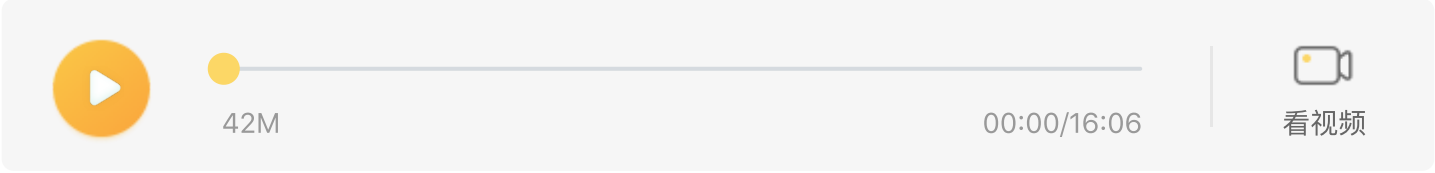
23 | 跟 React 学设计模式：掌握编程“套路”，打造高质量应用

2020/12/28 修言



这一讲我们将针对 React 中的设计模式进行探讨。

与性能优化的视角相似，当我们谈论 React 设计模式时，实际上是在谈论“React 组件的设计模式”。在 React 设计模式中，最重要、也是最为大家所津津乐道的几个模式分别是：

- **高阶组件（HOC）**
- **Render Props**
- **剥离有状态组件与无状态组件**

除此之外，每个团队或许都会有自己的一套在实践中摸索出的设计模式方法论。这些形态各异的 React 设计模式虽然实现思路有所不同，但本质上基本都是为了实现这样一个终极目标——以尽可能优雅的姿态，**实现组件逻辑的复用**。

而我们今天即将探讨的这 3 种模式，也无出其右。

在本讲，我们首先要做的第一件事情，就是把每一种设计模式到底是什么、怎么用给弄清楚。在这个过程中，你会对“单一职责”和“开放封闭”这两个非常重要的设计原则有所认知。

在此基础上，你还需要去思考这样一个问题：设计模式是否是万能的？如果不是，那么对框架来说，还有没有更加深刻、彻底的解法呢？

没错，最终的话题还是要回到 Hooks 上来，哈哈。接下来我们就先迈出第一步，一起认识/重温一下高阶组件这个概念。

高阶组件（HOC）：最经典的组件逻辑复用方式

什么是高阶组件

> 高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。——React 官方

高阶组件（Higher Order Components）在概念上沿袭了高阶函数（Higher-Order Function）。在第 20 讲，我们已经和高阶函数打过交道，这里我们再复习一下高阶函数的概念：**接收函数作为输入，或者输出另一个函数的一类函数，就是高阶函数。**

相应的，高阶组件指的就是**参数为组件，返回值为新组件的函数**。没错，**高阶组件本质上是一个函数**。下面是一个简单的高阶组件示例：

复制代码

```
const withProps = (WrappedComponent) => {
    const targetComponent = (props) => (
        <div className="wrapper-container">
            <WrappedComponent {...props} />
        </div>
    );
    return targetComponent;
};
```

在这段代码中，withProps 就是一个高阶组件。

高阶组件是如何实现逻辑复用的?

现在我们考虑这样一种情况：我有一个名为 checkUserAccess 的方法，这个方法专门用来校验用户的身份是否合法，若不合法，那么一部分组件就要根据这个不合法的身份调整自身的展示逻辑（比如查看个人信息界面需要提示“请校验身份”等）。

假如说页面中的 A、B、C、D、E 五个组件都需要甄别用户身份是否合法，那么这五个组件在理论上都需要先请求一遍 checkUserAccess 这个接口。但一个一个对组件进行修改未免太麻烦了，我们期望对“获取 checkUserAccess 接口信息，并通知到对应组件”这层逻辑进行复用，这时候就可以请出高阶组件来帮忙了。

