原生模块的开发与应用

> React Native 原生模块开发

原生模块

有两种方法可以为React Native应用程序编写原生模块：

1. 创建NPM包
2. 直接在React Native应用程序的iOS/Android项目中

Android 原生模块

Toast模块

首先创建一个原生模块。原生模块是一个继承了 `ReactContextBaseJavaModule` 的 Java 类，它可以实现一些 JavaScript 所需的功能。我们这里的目标是可以在 JavaScript 里写 `ToastExample.show('Awesome', ToastExample.SHORT);`，来调起一个短暂的 Toast 通知。

创建一个新的 Java 类并命名为 `ToastModule.java`，放置到 `android/app/src/main/java/com/your-app-name/` 目录下，其具体代码如下：

```java
// ToastModule.java
package com.your-app-name;

import android.widget.Toast;

import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;

import java.util.Map;
import java.util.HashMap;

public class ToastModule extends ReactContextBaseJavaModule {
  private static ReactApplicationContext reactContext;

  private static final String DURATION_SHORT_KEY = "SHORT";
  private static final String DURATION_LONG_KEY = "LONG";

```

```
  public ToastModule(ReactApplicationContext context) {
    super(context);
    reactContext = context;
  }
}
```

`ReactContextBaseJavaModule` 要求派生类实现 `getName` 方法。这个函数用于返回一个字符串名字，这个名字在 JavaScript 端标记这个模块。这里我们把这个模块叫做 `ToastExample` ，这样就可以在 JavaScript 中通过 `NativeModules.ToastExample` 访问到这个模块。

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

一个可选的方法 `getContants` 返回了需要导出给 JavaScript 使用的常量。它并不一定需要实现，但在定义一些可以被 JavaScript 同步访问到的预定义的值时非常有用。

```
@Override
public Map<String, Object getConstants() {
  final Map<String, Object constants = new HashMap<();
  constants.put(DURATION_SHORT_KEY, Toast.LENGTH_SHORT);
  constants.put(DURATION_LONG_KEY, Toast.LENGTH_LONG);
  return constants;
}
```

要导出一个方法给 JavaScript 使用，Java 方法需要使用注解 `@ReactMethod` 。方法的返回类型必须为 `void` 。React Native 的跨语言访问是异步进行的，所以想要给 JavaScript 返回一个值的唯一办法是使用回调函数或者发送事件（参见下文的描述）。

```
@ReactMethod
public void show(String message, int duration) {
  Toast.makeText(getReactApplicationContext(), message, duration).show();
}
```

注册模块

在 Java 这边要做的最后一件事就是注册这个模块。我们需要在应用的 Package 类的 `createNativeModules` 方法中添加这个模块。如果模块没有被注册，它也无法在 JavaScript 中

被访问到。

创建一个新的 Java 类并命名为 `CustomToastPackage.java`，放置到 `android/app/src/main/java/com/your-app-name/` 目录下，其具体代码如下：

```
// CustomToastPackage.java
package com.your-app-name;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CustomToastPackage implements ReactPackage {

  @Override
  public List<ViewManager createViewManagers(ReactApplicationContext
reactContext) {
    return Collections.emptyList();
  }

  @Override
  public List<NativeModule createNativeModules(
                                ReactApplicationContext reactContext) {
    List<NativeModule modules = new ArrayList<();

    modules.add(new ToastModule(reactContext));

    return modules;
  }

}
```

这个 package 需要在 `MainApplication.java` 文件的 `getPackages` 方法中提供。这个文件位于你的 react-native 应用文件夹的 android 目录中。具体路径是: `android/app/src/main/java/com/your-app-name/MainApplication.java`.

```
// MainApplication.java
import com.your-app-name.CustomToastPackage; // <-- 引入你自己的包

```

```
protected List<ReactPackage getPackages() {
  @SuppressWarnings("UnnecessaryLocalVariable")
  List<ReactPackage packages = new PackageList(this).getPackages();
  // Packages that cannot be autolinked yet can be added manually here, for example:
  // packages.add(new MyReactNativePackage());
  packages.add(new CustomToastPackage()); // <-- 添加这一行，类名替换成你的Package类的名字 name.
  return packages;
}
```

为了让你的功能从 JavaScript 端访问起来更为方便，通常我们都会把原生模块封装成一个 JavaScript 模块。这不是必须的，但省下了每次都从 `NativeModules` 中获取对应模块的步骤。这个 JS 文件也可以用于添加一些其他 JavaScript 端实现的功能。

```
import { NativeModules } from 'react-native';
// 下一句中的ToastExample即对应上文
// public String getName()中返回的字符串
export default NativeModules.ToastExample;
```

现在，在别处的 JavaScript 代码中可以这样调用你的方法：

```
import ToastExample from './ToastExample';

ToastExample.show('Awesome', ToastExample.SHORT);
```

参数类型

JAVA	JavaScript
Bool	Boolean
Integer	Number
Double	Number
Float	Number
String	String
Callback	function

ReadableMap	Object
ReadableArray	Array

注册模块

最后一件事就是注册这个模块。我们需要在应用的 Package 类的方法中添加这个模块。如果模块没有被注册，它也无法在 JavaScript 中被访问到。

创建一个新的 Java 类并命名为 `CustomToastPackage.java`，放置到目录下，其具体代码如下：`android/app/src/main/java/com/your-app-name/`

```
// CustomToastPackage.java

package com.your-app-name;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CustomToastPackage implements ReactPackage {

  @Override
  public List<ViewManager> createViewManagers(ReactApplicationContext
reactContext) {
    return Collections.emptyList();
  }

  @Override
  public List<NativeModule> createNativeModules(
                                ReactApplicationContext reactContext) {
    List<NativeModule> modules = new ArrayList<>();

    modules.add(new ToastModule(reactContext));

    return modules;
  }

}
```

这个文件需要在 `MainApplication.java` 文件中 `getPackages` 方法的提供。这个文件位于 android 目录中。具体路径是: `android/app/src/main/java/com/your-app-name/MainApplication.java`

```
// MainApplication.java
...
import com.your-app-name.CustomToastPackage; // <-- 引入你自己的包
...
protected List<ReactPackage> getPackages() {
  @SuppressWarnings("UnnecessaryLocalVariable")
  List<ReactPackage> packages = new PackageList(this).getPackages();
  // Packages that cannot be autolinked yet can be added manually here, for example:
  // packages.add(new MyReactNativePackage());
  packages.add(new CustomToastPackage()); // <-- 添加这一行，类名替换成你的Package类的名字 name.
  return packages;
}
```

通常我们都会把原生模块封装成一个 JavaScript 模块。（可选）

```
// ToastExample.js
/**
 * This exposes the native ToastExample module as a JS module. This has a
 * function 'show' which takes the following parameters:
 *
 * 1. String message: A string with the text to toast
 * 2. int duration: The duration of the toast. May be ToastExample.SHORT or
 *    ToastExample.LONG
 */
import { NativeModules } from 'react-native';
// 下一句中的ToastExample即对应上文
// public String getName()中返回的字符串
export default NativeModules.ToastExample;
```

