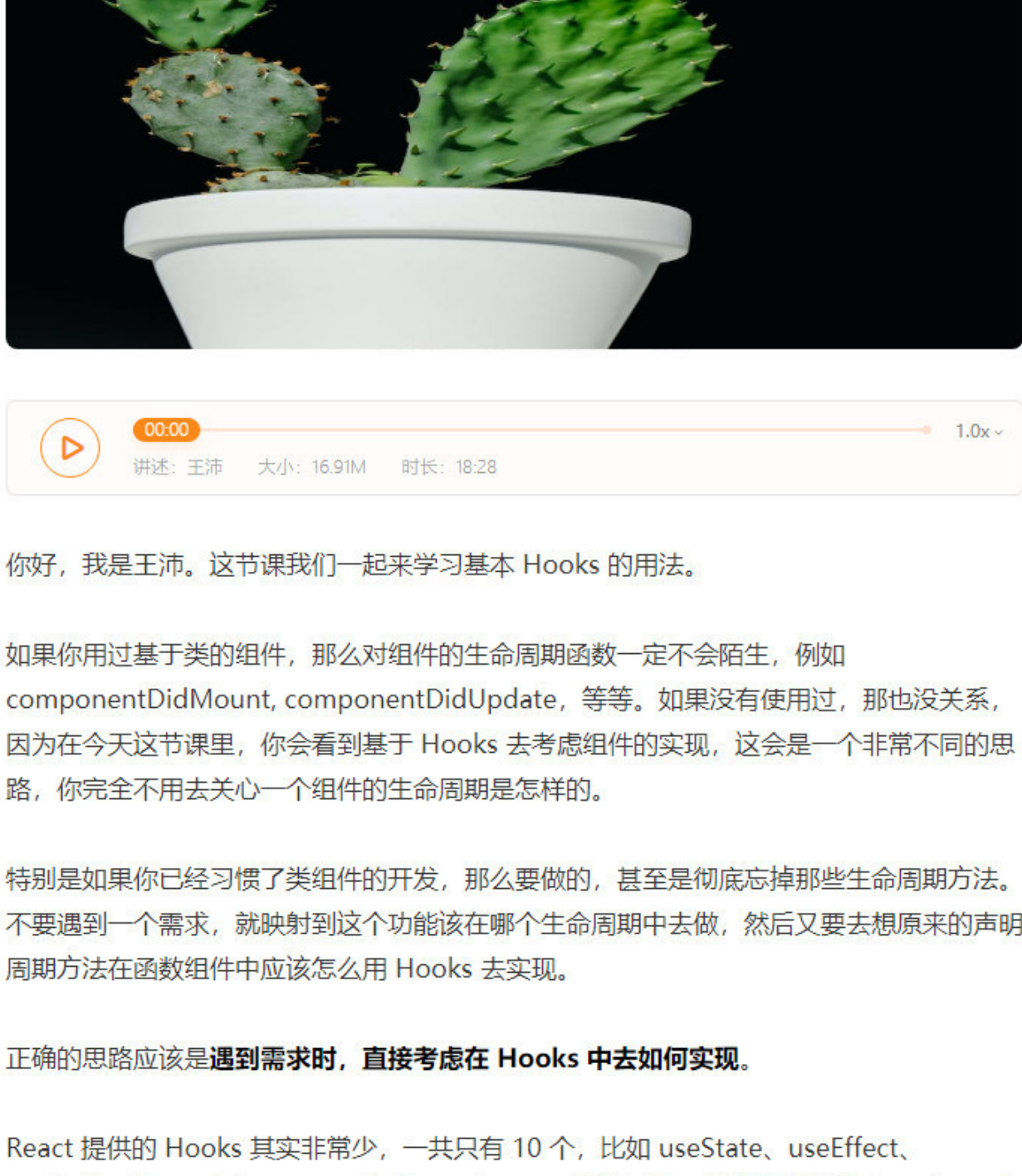


03 | 两个核心Hook：掌握 React 函数组件的开发思路

王沛 2021-05-29



你好，我是王沛。这节课我们一起来学习基本 Hooks 的用法。

如果你用过基于类的组件，那么对组件的生命周期函数一定不会陌生，例如 `componentDidMount`、`componentDidUpdate`，等等。如果没有使用过，那也没关系，因为在今天这节课里，你会看到基于 Hooks 去考虑组件的实现，这将会是一个非常不同的思路，你完全不用去关心一个组件的生命周期是怎样的。

