白话去哪儿RN大规模应用实践方案

React Native 框架是由 Facebook 开源的一套移动跨平台框架，与 Hybrid 方案相比具有原生应用的用户体验，经过两年多的蓬勃发展，目前的性能与稳定性相较于刚开始发布有了非常大的进步。去哪儿网（Qunar.com）从 2016 年 3 月上线之后开始在公司内部大规模使用，截止目前已经有三十多个业务接入 RN 业务。我们（YMFE）作为比较早期开始实践 RN 项目的团队，无论是抹平差异化、热更新系统方案还是核心组件的性能优化上等等，都积累了一些宝贵的经验，下面这就与你一一分享。

概述

由于 React Native（以下简称 RN）的设计理念并非是在 Android 和 iOS 两大移动端提供一致的 UI 风格，因此 RN 的组件样式在两大移动平台存在不小的差异；而在热更新上也仅仅只是提供了方便接入的接口，如何部署一个安全稳定的热更新平台也是一个需要考虑的问题；在大规模的业务场景中，RN 现有的 Bundle 机制也存在着严重的性能和体积问题。在接下来的篇幅中，笔者从下面几个方面详细介绍我们团队的一些主要工作成果。

- 平台差异化抹平。
- 热更新系统方案介绍。
- 核心组件 ListView 优化。
- 介绍 Android 实战中的一些坑。

平台差异化抹平

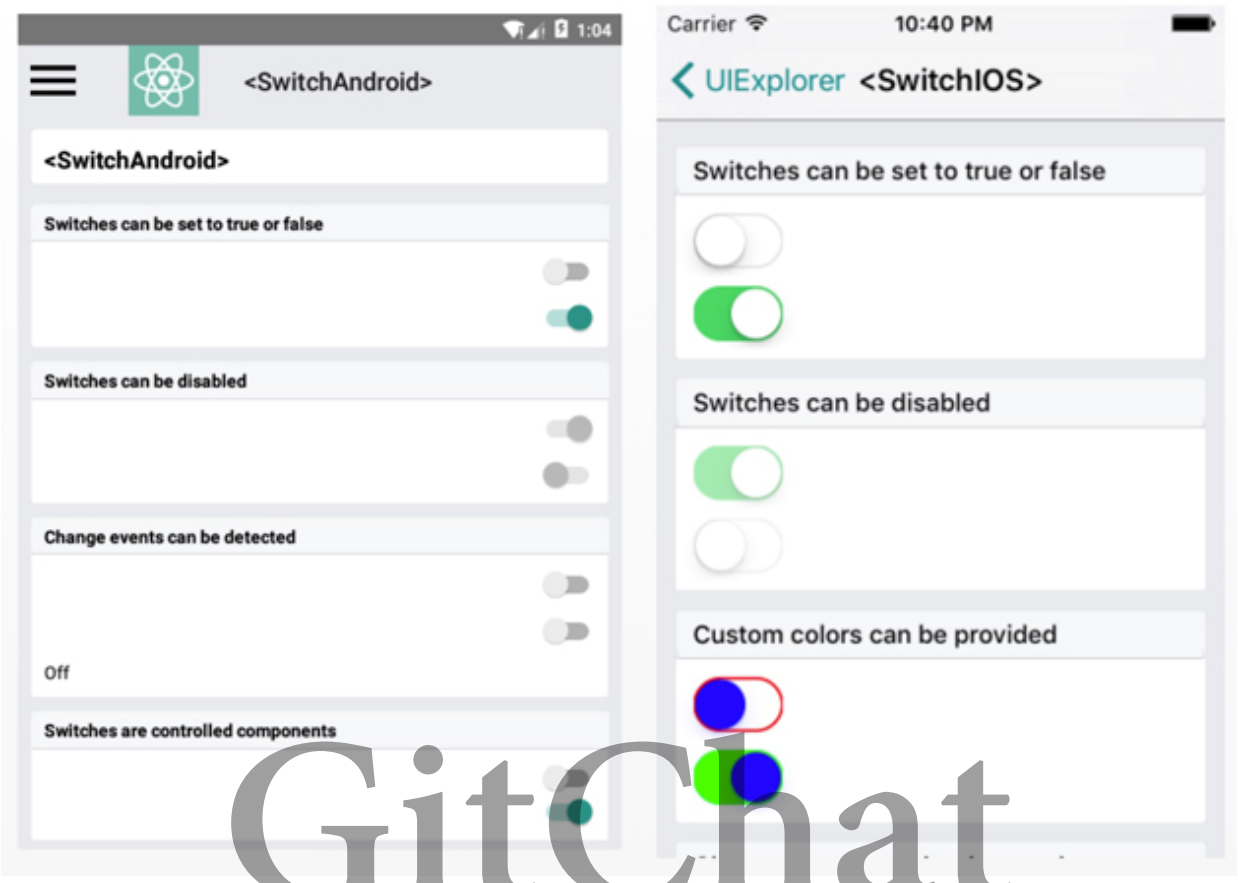
RN 框架不仅在 UI 组件上存在较大的差异化，也没有从 JS 层提供一套统一的 API 支持跨活动（Activity）的路由切换方案，提供的 Fetch 功能也十分有限。因此针对 RN 的这些短板，我们不仅基于 iOS 风格统一封装了一套 JS 组件，还开发了一套 Ext 拓展框架，支持了 RN 的路由方案，Fetch 的拓展等，从内部消化差异，提供一致对外的 API，提升了业务线童鞋的开发体验。

