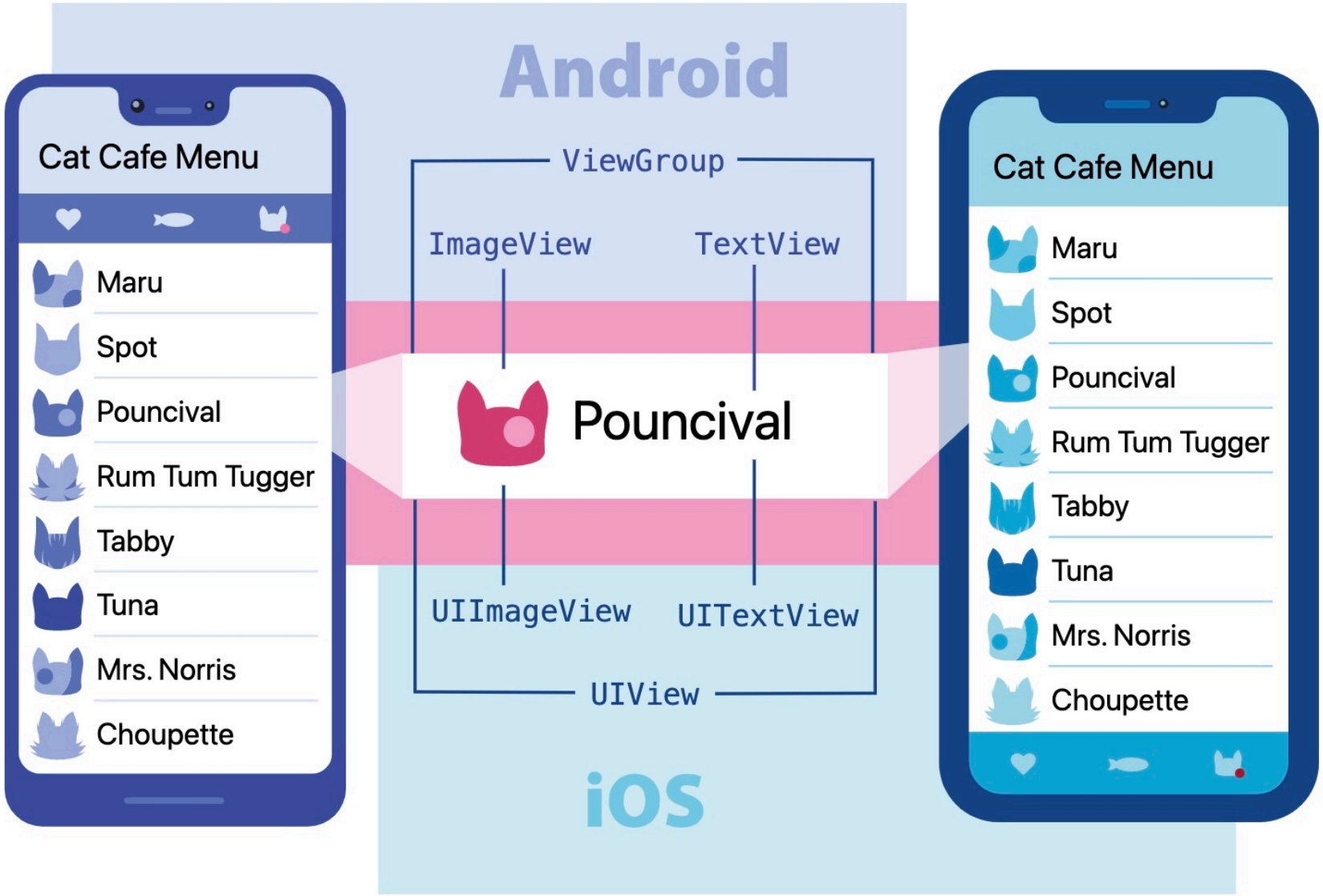
RN 架构介绍

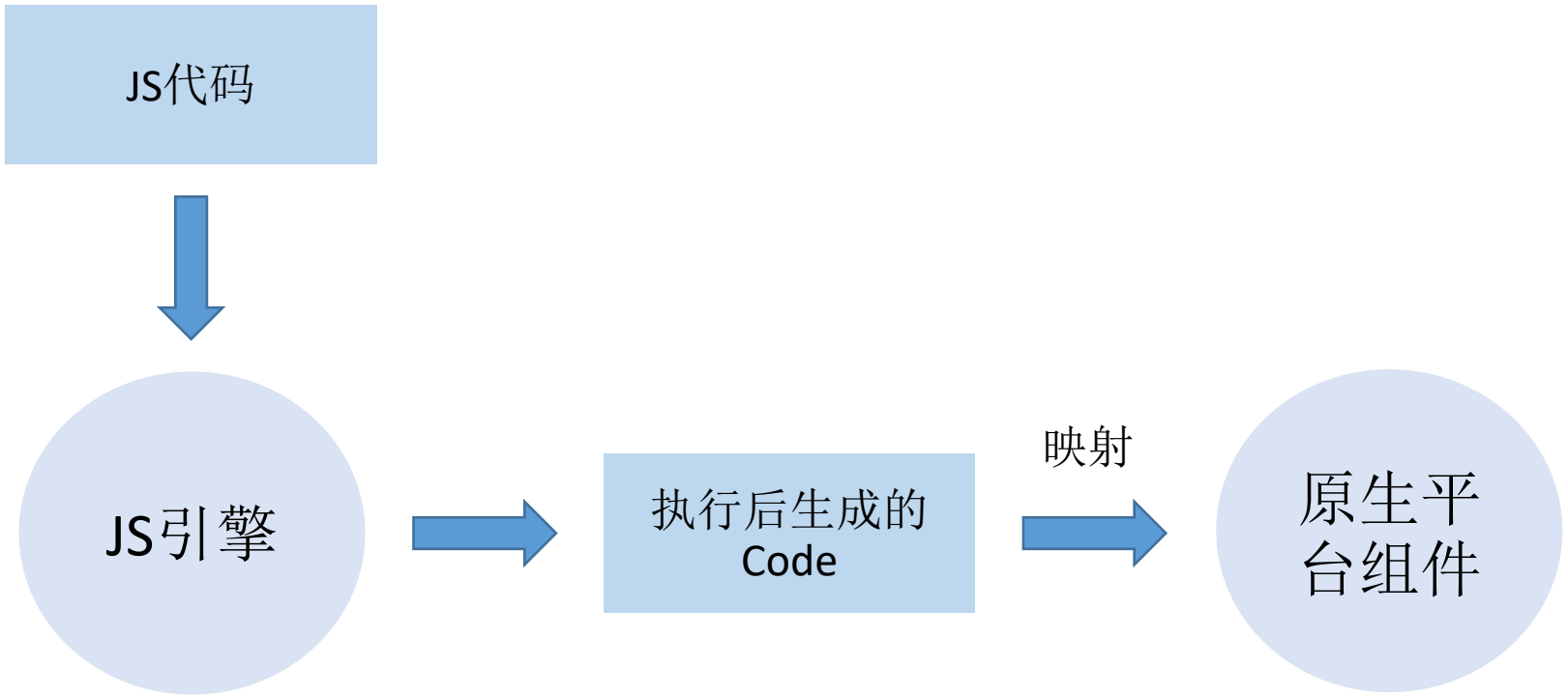
RN 使用 React 语法来书写组件，最终会被映射为原生平台的组件

REACT NATIVE UI COMPONENT	ANDROID VIEW	IOS VIEW	WEB ANALOG	DESCRIPTION
`<View>`	`<ViewGroup>`	`<UIView>`	A non-scrollling `<div>`	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
`<Text>`	`<TextView>`	`<UITextView>`	`<p>`	Displays, styles, and nests strings of text and even handles touch events
`<Image>`	`<ImageView>`	`<UIImageView>`	`<img>`	Displays different types of images
`<ScrollView>`	`<ScrollView>`	`<UIScrollView>`	`<div>`	A generic scrolling container that can contain multiple components and views
`<TextInput>`	`<EditText>`	`<UITextField>`	`<input type="text">`	Allows the user to enter text

