RN与原生模块交互

一、RN调用原生方法

Android端

1. 创建模块

- 创建一个module继承ReactContextBaseJavaModule
- 实现getName()方法，返回一个字符串，在JS端就用这个字符串标识此module
- 创建一个可供JS调用的测试方法，用注解@ReactMethod修饰

TestModule.java

```
public class TestModule extends ReactContextBaseJavaModule {
    public TestModule(ReactApplicationContext reactContext) {
        super(reactContext);
    }

    @Override
    public String getName() {
        return "TestModule";
    }


    @ReactMethod
    public void testFun(String str1, String str2) {
        System.out.println("收到JS传值" + str1 + "--" + str2);
    }
}
```

2. 注册模块

- 创建一个包管理类实现ReactPackage接口
- 实现createNativeModules()方法，并添加刚才创建的module

TestPackage.java

```
public class TestPackage implements ReactPackage {

    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext reactCon
text) {
        List<NativeModule> modules=new ArrayList<>();
        modules.add(new TestModule(reactContext));

        return modules;
    }

    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext reactConte
xt) {
        return Arrays.<ViewManager>asList(
                new TestTextViewManager()
        );
    }
}
```

3. 在Application类添加包管理类

MainApplication.java

```
@Override
protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
            new MainReactPackage(),
            new TestPackage()
    );
}
```

至此，Android端的代码工作完成。

IOS端

- 创建一个module实现RCTBridgeModule协议
- 添加宏 RCT_EXPORT_MODULE（这个宏也可以添加一个参数用来指定在JS中访问这个模块的名字。如果不指定，默认就会使用类名）
- 创建一个可供JS调用的测试方法,方法的实现需要在 RCT_EXPORT_METHOD 这个宏里面

TestModule.h

```
#ifndef TestModule_h
#define TestModule_h

#import <React/RCTBridgeModule.h>
#import <React/RCTBridge.h>
#import <React/RCTEventDispatcher.h>


@interface TestModule : NSObject <RCTBridgeModule>
@end
#endif /* TestModule_h */
```

TestModule.m

```
#import <Foundation/Foundation.h>
#import "TestModule.h"

@implementation TestModule

RCT_EXPORT_MODULE()

RCT_EXPORT_METHOD(testFun:(NSString *)str1 forSomething:(NSString *)str2) {
  NSString *info = [NSString stringWithFormat:@"收到JS传值%@---%@", str1, str2];
  NSLog(info);
}

@end
```

至此，IOS端的代码工作完成。

JS端

直接调用，其中TestModule为原生端暴露出来的模块名，testFun为原生方法

```
    import { NativeModules } from 'react-native';
    ...
    NativeModules.TestModule.testFun("参数1", "参数2");
```

二、原生调用RN方法

JS端

添加监听函数

```
import {
    DeviceEventEmitter,
    NativeModules,
} from 'react-native';
...
componentWillMount() {
    //监听事件名为TestNativeEvent的事件
    this.listener = DeviceEventEmitter.addListener('TestNativeEvent', (result)=> {
        console.log(result)
    });
}
```

Android端

通过 RCTDeviceEventEmitter 直接发送, 下面发送事件名为"TestNativeEvent"，与JS端监听的事件名一致

TestModule.java

```
private void sendEvent() {
    mContext.getJSModule(DeviceEventManagerModule.RCTDeviceEventEmitter.class).emi
t("TestNativeEvent", "我是原生字符串);
}
```

IOS端

TestModule.m

- 在module中声明支持的事件名

```
- (NSArray<NSString *> *)supportedEvents {
  return @[@"TestNativeEvent"]; //这里返回的将是你要发送的消息名的数组。
}
```

- 发送事件

```
[self.bridge.eventDispatcher sendAppEventWithName:@"TestNativeEvent" body:@{@"
testkey": @"我是原生字符串"}];
```

原生调用RN方法代码完成

三、callback与promise

上面所说的通信都是直接调用的方式，有时候会有这样一个需求，RN调用原生去执行耗时的操作，原生在执行完后需要给RN一个反馈，这就需要用到callback与promise了。

1. 使用callback

JS端

```
NativeModules.TestModule.rnCallNativeFromCallback("我是JS端callback",(result) => {
    console.log(result)
})
```

Android端

TestModule.java

```
@ReactMethod
public void rnCallNativeFromCallback(String msg, Callback callback) {
    String result = "Android Callback： " + msg;
    callback.invoke(result);
}
```

