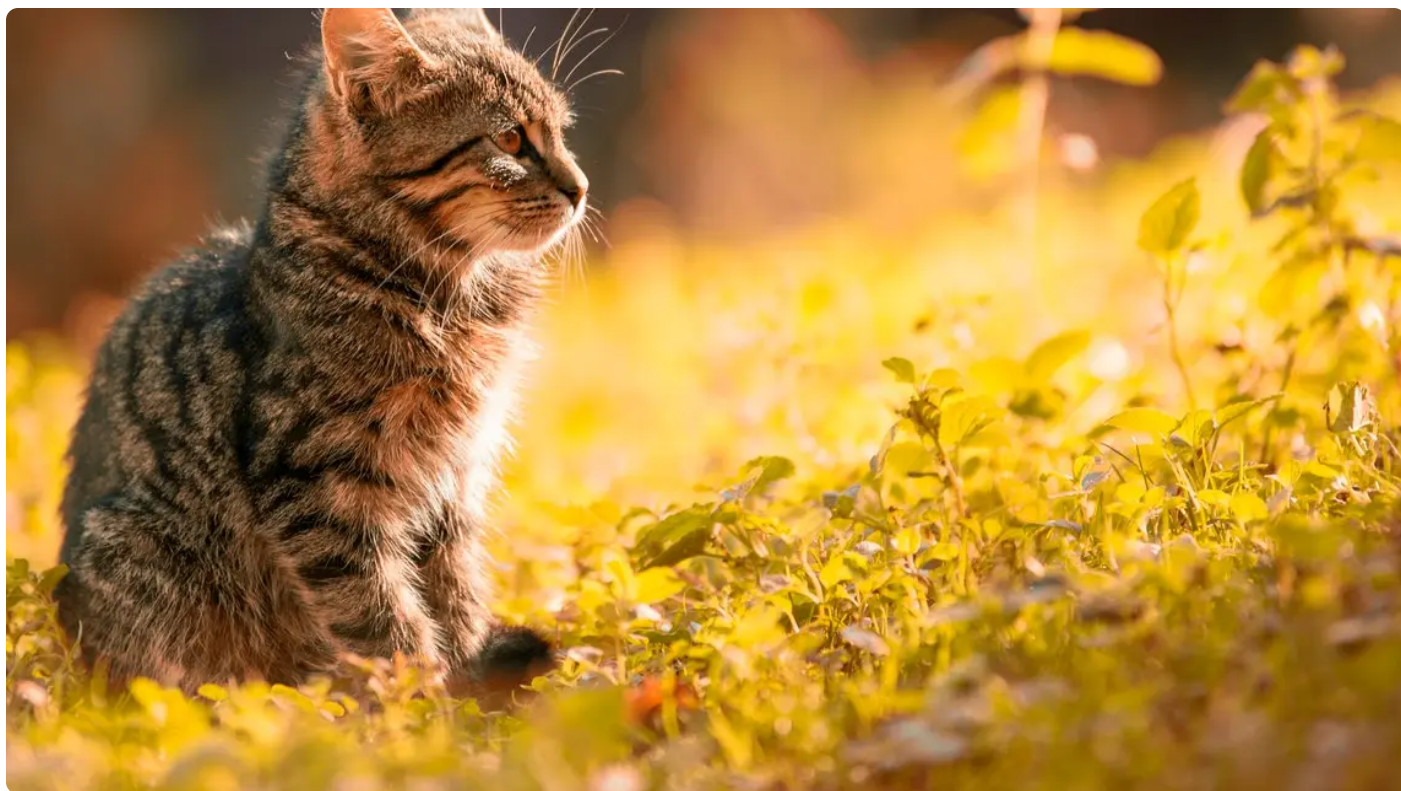


## 09 | Fast Refresh: 提高 UI 调试效率神器

2022-04-15 蒋宏伟

《React Native 新架构实战课》

课程介绍 >



讲述：蒋宏伟

时长 20:15 大小 18.55M




你好，我是蒋宏伟。今天我们来讲一讲提高 UI 调试效率的方法。

在开发 UI 时，大家一般都是——一边看设计稿，一边写代码，一边调试，三种行为交替进行的。谁的大脑都不是一台编译器，也不能安装真正的 **React Native** 环境。即使已经思考得很完备了，我们也不能保证写完的一段代码里面没有任何 **Bug**，每次写完的代码都能完美符合我们预期的设计。所以，我们离不开 UI 调试。

