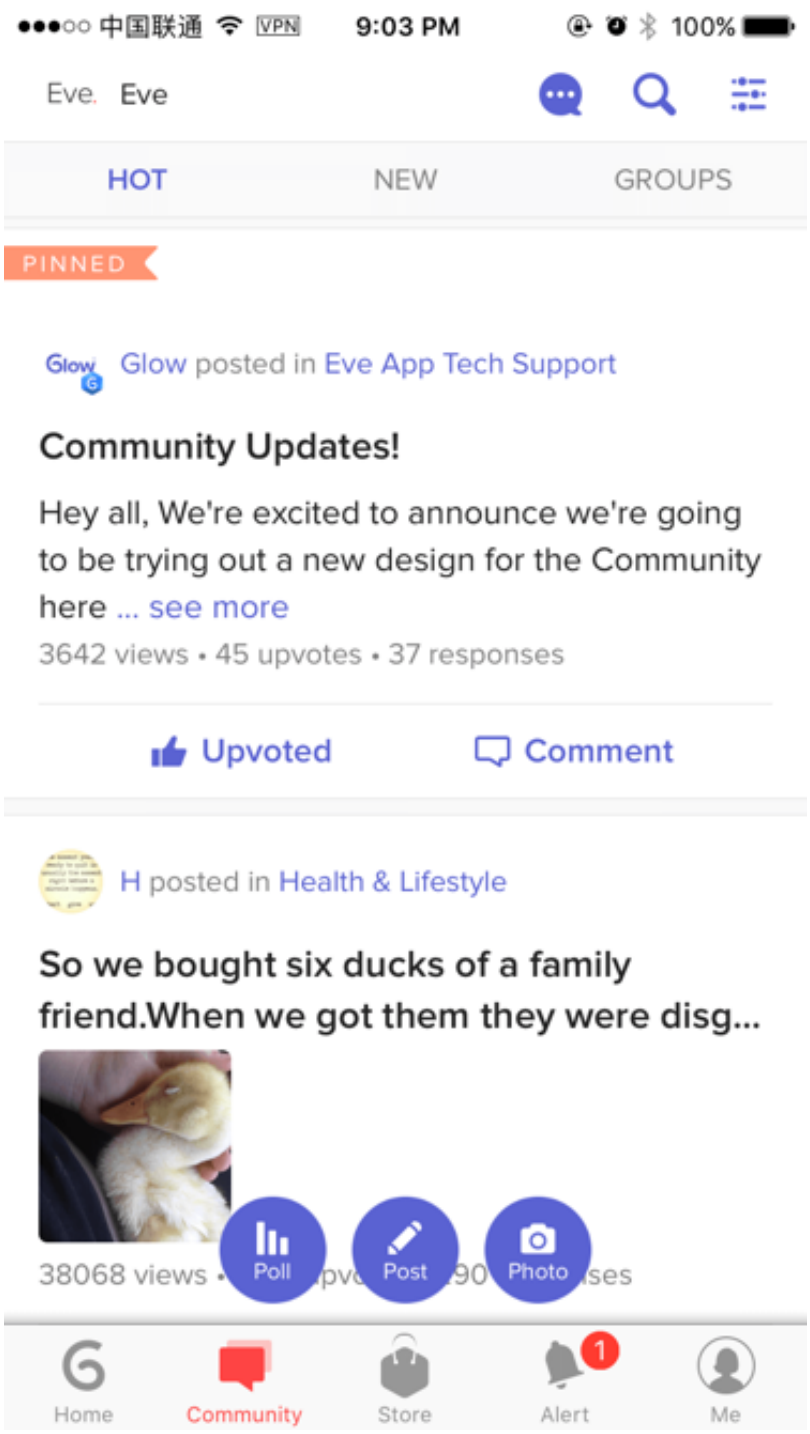
MENU

React Native 在 Glow 的实践

30 APRIL 2017

1. 为什么使用 React Native

在最近发布的 Eve v2.8 里，我们用 React Native 重构了几乎整个 Community。本文记录了 React Native 在 Glow 的实践经验，并主要从 iOS 角度展开一些细节实现。但本文不会涉及太多 React Native 的入门知识，如果你还没有接触过 React Native，推荐先阅读官方文档和 React Native Express。



使用 React Native 主要原因：

- 在 JSPatch 和 Rollout.io 被 Apple 封杀以后，需要一种方法热修复 Live Bug
- 可以通过热更新的方式发布新的功能或者修改 UI，加快迭代节奏

