

10-函数组件设计模式：如何应对复杂条件渲染场景？

你好，我是王沛。今天我们来聊聊函数组件中的设计模式。

所谓设计模式，就是**针对特定场景，提供一种公认的最佳实践**。在前面的课程中，我们已经提到了不少模式，比如保证状态的唯一数据源，语义化的拆分复杂组件，等等。熟练掌握这些模式，可以让我们的代码更加简洁直观。

那么今天这节课我会介绍另外两个模式：

1. 一个和 Hooks 相关，用于解决 Hooks 无法在条件语句中执行带来的一些难题；
2. 另一个则是经典的 render props 模式，用于实现 UI 逻辑的重用。

容器模式：实现按条件执行 Hooks

第2讲我们介绍了 Hooks 的一个重要规则，即：**Hooks 必须在顶层作用域调用**，而不能放在条件判断、循环等语句中，同时也不能在可能的 return 语句之后执行。换句话说，Hooks 必须按顺序被执行到。

这个规则存在的原因就在于，**React 需要在函数组件内部维护所用到的 Hooks 的状态**，所以我们无法在条件语句中使用 Hooks，这因而会给我们实现业务逻辑带来一定的局限。

比如说，对于一个对话框组件，通过 visible 属性来控制是否显示。那么在 visible 为 false 的时候，其实不应该执行任何对话框内部的逻辑，因为还没展示在 UI 上。

需要注意，只有在 visible 为true 的时候才应该去执行业务逻辑，展现数据。那么我们期望的代码可能是下面的方式：

```
import { Modal } from "antd";
import useUser from "../09/useUser";

function UserInfoModal({ visible, userId, ...rest }) {
  // 当 visible 为 false 时，不渲染任何内容
  if (!visible) return null;
  // 这一行 Hook 在可能的 return 之后，会报错！
  const { data, loading, error } = useUser(userId);

  return (
    <Modal visible={visible} {...rest}>
      { /* 对话框的内容 */ }
    </Modal>
  );
};
```

可以看到，我们期望在对话框隐藏时通过返回 null 不去渲染任何内容，这个逻辑看上去非常自然直观。

但是呢，它却通不过编译，因为在 return 语句之后使用了 useUser 这个 Hook。所以在你的编辑器配置了 React Hooks 的 ESLint 插件之后，会给出下面的错误提示：

```
useUser(userId);  
(alias) useUser(id: any): {  
  loading: boolean;  
  error: any;  
  data: any;  
}  
import useUser  
  
React Hook "useUser" is called conditionally. React Hooks must be  
called in the exact same order in every component render. (react-  
hooks/rules-of-hooks) eslint  
Quick Fix...
```

可以看到，因为 Hooks 使用规则的存在，使得有时某些逻辑无法直观地实现。换句话说，Hooks 在带来众多好处的同时，也或多或少带来了一些局限。因此，我们需要用一个间接的模式来实现这样的逻辑，可以称之为**容器模式**。

具体做法就是**把条件判断的结果放到两个组件之中，确保真正 render UI 的组件收到的所有属性都是有值的**。

针对刚才我们讲的例子，就可以在 UserInfoModal 外层加一个容器，这样就能实现条件渲染了。实现的代码如下：

```
// 定义一个容器组件用于封装真正的 UserInfoModal  
export default function UserInfoModalWrapper({  
  visible,  
  ...rest, // 使用 rest 获取除了 visible 之外的属性  
}) {  
  // 如果对话框不显示，则不 render 任何内容  
  if (!visible) return null;  
  // 否则真正执行对话框的组件逻辑  
  return <UserInfoModal visible {...rest} />;  
}
```

这样的话，我们就间接实现了按条件去执行 Hooks 的逻辑。

在实际的使用场景中，可能判断条件不止 visible 一个属性，而会是一些属性的组合，来具体决定 render 什么内容。虽然这样的做法不够直观，但其实也能带来一些好处。比如说，在函数组件你会少写一些条件判断语句，并且确保每个组件尽量短小，这样反而更加易读和维护。

在容器模式中我们其实也可以看到，条件的隔离对象是多个子组件，这就意味着它通常用于一些比较大块逻辑的隔离。所以对于一些比较细节的控制，其实还有一种做法，就是**把判断条件放到 Hooks 中去**。

