

## 26 | 客户端优化：如何把性能提升到极致？

2022-05-27 况众文 朴惠姝

《React Native 新架构实战课》

[课程介绍 >](#)



讲述：蒋宏伟

时长 21:49 大小 19.99M

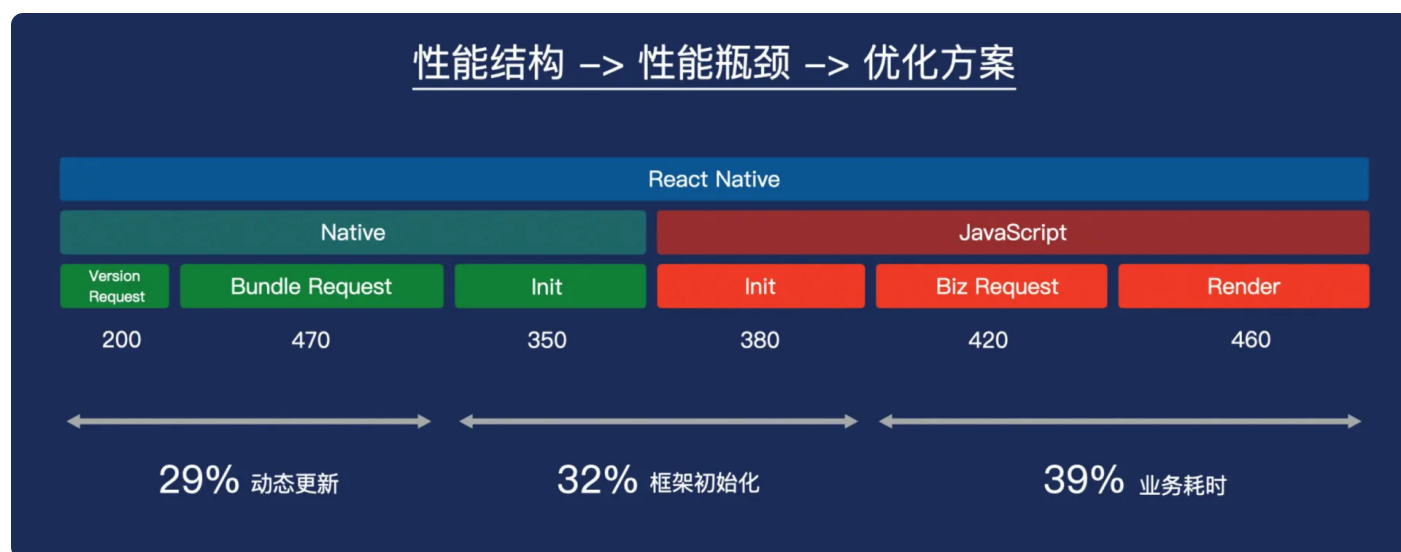


你好，我是众文，这一讲继续由我和惠姝来讲解。第 22 讲中我们讲解了如何用自定义组件满足业务的个性化需求，除了这一点之外，在 **React Native** 的应用中，还有一点是大家探讨得比较多的，就是性能优化这部分。

和原生开发相比，**React Native** 比较明显的不足在于页面加载速度，比如秒开率、页面加载的时长等。但在我们实际的落地过程中，**React Native** 页面达到了秒开的级别，我们是如何做到的呢？

其实，一个未经优化的、比较复杂的、动态更新的 **React Native** 应用，从大体上讲，可以分为 3 个瓶颈（以下数据来自我们的实际业务案例）：

动态更新瓶颈	占比为 29%
初始化瓶颈	占比为 32%
业务耗时瓶颈	占比 39%



当然，其中还涉及 JavaScript 侧的优化。今天我们主要站从客户端角度，讲述 React Native 如何在客户端侧将性能优化到极致，带你开启 React Native 的秒开世界。

## 环境预创建

在 React Native 最新架构中，Turbo Module 是按需加载，而不是像旧框架一般，一股脑初始化所有的 Native Modules，同时 Hermes 引擎放弃了 JIT，在启动速度方面也有明显提升。

那么，抛开这两个新版本的优化，在启动速度方面，客户端还能做些什么呢？有的，那就是 **React Native 环境预创建**。

在混合工程中，React Native 环境与加载页面的关系如下：



从上图中可以看到，在混合应用中，独立的 **React Native** 载体页都拥有自己的执行环境。**Native** 域包括 **React View**、**Native Modules**；**JavaScript** 域包括 **JavaScript** 引擎、**JS Modules**、业务代码；中间通信使用 **Bridge/JSI**。

当然，业内也有多个页面复用一个引擎的优化。但是多页面复用一个引擎存在一些问题，比如 **JavaScript** 上下文隔离、多页面渲染错乱、**JavaScript** 引擎不可逆异常，等等。而且复用的性能不稳定，考虑到投入产出比、维护成本等方面，通常在混合开发中，采用的是一个载体页一个引擎。

一个 **React Native** 页面加载渲染逻辑，可以大致分为以下几步：

[复制代码](#)

```
1 React Native 环境初始化 -> 下载/加载 bundle -> 执行 JavaScript 代码
```

环境初始化这一步包含创建 JavaScript 引擎、Bridge、加载 Native Modules（旧版）。根据我们的测试，初始化这一步，特别是在 Android 环境中，比较耗时。

那么，如何进行 React Native 环境初始化耗时优化呢？我们可以提前将 React Native 环境创建好，流程如下：



具体的代码如下（Java）：

复制代码

```
1 public class RNFactory {
```

```

2 // 单例
3 private static class Holder {
4     private static RNFactory INSTANCE = new RNFactory();
5 }
6
7 public static RNFactory getInstance() {
8     return Holder.INSTANCE;
9 }
10
11 private RNFactory() {
12 }
13
14 private RNEnv mRNEnv;
15
16 // App 启动时调用 init 方法，提前创建一个 RN 环境
17 public void init(Context context) {
18     mRNEnv = new RNEnv(context);
19 }
20
21 // 获取 RN 环境对象
22 public RNEnv getRNEnv(Context context) {
23     RNEnv rnEnv = mRNEnv;
24     mRNEnv = createRNEnv(context);
25     return rnEnv;
26 }
27 }

```

## RNEnv.java:

 复制代码

```

1 public class RNEnv {
2     private ReactInstanceManager mReactInstanceManager;
3     private ReactContext mReactContext;
4
5     public RNEnv(Context context) {
6         // 构建 ReactInstanceManager
7         buildReactInstanceManager(context);
8         // 其他初始化
9         ...
10    }
11
12    private void buildReactInstanceManager(Context context) {
13        // ...
14        mReactInstanceManager = ...
15    }
16
17    public void startLoadBundle(ReactRootView reactRootView, String moduleName,
18        // ...
19    }
20 }

