

24 | 拆包：Metro 拆包的工作原理是什么？

2022-05-23 蒋宏伟

《React Native 新架构实战课》

课程介绍 >



讲述：蒋宏伟

时长 19:13 大小 17.61M



你好，我是蒋宏伟。

在上一讲我们聊了热更新，今天这一讲我们聊聊热更新中的拆包环节。

热更新和拆包都是大家聊得比较多的话题，通常一个聊得比较多的技术话题都会有一套成熟的技术方案，比如热更新平台就有 **CodePush** 这样的成熟方案，但拆包却没有一套大家都公认成熟的方案，这也是很奇怪。

业内没有方案，换作以前大家就只能自己“造轮子”了，但去年我们开源了 **58RN** 的拆包工具 [🔗 metro-code-split](#)，你可以直接拿去用。

metro-code-split 是我们团队的赵琦同学写的一个基于 **Metro** 的拆包工具，它能帮你实现 **React Native** 的拆包工作。

要知道，无论我们是运行 `npm start` 启动本地调试，还是运行 `npm run-ios` 构建应用，底层都会用到 **Metro** 打包工具。

然而，**Facebook** 开源的 **Metro** 打包工具，本身并没有拆包功能。它只能将 **JavaScript** 代码，打包成一个 **Bundle** 文件，而且 **Metro** 也不支持第三方插件，所以社区也没有第三方拆包插件。但当初，我们在阅读 **Metro** 源码的时候，发现了一个可配置的函数 [🔗 customSerializer](#)，从而找到了不入侵 **Metro** 源码，通过 [🔗 配置的方式](#) 给 **Metro** 写第三方插件的方法。

有了 **Metro** 的 `customSerializer` 方法后，现在我们可以给 **Metro** 来写插件了，通过插件来提供单独拆包能力。`metro-code-split` 就是我们为 **Metro** 写的拆包插件。下面我们就来分析下这个插件的原理和使用。

拆包原理

我们开源的 `metro-code-split` 拆包工具，是**基于模块**来拆包的，它相对于基于文本的拆包方式，加载速度会更快一些。

为什么基于模块的拆包，要比基于文本的拆包加载速度更快一些呢？这是因为，基于模块的拆包方式能够独立运行。

在 2018 之前，我们使用的是文本拆包方案，在 2018 之后，我们改成了模块拆包方案。因为拆出来的包能够独立运行，即便热更新文件没有下载完成，内置在 **App** 中的公共文件也能提前运行。理论上，模块拆包配合框架初始化预执行，在 **Android** 上实测平均能快 310 ms，而且还支持文件的再次拆分，实现类似 **Web** 中的 **Dynamic Import** 的效果。

那为什么基于模块的拆包方式，能够独立运行，而基于文本的拆包方式不能独立运行呢？

我们先来看基于文本的拆包方式。假设我们采用的是多 **Bundle** 的基于文本的拆包方式。多个 **Bundle** 之间的公共代码部分是“**react**”和“**react-native**”库，这里我用 `console.log(“react”)`、`console.log(“react-native”)` 来代替。多个 **Bundle** 之间不同的代码部分是业务代码，这里用 `console.log(“Foo”)`，来代替某个具体业务代码。

基于文本的拆包，我们以前采用的是 **Google** 开源的 [🔗 diff-match-patch](#) 算法，它也提供了在线计算网站，你可以自己把玩一下，我也把计算热更新包的示意图放在了下面：

Old Version: 内置包

```
console.log("react")
console.log("react-native")
```

New Version: 完整 Bundle

```
console.log("react")
console.log("react-native")
console.log("Foo")
```

Compute Patch

Patch: 热更新包

```
@@ -41,8 +41,27 @@
    native%22)
+ %0Aconsole.log(%22Foo%22)
```

在基于文本计算热更新包的示意图中，我们会把 Old Version 的字符串文件进行内置。这部分代码除了升级 React Native 版本之外不会轻易改动。而 New Version 的字符串是本次热更新的目标代码，也就是完整的 Bundle 文件，但开发者并不需要下载完整的 Bundle 文件，因为 Old Version 已经内置到 App 中了，我们只需要下发 Patch 热更新包即可。

