

# React Native 从入门到原理



作者 bestswifter (/u/3e55748920d2) [+关注](#)

2016.06.11 16:31\* 字数 6383 阅读 36977 评论 61 喜欢 340

(/u/3e55748920d2)

React Native 是最近非常火的一个话题，介绍如何利用 React Native 进行开发的文章和书籍多如牛毛，但面向入门水平并介绍它工作原理的文章却寥寥无几。

本文分为两个部分：上半部分用通俗的语言解释了相关的名词，重点介绍 React Native 出现的背景和试图解决的问题。适合新手对 React Native 形成初步了解。(事实证明，女票能看懂这段)

下半部分则通过源码（0.27 版本）分析 React Native 的工作原理，适合深入学习理解 React Native 的运行机制。最后则是我个人对 React Native 的分析与前景判断。

## 动态配置

由于 AppStore 审核周期的限制，如何动态的更改 app 成为了永恒的话题。无论采用何种方式，我们的流程总是可以归结为以下三部曲：“从 Server 获取配置 --> 解析 --> 执行native代码”。

很多时候，我们自觉或者不自觉的利用 JSON 文件实现动态配置的效果，它的核心流程是：

1. 通过 HTTP 请求获取 JSON 格式的配置文件。
2. 配置文件中标记了每一个元素的属性，比如位置，颜色，图片 URL 等。
3. 解析完 JSON 后，我们调用 Objective-C 的代码，完成 UI 控件的渲染。

通过这种方法，我们实现了在后台配置 app 的展示样式。从本质上来说，移动端和服务端约定了一套协议，但是协议内容严重依赖于应用内要展示的内容，不利于拓展。也就是说，如果业务要求频繁的增加或修改页面，这套协议很难应付。

最重要的是，JSON 只是一种数据交换的格式，说白了，我们就是在解析文本数据。这就意味着它只适合提供一些配置信息，而不方便提供逻辑信息。举个例子，我们从后台可以配置颜色，位置等信息，但如果想要控制 app 内的业务逻辑，就非常复杂了。

记住，我们只是在解析字符串，它完全不具备运行和调试的能力。

## React

不妨暂时抛弃移动端的烦恼，来看看前端的“新玩意”。

## 背景

作为前端小白，我以前对前端的理解是这样的：

- 用 HTML 创建 DOM，构建整个网页的布局、结构
- 用 CSS 控制 DOM 的样式，比如字体、字号、颜色、居中等

- 用 JavaScript 接受用户事件，动态的操控 DOM

在这三者的配合下，几乎所有页面上的功能都能实现。但也有比较不爽地方，比如我想动态修改一个按钮的文字，我需要这样写：

```
<button type="button" id="button" onclick="onClick()">old button</button>
```

然后在 JavaScript 中操作 DOM：

```
<script>
function onClick() {
  document.getElementById('button').innerHTML='new button';
}
</script>
```

可以看到，在 HTML 和 JavaScript 代码中，id 和 onclick 事件触发的函数必须完全对应，否则就无法正确的响应事件。如果想知道一个 HTML 标签会如何被响应，我们还得跑去 JavaScript 代码中查找，这种原始的配置方式让我觉得非常不爽。

## 初识 React

随着 FaceBook 推出了 React 框架，这个问题得到了大幅度改善。我们可以把一组相关的 HTML 标签，也就是 app 内的 UI 控件，封装进一个组件(Component)中，我从阮一峰的 React 教程 (<http://www.ruanyifeng.com/blog/2015/03/react.html>)中摘录了一段代码：

```
var MyComponent = React.createClass({
  handleClick: function() {
    this.refs.myTextInput.focus();
  },
  render: function() {
    return (
      <div>
        <input type="text" ref="myTextInput" />
        <input type="button" value="Focus the text input" onClick={this.handleClick} />
      </div>
    );
  }
});
```

如果你想问：“为什么 JavaScript 代码里面出现了 HTML 的语法”，那么恭喜你已经初步体会到 React 的奥妙了。这种语法被称为 JSX，它是一种 JavaScript 语法拓展。JSX 允许我们写 HTML 标签或 React 标签，它们终将被转换成原生的 JavaScript 并创建 DOM。

在 React 框架中，除了可以用 JavaScript 写 HTML 以外，我们甚至可以写 CSS，这在后面的例子中可以看到。

## 理解 React

前端界总是喜欢创造新的概念，仿佛谁说的名词更晦涩，谁的水平就越高。如果你和当时的我一样，听到 React 这个概念一脸懵逼的话，只要记住以下定义即可：

React 是一套可以用简洁的语法高效绘制 DOM 的框架

上文已经解释过了何谓“简洁的语法”，因为我们可以暂时放下 HTML 和 CSS，只关心如何 JavaScript 构造页面。

所谓的“高效”，是因为 React 独创了 Virtual DOM 机制。Virtual DOM 是一个存在于内存中的 JavaScript 对象，它与 DOM 是一一对应的关系，也就是说只要有 Virtual DOM，我们就能渲染出 DOM。