```

在做预创建时，我们需要注意**线程同步**问题。在混合应用中，React Native 由应用级变成页面级使用，所以在线程安全这方面有不少的问题，预创建时会并发创建多个 React Native 环境，而 React Native 环境内部构建存在异步处理，一些全局的变量，如 ViewManagersPropertyCache:

```
1 class ViewManagersPropertyCache {
2     private static final Map<Class, Map<String, ViewManagersPropertyCache.PropS
3     private static final Map<String, ViewManagersPropertyCache.PropSetter> EMPTY
4
5     ViewManagersPropertyCache() {
6     }
7     ...
8 }
```

内部的 CLASS\_PROPS\_CACHE、EMPTY\_PROPS\_MAP 都是非线程安全的数据结构，并发时可能会存在 Map 扩容转换问题（HashMap Node 转红黑树结构），又比如 DynmicFromMap 也有此问题：

Fix DynamicFromMap object pool synchronization #17842

**Closed** haitaoli wants to merge 1 commit into facebook:master from haitaoli:android-fix-synchronization

Conversation 11 · Commits 1 · Checks 0 · Files changed 1

haitaoli commented on Feb 3, 2018 · edited

**Motivation**

DynamicFromMap internally uses SimplePool object to recycle dynamic prop objects. But the pool is not multi-thread safe. Currently the most used dynamic props are size props such as left, paddingVertical, marginTop and so on. These props are only accessed from the layout thread so the pool works fine. If a dynamic prop is needed in UI thread, then the two threads can access the same pool object and cause random errors. This PR make the pool object thread local to avoid synchronization. After this change there are two pool objects created in the process.

**Test Plan**

Tested in official Airbnb app after updating accessibilityComponentType to be dynamic.

**Related PRs**

Once this is merged, I'll send another PR to support "disabled" state in accessibilityComponentType.

**Release Notes**

[ANDROID] [BUGFIX] [DynamicFromMap] - Fix a crash caused by dynamic props

**Reviewers**

- janicduplessis ✓
- facebook-github... ✓

**Assignees**

No one assigned

**Labels**

- Bug
- CLA Signed
- Merged
- Ran Commands

**Projects**

None yet

**Milestone**

No milestone

**Linked issues**

那么，这个问题如何解决呢？你可以参考《混合应用：如何从零开始集成 React Native？》框架 Bug 修复部分，对同步的地方进行处理。



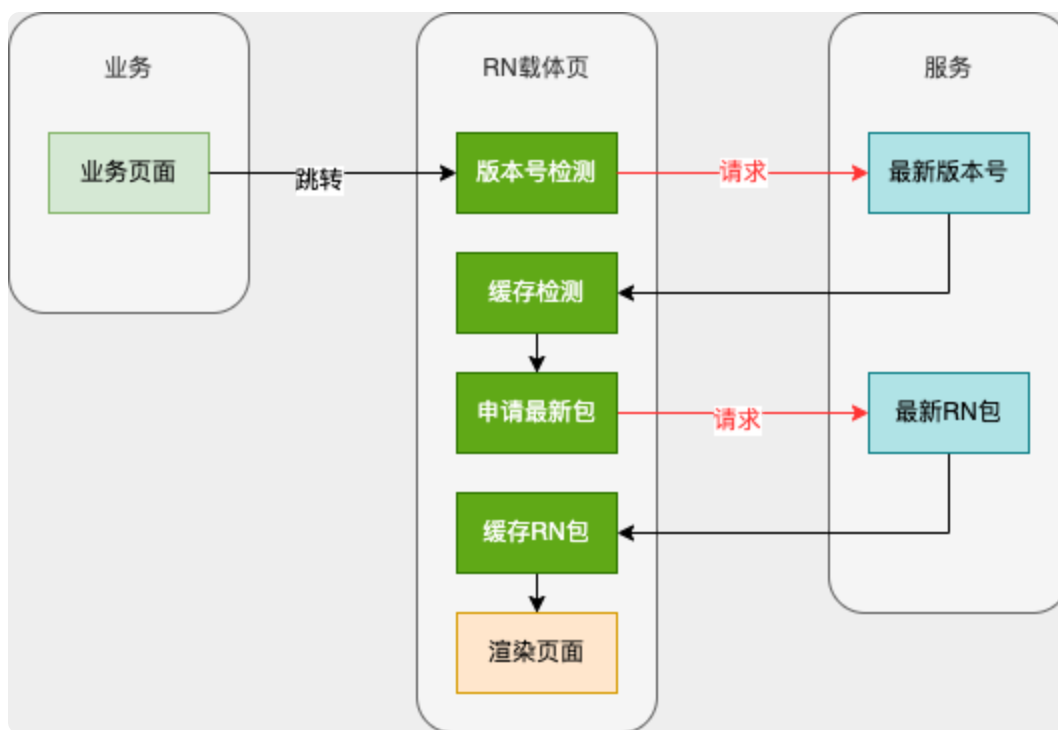
## 异步更新

原先我们进入 React Native 载体页后需要先下载最新的 JavaScript 代码包版本，若有更新，就要下载最新的包并加载。在这个过程中，我们会经历两次网络请求。如果用户网络比较差，那他从进入页面到渲染页面内容需要等待较长时间。