那 UI 调试效率重要吗？非常重要。你可以回想一下，是不是我们大部分的业务开发都会涉及到 UI 的开发。而在 UI 开发的过程中，你是不是会花费很多时间在调试代码上，甚至调试时间可能比真正写代码的时间还要多？正是如此，我们才更应该花点时间学一下调试技巧，把 UI 开发整体效率给提上去。

今天这节课，我会先从 **React Native** 快速刷新的使用讲起，然后再深入核心原理，帮你理解如何更好地使用快速刷新，提高你的 UI 开发效率。

## 使用快速刷新

React Native  快速刷新（Fast Refresh）是默认开启的，你不用做任何额外的配置，就能立刻体验到。

快速刷新提效的本质是**及时反馈**。也就是说，你写下代码后就能看到 UI，没有其他任何多余步骤。代码完成了，UI 就更新了，这就是及时反馈。

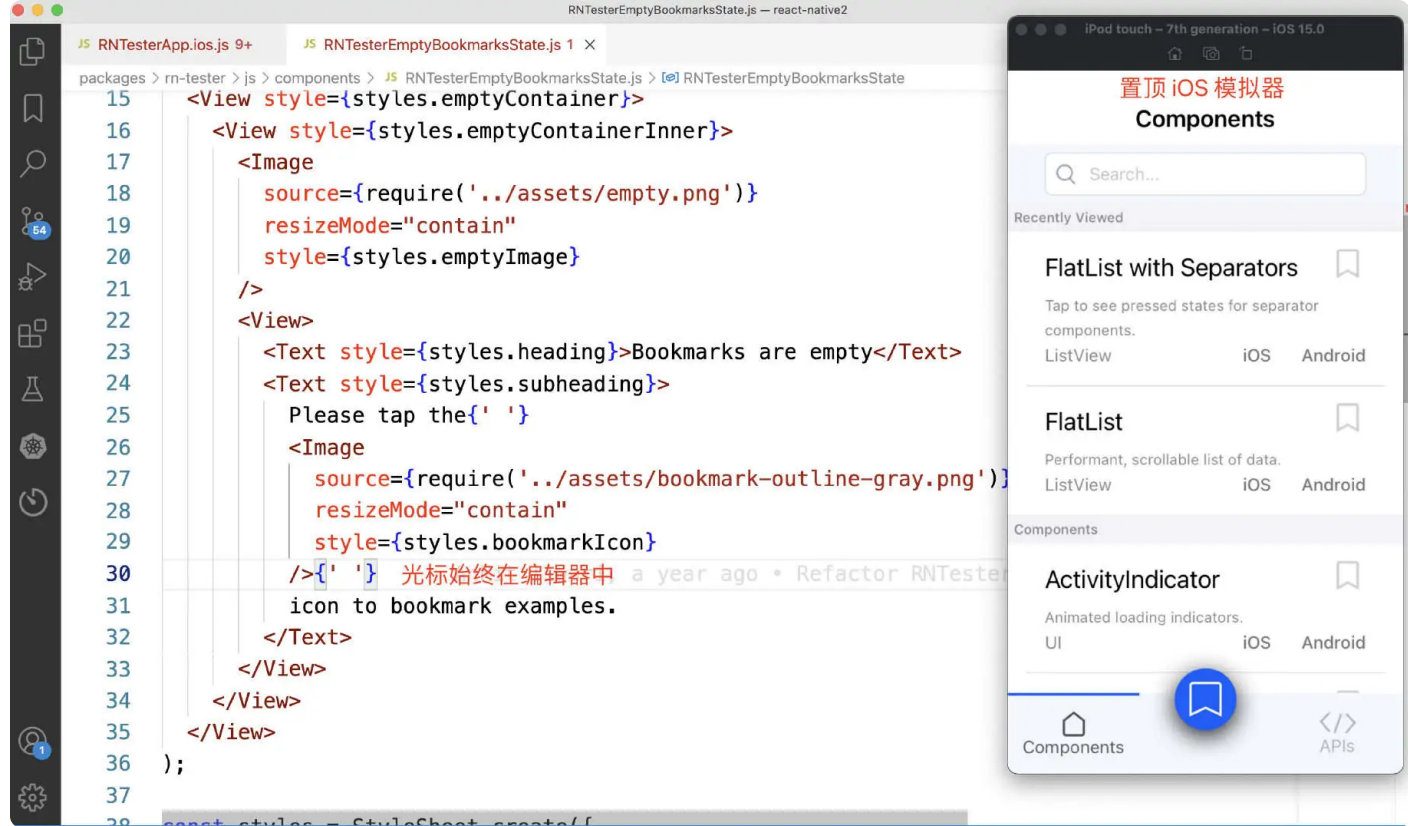
假设，你正在开发一个商品列表页面。UI 稿中图片左边距为 30px，你在 Image 的样式中增加了一行 `marginLeft: 30` 的代码。当你按下快捷键 `cmd + s` 保存代码时，不到 1s 的时间，你就看到你屏幕右侧模拟器中，所有商品图片都移到正确的位置上了。