基本上到这里就已经完成原生模块的开发。在别的地方可以这样调用你的方法：

```
import ToastExample from './ToastExample';

ToastExample.show('Awesome', ToastExample.SHORT);
```

桥接原生方法

回调函数

原生模块还支持一种特殊的参数——回调函数。它提供了一个函数来把返回值传回给 JavaScript。

```
import com.facebook.react.bridge.Callback;

public class UIManagerModule extends ReactContextBaseJavaModule {

  @ReactMethod
  public void measureLayout(
      int tag,
      int ancestorTag,
      Callback errorCallback,
      Callback successCallback) {
    try {
      measureLayout(tag, ancestorTag, mMeasureBuffer);
      float relativeX = PixelUtil.toDIPFromPixel(mMeasureBuffer[0]);
      float relativeY = PixelUtil.toDIPFromPixel(mMeasureBuffer[1]);
      float width = PixelUtil.toDIPFromPixel(mMeasureBuffer[2]);
      float height = PixelUtil.toDIPFromPixel(mMeasureBuffer[3]);
      successCallback.invoke(relativeX, relativeY, width, height);
    } catch (IllegalViewOperationException e) {
      errorCallback.invoke(e.getMessage());
    }
  }
}
```

在 JavaScript 可以里这样使用：

```
UIManager.measureLayout(
  100,
  100,
  (msg) => {
    console.log(msg);
  },
  (x, y, width, height) => {
    console.log(x + ':' + y + ':' + width + ':' + height);
  }
);
```

原生模块通常只应调用回调函数一次。但是，它可以保存 callback 并在将来调用。

请务必注意 callback 并非在对应的原生函数返回后立即被执行——注意跨语言通讯是异步的，这个执行过程会通过消息循环来进行。

Promises

原生模块还可以使用 promise 来简化代码，搭配 ES2016(ES7)标准的 `async/await` 语法则效果更佳。如果桥接原生方法是一个 `Promise` ，则对应的 JS 方法就会返回一个 Promise 对象。

我们把上面的代码用 promise 来代替回调进行重构：

```
import com.facebook.react.bridge.Promise;

public class UIManagerModule extends ReactContextBaseJavaModule {
  private static final String E_LAYOUT_ERROR = "E_LAYOUT_ERROR";

  @ReactMethod
  public void measureLayout(
      int tag,
      int ancestorTag,
      Promise promise) {
    try {
      measureLayout(tag, ancestorTag, mMeasureBuffer);

      WritableMap map = Arguments.createMap();

      map.putDouble("relativeX", PixelUtil.toDIPFromPixel(mMeasureBuffer[0]));
      map.putDouble("relativeY", PixelUtil.toDIPFromPixel(mMeasureBuffer[1]));
      map.putDouble("width", PixelUtil.toDIPFromPixel(mMeasureBuffer[2]));
      map.putDouble("height", PixelUtil.toDIPFromPixel(mMeasureBuffer[3]));

      promise.resolve(map);
    } catch (IllegalViewOperationException e) {
      promise.reject(E_LAYOUT_ERROR, e);
    }
  }
}
```

在 JavaScript 可以里这样使用：

虽然这样写着看起来像同步操作，但实际仍然是异步的，并不会阻塞执行来等待

```
async function measureLayout() {
  try {
    const { relativeX, relativeY, width, height } =
      await UIManager.measureLayout(100, 100);
```

```
5
6     console.log(
7       relativeX + ':' + relativeY + ':' + width + ':' + height
8     );
9   } catch (e) {
10    console.error(e);
11  }
12 }
13
14 measureLayout();
```

发送事件到 JavaScript

原生模块可以在没有被调用的情况下往 JavaScript 发送事件通知。最简单的办法就是通过 `RCTDeviceEventEmitter`，这可以通过 `ReactContext` 来获得对应的引用，像这样：

```
1  ...
2  import com.facebook.react.modules.core.DeviceEventManagerModule;
3  import com.facebook.react.bridge.WritableMap;
4  import com.facebook.react.bridge.Arguments;
5  ...
6  private void sendEvent(ReactContext reactContext,
7                         String eventName,
8                         @Nullable WritableMap params) {
9    reactContext
10       .getJSModule(DeviceEventManagerModule.RCTDeviceEventEmitter.class)
11       .emit(eventName, params);
12 }
13 @ReactMethod
14 public void addListener(String eventName) {
15   // Set up any upstream listeners or background tasks as necessary
16 }
17 @ReactMethod
18 public void removeListeners(Integer count) {
19   // Remove upstream listeners, stop unnecessary background tasks
20 }
21 ...
22 WritableMap params = Arguments.createMap();
23 params.putString("eventProperty", "someValue");
24 ...
25 sendEvent(reactContext, "EventReminder", params);
```

JavaScript 模块可以通过使用 `NativeEventEmitter` 模块来监听事件：

```
import { NativeEventEmitter, NativeModules } from 'react-native';
// ...

  componentDidMount() {
    // ...
    const eventEmitter = new NativeEventEmitter(NativeModules.ToastExample);
    this.eventListener = eventEmitter.addListener('EventReminder', (event) => {
       console.log(event.eventProperty) // "someValue"
    });
    // ...
  }
  componentWillUnmount() {
    this.eventListener.remove(); // 组件卸载时记得移除监听事件
  }
```

从 `startActivityForResult` 中获取结果

如果使用 `startActivityForResult` 调起了一个 activity 并想从其中获取返回结果，那么你需要监听 `onActivityResult` 事件。具体的做法是继承 `BaseActivityEventListener` 或是实现 `ActivityEventListener`。

推荐前一种做法，因为它相对来说不太会受到 API 变更的影响。然后你需要在模块的构造函数中注册这一监听事件。

```
reactContext.addActivityEventListener(mActivityResultListener);
```

现在你可以通过重写下面的方法来实现对 `onActivityResult` 的监听：

```
@Override
public void onActivityResult(
  final Activity activity,
  final int requestCode,
  final int resultCode,
  final Intent intent) {
  // 在这里实现你自己的逻辑
}
```