- 可以在 iOS 和 Android 之间共用更多客户端代码（90% 以上）
- React Native 的 JS Bundle 作为一种容器很适合实现功能的组件化，从而实现跨 App 的重用

选型阶段我们也比较了 PhoneGap (Cordova) 和 Weex。

PhoneGap 是基于 WebView 实现的，整体性能和用户体验没有 Native 实现那么好（不过使用 WKWebView 可以大幅提高性能）。而且因为在浏览器环境运行，所以对资源的读取会受到一些限制。此外 Web 的渲染结果在不同的平台、版本上会有一些差异，调试和 QA 成本较高。另外跟 Native 的通信方式比较老式。好处是基于 Web 开发，可以用到很多已有的 Mobile Web 开发资源和工具；使用 CSS 和 JS 就可以实现很多动画；布局方式不那么局限，几乎可以实现所有布局。所以 PhoneGap 是作为比较安全的最后的选择。

Weex 是阿里开源的类似 React Native 的框架，很多 Native 的实现部分应该也有借鉴 React Native。主要区别是 Weex 基于 Vue 做开发，语法更接近 Web 开发（直接写 HTML/CSS/JS），学习成本较 Reac Native 低。另外 Glow 的网站部分是基于 Vue 开发的，所以团队成员对 Vue 有一定了解。所以我们先用 Weex 开发了一个简单的 Demo，但开发调试过程中发现 Weex 还是有比较多的 Bug，而且配套工具也跟 React Native 的还有些差距就放弃了。目前来看 Weex 适合做一些单页如活动页面的开发，还不是很适合做完整 App 的开发。希望在加入 Apache Incubator 之后能有更好的发展。

React Native 是 Facebook 开源的框架，从我们实际使用情况来看，经历了两年的迭代，目前版本还是比较稳定的，文档和配套的开发工具也比较完善。此外，React Native 的学习成本其实并不高，上手还是比较快的。

所以我们最终的选择是 React Native，如果你也在这些方案中做选择的话，推荐先看看 React Native 能否满足你的需求。

2. 理解 JSX, Component 和 React 里的信息流

基于 React 做开发，理解 JSX 和 Component 是很重要的。Component 是 React 中最基础的元素，App 是 Component，View Controller 是 Component，View 也是 Component。JSX 可以理解为生成 Component 实例的快捷方式，参见 JSX In Depth，下面的代码：

```
<Text style={{color: 'blue'}} numberOfLines={2}>
  Hello World!
</Text>
```

会被编译成：

```
React.createElement(
  Text, // type
  {style: {color: 'blue'}, numberOfLines: 2}, //
props
  'Hello World!' // ...children
)
```

所以元素的嵌套就是生成子元素后作为 `children` 参数传给夫元素做初始化。这也是为什么 JSX 里的 style 不是 CSS string，而是一个

Object。

Component 里最重要的属性是 `props` 和 `state`。在 React 里，`props` 是只读的，`state` 是可变的，`props` 和 `state` 的改变都会触发 rerender。

可以看到 `createElement` 的参数里只有 `props`，并没有初始状态的 `state`，所以可以理解 React 里合理的信息流是自上而下的。意即：子元素里的操作如果不影响数据源（`props` 里的数据），可以通过 `setState` 触发 `rerender`；反之（影响数据源），不能直接修改 `props` 的数据或通过本地变量或 `state` 来覆盖 `props` 里的数据，应该通过合理的方式通知上层改变数据源，从而更新子元素的 `props`，再触发 rerender。理解 React 中的信息流可以很大程度地避免一些 bug，尤其是 Component 重用或数据不一致相关的。Redux 通过中心化的 state store，可以很大程度上解决这些问题（不过引入 Redux 可能会提高学习成本，但很适合大中型项目）。这里有关于 Redux 在 React 里的使用的简单教程，Facebook 官方的 F8 Demo App 也用 Redux 来管理 `state`，可以参考。

理解 JSX，Component 以及 React 里的信息流，剩下的在 React 这边的开发工作就几乎是纯 JS 了。

3. Native Navigation

React Native 的社区里有很多关于 Native Navigation 的讨论和实现（NavigatorIOS, react-native-navigation, native-navigation），AirBnb 的方案开源的比较晚，不然可能是一个不错的选择。我们做 Native Navigation 的主要原因是：我们的 App 不是纯 React Native 的，Community 是作为一个 Tab 被加到 Native