客户端接收到 Patch 热更新包后，会和 Old Version 代表的内置包进行合并，合并的结果就是 New Version 代表的完整 Bundle。

以上这些，就是基于文本拆包与合包的原理。但你可以看到，**Patch 热更新包是一段记录修改位置、修改内容的文本，而不是可独立执行的代码**，会导致内置包没法提前执行，只能等到下载完成，再完成合并，生成完整的 Bundle 文件，最后才能整体执行。这就是为什么基于文本拆包方式不可独立执行的原因。

但基于模块的拆包方式，内置包和热更新包就可以分别独立执行。

同样，还是以多 Bundle 模式的 Foo 业务热更新为例，我也弄了基于模块拆包示意图，你可以对比一下基于文本和基于模块拆包的区别：

内置包

```
console.log("react")  
console.log("react-native")
```

完整 Bundle

```
console.log("react")  
console.log("react-native")  
console.log("Foo")
```

模块拆包

热更新包

```
console.log("Foo")
```

你可以看到，基于模块拆包方案，拆出来的热更新包是代表 **Foo** 业务代码的 `console.log("Foo")`，这部分代码是可以独立运行的。

因此，你可以在客户端先运行内置包，然后并行下载热更新包，等热更新包下载完成，再接着运行热更新包。

以上这种把包拆成两份，分两次执行的效果，在功能上是等效于一次性运行完整的 **Bundle** 包的，但在性能上会更快一些，因为内置包提前运行了。而且它还能支持类似于 **Web** 一样的 **Dynamic Import** 功能，该功能还支持非首屏代码延迟加载，这又能进一步减少首屏代码的下载量，提高首屏代码的执行速度。

因此，我们开源的 **metro-code-split** 选用的是基于模块的拆包方案。

metro-code-split 的配置

那 **metro-code-split** 具体如何使用呢？我们先来看它的基础配置方法。

首先，在你的项目中安装 **metro-code-split** 插件：

```
1 npm i metro-code-split -D | yarn add metro-code-split -D
```

 复制代码

第二步是修改你的 `package.json` 文件：

 复制代码

```
1 "scripts": {  
2   "start": "mcs-scripts start -p 8081",  
3   "build:dllJson": "mcs-scripts build -t dllJson -od public/dll",  
4   "build:dll": "mcs-scripts build -t dll -od public/dll",  
5   "build": "mcs-scripts build -t busine -e index.js"  
6 }
```

其命令含义如下：

- **start**：启动本地调试服务；
- **build:dllJson**：构建公共包的模块文件；
- **build:dll**：构建公共包；
- **build**：构建业务包和按需加载包。

关于公共包、公共包的模块文件、业务包和按需加载包这四类包的区别，后面我们还有更详细的介绍。

第三步是修改 `metro.config.js` 配置：

 复制代码

```
1 const Mcs = require('metro-code-split')  
2  
3 // 拆包的配置  
4 const mcs = new Mcs({  
5   output: {  
6     // 配置你的 CDN 的 BaseURL  
7     publicPath: 'https://static001.geekbang.org/resource/rn',  
8   },  
9   dll: {  
10    entry: ['react-native', 'react'], // 要内置的 npm 库  
11    referenceDir: './public/dll', // 拆包的路径  
12  },  
13  dynamicImports: {}, // dynamic import 是默认开启的  
14 })  
15  
16 // 业务的 metro 配置  
17 const busineConfig = {  
18   transformer: {
```

```

19     getTransformOptions: async () => ({
20       transform: {
21         experimentalImportSupport: false,
22         inlineRequires: true,
23       },
24     })),
25   },
26 }
27
28 // Dynamic Import 在本地和线上环境的实现是不同的
29 module.exports = process.env.NODE_ENV === 'production' ? mcs.mergeTo(busineConf