比如上节课的例子，我们需要先发送请求，获得文章信息，从而知道作者的 ID 是什么，这样才能用 useUser 这个 Hook 去获取用户数据。

那么直观的写法是下面这样的：

```
const ArticleView = ({ id }) => {
  const { data: article, loading } = useArticle(id);
  let user = null;
  if (article?.userId) user = useUser(article?.userId).data;
  // 组件其它逻辑
}
```

可以看到，我们需要的 article 这个对象获取到之后，才能去用 useUser 这个 Hook 再去获取用户信息。那么同样的，既然 Hook 不能放到条件语句中，那我们应该如何做呢？

事实上，上一讲的例子已经给出了答案，那就是把条件语句自包含在 Hook 之中。这样当没有传递 userId 给 useUser 这个 Hook 的时候，副作用里实际上什么也不做，比如：

```
function useUser(id) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  useEffect(() => {
    // 当 id 不存在，直接返回，不发送请求
    if (!id) return
    // 获取用户信息的逻辑
  });
}
```

可以看到，在 useEffect 中我们会判断 ID 是否存在。如果不存在，就不发送请求。这样的话，这个 Hook 就可以在组件中无条件使用了。

总体来说，通过这样一个容器模式，我们把原来需要条件运行的 Hooks 拆分成子组件，然后通过一个容器组件来进行实际的条件判断，从而渲染不同的组件，实现按条件渲染的目的。这在一些复杂的场景之下，也能达到拆分复杂度，让每个组件更加精简的目的。

使用 render props 模式重用 UI 逻辑

对于 React 开发而言，如果要挑选一个最重要的设计模式，那一定是 render props。因为它解决了 UI 逻辑的重用问题，不仅适用于 Class 组件，在函数组件的场景下也不可或缺。

鉴于大家日常交流都习惯用这个英文的名字，所以这里我也就不翻译成中文了。顾名思义，render props 就是**把一个 render 函数作为属性传递给某个组件，由这个组件去执行这个函数从而 render 实际的内容。**

在 Class 组件时期，render props 和 HOC（高阶组件）两种模式可以说是进行逻辑重用的两把利器，但是实际上，HOC 的所有场景几乎都可以用 render props 来实现。可以说，**Hooks是逻辑重用的第一选择。**

不过在如今的函数组件情况下，**Hooks** 有一个局限，那就是**只能用作数据逻辑的重用**，而一旦涉及 UI 表现逻辑的重用，就有些力不从心了，而这正是 render props 擅长的地方。所以，**即使有了 Hooks，我们也要**

掌握 render props 这个设计模式的用法。

为了方便你理解 render props 这个模式，我先给你举一个数据逻辑重用的简单例子。这个例子仍然是我们熟悉的计数器。有两个按钮，加一和减一，并将当前值显示在界面上。执行的效果如下图所示：



如果不考虑 UI 的展现，这里要抽象的业务逻辑就是计数逻辑，包括三个部分：

1. count: 当前计数值；
2. increase: 让数值加 1 的方法；
3. decrease: 让数值减 1 的方法。

如果用 render props 模式把这部分逻辑封装起来，那就可以在不同的组件中使用，由使用的组件自行决定 UI 如何展现。下面的代码就是这个计数器的 render props 的实现：

```
import { useState, useCallback } from "react";

function CounterRenderProps({ children }) {
  const [count, setCount] = useState(0);
  const increment = useCallback(() => {
    setCount(count + 1);
  }, [count]);
  const decrement = useCallback(() => {
    setCount(count - 1);
  }, [count]);

  return children({ count, increment, decrement });
}
```

可以看到，我们要把计数逻辑封装到一个自己不 render 任何 UI 的组件中，那么在使用的时候可以用如下的代码：

```
function CounterRenderPropsExample() {
  return (
    <CounterRenderProps>
      {{{ count, increment, decrement }} => {
        return (
          <div>
            <button onClick={decrement}>-</button>
            <span>{count}</span>
            <button onClick={increment}>+</button>
          </div>
        );
      }}
    </CounterRenderProps>
  );
}
```