当界面发生变化时，得益于高效的 DOM Diff 算法，我们能够知道 Virtual DOM 的变化，从而高效的改动 DOM，避免了重新绘制 DOM。

当然，React **并不是前端开发的全部**。从之前的描述也能看出，它专注于 UI 部分，对应到 MVC 结构中就是 View 层。要想实现完整的 MVC 架构，还需要 Model 和 Controller 的结构。在前端开发时，我们可以采用 Flux 和 Redux 架构，它们并非框架(Library)，而是和 MVC 一样都是一种架构设计(Architecture)。

如果不从事前端开发，就不用深入的掌握 Flux 和 Redux 架构，但理解这一套体系结构对于后面理解 React Native **非常重要**。

## React Native

分别介绍完了移动端和前端的背景知识后，本文的主角——React Native 终于要登场了。

### 融合

前面我们介绍了移动端通过 JSON 文件传递信息的不足之处：只能传递配置信息，无法表达逻辑。从本质上讲，这是因为 JSON 毕竟只是纯文本，它缺乏像编程语言那样的运行能力。

而 React 在前端取得突破性成功以后，JavaScript 布道者们开始试图一统三端。他们利用了移动平台能够运行 JavaScript 代码的能力，并且发挥了 JavaScript 不仅仅可以传递配置信息，还可以表达逻辑信息的优点。

当痛点遇上特点，两者一拍即合，于是乎：

一个基于 JavaScript，具备动态配置能力，面向前端开发者的移动端开发框架，React Native，诞生了！

看到了么，这是一个面向前端开发者的框架。它的宗旨是让前端开发者像用 React 写网页那样，用 React Native 写移动端应用。这就是为什么 React Native 自称：

Learn once, Write anywhere!

而非很多跨平台语言，项目所说的：

Write once, Run anywhere!

React Native 希望前端开发者学习完 React 后，能够用同样的语法、工具等，分别开发安卓和 iOS 平台的应用并且不用一行原生代码。

如果用一个词概括 React Native，那就是：**Native 版本的 React**。

## 原理概述

React Native 不是黑科技，我们写的代码总是以一种非常合理，可以解释的方式的运行着，只是绝大多数人没有理解而已。接下来我以 iOS 平台为例，简单的解释一下 React Native 的原理。

首先要明白的一点是，即使使用了 React Native，我们依然需要 UIKit 等框架，调用的是 Objective-C 代码。总之，JavaScript 只是辅助，它只是提供了配置信息和逻辑的处理结果。React Native 与 Hybrid **完全没有关系**，它只不过是以 JavaScript 的形式告诉 Objective-C 该执行什么代码。

其次，React Native 能够运行起来，全靠 Objective-C 和 JavaScript 的交互。对于没有接触过 JavaScript 的人来说，非常有必要理解 JavaScript 代码如何被执行。

我们知道 C 系列的语言，经过编译，链接等操作后，会得到一个二进制格式的可执行文，所谓的运行程序，其实是运行这个二进制程序。

而 JavaScript 是一种脚本语言，它不会经过编译、链接等操作，而是在运行时才动态的进行词法、语法分析，生成抽象语法树(AST)和字节码，然后由解释器负责执行或者使用 JIT 将字节码转化为机器码再执行。整个流程由 JavaScript 引擎负责完成。

苹果提供了一个叫做 JavaScript Core 的框架，这是一个 JavaScript 引擎。通过下面这段代码可以简单的感受一下 Objective-C 如何调用 JavaScript 代码：

```
JSContext *context = [[JSContext alloc] init];
JSValue *jsVal = [context evaluateScript:@"21+7"];
int iVal = [jsVal toInt32];
```

这里的 JSContext 指的是 JavaScript 代码的运行环境，通过 evaluateScript 即可执行 JavaScript 代码并获取返回结果。

JavaScript 是一种单线程的语言，它不具备自运行的能力，因此总是被动调用。很多介绍 React Native 的文章都会提到“JavaScript 线程”的概念，实际上，它表示的是 Objective-C 创建了一个单独的线程，这个线程只用于执行 JavaScript 代码，而且 JavaScript 代码只会在这个线程中执行。

## Objective-C 与 JavaScript 交互

提到 Objective-C 与 JavaScript 的交互，不得不推荐 bang神的这篇文章：React Native 通信机制详解 (<http://blog.cnbang.net/tech/2698/>)。虽然其中不少细节都已经过时，但是整体的思路值得学习。

本节主要分析 Objective-C 与 JavaScript 交互时的整理逻辑与流程，下一节将通过源码来分析具体原理。

## JavaScript 调用 Objective-C

由于 JavaScript Core 是一个面向 Objective-C 的框架，在 Objective-C 这一端，我们对 JavaScript 上下文知根知底，可以很容易的获取到对象，方法等各种信息，当然也包括调用 JavaScript 函数。

真正复杂的问题在于，JavaScript 不知道 Objective-C 有哪些方法可以调用。

React Native 解决这个问题的方案是在 Objective-C 和 JavaScript 两端都保存了一份配置表，里面标记了所有 Objective-C 暴露给 JavaScript 的模块和方法。这样，无论是哪一方调用另一方的方法，实际上传递的数据只有 `ModuleId`、`MethodId` 和 `Arguments` 这三个元素，它们分别表示类、方法和方法参数，当 Objective-C 接收到这三个值后，就可以通过 runtime 唯一确定要调用的是哪个函数，然后调用这个函数。