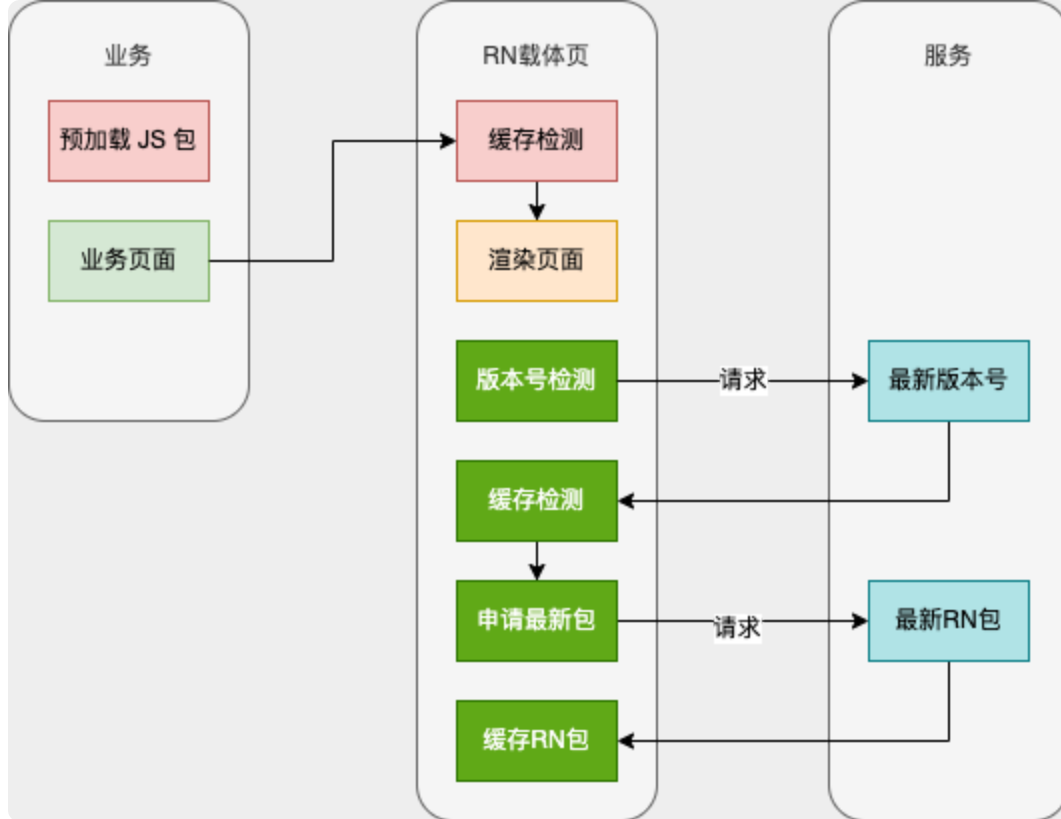
所以我们针对部分特殊的页面，采取了异步更新的策略。这里所说的特殊页面可以由业务来指定，比如更新频率相对比较低的页面、页面进入路径较短的页面，等等。

异步更新策略的主要思路为在进入页面之前选择性地提前下载 JavaScript 代码包，进入载体页后再看 JavaScript 代码包是否有缓存，如果有，我们就优先加载缓存并渲染；然后再异步检测是否有最新版本的 JavaScript 代码包，如果有，下载到本地并进行缓存，再等下次进入载体页时生效。

我们先看一下从一个页面进入到一个 React Native 载体页后需要哪些流程：



流程图中可以看出，我们从进入载体页到渲染页面，需要两次网络请求，不管网速快还是慢，这个流程算是比较漫长的，但在进行异步更新后，我们的流程就会变成下图这样：



在业务页面中，我们可以对 JavaScript 代码包进行提前下载并缓存，在用户跳转到 React Native 页面后，检测是否有缓存的 JavaScript 代码包，如果有我们就直接渲染页面。这样就不需要等待版本号检测网络接口以及下载最新包的接口，也不依赖于用户的网络情况，减少了用户等待时间。

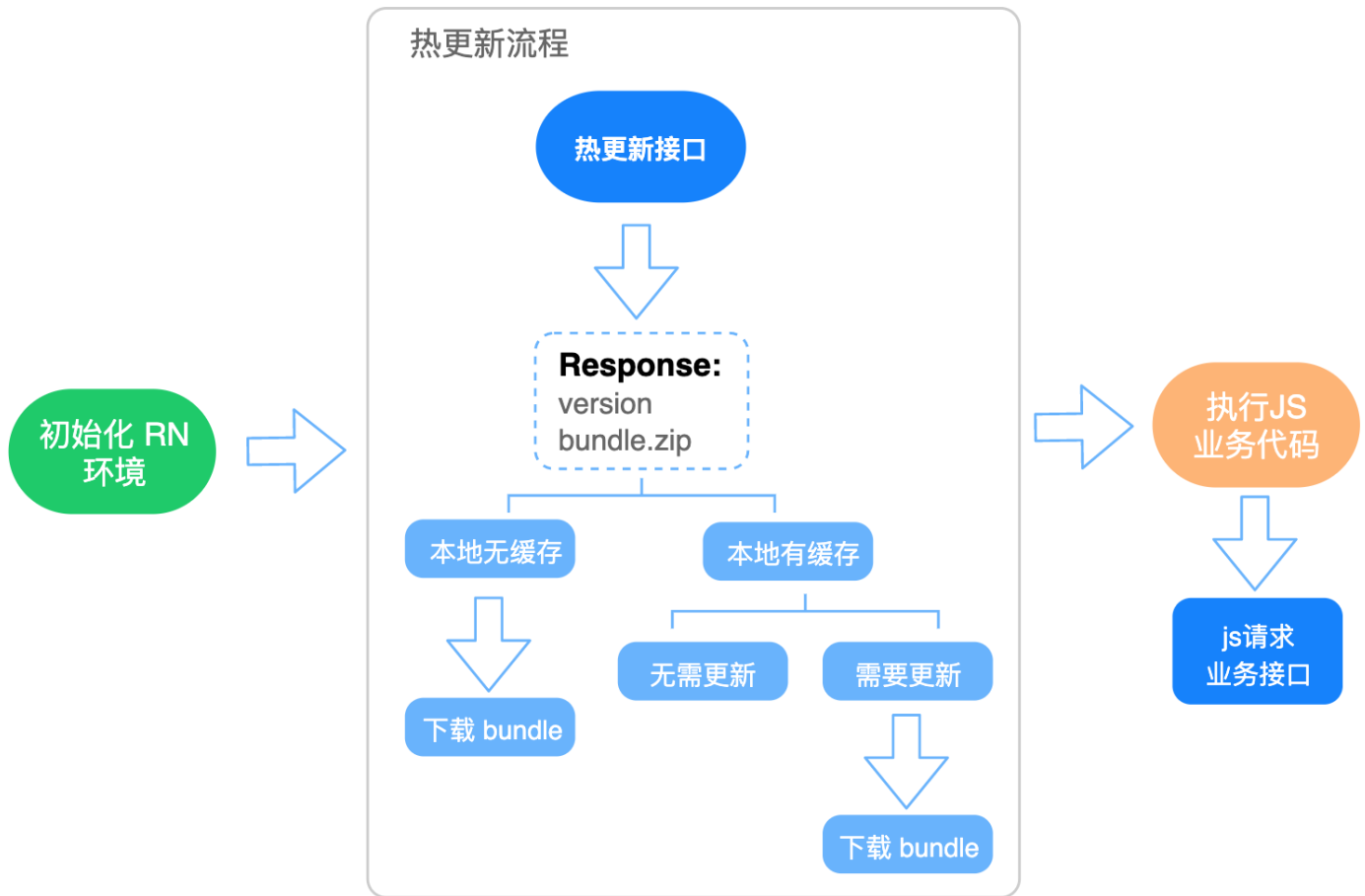
在渲染页面的同时，我们通过异步检测 JavaScript 代码包的版本，若有新版本就进行更新并缓存，下次生效。当然，业务也可以选择更新完最新包之后，提示用户有新版本页面，以及是否选择刷新并加载最新页面。

