {
44         // 如果 js 传入的 key 为空, 则回传失败码和信息
45         WritableMap result = new WritableNativeMap();
46         if (TextUtils.isEmpty(key)) {
47             result.putInt("code", CODE_ERROR);
48             result.putString("msg", "key is empty or null");
49             callback.invoke(result);
50             return;
51         }
52         // 调用 native 的 sp 获取 key 对应的 value 值
53         SharedPreferences sp = getReactApplicationContext().getSharedPreferences(
54         String value = sp.getString(key, "");
55         result.putInt("code", CODE_SUCCESS);
56         result.putString("data", value);
57         // 将结果回传给 js
58         callback.invoke(result);
59     }
60 }

```

然后是注册组件, 注意 `Package` 需要继承 `TurboReactPackage`:

 复制代码

```

1 public class MyTurboModulePackage extends TurboReactPackage {
2     @Override
3     public NativeModule getModule(String name, ReactApplicationContext reactCon
4         switch(name) {
5             case "StorageModule":
6                 return new StorageModule(reactContext);
7                 break;
8             default:

```

```

9         return null;
10    }
11 }
12
13 @Override
14 public ReactModuleInfoProvider getReactModuleInfoProvider() {
15     //...
16 }
17 }

```

接下来注册到 `ReactInstanceManager`（使用 `reactInstanceManagerBuilder` 注册）：

 复制代码

```

1 ReactInstanceManagerBuilder builder = ReactInstanceManager.builder()
2     .addPackage(new MyTurboModulePackage());
3 ... // 其他 RN 初始化配置
4 ReactInstanceManager reactInstanceManager = builder.build();

```

然后我们利用新架构提供的 `Codegen`，生成新架构需要的 `native` 代码。不过，在使用 `codegen` 之前，我们需要在项目中应用相关的插件：

(1) 在工程根目录安装 `react-native-gradle-plugin`。

 复制代码

```

1 yarn add react-native-gradle-plugin

```

(2) 在工程根目录的 `settings.gradle` 中配置 `react-native-gradle-plugin`，使用复合构建引入。

 复制代码

```

1 include ':app'
2 rootProject.name = "GeekTimeRNAndroid"
3 includeBuild('./node_modules/react-native-gradle-plugin')

```

(3) 在 `app/build.gradle` 中应用插件。

 复制代码

```

1 apply plugin: "com.facebook.react"

```

这样做后，工程的 gradle 任务中就会出现 generateCodegenArtifactsFromSchema task。

 复制代码

- 1 配置好后，我们以后就可以使用 codegen 能力了，然后执行 generateCodegenArtifactsFromSchema
- 2 ../gradlew generateCodegenArtifactsFromSchema

Android 端的就是这样，现在我们看 iOS 端需要怎么做。

iOS

在 iOS 端中，首先我们要创建一个类并遵循一个协议 spec，协议中包含注册的 API 声明。而且，该协议需要遵循 RCTBridgeModule 协议和 RCTTurboModule 协议，并且创建一个 JSI。示例代码如下：

 复制代码

```
1 //定义一个Spec协议
2 @protocol DataStorageTurboModuleSpec <RCTBridgeModule, RCTTurboModule>
3 - (NSString *)getString:(NSString *)string;
4 @end
5
6 //JSI实现
7 namespace facebook {
8 namespace react {
9 class JSI_EXPORT DataStorageTurboModuleSpecJSI : public ObjCTurboModule {
10     public:
11         DataStorageTurboModuleSpecJSI(const ObjCTurboModule::InitParams &params);
12     };
13 } // namespace react
14 } // namespace facebook
15
16 //定义一些方法
17 namespace facebook {
18     namespace react {
19         static facebook::jsi::Value __hostFunction_DataStorageTurboModuleSpecJSI_ge
20             facebook::jsi::Runtime &rt,
21             TurboModule &turboModule,
22             const facebook::jsi::Value *args,
23             size_t count){
24             return static_cast<ObjCTurboModule &>(turboModule)
25                 .invokeObjCMethod(rt, VoidKind, "getString", @selector(getString:), args,
26         }
27     DataStorageTurboModuleSpecJSI::NativeSampleTurboModuleSpecJSI(const ObjCTurb
28         : ObjCTurboModule(params)
```



```

29     {
30         //MethodMetadata 第一个参数为0代表该方法有一个参数
31         methodMap_["getString"] = MethodMetadata{1, __hostFunction_DataStorageTurbo
32     }
33 }
34 }
35
36 //TurboModule遵循Spec协议
37 @interface DataStorageTurboModule : NSObject <DataStorageTurboModuleSpec>
38
39 @end

```

之后，我们要在该类中注册组件名和 API：

 复制代码