这里利用了 children 这个特殊属性。也就是组件开始 tag 和结束tag 之间的内容，其实是会作为 children 属性传递给组件。那么在使用的时候，是直接传递了一个函数过去，由实现计数逻辑的组件去调用这个函数，并把相关的三个参数count, increase 和 decrease 传递给这个函数。

当然，我们**完全也可以使用其它的属性名字**，而不是 children。我们只需要**把这个 render 函数作为属性传递给组件**就可以了，这也正是 render props 这个名字的由来。

这个例子演示了纯数据逻辑的重用，也就是重用的业务逻辑自己不产生任何 UI。那么在这种场景下，其实用 Hooks 是更方便的，在第6讲中，我其实已经给过这么计数器的 Hooks 实现的例子，代码如下：

```
import { useState, useCallback } from 'react';

function useCounter() {
  // 定义 count 这个 state 用于保存当前数值
  const [count, setCount] = useState(0);
  // 实现加 1 的操作
  const increment = useCallback(() => setCount(count + 1), [count]);
  // 实现减 1 的操作
  const decrement = useCallback(() => setCount(count - 1), [count]);

  // 将业务逻辑的操作 export 出去供调用者使用
  return { count, increment, decrement };
}
```

很显然，使用 Hooks 的方式是更简洁的。这也是为什么我们经常说 Hooks 能够替代 render props 这个设计模式。但是，需要注意的是，Hooks 仅能替代纯数据逻辑的 render props。如果有 UI 展示的逻辑需要重用，那么我们还是必须借助于 render props 的逻辑，这就是我一再强调必须**要掌握 render props 这种设计模式**的原因。

为了解释这个用法，我给你举一个例子。比如，我们需要显示一个列表，如果超过一定数量，则把多余的部分折叠起来，通过一个弹出框去显示。下面这张图可以比较直观地展示这个需求的实际运行效果：

User Names

Liked by: Kennedy, Lucius, Carlos, and 7 more...

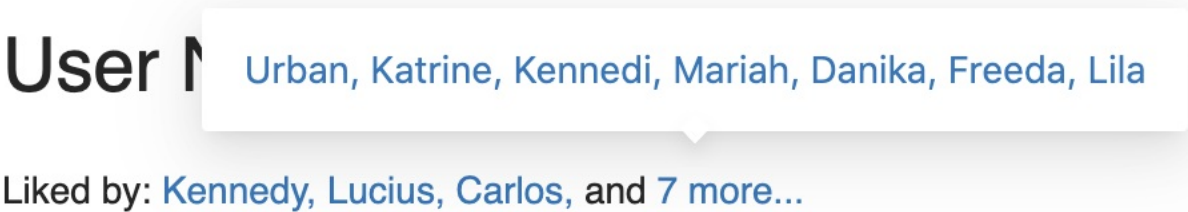
User List

| Name | City | Job Title |
|---------|---------------|-------------------------------|
| Kennedy | North Alec | Chief Mobility Orchestrator |
| Lucius | Littleland | Internal Research Manager |
| Carlos | South Lillian | Lead Configuration Analyst |
| Urban | Shieldshaven | Chief Operations Agent |
| Katrine | South Kyleigh | Legacy Solutions Orchestrator |

and 5 more...

可以看到，这里展示了两个列表。一个只显示用户名，这在一些社交软件的界面上很常见，只显示几个点赞的用户，多余的用一个数字来表示，鼠标移上去则跳转或者显示完整列表。

另外一个表格，但是也只显示前面5个，多余的折叠到“更多...”里面。比如说，对于第一个，鼠标移上去后的效果如下图所示：



我们来分析一下。对于这一类场景，**功能相同的部分**是：数据超过一定数量时，显示一个“更多...”的文字；鼠标移上去则弹出一个框，用于显示其它的数据。

功能不同的部分是：每一个列表项如何渲染，是在使用的时候决定的。

因此，对于这一类具有 UI 逻辑重用需求的场景，我们就无法通过 Hooks 实现，而是需要通过 render props 这个设计模式。

下面这段代码展示了如何实现这个包含了 render props 的 ListWithMore 组件：

```
import { Popover } from "antd";

function ListWithMore({ renderItem, data = [], max }) {
  const elements = data.map((item, index) => renderItem(item, index, data));
  const show = elements.slice(0, max);
  const hide = elements.slice(max);
  return (
    <span className="exp-10-list-with-more">
      {show}
      {hide.length > 0 && (
        <Popover content={<div style={{ maxWidth: 500 }}>{hide}</div>}>
          <span className="more-items-wrapper">
            and{" "}
            <span className="more-items-trigger"> {hide.length} more...</span>
          </span>
        </Popover>
      )}
    </span>
  );
}
```