下面我们写一个简单的图片选择器来实践一下。这个图片选择器会把 `pickImage` 方法暴露给 JavaScript，而这个方法在调用时就会把图片的路径返回到 JS 端。

```
public class ImagePickerModule extends ReactContextBaseJavaModule {
```

```
2
3   private static final int IMAGE_PICKER_REQUEST = 467081;
4   private static final String E_ACTIVITY_DOES_NOT_EXIST =
"E_ACTIVITY_DOES_NOT_EXIST";
5   private static final String E_PICKER_CANCELLED = "E_PICKER_CANCELLED";
6   private static final String E_FAILED_TO_SHOW_PICKER =
"E_FAILED_TO_SHOW_PICKER";
7   private static final String E_NO_IMAGE_DATA_FOUND = "E_NO_IMAGE_DATA_FOUND";
8
9   private Promise mPickerPromise;
10
11  private final ActivityEventListener mActivityEventListener = new
BaseActivityEventListener() {
12
13    @Override
14    public void onActivityResult(Activity activity, int requestCode, int
resultCode, Intent intent) {
15      if (requestCode == IMAGE_PICKER_REQUEST) {
16        if (mPickerPromise != null) {
17          if (resultCode == Activity.RESULT_CANCELED) {
18            mPickerPromise.reject(E_PICKER_CANCELLED, "Image picker was
cancelled");
19          } else if (resultCode == Activity.RESULT_OK) {
20            Uri uri = intent.getData();
21
22            if (uri == null) {
23              mPickerPromise.reject(E_NO_IMAGE_DATA_FOUND, "No image data
found");
24            } else {
25              mPickerPromise.resolve(uri.toString());
26            }
27          }
28
29          mPickerPromise = null;
30        }
31      }
32    }
33  };
34
35  ImagePickerModule(ReactApplicationContext reactContext) {
36    super(reactContext);
37
38    // Add the listener for onActivityResult
39    reactContext.addActivityEventListener(mActivityEventListener);
40  }
41
42  @Override
```

```
  public String getName() {
    return "ImagePickerModule";
  }

  @ReactMethod
  public void pickImage(final Promise promise) {
    Activity currentActivity = getCurrentActivity();

    if (currentActivity == null) {
      promise.reject(E_ACTIVITY_DOES_NOT_EXIST, "Activity doesn't exist");
      return;
    }

    // Store the promise to resolve/reject when picker returns data
    mPickerPromise = promise;

    try {
      final Intent galleryIntent = new Intent(Intent.ACTION_PICK);

      galleryIntent.setType("image/*");

      final Intent chooserIntent = Intent.createChooser(galleryIntent, "Pick an image");

      currentActivity.startActivityForResult(chooserIntent, IMAGE_PICKER_REQUEST);
    } catch (Exception e) {
      mPickerPromise.reject(E_FAILED_TO_SHOW_PICKER, e);
      mPickerPromise = null;
    }
  }
}
```

监听生命周期事件

监听 activity 的生命周期事件（比如 `onResume`，`onPause` 等等）和我们在前面实现 `ActivityEventListener` 的做法类似。模块必须实现 `LifecycleEventListener`，然后需要在构造函数中注册一个监听函数：

```
reactContext.addLifecycleEventListener(this);
```

现在你可以通过实现下列方法来监听 activity 的生命周期事件了：

```java
@Override
public void onHostResume() {
    // Activity onResume
}

@Override
public void onHostPause() {
    // Activity onPause
}

@Override
public void onHostDestroy() {
    // Activity onDestroy
}
```

iOS 原生模块

参数类型

Objective-C	JavaScript
NSString	string
NSInteger	number
float	number
double	number
CGFloat	number
NSNumber	number
BOOL | NSNumber	boolean
NSArray	array
NSDictionary	object
RCTResponseSenderBlock	function

枚举常量

用`NS_ENUM`定义的枚举类型必须要先扩展对应的 RCTConvert 方法才可以作为函数参数传递。

假设我们要导出如下的`NS_ENUM`定义：

```
typedef NS_ENUM(NSInteger, UIStatusBarAnimation) {
    UIStatusBarAnimationNone,
    UIStatusBarAnimationFade,
    UIStatusBarAnimationSlide,
};
```

你需要这样来扩展 RCTConvert 类：

```
@implementation RCTConvert (StatusBarAnimation)
  RCT_ENUM_CONVERTER(UIStatusBarAnimation, (@{ @"statusBarAnimationNone" : @(UIStatusBarAnimationNone),
                                               @"statusBarAnimationFade" : @(UIStatusBarAnimationFade),
                                               @"statusBarAnimationSlide" : @(UIStatusBarAnimationSlide)}),
                     UIStatusBarAnimationNone, integerValue)
@end
```

接着你可以这样定义方法并且导出 enum 值作为常量：

```
- (NSDictionary *)constantsToExport
{
  return @{ @"statusBarAnimationNone" : @(UIStatusBarAnimationNone),
            @"statusBarAnimationFade" : @(UIStatusBarAnimationFade),
            @"statusBarAnimationSlide" : @(UIStatusBarAnimationSlide) };
};

RCT_EXPORT_METHOD(updateStatusBarAnimation:(UIStatusBarAnimation)animation
                                completion:(RCTResponseSenderBlock)callback)
```

你的枚举现在会用上面提供的选择器进行转换（上面的例子中是 `integerValue`），然后再传递给你导出的函数。

除此以外，任何 `RCTConvert` 类支持的的类型也都可以使用(参见 `RCTConvert` 了解更多信息)。`RCTConvert` 还提供了一系列辅助函数，用来接收一个 JSON 值并转换到原生 Objective-C 类型或类。

桥接原生方法

回调函数

原生模块还支持一种特殊的参数——回调函数。它提供了一个函数来把返回值传回给 JavaScript。

```objc
RCT_EXPORT_METHOD(findEvents:(RCTResponseSenderBlock)callback)
{
  NSArray *events = ...
  callback(@[[NSNull null], events]);
}
```

`RCTResponseSenderBlock` 只接受一个参数——传递给 JavaScript 回调函数的参数数组。在上面这个例子里我们用 Node.js 的常用习惯：第一个参数是一个错误对象（没有发生错误的时候为 null），而剩下的部分是函数的返回值。

```js
CalendarManager.findEvents((error, events) => {
  if (error) {
    console.error(error);
  } else {
    this.setState({ events: events });
  }
});
```

原生模块通常只应调用回调函数一次。但是，它可以保存 callback 并在将来调用。这在封装那些通过“委托函数”来获得返回值的 iOS API 时最为常见。

如果你想传递一个更接近 `Error` 类型的对象给 JavaScript，可以用 `RCTUtils.h` 提供的 `RCTMakeError` 函数。现在它仅仅是发送了一个和 Error 结构一样的 dictionary 给 JavaScript。

Promises

原生模块还可以使用 promise 来简化代码。如果桥接原生方法的最后两个参数是 `RCTPromiseResolveBlock` 和 `RCTPromiseRejectBlock`，则对应的 JS 方法就会返回一个 Promise 对象。

我们把上面的代码用 promise 来代替回调进行重构：

```objc
RCT_REMAP_METHOD(findEvents,
                 findEventsWithResolver:(RCTPromiseResolveBlock)resolve
                 rejecter:(RCTPromiseRejectBlock)reject)
{
  NSArray *events = ...
  if (events) {
    resolve(events);
  } else {
    NSError *error = ...
    reject(@"no_events", @"There were no events", error);
  }
}
```

现在 JavaScript 端的方法会返回一个 Promise。这样你就可以在一个声明了 `async` 的异步函数内使用 `await` 关键字来调用，并等待其结果返回。