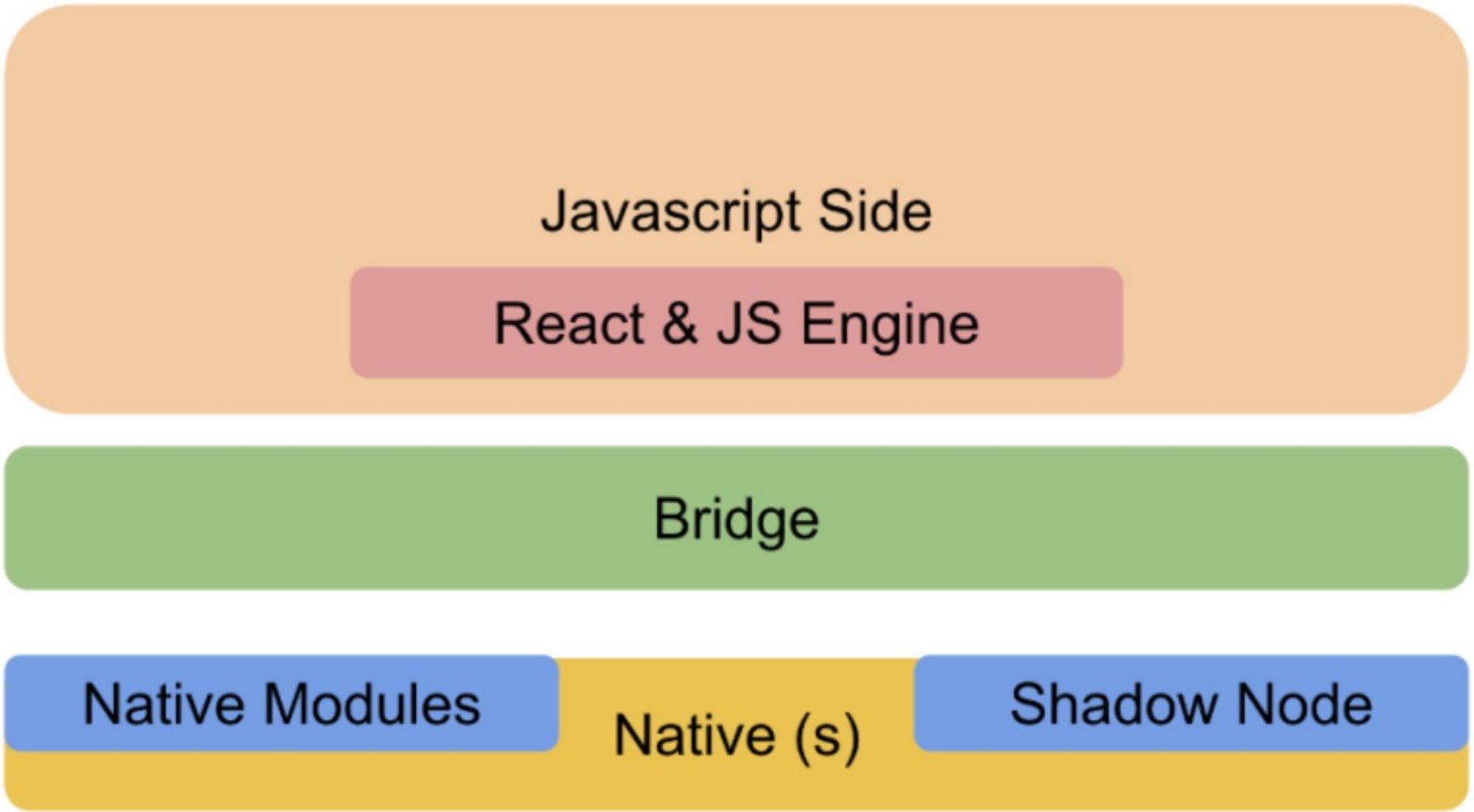
RN 使用 React 语法来书写组件，最终会被映射为原生平台的组件



RN 使用 React 语法来书写组件，最终会被映射为原生平台的组件



RN 架构示意图


Javascript Side
React & JS Engine
Bridge
Native Modules
Native (s)
Shadow Node

RN 架构示意图


Communication RN
JS thread
JSON
Bridge
JSON
Native threads

我们先需要解释一些基础概念：

UIManager：在Native侧，是在iOS/Android里主要运行的线程。只有它有权限可以修改客户端UI。

JS Thread：运行打包好的main.bundle.js文件，这个文件包含了RN的所有业务逻辑、行为和组件。

Shadow Node/Tree：在Native层的一个组件树，可以帮助监听App内的UI变化，有点像ReactJS里的虚拟Dom和Dom之间的关系。

Yoga：用来计算layout。是Facebook写的一个C引擎，用来把基于Flexbox的布局转换到Native的布局系统。

此架构下的整体流程：

1. 用户点击App的图标

