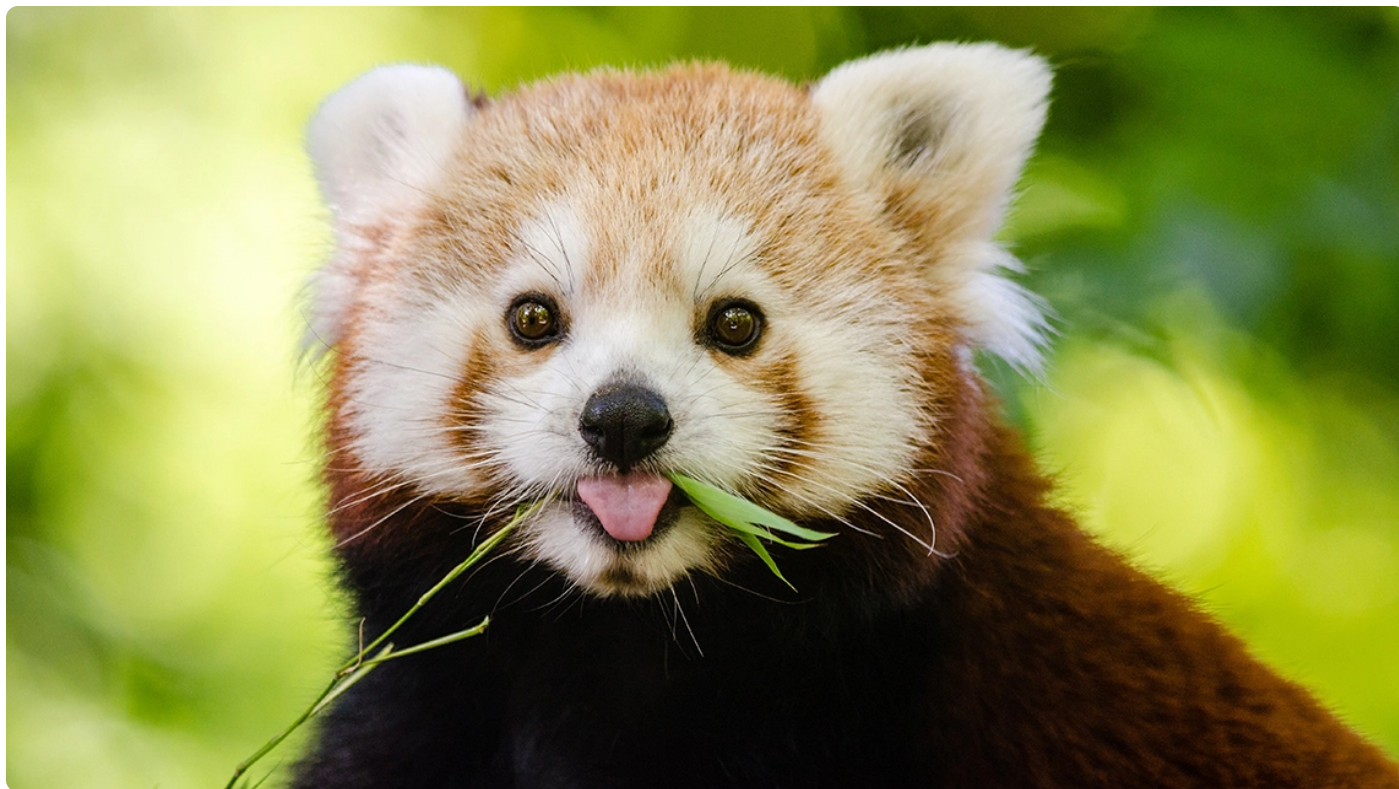


## 05 | Image: 选择适合你的图片加载方式

2022-04-06 蒋宏伟

《React Native 新架构实战课》

[课程介绍 >](#)



讲述：蒋宏伟

时长 29:20 大小 26.87M



你好，我是蒋宏伟。

今天我们来讲解 React Native 框架中的 Image 组件。顾名思义，图片组件 Image 就是用来加载和展示图片的。

你可能会觉得，图片组件的基础用法非常简单呀，学起来也很容易上手，这有什么好讲的呢？没错，正因为它很简单，有时候，我们可能会忽视对这些基础知识的琢磨。在日常开发中，图片是影响用户体验的关键因素之一，它很常见，基本上哪里都有它。而且相对于文字，图片也更容易抓住用户的眼球。图片组件很重要，但要用好却不那么容易。