你能在心中快速验证一下，是对的，然后你又添加一行上下居中的代码，又是不到 1s 的时间，商品图片又位移了一下。嗯，完美居中。每一次的 UI 调试都是，所码即所见，无与伦比的开发体验，让你沉浸在这开发的心流中。

这实际上就是我日常使用 React Native 快速刷新能力开发 UI 界面的感受。

在使用快速刷新时，你应该知道一个提升开发效率的小技巧。我日常开发时习惯把模拟器放在代码编辑器右边，并且会把模拟器勾选 `window => stay on top` 选项，在把模拟器置顶在编辑器上方。

这样，我们就能在写代码和调试的同时，立刻看到模拟器中的效果。相比真机调试或者多屏来回切换，置顶模拟器可以减少手离开键盘和视野来回切换的次数，提高你的开发效率。



看到这里，你是不是很好奇，快速刷新带来的“所码即所见”能力的原理究竟是什么样的？

## 基础原理：模块热替换

React Native 的快速刷新功能的最早期版本，叫做热重载 Hot Reload，是基于 Webpack 的模块热替换 [\(Hot Module Replacement\)](#) 的原理开发的。我们写 React Native 之前，都会运行一个 `react-native start` 命令，启动一个 Metro 服务，而 Metro 服务就实现了模块热替换的功能。

Metro 服务会把更新的代码打包发送给 React Native 应用，让应用能够及时更新，那这个过程大概是怎么样的呢？

首先，Metro 服务会监听代码文件的变化，当你修改完代码（①），保存文件时（②），Metro 服务就会收到通知。在你保存好后，Metro 就会编译涉及到的更新文件（③），编译完成后再生成一个用于更新的 bundle。