UI 组件

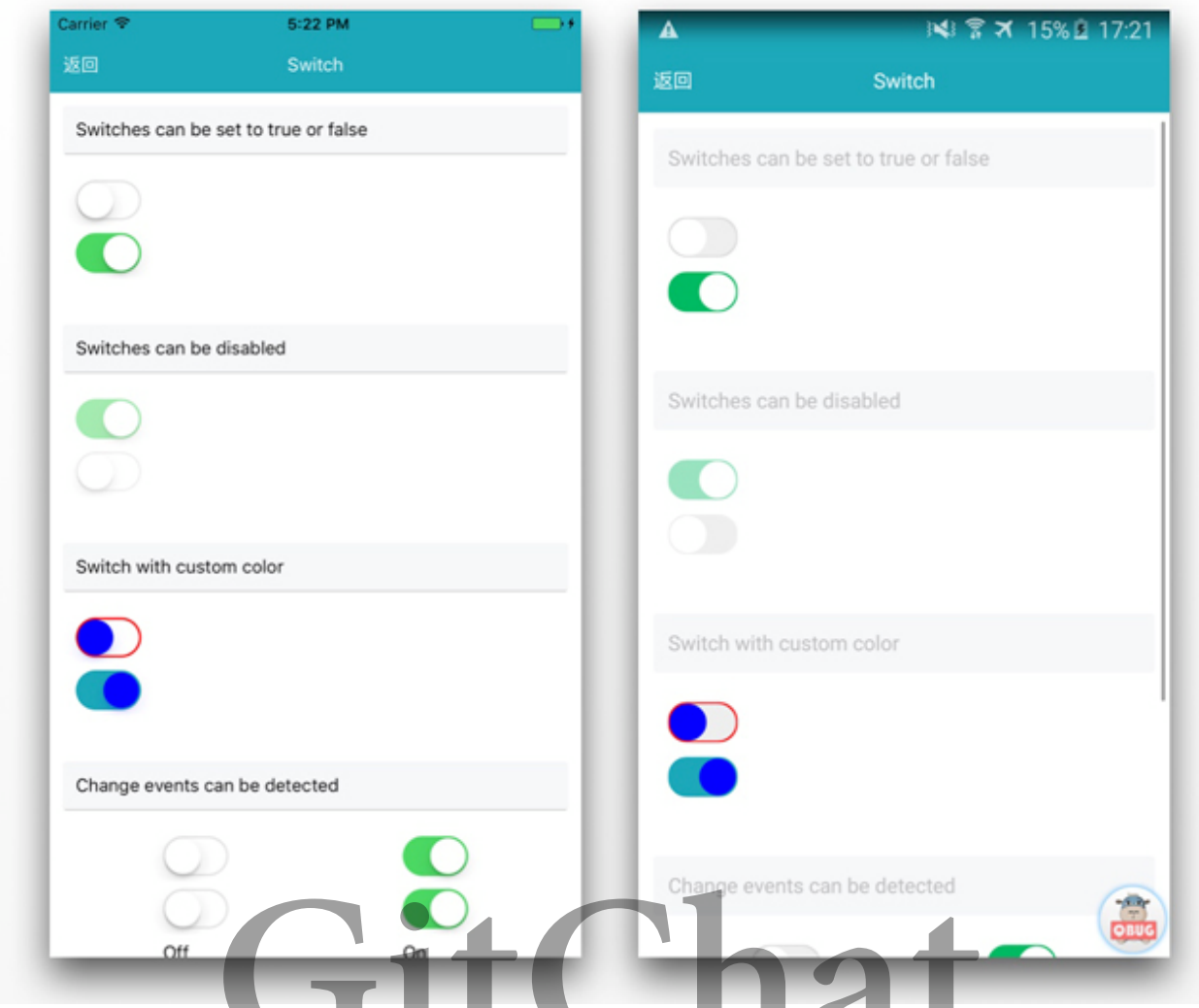
由于 RN 原生组件的样式存在很多差异，针对这种情况我们从 JS 层统一封装了一套对外一致的组件，其中有些组件仅仅在 JS 层组装现有的 UI 组件即可，比如