React Native 的 Image 组件一共支持 4 种加载图片的方法：

- 静态图片资源；

- 网络图片；
- 宿主应用图片；
- Base64 图片。

这 4 种方案给我们业务提供了更灵活的选择空间，但同时也让不少同学犯了选择困难症，不同情况下我该怎么选呢？今天，我们就来深度剖析这 4 种方案分别的适用场景是什么，并给你介绍一下我推荐的最佳实践。

## 静态图片资源

静态图片资源（Static Image Resources）是一种**使用内置图片**的方法。静态图片资源中的“静态”指的是每次访问时都不会变化的图片资源。站在用户的视角看，App 的 logo 图片就是不会变化的静态图片资源，而每次访问新闻网站的新闻配图就是动态变化的图片。

如果图片每次都不会变化，那么你就可以把这张图片作为静态图片资源，内置在 App 中。这样，用户在打开你的 App 时，图片是从本地直接读取的，直接读取图片的速度比走网络请求先下载再加载的速度要快上很多。一张网络图片从下载到展示的耗时通常需要 100ms 以上，而一张内置图片从读取到展示的耗时通常只有几 ms，甚至更低，二者耗时相差了两个数量级。

因此，在一些高性能场景下，你应该选择把这些不经常变动的静态图片资源内置到 App 中。当用户打开 App 时，这些图片就能够立刻展示出来了。

那我们具体是怎么使用静态图片资源的呢？这里我们可以分为 3 步。首先，把图片放到 React Native 的代码仓库中，然后通过 `require` 的方式引入图片，最后把图片的引用值传给 `source` 属性。`Image.source` 属性是用来设置图片加载来源的。

这里我们需要注意的是，`require` 函数的入参必须是字面常量，而不能是变量。你可以看下这段代码：

```
1 // 方案一：正确
2 const dianxinIcon = require('./dianxin.jpg')
3 <Image source={dianxinIcon}/>
4
5 // 方案二：错误
```

 复制代码

```
6 const path = './dianxin.jpg'
7 const dianxinIcon = require(path)
8 <Image source={dianxinIcon}/>
```

在这段代码中，方案一是静态图片资源正确的使用方式，方案二是错误的。方案一用的是图片相对路径的字面常量，也就是 `'./dianxin.jpg'`。而方案二，用的是图片相对路径的变量，也就是 `path`。

你是不是很好奇，为什么使用 `require` 函数引入静态图片资源时，`require` 入参，也就是图片的相对路径，必须用字面常量表示，而不能用变量表示？静态图片资源的加载原理又是什么呢？我们接下来继续分析。

## 静态图片资源的加载原理

我们还是用加载点心图片（`dianxin.jpg`）为例，从编译时到运行时，剖析加载静态资源图片的全过程，一共分为三步。

**第一步编译：**在编译过程中，图片资源本身是独立于代码文件之外的文件，图片资源本身是不能编译到代码中的，所以，我们需要把图片资源的路径、宽高、格式等信息记录到代码中，方便后面能从代码中读取到图片。

你可以选一张你喜欢吃的点心的图片，命名为 `dianxin.jpg`，并把点心图片和 `index.js` 文件放在同一层级目录下。然后在 `index.js` 中通过 `require` 方法把点心图片引入进来，交由 `Image` 组件使用。

我在这一讲末尾的补充材料中，为你留了示例代码，课后你可以参考示例代码，一步步地自己动手操作。现在，你应该把注意力放在理解静态图片资源的编译、构建和加载的流程上，具体的执行细节可以后续再了解。

在你引入静态图片资源完成后，可以先本地试试图片是否能正常展示。如果展示没有问题，直接运行 `react-native bundle` 的打包命令，开始打包编译：

 复制代码

```
1 npx react-native bundle --entry-file index.tsx --dev false --minify false --bun
```

这段打包（**bundle**）命令的意思是，以根目录的 **index.tsx** 文件为入口（**entry file**），产出 **release**（**dev=false**）环境的包，这个包不用压缩（**minify=false**），并将这个包命名为 **./build/index.bundle**，同时将静态资源编译产物放到 **./build** 目录。这个 **build** 目录结构如下：