特别是如果你已经习惯了类组件的开发，那么要做的，甚至是彻底忘掉那些生命周期方法。不要遇到一个需求，就映射到这个功能该在哪个生命周期中去做，然后又要想原来的声明周期方法在函数组件中应该怎么用 Hooks 去实现。

正确的思路应该是**遇到需求时，直接考虑在 Hooks 中去如何实现**。

React 提供的 Hooks 其实非常少，一共有 10 个，比如 `useState`、`useEffect`、`useCallback`、`useMemo`、`useRef`、`useContext` 等等。这一讲我们会先学习 `useState` 和 `useEffect` 这两个最为核心的 Hooks。下一讲则会介绍另外四个常用的 Hooks。掌握了这些 Hooks，你就能进行 90% 的 React 开发了。

不过在讲之前我想强调一点，这些 Hooks 的功能其实非常简单，多看看官方文档就可以了。因为这节课的目的，其实是让你学会如何用 Hooks 的思路去进行功能的实现。

useState：让函数组件具有维持状态的能力

在第一讲中，你已经知道了 state 是 React 组件的一个核心机制，那么 `useState` 这个 Hook 就是用来管理 state 的，它可以**让函数组件具有维持状态的能力**。也就是说，在一个函数组件的多次渲染之间，这个 state 是共享的。下面这个例子就显示了 `useState` 的用法：

```
1 import React, { useState } from 'react';
2
3 function Example() {
4   // 创建一个保存 count 的 state，并给初始值 0
5   const [count, setCount] = useState(0);
6
7   return (
8     <div>
9       <p>{count}</p>
10      <button onClick={() => setCount(count + 1)}>
11        +
12      </button>
13    </div>
14  );
15 }
```

在这个例子中，我们声明了一个名为 `count` 的 state，并得到了设置这个 `count` 值的函数 `setCount`。当调用 `setCount` 时，`count` 这个 state 就会被更新，并触发组件的刷新。那么 `useState` 这个 Hook 的用法总结出来就是这样的：

- `useState(initialState)` 的参数 `initialState` 是创建 state 的初始值，它可以是任意类型，比如数字、对象、数组等等。
- `useState()` 的返回值是一个有着两个元素的数组。第一个数组元素用来读取 state 的值，第二个则是用来设置这个 state 的值。在这里要注意的是，state 的变量（例子中的 `count`）是只读的，所以我们必须通过第二个数组元素 `setCount` 来设置它的值。
- 如果要创建多个 state，那么我们就需要多次调用 `useState`。比如要创建多个 state，使用的代码如下：

```
1 // 定义一个年龄的 state，初始值是 42
2 const [age, setAge] = useState(42);
3 // 定义一个水果的 state，初始值是 banana
4 const [fruit, setFruit] = useState('banana');
5 // 定一个数组 state，初始值是包含一个 todo 的数组
6 const [todos, setTodos] = useState([{}: 'Learn Hooks' ]]);
```

从这段代码可以看到，`useState` 是一个非常简单的 Hook，它让你很方便地去创建一个状态，并提供一个特定的方法（比如 `setAge`）来设置这个状态。

如果你之前用过类组件，那么这里的 `useState` 就和类组件中的 `setState` 非常类似。不过两者最大的区别就在于，类组件中的 state 只能有一个，所以我们一般都是把一个对象作为一个 state，然后再通过不同的属性来表示不同的状态。而函数组件中用 `useState` 则可以很容易地创建多个 state，因为它更加语义化。

可以说，state 是 React 组件非常重要的一个机制，那么什么样的值应该保存在 state 中呢？这是日常开发中需要经常思考的问题。通常来说，我们要遵循的一个原则就是：**state 中永远不要保存可以通过计算得到的值**。比如说：

- 从 props 传递过来的值。有时候 props 传递过来的值无法直接使用，而是要通过一定的计算后再在 UI 上展示，比如说排序。那么我们要做的就是每次用的时候，都重新排序一下，或者利用某些 cache 机制，而不是将结果直接放到 state 里。
- 从 URL 中读到的值。比如有时需要读取 URL 中的参数，把它作为组件的一部分状态。那么我们可以在每次需要用的时候从 URL 中读取，而不是读出来直接放到 state 里。
- 从 cookie、localStorage 中读取的值。通常来说，也是每次要用的时候直接去读取，而不是读出来后放到 state 里。