虽然这样写着看起来像同步操作，但实际仍然是异步的，并不会阻塞执行来等待。

```
async function updateEvents() {
  try {
    const events = await CalendarManager.findEvents();

    this.setState({ events });
  } catch (e) {
    console.error(e);
  }
}

updateEvents();
```

多线程

- 原生模块不应对自己被调用时所处的线程做任何假设。React Native 在一个独立的串行 GCD 队列中调用原生模块的方法，但这属于实现的细节，并且可能会在将来的版本中改变。通过实现方法 `(dispatch_queue_t)methodQueue` ，原生模块可以指定自己想在哪个队列中被执行。具体来说，如果模块需要调用一些必须在主线程才能使用的 API，那应当这样指定：

```
- (dispatch_queue_t)methodQueue
{
  return dispatch_get_main_queue();
}
```

类似的，如果一个操作需要花费很长时间，原生模块不应该阻塞住，而是应当声明一个用于执行操作的独立队列。举个例子， `RCTAsyncLocalStorage` 模块创建了自己的一个 queue，这样它在做一些较慢的磁盘操作的时候就不会阻塞住 React 本身的消息队列：

```
- (dispatch_queue_t)methodQueue
{
  return dispatch_queue_create("com.facebook.React.AsyncLocalStorageQueue",
DISPATCH_QUEUE_SERIAL);
}
```

指定的 `methodQueue` 会被你模块里的所有方法共享。如果你的方法中“只有一个”是耗时较长的（或者是由于某种原因必须在不同的队列中运行的），你可以在函数体内用 `dispatch_async` 方法来在另一个队列执行，而不影响其他方法：

```
RCT_EXPORT_METHOD(doSomethingExpensive:(NSString *)param callback:(RCTResponseSenderBlock)callback)
{
  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // 在这里执行长时间的操作
    ...
    // 你可以在任何线程/队列中执行回调函数
    callback(@[...]);
  });
}
```

> 注意: 在模块之间共享分发队列
>
> `methodQueue` 方法会在模块被初始化的时候被执行一次，然后会被 React Native 的桥接机制保存下来，所以你不需要自己保存队列的引用，除非你希望在模块的其它地方使用它。但是，如果你希望在若干个模块中共享同一个队列，则需要自己保存并返回相同的队列实例；仅仅是返回相同名字的队列是不行的。

依赖注入

bridge 会自动注册实现了 `RCTBridgeModule` 协议的模块，但是你可能也希望能够自己去初始化自定义的模块实例（这样可以注入依赖）。

要实现这个功能，你需要实现 `RCTBridgeDelegate` 协议，初始化 `RCTBridge` ，并且在初始化方法里指定代理。然后用初始化好的 `RCTBridge` 实例初始化一个 `RCTRootView` 。

```
id<RCTBridgeDelegate moduleInitialiser =
[[classThatImplementsRCTBridgeDelegate alloc] init];

RCTBridge *bridge = [[RCTBridge alloc] initWithDelegate:moduleInitialiser
launchOptions:nil];

RCTRootView *rootView = [[RCTRootView alloc]
                    initWithBridge:bridge
                        moduleName:kModuleName
                 initialProperties:nil];
```

发送事件到 JavaScript

即使没有被 JavaScript 调用，原生模块也可以给 JavaScript 发送事件通知。最好的方法是继承 `RCTEventEmitter`，实现 `suppportEvents` 方法并调用 `self sendEventWithName:`。

```objc
// CalendarManager.h
#import <React/RCTBridgeModule.h>
#import <React/RCTEventEmitter.h>

@interface CalendarManager : RCTEventEmitter <RCTBridgeModule>

@end
```

```objc
// CalendarManager.m
#import "CalendarManager.h"

@implementation CalendarManager

RCT_EXPORT_MODULE();

- (NSArray<NSString *> *)supportedEvents
{
  return @[@"EventReminder"];
}

- (void)calendarEventReminderReceived:(NSNotification *)notification
{
  NSString *eventName = notification.userInfo[@"name"];
  [self sendEventWithName:@"EventReminder" body:@{@"name": eventName}];
}

@end
```

JavaScript 端的代码可以创建一个包含你的模块的 `NativeEventEmitter` 实例来订阅这些事件。

```javascript
import { NativeEventEmitter, NativeModules } from 'react-native';
const { CalendarManager } = NativeModules;

const calendarManagerEmitter = new NativeEventEmitter(CalendarManager);

const subscription = calendarManagerEmitter.addListener(
  'EventReminder',
  (reminder) => console.log(reminder.name)
);
```

```
...
// 取消订阅
subscription.remove();
```

优化无监听处理的事件

如果你发送了一个事件却没有任何监听处理，则会因此收到一个资源警告。要优化因此带来的额外开销，你可以在你的 `RCTEventEmitter` 子类中覆盖 `startObserving` 和 `stopObserving` 方法。

```
@implementation CalendarManager
{
  bool hasListeners;
}

// 在添加第一个监听函数时触发
-(void)startObserving {
    hasListeners = YES;
    // Set up any upstream listeners or background tasks as necessary
}

// Will be called when this module's last listener is removed, or on dealloc.
-(void)stopObserving {
    hasListeners = NO;
    // Remove upstream listeners, stop unnecessary background tasks
}

- (void)calendarEventReminderReceived:(NSNotification *)notification
{
  NSString *eventName = notification.userInfo[@"name"];
  if (hasListeners) { // Only send events if anyone is listening
    [self sendEventWithName:@"EventReminder" body:@{@"name": eventName}];
  }
}
```

导出常量

原生模块可以导出一些常量，这些常量在 JavaScript 端随时都可以访问。用这种方法来传递一些静态数据，可以避免通过 bridge 进行一次来回交互。

```
- (NSDictionary *)constantsToExport
{
  return @{ @"firstDayOfTheWeek": @"Monday" };
}
```

JavaScript 端可以随时同步地访问这个数据：

```
console.log(CalendarManager.firstDayOfTheWeek);
```

但是注意这个常量仅仅在初始化的时候导出了一次，所以即使你在运行期间改变 `constantToExport` 返回的值，也不会影响到 JavaScript 环境下所得到的结果。

原生UI组件

Android 原生UI组件

构建一个原生 UI 组件，了解 React Native 核心库中 `ImageView` 组件的具体实现。

ImageView 示例

在这个例子里，我们来看看为了让 JavaScript 中可以使用 ImageView，需要做哪些准备工作。

创建原生视图很简单：

1. 创建一个 ViewManager 的子类。

在这个例子里我们创建一个视图管理类 `ReactImageManager` ，它继承自 `SimpleViewManager<ReactImageView>` 。 `ReactImageView` 是这个视图管理类所管理的对象类型，也就是我们自定义的原生视图。 `getName` 方法返回的名字会用于在 JavaScript 端引用。

```
...

public class ReactImageManager extends SimpleViewManager<ReactImageView {

  public static final String REACT_CLASS = "RCTImageView";
  ReactApplicationContext mCallerContext;

  public ReactImageManager(ReactApplicationContext reactContext) {
    mCallerContext = reactContext;
  }

  @Override
  public String getName() {
    return REACT_CLASS;
  }
```