我们可以像下面代码这样在高阶组件中定义这层通用的逻辑：

复制代码

```
// 假设 checkUserAccess 已经在 utils 文件中被封装为了一段独立的逻辑
import checkUserAccess from './utils
// 用高阶组件包裹目标组件
const withCheckAccess = (WrappedComponent) => {
    // 这部分是通用的逻辑：判断用户身份是否合法
    const isAccessible = checkUserAccess()
    // 将 isAccessible（是否合法） 这个信息传递给目标组件
    const targetComponent = (props) => (
        <div className="wrapper-container">
            <WrappedComponent {...props} isAccessible={isAccessible} />
        </div>
    );
    return targetComponent;
};
```

这样当我们需要为某个组件复用这层请求逻辑的时候，只需要直接用 withCheckAccess 包裹这个组件就可以了。以 A 组件为例，假设 A 组件的原始版本为 AComponent，那么包裹它的形式就是下面代码这样：

```
const EnhancedAComponent = withCheckAccess(Acomponent);
```

复制代码

通过简单地对高阶组件 withCheckAccess 进行引入，EnhancedAComponent 轻松具备了校验用户合法性的能力。这样一来，即便再多出 5 个组件想要引入 checkUserAccess，我们也不会怂——毕竟包裹五个组件和重写五段逻辑的工作量是没法相提并论的，哈哈。

高阶组件不仅能够帮助我们简化逻辑的引入过程，还可以帮助我们规避掉逻辑变更带来的烦琐的修改步骤：假如这段 checkUserAccess 的逻辑是散落在 A、B、C、D、E 这五个组件之中的，那么一旦 checkUserAccess 的判定规则需要修改，我们就得需要去修改五段代码；但现在，checkUserAccess 被抽离进了一个独立的高阶组件里，我们在高阶组件中的一次修改，将在所有被它处理过的组件中生效。

由此可以看出，高阶组件可以帮助我们**从根本上减少重复的编写和修改工作**，这不仅是高阶组件这一种模式的利好，更是“逻辑复用”这件事情的意义所在。

Render Props：逻辑复用的另一种思路

> 术语“render prop”是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术。——React 官方

什么是 render props?

render props 是 React 中复用组件逻辑的另一种思路，它在实现上和高阶组件有异曲同工之妙——两者都是把通用的逻辑提取到某一处。区别主要在于使用层面，**高阶组件的使用姿势是用“函数”包裹“组件”，而 render props 恰恰相反，它强调的是用“组件”包裹“函数”**。

一个简单的 render props 可以是这样的，见下面代码：

```
import React from 'react'
const RenderChildren = (props) => {
  return(
     <React.Fragment>
        {props.children(props)}
     </React.Fragment>
  );
};
```

复制代码

RenderChildren 将渲染它所有的子组件。从这段代码里，你需要把握住两个要点：

1. render props 的载体应该是一个**React 组件**，这一点是与高阶组件不同的（高阶组件本质是函数）；
2. render props 组件正常工作的前提是它的**子组件需要以函数形式存在**。

第 1 点相对明显一点，你可能会对第 2 点感到迷惑。没关系，我们直接来看 RenderChildren 的使用方式，请看下面代码：

```
<RenderChildren>
  {() => <p>我是 RenderChildren 的子组件</p>}
</RenderChildren>
```

复制代码

RenderChildren 本身是一个 React 组件，它可以包裹其他的 React 组件。一般来说，我们习惯于看到的包裹形式是“标签包裹着标签”，也就是下面代码演示的这种效果：

```
<RenderChildren>
  <p>我是 RenderChildren 的子组件</p>
</RenderChildren>
```

复制代码

但在 render props 这种模式下，它要求被 render props 组件标签包裹的一定是个函数，也就是所谓的“函数作为子组件传入”。这样一来，render props 组件就可以通过调用这个函数，传递 props，从而实现和目标组件的通信了。

render props 是如何实现逻辑复用的?