`Button`、 `Modal`和 `Loading`等，针对较为复杂的组件，如 `Picker`、 `TimePicker`和 `Switch`等，则需要从 Native 层进行渲染，封装一个原生的UI模块。

RN 原生的 Switch 组件效果图：



经过封装的 Switch 组件效果图：



Ext 拓展框架

Ext 是我们基于 RN 深度定制的 QRN（Qunar React Native）框架的重要组成部分，其主要包括了 `Router`、`Data Stream` 和 `WebX` 三大核心部分，各个部分的职责如下：

- Router：路由部分，除了支持 RN 内部的路由切换，通过 Native 层实现的 `VC Manager` 原生模块，支持跨活动的路由切换，对外统一的 `Nativegator` 接口；
- Data Stream：数据流部分，包含 `Redux` 和 `Fetch` 两个模块，针对 `Redux` 部分，我们改造了 `Redux` 数据流部分，简化了 `store` 的配置方式，结合 Ext 提供多种 `store` 管理方式；对于 `Fetch` 部分，我们提供中间件支持，支持简便的请求配置，增加 `about` 和 `timeout` 等特性支持。
- WebX：增强 Web 式的开发体验，支持 JSX 直接定义 `class` 属性，支持 `vh` 和 `vw` 单位等。

Ext 的 `Router` 路由部分，比较核心的就是封装了一套 API 实现了 JS 栈和 Native 栈的无缝处理了。由于公司内部跨业务线的开发场景很多，比如某个 RN 的下单页面需要从支付的 Native 页面获取一些支付信息，又或者需要从 RN 页面跳转到某个 H5 页面，然后又从 H5 页面唤起另一个 RN 的页面。针对这类业务需求，我们提供了 `Scheme` 和

`Broadcast` 两种方式的通信机制，通过 `Scheme` 可以直接唤起一个新的原生页面，通过 `Broadcast` 可以静默发送消息给其他的 RN 或者 Native 页面。