2. 实现 `createViewInstance` 方法。

视图在 `createViewInstance` 中创建，且应当把自己初始化为默认的状态。所有属性的设置都通过后续的 `updateView` 来进行。

```
1   @Override
2   public ReactImageView createViewInstance(ThemedReactContext context) {
3     return new ReactImageView(context, Fresco.newDraweeControllerBuilder(),
  null, mCallerContext);
4   }
```

3. 通过 `@ReactProp` （或 `@ReactPropGroup` ）注解来导出属性的设置方法。

要导出给 JavaScript 使用的属性，需要申明带有 `@ReactProp` （或 `@ReactPropGroup` ）注解的设置方法。方法的第一个参数是要修改属性的视图实例，第二个参数是要设置的属性值。方法的返回值类型必须为 `void` ，而且访问控制必须被声明为 `public` 。JavaScript 所得知的属性类型会由该方法第二个参数的类型来自动决定。

`@ReactProp` 注解必须包含一个字符串类型的参数 `name` 。这个参数指定了对应属性在 JavaScript 端的名字。

除了 `name` ， `@ReactProp` 注解还接受这些可选的参数： `defaultBoolean` , `defaultInt` , `defaultFloat` 。这些参数必须是对应的基础类型的值（也就是 `boolean` , `int` , `float` ），这些值会被传递给 setter 方法，以免 JavaScript 端某些情况下在组件中移除了对应的属性。注意这个"默认"值只对基本类型生效，对于其他的类型而言，当对应的属性删除时， `null` 会作为默认值提供给方法。

使用 `@ReactPropGroup` 来注解的设置方法和 `@ReactProp` 不同。请参见 `@ReactPropGroup` 注解类源代码中的文档来获取更多详情。

重在 ReactJS 里，修改一个属性会引发一次对设置方法的调用。有一种修改情况是，移除掉之前设置的属性。在这种情况下设置方法也一样会被调用，并且“默认”值会被作为参数提供（对于基础类型来说可以通过 `defaultBoolean` 、 `defaultFloat` 等 `@ReactProp` 的属性提供，而对于复杂类型来说参数则会设置为 `null` ）

```
1   @ReactProp(name = "src")
2   public void setSrc(ReactImageView view, @Nullable ReadableArray sources) {
3     view.setSource(sources);
4   }
5
6   @ReactProp(name = "borderRadius", defaultFloat = 0f)
7   public void setBorderRadius(ReactImageView view, float borderRadius) {
8     view.setBorderRadius(borderRadius);
9   }
10
11  @ReactProp(name = ViewProps.RESIZE_MODE)
```

```
  public void setResizeMode(ReactImageView view, @Nullable String resizeMode) {
    view.setScaleType(ImageResizeMode.toScaleType(resizeMode));
  }
```

4. 注册视图

在 Java 中的最后一步就是把视图控制器注册到应用中。这和原生模块的注册方法类似，唯一的区别是我们把它放到 `createViewManagers` 方法的返回值里。

```
  @Override
  public List<ViewManager createViewManagers(
                              ReactApplicationContext reactContext) {
    return Arrays.<ViewManagerasList(
      new ReactImageManager(reactContext)
    );
  }
```

完成上面这些代码后，请一定记得要重新编译！（运行 `yarn android` 命令）

5. 实现对应的 JavaScript 模块

整个过程的最后一步就是创建 JavaScript 模块并且定义 Java 和 JavaScript 之间的接口层。使用 `TypeScript` 来规范定义接口的具体结构

```
// ImageView.js

import { requireNativeComponent } from 'react-native';

/**
Composes View.
 *
src: string
borderRadius: number
resizeMode: 'cover' | 'contain' | 'stretch'
 */
module.exports = requireNativeComponent('RCTImageView');
```

`requireNativeComponent` 目前只接受一个参数，即原生视图的名字。

如果还需要做一些复杂的逻辑譬如事件处理，那么可以把原生组件用一个普通 React 组件封装。

事件

处理来自用户的事件，譬如缩放操作或者拖动，当一个原生事件发生的时候，它应该也能触发 JavaScript 端视图上的事件，这两个视图会依据 `getId()` 而关联在一起。

```
class MyCustomView extends View {
   ...
    public void onReceiveNativeEvent() {
       WritableMap event = Arguments.createMap();
       event.putString("message", "MyMessage");
       ReactContext reactContext = (ReactContext)getContext();
       reactContext.getJSModule(RCTEventEmitter.class).receiveEvent(
           getId(),
           "topChange",
           event);
    }
}
```

要把事件名 `topChange` 映射到 JavaScript 端的 `onChange` 回调属性上，需要在你的 `ViewManager` 中覆盖 `getExportedCustomBubblingEventTypeConstants` 方法，并在其中进行注册：

```
public class ReactImageManager extends SimpleViewManager<MyCustomView {
    ...
    public Map getExportedCustomBubblingEventTypeConstants() {
        return MapBuilder.builder()
            .put(
                "topChange",
                MapBuilder.of(
                    "phasedRegistrationNames",
                    MapBuilder.of("bubbled", "onChange")))
                    .build();
    }
}
```

这个回调会传递一个原生事件对象，一般来说我们会在封装组件里进行处理以便外部使用：

```
// MyCustomView.js

class MyCustomView extends React.Component {
  constructor(props) {
    super(props);
    this._onChange = this._onChange.bind(this);
  }
```

```
  _onChange(event: Event) {
    if (!this.props.onChangeMessage) {
      return;
    }
    this.props.onChangeMessage(event.nativeEvent.message);
  }
  render() {
    return (
      <RCTMyCustomView
        {...this.props}
        onChange={this._onChange}
      />
    );
  }
}

const RCTMyCustomView = requireNativeComponent(`RCTMyCustomView`);
```

与 Android Fragment 的整合实例

为了将现有的原生 UI 元素整合到 React Native 应用中，可能需要使用 `Android Fragments` 来对本地组件进行更精细的控制，而不是从 `ViewManager` 返回一个 `View`。如果你想在生命周期方法的帮助下添加与视图绑定的自定义逻辑，如 `onViewCreated`、`onPause`、`onResume`，你会用得到它。

1. 创建一个 `Fragment`

`MyFragment.java`

```
// replace with your package
package com.mypackage;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.fragment.app.Fragment;
// replace with your view's import
import com.mypackage.CustomView;

public class MyFragment extends Fragment {
    CustomView customView;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup parent, Bundle savedInstanceState) {
        super.onCreateView(inflater, parent, savedInstanceState);
        customView = new CustomView();
```

```
        return customView; // this CustomView could be any view that you want to render
    }
    @Override
    public void onViewCreated(View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        // do any logic that should happen in an onCreate method, e.g:
        // customView.onCreate(savedInstanceState);
    }
    @Override
    public void onPause() {
        super.onPause();
        // do any logic that should happen in an onPause method
        // e.g.: customView.onPause();
    }
    @Override
    public void onResume() {
        super.onResume();
       // do any logic that should happen in an onResume method
       // e.g.: customView.onResume();
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        // do any logic that should happen in an onDestroy method
        // e.g.: customView.onDestroy();
    }
}
```