## 接口预缓存

对 React Native 环境初始化、bundle 加载流程进行优化后，我们的 React Native 页面就可以达到秒开级别了。不过，React Native 页面加载后，进入 JavaScript 业务执行区间，大部分业务都不可避免地会进行网络交互，请求服务器数据进行渲染，这部分其实也有很大的优化空间。我们现在就来分析分析。

我们先来看下具备热更新能力的 React Native 加载流程：

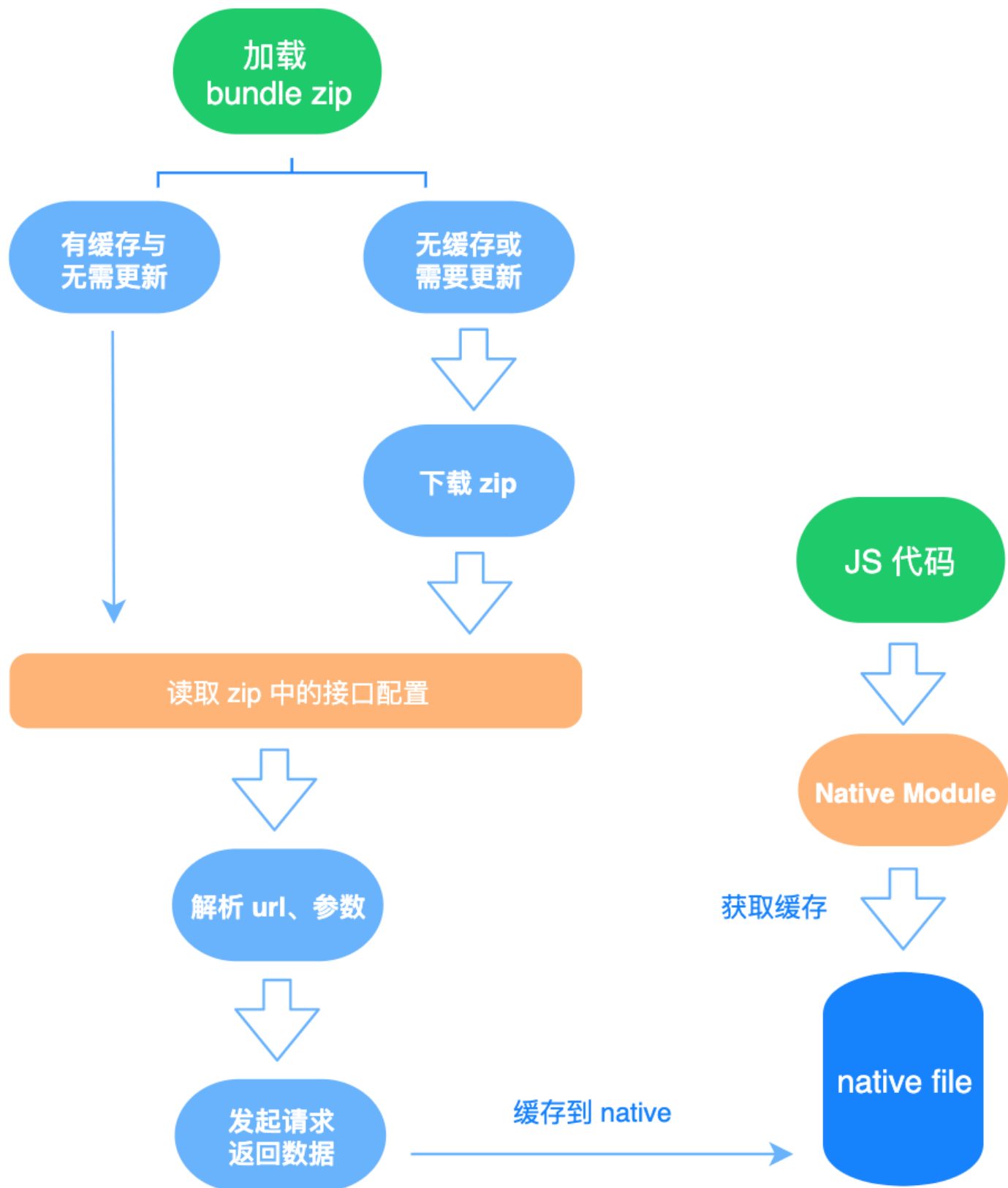




你可以看到，整个流程是从 **React Native** 环境初始化到热更新，再到 **JavaScript** 业务代码执行，最后到业务界面展示。链路比较长，而且每一个步骤都依赖前一个步骤的结果。特别是热更新流程，最长可涉及两次网络调用，分别是检测是否需要更新与下载最新 **bundle** 文件。

那么这个时候，我们就想到，在等待网络返回的过程中，**Native** 能不能把闲置的 **CPU** 资源利用起来呢？

我们都知道，目前手机性能越来越强大，多核、**4G/5G** 使我们的“冲浪”体验越来越好。而且，**Native** 具备先天的多线程能力。在纯客户端开发中，我们经常使用接口数据缓存策略来提升用户体验，在最新数据返回前，先使用缓存数据进行页面渲染。那么在 **React Native** 中，我们也可以参考这一思路，对整个流程进行优化：



具体代码，我也放在了下面 (Java)。

- 首先是预请求实体类：

```
1 public class PrefetchBean {
```

复制代码

```

2   public String url; // 预加载的接口
3   public String method; // 请求方式: GET/POST...
4   public Map<String, String> headers; // 请求头
5   public Map<String, String> params; // 请求参数
6 }

```

- 打开载体页时，解析对应 **bundle** 缓存中的预请求接口配置数据，发起请求缓存数据：

 复制代码

```

1 public class RNApiPreloadUtils {
2     public static void preloadData(String bundleId) {
3         // 根据 bundle id 解析对应的预请求接口配置，可存在多个接口
4         List<PrefetchBean> prefetchBeans = parsePrefetchBeans(bundleId);
5         // 请求接口，成功后缓存到本地存储
6         requestDatas(prefetchBeans);
7     }
8
9     public static String prefetchData(String url) {
10        // 从本地缓存中，根据 url 获取对应的接口数据
11    }
12 }

```

- 获取接口缓存数据的 Module:

 复制代码

```

1 public class PrefetchBusinessModule extends ReactContextBaseJavaModule
2     implements ReactModuleWithSpec, TurboModule {
3     public PrefetchBusinessModule(ReactApplicationContext reactContext) {
4         super(reactContext.react());
5     }
6
7     @ReactMethod
8     public void prefetchData(String url, Callback callback) {
9         String data = RNApiPreloadUtils.prefetchData(url);
10        // 回传数据给 JS
11        WritableMap resultMap = new WritableNativeMap();
12        map.putInt("code", 1);
13        map.putString("data", data);
14        callback.invoke(resultMap);
15    }
16 }

```

- JavaScript 调用：

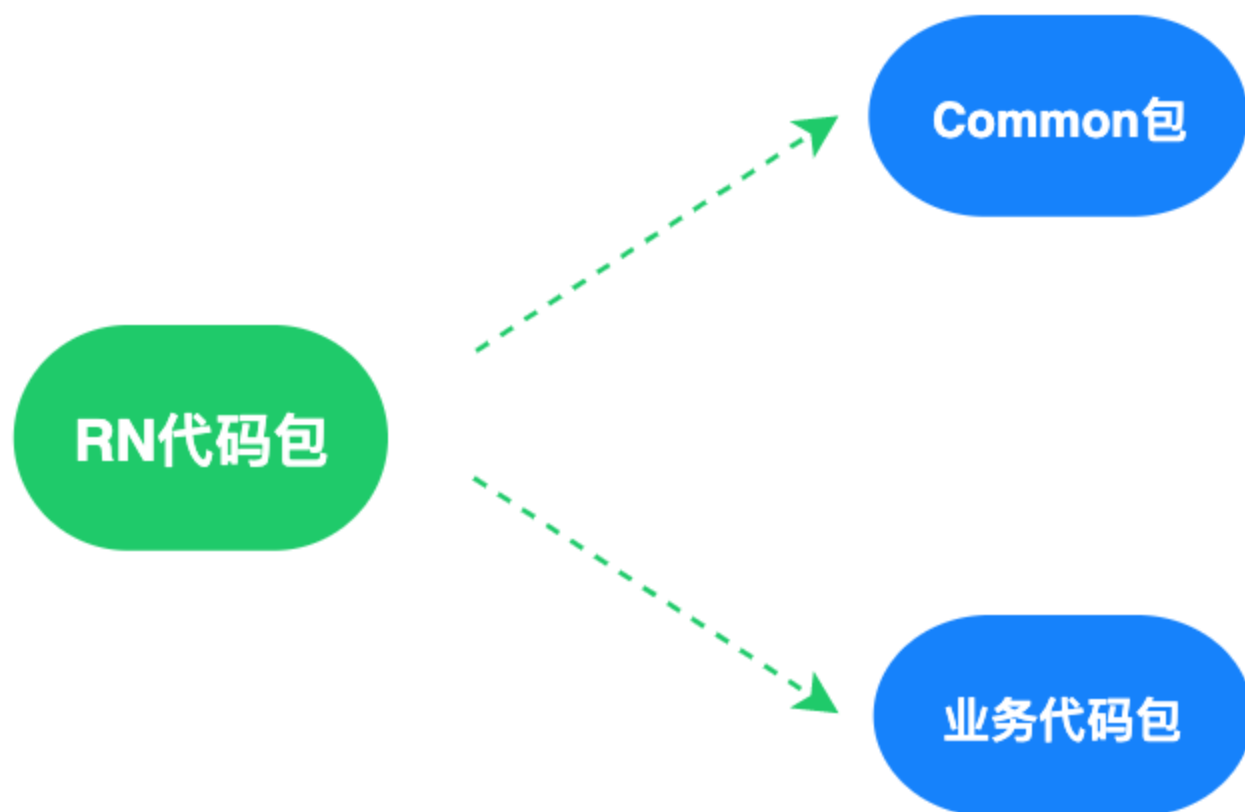
```
1 NativeModules.PreFetchBusinessModule.prefetchData(url, (result)=>{  
2     // 获取到结果后，判断是否为空，不为空解析数据，渲染页面  
3     console.info(result);  
4 }  
5 );
```

## 拆包

前面也提到了，React Native 页面的 JavaScript 代码包是热更新平台根据版本号进行下发的，每次有业务改动，我们都需要通过网络请求更新代码包。

不过，其实只要 React Native 官方版本没有发生变化，JavaScript 代码包中 React Native 源码相关的部分是不会发生变化的，所以我们不需要在每次业务包更新的时候都进行下发，在工程中内置一份就好了。

因此，我们可以将一个 JavaScript 代码包拆分成两个部分：一个是 Common 部分，也就是 React Native 源码部分，这一部分除非 React Native 官方版本进行升级，几乎不会发生变化；另一个是业务代码部分，也就是我们需要动态下载的部分。



我们在打包时，对 **React Native** 代码包进行处理，拆分成 **Common** 包和业务代码包。

**Common** 包内置到工程中（至少为几百 **kb** 的大小），业务代码包进行动态下载。然后我们利用 **JSContext** 环境，在进入载体页后在环境中先加载 **Common** 包，再加载业务代码包就可以完整的渲染出 **React Native** 页面：