复制代码

```
1 ./build
2 |— assets
3 |   |— src
4 |       |— Lesson3Image
5 |           |— dianxin.jpg
6 |— index.bundle
```

编译后的产物会都存在 **build** 目录中，这个目录需要你提前创建好，否则会有报错提示。编译完成后，你可以在 **build** 目录中找到 **index.bundle** 文件，它是编译后的 **JavaScript** 代码。另外，**build** 目录中还有一个 **assets** 目录，**assets** 目录放的是编译后的图片 **dianxin.jpg**。

然后我们再打开 **index.bundle** 文件，搜索 **dianxin** 关键字。我们可以找到一个和 **dianxin** 关键字相关的独立模块，这个模块的作用就是将静态图片资源的路径、宽高、格式等信息，注册到一个全局管理静态图片资源中心。这个独立模块的代码如下：

复制代码

```
1 module.exports = _$$_REQUIRE(_dependencyMap[0]).registerAsset({
2   "__packager_asset": true,
3   "httpServerLocation": "/assets/src/Lesson3Image",
4   "width": 190,
5   "height": 190,
6   "scales": [1],
7   "hash": "0d4ac32eb69529cf90a7b248fee00592",
8   "name": "dianxin",
9   "type": "jpg"
10 });
```

它主要包括该图片的注册函数 **registerAsset** 和其注册信息。其中图片的注册信息包括，目录信息（**/assets/src/Lesson3Image**）、宽高信息（**width** 和 **height**）、图片哈希值（**hash**）、图片名字（**dianxin**）、图片格式（**jpg**）等等。

很明显，这个静态图片资源的注册函数和相关的图片信息代码，并不是你写的。那这段代码是怎么来的呢？它是由打包工具根据字面常量 **'./dianxin.jpg'**，找到真正的点心静态图片资

源后，读取图片信息自动生成的。

这里敲一下黑板，在使用 `require` 函数引入静态图片资源时，图片的相对路径必须用字面常量表示的原因是，**字面常量**`'./dianxin.jpg'` 提供的是一个直接的明确的图片相对路径，打包工具很容易根据字面常量`'./dianxin.jpg'` 找到真正的图片，提取图片信息。而变量`path` 提供的是一个间接的可变化的图片路径，你光看`require(path)` 这段代码是不知道真正的图片放在哪的，打包工具也一样，更别提自动提取图片信息了。

这里还需要注意一下，我们第一步“编译时”生成的图片注册函数和其注册的信息，我们在后面的第三步“运行时”还会用到。

**第二步构建：**编译后的 `Bundle` 和静态图片资源，会在构建时内置到 `App` 中。

如果你搭建的是 `iOS` 原生环境，那么你应该运行 `react-native run-ios` 构建 `iOS` 应用。如果你搭建的是 `Android` 原生环境，那么你应该运行 `react-native run-android` 构建 `Android` 应用。

不过，默认构建的是调试包，而我们想要的是正式包，因此我们还需要在命令后面加一行配置 `--configuration Release`。这样就能在你的真机或者模拟器上，构建出一个 `React Native` 应用了，具体命令如下：

```
1 $ npx react-native run-ios --configuration Release
```

 复制代码

在这一步，编译后的 `Bundle`，包括 `Bundle` 中的静态图片资源信息，和真正的静态图片资源都已经内置到 `App` 中了。现在你可以关闭网络，然后打开 `App` 试试，如果这时页面和图片依旧能正常展示，那就证明图片确实内置成功了。

实际上，上面的命令 `react-native run-ios` 既包括第一步的编译 `react-native bundle` 又包括第二步的构建。在真正编译和构建内置时候，你只需要运行 `react-native run-ios` 即可。

**第三步运行：**在运行时，`require` 引入的并不是静态图片资源本身，而是静态图片资源的信息。`Image` 元素要在获取到图片路径等信息后，才会按配置的规则加载和展示图片。



还记得吗？我们第一步“编译时”，生成了图片注册函数和其注册的信息，在第二步“构建时”，我们将真正图片内到了 **App** 中。那在第三步“运行时”，我们怎么拿到这些图片信息，并加载和展示真正的内置图片呢？

