05 React Hooks

常用内置 Hooks

useState

概述

1. 解释：State, A component’s memory。
2. 作用：**state 的变化会触发组件更新，从而 rerender 组件**。
3. 语法

```
import {useState} from "react"
const [state, setState] = useState(initialState);
```

4. 使用原则：**当一个变量用于 JSX 中显示，才建议使用 `useState` 来管理**，否则使用 `useRef`。
5. state vs. Props
   a. state 是组件内部的状态信息
   b. props 是父组件传递过来的信息

特点

异步更新

```
const [count, setCount] = useState<number>(0);
const add = () => {
    setCount(count + 1);
    console.log('current count is ', count); // 3
    console.log('supposed count is ', count + 1); // 4
};
```

可能会被合并

```
const [count, setCount] = useState<number>(0);
const add = () => {
```

```
    // 可能会被合并
    // 参数为非函数时，会合并
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
    // 参数为函数时，不会合并，函数参数为当前 state，返回值为更新后的 state。
    // 一系列的函数会被依次调用，因此更新不会被合并。
    setCount(prevCount => prevCount + 1);
    setCount(prevCount => prevCount + 1);
    setCount(prevCount => prevCount + 1);
    setCount(prevCount => prevCount + 1);
    setCount(prevCount => prevCount + 1);
};
```

不可变数据

state 是不可变数据：`setState` **函数要传入一个新的值，不能修改 state 的值**。即 state 和传入的参数引用不能相同。理解 state 为 `read-only`！

```
const [randArr, setRandArr] = useState<number[]>([]);
const create = () => {
    setRandArr([...randArr, Number((Math.random() * 100).toFixed(0))]);
};
```

```
const [userInfo, setUserInfo] = useState<UserInfo>({ username: 'yiTu', luckyIndex: 0 });
const changeIndex = () => {
    setUserInfo({ ...userInfo, luckyIndex: Number((Math.random() * 100).toFixed(0)) });
};
```

使用 Immer 更新不可变数据

Step 1. 下载 Immer。

```
npm install immer --save
```

Step 2. 引入 Immer。

```
1 import produce from "immer"
```

Step 3. React + Immer 更新不可变数据 state。

- 语法 `produce(recipe: (draftState) => void): nextState`

  > 注意，`recipe` 函数通常不会返回任何内容。

- 示例

```
1 const [randArr, setRandArr] = useState<number[]>([]);
2 const create = () => {
3     setRandArr(
4         produce(draft => {
5             draft.push(Number((Math.random() * 100).toFixed(0)));
6         })
7     );
8 };
```

```
1 const [userInfo, setUserInfo] = useState<UserInfo>({ username: 'yiTu',
  luckyIndex: 0 });
2 const changeIndex = () => {
3     setUserInfo(
4         produce(draft => {
5             draft.luckyIndex = Number((Math.random() * 100).toFixed(0));
6         })
7     );
8 };
```

Vscode 插件：Code Spell Checker

该插件用于避免单词的拼写错误。

useEffect

概述

1. 解释：Effect，指的是那些**在组件渲染过程中，除了计算输出值（JSX）之外的操作**，例如数据获取、订阅事件、修改外部状态等可能影响组件之外的操作。
2. 作用：useEffect 允许组件在**渲染完成时**，或者在**某个 state 变化时**执行某些副作用，如 ajax 加载数据等。

3. 语法

```
import {useEffect} from "react"
useEffect(() => {
    /* 组件挂载或依赖项更新时，这里执行副作用 */
    return () => {
        /* 组件卸载时，这里清理副作用 */
    }
},[])
// useEffect 的第二个参数，即依赖项
// 第二个参数为空 => 组件的每次渲染都会执行副作用，表示依赖所有 props 和 state
// 第二个参数为 [] => 组件只会在第一次渲染执行副作用，表示谁都不依赖
// 第二个参数为 [state, props] => 除了首次渲染，组件还会在指定 state 或 props 变化时执行副作用
```

说明

- 从 React18 开始，useEffect **在开发环境下会执行两次**，用于模拟组件**创建-销毁-再创建**的流程，及早暴露问题。**在生产环境下执行一次**。
- 组件销毁时一定要**取消定时任务和解绑 DOM 事件！！！**

其他内置 Hooks

useRef

1. 语法 `const xxxRef = useRef(initValue)`

   > 可以通过 `xxxRef.current` 获取到 DOM 节点或变量值

2. 作用

   a. 一般用于**操作 DOM**

```
import {useRef, FC} from "react"

const Demo: FC = () => {
    const inputRef = useRef<HTMLInputElement>(null);

    function selectInput(){
        const inputElement = inputRef.current;
        if(inputElement) inputElement.select();
    }

    return (
```

```
        <>
            <input ref={inputRef} defaultValue="hello world" />
            <button onClick={selectInput}> 选中 input </button>
        </>
    )
}

export default Demo;
```

b. 也可传入**普通 JS 变量**，但是其**更新不会触发 rerender**

```
import {useRef, FC} from "react"

const Demo: FC = () => {
    const nameRef = useRef<string>("yiTu");

    function changeName(){
        nameRef.current = "yiTuChuan";
    }

    return (
        <>
            <p>name {nameRef.current}</p>
            <div>
                <button onClick={changeName}>change name</button>
            </div>
        </>
    )
}

export default Demo;
```

useMemo

1. 语法：`const xxx = useMemo(()=>{return computedValue}, [state, props]);`
2. 作用：用于**缓存数据，只有在依赖项更新时才重新计算**，解决了函数组件每次 state 更新都会重新执行函数的弊端。对于计算量较大的场景，有利于**性能提高**。

useCallback

1. 语法：`const xxx = useCallback(fn, [state, props])`
2. 作用：与 useMemo 类似，useCallback 用于**缓存函数**。