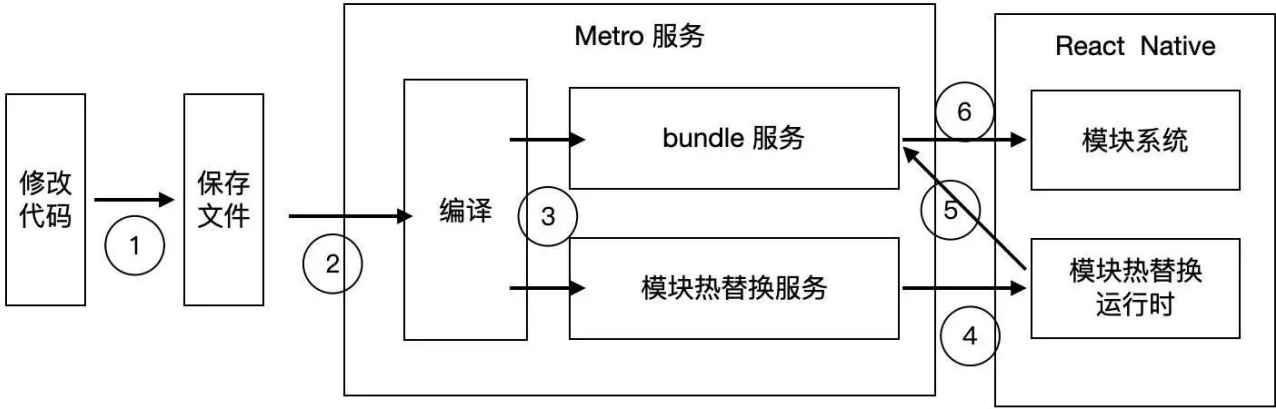
而 Metro 的模块热替换服务和 React Native 应用中的模块热替换客户端（HMR Client），在启动时其实已经建立好了 socket 连接。

所以，当新 **bundle** 生成时，模块热替换服务会通过 **socket** 通知块热替换客户端，热替换客户端实际就是运行在 **React Native** 应用中的一段 **JavaScript** 代码，它一开始就执行了一个 **socket** 监听事件（④）。

**React Native** 收到通知后，就会向请求 **bundle** 服务发起请求。然后，**bundle** 服务会返回一个用于更新的 **bundle**（⑤），并使用 **JavaScript** 引擎，在原来 **React Native** 应用的 **JavaScript** 上下文中执行用于更新的 **bundle**。

这个 **bundle** 是由多个模块组成的，你修改代码文件对应的模块及其依赖模块都是新模块，新模块会把原先的旧模块替换掉。⑥

这就是整个模块热替换的全部过程，这里我放了一张流程图，你可以参考一下：



但是这里会有一个问题，仅仅只是用新模块替换旧模块，会导致原生视图重新渲染，并且丢失原有状态。

这是因为，新模块的重新执行就意味着，每个新模块中的组件，无论是类组件或者函数组件，都会被重新创建。而 **React** 在判断是否要更新的时候，会判断更新前后的两个组件是否相等。这样一来，即便新旧组件的代码完全一样，**React** 也会认为你销毁了原有组件，又创建了一个新的组件。而组件所对应的原生视图，也会发生销毁和重建。

这就好比，你先创建了一个旧的空对象，然后又创建了一个新的空对象。虽然代码完全一样，都是空对象，但是你用全等去判断时，因为对象是引用类型，创建了一个新对象就创建了一个新的引用，新的引用又不等于旧的引用，所以新对象是不等于旧对象的：



```
1 // 新的空对象 ≠ 旧的空对象
2 {} !== {}
```

同理，当你保存 **List** 组件时，即便你没有对 **List** 组件中的代码做任何修改，模块热替换后，**React** 也会认为，你保存之前的是旧组件，保存之后的是新组件。而新组件不等于旧组件，那它就会帮你销毁旧的原生视图，并重新创建新的原生视图。这个时候，原有组件状态 **state** 和原生列表的滚动位置都会丢失：

 复制代码

```
1 // 保存前: oldList.js
2 export default function List {}
3 // 保存后: newList.js
4 export default function List {}
5 // 渲染的都是 List 组件
6 render(){ <List /> }
7 // 但是，因为 newList ≠ oldList
8 require('newList').default !== require('oldList').default
9 // 所以，React 会销毁旧的 List 原生视图，创建新的 List 原生视图
```