这里我仍然以 checkUserAccess 这个场景举例。使用 render props 复用 checkUserAccess 这段逻辑，我们可以这样做，请看下面代码：

```
// 假设 checkUserAccess 已经在 utils 文件中被封装为了一段独立的逻辑
import checkUserAccess from './utils
// 定义 render props 组件
const CheckAccess = (props) => {
    // 这部分是通用的逻辑：判断用户身份是否合法
    const isAccessible = checkUserAccess()
    // 将 isAccessible（是否合法） 这个信息传递给目标组件
    return <React.Fragment>
        {props.children({ ...props, isAccessible })}
      </React.Fragment>
};
```

复制代码

接下来 CheckAccess 子组件就可以这样获取 isAccessible 的值，见下面代码：

```
<CheckAccess>
  {
    (props) => {
      const { isAccessible } = props;
      return <ChildComponent {...props} isAccessible={isAccessible} />
    }
  }
</CheckAccess>
```

到这里，“函数作为子组件传入”这种情况，我们已经了解了它的来龙去脉。但其实，对于 render props 这种模式来说，**函数并不一定要作为子组件传入，它也可以以任意属性名传入，只要 render props 组件可以感知到它就行**。

举个例子，我可以允许函数通过一个名为 checkTaget 的属性传入 render props 组件，那么 CheckAccess 组件只需要改写一下它接收函数的形式即可，见下面代码：

```
// 假设 checkUserAccess 已经在 utils 文件中被封装为了一段独立的逻辑
import checkUserAccess from './utils
// 定义 render props 组件
const CheckAccess = (props) => {
    // 这部分是通用的逻辑：判断用户身份是否合法
    const isAccessible = checkUserAccess()
    // 将 isAccessible（是否合法） 这个信息传递给目标组件
    return <React.Fragment>
        {props.checkTaget({ ...props, isAccessible })}
      </React.Fragment>
};
```

在使用 CheckAccess 组件的时候，我们将函数放在 checkTaget 中传入组件即可，见下面代码：

```
<CheckAccess
    checkTaget={(props) => {
      const { isAccessible } = props;
      return <ChildComponent {...props} isAccessible={isAccessible} />
    }}
  />
```

像这样使用 render props，也是完全可以的。

理解 render props 的灵活之处

读到这里，你不免会产生这样的困惑：高阶组件和 render props 都能复用逻辑，那我到底用哪个好呢?

这里我先给出结论：render props 将是你更好的选择，因为它**更灵活**。这“更灵活”从何说起呢?

render props 和高阶组件一个非常重要的区别，在于对数据的处理上：在高阶组件中，目标组件对于数据的获取没有主动权，**数据的分发逻辑全部收敛在高阶组件的内部**；而在 render props 中，除了父组件可以对数据进行分发处理之外，**子组件也可以选择性地对数据进行接收**。

这样说你可能会觉得有点抽象，我举个例子：假如说我们现在多出一个 F 组件，它同样需要 checkUserAccess 这段逻辑。但是这个 F 组件是一个老组件，它识别不了 props.isAccessible，只认识 props.isValidated。带着这个需求，我们先来看看高阶组件怎么解决问题。原有的高阶组件逻辑是下面这样的：

复制代码

```
// 假设 checkUserAccess 已经在 utils 文件中被封装为了一段独立的逻辑
import checkUserAccess from './utils
// 用高阶组件包裹目标组件
const withCheckAccess = (WrappedComponent) => {
    // 这部分是通用的逻辑：判断用户身份是否合法
    const isAccessible = checkUserAccess()
    // 将 isAccessible（是否合法） 这个信息传递给目标组件
    const targetComponent = (props) => (
        <div className="wrapper-container">
            <WrappedComponent {...props} isAccessible={isAccessible} />
        </div>
    );
    return targetComponent;
};
```

它会不由分说地给所有组件安装上 isAccessible 这个变量。要想让它适配 F 组件的逻辑，最直接的一个思路就是在 withCheckAccess 中增加一个组件类型的判断，一旦判断出当前入参是 F 组件，就专门将 isAccessible 改名为 isValidated。

这样做虽然能够暂时解决问题，但这并不是一个灵活的解法：假如需要改属性名的组件越来越多，那么 withCheckAccess 内部将不可避免变得越来越臃肿，长此以往将难以维护。

事实上，在软件设计模式中，有一个非常重要的原则，叫“**开放封闭原则**”。一个好的模式，应该尽可能做到**对拓展开放，对修改封闭**。

当我们发现 withCheckAccess 的内部逻辑需要频繁地跟随需求的变化而变化时，此时就应该提高警惕了，因为这已经违反了“对修改封闭”这一原则。

处理同样的需求，render props 就能够在保全“开放封闭”原则的基础上，帮我们达到目的。

前面说过，在 render props 中，除了父组件可以对数据进行分发处理之外，**子组件也可以选择性地对数据进行接收**。这就意味着我们可以在新增的 F 组件相关的逻辑中把数据适配这件事情给做掉（如下面代码所示），而不会影响老的 CheckAccess 组件中的逻辑。

```
<CheckAccess>
  {
    (props) => {
      const { isAccessible } = props;
      return <ChildComponent {...props} isValidated={isAccessible} />
    }
  }
</CheckAccess>
```