不过，state 虽然便于维护状态，但也有自己的弊端。**一旦组件有自己状态，意味着组件如果重新创建，就需要有恢复状态的过程，这通常会让组件变得更复杂。**

比如一个组件想在服务器端请求获取一个用户列表并显示，如果把读取到的数据放到本地的 state 里，那么每个用到这个组件的地方，就都需要重新获取一遍。

而如果通过一些状态管理框架，去管理所有组件的 state 的话，比如我在第 7 讲会介绍的 Redux，那么组件本身就可以是无状态的了。无状态组件可以成为更纯粹的表现层，没有太多的业务逻辑，从而更易于使用、测试和维护。

useEffect：执行副作用

`useEffect`，顾名思义，用于执行一段副作用。

什么是副作用呢？通常来说，副作用是指一段和当前执行结果无关的代码，比如说要修改函数外部的某个变量，要发起一个请求，等等。也就是说，在函数组件的当次执行过程中，`useEffect` 中代码的执行是不影响渲染出来的 UI 的。

我们先来看一下它的具体用法。`useEffect` 可以接收两个参数，函数签名如下：

```
1 useEffect(callback, dependencies)
```

第一个为要执行的函数 `callback`，第二个是可选的依赖项数组 `dependencies`。其中依赖项是可选的，如果不指定，那么 `callback` 就会在每次函数组件执行完后都执行；如果指定了，那么只有依赖项中的值发生变化的时候，它才会执行。

对应到 Class 组件，那么 `useEffect` 就涵盖了 `componentDidMount`、`componentDidUpdate` 和 `componentWillUnmount` 三个生命周期方法。不过如果你习惯了使用 Class 组件，那千万不要按照把 `useEffect` 对应到某个或者某几个生命周期的方法。你只要记住，**`useEffect` 是每次组件 render 完后判断依赖并执行**就可以了。

举个例子，某个组件用于显示一篇 Blog 文章，那么这个组件会接收一个参数来表示 Blog 的 ID。而当 ID 发生变化时，组件需要发起请求来获取文章内容并展示：

```
1 import React, { useState, useEffect } from "react";
2
3 function BlogView({ id }) {
4   // 设置一个本地 state 用于保存 blog 内容
5   const [blogContent, setBlogContent] = useState(null);
6
7   useEffect(() => {
8     // useEffect 的 callback 要避免直接的 async 函数，需要封装一下
9     const doAsync = async () => {
10       // 当 id 发生变化时，将当前内容清空以保持一致性
11       setBlogContent(null);
12       // 发起请求获取数据
13       const res = await fetch(`/blog-content/${id}`);
14       // 将获取的数据放入 state
15       setBlogContent(await res.text());
16     };
17     doAsync();
18   }, [id]); // 使用 id 作为依赖项，变化时则执行副作用
19
20   // 如果没有 blogContent 则认为是在 loading 状态
21   const isLoading = !blogContent;
22   return <div>{isLoading ? "Loading..." : blogContent}</div>;
23 }
```

这样，我们就利用 `useEffect` 完成了一个简单的数据请求的需求。在这段代码中，我们把 ID 作为依赖项参数，这样就很自然地在 ID 发生变化时，利用 `useEffect` 执行副作用去获取数据。如果在之前的类组件中要完成类似的需求，我们就需要在 `componentDidUpdate` 这个方法里，自己去判断两次 ID 是否发生了变化，如果变了，才去发起请求。这样的话，逻辑上就不如 `useEffect` 来得直观。

`useEffect` 还有两个特殊的用法：**没有依赖项，以及依赖项作为空数组**。我们来具体分析下。

- 没有依赖项，则每次 render 后都会重新执行。例如：

```
1 useEffect(() => {
2   // 每次 render 完一定执行
3   console.log('re-rendered');
4 });
```

- 空数组作为依赖项，则只在首次执行时触发，对应到 Class 组件就是 `componentDidMount`。例如：

```
1 useEffect(() => {
2   // 组件首次渲染时执行，等价于 class 组件中的 componentDidMount
3   console.log('did mount');
4 }, [])
```

除了这些机制之外，`useEffect` 还**允许你返回一个函数，用于在组件销毁的时候做一些清理的操作**。比如移除事件的监听。这个机制就几乎等价于类组件中的 `componentWillUnmount`。举个例子，在组件中，我们需要监听窗口的大小变化，以便做一些布局上的调整：

```
1 // 设置一个 size 的 state 用于保存当前窗口尺寸
2 const [size, setSize] = useState({});
3 useEffect(() => {
4   // 设置一个本地 state 用于保存 blog 内容
5   const handler = () => {
6     setSize(getSize());
7   };
8   // 监听 resize 事件
9   window.addEventListener('resize', handler);
10
11   // 返回一个 callback 在组件销毁时调用
12   return () => {
13     // 移除 resize 事件
14     window.removeEventListener('resize', handler);
15   };
16 }, []);
```