IOS端

TestModule.m

```
RCT_EXPORT_METHOD(rnCallNativeFromCallback:(NSString*) str andCallback:(RCTRespons
eSenderBlock)callback) {
  NSArray *result = @[@"ios result for Callback"];
  callback(result);
}
```

2. 使用promise

JS端

```
NativeModules.TestModule.rnCallNativeFromPromise("我是JS端promise")
    .then((result) => {
        this.setState({content: result})
    }).catch((error) => {
        this.setState({content: "errorCode:" + error.code});
    });
```

Android端

TestModule.java

```
@ReactMethod
public void rnCallNativeFromPromise(String msg, Promise promise) {
    String result = "Android Promise: " + msg;
    promise.resolve(result);    //  promise.reject();
}
```

IOS端

TestModule.m

```
RCT_EXPORT_METHOD(rnCallNativeFromPromise:(NSString*)str resolve:(RCTPromiseResolv
eBlock)resolve rejecter:(RCTPromiseRejectBlock)reject) {
  if (TRUE) {
    resolve(@[@"ios result for Promise"]);
  } else {
    reject(@"404",@"错误信息",[NSError errorWithDomain:@"错误" code:1 userInfo:nil])
;
  }
}
```

四、常量传递

除了上述的通信方式，RN还提供了一种简单的常量传递方式，这种方式只能从原生端向RN端传递。
RN端可通过 NativeModules.[module名].[参数名] 的方式获取

Android端

TestModule.java

```
@Override
public Map<String, Object> getConstants() {
    Map<String,Object> params = new HashMap<>();
    params.put("MyConstant","我是Android常量，传递给RN");
    return params;
}
```

IOS端

TestModule.m

```
- (NSDictionary *)constantsToExport
{
  return @{ @"MyConstant": @"我是IOS常量，传递给RN" };
}
```

JS端

```
let result = NativeModules.TestModule.MyConstant
```

构建原生UI模组件

有时候RN提供的UI组件不能满足我们的需求，这时候就需要用到原生的组件去完成。 下面举的例子是构建Android的TextView，IOS的UITextView，使在JS端能使用原生组件。

Android端

- 创建一个视图管理类继承SimpleViewManager<T>, T是泛型，我们这里需要构建的是TextView，
- 实现getName()方法，返回一个字符串，在JS端就用这个字符串标识此视图
- 实现createViewInstance()方法
- 设置视图的自定义属性供JS端调用，属性用@ReactProp（或@ReactPropGroup）注解
- 把这个视图管理类注册到package的createViewManagers里。

TestTextViewManager.java

```
public class TestTextViewManager extends SimpleViewManager<TextView> {

    @Override
    public String getName() {
        return "TestReactText"; // 暴露给JS端的标识
    }

    @Override
    protected TextView createViewInstance(ThemedReactContext reactContext) {
        return new ReactTextView(reactContext);
    }

    @ReactProp(name = "myText")
    public void setMyText(TextView view, @Nullable String src) {    // 自定义属性
        view.setText(src);
        view.setTextColor(Color.parseColor("#00FF00"));
    }
}
```

TestPackage.java

```
@Override
public List<ViewManager> createViewManagers(ReactApplicationContext reactContext)
{
    return Arrays.<ViewManager>asList(
            new TestTextViewManager()
    );
}
```

IOS端

1.创建视图组件

TestReactTextView.h

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>


@interface TestReactTextView : UITextView

@end
```

TestReactTextView.m

```
#import "TestReactTextView.h"

@implementation TestReactTextView

-(void)setMyText:(NSString*) str
{
  self.text = str;
  [self layoutSubviews];
}


@end
```

2.创建视图管理类

- 创建一个视图管理类继承RCTViewManager
- 添加宏RCT*EXPORT*MODULE()
- 实现-(UIView *)view方法
- 设置视图的自定义属性供JS端调用

TestReactTextManager.h

```
#import <Foundation/Foundation.h>
#import <React/RCTViewManager.h>

@interface TestReactTextManager : RCTViewManager

@end
```

TestReactTextManager.m

```
#import <Foundation/Foundation.h>
#import <React/RCTViewManager.h>

@interface TestReactTextManager : RCTViewManager

@end


TestReactTextManager.m
#import "TestReactTextManager.h"
#import "TestReactTextView.h"

@implementation TestReactTextManager

RCT_EXPORT_MODULE();

RCT_EXPORT_VIEW_PROPERTY(myText, NSString);


