06 | React-Hooks 设计动机与工作模式（上）

2020/10/28 修言

从本课时开始，我们将逐步进入 React-Hooks 的世界。

在动笔写 React-Hooks 之前，我发现许多人对这块的知识非常不自信，至少在面试场景下，几乎没有几个人在聊到 React-Hooks 的时候，能像聊 Diff 算法、Fiber 架构一样滔滔不绝、言之有物。后来我仔细反思了一下，认为问题应该出在学习姿势上。

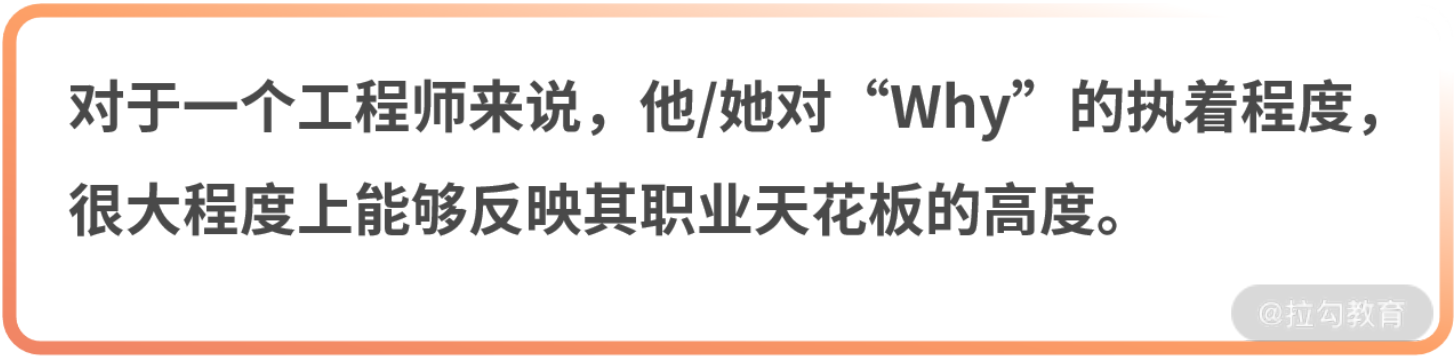
提起 React-Hooks，可能很多人的第一反应，都会是 useState、useEffect、useContext 这些琐碎且繁多的 API。似乎 React-Hooks 就是一坨没有感情的工具性代码，压根没有啥玄妙的东西在里面，那些大厂面试官天天让咱聊 React-Hooks，到底是想听啥呢?

掌握 React-Hooks 的正确姿势

前面我和你聊到过，当我们由浅入深地认知一样新事物的时候，往往需要遵循“Why→What→How”这样的一个认知过程。

在我的读者中，不少人在“What”和“How”这两个环节做得都不错，但是却疏于钻研背后的“Why”。其实这三者是相辅相成、缺一不可的：当我们了解了具体的“What”和“How”之后，往往能够更加具象地回答理论层面“Why”的问题；而我们对“Why”的探索和认知，也必然会反哺到对“What”的理解和对“How”的实践。

这其中，我们尤其不能忽略对“Why”的把控。



React-Hooks 自 React 16.8 以来才真正被推而广之，对我们每一个老 React 开发来说，它都是一个新事物。如果在认知它的过程当中，我们能够遵循“Why→What→How”这样的一个学习法则，并且以此为线索，梳理出属于自己的完整知识链路。那么我相信，面对再刁钻的面试官，你都可以做到心中有数、对答如流。

接下来两个课时，我们就遵循这个学习法则，向 React-Hooks 发起挑战，真正理解它背后的设计动机与工作模式。

React-Hooks 设计动机初探

开篇我们先来聊“Why”。React-Hooks 这个东西比较特别，它是 React 团队在真刀真枪的 React 组件开发实践中，逐渐认知到的一个改进点，这背后其实涉及对**类组件**和**函数组件**两种组件形式的思考和侧重。因此，你首先得知道，什么是类组件、什么是函数组件，并完成对这两种组件形式的辨析。

何谓类组件（Class Component）

所谓类组件，就是基于 ES6 Class 这种写法，通过继承 React.Component 得来的 React 组件。以下是一个典型的类组件：

复制代码

```
class DemoClass extends React.Component {

  // 初始化类组件的 state
  state = {
    text: ""
  };
  // 编写生命周期方法 didMount
  componentDidMount() {
    // 省略业务逻辑
  }
  // 编写自定义的实例方法
  changeText = (newText) => {
    // 更新 state
    this.setState({
      text: newText
    });
  };
  // 编写生命周期方法 render
  render() {
    return (
      <div className="demoClass">
        <p>{this.state.text}</p>
        <button onClick={this.changeText}>点我修改</button>
      </div>
    );
  }
}
```