 复制代码

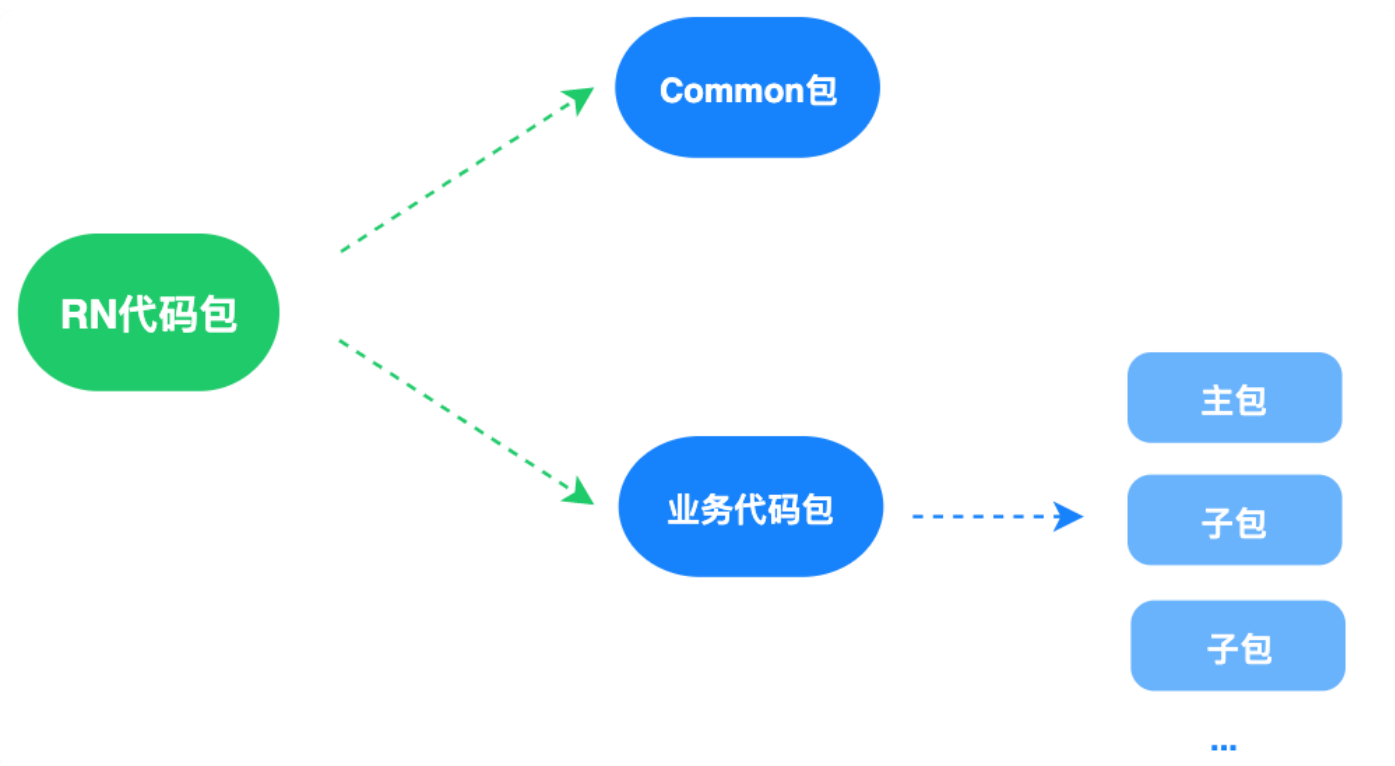
```
1 //载体页
2 - (void)loadSourceForBridge:(RCTBridge *)bridge
3         onProgress:(RCTSourceLoadProgressBlock)onProgress
4         onComplete:(RCTSourceLoadBlock)loadCallback{
5     if (!bridge.bundleURL) return;//加载新资源
6     //开始加载bundle, 先执行common bundle
7     [RCTJavaScriptLoader loadCommonBundleOnComplete:^(NSError *error, RCTSource
8         loadCallback(error,newSource);
9     }]];
10 }
11
12 //common执行完毕
13 + (void)commonBundleFinished{
14     //开始执行buz bundle代码
15     [RCTJavaScriptLoader loadBuzBundle:self.bridge.bundleURL onComplete:^(NSEr
16         loadCallback(error,newSource);
17     }]];
18 }
19
20 //RCTJavaScriptLoader.mm
21 + (void)loadBuzBundle:(NSURL *)buzURL
22         onComplete:(WBSourceLoadBlock)onComplete{
23     //执行buz包代码
24     [self executeSource:buzURL onComplete:^(NSError *error){
25         onComplete(error);//执行完毕
26     }]];
27 }
```

在这里要注意，**Common** 包和业务代码包必须要成对进行加载，否则页面无法展示。

## 按需加载

其实我们通过前面拆包的方案，已经减少了动态下载的业务代码包的大小。但是还会存在部分业务非常庞大，拆包后业务代码包的大小依然很大的情况，依然会导致下载速度较慢，并且还会受网络情况的影响。

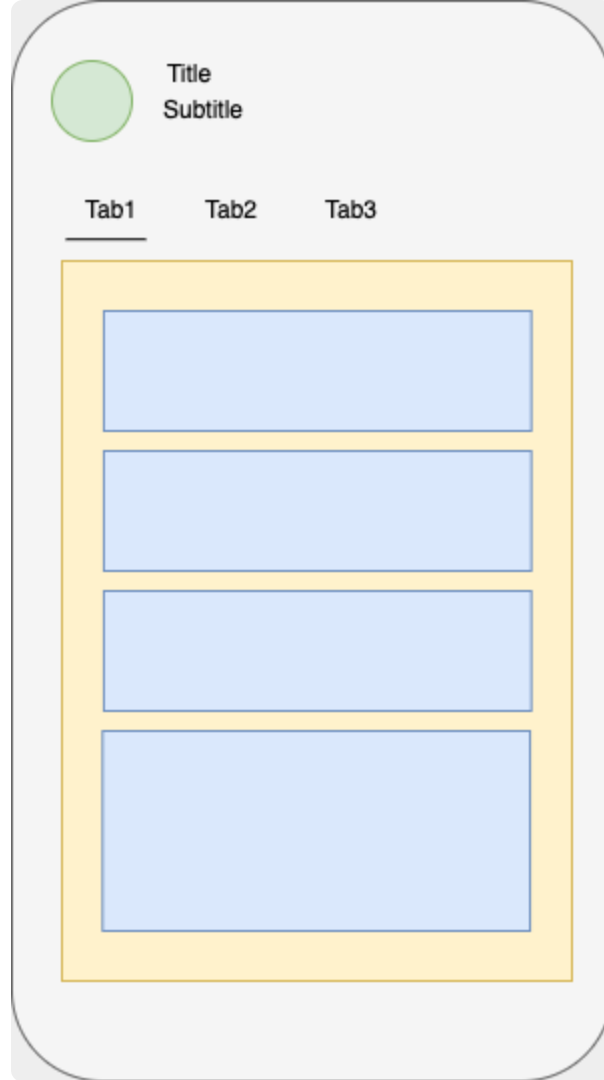
因此，我们可以再次针对业务代码包进行拆分，**将一个业务代码包拆分为一个主包和多个子包的方式**。在进入页面后优先请求主包的 JavaScript 代码资源，能够快速地渲染首屏页面，紧接着用户点击某一个模块时，再继续下载对应模块的代码包并进行渲染，就能再进一步减少加载时间。示例如图：



那么，什么时候需要把业务代码包拆分成一个主包和多个子包呢？把什么模块作为主包，什么模块作为子包比较合适呢？我举一个简单的例子给你解释一下。

其实在简单的业务中，我们并不需要对业务代码包进行拆分，但是在交互较为复杂的页面中，可能需要进行拆包。下面我们看一下这个包含 Tab 的业务页面：





这个页面中包含三个 **Tab**，也就是 **Tab1**、**Tab2** 和 **Tab3**。如果这三个 **Tab** 中的内容相似，我们当然就不需要对业务代码包进行拆分了。但是如果这三个 **Tab** 中的内容差异化较大，页面模版完全不相同，我们就可以对业务代码包进行拆分。