再次重申，上述解决方案只是一个抽象概念，可能与实际的解决方案有微小差异，比如实际上 Objective-C 这一端，并没有直接保存这个模块配置表。具体实现将在下一节中随着源码一起分析。

## 闭包与回调

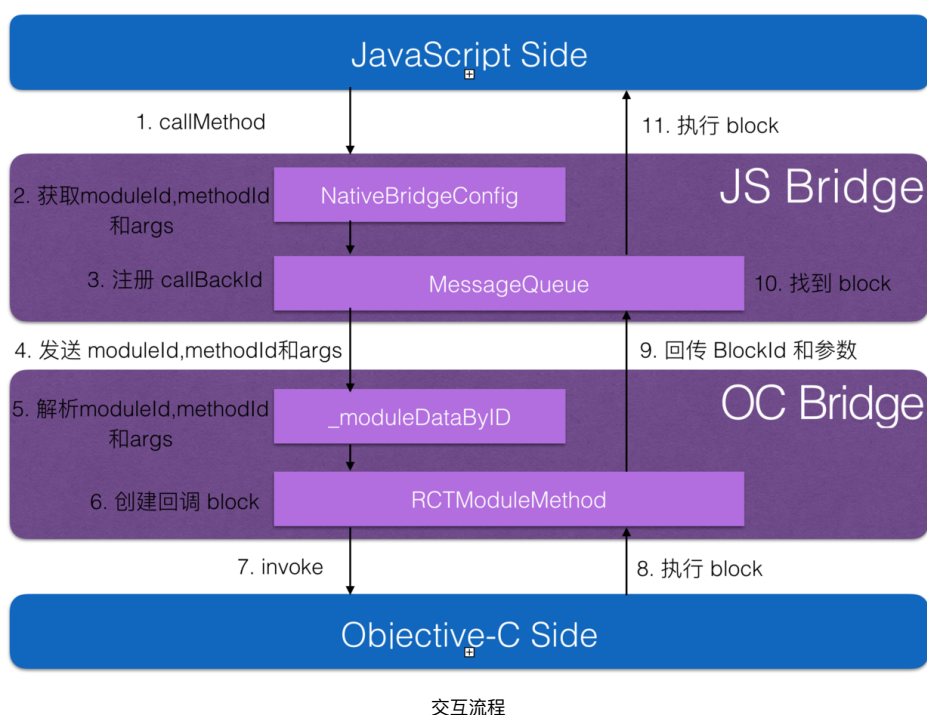
既然说到函数互调，那么就不得不提到回调了。对于 Objective-C 来说，执行完 JavaScript 代码再执行 Objective-C 回调毫无难度，难点依然在于 JavaScript 代码调用 Objective-C 之后，如何在 Objective-C 的代码中，回调执行 JavaScript 代码。

目前 React Native 的做法是：在 JavaScript 调用 Objective-C 代码时，注册要回调的 Block，并且把 `BlockId` 作为参数发送给 Objective-C，Objective-C 收到参数时会创建 Block，调用完 Objective-C 函数后就会执行这个刚刚创建的 Block。

Objective-C 会向 Block 中传入参数和 `BlockId`，然后在 Block 内部调用 JavaScript 的方法，随后 JavaScript 查找到当时注册的 Block 并执行。

## 图解

好吧，如果你是新手，并且坚持读到了这里，估计已经懵逼了。不要担心，与 JavaScript 的交互确实不是一下子能够完全理清楚的，你可以先参考这个示意图：



交互流程

注：

1. 本图由 bang 的文章中的图片修改而来
2. 本图只是一个简单的示意图，不建议当做时序图使用，请参考下一节源码分析。

3. Objective-C 和 JavaScript 的交互总是由前者发起，本图为了简化，省略了这一步骤。

## React Native 源码分析

要想深入理解 React Native 的工作原理，有两个部分有必要阅读一下，分别是初始化阶段和方法调用阶段。

为了提炼出代码的核心含义，我会在不改变代码意图的基础上对它做一些删改，以便阅读。

写这篇文章是，React Native 还处于 0.27 版本，由于在 1.0 之前的变动幅度相对较大，因此下面的源码分析很可能随着 React Native 的演变而过时。但不管何时，把下面的源码读一遍都有助于你加深对 React Native 原理的理解。

## 初始化 React Native

每个项目都有一个入口，然后进行初始化操作，React Native 也不例外。一个不含 Objective-C 代码的项目留给我们的唯一线索就是位于 AppDelegate 文件中的代码：

```
RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:jsCodeLocation
                                                    moduleName:@"PropertyFinder"
                                                    initialProperties:nil
                                                    launchOptions:launchOptions];
```

用户能看到的一切内容都来源于这个 `RootView`，所有的初始化工作也都在这个方法内完成。

在这个方法内部，在创建 `RootView` 之前，React Native 实际上先创建了一个 `Bridge` 对象。它是 Objective-C 与 JavaScript 交互的桥梁，后续的方法交互完全依赖于它，而整个初始化过程的最终目的其实也就是创建这个桥梁对象。