通过这样一个简单的机制，我们能够更好地管理副作用，从而确保组件和副作用的一致性。

总结一下，`useEffect` 让我们能够在下面四种时机去执行一个回调函数产生副作用：

- 每次 render 后执行：不提供第二个依赖项参数。比如 `useEffect(() => {})`。
- 仅第一次 render 后执行：提供一个空数组作为依赖项。比如 `useEffect(() => {}, [])`。
- 第一次以及依赖项发生变化后执行：提供依赖项数组。比如 `useEffect(() => {}, [deps])`。
- 组件 unmount 后执行：返回一个回调函数。比如 `useEffect() => { return () => {}, []}`。

理解 Hooks 的依赖

在 `useEffect` Hook 中你已经看到了依赖项的概念。其实除了在 `useEffect` 中会用到外，在第 5 讲的 `useCallback` 和 `useMemo` 中也会用到。所以接下来我们就先透彻了解它的工作机制，帮助你在实际开发中能够正确地使用。

正如在第 2 讲中介绍的，Hooks 提供了让你监听某个数据变化的能力。这个变化可能会触发组件的刷新，也可能是去创建一个副作用，又或者是刷新一个缓存。那么定义要监听哪些数据变化的机制，其实就是指定 Hooks 的依赖项。

不过需要注意的是，依赖项并不是内置 Hooks 的一个特殊机制，而可以认为是一种设计模式。有类似需求的 Hooks 都可以用这种模式去实现。

那么在定义依赖项时，我们需要注意以下三点：

- 依赖项中定义的变量一定是在回调函数中用到的，否则声明依赖项其实是没有意义的。
- 依赖项一般是一个常量数组，而不是一个变量。因为一般在创建 callback 的时候，你其实非常清楚其中要用到哪些依赖项了。
- React 会使用浅比较来对比依赖项是否发生了变化，所以要特别注意数组或者对象类型。如果你是每次创建一个新对象，即使和之前的值是等价的，也会被认为是依赖项发生了变化。这是一个刚开始使用 Hooks 时很容易导致 Bug 的地方。例如下面的代码：

```
1 function Sample() {
2   // 这里每次组件执行时创建了一个新数组
3   const todos = [{}: 'Learn Hooks'];
4   useEffect(() => {
5     console.log('Todos changed.');
```

代码的原意可能是在 todos 变化的时候去产生一些副作用，但是这里的 todos 变量是在函数内创建的，实际上每次都产生了一个新数组。所以在作为依赖项的时候进行引用的比较，实际上被认为是发生了变化的。

掌握 Hooks 的使用规则

Hooks 本身作为纯粹的 JavaScript 函数，不是通过某个特殊的 API 去创建的，而是直接定义一个函数。它需要在降低学习和使用成本的同时，还需要遵循一定的规则才能正常工作。因而 Hooks 的使用规则包括以下两个：**只能在函数组件的顶级作用域使用；只能在函数组件或者其他 Hooks 中使用。**

Hooks 只能在函数组件的顶级作用域使用

所谓**顶层作用域**，就是 Hooks 不能在循环、条件判断或者嵌套函数内执行，而必须是在顶层。同时 Hooks 在组件的多次渲染之间，必须按顺序被执行。因为在 React 组件内部，其实是维护了一个对应组件的固定 Hooks 执行列表的，以便在多次渲染之间保持 Hooks 的状态，并做对比。

比如说下面的代码是可行的，因为 Hooks 一定会被执行到：

```
1 function MyComp() {
2   const [count, setCount] = useState(0);
3   // ...
4   return <div>{count}</div>;
5 }
6
```

而下面的代码是错误的，因为在某些条件下 Hooks 是不会被执行到的：

```
1 function MyComp() {
2   const [count, setCount] = useState(0);
3   if (count > 10) {
4     // 错误：不能将 Hook 用在条件判断里
5     useEffect(() => {
6       // ...
7     }, [count])
8   }
9
10  // 这里可能提前返回组件渲染结果，后面就不能再用 Hooks 了
11  if (count === 0) {
12    return 'No content';
13  }
14
15  // 错误：不能将 Hook 放在可能的 return 之后
16  const [loading, setLoading] = useState(false);
17
18  //...
19  return <div>{count}</div>
20 }
```