```

这里有两个拆包的参数需要你注意。第一个是 **publicPath**，它是用于配置线上环境中，按需加载包的根路径的。

比如，你的热更新包的地址为：

 复制代码

```
1 https://static001.geekbang.org/resource/rn/a1dc...d055.bundle
```

那么 **publicPath** 就应该配置为：

 复制代码

```
1 https://static001.geekbang.org/resource/rn
```

而包名 **a1dc...d055.bundle** 是通过 **Bundle** 内容的 MD5 值生成的。

第二个要注意的参数是 **dll**，它用于配置需要内置 **npm** 库。

通常在一个混合开发的 **React Native** 应用中，“**react**”和“**react-native**”这两个包基本上不会变动，所以你可以把这两个 **npm** 库拆到一个公共包中，这个公共包只能跟随 **App** 发版更新。而其他的业务代码或者第三方库，比如“**reanimated**”，这些代码变动相对频繁，就可以都跟着进行业务包进行集成，方便动态更新。

如果是你维护的是一个纯 **React Native** 应用，或者是一个小型的 **React Native** 应用，那么你还可以多内置一些 **npm** 库。你可以按一个发版周期来估计哪些需要内置，哪些不需要内置。

比如，某个第三方库在一个发版周期内完全没有改动的可能性，那么你就可以考虑内置，如果某个第三方库会进行改动，那么就走热更新流程。

完成以上三步后，你 `metro-code-split` 拆包插件也就配置完成了。

metro-code-split 的使用

那么，配置完成 `metro-code-split` 之后，如何使用 `metro-code-split` 进行拆包呢？

`metro-code-split` 支持三类包的拆分，包括公共包、业务包和按需加载包。接下来，我按这三类拆包方式和你进行讲解。

我们先来看公共包。

当你在 `dll` 配置项中填写了“`react`”和“`react-native`”之后，每次打包时，“`react`”和“`react-native`”都会被当作公共包来处理：

```
1  dll: {
2    entry: ['react-native', 'react'], // 要内置的 npm 库
3    referenceDir: './public/dll', // 拆包的路径
4  },
```

 复制代码


你可以直接运行如下命令，把公共包拆出来：

```
1 $ yarn build:dll
```

 复制代码

运行完成后，你再查看 `public/dll` 目录，你会发现该目录下面多了两个文件，分别是 `_dll.android.bundle` 和 `_dll.ios.bundle`，这两个文件就是集成了“`react`”和“`react-native`”所有代码的公共包。

你也可以运行如下命令，查看公共包中包含的模块：

 复制代码


```
1 $ yarn build:dllJson
```

运行上述命令后，你可以找到 `_dll.android.json` 和 `_dll.ios.json` 两个文件，这两个包含了“react”和“react-native”依赖的所有模块，摘要示例如下：

 复制代码

```
1 [
2   "__prelude__", // 框架预制模块
3   "require-node_modules/react-native/Libraries/Core/InitializeCore.js", // react-native 的入口模块
4   "node_modules/@babel/runtime/helpers/createClass.js", // babel 的类模块
5   "node_modules/react-native/index.js", // react-native 入口模块
6   "node_modules/metro-runtime/src/polyfills/require.js", // require 运行时模块
7   "node_modules/react/index.js" // react 模块
8 ]
```

`_dll.json` 记录了所有的公共模块，`_dll.bundle` 包含所有公共模块代码，比如管理 React Native 全局变量的框架预制模块 `__prelude__`、管理初始化的 `InitializeCore` 模块、管理 `babel`、`require` 的模块，以及 `react` 和 `react-native` 框架的入口模块。

一般情况下，你不需要对 `_dll.json` 文件进行任何处理，你只需关注 `_dll.bundle` 文件即可，`_dll.bundle` 文件才是内置到客户端的公共包。