```

1 //DataStorageTurboModule.mm
2 @implementation DataStorageTurboModule
3 RCT_EXPORT_MODULE()
4
5 - (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:
6     (const facebook::react::ObjCTurboModule::InitParams &)params{
7     //指定JSI
8     return std::make_shared<DataStorageTurboModuleSpecJSI>(params);
9 }
10
11 RCT_EXPORT_METHOD(getString:(NSString *)string){
12     NSLog(@"");
13 }
14 @end

```

以上便是自定义一个 TurboModule 的流程。其实定义 TurboModule 并不复杂，而且 Facebook 也提供了代码生成工具 codegen，比较复杂的是在混合工程中搭建新架构的运行环境。前面我们花了不少内容讲述如何在客户端开启新架构，接下来的 Fabric 组件介绍也将在新架构环境基础上进行讲解，接下来我们继续来看 Fabric 自定义组件。

Fabric：视频播放

Fabric 对标旧框架的 UIManager。FabricUIManager 可以和 C++ 层直接进行通讯，解除了原有的 UIManager 依赖单个 bridge 的问题。有了 JSI 后，以前批量依赖 bridge 的 UI 操作，都可以同步执行到 C++ 层，性能得到大幅提升，特别是在列表快速滑动、复杂动画交互方面提升更加明显。

现在，我们以一个视频播放组件为例，讲讲如何定义 **Fabric** 组件。我们先来看下 **JavaScript** 端的实现。

JavaScript

JavaScript 需要定义属性以及 API，并 **export** 组件。示例代码如下：

 复制代码

```
1  type NativeProps = $ReadOnly<{|
2    ...ViewProps,
3    url?: string
4 |}>; // 定义视频播放的属性，url 为视频地址
5
6  export type VideoViewType = HostComponent<NativeProps>;
7
8  // 定义视频播放的方法，包括开始播放、停止播放、暂停播放
9  interface NativeCommands {
10    +callNativeMethodToPlayVideo: (
11      ) => void;
12    +callNativeMethodToStopVideo: (
13      ) => void;
14    +callNativeMethodToPauseVideo: (
15      ) => void;
16  }
17
18  // 导出外部调用的命令，包括开始播放、停止播放、暂停播放
19  export const Commands: NativeCommands = codegenNativeCommands<NativeCommands>({
20    supportedCommands: ['callNativeMethodToPlayVideo'],
21    supportedCommands: ['callNativeMethodToStopVideo'],
22    supportedCommands: ['callNativeMethodToPauseVideo'],
23  });
24
25  // 导出包装好的组件，其中 VideoView 为引入 Native 的组件
26  export default (codegenNativeComponent<NativeProps>(<
27    'VideoView',
28  ): VideoViewType);
29
```

JavaScript 端使用该组件：

 复制代码

```
1  // 导入 ViewView 组件和工具
2  import VideoView, {
3    Commands as VideoViewCommands,
4  } from './VideoNativeComponent';
5
```

```

6 // 外部调用此方法即可调用 VideoView 视频播放能力
7 export default function MyView(props: {}): React.Node {
8   return (
9     <View>
10       <VideoView url={"url"} style={{flex: 1}} />
11       <Button title="play" onPress={()=>{
12         VideoViewCommands.callNativeMethodToPlayVideo();
13       }} />
14     </View>
15   )
16 }
17 }

```

接下来我们再看看 Android 端和 iOS 端的实现。

Android

由于在前面 TurboModule 的部分，我们已经讲解了如何在混合工程中开启新架构运行模式，我么这里就不再重复了。前面的方法同样适用于 Fabric，我们只需要搭建一次就好了。所以现在要在 Android 端实现 Fabric 组件也非常简单，我们来看下具体实现。

第一步，定义视频播放接口。

这里我们要定义 VideoViewManagerInterface，其中包含三个方法：播放视频、停止播放、暂停播放：

```

1 public interface VideoViewManagerInterface<T extends View> {
2     void playVideo(T view, String url);
3     void stopVideo(T view);
4     void pauseVideo(T view);
5 }

```

 复制代码

第二步，定义视频播放 View。

这一步中，我们要实现视频播放的 View。在 View 中，我们需要实现视频的播放、停止和暂停功能。但播放能力的实现并不是我们讲解的重点，我们这一讲侧重于新架构中 Fabric 组件的实现流程，所以我们这边使用伪代码：

```
1 public class MyVideoView extends View {
2     // ...
3
4     public void playVideo(String url) {
5         // 播放视频实现
6     }
7
8     public void stopVideo() {
9         // 停止视频播放实现
10    }
11
12    public void pauseVideo() {
13        // 暂停视频播放实现
14    }
15 }
```

第三步，定义 ViewManager。

在这一步中，我们要实现暴露给 React Native 调用的能力，包括视频播放、停止，以及暂停，内部会转发到上面我们定义的视频播放 View 的实现中。示例代码如下：