这样一来，不管你新来的组件有多少个，需要变更的属性名有多少个，影响面都会被牢牢地控制在“新增逻辑”这个范畴里。契合了“开放封闭”原则的 render props 模式显然比高阶组件灵活多了。

有状态组件与无状态组件：“单一职责”原则在组件设计模式中的实践

什么是“单一职责”原则?

单一职责原则又叫“单一功能原则”，它指的是一个类或者模块应该有且只有一个改变的原因。通俗来讲，就是说咱们的组件功能要尽可能地聚合，不要试图让一个组件做太多的事情。

什么是有状态组件？什么是无状态组件?

无状态组件这个概念我们在第 06 讲中已经介绍过了，这里简单复习一下：

> 函数组件顾名思义，就是以函数的形态存在的 React 组件。早期并没有 React-Hooks 的加持，函数组件内部无法定义和维护 state，因此它还有一个别名叫“无状态组件”。

如下面代码所示，就是一个典型的无状态组件：

```
function DemoFunction(props) {
  const { text } = props
  return (
    <div className="demoFunction">
      <p>{`function 组件所接收到的来自外界的文本内容是：[${text}]`}</p>
    </div>
  );
}
```

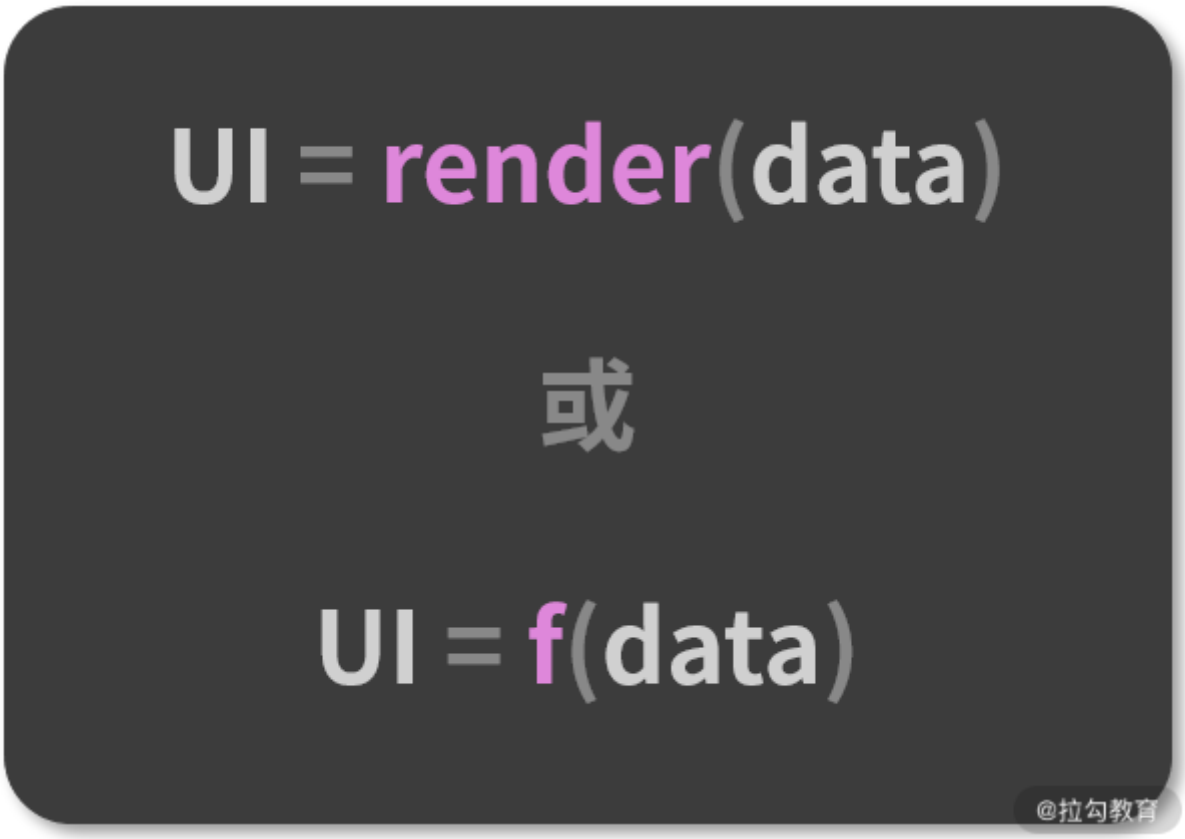
无状态组件不一定是函数组件，不维护内部状态的类组件也可以被认为是无状态组件。

相比之下，能够在组件内部维护状态、管理数据的组件，就是“有状态组件”。

为何需要剥离有状态组件和无状态组件?

有状态组件和无状态组件有很多别名，有的书籍里也会管它们叫“容器组件”和“展示组件”，甚至“聪明组件”和“傻瓜组件”。不管叫啥，核心目的就一个——把数据处理和界面渲染这两个工作剥离开来。

为什么要这样做？别忘了，React 的核心特征是“数据驱动视图”，我们经常用下图的公式来表达它的工作模式：



因此对一个 React 组件来说，它做的事情说到底无外乎是这两件：

1. 处理数据（包括数据的获取、格式化、分发等）
2. 渲染界面

我们当然也可以在一个组件里面做完这两件事情，但这样不够优雅。