也就是说，基础的模块热替换功能只能实现组件级别的强制刷新，而组件状态的丢失，会导致开发效率的降低。

你想啊，当你要在商品列表页面中开发一个弹窗时，你修改了弹窗组件，一保存，弹窗组件强制刷新了，然后就消失了。你要又点开弹窗，重来一次。弹窗还稍微好点，如果是层级更新的组件，你要多次操作才能使用，如此反复操作，开发效率会变得很低。

## 进阶能力：复用组件及其状态

那么，**React Native** 的快速刷新功能，是如何实现组件状态不丢失，原生视图不重建的呢？

快速刷新功能复用组件和状态的原理分为两个步骤：

1. 在编译时，修改组件的注册方式；
2. 在运行时，用“代理”的方式管理新旧组件的切换。

在编译时，快速刷新的 **babel** 插件 [@ReactFreshBabelPlugin](#) 修改你的代码，将你的组件转换成可被代理的组件。快速刷新 **babel** 插件和其他 **babel** 插件一样，它的功能都是对代码进行

转换。正如你使用 **babel** 可以把 **JSX** 转换为 **JavaScript** 一样，快速刷新 **babel** 插件也可以在你组件源代码中插入一些代码，实现组件的“代理”。

打一个比方，如果我们要对一个自定义的 **Counter** 函数组件实现代理。那我们要怎么做呢？首先，在 **metro** 打包时，快速刷新 **babel** 插件，找到文件中要导出的 **Counter** 组件；然后，通过它的函数名、文件名生成一个全局唯一的 **ID**，例如 **'Counter.js#Counter'**；最后，生成一行注册代码。这行代码的作用是，将 **ID** 作为一个不变的对象标识，用这个不变的对象去“代理”，因模块热替换而变化 **Counter** 组件，具体你可以看下这里：

 复制代码

```
1 // 源代码
2 export function Counter() {
3   const [count, setCount] = useState(0);
4   const handlePress = () => setCount(count + 1)
5   return <Text onPress={handlePress}>times:{count}</Text>
6 }
7
8 const __exports_default = Counter
9 export default __exports_default
10
11 // 由快速刷新 babel 生成
12 // 将组件注册到组件管理中心
13 register('Counter.js#Counter', Counter)
```

有了编译时插入的注册代码，在运行时，我们就可以用“代理”的方式，管理新旧组件的切换了。

无论是初次加载的 **Count** 组件，还是后续模块热替换不断新建的 **Counter** 组件，都会放在组件注册中心。而“代理”只会在 **Count** 组件初次加载时创建，创建之后就作为一个不变的对象放在“代理”注册中心。

在代码保存后，模块热替换会将新的组件代码运行，在新组件被创建的同时，新组件的注册函数就会被执行了。通过唯一的 **ID**，找到对应的不变“代理”，并将代理的 **current** 引用，切换到新组件上，完成新旧组件的切换。

 复制代码

```
1 // ReactFreshRuntime.js
2 // “代理”注册中心
3 const allFamiliesByID = {}
```

```

4 // 组件注册中心
5 const allFamiliesByType = {}
6
7 function register(id, componentType) {
8   let family = allFamiliesByID[id];
9
10  if (family === undefined) {
11    family = {current: componentType};
12    // 将不变的“代理”放入“代理”注册中心
13    allFamiliesByID[id] = family
14  } else {
15    // 用不变的“代理”，管理新旧组件的切换
16    const prevType = family.current;
17    family.current = componentType;
18  }
19  // 将所有组件都放入组件注册中心
20  allFamiliesByType[componentType] = family;
21 }
22

```

这就在保证组件不变的情况下，完成了新旧组件的切换。

因为代理组件是存在全局对象上的，所以当你保存代码引起模块系统更新时，代理组件的引用也不会发生改变。接着，页面开始更新，此时调用的是代理组件的 **render** 方法，然后代理组件调用的更新后的新模块组件的 **render** 方法。你每保存一次代码，模块系统都更新一次，代理组件实际 **render** 也会进行一次切换，但是只要你的代码没有变化，**React** 也不会重新创建原生视图。**React Native** 组件级别的快速刷新，就是通过**代理组件**实现的。