的 UITabBarController 中。React Native 没有提供很好的方法对应到 iOS 里的 `hidesBottomBarWhenPushed`，也没有很好的方法实现 View controller-based status bar appearance。此外，不管是单 View Controller 的实现，还是通过 NavigatorIOS 实现的 NavigationController，在 Native 都是被包在一个根 UIViewController 里的，这一点对于管理 View Controller Stack 带来很多问题。Native Navigation 在开发后期为跨 Bundle 的跳转和热更新也带来了一些好处。

为了实现 Native Navigation，我们提供了两个很简单的类：`GLRNViewController` 和 `GLRNNavigationController`，分别对应和继承自 `UIViewController` 和 `UINavigationController`。

根据我们 App 的需求，我们实现 Native Navigation 时的目标和原则：

- 所有 `RCTRootView` 都对应到一个 `GLRNViewController`
- 所有的 `RCTRootView` 使用同一个 `RCTBridge`（注，热更新完成后可能存在多个不同版本 Bridge 的内存实例）
- 如果 `GLRNViewController` 是 Navigation Controller 里第二个或之后的 Controller 时，`hidesBottomBarWhenPushed` 返回 `YES`，来隐藏 Tab Bar，支持 View controller-based status bar appearance
- 所有的 `GLRNViewController` 都会被丢到 `GLRNNavigationController` 里，不论是否需要 push 其他的 View Controller
- `GLRNNavigationController` 的 Navigation Bar 会被隐藏，如有

需要，在 RN 侧绘制，通过 Native Module 实现 present/dismiss/push/pop

- 因为隐藏了 Navigation Bar，需要一些 hack 来支持滑动后退的手势

详见 `GLRNViewController` 和 `GLRNNavigationController`，Demo 里的省去了热更新和本地 bundle 管理的相关逻辑。

有了这些 Controller 以后，第二步是通过 Native Module 让 JS 侧调用 Native 方法 present/dismiss/push/pop controller。`NativeNavigator` module 提供了这些功能。虽然我们提供了 push/present/show 三种现实方式，其实为了保证上面提到的保证有 `GLRNNavigationController` 嵌套，因此 push 的实现也是调用 show，show 的逻辑是：

- 找到 Controller Stack 里最顶部的 Navigation Controller，所谓最顶部的定义见 `GLNavigator` 的 `+ (UIViewController *)topNavigationViewController`
- 如果他是一个 `GLRNNavigationController`，push 下一个 RN View Controller
- 如果不是，用 `GLRNNavigationController` 包起来以后 present

present 方法本身则比较简单，就是找到 topViewController 并用它 present 用 `GLRNNavigationController` 包起来的 RN View Controller。注：在 Android 上，push/present/show 的实现都是一样的 :)

在 JS 里，使用的方法非常简单：

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

var NativeNavigator = NativeModules.NativeNavigator;

NativeNavigator.openURL(url, options);
NativeNavigator.showURL(url, props, options);
NativeNavigator.pushURL(url, props, options);
NativeNavigator.presentURL(url, props, options);
NativeNavigator.popOrDismiss();
```

到这里，我们就完成了 Native Navigation 的实现。完整的可运行的代码见 Demo Project。

4. URL Routing

在 Native Navigation 章节里你可能已经发现，我们的页面跳转的方法都需要一个 URL 参数，GLRNViewController 和 RCTRoot 的初始化方法也需要 URL 参数。这是为了更好的支持 Deep Link 和兼容其他已经使用 URL 做跳转的场景，我们决定用 URL 来一一对应 RN 里的 Scene，并通过 URL 传递必要的参数。虽然在 Native 侧以及 Native Module 暴露的方法里，URL 是独立于 initialProps 传递的，但是在真正初始化 RCTRootView 的时候，URL 是作为 initialProps 里的一个字段被传到 RN 里的。

在 RN 侧，我们有一个 Root Component，他的 render 方法则是根据 this.props.url 来路由并最终渲染不同的 Scene Component 的。

不管是 iOS/Android 还是 Web 开发，用 URL 做 Routing 其实是比较常见的方法，所以不过多深入实现细节。一个标准的 URL 由如下部分组成：

```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

我看到有些项目里在使用 custom scheme 时会用 host 来传递命令，但我们为了更好的同时支持 custom scheme 和 deep link，在做 URL Routing 时并不关心 scheme 和 host 部分，从 path 部分往后 parse。（注意：`scheme://a/b/c` 的 path 是 `/b/c`，`scheme:///a/b/c` 的 path 则是 `/a/b/c`。）定义路由规则的时候我们用了类似 Express 的语法：比如对于规则 `/community/topic/:topic_id`，URL `https://glowing.com/community/topic/72057594040140879?page_source=apns` 会被路由到 Topic Detail Page，并带着形如 `{topic_id: '72057594040140879', page_source: 'apns'}` 的参数。