首先，你可以通过 `Image.resolveAssetSource` 方法来获取图片信息。具体的示例代码如下：

 复制代码

```
1 const dianxinIcon = require('./dianxin.jpg')
2
3 alert(JSON.stringify(Image.resolveAssetSource(dianxinIcon)))
4
5 // 弹出的信息如下：
6 {
7   "__packager_asset": true,
8   "httpServerLocation": "/assets/src/Lesson3Image",
9   "width": 190,
10  "height": 190,
11  "scales": [1],
12  "hash": "0d4ac32eb69529cf90a7b248fee00592",
13  "name": "dianxin",
14  "type": "jpg"
15 }
```

这段代码很简单，关键代码只有两行，第一行是通过 `require` 引入点心图片，并将其赋值给变量 `dianxinIcon`。第二行是通过调用 `Image.resolveAssetSource` 方法，并传入点心图片变量 `dianxinIcon`，获取我们在编译时生成的图片信息。你可以通过 `alert` 字符串的方式，将它打印在屏幕上，现在你就可以在运行时，看到编译时自动生成的静态图片资源的信息了。

在 `Image` 组件底层，使用的就是 `Image.resolveAssetSource` 来获取图片信息的，包括图片目录（`httpServerLocation`）、宽高信息（`width` 和 `height`）、图片哈希值（`hash`）、图片名字（`dianxin`）、图片格式（`jpg`），等等。然后，再根据这些图片信息，找到“构建时”内置在 **App** 中的静态图片资源，并将图片加载和显示的。这就是静态图片资源的加载原理。

正是因为静态图片资源加载方式，它在“编译时”提前获取了图片宽高等信息，在“构建时”内置了静态图片资源，因此在“运行时”，程序可以提前获取图片宽高和真正的图片资源。相对于我们后面要介绍的网络图片等加载方式，使用静态图片资源加载，即使不设置图片宽高，也有一个默认宽高来进行展示，而且加载速度更快。

## 网络图片

静态图片资源虽好，但它只适用于“静态不变的”图片资源，对于那些“动态变化的”和不方便内置的业务场景，那就要用到网络图片了。

网络图片（Network Images）指的是**使用 http/https 网络请求**加载远程图片的方式。

在使用网络图片时，我建议你**将宽高属性作为一个必填项**来处理。为什么呢？和前面介绍的静态图片资源不同的是，网络图片下载下来之前，**React Native** 是没法知道图片的宽高的，所以它只能用默认的 0 作为宽高。这个时候，如果你没有填写宽高属性，初始化默认宽高是 0，网络图片就展示不了。

具体的代码是这样：

 复制代码

```
1 // 建议
2 <Image source={{uri: 'https://reactjs.org/logo-og.png'}}
3     style={{width: 400, height: 400}} />
4
5 // 不建议
6 <Image source={{uri: 'https://reactjs.org/logo-og.png'}} />
```

## 缓存与预加载

不过，网络图片虽然指的是走网络请求下载的图片，但也并不用每次都走网络下载，只要有缓存就能直接从本地加载。所以这里我们也简单介绍一下 **React Native** 的缓存和预加载机制。

**React Native Android** 用的是 **Fresco** 第三方图片加载组件的缓存机制，**iOS** 用的是 **NSURLCache** 系统提供的缓存机制。

**Android** 和 **iOS** 的缓存设置方式和实现原理虽然有所不同，但整体上采用了内存和磁盘的综合缓存机制。第一次访问时，网络图片是先加载到内存中，然后再落盘存在磁盘中的。后续如果我们再次访问，图片就会从缓存中直接加载，除非超出了最大缓存的大小限制。

例如，**iOS** 的 **NSURLCache** 遵循的是 **HTTP** 的 **Cache-Control** 缓存策略，同时当 **CDN** 图片默认都已经设置了 **Cache-Control** 时，**iOS** 图片就是有缓存的。

而 `NSURLCache` 的默认最大内存缓存为 512kb，最大磁盘缓存为 10MB，如果缓存图片的体积超出了最大缓存的大小限制，那么一些老的缓存图片就会被删除。