> 关于 Ext 的更多信息，可以访问我们的文档站了解。YMFE Ext 相关文档

热更新

相信很多公司使用 RN 的重要因素之一是看中了它与生俱来的热更新能力，然而 RN 框架本身并未提供热更新解决方案，在面对国内快速变化的市场，敏捷开发变得炙手可热的情况下，设计一套安全稳定的热更新解决方案无疑有助于产品在市场中处于竞争有利的地位。我们的热更新框架具有以下几个优势：

- 灵活的版本控制。
- 安全的数字签名验证。
- 完善的数据统计。
- 稳定的服务集群。
- 适用于任何采用文件更新机制的移动框架。

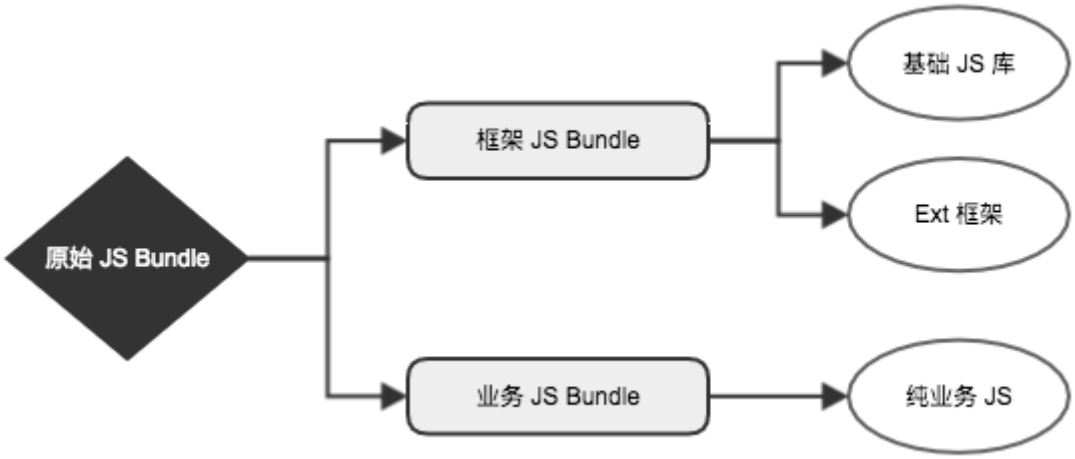
QP 包

在介绍公司的热更新框架之前，需要介绍下我们自定义的 `QP`（Quick Package）文件，它采用二进制格式存储，是 Hybrid 和 QRN 框架使用的离线包格式，具备加载迅速，资源映射等特点。

> 关于 QP 包的更多信息，可以点此了解。

JS Bundle 拆分

JS Bundle 拆分算是我们实现热更新系统最重要的前提之一，RN 的 JS Bundle 默认将所有的 JS 资源打成一份，在 Bridge 初始化的时候进行加载，在小型的应用开发中，这样做没什么问题，然而对于一个大型的 App 而言，每个 JS Bundle 都会重复包含一份相同的框架代码，这种情况引发的 Size 问题是令人难以接受的。又因为 JS Bundle 本身带有特定的业务代码，在预加载机制上的实现也是困难重重。



针对以上的问题，我们将 JS Bundle 拆分成了两个文件，分别是 框架 JS Bundle 和 业务 JS Bundle，下面是这两个 JS Bundle 的职责说明：

- 框架 JS Bundle：包含了基础的 JS 库和我们封装的 Ext 框架。
- 业务 JS Bundle：仅包含业务自身的 JS 相关文件。

将重合部分抽成独立的 JS Bundle，我们可以轻松的实现预加载机制，即 Bridge 可以静默创建并预加载好通用的 JS Bundle 部分，在进入具体业务的时候仅需提取预加载的 Bridge 注入业务的 JS 即可实现 RN 页面的秒开效果。