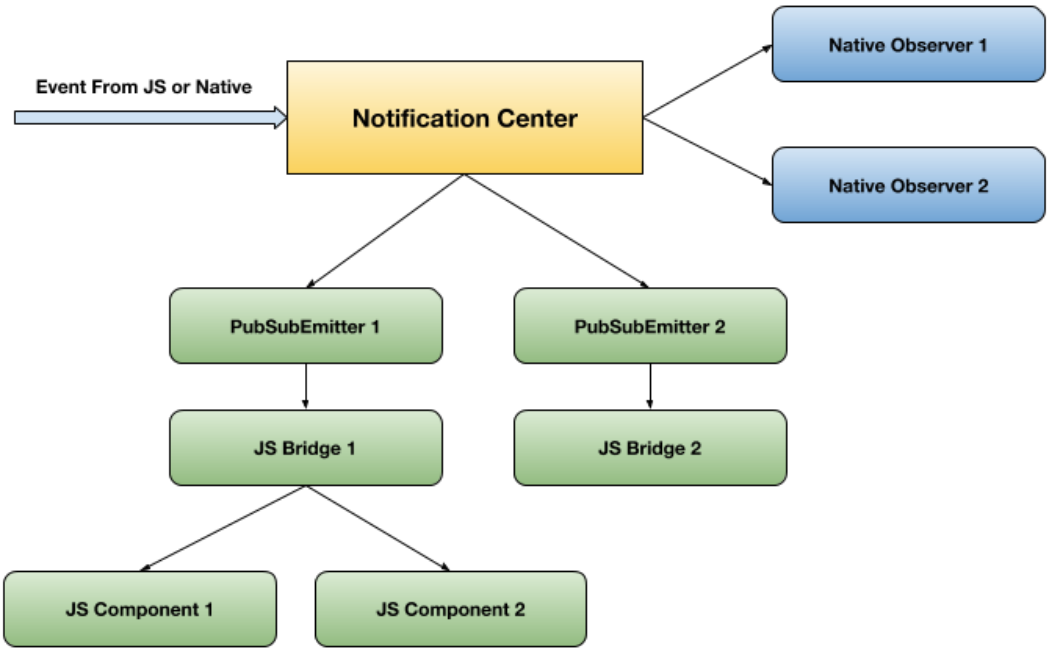
使用 URL 做路由为后文提到的组件化也带来了很多方便，详见组件化章节。

5. Notification Center

在 iOS App 的 Native 部分，我们有很多逻辑是通过 Notification Center 实现的，诸如一些 Topic 和 Comment 列表的刷新、用户信息的刷新等；另一方面，我们希望在 RN 侧实现一种 Pub Sub 的方法，来做页面间的广播；最好可以支持 RN 往 Native Notification

Center publish 一些事件，来支持触发一些旧有逻辑。所以我们实现了这样一个 PubSub Module。其中，因为 `RCTEventEmitter` 会检查 `supportedEvents` 并只发送支持的事件，所以为了支持监听任意的 Native Event，这个 Module 监听了所有 Notification，如果他被 JS 监听的话，包装成同一个 JS Event 后分发到 JS。

JS 发布事件的时候也会调用 Native Module 走 Notification Center，这样在不同的 Bridge 里发出来的 Event 也可以被其他 Bridge 收到，无形中也打通了各个 JS 环境。



JS 侧使用也很简单，常见的使用方法：

```
import PubSub from './PubSub';

componentDidMount() {
  PubSub.subscribe(this, 'event_name', (data:any) =>
{
    this.setState({
      // update state according to new data
    });
  });
}
componentWillUnmount() {
  PubSub.unsubscribe(this);
}
```

注意一定要在 componentWillUnmount 或者其他地方 unsubscribe，否则 component 会因为被 PubSub module 持有而一直存活在内存里。

6. 组件化

React Native 里有几个可以做组件化的点：

1. JS Bundle，JS Bundle 本身是很好的一个 container，无论是代码还是资源文件
2. React Native 里的 `AppRegistry.registerComponent` 可以注册多个 Root Component，通过名字区分