图片缓存机制有什么用呢？

**通过图片缓存机制和预加载机制的配合，我们可以合理地利用缓存来提高图片加载速度，这能进一步地提升用户体验。**

使用图片预加载机制，可以提前把网络图片缓存到本地。对于用户来说，提前缓存的图片是第一次看到的，但对于系统缓存来说图片是第二次加载，它的加载速度是毫秒级的甚至亚秒级的。这就是预加载机制，提升图片加载性能的原理。

举个例子，你打算买个机械键盘，打开了个购物 App，滑动手机翻页选购，键盘图片和介绍都能马上地呈现出来。你没有感受丝毫的等待和卡顿，你可能就会直接下单买了。相反，如果你选购的过程中图片加载很慢，翻页还要等待很久，你就可能会考虑换个购物 App。

在这种无限滚动的长列表场景中，图片预加载就非常适合了。`React Native` 也提供了非常方便的图片预加载接口 `Image.prefetch`：

```
1 Image.prefetch(url);
```

 复制代码

也就是说，函数 `Image.prefetch` 接收一个参数 `url`，也就是图片的远程地址，函数调用后，`React Native` 会帮你在后台进行下载和缓存图片。这样，你下拉加载的图片时，网络图片是从本地缓存中加载的，就感受不到网络加载的耗时过程了。

## 宿主应用图片

宿主应用图片（`Images From Hybrid App's Resources`）指的是 `React Native` 使用 **Android/iOS 宿主应用的图片**进行加载的方式。在 `React Native` 和 `Android/iOS` 混合应用中，也就是一部分是原生代码开发，一部分是 `React Native` 代码开发的情况下，你可能会用到这种加载方式。

使用 `Android drawable` 或 `iOS asset` 文件目录中的图片资源时，我们可以直接通过统一资源名称 `URN`（`Uniform Resource Name`）进行加载。不过，使用 `Android asset` 文件目录中图片资



源时，我们需要在指定它的统一资源定位符 URL（Uniform Resource Locator）。

这里插个小知识，在 React Native 中，我们为什么要用 URI，比如 { uri: 'app\_icon' }，来代表图片，而不是用更常用的 URL，比如 { url: 'app\_icon' }，代表图片呢？

这是因为，URI 代表的含义更广泛，它既包括 URN 这种用名称代表图片的方式，也包括用 URL 这种地址代表图片的方式。以 iOS 和 Android 宿主图片为例，代码如下：

 复制代码

```
1 // Android drawable 文件目录
2 // iOS asset 文件目录
3 <Image source={{ uri: 'app_icon' }} />
4
5 // Android asset 文件目录
6 <Image source={{ uri: 'asset:/app_icon.png' }} />
```

你可以看到，iOS 宿主图片用的是图片名称 app\_icon，是 URN。而 Android 宿主图片用的是图片位置 asset:/app\_icon.png，是 URL。而 URI 的所代表的含义更广，既包括图片名称 URN，又包括图片位置 URL，所以 Image 组件的 source 属性中，代表图片名称或地址的键名是 URI。

在我国内，绝大多数的 React Native 应用都是混合应用，都是把 React Native 当做一个支持动态更新的跨端框架来使用的。那这种情况下，我们在 React Native 中直接用宿主应用图片资源不是更好吗？

你看，React Native 静态图片资源也是内置，Android/iOS 自身图片也要内置，搞一套图片管理机制不更简单一些嘛？而且部分图片还可以跨 React Native 和 Android/iOS 两个技术栈复用，减少一些 App 体积，这听起来很不错啊。

**但在实际工作中，我不推荐你在 React Native 中使用宿主应用图片资源。**首先，这种加载图片的方法没有任何的安全检查，一不小心就容易引起线上报错。第二，大多数 React Native 是动态更新的，最新代码是跨多个版本运行的，而 Native 应用是发版更新的，应用的最新代码只在最新版本运行，这就导致 React Native 需要确切知道 Native 图片到底内置在哪些版本中，才能安全地使用，这对图片管理要求太高了，实现起来太麻烦了。

最后，开发 **React Native** 的团队，和开发 **Android/iOS** 的团队很可能不是一个团队，甚至可能跨部门。复用的收益抵不上复用带来的安全风险、维护成本和沟通成本，因此我并不推荐你使用。