那究竟是如何实现复用组件及其状态的呢？

我们先来说状态复用。在我们前面的示例中，我们把 **Counter** 函数组件，放在了 **Counter.js** 的文件中，一个文件就是一个模块，如果里面只有一个函数组件的话，我们就可以把它叫做一个函数组件模块。模块代码是执行在该模块的上下文中的，上下文有着各种变量，其中就包括状态。通过“代理”组件的方式，就可以实现在同一个组件模块的上下文中，执行不同的函数组件。无论是新函数组件，还是旧函数组件，用的都是相同的状态，这就是状态复用。

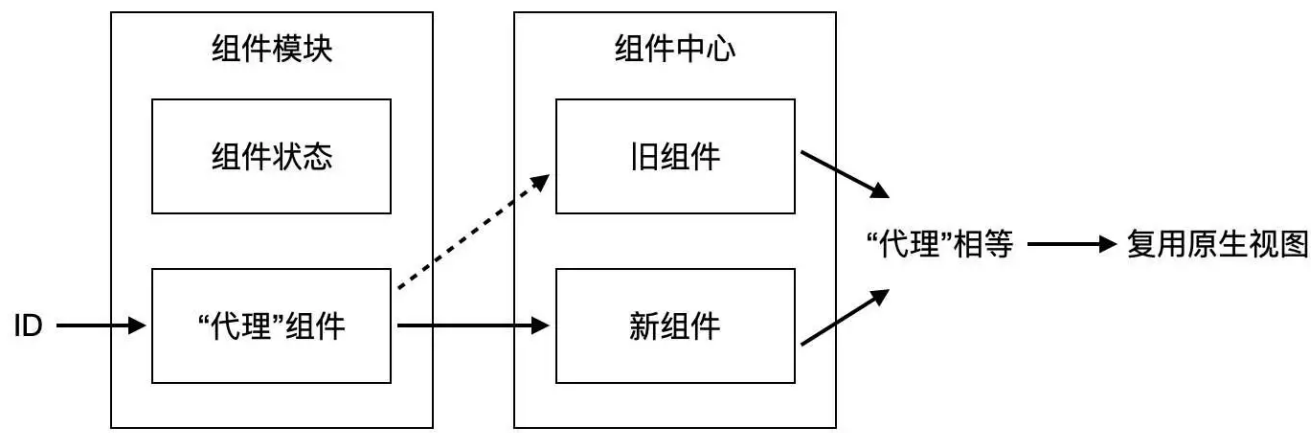
那么组件所代表的原始视图的复用又指的是什么呢？

我们同样打个比方，假设现在你改动的组件要开始渲染 **render** 了。我们前面提到过，**render** 时判断是否要重新创建原生视图，是通过浅对比算法 **shallowCompare** 实现的。如果新旧组件的类型相等就走 **re-update** 的逻辑不创建，如果新旧组件的类型相等就走 **re-mount** 的逻辑重

新创建。现在，新旧组件的“代理”是就是同一个对象，状态也不会发生改变，浅对比算法判断肯定相等，所以原生视图不会重新创建，从而实现了原生视图的复用。

简而言之，**React Native** 的快速刷新功能，就是通过“代理”组件的方式，实现了组件状态不丢失，原生视图不重建。

这里我放了快速刷新 **babel** 编译后的复用模型，可以帮助你理解复用的实现原理：



当然，并不是所有情况都会复用状态和原生视图。

这又从何说起呢？我从组件类型的角度来给你解释。组件有两种类型：函数组件和类组件。

对于函数组件来说，**hooks** 的顺序非常重要，相同的状态下，不同的顺序会有不同的结果。如果你修改了 **hooks** 的顺序，快速刷新时就会重新初始化状态。在其他情况下，函数组件的快速刷新都会为你保留状态。

对于类组件来说，只要是类组件的代码发生更新，组件的内部状态都要重新初始化。关于这点，快速刷新功能的作者 **Dan** 在博客中解释到，“（保留类组件的状态）热重载是非常不可靠的，最大原因就是类组件的方法是可以被动态替换的。是的，在实例原型链上替换它们很简单，但是根据我的经验，有太多边缘情况了，它根本没有办法可靠地工作”。

🔗 The simple answer is “replace them on the prototype” but even with Proxies, in my experience there are too many gnarly edge cases for this to work reliably.