何谓函数组件/无状态组件（Function Component/Stateless Component）

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”。以下是一个典型的函数组件：

```
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>
  );
}
```

复制代码

函数组件与类组件的对比：无关“优劣”，只谈“不同”

我们先基于上面的两个 Demo，从形态上对两种组件做区分。它们之间肉眼可见的区别就包括但不限于：

- 类组件需要继承 class，函数组件不需要；
- 类组件可以访问生命周期方法，函数组件不能；
- 类组件中可以获取到实例化后的 this，并基于这个 this 做各种各样的事情，而函数组件不可以；
- 类组件中可以定义并维护 state（状态），而函数组件不可以；
- ……

单就我们列出的这几点里面，频繁出现了“类组件可以 xxx，函数组件不可以 xxx”，这是否就意味着类组件比函数组件更好呢?

答案当然是否定的。你可以说，在 React-Hooks 出现之前的世界里，**类组件的能力边界明显强于函数组件**，但要进一步推导“类组件强于函数组件”，未免显得有些牵强。同理，一些文章中一味鼓吹函数组件轻量优雅上手迅速，不久的将来一定会把类组件干没（类组件：我做错了什么？）之类的，更是不可偏听偏信。

当我们讨论这两种组件形式时，**不应怀揣“孰优孰劣”这样的成见，而应该更多地去关注两者的不同，进而把不同的特性与不同的场景做连接**，这样才能求得一个全面的、辩证的认知。

重新理解类组件：包裹在面向对象思想下的“重装战舰”

类组件是面向对象编程思想的一种表征。面向对象是一个老生常谈的概念了，当我们应用面向对象的时候，总是会有意或无意地做这样两件事情。

1. 封装：将一类属性和方法，“聚拢”到一个 Class 里去。
2. 继承：新的 Class 可以通过继承现有 Class，实现对某一类属性和方法的复用。

React 类组件也不例外。我们再次审视一下这个典型的类组件 Case：

复制代码

```
class DemoClass extends React.Component {

  // 初始化类组件的 state
  state = {
    text: ""
  };
  // 编写生命周期方法 didMount
  componentDidMount() {
    // 省略业务逻辑
  }
  // 编写自定义的实例方法
  changeText = (newText) => {
    // 更新 state
    this.setState({
      text: newText
    });
  };
  // 编写生命周期方法 render
  render() {
    return (
      <div className="demoClass">
        <p>{this.state.text}</p>
        <button onClick={this.changeText}>点我修改</button>
      </div>
    );
  }
}
```

不难看出，React 类组件内部预置了相当多的“现成的东西”等着你去调度/定制，state 和生命周期就是这些“现成东西”中的典型。要想得到这些东西，难度也不大，你只需要轻轻地**继承**一个 React.Component 即可。

这种感觉就好像是你不费吹灰之力，就拥有了一辆“重装战舰”，该有的枪炮导弹早已配备整齐，就等你操纵控制台上的一堆开关了。

毋庸置疑，类组件给到开发者的东西是足够多的，但“多”就是“好”吗？其实未必。

把一个人塞进重装战舰里，他就一定能操纵这台战舰吗？如果他没有经过严格的训练，不清楚每一个操作点的内涵，那他极有可能会把炮弹打到友军的营地里去。

React 类组件，也有同样的问题——它提供了多少东西，你就需要学多少东西。假如背不住生命周期，你的组件逻辑顺序大概率会变成一团糟。**“大而全”的背后，是不可忽视的学习成本**。

再想这样一个场景：假如我现在只是需要打死一只蚊子，而不是打掉一个军队。这时候继续开动重装战舰，是不是正应了那句老话——“可以，但没有必要”。这也是类组件的一个不便，**它太重了**，对于解决许多问题来说，编写一个类组件实在是一个过于复杂的姿势。复杂的姿势必然带来高昂的理解成本，这也是我们所不想看到的。

更要命的是，由于开发者编写的逻辑在**封装**后是和组件粘在一起的，这就使得类**组件内部的逻辑难以实现拆分和复用。**如果你想要打破这个僵局，则需要进一步学习更加复杂的设计模式（比如高阶组件、Render Props 等），用更高的学习成本来交换一点点编码的灵活度。

这一切的一切，光是想想就让人头秃。所以说，**类组件固然强大， 但它绝非万能**。

深入理解函数组件：呼应 React 设计思想的“轻巧快艇”