2. UIManager线程：加载所有的Native库和Native组件比如 Text、Button、Image等

3. 告诉JS线程，Native部分准备好了，JS侧开始加载main.bundle.js，这里面包含了所有的js和react逻辑以及组件。

4. JS侧通过Bridge发送一条JSON消息到Native侧，告诉Native怎么创建UI。值得一提的是：所有经过Bridge的通信都是异步的，并且是打包发送的。这是为了避免阻塞UI

5. Shadow线程最先拿到消息，然后创建UI树

6. 然后，它使用Yoga布局引擎去获取所有基于flex样式的布局，并且转化成Native的布局，宽、高、间距等。。

7. 之后UIManager执行一些操作并且像这样在屏幕上展示UI

优点：

UI不会被阻塞：用户感觉到更加流畅

不需要写Native侧的代码：使用RN库的话，很多代码可以只写JavaScript的

性能更加接近Native

整个流程是完整的。开发者不用去控制并且完全了解它

缺点：

有两个不同的领域：JS和Native，他们彼此之间并不能真正互相感知，并且也不能共享相同的内存。

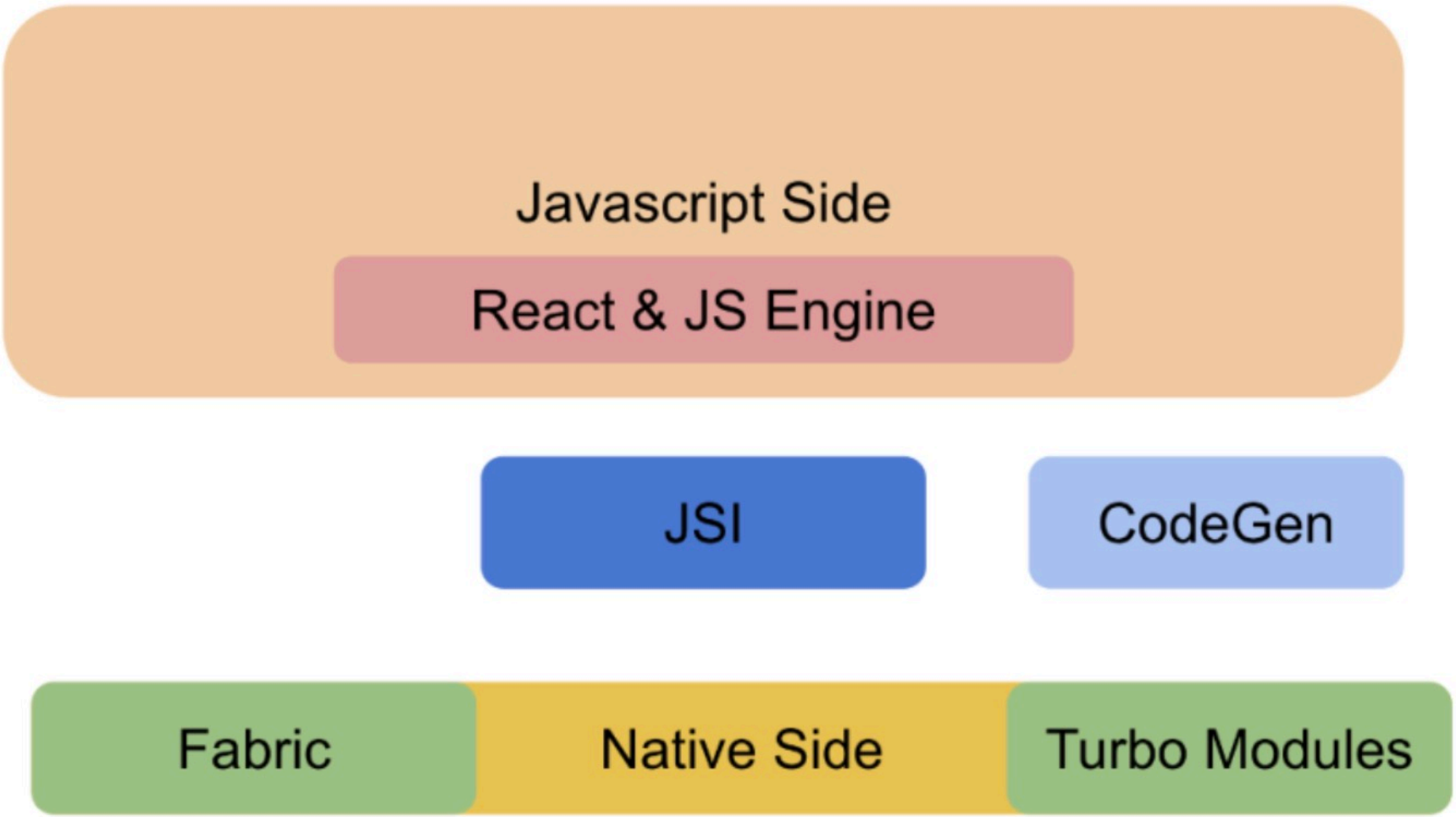
它们之间的通信是基于Bridge的异步通信。但是这也意味着，并不能保证数据100%并及时地到达另一侧。

