



React Hooks 分享

数据治理产品部 夫间 1835

开始 →

React Hooks 是干啥的?

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

优点  + 

1. 可重用性
2. 可读性
3. 可测试性
4. 不用再考虑 this 的问题啦

少写代码 😊 -> 少加班 😎

导览

1. `useState` 🤖
2. `useEffect` & `useLayoutEffect` 🤖
3. `useRef` & `useImperativeHandle` 🤖
4. `useMemo` & `useCallback` 🤖
5. 自定义 Hooks 🤖

注意事项

- 只在函数组件中调用 Hook
- 不能在循环、条件或嵌套函数中调用 Hooks
- 搭配 eslint 插件使用

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

```
1  // 你的 ESLint 配置
2  {
3    "plugins": [
4      // ...
5      "react-hooks"
6    ],
7    "rules": {
8      // ...
9      "react-hooks/rules-of-hooks": "error", // 检查 Hook 的规则
10     "react-hooks/exhaustive-deps": "warn" // 检查 effect 的依赖
11   }
12 }
```

useState

为函数组件添加状态

```
1  import { useState } from "react";
2  function Counter() {
3    const [count, setCount] = React.useState(0);
4    const increment = () => setCount(count + 1);
5    return <button onClick={increment}>{count}</button>;
6  }
```

```
1  const [state, setState] = useState(initialState);
```

参数: `initialState` 可以是任何值

返回值: `[state, changeStateFunc]` (数组 ?)

useState 使用注意点

1. 当属性为 Object 时，使用不可变数据结构

```
1  const Message = () => {  
2    const [messageObj, setMessage] = useState({ message: "" });  
3    return (  
4      <div>  
5        <input  
6          type="text"  
7          value={messageObj.message}  
8          onChange={(e) => {  
9            messageObj.message = e.target.value;  
10           setMessage(messageObj);  
11          }}  
12        />  
13      </div>  
14    );  
15  };
```

React 使用 `Object.is` 来比较数据

2. `useState` 是异步更新的

React 的 batch update 机制

```
1  function Counter() {  
2    const [count, setCount] = React.useState(0);  
3    const increment = () => {  
4      setCount(count + 1);  
5      // 获取不到最新的 count  
6      console.log(count);  
7    };  
8    return <button onClick={increment}>{count}</button>;  
9  }
```

1. 使用 函数式更新 `prevState => prevState + 1`

2. 使用 useRef

举个例子 🙋🍎

3. Lazy initialization `useState` 参数的惰性初始化

```
1 // 执行一个 IO 操作
2 const [count, setCount] = React.useState(
3   Number(window.localStorage.getItem("count"))
4 );
5
6 const getInitialState = () => Number(window.localStorage.getItem("count"));
7 const [count, setCount] = React.useState(getInitialState);
```

性能优化的一种方式

举个例子 🙋🍎

useEffect & useLayoutEffect

处理副作用的函数，纯函数中和入参无关的处理值（数据获取，创建订阅，清理定时器等）

```
1  useEffect(() => {  
2    effect;  
3    return () => {  
4      cleanup;  
5    };  
6  }, [deps]);
```

1. useEffect 会在每次渲染后都执行吗?
2. React 何时清除 effect？在啥时执行 cleanup 函数

CodeSandBox

useEffect 的执行时机

1. useEffect 会在每次渲染后都执行吗?

是的，默认情况下，它在第一次渲染之后和每次更新之后都会执行。React 保证了每次运行 effect 的同时，DOM 都已经更新完毕。

2. React 何时清除 effect?

React 会在组件卸载的时候执行清除操作。正如之前学到的，effect 在每次渲染的时候都会执行。这就是为什么 React 会在执行当前 effect 之前对上一个 effect 进行清除。