## Base64 图片

最后一类常见的 **React Native** 图片加载方式是 **Base64** 图片。

**Base64** 指的是一种基于 **64** 个可见字符表示二进制数据的方式，**Base64** 图片指的是**使用 Base64 编码**加载图片的方法，它适用于那些图片体积小的场景。

**Base64** 图片的示例代码，如下：

 复制代码

```
1 <Image
2   source={{
3     uri: 'data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADMAAAAzCAYAAAA6oTAqAAA
4   }}
5 />
```

你可以看到 **Base64** 图片并不是图片地址，而是以一大长串的以 **data:image/png; base64** 开头的文本。

通常我们看的图片资源 **.jpg**、**.png** 都是二进制格式的，二进制格式的图片是以独立文件存在的。而当二进制图片 **Base64** 化后，就变成了一段由字母、数字和符号组成的字符串。

通常我们看的图片资源 **.jpg**、**.png** 都是二进制格式的，二进制格式的图片是以独立文件存在的。而 **Base64** 图片并不是单独图片文件，而是以文本形式存在 **.js** 文件中的。

字符串是可以嵌套到 **.js** 文件中的，因此 **Base64** 图片也可以嵌入到 **.js** 文件中。在线上，**Base64** 图片是嵌套在 **Bundle** 文件中的，在加载 **React Native** 页面的同时，**Base64** 字符串也能很快地解析成真正的图片，并展示出来。

由于 **Base64** 图片是嵌套在 **Bundle** 文件中的，所以 **Base64** 图片的优点是无需额外的网络请求展示快，缺点是它会增大 **Bundle** 的体积。在动态更新的 **React Native** 应用中，**Base64** 图

片展示快是以 **React Native** 页面整体加载慢为代价的。原因就是它会增加 **Bundle** 的体积，增加 **Bundle** 的下载耗时，从而导致 **React Native** 页面展示变慢。

即便是相同的图片，**Base64** 字符串的体积也要比二进制字节码的体积要大 **1/3**，这又进一步增加 **Bundle** 的大小。那为什么 **Base64** 转换后的体积，会比二进制的字节码的体积还要大 **1/3** 呢？这就要看下 **Base64** 图片的转换原理了。

我这里放了一张二进制图片转 **Base64** 图片的原理示意图，我们根据这张原理示意图来解释图片 **Base64** 后体积增加的原因。

ASCII	二进制	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0																											
	索引	0								97								110											
	字符	\x00 (不可见字符)								a								n											
Base64	二进制	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	1	1	1	0	0	0	0
	索引	0								6								5								46			
	字符	A								G								F								u			

首先，我们要树立两个基本认知。一是二进制图片不能直接以字符串形式存在 **.js** 文件中，因为直接用 **ASCII** （美国信息交换标准代码）表示的二进制字符串太大了。一个二进制“00000000”，占 8 比特大小，也就是 1 个字节的大小。一个直接用 **ASCII** 表示的字符串“00000000”，占 8\*8 比特大小，也就是 8 个字节的大小，是二进制大小的 8 倍。

二是，二进制图片不能直接以 **ASCII** 的格式转换为字符串，这是因为 **ASCII** 字符集不仅存在可见字符，还存在不可见字符。例如，二进制“00000000”对应的 **ASCII** 字符是“空字符（Null）”，空字符是不可见字符。

但是，二进制图片可以借助 **Base64** 进行转换。**Base64** 从 **ASCII** 256 个字符中选取了 64 个可见字符作为基础，这样就二进制就能以 **Base64** 的格式转换为 **ASCII** 字符串了。例如，二进制 00000000 对应的 **Base64** 字符是 **A**，是可见字符，可见字符 **A** 是可以存在在 **.js** 文件中的。

需要注意的是，**ASCII** 256 个字符需要 8 个比特来表示（ $2^8=256$ ），**Base64** 的 64 个字符只需要 6 个比特位来表示（ $2^6=64$ ）。但实际上，**Base64** 字符也是以 **ASCII** 码的形式存在，因此这里就有 2 个比特的浪费（ $8-6=2$ ）。

你可以再仔细观察一下前面我提供的 **Base64** 转换的原理示意图，相信你一下就能明白其中原理。**Base64** 以 3 个字节作为一组，一共是 24 比特。将这 24 个比特分成 4 个单元，每个单元 6 个比特。每个单元前面加 2 个 0 作为补位，一共 8 个比特，凑整 1 个字符。转换后原来的 24 比特，就变成了 32 比特，因此转换后的体积就大了  $1/3$ （ $1/3 = 1 - 24/32$ ）。