2. 创建 `ViewManager` 子类

```
// replace with your package
package com.mypackage;
import android.view.Choreographer;
import android.view.View;
import android.widget.FrameLayout;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.fragment.app.FragmentActivity;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReadableArray;
import com.facebook.react.common.MapBuilder;
import com.facebook.react.uimanager.annotations.ReactProp;
import com.facebook.react.uimanager.annotations.ReactPropGroup;
import com.facebook.react.uimanager.ViewGroupManager;
```

```java
import com.facebook.react.uimanager.ThemedReactContext;
import java.util.Map;

public class MyViewManager extends ViewGroupManager<FrameLayout> {
  public static final String REACT_CLASS = "MyViewManager";
  public final int COMMAND_CREATE = 1;
  ReactApplicationContext reactContext;
  public MyViewManager(ReactApplicationContext reactContext) {
    this.reactContext = reactContext;
  }
  @Override
  public String getName() {
    return REACT_CLASS;
  }
  /**
   * 返回一个 FrameLayout
   */
  @Override
  public FrameLayout createViewInstance(ThemedReactContext reactContext) {
    return new FrameLayout(reactContext);
  }

  @Nullable
  @Override
  public Map<String, Integer getCommandsMap() {
    return MapBuilder.of("create", COMMAND_CREATE);
  }

  @Override
  public void receiveCommand(@NonNull FrameLayout root, String commandId,
@Nullable ReadableArray args) {
    super.receiveCommand(root, commandId, args);
    int reactNativeViewId = args.getInt(0);
    int commandIdInt = Integer.parseInt(commandId);
    switch (commandIdInt) {
      case COMMAND_CREATE:
        createFragment(root, reactNativeViewId);
        break;
      default: {}
    }
  }

  public void createFragment(FrameLayout root, int reactNativeViewId) {
    ViewGroup parentView = (ViewGroup)
root.findViewById(reactNativeViewId).getParent();
    setupLayout(parentView);
    final MyFragment myFragment = new MyFragment();
```

```
60      FragmentActivity activity = (FragmentActivity)
    reactContext.getCurrentActivity();
61      activity.getSupportFragmentManager()
62              .beginTransaction()
63              .replace(reactNativeViewId, myFragment,
    String.valueOf(reactNativeViewId))
64              .commit();
65    }
66
67    public void setupLayout(View view) {
68      Choreographer.getInstance().postFrameCallback(new
    Choreographer.FrameCallback() {
69        @Override
70        public void doFrame(long frameTimeNanos) {
71          manuallyLayoutChildren(view);
72          view.getViewTreeObserver().dispatchOnGlobalLayout();
73          Choreographer.getInstance().postFrameCallback(this);
74        }
75      });
76    }
77
78    public void manuallyLayoutChildren(View view) {
79        int width = propWidth;
80        int height = propHeight;
81        view.measure(
82                View.MeasureSpec.makeMeasureSpec(width,
    View.MeasureSpec.EXACTLY),
83                View.MeasureSpec.makeMeasureSpec(height,
    View.MeasureSpec.EXACTLY));
84        view.layout(0, 0, width, height);
85    }
```

3. 注册 `ViewManager`

```
1  package com.mypackage;
2  import com.facebook.react.ReactPackage;
3  import com.facebook.react.bridge.ReactApplicationContext;
4  import com.facebook.react.uimanager.ViewManager;
5  import java.util.Arrays;
6  import java.util.List;
7  public class MyPackage implements ReactPackage {
8     @Override
9     public List<ViewManager createViewManagers(ReactApplicationContext
    reactContext) {
10          return Arrays.<ViewManagerasList(
```

```
            new MyViewManager(reactContext)
        );
    }
}
```

4. 注册 `Package`

`MainApplication.java`

```
    @Override
    protected List<ReactPackage getPackages() {
      List<ReactPackage packages = new PackageList(this).getPackages();
      ...
      packages.add(new MyPackage());
      return packages;
    }
```

5. 执行 JavaScript 模块

- `MyViewManager.jsx`

```
import { requireNativeComponent } from 'react-native';
export const MyViewManager =
  requireNativeComponent('MyViewManager');
```

- `MyView.jsx` 调用 `create` 方法

```
import React, { useEffect, useRef } from 'react';
import { UIManager, findNodeHandle } from 'react-native';
import { MyViewManager } from './my-view-manager';

const createFragment = (viewId) =>
  UIManager.dispatchViewManagerCommand(
    viewId,
    UIManager.MyViewManager.Commands.create.toString(), // we are calling the 'create' command
    [viewId]
  );

export const MyView = ({ style }) => {
  const ref = useRef(null);

```

```
  useEffect(() => {
    const viewId = findNodeHandle(ref.current);
    createFragment(viewId!);
  }, []);

  return (
    <MyViewManager
      style={{
        ...(style || {}),
        height: style && style.height !== undefined ? style.height || '100%',
        width: style && style.width !== undefined ? style.width || '100%'
      }}
      ref={ref}
    />
  );
};
```

iOS 原生UI组件

构建一个原生 UI 组件，了解 React Native 核心库中 `MapView` 组件的具体实现。

iOS MapView 示例

假设我们要把地图组件植入到我们的 App 中，我们需要用到的是 `MKMapView`，而现在只需要让它可以在 Javascript 端使用。

提供原生视图很简单：

- 首先创建一个 `RCTViewManager` 的子类。
- 添加 `RCT_EXPORT_MODULE()` 宏标记。
- 实现 `-(UIView *)view` 方法。

```
// RNTMapManager.m
import <MapKit/MapKit.h

import <React/RCTViewManager.h

@interface RNTMapManager : RCTViewManager
@end

@implementation RNTMapManager

RCT_EXPORT_MODULE(RNTMap)

- (UIView *)view
{
```

```
  return [[MKMapView alloc] init];
}

@end
```

🎉 注意： 请不要在 `view` 中给 `UIView` 实例设置 `frame` 或是 `backgroundColor` 属性。为了和js端的布局属性一致，React Native 会覆盖你所设置的值。 如果您需要这种粒度的操作的话，比较好的方法是用另一个 `UIView` 来封装你想操作的 `UIView` 实例，并返回外层的 `UIView` 。

完成上面这些代码后，请一定记得要重新编译！（运行 `yarn ios` 命令）

接下来你需要一些 Javascript 代码来让这个视图变成一个可用的 React 组件：

```
// MapView.js

import { requireNativeComponent } from 'react-native';

// requireNativeComponent 自动把'RNTMap'解析为'RNTMapManager'
export default requireNativeComponent('RNTMap');

// MyApp.js

import MapView from './MapView.js';

...

render() {
  return <MapView style={{ flex: 1 }} />;
}
```