差分更新

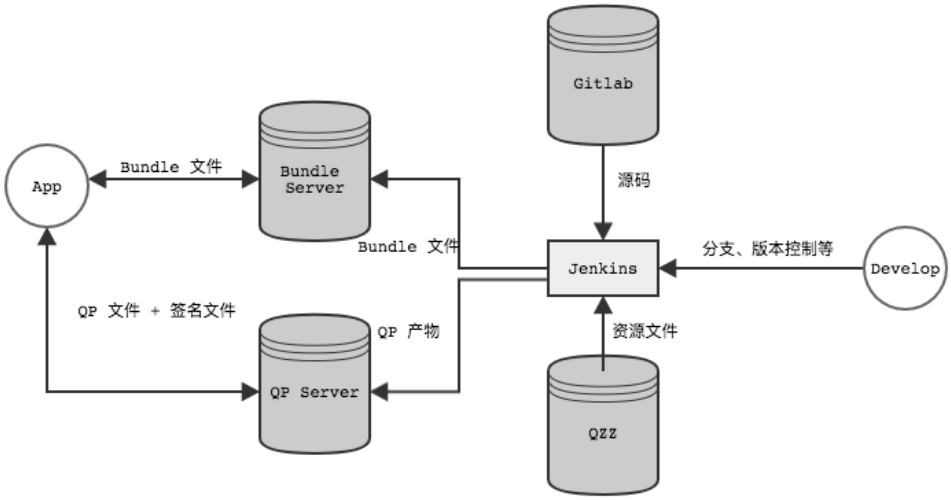
业务线快速迭代产品过程中可能需要频繁的发布更新包，如果每次都让用户去下载一个完整的离线包，无论对用户流量还是服务器资源来说都是一个极大的浪费，为此我们会在请求的时候带上当前的离线包版本信息，通过二进制差分算法 diff 出目标文件，改动部分越小则差分文件越小，一般只有几 kb，通过这种方式无疑为用户节省了非常多的宝贵流量。

版本控制

RN 的方案与 HyBrid 方案区别在于 RN 的代码结构目前来说不算稳定，RN 版本的升级可能导致 JS Bundle 无法向下兼容，而业务线可能会针对某个历史版本发布一个修复，例如解决厂商预装的问题等。为此版本控制除了要限制生效的版本下限，同时还需要提供作用的上线支持区间版本热更新。

热更新框架

QP 包作为我们离线包数据载体，包含了 JS Bundle 文件以及 IconFont和图片等资源文件，由我们的 QP 服务器下发给用户终端，下图是一个简化的流程图。



一次完整的热更新流程如下：

1. Develop 角色在 Jenkins 上配置要发布的 Job 相关信息，包括本次发布的业务 Git 分支和生效的目标版本和组件版本范围，配置完成开始打包。

2. Jenkins 在打包开始时候从 Gitlab 服务器拉取业务的 Git 源码以及打包脚本，根据业务使用的资源配置从 QZZ 服务器中获取相关的资源文件，如 IconFont 和图片资源等。

3. 打包脚本执行后将本次生成的 JS Bundle 文件上传到 Bundle 服务器中，将 JS Bundle 和资源文件打包成 QP 文件，连同生成的加密 MD5 签名文件上传到 QP Server 中。

4. App 进入 RN 页面时候会请求更新离线包资源，如果线上 QP Server 的 QP 包资源有更新，则返回与当前 App 本地的离线包差分之后的文件，App 下载并合并差分的离线包资源，解密 MD5 校验 QP 资源完整性，合并校验完成之后，一般情况下重启应用生效。