初始化方法的核心是 `setUp` 方法，而 `setUp` 方法的主要任务则是创建 `BatchedBridge`。

`BatchedBridge` 的作用是批量读取 JavaScript 对 Objective-C 的方法调用，同时它内部持有一个 `JavaScriptExecutor`，顾名思义，这个对象用来执行 JavaScript 代码。

创建 `BatchedBridge` 的关键是 `start` 方法，它可以分为五个步骤：

1. 读取 JavaScript 源码
2. 初始化模块信息
3. 初始化 JavaScript 代码的执行器，即 `RCTJSExecutor` 对象
4. 生成模块列表并写入 JavaScript 端
5. 执行 JavaScript 源码

我们逐个分析每一步完成的操作：

## 读取 JavaScript 源码

这一部分的具体代码实现没有太大的讨论意义。我们只要明白，JavaScript 的代码是在 Objective-C 提供的环境下运行的，所以第一步就是把 JavaScript 加载进内存中，对于一个空的项目来说，所有的 JavaScript 代码大约占用 1.5 Mb 的内存空间。

需要说明的是，在这一步中，JSX 代码已经被转化成原生的 JavaScript 代码。

## 初始化模块信息

这一步在方法 `initModulesWithDispatchGroup:` 中实现，主要任务是找到所有需要暴露给 JavaScript 的类。每一个需要暴露给 JavaScript 的类(也成为 `Module`，以下不作区分)都会标记一个宏：`RCT_EXPORT_MODULE`，这个宏的具体实现并不复杂：

```
#define RCT_EXPORT_MODULE(js_name) \
RCT_EXTERN void RCTRegisterModule(Class); \
+ (NSString *)moduleName { return @"js_name; } \
+ (void)load { RCTRegisterModule(self); }
```

这样，这个类在 `load` 方法中就会调用 `RCTRegisterModule` 方法注册自己：

```
void RCTRegisterModule(Class moduleClass)
{
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        RCTModuleClasses = [NSMutableArray new];
    });

    [RCTModuleClasses addObject:moduleClass];
}
```

因此，React Native 可以通过 `RCTModuleClasses` 拿到所有暴露给 JavaScript 的类。下一步操作是遍历这个数组，然后生成 `RCTModuleData` 对象：

```
for (Class moduleClass in RCTGetModuleClasses()) {
    RCTModuleData *moduleData = [[RCTModuleData alloc] initWithModuleClass:moduleClass
                                                                    bridge:
                                                                    self];
    [moduleClassesByID addObject:moduleClass];
    [moduleDataByID addObject:moduleData];
}
```

可以想见，`RCTModuleData` 对象是模块配置表的主要组成部分。如果把模块配置表想象成一个数组，那么每一个元素就是一个 `RCTModuleData` 对象。

这个对象保存了 `Module` 的名字，常量等基本信息，最重要的属性是一个数组，保存了所有需要暴露给 JavaScript 的方法。

暴露给 JavaScript 的方法需要用 `RCT_EXPORT_METHOD` 这个宏来标记，它的实现原理比较复杂，有兴趣的读者可以自行阅读。简单来说，它为函数名加上了 `__rct_export__` 前缀，再通过 `runtime` 获取类的函数列表，找出其中带有指定前缀的方法并放入数组中：

```
- (NSArray<id<RCTBridgeMethod>> *)methods{
    unsigned int methodCount;
    Method *methods = class_copyMethodList(object_getClass(_moduleClass), &methodCount); // 获取方法列表
    for (unsigned int i = 0; i < methodCount; i++) {
        RCTModuleMethod *moduleMethod = /* 创建 method */
        [_methods addObject:moduleMethod];
    }
}
return _methods;
}
```

因此 Objective-C 管理模块配置表的逻辑是：Bridge 持有一个数组，数组中保存了所有的模块的 `RCTModuleData` 对象。只要给定 `ModuleId` 和 `MethodId` 就可以唯一确定要调用的方法。

## 初始化 JavaScript 代码的执行器，即 RCTJSCExecutor 对象

通过查看源码可以看到，初始化 JavaScript 执行器的时候，`addSynchronousHookWithName` 这个方法被调用了多次，它其实向 JavaScript 上下文中添加了一些 Block 作为全局变量：

```
- (void)addSynchronousHookWithName:(NSString *)name usingBlock:(id)block {
    self.context.context[name] = block;
}
```

有些同学读源码时可能会走进一个误区，如果在 Block 中打一个断点就会发现，Block 其实是被执行了，但却找不到任何能够执行 Block 的代码。

这其实是因为这个 Block 并非由 Objective-C 主动调用，而是在第五步执行 JavaScript 代码时，由 JavaScript 在上下文中获取到 Block 对象并调用，有兴趣的读者可以自行添加断点并验证。

这里我们需要重点注意的是名为 `nativeRequireModuleConfig` 的 Block，它在 JavaScript 注册新的模块时调用：

```
get: () => {
    let module = RemoteModules[moduleName];
    const json = global.nativeRequireModuleConfig(moduleName); // 调用 OC 的 Block
    const config = JSON.parse(json); // 解析 json
    module = BatchedBridge.processModuleConfig(config, module.moduleID); // 注册 config
    return module;
},
```