自定义 Hooks

自定义 Hooks 可以**抽离公共逻辑，复用到多个组件中**。自定义的 Hooks 存放在 `src/hooks` 文件夹中。

1. 示例一：修改网页标题

```ts
// useTitle.ts
import { useEffect } from 'react';

const useTitle = (title: string) => {
  useEffect(() => {
    document.title = title;
  }, []);
};

export default useTitle;
```

2. 示例二：获取鼠标位置

```ts
// useMouse.ts
import { useCallback, useEffect, useState } from 'react';

const useMouse = () => {
  const [x, setX] = useState(0);
  const [y, setY] = useState(0);

  const mouseMoveHandler = useCallback((event: MouseEvent) => {
    setX(event.clientX);
    setY(event.clientY);
  }, []);

  useEffect(() => {
    document.addEventListener('mousemove', mouseMoveHandler);
    return () => {
      document.removeEventListener('mousemove', mouseMoveHandler);
    };
  }, []);

  return { x, y };
};

export default useMouse;
```

3. 示例三：异步获取信息

```ts
// useGetInfo.ts
import { useEffect, useState } from 'react';

const useGetInfo = () => {
  const [isLoading, setIsLoading] = useState(true);
  const [info, setInfo] = useState('');

  useEffect(() => {
    getInfo().then(info => {
      setIsLoading(false);
      setInfo(info);
    });
  }, []);

  return { isLoading, info };
};

export default useGetInfo;

const getInfo: () => Promise<string> = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(Date.now().toString());
    }, 1500);
  });
};
```

第三方 Hooks

第三方 Hooks 有利于**提高开发效率**，常用的有

- 国内流行的 ahooks

  > 通过 `npm install ahooks --save` 安装

- 国外流行的 react-use

Hooks 使用规则

1. 命名规则：Hook 必须以 `useXxx` 的格式命名。
2. 调用位置：**组件**内部或**其他 Hook** 内部。
3. 调用顺序：Hook 在每次渲染时必须按照相同的顺序被调用，这就要求，

a. Hook 必须是组件**第一层代码**

b. Hook **不可放在 if 等条件语句中**

> 注意：如果 Hook 前边有 return，也算是一种条件

c. Hook **不可放在 for 等循环语句中**

闭包陷阱

1. 闭包陷阱：在使用异步函数（如 `setTimeout` 或 `setInterval`）时，它们可能会捕获到旧的 state，而不是当前最新的 state。这是因为异步函数在其回调执行时，访问的是创建该函数时的闭包环境中的变量。

```tsx
import { FC, useEffect, useState } from 'react';

const Demo: FC = () => {
  const [count, setCount] = useState(0);

  const add = () => {
    setCount(count + 1);
  };

  const alertCount = () => {
    let timer: ReturnType<typeof setTimeout> | null = null;
    return () => {
      if (timer) clearTimeout(timer);
      timer = setTimeout(() => {
        alert(count);
      }, 3000);
    };
  };

  return (
    <>
      <p>闭包陷阱</p>
      <div>
        <span>{count}</span>
        <button onClick={add}>add count</button>
        <button onClick={alertCount()}>alert count</button>
      </div>
    </>
  );
};

export default Demo;
```

2. 解决方式：可以使用 `useRef` 来解决这个问题。每次 state 更新时，及时更新 `ref.current`。由于异步函数维护的是 `useRef` 返回的引用对象，而这个对象在组件的整个生命周期中保持不变，因此引用对象的 `current` 属性始终保持最新的值。这样，异步函数在执行时，访问到的 `ref.current` 就是最新的 state 值。

```tsx
import { FC, useEffect, useRef, useState } from 'react';

const Demo: FC = () => {
  const [count, setCount] = useState(0);
  const countRef = useRef(0);

  useEffect(() => {
    countRef.current = count;
  }, [count]);

  const add = () => {
    setCount(count + 1);
  };

  const alertCount = () => {
    let timer: ReturnType<typeof setTimeout> | null = null;
    return () => {
      if (timer) clearTimeout(timer);
      timer = setTimeout(() => {
        alert(countRef.current);
      }, 3000);
    };
  };

  return (
    <>
      <p>闭包陷阱</p>
      <div>
        <span>{count}</span>
        <button onClick={add}>add count</button>
        <button onClick={alertCount()}>alert count</button>
      </div>
    </>
  );
};

export default Demo;
```