如果我们在 useEffect 的返回函数中使用 state, 我们要确保 deps 里包含它，否则只会取到 init 的值

deps 参数

useEffect 在没有设置第二个参数的时候，会在每次渲染的时候执行其回调

```
1  const Example = () => {  
2    const [count, setCount] = useState(0);  
3  
4    useEffect(() => {  
5      document.title = `You clicked ${count} times`;  
6    });  
7  
8    return (  
9      <div>  
10        <p>You clicked {count} times</p>  
11        <button onClick={() => setCount(count + 1)}>Click me</button>  
12      </div>  
13    );  
14  };
```

useEffect 有第二个参数，称为依赖数组，只有当依赖数组内的元素发生变化的时候，才会执行 useEffect 的回调。这么做就能够优化 effect 执行的次数。

deps

当数组元素类型是基本数据类型的时候可以起作用，但是对于复杂的数据类型：对象、数组、函数来说，React 会用 引用比较(`Object.is`) 来对比前后是否有不同。检查当前渲染下的这个对象和上一次渲染下的对象的内存地址是否一致。

```
1  import React, { useState, useEffect } from "react";
2  import { getPlayers } from "../api";
3  import Players from "../components/Players";
4
5  // 传入 team 参数，但是我们没法保证传入的 team 属性的地址是一致的
6  const Team = ({ team }) => {
7    const [players, setPlayers] = useState([]);
8
9    useEffect(() => {
10      if (team.active) {
11        getPlayers(team.id).then(setPlayers);
12      }
13      // 1. team.id, team.active
14    }, [team]);
15
16    return <Players team={team} players={players} />;
17  };
```

解决方法：

1. 使用 team 对象里的一些属性，而不是使用整个对象
2. 在组件内部创建对象也行

如果在组件内部创建对象的同时，还使用整个对象呢？

1. 使用 useMemo 来缓存变量

useLayoutEffect

99% 的情况下使用 useEffect, 两者的 api 相同, 但是 useEffect 不适合执行修改 dom 的工作

差异 useEffect 是异步执行的, 而 useLayoutEffect 是同步执行的。

useEffect:

1. 触发渲染
2. React 把 vdom -> dom
3. 屏幕更新
4. useEffect 运行

useLayoutEffect:

1. 触发渲染
2. React 把 vdom -> dom
3. useLayoutEffect 运行
4. 屏幕更新

useRef & useImperativeHandle

Refs 提供了一种方式，允许我们访问 DOM 节点或在 render 方法中创建的 React 元素。

```
1  const ref = useRef(initialValue);  
2  // ref: { current: initialValue }
```

- 可以保存任何值
- 与直接在组件内部声明的 { current: '' } 对象的区别？

```
1  const ref = useRef("");  
2  const ref = { current: "" };
```

- 不会触发组件的重新渲染 (尽量不要在 UI 中使用，最好把改动动作放到 useState 前)

使用的地方

1. 在 Hooks 中作为一个全局变量使用。
2. 管理焦点（受控组件），文本选择或媒体播放
3. 触发强制动画
4. 控制 dom 节点，控制子组件(配合 useImperativeHandle)

useImperativeHandle

函数组件没有实例

```
1 useImperativeHandle(ref(父组件通过 ref 定义的引用变量), func (子组件想要暴露给父组件的方法), [deps]);
```

函数式组件是没有实例的，所以我们不能直接通过 `ref` 来调用子组件的方法。`useImperativeHandle` 可以让父组件获取并执行子组件内某些自定义函数(方法)。

本质上其实是子组件将自己内部的函数(方法)通过 `useImperativeHandle` 添加到父组件中 `useRef` 定义的对象中。

使用流程

1. `useRef` 创建引用变量
2. `React.forwardRef` 将引用变量传递给子组件
3. `useImperativeHandle` 将子组件内定义的函数作为属性，添加到父组件中的 `ref` 对象上。

[查看例子](#)

useMemo & useCallback

```
1  export default function App() {
2    const [count, setCount] = React.useState(0);
3    const value = { name: 1 }; // 声明变量 value
4
5    React.useEffect(() => {
6      setCount(Math.random()); // 修改 count
7      alert("render");
8    }, [value]); // value 作为更新依赖
9    return (
10     <div className="App">
11       <h1>Hello CodeSandbox</h1>
12       <h2>Edit to see some magic happen!</h2>
13     </div>
14   );
15 }
```