这就是模块配置表能够加载到 JavaScript 中的原理。

另一个值得关注的 Block 叫做 `nativeFlushQueueImmediate`。实际上，JavaScript 除了把调用信息放到 `MessageQueue` 中等待 Objective-C 来取以外，也可以主动调用 Objective-C 的方法：

```
if (global.nativeFlushQueueImmediate &&
    now - this._lastFlush >= MIN_TIME_BETWEEN_FLUSHES_MS) {
    global.nativeFlushQueueImmediate(this._queue); // 调用 OC 的代码
}
```

目前，React Native 的逻辑是，如果消息队列中有等待 Objective-C 处理的逻辑，而且 Objective-C 超过 5ms 都没有来取走，那么 JavaScript 就会主动调用 Objective-C 的方法：

```
[self addSynchronousHookWithName:@"nativeFlushQueueImmediate" usingBlock:^(NSArray<NSArray *> *calls){
    [self->_bridge handleBuffer:calls batchEnded:NO];
}];
```

这个 `handleBuffer` 方法是 JavaScript 调用 Objective-C 方法的关键，在下一节——**方法调用中**，我会详细分析它的实现原理。

一般情况下，Objective-C 会定时、主动的调用 `handleBuffer` 方法，这有点类似于轮询机制：



```
// 每个一段时间发生一次：
Objective-C：嘿，JavaScript，有没有要调用我的方法呀？
JavaScript：有的，你从 MessageQueue 里面取出来。
```

然而由于卡顿或某些特殊原因，Objective-C 并不能总是保证能够准时的清空 MessageQueue，这就是为什么 JavaScript 也会在一定时间后主动的调用 Objective-C 的方法。查看上面 JavaScript 的代码可以发现，这个等待时间是 5ms。

请牢牢记住这个 5ms，它告诉我们 JavaScript 与 Objective-C 的交互是存在一定开销的，不然就不会等待而是每次都立刻发起请求。其次，这个时间开销大约是毫秒级的，不会比 5ms 小太多，否则等待这么久就意义不大了。

## 生成模块配置表并写入 JavaScript 端

复习一下 `nativeRequireModuleConfig` 这个 Block，它可以接受 `ModuleName` 并且生成详细的模块信息，但在前文中我们没有提到 JavaScript 是如何知道 Objective-C 要暴露哪些类的(目前只是 Objective-C 自己知道)。

这一步的操作就是为了让 JavaScript 获取所有模块的名字：

```
- (NSString *)moduleConfig{
    NSMutableArray<NSArray *> *config = [NSMutableArray new];
    for (RCTModuleData *moduleData in _moduleDataByID) {
        [config addObject:@[moduleData.name]];
    }
}
```

查看源码可以发现，Objective-C 把 `config` 字符串设置成 JavaScript 的一个全局变量，名字叫做： `__fbBatchedBridgeConfig` 。

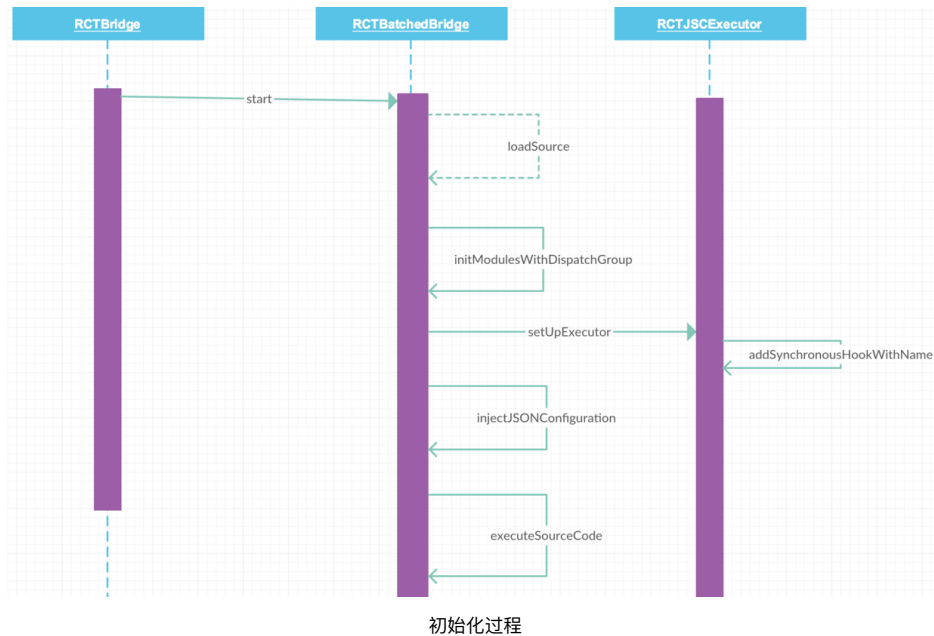
## 执行 JavaScript 源码

这一步也没什么技术难度可以，代码已经加载进了内存，该做的配置也已经完成，只要把 JavaScript 代码运行一遍即可。

运行代码时，第三步中所说的那些 Block 就会被执行，从而向 JavaScript 端写入配置信息。

至此，JavaScript 和 Objective-C 都具备了向对方交互的能力，准备工作也就全部完成了。

画了一个简陋的时序图以供参考：



## 方法调用

如前文所述，在 React Native 中，Objective-C 和 JavaScript 的交互都是通过传递 ModuleId、MethodId 和 Arguments 进行的。以下是分情况讨论：