3. 在 Root Component 下根据条件 render 不同的 scene component

用 JS Bundle 做组件化的好处是版本控制的粒度比较细，可以分别更新某个组件；缺点是更新和版本控制逻辑以及组件间跳转相对复杂。

用多 Root Component 实现的缺点是 Native 需要知道各 Root Component 的名字，当接到一个 URL 的时候需要一个从 URL 到 Component 名字的映射方法，增加了 Native 部分的复杂度，而且路由逻辑会分散到 Native 和 JS 两部分。

我们最后决定在 JS 里实现组件化，在 Native，每个 App 有且只有一个 `main.jsbundle`，但打包时候不同的 App 对应不同的入口文件，在入口文件里再按需 import 和组装不同的组件，并且 Root Component 都被命名为 `main`。这样一来 Native 对包的管理以及 Root Component 的初始化就变得比较简单了，而且后续增加新的组件也完全可以在 JS 侧完成。

7. 热更新

为了增加灵活性，我们没有用 code push，自己实现了一套简单的包管理的 API。我们的 Community 的 JS Bundle 在压缩后只有不到 600KB，所以完全可以做全量更新。

- 首先，每次提交新的客户端版本时，会把最新的稳定版本以文件夹形式打包进 ipa（Create folder references in Xcode）
- Native 这边会从 app 目录，以及遍历 `NSDocumentDirectory` 下的 `react-native` 所有子目录，得到当前机器上所有版本，并使用最新的版本初始化 JS Bridge

- 每次 app 启动的时候，会向服务器询问当前版本客户端支持的最新的 JS Bundle，如果他比当前最新版本高，会下载 zip 包并解压到 `NSDocumentDirectory` 下的 `react-native` 下，更新 JS Bridge 并发出更新完成的通知，如有必要，会刷新页面

8. 一些坑

重用

RN 里同一个 Root Component 下的同类 Component 实例会被重用，重用的时候不会走 constructor/componentWillMount/componentDidMount 等方法，而是 componentWillReceiveProps/componentWillUpdate/componentDidUpdate。关于 Component 的 Lifecycle 参见 React 官方文档。

int64

JS 天生缺少对 64 位整形的支持，所以对于一些 id 类型，我们不得不都转成 string 来处理了，如果你 app 里的 id 或者其他整形超过 `Number.MAX_SAFE_INTEGER (Math.pow(2, 53) - 1, 9007199254740991)`，就要注意了

Crash

RN 对于 JS 里的 fatal error 默认是会 crash app 的，在 iOS 上可以使用 `RCTSetFatalHandler()` 设置 fatal handler 把 error 上报但不 crash app，比如 用 Fabric/Crashlytics：

```
RCTSetFatalHandler(^(NSError *error) {
```

```
    [CrashlyticsKit recordError:error];
});
```

9. 一些建议

- 全局启用 flow type check，并重视和解决 flow type error
- 可以用 sentry 来上报 JS error
- 推荐用 Visual Studio Code + Flow Language Support 插件做开发
- 推荐用 yarn 代替 npm，npm 对 version lock 做的比较差
- React Native 官方推荐通过 npm 来管理 sdk，很久没有更新 CocoaPods 了，我们为了方便在不同的 app 里引用，尤其是在公用的基础库里使用，把 React 和 Yoga 发布到了私有的 Pod Repo 上
- 一组 style 或者 component 组合用了 3 次以上就可以考虑抽成 common component 了，比如定义默认字体，一些按钮和图片的基础样式

10. Demo Project

写了一个 Demo Project：

https://github.com/allenhsu/react-native-at-glow

如果你还没安装 React Native 开发环境，先看这个官方文档配置环境。

克隆代码；

```
git clone git@github.com:allenhsu/react-native-at-glow.git
```

安装 node modules，也可以使用 `yarn install`：

```
cd react-native-at-glow/Demo
npm install
```

在 Simulator 里运行 Demo app，也可以打开 `Demo.xcodeproj` 手动运行：

```
react-native run-ios
```

这篇先写到这里，如果有什么问题欢迎留言讨论或在微博 at 我：@许小帅_allen

Allen 许帅

iOS engineer at Glow, Inc., a great fan of Steve Jobs, Apple and Pixar.

Shanghai, China *http://www.weibo.com/a11en*

Share this post

Glow Android 优化实践

了解 Glow 的朋友应该知道，我们主营四款 App，分别是Eve、Glow、Nuture和Baby。作为创业公司，我们的四款 App 都处于高速开发中，平均每个 Android App 由两人负责开发，包括 Android 和...