所以，我给你的建议是，**尽可能地拥抱函数组件，放弃类组件**。这样你在 UI 调试的时候，就能更多的享受函数组件带来的状态保留好处。特别是一些入口很深的组件，需要多次操作后才能调试，一旦导航、蒙层、滚动之类的组件状态丢失了，整个操作就要重新再来一遍，才能重新进行调试。拥抱函数组件，你的调试效率才会更高。

当然，如果项目中已经用到了很多类组件，又要调试一些入口很深的组件，怎么办？方法也很简单，你应该把你要调试的组件，单独拎出来调试。如果拎出来的组件和其他组件有依赖关系，也可以通过 **mock** 数据的形式对其依赖进行解耦，实现快速调试。

想象你已经理解快速刷新的基本原理，接下来我们会站在一个更高的视角，看一下快速刷新的完整策略。

## 整体策略：逐步降级

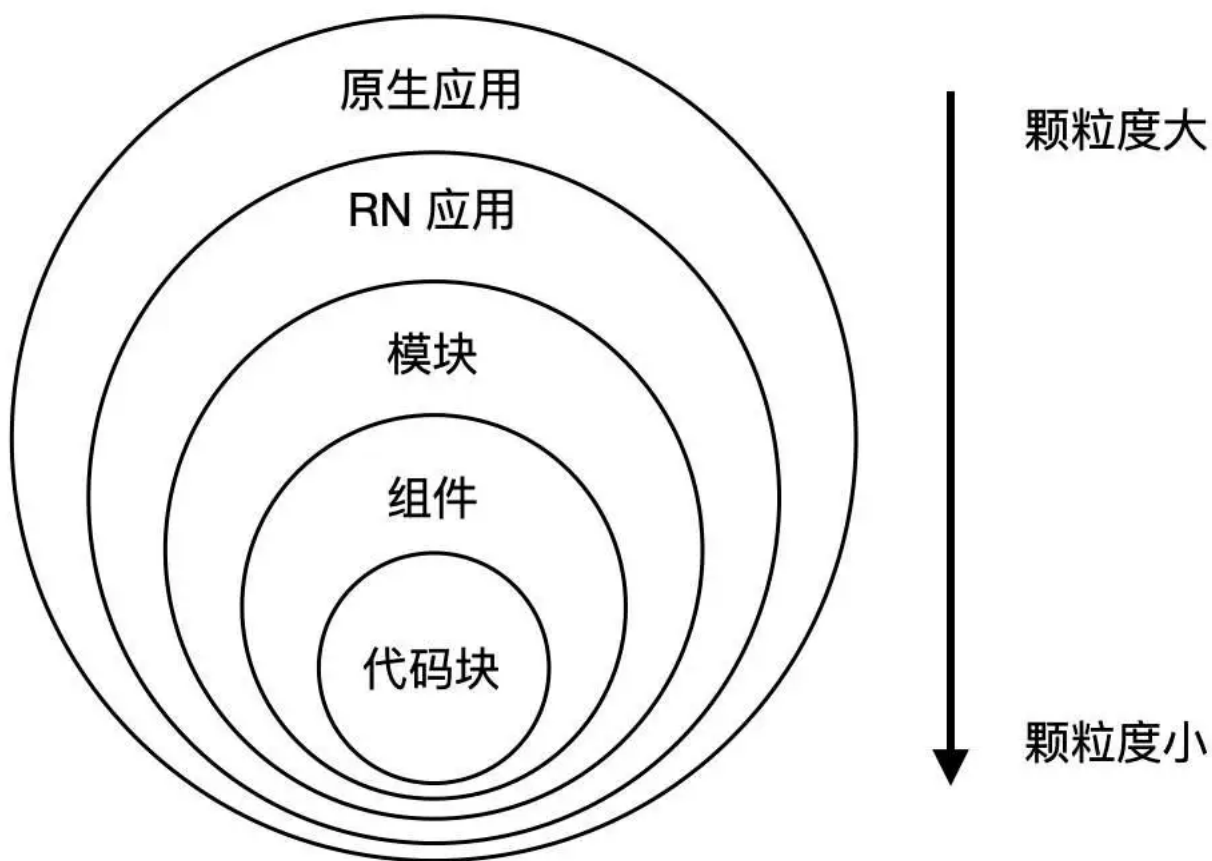
在编程调试时，有各式各样的代码，有函数组件、类组件、工具函数和常量等等。那么，是什么样的策略，能让你尽可能地快看到调试结果呢？