传输大数据非常慢，因为内存不能共享，所有在js和native之间传输的数据都是一次新的复制。

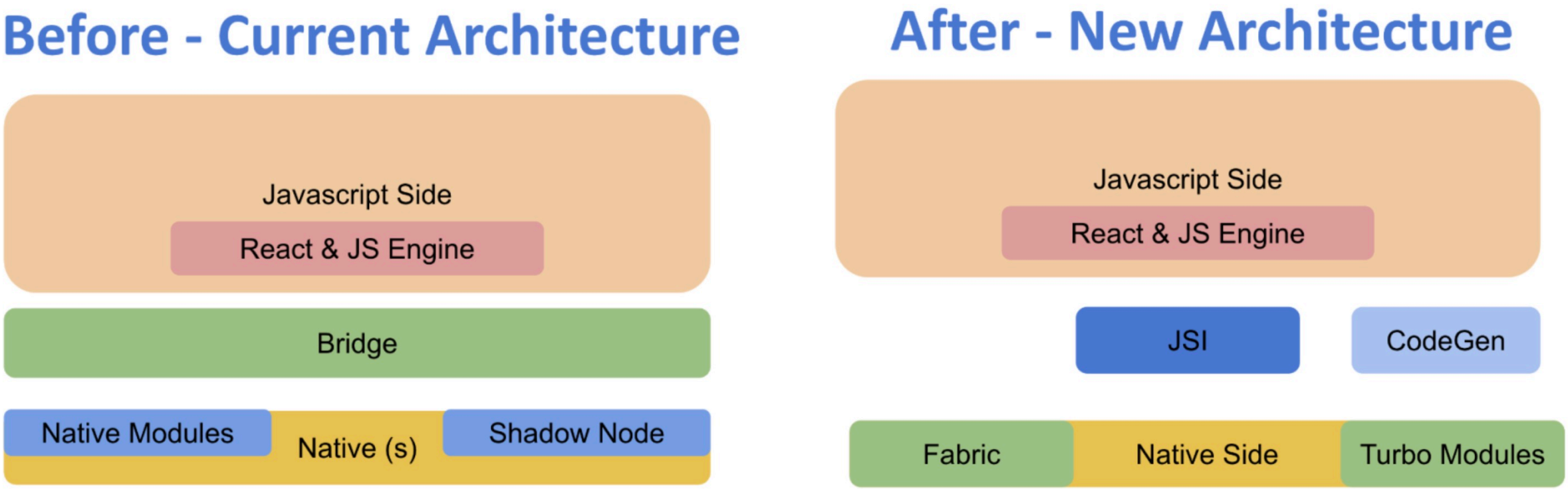
无法同步更新UI。比方说有个FlatList，当我滑动的时候会加载大量的数据。在某些边界场景，当有用户界面交互，但数据还没有返回，屏幕可能会发生闪烁。