按照“单一职责”的原则，我们应该将数据处理的逻辑和界面渲染的逻辑剥离到不同的组件中去，这样功能模块的组合将会更加灵活，也会更加有利于逻辑的复用。此外，单一职责还能够帮助我们尽可能地控制变更范围，降低代码的维护成本：当数据相关的逻辑发生变化时，我们只需要去修改有状态组件就可以了，无状态组件将完全不受影响。

Why Hooks：设计模式解决不了所有的问题

设计模式虽好，但它并非万能。

就 React 来说，无论是高阶组件，还是 render props，两者的出现都是为了弥补类组件在“逻辑复用”这个层面的不灵活性。它们各自都有着自己的不足，这些不足包括但不限于以下几点：

1. 嵌套地狱问题，当嵌套层级过多后，数据源的追溯会变得十分困难
2. 较高的学习成本
3. props 属性命名冲突问题
4. ……

总体来看，“HOC/render props+类组件”这种研发模式，还是不够到位。当设计模式解决不了问题时，我们本能地需要从编程模式上寻找答案。于是便有了如今大家在 React 中所看到的 “函数式编程”对“面向对象”的补充（并且大有替代之势），有了今天我们所看到的“一切皆可 Hooks”的大趋势。

现在，当我们想要去复用一段逻辑时，第一反应肯定不是“高阶函数”或者“render props”，而应该是“自定义 Hook”。Hooks 能够很好地规避掉旧时类组件中各种设计模式带来的弊端，比如说它不存在嵌套地狱，允许属性重命名、允许我们在任何需要它的地方引入并访问目标状态等。由此可以看出，一个好的编程模式可以帮我们节约掉大量“打补丁”式地学习各种组件设计模式的时间。框架设计越合理，开发者的工作就越轻松。

总结

本讲，我们围绕“React 组件设计模式”这一专题进行学习。在认识高阶组件、render props 两种经典设计模式的同时，也对“单一职责”“开放封闭”这两个重要的软件设计原则形成了初步的认识。

软件领域没有银弹，就算有，也不可能是设计模式。通过本讲的学习，相信你在认识设计模式的利好之余，也认识到了它的局限性。在此基础上，相信你会对 React-Hooks 及其背后的“函数式编程”思想建立更加强烈的正面认识。

行文至此，整个 React 专栏的知识讲解部分就结束了。相信不少同学在学习完毕之后，都会对 React 的运行机制，甚至前端框架这个领域产生强烈的兴趣。在下一讲，我将分享自己对框架的一些理解，也会借机和你聊聊前不久刚刚推出的 React 17，我们不见不散呀！