## 调用 JavaScript 代码

也许你在其他文章中曾经多次听说 JavaScript 代码总是在一个单独的线程上面调用，它的实际含义是 Objective-C 会在单独的线程上运行 JavaScript 代码：

```

- (void)executeBlockOnJavaScriptQueue:(dispatch_block_t)block
{
    if ([NSThread currentThread] != _javaScriptThread) {
        [self performSelector:@selector(executeBlockOnJavaScriptQueue:)
                    onThread:_javaScriptThread withObject:block waitUntilDone:NO];
    } else {
        block();
    }
}

```

调用 JavaScript 代码的核心代码如下：

```

- (void)_executeJSCall:(NSString *)method
    arguments:(NSArray *)arguments
    callback:(RCTJavaScriptCallback)onComplete{
    [self executeBlockOnJavaScriptQueue:^(
        // 获取 contextJSRef、methodJSRef、moduleJSRef
        resultJSRef = JSObjectCallAsFunction(contextJSRef, (JSObjectRef)methodJSR
ef, (JSObjectRef)moduleJSRef, arguments.count, jsArgs, &errorJSRef);
        objcValue = /*resultJSRef 转换成 Objective-C 类型*/
        onComplete(objcValue, nil);
    )];
}

```

需要注意的是，这个函数名是我们要调用 JavaScript 的中转函数名，比如 callFunctionReturnFlushedQueue。也就是说它的作用其实是处理参数，而非真正要调用的 JavaScript 函数。

这个中转函数接收到的参数包含了 ModuleId、MethodId 和 Arguments，然后由中转函数查找自己的模块配置表，找到真正要调用的 JavaScript 函数。

在实际使用的时候，我们可以这样发起对 JavaScript 的调用：

```
[_bridge.eventDispatcher sendAppEventWithName:@"greeted"
                        body:@{ @"name": @"nmae"}];
```

这里的 Name 和 Body 参数分别表示要调用的 JavaScript 的函数名和参数。

## JavaScript 调用 Objective-C

在调用 Objective-C 代码时，如前文所述，JavaScript 会解析出方法的 ModuleId、MethodId 和 Arguments 并放入到 MessageQueue 中，等待 Objective-C 主动拿走，或者超时后主动发送给 Objective-C。

Objective-C 负责处理调用的方法是 `handleBuffer`，它的参数是一个含有四个元素的数组，每个元素也都是一个数组，分别存放了 ModuleId、MethodId、Params，第四个元素目前用处不大。

函数内部在每一次方调用中调用 `_handleRequestNumber:moduleID:methodID:params` 方法。，通过查找模块配置表找出要调用的方法，并通过 runtime 动态的调用：

```
[method invokeWithBridge:self module:moduleData.instance arguments:params];
```

在这个方法中，有一个很关键的方法：`processMethodSignature`，它会根据 JavaScript 的 CallbackId 创建一个 Block，并且在调用完函数后执行这个 Block。

## 实战应用

俗话说：“思而不学则神棍”，下面举一个例子来演示 Objective-C 是如何与 JavaScript 进行交互的。首先新建一个模块：

```
// .h 文件
#import <Foundation/Foundation.h>
#import "RCTBridgeModule.h"

@interface Person : NSObject<RCTBridgeModule, RCTBridgeMethod>

@end
```

Person 这个类是一个新的模块，它有两个方法暴露给 JavaScript：

```
#import "Person.h"
#import "RCTEventDispatcher.h"
#import "RCTConvert.h"

@implementation Person
@synthesize bridge = _bridge;

RCT_EXPORT_MODULE();

RCT_EXPORT_METHOD(greet:(NSString *)name)
{
    NSLog(@"Hi, %@!", name);
    [_bridge.eventDispatcher sendAppEventWithName:@"greeted"
                                             body:@{ @"name": @"nmae"}];
}

RCT_EXPORT_METHOD(greetss:(NSString *)name name2:(NSString *)name2 callback:(RCTResponseSenderBlock)callback)
{
    NSLog(@"Hi, %@! %@!!", name, name2);
    callback(@[@12,@23,@34]);
}

@end
```

在 JavaScript 中，可以这样调用：

```
Person.greet('Tadeu');
Person.greetss('Haha', 'Heihei', (events) => {
    for (var i = 0; i < events.length; i++) {
        console.log(events[i]);
    }
});
```

有兴趣的同学可以复制以上代码并自行调试。

## React Native 优缺点分析

经过一长篇的讨论，其实 React Native 的优缺点已经不难分析了，这里简单总结一下：

### 优点

1. 复用了 React 的思想，有利于前端开发者涉足移动端。
2. 能够利用 JavaScript 动态更新的特性，快速迭代。
3. 相比于原生平台，开发速度更快，相比于 Hybrid 框架，性能更好。