我们再来看这个函数组件的 case：

复制代码

```
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>
  );
}
```

当然啦，要是你以为函数组件的简单是因为它只能承担渲染这一种任务，那可就太小瞧它了。它同样能够承接相对复杂的交互逻辑，像这样：

复制代码

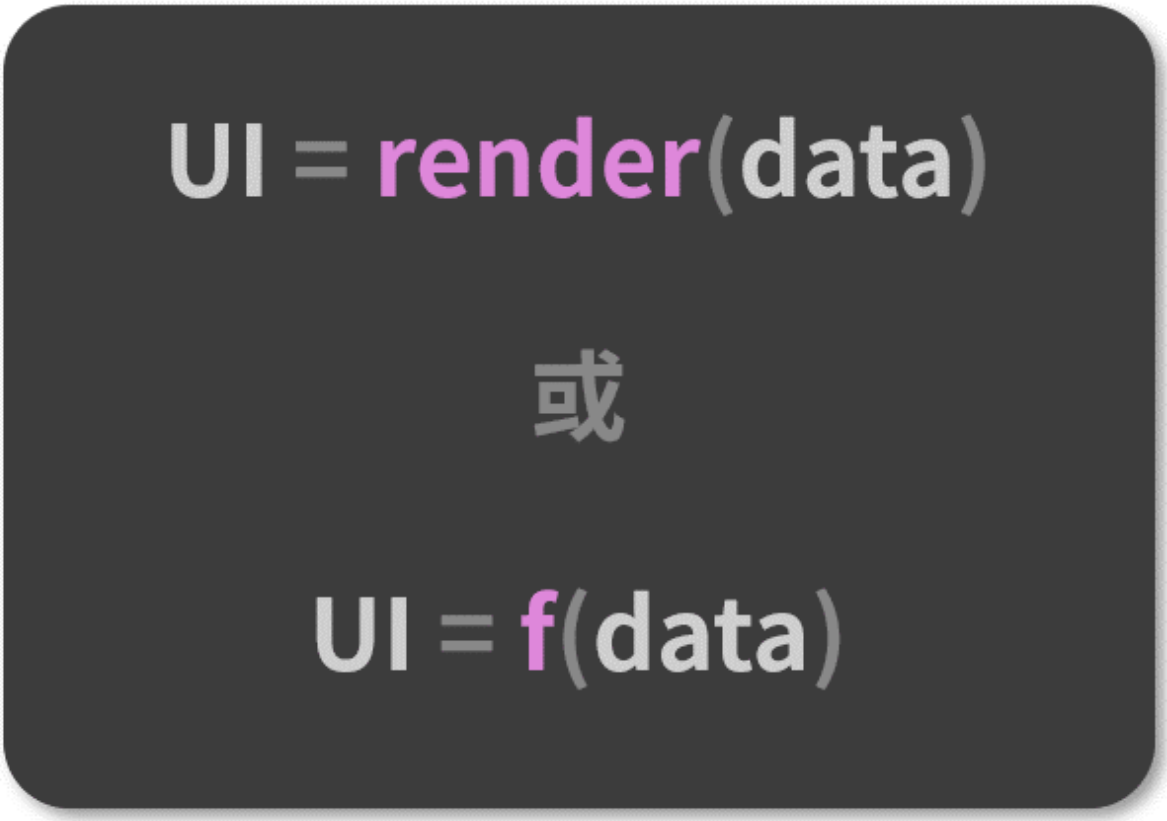
```
function DemoFunction(props) {
  const { text } = props

  const showAlert = ()=> {
    alert(`我接收到的文本是${text}`)
  }

  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>
      <button onClick={showAlert}>点击弹窗</button>
    </div>
  );
}
```

相比于类组件，函数组件肉眼可见的特质自然包括轻量、灵活、易于组织和维护、较低的学习成本等。这些要素毫无疑问是重要的，它们也确实驱动着 React 团队做出改变。但是除此之外，还有一个非常容易被大家忽视、也极少有人能真正理解到的知识点，我在这里要着重讲一下。这个知识点缘起于 React 作者 Dan 早期特意为类组件和函数组件写过的一篇非常棒的对比文章，这篇文章很长，但是通篇都在论证这一句话：

> **函数组件会捕获 render 内部的状态，这是两类组件最大的不同。**

初读这篇文章时，我像文中的作者一样，感慨 JS 闭包机制竟能给到我们这么重要的解决问题的灵感。但在反复思考过后的现在，我更希望引导我的读者们去认知到这样一件事情——**类组件和函数组件之间，纵有千差万别，但最不能够被我们忽视掉的，是心智模式层面的差异**，是面向对象和函数式编程这两套不同的设计思想之间的差异。

说得更具体一点，**函数组件更加契合 React 框架的设计理念**。何出此言？不要忘了这个赫赫有名的 React 公式：



不夸张地说，**React 组件本身的定位就是函数，一个吃进数据、吐出 UI 的函数**。作为开发者，我们编写的是声明式的代码，而 React 框架的主要工作，就是及时地**把声明式的代码转换为命令式的 DOM 操作，把数据层面的描述映射到用户可见的 UI 变化中去**。这就意味着从原则上来讲，**React 的数据应该总是紧紧地和渲染绑定在一起的，而类组件做不到这一点**。