RN代码仓库太大了。导致库更重，开源社区贡献代码或发布修复也更慢。

RN 架构示意图（新）

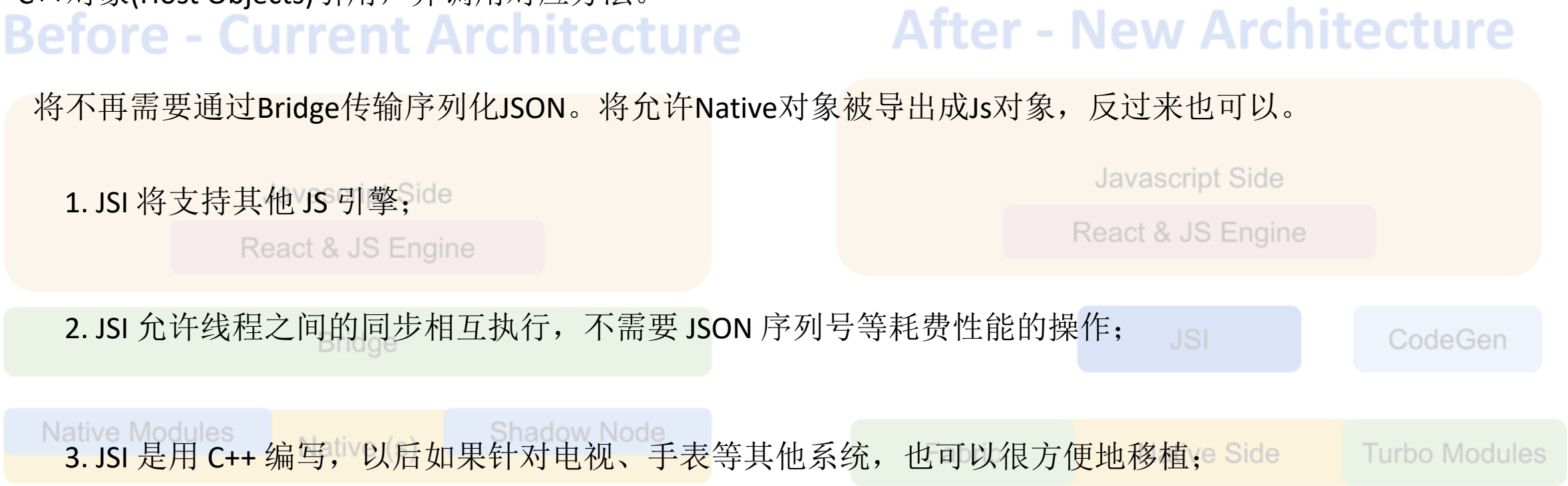

Javascript Side
React & JS Engine
JSI
CodeGen
Fabric
Native Side
Turbo Modules

RN 架构示意图（新旧对比）



一些新的概念

JSI是Javascript Interface的缩写，一个用C++写成的轻量级框架，它作用就是通过JSI，JS对象可以直接获得C++对象(Host Objects)引用，并调用对应方法。



将不再需要通过Bridge传输序列化JSON。将允许Native对象被导出成Js对象，反过来也可以。

1. JSI 将支持其他 JS 引擎；
2. JSI 允许线程之间的同步相互执行，不需要 JSON 序列号等耗费性能的操作；
3. JSI 是用 C++ 编写，以后如果针对电视、手表等其他系统，也可以很方便地移植；

一些新的概念

Fabric 是新的渲染系统，它将取代当前的 UI Manager。

UI Manager：

当 App 运行时，React 会执行你的代码并在 JS 中创建一个 ReactElementTree ，基于这棵树渲染器会在 C++ 中创建一个 ReactShadowTree。UI Manager 会使用 Shadow Tree 来计算 UI 元素的位置

一旦 Layout 完成，Shadow Tree 就会被转换为由 Native Elements 组成的 HostViewTree（例如：RN 里的 <View/> 会变成 Android 中的 ViewGroup 和 iOS 中的 UIView）。

而之前线程之间的通信都发生在 Bridge 上，这就意味着需要在传输和数据复制上耗费时间。通过JSON格式来传递消息，每次都要经历序列化和反序列化。

Fabric：

得益于前面的 JSI， JS 可以直接调用 Native 方法，其实就包括了 UI 方法，所以 JS 和 UI 线程可以同步执行从而提高列表、跳转、手势处理等的性能。