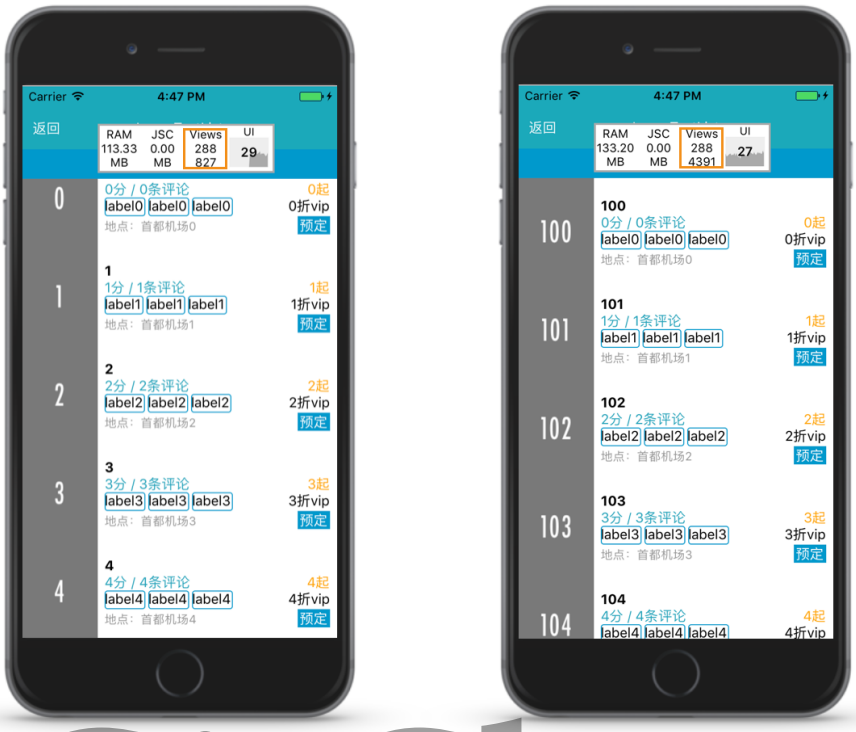
5. App 请求 QP 离线包是一个静默过程，在进入 RN 页面时候如果不存在相应的 QP 离线包，则直接请求 Bundle Server 加载线上 Bundle 资源文件。

ListView 优化

RN 原生的 ListView 不像原生的 UITableView 或者 ListView 可以复用子节点，在短列表上表现尚可，长列表则存在渲染速度慢，内存占用大等严重的性能问题；并且在 Android 平台上使用原生 ListView 频繁滚动还会出现 JS 组件响应严重延迟的现象。针对实战中发现的这些问题，我们写了一套 JS 层的 ListView 优化。

RN ListView 原理

RN 的 ListView 组件为了保存每一个节点的状态，当某个节点从屏幕区域内移出时，其虚拟 DOM 中依然保存该节点，并没有做到真正的节点复用，RN ListView 采用的是渐进式渲染，虚拟 DOM 中的 View 数量随着节点的增加不断增长，内存也随之上升，在无尽列表的情况下可能会因为内存耗尽而导致应用崩溃。