比如三个 **Tab** 页面中，**A** 页面是列表布局，**B** 页面是瀑布流布局，**C** 页面是视频页面，这几个页面之间的布局、样式、方案均无法统一管理。我们就对这三个不同的页面进行拆分，当用户选择某一个页面时，加载对应页面的样式以及布局。

我们可以将头部 **title**、**subtitle** 部分以及三个 **tab** 作为主包优先进行渲染，其次 **Tab1**、**Tab2**、**Tab3** 部分再分别打成子包，然后再根据用户选中的 **Tab**，将对应的代码包下载下来并渲染。这样我们就可以减少每次下载的代码包的大小，加快渲染速度

不过，在 **React Native** 移动端的性能优化中，除了 **React Native** 环境创建、**bundle** 文件、接口数据等方面的优化外，还有一个大的优化点，就是 **React Native 运行时优化**。

React Native 旧版本的运行效率有两大痛点：一是 JSC 引擎解释执行 JavaScript 代码效率低，引擎启动速度慢；二是 JavaScript 与 Native 通信效率低，特别是涉及批量地 UI 交互，如列表时更是如此。

针对于第二点，在 [🔗《自定义组件：如何满足业务的个性化需求？》](#) 中，我们讲解了 React Native 新架构采用了 JSI 进行通信，替换了 JSBridge，无异步地序列化与反序列化操作、无内存拷贝，可以做到同步通信。

而除了 JSI 之外，React Native 0.60 以后的版本开始支持 [🔗Hermes 引擎](#)。对比 JSC 引擎，Hermes 引擎在启动速度、代码执行效率上都有大幅提升，所以接下来我们就来重点讲解 Hermes 引擎的特点、它的优化手段以及如何在移动端启用。

## Hermes 引擎

Facebook 在 ChainReact 2019 大会上正式推出了新一代 JavaScript 执行引擎 Hermes。Hermes 是个轻量级的 JavaScript 引擎，专门对移动端上运行 ReactNative 进行了优化，Hermes 可执行字节码，也可以执行 JavaScript。



在分析性能数据时，Facebook 团队发现 JavaScript 引擎是影响启动性能和应用包体积的重要因素。JavaScriptCore 最初是为桌面浏览器端设计，相较于桌面端，移动端能力有太多的限制。所以，为了能从底层对移动端进行性能优化，Facebook 团队选择自建 JavaScript 引擎，设计了 Hermes。

那新设计的 Hermes 引擎能带来怎样的提升呢？Chain React 大会上官方给出了 Hermes 引擎一组数据：

- 从页面启动到用户可操作的时间长短（Time To Interact: TTI），从 4.3s 减少到 2.01s；
- App 的下载大小，从 41MB 减少到 22MB；
- 内存占用，从 185MB 减少到 136MB。

Hermes 的优化主要体现在**字节码预编译**和**放弃 JIT**这两点上。

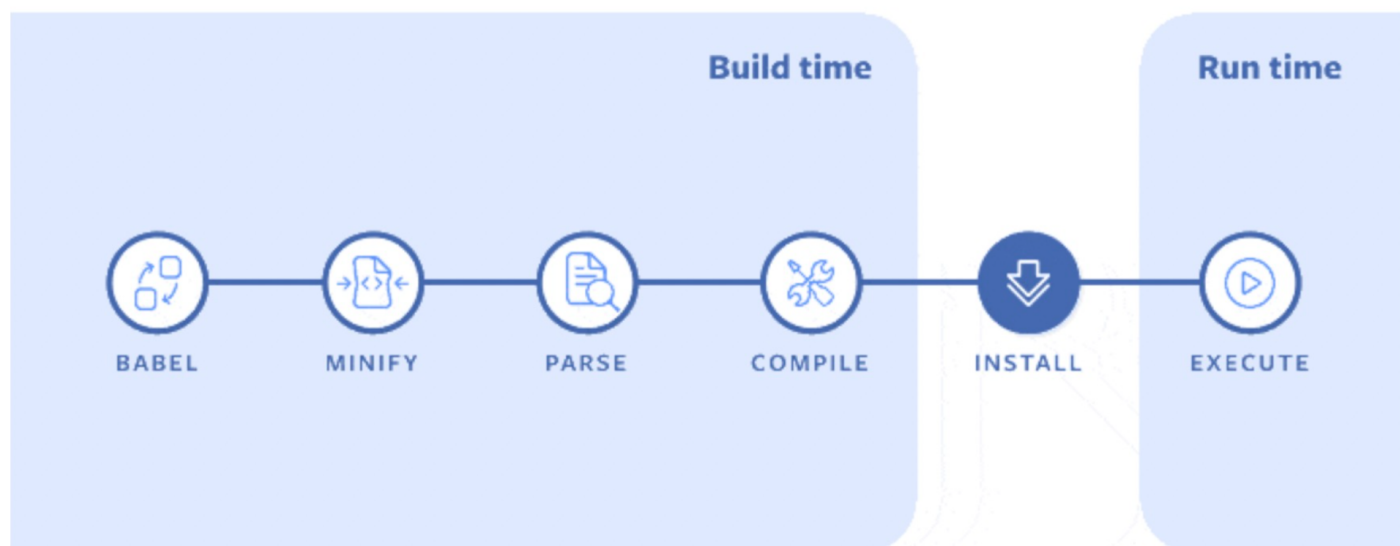
首先来看下字节码预编译。现代主流的 JavaScript 引擎执行一段 JavaScript 代码的大概流程是：

```
1 读取源码文件 -> 解析转换成字节码 -> 执行
```

 复制代码

不过，在运行时解析源码转换字节码是一种时间浪费，所以 Hermes 选择预编译的方式在编译期间生成字节码。这样做，一方面避免了不必要的转换时间；另一方面，多出的时间可以用来优化字节码，从而提高执行效率。相关示意图如下：

# Bytecode precompilation with Hermes



第二点是放弃了 JIT。为了加快执行效率，现在主流的 JavaScript 引擎都会使用一个 JIT 编译器，在运行时通过转换成机器码的方式优化 JavaScript 代码。Facebook 团队认为 JIT 编译器主要有两个问题：

- 要在启动时候预热，对启动时间有影响；
- 会增加引擎 size 大小和运行时内存消耗。

但是这里需要注意，放弃了 JIT，纯文本 JavaScript 代码执行效率会降低。放弃 JIT，是指放弃运行时 Hermes 引擎对纯文本 JavaScript 代码的编译优化。