一些新的概念

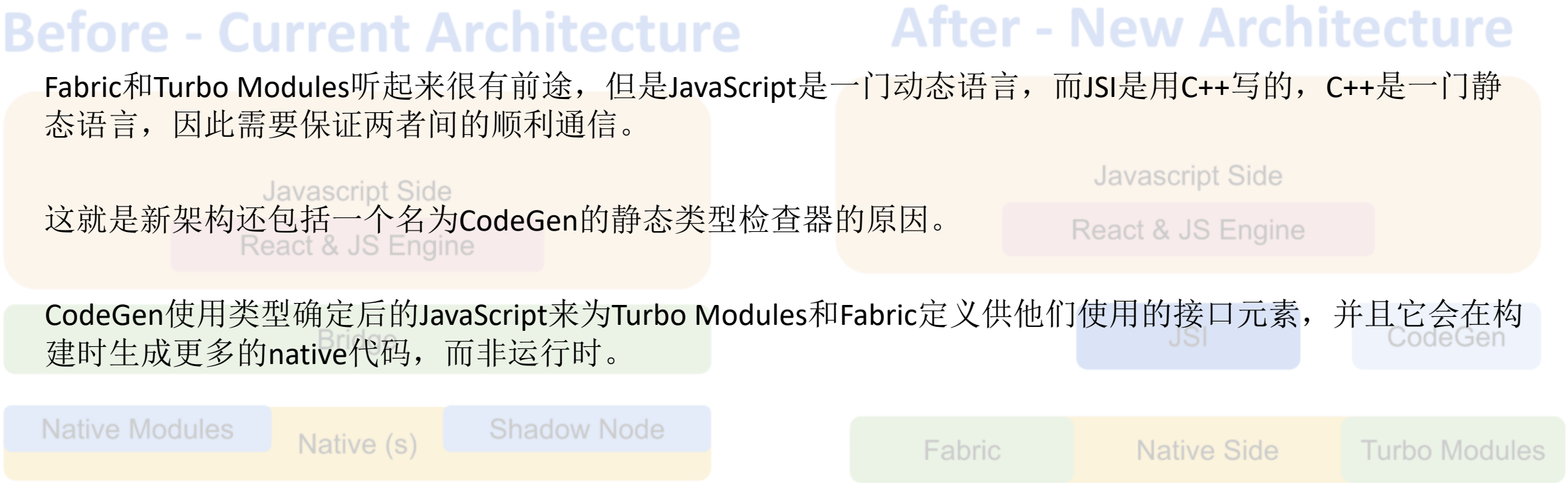
Turbo Modules

在之前的架构中 JS 使用的所有 Native Modules（例如蓝牙、地理位置、文件存储等）都必须在应用程序打开之前进行初始化，这意味着即使用户不需要某些模块，但是它仍然必须在启动时进行初始化。

Turbo Modules 基本上是对这些旧的 Native 模块的增强，正如在前面介绍的那样，现在 JS 将能够持有这些模块的引用，所以 JS 代码可以仅在需要时才加载对应模块，这样可以将显着缩短 RN 应用的启动时间。

一些新的概念

Codegen



Fabric和Turbo Modules听起来很有前途，但是JavaScript是一门动态语言，而JSI是用C++写的，C++是一门静态语言，因此需要保证两者间的顺利通信。

这就是新架构还包括一个名为CodeGen的静态类型检查器的原因。

CodeGen使用类型确定后的JavaScript来为Turbo Modules和Fabric定义供他们使用的接口元素，并且它会在构建时生成更多的native代码，而非运行时。

新架构下的整体流程

1. 用户点击App的图标

2. Fabric加载Native侧(没有Native组件)

3. 然后通知JS线程Native侧准备好了，JS侧会加载所有的main.bundle.js，里面包含了所有的js和react逻辑+组件

4. JS通过一个Native函数的引用(JSI API导出的)调用到Fabric，同时Shadow Node创建一个和以前一样的UI树。

5. Yoga执行布局计算，把基于Flexbox的布局转化成终端的布局。

6. Fabric执行操作并且显示UI