然后我们再看业务包和按需加载包。

当你拿到内置包后，除了“react”和“react-native”的内置代码以外，其他所有代码都归属于业务包，但有一类文件例外，就是按需加载模块。

不过因为业务包和按需加载包的耦合性很强，按需加载包没办法脱离业务包进行独立打包，所以接下来我会把业务包和按需加载包一起介绍。

通常，你引入普通业务模块，使用的是 `import * from "xxx"`，那么该模块的代码都会直接打到业务包中。但在引入按需加载业务模块时，使用的是 `import("xxx")` 引入的，那么该模块代码会直接打到按需加载包中。

示例代码如下：


```

1 import React, {lazy, Suspense} from 'react';
2 import {
3   Text,
4 } from 'react-native';
5 import {NavigationContainer} from '@react-navigation/native';
6 import {
7   createNativeStackNavigator,
8 } from '@react-navigation/native-stack';
9 import {Views, RootStackParamList} from './types';
10 import Main from './component/Main';
11
12 const Stack = createNativeStackNavigator<RootStackParamList>();
13
14 const Foo = lazy(() => import('./component/Foo'));
15 const Bar = lazy(() => import('./component/Bar'));
16
17 export default function App() {
18   return (
19     <Suspense fallback={<Text>Loading...</Text>}>
20       <NavigationContainer>
21         <Stack.Navigator initialRouteName={Views.Main}>
22           <Stack.Screen name={Views.Main} component={Main} />
23           <Stack.Screen name={Views.Foo} component={Foo} />
24           <Stack.Screen name={Views.Bar} component={Bar} />
25         </Stack.Navigator>
26       </NavigationContainer>
27     </Suspense>
28   );
29 }

```

你可以看到，Main 组件是通过 `import * from "xxx"` 引入的，它属于普通的业务模块；而 Foo 组件和 Bar 组件是通过 `import("xxx")` 引入的，它们属于按需加载的业务模块。

简单地讲，所有通过 `import * from "xxx"` 引入的普通业务模块都会打到业务包中，而使用 `import("xxx")` 引入的按需加载业务模块，每个模块都会打成一个独立的按需加载包。而且按需加载模块中的子模块的情况比较复杂，它们既有可能打在业务包中，也有可能打在按需加载包中，实际算法非常复杂。你可以参考我在 GMTC 中关于 Dynamic Import 的[分享](#)，这里就不展开介绍了。

当你完成代码的编写后，再使用如下命令，就可以生成业务包和按需加载包了：

```
1 $ yarn build
```

构建完成后，业务包和按需加载包会放在 `dist` 目录下，其中 `buz.android.bundle` 和 `buz.ios.bundle` 就是业务包，`chunks` 目录下以 MD5 值开头的包就是按需加载包：