当然了，Hermes 也会带来一些问题，首先就是 Hermes 编译的字节码文件比纯文本 JavaScript 文件增大不少，第二点就是执行纯文本 JavaScript 耗时长。

那么，我们要如何开启 Hermes 呢？除了可以参考[官方文档](#)快速开启 Hermes，下面我也会给你介绍如何在混合工程中开启 Hermes 引擎，我们以 Android 为例进行讲解。

第一步，获取 hermes.aar 文件（node\_modules/hermes-engine）：

▼ hermes-engine

▼ android

▶ include

hermes-cppruntime-debug.aar

hermes-cppruntime-release.aar

hermes-debug.aar

hermes-release.aar

▶ linux64-bin

▶ osx-bin

▶ win64-bin

package.json

README.md

第二步，将 hermes-cppruntime-release.aar 与 hermes-release.aar 放到工程的 libs 目录总，然后在模块的 build.gradle 中添加依赖，这两个 aar 中主要是 hermes 和 libc++\_shared so 文件：

复制代码

```
1 dependencies {  
2     implementation(name: 'hermes-cppruntime-release', ext: 'aar')  
3     implementation(name: 'hermes-release', ext: 'aar')  
4 }
```

第三步，设置 JavaScript 引擎：

复制代码

```
1 ReactInstanceManagerBuilder builder = ReactInstanceManager.builder()  
2     .setApplication((Application) context.getApplicationContext())  
3     .addPackage(new MainReactPackage())  
4     .setRedBoxHandler(mExceptionHandler)  
5     .setUseDeveloperSupport(RNDebugSwitcher.getInstance().isDebug())  
6     .setInitialLifecycleState(LifecycleState.BEFORE_CREATE)  
7     .setJavaScriptExecutorFactory(new HermesExecutorFactory()); // 设置为 hermes
```

最后，运行 hermes 编译出的字节码 bundle 文件就可以了。这一步又分为了几个小步骤，你参照下面步骤即可：

- 将 JavaScript 打包成 bundle 文件。

 复制代码

```
1 react-native bundle --platform android --entry-file index.android.js
2 --bundle-output ./bundles/index.android.bundle --assets-dest ./bundles
3 --dev false
```

- 使用 hermes-engine 将 bundle 文件转换成字节码文件。下载 hermes-engine，使用 hermes 命名进行转换。

 复制代码

```
1 ./hermes -emit-binary -out index.android.bundle.hbc
2 xxx/react-native/app/bundles/index.android.bundle
```

- 重命名 bundle 文件。

这里要将之前 bundle 目录下的 index.android.bundle 删掉，将当前的 index.android.bundle.hbc 重命名为 index.android.bundle

讲完了 Hermes 引擎，我们最后再来了解下引擎的复用优化。Hermes 引擎是运行时执行效率的优化，而引擎复用是 React Native 创建引擎成本的优化。

## 引擎复用

在混合应用中，React Native 由应用级的使用变更为页面级，每一个页面都使用一个 React Native 引擎 (包括 JSC/Hermes、Bridge/JSI)，除了内存占用高以外，React Native 引擎的创建耗时也是比较严重的。前面我们讲了环境预创建，就是对于引擎创建成本的优化。在这一块儿，除了预创建外，我们还可以进行**引擎复用优化**。

以 Android 为例，React Native 引擎的直接表现就是 ReactInstanceManager，内部会初始化 React Native 相关的环境。而在混合应用中，一般会配合热更新策略进行页面加载，所以使用的是 JSC/Hermes 动态加载脚本的能力。从这个场景来看，似乎一个引擎可以运行不同的 bundle 文件，即可达到复用的目的。

但是引擎复用的坑也非常多，目前我们并未直接落地使用：



1. 创建和复用引擎的成本可能会导致不少页面，第一次进入和后续进入的速度，表现不一致，因此这类体验问题还需要专项排查并优化；
2. 在多页面同时在前台的状态下，比如首页 TAB 不同页面使用的都是 **React Native** 页面，会存在莫名的同步问题；
3. 复用 **React Native** 容器内容时，会保持上一次会话的全局变量，容易造成业务逻辑错误。同一个引擎加载不同 **bundle**，**JavaScript** 上下文与新加载进去的代码能否实现 **100%** 隔离无污染可能是未知数。同时多页面 **JavaScript** 上下文隔离。目前引起复用的一大坑其实来源于 **JavaScript** 上下文多个页面混在一起，容易出错；
4. **JSC/Hermes** 随时有可能发生不可逆转的异常，因此引擎维护的过程中异常状态识别也是一个问题。

如果你有什么好的解决思路 and 想法，也欢迎在评论区留言，我们一起讨论。

## 总结

今天我们学习了如何在客户端将 **React Native** 性能优化到极致，包括环境预创建、异步更新、接口预缓存、拆包、按需加载、**Hermes** 引擎、引擎复用等。这些手段在实际业务中非常实用，当然 **React Native** 框架也在从自身上不断优化、迭代，追求性能的更高水平。

接下来我们回顾下今天讲过的几个重点：

- 在优化 **React Native** 环境创建的耗时方面，我们可以使用环境预创建和引擎复用的方式进行优化。环境预创建更容易落地，而且坑更少。引擎复用在内存占用这块比环境预创建方式好，但是需要解决的问题更多；
- 在热更新流程优化上，我们可以使用异步更新和预加载 **bundle** 的方式，优先使用 **bundle** 缓存进行加载，同时 **JavaScript** 业务可控制新版本更新策略；
- 另外，如何在初始化 -> 热更新 -> **bundle** 下载 -> 加载 **JavaScript** -> **JavaScript** 业务接口请求的链路中，利用客户端多线程的优势，接口预缓存是不错的选择。
- 如果 **bundle** 过大，你可以拆分 **common** 包和业务包。为进一步提高加载速度，你还可以利用 **JavaScript** 引擎动态加载脚本的能力，按需加载子 **bundle**。
- 最后，你也可以多关注最新的 **Hermes** 引擎，看看它的优缺点，以及它是如何实现优化的。

至此，Native 相关的三讲就告一段落了，后续我和惠妹还会参与 React Native 新框架原理篇的编写。


## 作业

1. 运行 Hermes 引擎 demo，并实现环境预创建功能。

如果有什么问题，欢迎在评论区留言，咱们下一讲见。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 25 | 性能优化：如何设计一个合适的性能优化方案？

## 精选留言 (1)

 写留言



Saigō

2022-05-27

干货多多