鉴于这样的情况，我的建议是 **Base64 图片只适合用在体积小的图片或关键的图片上**。

## 最佳实践

看到这里，想必现在你已经对 4 类图片的使用场景，有一定的了解了。但是在具体实践中，我们应该怎么用呢？

现在我给你分享一下我的最佳实践，这套方案我也是在实践中，摸索了很久才得出的。这套最佳实践，适用于那些**将 React Native 当做一个动态更新框架来使用的应用中**。

首先是静态图片资源。如果你使用的是自研的热更新平台，就需要注意图片资源一定要先于 bundle 或和 bundle 一起下发，因为在执行 bundle 时，图片资源是必须已经存在的。

接着是网络图片和 **Base64** 图片。这两类图片之所以放在一起说，是因为它们单独管理起来都不方便，一张张手动上传网络图片不方便，一张张手动把图片 **Base64** 化也不方便，所以我们需要一个自动化的工具来管理它们。

比如，你可以把需要上传到网络的图片放在代码仓库的 assets/network 目录，把需要 **Base64** 化的图片放在 assets/base64 目录。

你在本地开发的时候，可以通过使用 require 静态图片资源的形式，引入 assets/network 或 assets/base64 目录中的图片来进行本地调试。在代码编译打包的时候，通过工具将 assets/network 目录中的图片上传到 CDN 上，将 assets/base64 目录中的图片都 **Base64** 化，并将 require 形式的静态图片资源代码转换为网络图片或 **Base64** 图片的代码。使用自动化工具来管理图片，代替人工手动管理，可以提高你的开发效率。

最后是宿主应用图片，这种加载图片的方式我不建议你使用，具体的原因我们前面已经分析过了。

我把这节课讲的这四种图片的使用总结成了这张图，你可以看看，加深一下印象：

	使用方法	底层 scheme	使用建议	使用场景
静态图片资源	<code>require('./dianxin')</code>	<code>file://</code>	跟随 bundle 压缩包下发	关键图片
网络图片	<code>{uri: 'http://cdn.dianxin.jpg'}</code>	<code>http://</code> 或 <code>https://</code>	自研图片管理工具	绝大部分场景
宿主应用图片	<code>uri: 'dianxin'</code>	<code>file://</code> 或 <code>asset://</code>	不建议使用	复用场景
Base64 图片	<code>uri: 'data:image/jpeg;base64...'</code>	<code>data:mime/type;base64</code>	自研图片管理工具	小图片、关键图片



## 总结

今天的课程到这里就结束了，这里我再给你总结一下。

今天我们学习 React Native 中的图片组件 Image，了解了 4 种图片加载的方式和其最佳实践。

首先，发版更新的 React Native 应用，使用内置图片的最佳方式是静态图片资源，但对于动态更新的 React Native 应用而言，需要注意静态图片资源并不是真正的“内置”，而是必须和 Bundle 执行文件“同步”的加载。

然后，我推荐你自研一个图片管理工具，把设计师给你的图片管理起来，并按照指定的配置规则转换为 Base64 图片或网络图片，这样可以提高你的开发效率。

不过，React Native 复用宿主应用图片的这种方式，不推荐你使用。它有加载失败的风险，而且有较高的维护成本和沟通成本。

## 补充材料

1. 各类图片使用区别的 [🔗 React Native 中文网连接在这儿](#)，你可以点击查看一下。
2. React Native 框架对图片的默认缓存处理并不是最优的方案，社区中提供了替代方案 [🔗 FastImage](#)，它是基于 SDWebImage (iOS) 和 Glide (Android) 实现的性能和效果会更好一些。
3. 这节课的示例代码，我放在了 [🔗 GitHub](#) 上，你可以参考一下。




# 作业

1. 请你分别实现一下静态图片资源、网络图片和 **Base64** 图片这三类图片。体会一下，是否有资源管理上改进空间？
2. 如果要你来实现图片管理工具，你会怎么实现？

欢迎在留言区分享你的想法。我是蒋宏伟，咱们下节课见。

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 04 | State：如何让页面“动”起来？

下一篇 06 | Pressable：如何实现一个体验好的点按组件？

## 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。