```
1 @ReactModule(name = VideoViewManager.REACT_CLASS)
2 public class VideoViewManager: ViewGroupManager<VideoView>(), VideoViewManagerI
3     private static final String REACT_CLASS = "VideoView";
4
5     public VideoViewManager() {
6     }
7
8     override
9     public String getName() {
10         return REACT_CLASS;
11     }
12
13     override
14     public VideoView createViewInstance(ThemedReactContext reactContext) {
15         return new MyVideoView(reactContext);
16     }
17
18     @ReactProp(name = "url")
19     override
20     public void playVideo(VideoView view, String url) {
21         view.playVideo(url);
22     }
23
24     override
25     public void stopVideo(VideoView view) {
```

```

26         view.stopVideo();
27     }
28
29     override
30     public void pauseVideo(VideoView view) {
31         view.pauseVideo();
32     }
33 }

```

最后一步就是注册 ViewManager。我们在 ReactInstanceManager 的 JSIModulesPackage 中注册 VideoViewManager:

 复制代码

```

1 List<ViewManager> viewManagers = new ArrayList<>();
2 viewManagers.add(new VideoViewManager())
3 ViewManagerRegistry viewManagerRegistry = new ViewManagerRegistry(viewManagers)
4
5 return new FabricJSIModuleProvider(
6     reactApplicationContext,
7     componentFactory,
8     new EmptyReactNativeConfig(),
9     viewManagerRegistry);

```

然后我们利用新架构提供的 Codegen，调用 gradlew generateCodegenArtifactsFromSchema 生成代码 Native 代码:

 复制代码

```

1 ../gradlew generateCodegenArtifactsFromSchema

```

最后，运行即可。Android 端的实现就是这样，接下来我们再看下 iOS 端。

iOS

在 iOS 端汇总，首先我们要创建一个继承于 RCTViewComponentView 的一个类作为视频组件，如下:

 复制代码

```

1 @interface VideoComponentView : RCTViewComponentView
2 //声明播放器组件的一些方法
3 //播放视频

```

```

4 - (void)playVideo;
5 //停止视频
6 - (void)stopVideo;
7 //暂停视频
8 - (void)pauseVideo;
9 @end

```

之后，该类需要遵循一个协议，协议中需要声明执行 **Command** 的方法名，示例代码如下：

 复制代码

```

1 @protocol VideoComponentViewProtocol <NSObject>
2 - (void)callNativeMethodToPlayVideo;
3 - (void)callNativeMethodToStopyVideo;
4 - (void)callNativeMethodToPauseVideo;
5 @end
6
7 RCT_EXTERN inline void VideoComponentCommand(
8     id<VideoComponentViewProtocol> componentView,
9     NSString const *commandName,
10    NSArray const *args)
11
12    if([commandName isEqualToString:@"callNativeMethodToPlayVideo"]){
13        [componentView callNativeMethodToPlayVideo];
14        return;
15    }
16
17    if(![commandName isEqualToString:@"callNativeMethodToStopVideo"]){
18        [componentView callNativeMethodToStopVideo];
19        return;
20    }
21
22    if(![commandName isEqualToString:@"callNativeMethodToPauseVideo"]){
23        [componentView callNativeMethodToPauseVideo];
24        return;
25    }
26    return;
27 )

```

接下来，**ComponentView** 需要遵循该 **Protocol** 协议，并在执行 **common** 时调用对应的方法。此外，我们还可以设置组件的属性：

 复制代码

```

1 using namespace facebook::react;
2
3 @interface VideoComponentView() <VideoComponentViewProtocol>

```