 复制代码

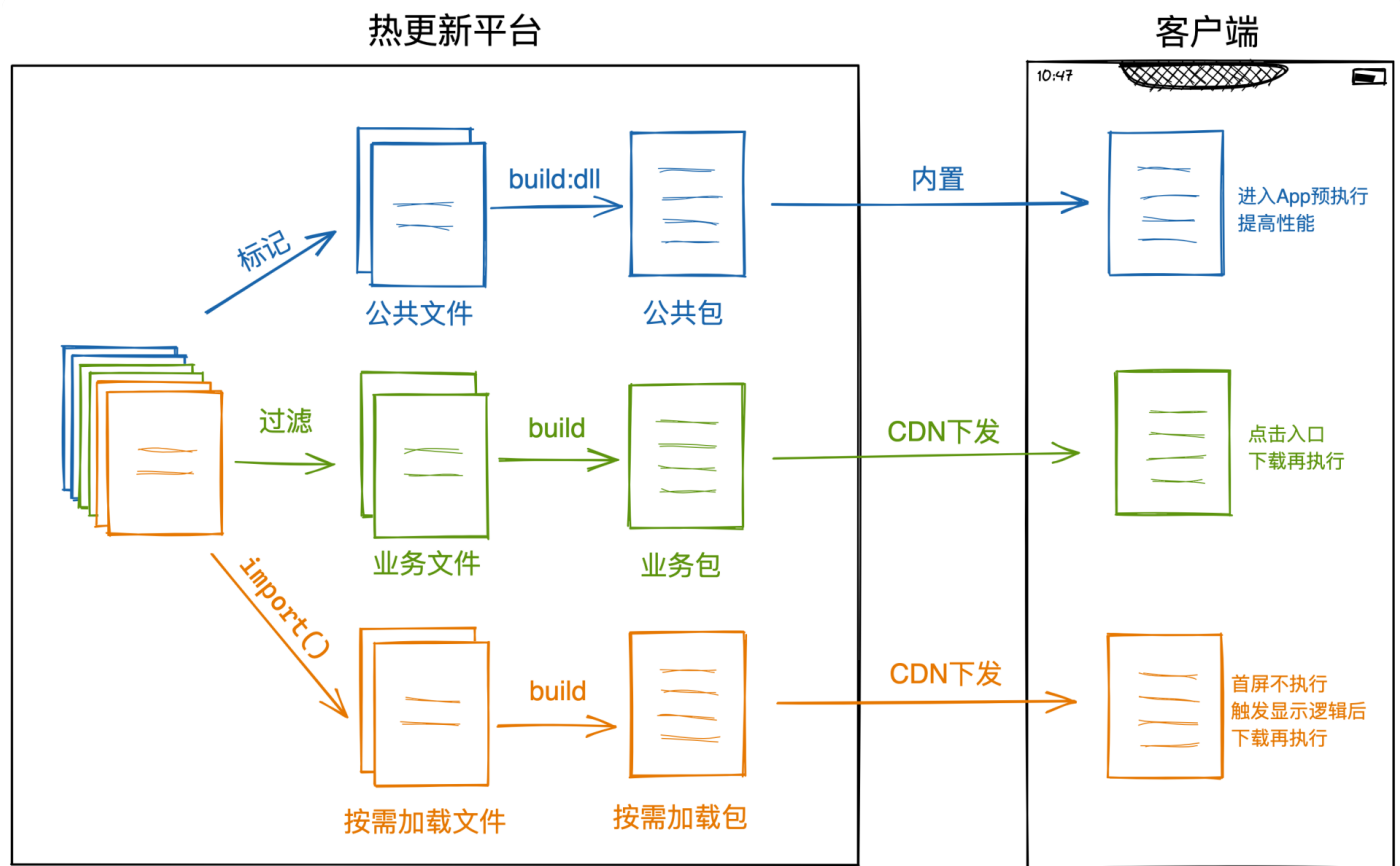
```
1 dist
2 |— buz.android.bundle
3 |— buz.ios.bundle
4 |— chunks
5   |— 22b3a0e5af84f7184abd.bundle
6   |— 479c3b2dc4e8fef12a34.bundle
```

通过 `yarn build:dll` 和 `yarn build`，我们就完成了公共包、业务包、按需加载包的构建。

热更新与拆包

那么，接下来的问题就是，如何将生成好的公共包、业务包、按需加载包与热更新流程配合一起使用呢？

我为你画了一张拆包方案的热更新示意图，你可以看下：



这张示意图，我们以自研热更新流程为例，简单介绍了这三类包的作用和使用方法。

因为我们采用的是模块拆包方案，虽然理论上每个包都是可以独立运行的，但实际上模块和模块之间是有依赖关系的，整体上讲，按需加载包会依赖业务包中的模块，业务包会依赖公共包中的模块。因此，需要先执行公共包、再执行业务包，最后执行按需加载包。

当然每个独立的按需加载包之间也会有依赖关系，不过这些加载的依赖关系，metro-code-split都已经帮你考虑到了，你直接用就行了。

对于首页是 **Native** 页面，而其他页面是 **React Native** 页面的多 **Bundle** 混合应用而言，整体加载流程如下：

首先，在启动 **App** 之后，找一个空闲时间，把 **React Native** “环境预创建” 好，然后把 “拆出来的公共包” 进行预加载。具体客户端实现，你可以参考后面众文和惠姝老师的《客户端优化》一讲中的相关讲解。