所以 Hooks 的这个规则可以总结为两点：**第一，所有 Hook 必须要被执行到。第二，必须按顺序执行。**

Hooks 只能在函数组件或者其它 Hooks 中使用

Hooks 作为专门为函数组件设计的机制，使用的情況只有两种，一种是在函数组件内，另外一种则是在自定义的 Hooks 里面。

这个规则在函数组件和类组件同时存在的项目中，可能会造成一定的困扰，因为 Hooks 简洁、直观，我们可能都倾向于用 Hooks 来实现逻辑的重用，但是如果一定要在 Class 组件中使用，那应该如何做呢？其实有一个通用的机制，那就是**利用高阶组件的模式，将 Hooks 封装成高阶组件，从而让类组件使用**。

举个例子。我们已经定义了监听窗口大小变化的一个 Hook：`useWindowSize`。那么很容易就可以将其转换为高阶组件：

```
1 import React from 'react';
2 import { useWindowSize } from './withWindowSize';
3
4 export const withWindowSize = (Comp) => {
5   return props => {
6     const windowSize = useWindowSize();
7     return <Comp windowSize={windowSize} {...props} />;
8   };
9 };
```

那么我们就可以通过如下代码来使用这个高阶组件：

```
1 import React from 'react';
2 import { withWindowSize } from './withWindowSize';
3
4 class MyComp {
5   render() {
6     const { windowSize } = this.props;
7     // ...
8   }
9 }
10
11 // 通过 withWindowSize 高阶组件给 MyComp 添加 windowSize 属性
12 export default withWindowSize(MyComp);
13
```

这样，通过 `withWindowSize` 这样一个高阶组件模式，你就可以把 `useWindowSize` 的结果作为属性，传递给需要使用窗口大小的类组件，这样就可以实现在 Class 组件中复用 Hooks 的逻辑了。

使用 ESLint 插件帮助检查 Hooks 的使用

刚才你已经看到了使用 Hooks 的一些特性和要遵循的规则，那么应用到日常的开中，就必须时刻注意不能写错。我总结了一下，包括这么三点：

- 在 `useEffect` 的回调函数中使用的变量，都必须在依赖项中声明；
 - Hooks 不能出现在条件语句或者循环中，也不能出现在 `return` 之后；
 - Hooks 只能在函数组件或者自定义 Hooks 中使用。
- 那么你可能要问，要保证完全遵循规则，看上去好像挺困难的，必须得非常小心，不知道有什么好办法可以帮助掌握吗？贴心的 React 官方为我们提供了一个 ESLint 的插件，**专门用来检查 Hooks 是否正确被使用，它就是 `eslint-plugin-react-hooks`**。
- 通过这个插件，如果发现缺少依赖项定义这样违反规则的情况，就会报一个错误提示（类似于语法错误的提示），方便进行修改，从而避免 Hooks 的错误使用。

使用的方法也很简单。首先，我们通过 `npm` 或者 `yarn` 安装这个插件：

```
1 npm install eslint-plugin-react-hooks --save-dev
```

然后在你的 ESLint 配置文件中加入两个规则：**`rules-of-hooks` 和 `exhaustive-deps`**。如下：

```
1 {
2   "plugins": [
3     // ...
4     "react-hooks"
5   ],
6   "rules": {
7     // ...
8     // 检查 Hooks 的使用规则
9     "react-hooks/rules-of-hooks": "error",
10    // 检查依赖项的声明
11    "react-hooks/exhaustive-deps": "warn"
12  }
13 }
```

要知道，这个插件几乎是 React 函数组件开发必备的工具，能够避免很多可能看上去很奇怪的错误。所以作为开始开发的第一步，一定要安装并配置好这个插件。

小结

在这节课，我们学习了 `useState` 和 `useEffect` 这两个核心 Hooks 的用法，一个用于保存状态，一个用于执行副作用。可以说，**掌握了这两个 Hooks，几乎就能完成大部分的 React 的开发了**。

同时，根据这两个 Hooks 的用法实例，我们还进一步学习了 Hooks 依赖项的含义，以及使用规则。通过定义不同类型的依赖项，你就可以在组件的不同生命周期中去执行不同的逻辑。

要知道，理解了这些机制是一劳永逸的，因为无论是其它的内置 Hooks，还是自定义 Hooks，都是一样的，这将有助于后面 Hooks 的学习。

思考题

- 在 `useEffect` 中如果使用了某些变量，却没有在依赖项中指定，会发生什么呢？
 - 对于这节课中显示的 Blog 文章的例子，我们在 `useEffect` 中使用了 `setBlogContent` 这样一个函数，本质上它也是一个局部变量，那么这个函数需要被作为依赖项吗？为什么？
- 欢迎在评论区分享你的想法和思考，我会和你一起交流讨论！我们下节课再见！