### 缺点

1. 做不到 Write once, Run everywhere，也就是说开发者依然需要为 iOS 和 Android 平台提供两套不同的代码，比如参考官方文档 (<https://facebook.github.io/react-native/docs/getting-started.html>)可以发现不少组件和API都区分了 Android 和 iOS 版本。即使是共用组件，也会有平台独享的函数。
2. 不能做到完全屏蔽 iOS 端或 Android 的细节，前端开发者必须对原生平台有所了解。加重了学习成本。对于移动端开发者来说，**完全不具备**用 React Native 开发的能力。
3. 由于 Objective-C 与 JavaScript 之间切换存在固定的时间开销，所以性能必定不及原生。比如目前的官方版本无法做到 UITableView(ListView) 的视图重用，因为滑动过程中，视图重用需要在异步线程中执行，速度太慢。这也就导致随着 Cell 数量的增加，占用的内存也线性增加。

综上，我对 React Native 的定位是：

利用脚本语言进行原生平台开发的一次成功尝试，降低了前端开发者入门移动端的门槛，一定业务场景下具有独特的优势，几乎不可能取代原生平台开发。

## 参考资料

1. React Native 官方文档 (<https://facebook.github.io/react-native/docs/getting-started.html>)、React Native 官方文档中文版 (<http://wiki.jikexueyuan.com/project/react-native/>)
2. React Native通信机制详解 (<http://blog.cnbang.net/tech/2698/>)
3. React 入门实例教程 (<http://www.ruanyifeng.com/blog/2015/03/react.html>)

OC的基础知识 (/nb/2876154)

举报文章 © 著作权归作者所有



bestswifter (/u/3e55748920d2)

写了 249286 字，被 7061 人关注，获得了 5816 个喜欢  
(/u/3e55748920d2)

+ 关注

个人博客：<http://www.bestswifter.com> 我的微博：<http://we...>

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

赞赏支持



♡ 喜欢 (/sign\_in) | 340



更多分享

(<http://cwb.assets.jianshu.io/notes/images/4325467/>)



登录 (/sign\_in) 发表评论

61条评论

只看作者

按喜欢排序 按时间正序 按时间倒序



请叫我留胡渣 (/u/5f902876a97a)


19楼 · 2016.09.12 12:09

(/u/5f902876a97a)


要用rn做出比较优秀的app的话必须要学习oc，但是学习oc为什么还要用rn，哈哈，只是说在一些业务场景下可以用起来，我们小组有俩个rn app在做，挺不错的，对于最后老板那句不可能取代oc，个人觉得，分情况来看，哈哈。

👍 2人赞 🗨 回复


bestswifter (/u/3e55748920d2)：@请叫我留胡渣 (/users/5f902876a97a) 学习oc和使用rn不冲突，就像你说的，分情况来

2016.09.12 12:42  回复

请叫我留胡渣 (/u/5f902876a97a): @bestswifter (/users/3e55748920d2) 看了下bang的博客, 然后又看到了gmtc全球移动技术大会, 一个叫郭晓铭的架构师惊呆了,

2016.09.12 15:13  回复

RemisKrlet (/u/3b5a95e93778): @请叫我留胡渣 (/users/5f902876a97a) 我们公司的

2017.02.19 14:05  回复

 添加新评论




AidenRao (/u/263107bb1199)

2楼 · 2016.06.11 16:35


(/u/263107bb1199)  
博主高产😁先回再看

 赞  回复


bestswifter (/u/3e55748920d2): @饶志臻 (/users/263107bb1199) 额。。。半个月一篇而已。


2016.06.11 16:37  回复

AidenRao (/u/263107bb1199): @bestswifter (/users/3e55748920d2) 可是长啊, 内容多。感觉写这么一篇得花很多时间。

2016.06.11 21:29  回复

bestswifter (/u/3e55748920d2): @饶志臻 (/users/263107bb1199) 嗯, 光写就花了很久很久。。。还画了两个图

2016.06.11 21:32  回复

 添加新评论




asce1885 (/u/4ef984470da8)

3楼 · 2016.06.12 10:08


(/u/4ef984470da8)  
介绍 React Native 的书籍寥寥无几

 赞  回复


chacha (/u/0bc2bb63a2b7): @asce1885 (/users/4ef984470da8) 官方文档最清楚了

2016.08.27 23:54  回复

4a5a47577505 (/u/4a5a47577505): 看到Android高级进阶的作者。😁

2016.12.11 15:00  回复

asce1885 (/u/4ef984470da8): @4a5a47577505 (/users/4a5a47577505) 🍷

2016.12.11 18:18  回复

 添加新评论



柴茵 (/u/fddb055efe32)

4楼 · 2016.06.12 18:58

(/u/fddb055efe32)  
讲的很赞啊~

 赞  回复



夜殇、夜逝 (/u/dcbdf07ae9de)

5楼 · 2016.06.12 21:49

(/u/dcbdf07ae9de)

nice

👍 赞    💬 回复



Ellian (/u/1e7f1160a725)

7楼 · 2016.06.13 11:20

(/u/1e7f1160a725)  
可以转载吗？

👍 赞    💬 回复



summer\_liu的自留地 (/u/9d9cf9760217)