快速刷新的整体策略就是**逐步降级**。如果颗粒度最小的更新不能使用，就换成颗粒度大一些的更新：

- 代码块：如果你只修改了函数组件中的一些代码块，并且没有改动 **hooks** 的顺序。快速刷新在复用状态和原生视图的同时，你对该文件的所有修改都会生效，包括样式、渲染逻辑、事件处理、甚至一些副作用；
- 组件：如果你修改了类组件中的任意代码，快速刷新会使用新的类组件进行重新渲染，原来的状态和原生视图都会被销毁；
- 模块：如果你修改的模块导出的东西不只是 **React** 组件，快速刷新将重新执行该模块以及所有依赖它的模块；
- **React Native** 应用：如果你修改的文件被 **React** 组件树之外的模块引用了，快速刷新将重新渲染整个 **React Native** 应用。

可以看到，快速刷新的逐步降级策略是，**从更新颗粒度最小代码块开始的，然后是组件、模块，最后是大颗粒度的 React Native 应用**。越小颗粒度的更新，为我们保留了越多原来的状态和环境，我们的开发调试效率也更高。

更新的颗粒度越小，效率越高



在调试的过程中，还会有奇奇怪怪的报错发生，比如语法错误、运行时错误和错误边界，这些错误快速刷新都帮你捕获到了。因此，快速刷新还有很强的鲁棒性。

## 总结

现在，再回到我们最初的话题，如何提高 UI 调试效率？我相信现在你已经有了答案。

调试 UI 最重要的是即使反馈和所码即所见。**React Native** 的快速刷新能力，会把我们的代码修改，尽可能快地展示出来。

能够实现快速刷新原因是，快速刷新能够通过模块热替换的方式，用我们修改后的新模块替换原来的旧模块。如果，该模块导出的是组件，那么“代理”组件就会将引用从旧组件切到新组件上，实现组件级别的刷新。如果，函数组件且 **hooks** 顺序没有发生改变，快速刷新时原有的组件状态也会保留。快速刷新时，越小颗粒度的更新，速度越快，调试效率更高。

要用好快速刷新功能，还有三个小技巧：

1. 同屏预览。将模拟器置顶在编辑器上方，减少你视野来回切换频率；
2. 拥抱函数组件。函数组件能保留原有组件状态，减少你操作交互的次数；
3. 单独拎出来调试。单独拎出来先开发独立组件再集成，可能会比在层层嵌套的代码结构中开发效率更高。


## 思考题

今天这一讲介绍的主要介绍的是理论知识，我就不给你留作业了，请你思考一下为什么快速刷新功能的作者 **Dan** 认为，保留类组件的热重载非常不可靠的，但函数组件却是可行的？

欢迎在留言区写下你的想法。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](#) 08 | List：如何实现高性能的无限列表？

## 精选留言 (1)

 写留言



袁德圣

2022-04-16

请教一下老师用的什么模拟器？