上图是使用 RN ListView 从节点 0 滑动到节点 100 的截图，虚拟 DOM 的 View 数量不断增加。

RN ListView 性能

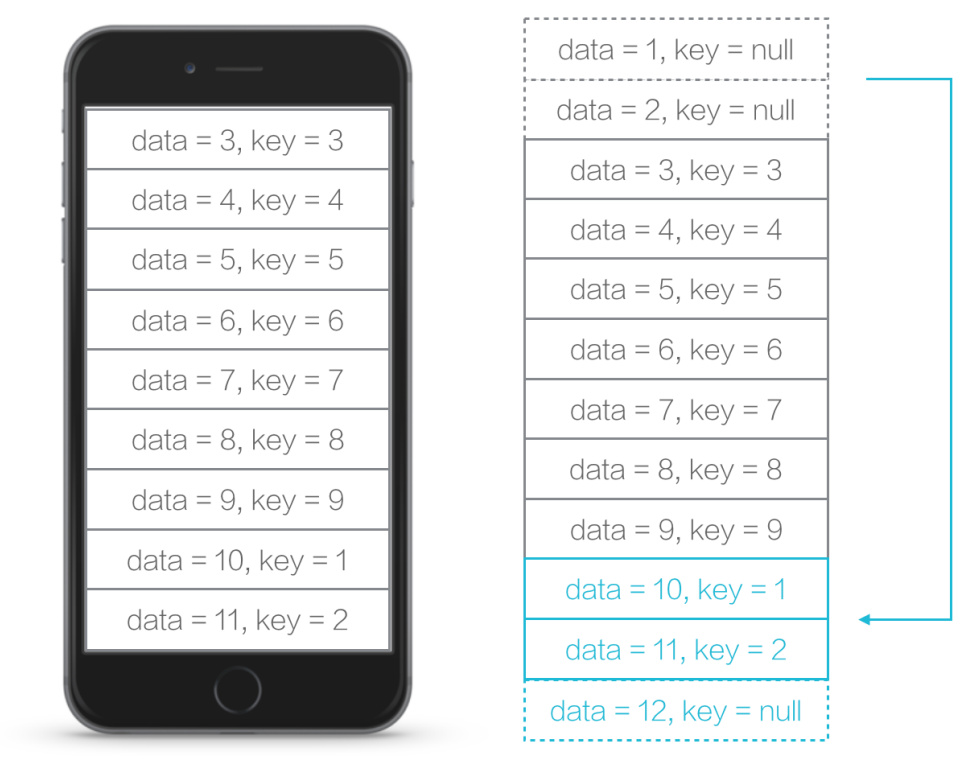
通过改变列表的行数，其内存占用的情况如下图所示，其内存增长情况比较明显。

行数	RN ListView 占用内存
20行	123MB
200行	156MB
2000行	432MB

QRN ListView 原理

针对 RN ListView 存在的问题，我们开发出了 QRN ListView，与 RN ListView 维持节点引用的方式不同，而是通过 React 的虚拟 DOM 机制实现了节点复用。

React 在比较列表节点时候，为了保证操作虚拟 DOM 节点的高效，要求传入 `key` 属性，我们通过将被移除的 `key` 的节点赋给即将渲染的数据项来完成复用逻辑。



如上图所示，在操作虚拟 DOM 树时候，原先 key 为 1 的节点绑定第 1 个数据项，当第 1 个数据项移出屏幕之后，将 key 为 1 的节点移动到 key 为 9 的节点之后，并绑定第 10 个数据项，从而完成节点复用。

QRN ListView 性能

行数	RN ListView占用内存	QRN ListView占用内存
20行	123MB	136MB
200行	156MB	140MB
2000行	432MB	155MB

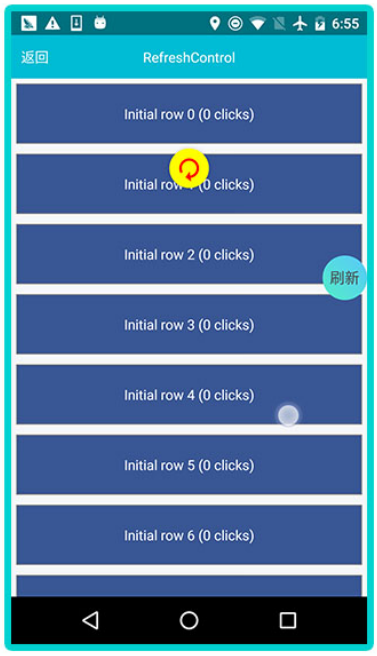
从上图可以看出，当列表数量不断增加的时候，QRN ListView 的内存占用量依然保持稳定，从而解决了 RN ListView 占用过高的情况。

Refresh Controller

由于平台差异化的实现，ListView 的下拉刷新在 iOS 和 Android 两端表现也不一致，Android 的 ListView 列表没有弹性下拉刷新的效果，与 iOS 的原生效果并不一致。针对此类情况我们从 JS 层面封装了 Refresh Controller 组件。