注意： 在渲染时，不要忘记布局视图，否则您只能面对一个空荡荡的屏幕。

```
  render() {
    return <MapView style={{flex: 1}} />;
  }
```

现在我们就已经实现了一个完整功能的地图组件了，诸如捏放和其它的手势都已经完整支持。但是现在还不能真正的在 Javascript 端控制

属性

封装原生属性供 Javascript 使用。

举例来说，我们希望能够禁用手指捏放操作，然后指定一个初始的地图可见区域。

禁用捏放操作只需要一个布尔值类型的属性就行了，所以我们添加这么一行：

```
// RNTMapManager.m
RCT_EXPORT_VIEW_PROPERTY(zoomEnabled, BOOL)
```

现在要想禁用捏放操作，我们只需要在 JS 里设置对应的属性：

```
// MyApp.js
<MapView zoomEnabled={false} style={{ flex: 1 }} />
```

但这样并不能很好的说明这个组件的用法

使用 `Typescript` 来定义 `Props` 值

```
// MapView.js
import PropTypes from 'prop-types';
import React from 'react';

import { requireNativeComponent } from 'react-native';

type Props = {
    zoomEnabled: boolean
}

function MapView(props:Props) {
  return <RNTMap {...props} />;
}

const RNTMap = requireNativeComponent('RNTMap', MapView);

export default MapView;
```

把 `requireNativeComponent` 的第二个参数从 null 变成了用于封装的组件 `MapView`。这使得 React Native 的底层框架可以检查原生属性和包装类的属性是否一致，来减少出现问题的可能。

现在，让我们添加一个更复杂些的 `region` 属性。我们首先添加原生代码：

```
// RNTMapManager.m
RCT_CUSTOM_VIEW_PROPERTY(region, MKCoordinateRegion, MKMapView)
{
  [view setRegion:json ? [RCTConvert MKCoordinateRegion:json] :
defaultView.region animated:YES];
}
```

这段代码比刚才的一个简单的 `BOOL` 要复杂的多了。现在我们多了一个需要做类型转换的 `MKCoordinateRegion` 类型，还添加了一部分自定义的代码，这样当我们在 JS 里改变地图的可视区域的时候，视角会平滑地移动过去。在我们提供的函数体内，`json` 代表了 JS 中传递的尚未解析的原始值。函数里还有一个 `view` 变量，使得我们可以访问到对应的视图实例。最后，还有一个 `defaultView` 对象，这样当 JS 给我们发送 null 的时候，可以把视图的这个属性重置回默认值。

为视图编写任何你所需要的转换函数

下面就是用 `RCTConvert` 实现的 `MKCoordinateRegion`。它使用了 ReactNative 中已经存在的 `RCTConvert+CoreLocation`：

```
// RNTMapManager.m

import "RCTConvert+Mapkit.h"

// RCTConvert+Mapkit.h

import <MapKit/MapKit.h
import <React/RCTConvert.h
import <CoreLocation/CoreLocation.h
import <React/RCTConvert+CoreLocation.h

@interface RCTConvert (Mapkit)

+ (MKCoordinateSpan)MKCoordinateSpan:(id)json;
+ (MKCoordinateRegion)MKCoordinateRegion:(id)json;

@end

@implementation RCTConvert(MapKit)

+ (MKCoordinateSpan)MKCoordinateSpan:(id)json
{
  json = [self NSDictionary:json];
  return (MKCoordinateSpan){
    [self CLLocationDegrees:json[@"latitudeDelta"]],
    [self CLLocationDegrees:json[@"longitudeDelta"]]
  };
```

```
}

+ (MKCoordinateRegion)MKCoordinateRegion:(id)json
{
  return (MKCoordinateRegion){
    [self CLLocationCoordinate2D:json],
    [self MKCoordinateSpan:json]
  };
}

@end
```

为了完成 `region` 属性的支持，我们还需要在 `propTypes` 里添加相应的说明（否则我们会立刻收到一个错误提示），然后就可以像使用其他属性一样使用了：

```
// MapView.js

type Props = {
    zoomEnabled: boolean
    region: {
        latitude: number,
        longitude: number,
        latitudeDelta: number,
        longitudeDelta: number,
    }
}

// MyApp.js

render() {
  const region = {
    latitude: 37.48,
    longitude: -122.16,
    latitudeDelta: 0.1,
    longitudeDelta: 0.1,
  };
  return (
    <MapView
      region={region}
      zoomEnabled={false}
      style={{ flex: 1 }}
    />
  );
}
```

原生组件有一些特殊的属性希望导出，但并不希望它成为公开的接口。

举个例子：

`Switch` 组件可能会有一个 `onChange` 属性用来传递原始的原生事件，然后导出一个 `onValueChange` 属性，这个属性在调用的时候会带上 `Switch` 的状态作为参数之一。这样的话你可能不希望原生专用的属性出现在 API 之中，也就不希望把它放到 `Props` 里。可是如果你不放的话，又会出现一个报错。解决方案就是带上额外的 `nativeOnly` 参数，像这样：

```
const RCTSwitch = requireNativeComponent('RCTSwitch', Switch, {
  nativeOnly: { onChange: true }
});
```

事件

处理来自用户的事件，譬如缩放操作或者拖动来改变可视区域

截至目前，我们从 manager 的 `-(UIView *)view` 方法返回了 `MKMapView` 实例。我们没法直接为 `MKMapView` 添加新的属性，所以我们只能创建一个 `MKMapView` 的子类用于我们自己的视图中。我们可以在这个子类中添加 `onRegionChange` 回调方法：

```
// RNTMapView.h

import <MapKit/MapKit.h

import <React/RCTComponent.h

@interface RNTMapView: MKMapView

@property (nonatomic, copy) RCTBubblingEventBlock onRegionChange;

@end

// RNTMapView.m

import "RNTMapView.h"

@implementation RNTMapView

@end
```

需要注意的是，所有 `RCTBubblingEventBlock` 必须以 `on` 开头。然后在 `RNTMapManager` 上声明一个事件处理函数属性，将其作为所暴露出来的所有视图的委托，并调用本地视图的事件处理将事件转发至 JS。

```objc
// RNTMapManager.m

#import <MapKit/MapKit.h>
#import <React/RCTViewManager.h>

#import "RNTMapView.h"
#import "RCTConvert+Mapkit.h"

@interface RNTMapManager : RCTViewManager <MKMapViewDelegate>
@end

@implementation RNTMapManager

RCT_EXPORT_MODULE()


RCT_EXPORT_VIEW_PROPERTY(zoomEnabled, BOOL)
RCT_EXPORT_VIEW_PROPERTY(onRegionChange, RCTBubblingEventBlock)

RCT_CUSTOM_VIEW_PROPERTY(region, MKCoordinateRegion, MKMapView)
{
        [view setRegion:json ? [RCTConvert MKCoordinateRegion:json] :
defaultView.region animated:YES];
}

(UIView *)view
{
  RNTMapView *map = [RNTMapView new];
  map.delegate = self;
  return map;
}

#pragma mark MKMapViewDelegate

(void)mapView:(RNTMapView *)mapView regionDidChangeAnimated:(BOOL)animated
{
  if (!mapView.onRegionChange) {
    return;
  }

  MKCoordinateRegion region = mapView.region;
  mapView.onRegionChange(@{
    @"region": @{
      @"latitude": @(region.center.latitude),
      @"longitude": @(region.center.longitude),
      @"latitudeDelta": @(region.span.latitudeDelta),
```

```
      @"longitudeDelta": @(region.span.longitudeDelta),
    }
  });
}
@end
```