可以看到，这个组件接收了三个参数，分别是：

1. renderItem：用于接收一个函数，由父组件决定如何渲染一个列表项；
2. data：需要渲染的数据；
3. max：最多显示几条数据。

这样，任何有类似需求的场景就都可以用这个组件去实现了。下面这段代码展示了上面示意图中两个场景的实现代码，你可以体会一下：

```
// 这里用一个示例数据
import data from './data';

function ListWithMoreExample () => {
  return (
    <div className="exp-10-list-with-more">
      <h1>User Names</h1>
      <div className="user-names">
        Liked by:{" "}
        <ListWithMore
          renderItem={(user) => {
            return <span className="user-name">{user.name}</span>;
          }}
          data={data}
          max={3}
        />
      </div>
      <br />
      <br />
      <h1>User List</h1>
      <div className="user-list">
        <div className="user-list-row user-list-row-head">
          <span className="user-name-cell">Name</span>
          <span>City</span>
          <span>Job Title</span>
        </div>
      </div>
    </div>
  );
}
```

```
<ListWithMore
  renderItem={({user}) => {
    return (
      <div className="user-list-row">
        <span className="user-name-cell">{user.name}</span>
        <span>{user.city}</span>
        <span>{user.job}</span>
      </div>
    );
  }}
  data={data}
  max={5}
/>
</div>
</div>
);
};
```

可以看到，代码里使用了两个 ListWithMore 组件，通过 renderItem 这个属性，我们可以自主决定该如何渲染每一个列表项，从而把一部分 UI 逻辑抽象出来，形成一个可复用的逻辑，以简化不同场景的使用。

小结

在今天这节课，我们介绍了两个设计模式。

一个是容器模式，可以实现类似于按条件执行 Hooks 的功能。虽然这是一个间接的方式，但是能够帮助我们更好地做组件逻辑的分离。

第二个则是经典的 render props 模式，虽然它和 Hooks 没有任何关系，但它可以作为 Hooks 的一个补充。在我们需要重用某些 UI 逻辑的时候，提供一个实现方案。

课程中的代码，以及在线运行效果的链接在这里：<https://codesandbox.io/s/react-hooks-course-20vzg>。你可以 fork 后自己动手尝试。

思考题

思考一下你做过的项目，你能想到哪些 render props 模式的使用场景呢？欢迎交流分享。

精选留言：

- Free fall 2021-06-17 21:00:15

做过一个文件预览的功能，根据选中的文件类型，用对应的插件预览 [2赞]

- 知鱼 2021-06-18 15:11:19

后期会有视频吗？

编辑回复2021-06-18 16:27:02

没有啦，知鱼同学～这次王沛老师带来的是专栏，主要是文字+音频的形式哦～对于React Hooks的学习来说完全够用了，老师也讲得很清晰。

- 何用 2021-06-18 08:10:37

UserInfoModalWrapper 考虑 Modal 动画了吗？这种改写不是等价的，会散失动画效果

作者回复2021-06-18 18:34:02

很好的问题。如果考虑动画的场景，那么 visible 从 false -> true 是会默认保留动画；如果是 true -> false，那取决于什么时候设置 visible。比如对于 antd 的 Modal，可以在 afterClose 事件里再去设置 visible 为 false，也就是动画结束后才是真正的 visible=false，那就会保留动画了。所以这属于具体如何封装的实现细节，具体情况具体考虑，总体的模式是不变的。

- Geek_71adef 2021-06-18 05:54:40

对于新加/修改表单(表单字段较多)，这种是不是不适合用 render props，而组件直接引入，传递参数这样更好，这样的理解对么

- 与你. 2021-06-17 22:23:20

学 react 的时候直接学过 hooks，对于这个 render props 好像还是不理解