原生 RN 的 Refresh Control 如下图：

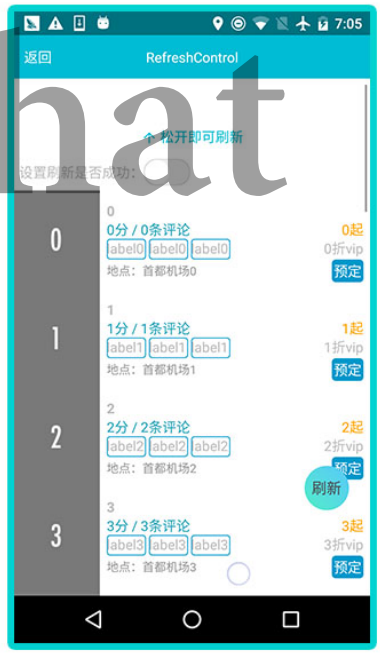
IPhone 7



Nexus 5

QRN 的 Refresh Control 如下图：

IPhone 7



Nexus 5

RN ListView 导致 JS 事件阻塞的问题

这里再说下前面提到的 RN ListView 存在的一个问题，在使用 RN ListView 进行快速滚动的过程中，如果点击了页面的某个按钮（例如返回按钮），在 Android 平台上很容易出现响应迟滞的现象。
具体问题场景如下所示：

问题出现的原因是 Android 的 RN ListView 不支持 `scrollEventThrottle` 属性，这个属性针对 Scroll 事件进行了节流。解决的思路也很简单，就是在 RecyclerViewBackedScrollView 中 `onScrollChanged` 方法中增加一个节流器，限制单位时间内的发送的 Scroll 事件数量。

> 关于 QRN ListView 的更多信息可以参考这里

Android 实践中的一些坑

众所周知，Android 由于其开源性质，各类厂商定制 ROM 和纷繁的系统版本导致其平台严重的碎片化。在我们使用 RN 的过程中也踩了不少的坑，当然其中也是框架选型导致的问题的。

SoLoader 加载问题

问题描述

SoLoader 是 Facebook 提供的一个用于加载 so 的开源库，我们发现在使用了插件化方案的 App 中，可能一些 7.0 设备会出现 so 依赖加载不全的问题，出现问题是因为 SoLoader 仅在 SDK <= 17 的情况下才会主动解决 so 加载依赖。

解决方案

如果你在实践中也遇到这个问题，比较简单的方式就是通过定制 SoLoader 库解决。

自定义手势导致的崩溃

问题描述

在华为或者 OPPO 的一些机型上，可能有类似 三指截屏 之类的自定义手势操作，我们发现这种类型的手势很容易引发 RN 的崩溃，抛出此类异常信息 `Tried to get non-existent cookie` ，通过对比事件序列的信息，我们发现自定义手势操作产生的事件序列，Action为 `Down` 时候的 `downTime` 与 Action 为 `Move` 和 `Up` 事件的均不相同，而 RN 使用 Touch 事件的 `downTime` 对同一个事件序列进行跟踪。

解决方案

厂商定制 ROM 中的自定义手势一般都有特定的功能被触发，RN 可以直接忽略掉这类特殊的事件，可以在 `JSTouchDispatcher` 的 `onChildStartedNativeGesture` 方法中可以针对这类事件直接忽略就行。

IconFont 异步加载

问题描述

RN 官方并不支持 IconFont 的异步加载方案，在业务场景中，我们可能需要从线上获取 IconFont 文件并在异步下载完成之后刷新。

解决方案

针对 IconFont 的异步加载，这里提供一个简单的思路， `CustomStyleSpan` 类是 RN 的 Text 组件加载自定义 `Typeface` 的关键类，通过修改 `apply` 方法签名传入 `ReactShadowNode` 的软引用实例，并在 `apply` 方法中加入一个自定义的 IconFontManager；当 IconFont 异步加载完成之后，通过 `ReactShadowNode` 实例调用 `dirty()` 并通过 `UIImplementation` 实例调用 `dispatchViewUpdates(-1)` 即可。

欢迎访问我们的团队博客，获取更多有用的干货。博客：YMFE BLOG.