为什么类组件做不到？这里我摘出上述文章中的 Demo，站在一个新的视角来解读一下"**函数组件会捕获 render 内部的状态，这是两类组件最大的不同"**这个结论。首先我们来看这样一个类组件：

复制代码

```
class ProfilePage extends React.Component {
  showMessage = () => {
    alert('Followed ' + this.props.user);
  };
  handleClick = () => {
    setTimeout(this.showMessage, 3000);
  };
  render() {
    return <button onClick={this.handleClick}>Follow</button>;
  }
}
```

这个组件返回的是一个按钮，交互内容也很简单：点击按钮后，过 3s，界面上会弹出"Followed xxx"的文案。类似于我们在微博上点击"关注某人"之后弹出的"已关注"这样的提醒。

看起来好像没啥毛病，但是如果你在这个在线 Demo中尝试点击基于类组件形式编写的 ProfilePage 按钮后 3s 内把用户切换为 Sophie，你就会看到如下图所示的效果：

图源：https://overreacted.io/how-are-function-components-different-from-classes/

明明我们是在 Dan 的主页点击的关注，结果却提示了"Followed Sophie"！

这个现象必然让许多人感到困惑：user 的内容是通过 props 下发的，props 作为不可变值，为什么会从 Dan 变成 Sophie 呢?

因为**虽然 props 本身是不可变的，但 this 却是可变的，this 上的数据是可以被修改的**，this.props 的调用每次都会获取最新的 props，而这正是 React 确保数据实时性的一个重要手段。

多数情况下，在 React 生命周期对执行顺序的调控下，this.props 和 this.state 的变化都能够和预期中的渲染动作保持一致。但在这个案例中，**我们通过 setTimeout 将预期中的渲染推迟了 3s，打破了 this.props 和渲染动作之间的这种时机上的关联**，进而导致渲染时捕获到的是一个错误的、修改后的 this.props。这就是问题的所在。

但如果我们把 ProfilePage 改造为一个像这样的函数组件：

复制代码

```
function ProfilePage(props) {
  const showMessage = () => {
    alert('Followed ' + props.user);
  };
  const handleClick = () => {
    setTimeout(showMessage, 3000);
  };
  return (
    <button onClick={handleClick}>Follow</button>
  );
}
```

事情就会大不一样。

props 会在 ProfilePage 函数执行的一瞬间就被捕获，而 props 本身又是一个不可变值，因此**我们可以充分确保从现在开始，在任何时机下读取到的 props，都是最初捕获到的那个 props**。当父组件传入新的 props 来尝试重新渲染 ProfilePage 时，本质上是基于新的 props 入参发起了一次全新的函数调用，并不会影响上一次调用对上一个 props 的捕获。这样一来，我们便确保了渲染结果确实能够符合预期。

如果你认真阅读了我前面说过的那些话，相信你现在一定也**不仅仅能够充分理解 Dan 所想要表达的“函数组件会捕获 render 内部的状态”**这个结论，而是能够更进一步地意识到这样一件事情：**函数组件真正地把数据和渲染绑定到了一起**。

经过岁月的洗礼，React 团队显然也认识到了，**函数组件是一个更加匹配其设计理念、也更有利于逻辑拆分与重用的组件表达形式**，接下来便开始“用脚投票”，用实际行动支持开发者编写函数式组件。于是，React-Hooks 便应运而生。

Hooks 的本质：一套能够使函数组件更强大、更灵活的“钩子”

React-Hooks 是什么？它是一套能够使函数组件更强大、更灵活的“钩子”。

前面我们已经说过，函数组件比起类组件“少”了很多东西，比如生命周期、对 state 的管理等。这就给函数组件的使用带来了非常多的局限性，导致我们并不能使用函数这种形式，写出一个真正的全功能的组件。

React-Hooks 的出现，就是为了帮助函数组件补齐这些（相对于类组件来说）缺失的能力。

如果说函数组件是一台轻巧的快艇，那么 React-Hooks 就是一个内容丰富的零部件箱。“重装战舰”所预置的那些设备，这个箱子里基本全都有，同时它还不强制你全都要，而是**允许你自由地选择和使用你需要的那些能力**，然后将这些能力以 Hook（钩子）的形式“钩”进你的组件里，从而定制出一个最适合你的“专属战舰”。

总结

行文至此，关于“Why”的研究已经基本到位，对于“What”的认知也已经初见眉目。虽然本课时并没有贴上哪怕一行 React-Hooks 相关的代码，但我相信，你对 React-Hooks 本质的把握已经超越了非常多的 React 开发者。

在下个课时，我们将会和 React-Hooks 面对面交锋，从编码层面上认知“What”，从实践角度理解“How”。相信在课时的最后，你会对本文所讲解的“Why”有更深刻的理解和感悟。