- (UIView *)view
{
  return [[TestReactTextView alloc] init];
}

@end
```

JS端

创建JavaScript模块并且定义Java和JavaScript之间的接口层。

TestReactText.js

```
import { requireNativeComponent, View } from 'react-native';
import PropTypes from 'prop-types';  //npm install --save prop-types

var iface = {
    name: 'MyText',
    propTypes: {
        myText: PropTypes.string,
        ...View.propTypes
    },
};

module.exports = requireNativeComponent('TestReactText', iface);
```

说明几点：

- 组件接口声明propTypes字段，用来对应到原生视图上自定义的属性。
- "...View.propTypes" 这句表明包含默认的View的全部属性
- requireNativeComponent 通常接受两个参数，第一个参数是原生视图暴露出来的标识，而第二个参数是一个描述组件接口的对象。

使用

```
import MyText from './TestReactText.js';
...

<MyText myText="测试自定义view" style={{width: 200, height: 30, marginBottom: 40}}/>
```

打包发布

先来看下打包的命令参数

```
react-native bundle [参数]
  构建 js 离线包

  Options:

-h, --help                    输出如何使用的信息
--entry-file <path>           RN入口文件的路径，绝对路径或相对路径
--platform [string]           ios 或 andorid
--transformer [string]        Specify a custom transformer to be used
--dev [boolean]               如果为false，警告会不显示并且打出的包的大小会变小
--prepack                     当通过时，打包输出将使用Prepack格式化
--bridge-config [string]      使用Prepack的一个json格式的文件__fbBatchedBridgeConfig 例
如：./bridgeconfig.json
--bundle-output <string>      打包后的文件输出目录，例：/tmp/groups.bundle
--bundle-encoding [string]    打离线包的格式 可参考链接https://nodejs.org/api/buffer.ht
ml#buffer_buffer.
--sourcemap-output [string]  生成Source Map，但0.14之后不再自动生成source map，需要手动
指定这个参数。例：/tmp/groups.map
--assets-dest [string]        打包时图片资源的存储路径
--verbose                     显示打包过程
--reset-cache                 移除缓存文件
--config [string]             命令行的配置文件路径
```

Android打包命令如下:

```
react-native bundle --platform android --dev false --entry-file index.js --bundle-
output ./android/app/src/main/assets/index.android.bundle --assets-dest ./android/
app/src/main/res
```

需要注意的是

- android/app/src/main/ 目录下需有assets文件夹，如没有则手动创建，否则命令执行不成功。
- 执行命令成功后，需要的离线bundle js文件是 ./android/app/src/main/assets/index.android.bundle ， 资源文件如图片等在 ./android/app/src/main/res 目录中。

最后，将整个android项目打成release包即可。

IOS打包命令如下

```
react-native bundle --platform ios --dev false --entry-file index.js --bundle-outp
ut ./ios/bundle/index.ios.jsbundle --assets-dest ./ios/bundle/
```

需要注意的是

- ./ios 目录下需有bundle文件夹，如没有则手动创建，否则命令执行不成功。
- 执行命令成功后，需要的离线bundle js文件是 ./ios/bundle/index.ios.jsbundle ， 资源文件如图片等在

./ios/bundle/ 目录中。

打包完成后，需要将 js 文件 和资源文件夹手动引入项目（必须使用Create folder references的方式添加文件夹，参考https://www.jianshu.com/p/ce71b4a8a246），并在ios代码中修改bundle文件获取方式

AppDelegate.m

```
#import "AppDelegate.h"

#import <React/RCTBundleURLProvider.h>
#import <React/RCTRootView.h>

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NS
Dictionary *)launchOptions
{
  NSURL *jsCodeLocation;

 // jsCodeLocation = [[RCTBundleURLProvider sharedSettings] jsBundleURLForBundleRo
ot:@"index" fallbackResource:nil];
  jsCodeLocation = [[NSBundle mainBundle] URLForResource:@"bundle/index.ios" withE
xtension:@"jsbundle"];

  RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:jsCodeLocation
                                                      moduleName:@"RnDemo"
                                               initialProperties:nil
                                                   launchOptions:launchOptions];
  rootView.backgroundColor = [[UIColor alloc] initWithRed:1.0f green:1.0f blue:1.0
f alpha:1];

  self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
  UIViewController *rootViewController = [UIViewController new];
  rootViewController.view = rootView;
  self.window.rootViewController = rootViewController;
  [self.window makeKeyAndVisible];
  return YES;
}

@end
```