然后，在用户点击进入 **React Native** 页面时，在相关跳转协议中传入 **React Native** 页面的唯一标识符或者 **CDN** 地址，下载业务包并进行页面加载：

```
1 https://static001.geekbang.org/resource/rn/id999.buz.android.bundle
```

[📄 复制代码](#)

不过，对于一些复杂业务来说，页面内容会比较多，把一些非首屏的代码放在业务包中会拖慢首屏的加载速度，因此更好的方案是，把这些代码放在按需加载包中进行加载。当用户点击某个按钮或者下拉时，会再触发相关的按需加载逻辑。

此时，metro-code-split 会根据 `import('xxx')` 中的参数路径，找到对应的 CDN 地址，比如 `Foo.js` 模块对应的就是如下 CDN 地址：

```
1 // import("./Foo.js") 对应
2 https://static001.geekbang.org/resource/rn/03ad61906ed0e1ec92c2.bundle
```

[📄 复制代码](#)

然后，再根据该 CDN 地址请求按需加载包，并通过 `new Function(code)` 的方式执行下载回来的代码，把 `Foo` 组件加载到当前 JavaScript 的上下文中，并进行最终的渲染。

以上方案适合首页是 Native 页面的混合应用，如果首页也是 React Native 页面怎么办呢？

如果首页是 React Native 页面，而且采用的是多 Bundle 策略。

那么，公共包依旧需要内置，并且首页业务包也需要内置。此时，首页业务包采用静默更新策略，也就是当次下载、下次生效的策略。这样每次启动时首页，首页的业务包是从本地加载的，不走网络请求，首页的启动速度就会变快。其他页面的业务包或按需加载包继续采用，当次生效的动态下发形式进行更新。

当次生效的方式，大概多了 300ms~500ms 的 Bundle 下载时间，但带来的好处是业务能够随时更新、Bug 能够随时修复，不用等到用户下次进入页面再生效。

如果首页是 React Native 页面，但采用的是单 Bundle 策略，比如 CodePush。

那么，公共包和业务包需要分别内置，其中公共包走发版更新流程，业务包走 CodePush 静默更新流程。相对于纯 CodePush 方案，通过拆包的方式，能够节约 CodePush 更新的下载量体积。如果你还同时使用了按需加载包，那么还能节约非首屏代码的执行时间。

如果遇到紧急 Bug，CodePush 也支持当次生效。但由于 CodePush 底层机制的原理，它不仅需要下载热更新 Bundle，还需要重新加载整个 JavaScript 环境，耗时比较长，因此不建议你把它用作默认的更新方式。

今天的 Demo 我也放到了 [🔗 GitHub](#) 上，你可参考一下。

总结

今天我们介绍了 React Native 热更新流程中，一种最常见的性能优化手段，拆包。

我们的开源拆包工具 metro-code-split 能够很方便地帮你把整个 Bundle 包拆分成公共包、业务包和按需加载包。你只需要下载、配置和执行命令，就可以完成拆包操作了。

本地拆包只是热更新流程中的一个环节，因此你需要配合你的热更新流程一起使用。根据业务的不同，应用可大致分为三种形态，包括单 Bundle 的纯 React Native 应用、多 Bundle 的纯 React Native 以及多 Bundle 的混合应用，每种不同的形态的应用采用的热更新方式和拆包策略都有所区别，你需要结合具体的场景进行分析。

虽然使用 metro-code-split 进行拆包很简单，但要实现 metro-code-split 并不容易，在编译时、运行时有大量的工作需要处理，你还得把所有模块的正向依赖、逆向依赖给理清楚，才能合理的进行拆包。幸运地是，在 Web 领域的 Webpack 其实已经给出了 Web 领域的“拆包”解决方案，开源 metro-code-split 也算是我们回馈社区的一种形式。

希望我们开源的 metro-code-split 你能喜欢。

作业

1. 请你参考 metro-code-split 的 [🔗 示例](#)，完成在本地完成拆包和按需加载的调试。

如果遇到了什么问题，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

© 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 23 | 热更新：如何搭建一个热更新平台？

下一篇 25 | 性能优化：如何设计一个合适的性能优化方案？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。