```

4  @end
5
6  @implementation VideoComponentView{
7      VideoPlayer *_videoPlayer;
8  }
9
10 #pragma mark - Native Commands
11 - (void)handleCommand:(const NSString *)commandName args:(const NSArray *)args{
12     VideoComponentCommand(self, commandName, args);
13 }
14
15 - (void)callNativeMethodToPlayVideo{
16     //实现视频播放功能
17     [_videoPlayer startPlay];
18 }
19
20 - (void)callNativeMethodToStopVideo{
21     //实现视频停止功能
22     [_videoPlayer stopPlay];
23 }
24
25 - (void)callNativeMethodToPauseVideo{
26     //实现视频暂停功能
27     [_videoPlayer pausePlay];
28 }
29
30 #pragma mark - Props
31 //遵循descriptor协议
32 + (ComponentDescriptorProvider)componentDescriptorProvider{
33     return concreteComponentDescriptorProvider<VideoComponentDescriptor>();
34 }
35
36 - (instancetype)initWithFrame:(CGRect)frame{
37     if (self = [super initWithFrame:frame]) {
38         static const auto defaultProps = std::make_shared<const ComponentViewProps>
39             _props = defaultProps;
40
41         _videoPlayer = [[VideoPlayer alloc] init];
42         self.contentView = _videoPlayer;
43     }
44     return self;
45 }
46
47 - (void)updateProps:(Props::Shared const &)props oldProps:(Props::Shared const
48     [super updateProps:props oldProps:oldProps];
49 }
50
51 - (void)onChange:(UIView *)sender{
52     // No-op
53     // std::dynamic_pointer_cast<const ViewEventEmitter>(_eventEmitter)
54     //     ->onChange(ViewEventEmitter::OnChange{.value = static_cast<bool>(send
55 }

```

```

56 @end
57
58 Class<RCTComponentViewProtocol> VideoViewCls(void){
59     return VideoComponentView.class;
60 }
61

```

接着，注册属性遵循 `VideoComponentDescriptor`，并且需要指定该组件的名字：

 复制代码

```

1 namespace facebook {
2 namespace react {
3     using VideoComponentDescriptor = ConcreteComponentDescriptor<VideoViewShadowNode>
4 } // namespace react
5 } // namespace facebook
6
7 namespace facebook {
8 namespace react {
9     extern const char VideoViewComponentName[];
10    using VideoViewShadowNode = ConcreteViewShadowNode<
11        VideoViewComponentName, // 组件名
12        VideoViewProps>; // 注册属性
13 } // namespace react
14 } // namespace facebook
15
16 namespace facebook {
17 namespace react {
18     extern const char VideoViewComponentName[] = "VideoView"; // 组件名
19 } // namespace react
20 } // namespace facebook
21
22 namespace facebook {
23 namespace react {
24     // 属性定义
25     class VideoViewProps final : public ViewProps {
26     public:
27         VideoViewProps() = default;
28         VideoViewProps(const PropsParserContext& context, const VideoViewProps &source
29
30         #pragma mark - Props
31         std::string url{" "}; // 视频url
32     };
33 } // namespace react
34 } // namespace facebook
35

```

这样，我们就创建好了一个 React Native 的 Fabric 组件、定义属性以及 API 的方法。

以上便是如何使用 **Fabric** 自定义视频播放组件，在混合工程中搭建好新架构的运行环境后，只需要遵守 **Fabric** 的组件定义方式，进行接口定义、功能实现和组件注册即可。关于一些复杂的 **Fabric** 组件，可以查看 <https://github.com/software-mansion/react-native-gesture-handler/tree/main/FabricExample>，<https://github.com/software-mansion/react-native-reanimated/tree/main/FabricExample>，目前 **react-native-gesture-handler**、**react-native-reanimated** 都已经适配了新架构，感兴趣的同学可以去学习下。

总结

这一讲，我们系统讲解了个性化组件的使用场景、生命周期、传输类型，以及通信方式，并通过两个实际案例讲解了如何在新架构下定制个性化的 **TurboModules** 与 **Fabric**。而且，我们也简单介绍了一下 **React Native** 新架构，你可以通过官方文档进行新架构的体验。

这一讲是我们 **Native** 相关的三讲中花的时间最长的，也是最“伤肝”的，我们前前后后加调研花了两个月的时间。但我们相信，新架构在未来会有很好的发展，这是可以预见的。因为它解决了 **React Native** 几个最痛的点，包括启动速度、运行时性能等。如果新架构还能在易用性上继续优化，将会大大拓展 **React Native** 的用户群体。

因为目前新架构还处于未发布的状态，网上相关的文章大都是对官方纯 **React Native** 模式 **Demo** 和介绍，少数几篇会深挖原理，但讲混合模式的新架构运行文章几乎没有。我们这一讲中对 **TurboModule** 和 **Fabric** 的讲解，更侧重于如何在混合工程中开启并运行。如果有对 **TurboModule**、**Fabric**、**JSI** 的原理感兴趣的同学，后面有机会我们再来分享。


作业

1. 设计一个打印 **Native** 日志的 **TurboModule**，以及一个 **Native** 加载进度条的 **Fabric** 组件。

欢迎在评论区写下你的想法和经验，和我们多多交流。我们下一讲见。

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

上一篇 21 | 混合应用：如何从零开始集成 React Native？

下一篇 23 | 热更新：如何搭建一个热更新平台？

精选留言 (1)





万千观众之一

2022-05-18

收获满满，期待老师再次更新！！