8楼 · 2016.06.13 18:45

(/u/9d9cf9760217)

为什么不直接用moduleName, methodName呢, 要用id来中转？

👍 赞    💬 回复

bestswifter (/u/3e55748920d2): @summer\_liu (/users/9d9cf9760217) 感谢提出如此优秀的问题, 最近比较忙, 抱歉回复慢了。我的理解是: 如果要传字符串, 那么两端其实保存的是一个字典(JS<->OC映射关系), 如果模块名和方法名非常多的话, 这样可能存在性能问题。而现在的方案, 两端其实存了不同的两个数组, 不存在性能问题。另外, 对于CallbackId这样的东西, 用字符串似乎不方便表示, 所以为了统一, 就都用ID了。

以上是我的理解, 欢迎交流

2016.06.14 21:03    💬 回复

summer\_liu的自留地 (/u/9d9cf9760217): @bestswifter (/users/3e55748920d2)

找了下配置表的对应关系。

```
"ModuleName1": {
  "moduleId": 0,
  "methods": {
    "methodName1": {
      "methodID": 0,
      "type": "remote"
    },
    "methodName2": {
      "methodID": 1,
      "type": "remote"
    },
    etc...
  },
  "constants": {
    ...
  }
},
```

我猜可能是因为id传起来比较方便清晰, 不会显得那么冗余。  
感觉直接用modulename也是可以的。

另外, 性能问题是指哪里的？

2016.06.14 21:29    💬 回复

bestswifter (/u/3e55748920d2): @summer\_liu (/users/9d9cf9760217) 之前理解错了, 应该不涉及性能问题。“清晰”应该只是最显而易见的有点, 但如果没有别的好处, 我是不会弄出配置表这种东西来把传递的信息简化成ID的。我猜测, 有了配置表以后, JavaScript 这一端就可以知道模块和方法是否存在。既然配置表是有必要的, 那么利用配置表传递ID就比较合理了。此外, OC 的类名并不一定就是暴露给JS的模块名, 这也许是不传moduleName的原因之一吧。

2016.06.15 00:03    💬 回复

✎ 添加新评论    |    还有3条评论, 展开查看



布里丹之驴 (/u/8cfce925d90c)

9楼 · 2016.06.14 08:06

(/u/8cfce925d90c)

真的很详细，看了一个下午加上早上挤车的时间

👍 赞    💬 回复



七秒小鱼人 (/u/e9cc069c4f8f)

10楼 · 2016.06.14 20:08

(/u/e9cc069c4f8f)

楼主，你这个画的2张图是用什么软件画的？？我的角度是不是有点刁专

👍 赞    💬 回复

bestswifter (/u/3e55748920d2): @七秒小鱼人\_ (/users/e9cc069c4f8f) Keynote上面普通的图形拖拽出来的

2016.06.14 20:09    💬 回复

七秒小鱼人 (/u/e9cc069c4f8f): @bestswifter (/users/3e55748920d2) 好的，谢谢啦。

2016.06.14 20:15    💬 回复

✎ 添加新评论



晴天 (/u/7cef2e74c910)

11楼 · 2016.06.23 17:58

(/u/7cef2e74c910)

听说这个火的不行，

👍 赞    💬 回复



Codepgq (/u/3f246c8f1c8f)

12楼 · 2016.07.08 17:41

(/u/3f246c8f1c8f)

花了点时间从头到尾看了一遍，还没有形成概念。看来还是需要多学习学习！

👍 赞    💬 回复



Codepgq (/u/3f246c8f1c8f)

13楼 · 2016.07.08 17:42

(/u/3f246c8f1c8f)

需要去练习练习！！

👍 赞    💬 回复



walter211 (/u/6b0bf4698022)

14楼 · 2016.07.12 11:17

(/u/6b0bf4698022)

```
self addSynchronousHookWithName:@"nativePerformanceNow" usingBlock:^(
return @(CACurrentMediaTime() * 1000);
});
```

5ms 哪里算出来的？


👍 赞    💬 回复


bestswifter (/u/3e55748920d2): @walter211 (/users/6b0bf4698022) 这个是官方设置的常量，估计是经过多次测算得出的一个阈值





2016.07.12 21:33    💬 回复

walter211 (/u/6b0bf4698022): @bestswifter (/users/3e55748920d2) 代码里没看到，在哪个文件？




2016.07.13 15:34  回复

 添加新评论

 极乐君 (/u/55af8d0de729)  
15楼 · 2016.08.22 16:25  
(/u/55af8d0de729)  
   很赞的文章!

 赞  回复

 GavinWyf (/u/7128bf2b8fc8)  
16楼 · 2016.08.26 16:40  
(/u/7128bf2b8fc8)  
赞一个，刚接触RN，还不了解原理是怎么玩的。

 赞  回复

被以下专题收入，发现更多相似内容

移动前沿

首页投稿

iOS

iOS

iOS记录篇

iOS Dev...

React N...

不明觉厉 iOS

iOS程序猿

MobDev G...

寒哥管理的技术专

伪程序员

React N...

iOS 开发随笔

React N...

加载更多...