在 `regionDidChangeAnimated:` 中，根据对应的视图调用事件处理函数并传递区域数据。调用 `onRegionChange` 事件会触发 JavaScript 端的同名回调函数。这个回调会传递原生事件对象，然后我们通常都会在封装组件里来处理这个对象，以使 API 更简明：

```
// MapView.js

type Props = {
    onRegionChange: (data: object) => void
}

function MapView(props: Props) {
    _onRegionChange = (event) => {
        if (!props.onRegionChange) {
          return;
        }

        .props.onRegionChange(event.nativeEvent);
    }
    return (
      <RNTMap
        {...this.props}
        onRegionChange={_onRegionChange}
      />
    );
}

// MyApp.js

class MyApp extends React.Component {
  onRegionChange(event) {
  }

  render() {
    const region = {
      latitude: 37.48,
      longitude: -122.16,
      latitudeDelta: 0.1,
      longitudeDelta: 0.1,
    };
```

```
    return (
      <MapView
        region={region}
        zoomEnabled={false}
        onRegionChange={this.onRegionChange}
      />
    );
  }
}
```

处理多个视图

React Native 视图在视图树中可以有多个子视图，例如。

```
<View
  <MyNativeView />
  <MyNativeView />
  <Button />
</View
```

在这个例子中，`MyNativeView` 类是 `NativeComponent` 的一个包装器，并公开了一些方法，这些方法将在 iOS 平台上被调用。`MyNativeView` 类在 `MyNativeView.ios.js` 文件中定义，其中包含 `NativeComponent` 的代理方法。

当用户与组件交互时（例如点击按钮），`MyNativeView` 的 `backgroundColor` 会发生变化。在这种情况下，`UIManager` 并不知道应该处理哪个 `MyNativeView` 以及应该更改哪个 `backgroundColor`。下面您将找到此问题的解决方案：

```
<View
<MyNativeView ref={this.myNativeReference}/>
<MyNativeView ref={this.myNativeReference2}/>
<Button onPress={() => { this.myNativeReference.callNativeMethod() }}/>
</View
```

现在，上述组件具有对特定 `MyNativeView` 的引用，这允许我们使用特定的 `MyNativeView` 实例。现在，按钮可以控制哪个 `MyNativeView` 应该更改其 `backgroundColor`。在本示例中，我们假设 `callNativeMethod` 更改 `backgroundColor`。

`MyNativeView.ios.js` 包含以下代码：

```
class MyNativeView extends React.Component< {
  callNativeMethod = () => {
```

```
    UIManager.dispatchViewManagerCommand(
      ReactNative.findNodeHandle(this),
      UIManager.getViewManagerConfig('RNCMyNativeView').Commands
        .callNativeMethod,
      []
    );
  };
  render() {
    return <NativeComponent ref={NATIVE_COMPONENT_REF} />;
  }
}
```

`callNativeMethod是我们自定义的 iOS 方法，它可以通过MyNativeView修改backgroundColor。这个方法使用UIManager.dispatchViewManagerCommand，需要传入三个参数:`

- `(nonnull NSNumber \*)reactTag` - React 视图的标识符。
- `commandID:(NSInteger)commandID` - 要调用的本地方法的 ID。
- `commandArgs:(NSArray<id> \*)commandArgs` - 我们可以从JS传递给本地方法的参数。

`RNCMyNativeViewManager.m`

```
import <React/RCTViewManager.h
import <React/RCTUIManager.h
import <React/RCTLog.h

RCT_EXPORT_METHOD(callNativeMethod:(nonnull NSNumber*) reactTag) {
  [self.bridge.uiManager addUIBlock:^(RCTUIManager *uiManager,
NSDictionary<NSNumber *,UIView * *viewRegistry) {
      NativeView *view = viewRegistry[reactTag];
      if (!view || ![view isKindOfClass:[NativeView class]]) {
          RCTLogError(@"Cannot find NativeView with tag #%@", reactTag);
          return;
      }
      [view callNativeMethod];
  }];
}
```

在`RNCMyNativeViewManager.m`文件中定义了`callNativeMethod`，它只包含一个参数，即`(nonnull NSNumber*) reactTag`。这个导出函数将使用`addUIBlock`查找包含`viewRegistry`参数的特定视图，并基于`reactTag`返回组件，从而允许它在正确的组件上调用方法。

样式

因为我们所有的视图都是 `UIView` 的子类，大部分的样式属性应该直接就可以生效。但有一部分组件会希望使用自己定义的默认样式，例如 `UIDatePicker` 希望自己的大小是固定的。这个默认属性对于布局算法的正常工作来说很重要，但我们也希望在使用这个组件的时候可以覆盖这些默认的样式。`DatePickerIOS` 实现这个功能的办法是通过封装一个拥有弹性样式的额外视图，然后在内层的视图上应用一个固定样式（通过原生传递来的常数生成）：

```
// DatePickerIOS.ios.js

import { UIManager } from 'react-native';
const RCTDatePickerIOSConsts = UIManager.RCTDatePicker.Constants;
...
  render: function() {
    return (
      <View style={this.props.style}
        <RCTDatePickerIOS
          ref={DATEPICKER}
          style={styles.rkDatePickerIOS}
          ...
        />
      </View
    );
  }
});

const styles = StyleSheet.create({
  rkDatePickerIOS: {
    height: RCTDatePickerIOSConsts.ComponentHeight,
    width: RCTDatePickerIOSConsts.ComponentWidth,
  },
});
```

常量 `RCTDatePickerIOSConsts` 在原生代码中导出，从一个组件的实际布局上获取到：

```
// RCTDatePickerManager.m

- (NSDictionary *)constantsToExport
{
  UIDatePicker *dp = [[UIDatePicker alloc] init];
  [dp layoutIfNeeded];

  return @{
```

```
    @"ComponentHeight": @(CGRectGetHeight(dp.frame)),
    @"ComponentWidth": @(CGRectGetWidth(dp.frame)),
    @"DatePickerModes": @{
      @"time": @(UIDatePickerModeTime),
      @"date": @(UIDatePickerModeDate),
      @"datetime": @(UIDatePickerModeDateAndTime),
    }
  };
}
```

实战项目

海康视频监控插件

> 采用NPM包形式

通话助手模块

> 直接在Android项目中创建Android模块