App 渲染几次，value 被定义几次， alert 会被弹出几次？

无限循环，全都无限次。 组件渲染 → 创建一个新的 value -> useEffect 执行 → setCount 触发循环 → 组件渲染 → 创建一个新的 value -> useEffect 执行 → setCount 触发循环...

useMemo & useCallback

解决方法:

```
1  export default function App() {
2    const [count, setCount] = React.useState(0);
3    const value = React.useMemo(() => {
4      // 使用 useMemo 来缓存 value
5      return { name: 1 };
6    }, []);
7
8    React.useEffect(() => {
9      setCount(Math.random());
10     alert("render");
11   }, [value]);
12   return (
13     <div className="App">
14       <h1>Hello CodeSandbox</h1>
15       <h2>Edit to see some magic happen!</h2>
16     </div>
17   );
18 }
```

组件渲染 → 创建一个新的 value -> useEffect 执行 → setCount 触发循环 → 组件渲染 → 对比 Value 和前一次的引用地址一致 -> 结束

useMemo & useCallback

useMemo 的意思就是：不要每次渲染都重新定义，而是我让你重新定义的时候再重新定义(第二个参数，依赖列表)。大家看到这里的依赖列表是空的，是因为 useMemo 里的回调函数确实没用到啥变量，如果有变量的话大家的 IDE 就会提醒加上依赖了。

这就是使用 useMemo 的原理，useMemo 适用于所有类型的值，加入这个值恰好是函数，那么用 useCallback 也可以。也就是说，useCallback 是一种特殊的 useMemo。

在这里再粗暴地给大家总结一下日常使用的场景：

如果你定义了一个变量，满足下面的条件就最好用 useMemo 或 useCallback 给包裹住：

1. 它不是状态，也就是说，不是用 useState 定义的(redux 中的状态实际上也是用 useState 定义的)
2. 它不是基本类型
3. 它会被放在 useEffect 的依赖列表里 || 自定义 hook 的返回值

useCallback

1. 使用例子 1 useCallback 用来作为 Effect 依赖列表的缓存

```
1  const fetchData = useCallback(() => {
2    fetchAPI(a, b);
3  }, [a, b]);
```

```
1  useEffect(() => {
2    fetchData();
3  }, [fetchData]);
```

2. 使用例子 2 useCallback + React.memo 当父组件传给子组件方法时

```
1  const DemoUseCallback = ({ id }) => {
2    const [number, setNumber] = useState(1);
3    /* 此时usecallback的第一参数 (sonName)=>{ console.log
4    const getInfo = useCallback(
5      (sonName) => {
6        console.log(sonName);
7      },
8      [id]
9    );
10   return (
11     <div>
12       { /* 点击按钮触发父组件更新，但是子组件没有更新 */ }
13       <button onClick={() => setNumber(number + 1)}>
14       <DemoChildren getInfo={getInfo} />
15     </div>
```

```
1  const DemoChildren = React.memo((props) => {
2    /* 只有初始化的时候打印了 子组件更新 */
3    console.log("子组件更新");
4    useEffect(() => {
5      props.getInfo("子组件");
6    }, [props.getInfo]);
7    return <div>子组件</div>;
8  });
```

自定义 Hooks

通过自定义 Hook，可以将组件逻辑提取到可重用的函数中。 注意点：状态都是在 hooks 中自己维护

例子

```
1  const [form] = Form.useForm(); // dtd 2.0 中的表格交互
```

userCounter Hooks: ``const [count, controlCount] = useCounter(10);``

useModal Hooks(Dom Hooks): ``const [modal, toggleModal] = useModal();``

更多

优秀的第三方自定义 hooks 库

- [awesome-xxx 系列](#)
- [ahooks 阿里沉淀的 Hooks 库](#)

优秀的 Hooks 资料

- [React 官方文档](#)
- [Dan Abramov 的博客](#)

优秀的 Hooks 使用方法

- [Antd Table 的源码](#)

作业