React hooks分享

1.前言

React 是主流的前端框架，v16.8 版本引入了全新的 API，叫做 React Hooks，颠覆了以前的用法。
Hook不包含任何破坏性改动（100%向后兼容），并且没有计划从react中移除class。Hook不会影响你对React概念的理解。恰恰相反，Hook为已知的React概念提供了更直接的API：props,state,context,refs以及生命周期。 本文就谈谈我对hook的理解，以及简单介绍它的用法。

2.类组件和函数组件

1.类组件

```javascript 复制代码
import React, { Component } from "react";

export default class Button extends Component {

    constructor() {

        super();

        this.state = { buttonText: "Click me, please"};

        this.handleClick = this.handleClick.bind(this);
     }

handleClick() {

    this.setState(() => {

         return { buttonText: "Thanks, been clicked!"};

    });

  }
```

```
render() {

    const { buttonText } = this.state;

    return <button onClick={this.handleClick}>{buttonText}</button>;

    }
}
```

类组件的缺点：

- 组件类引入了复杂的编程模式，比如 render、props和高阶组件。
- 业务逻辑分散在组件的各个方法之中，导致重复逻辑或关联逻辑。

2.函数组件

组件的最佳写法应该是函数，而不是类。React早就支持函数组件，下面就是一个例子。

```
javascript 复制代码
function Welcome(props) {

    return <h1>Hello, {props.name}</h1>;

}
```

这种写法有重大限制，必须是纯函数，不能包含状态，也不支持生命周期方法，因此无法取代类。

什么是纯函数组件？所谓纯函数，它是这样一种函数：**即相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用**

什么是副作用？副作用（Side Effect）是指一个 function 做了和本身运算返回值无关的事，比如：修改了全局变量、修改了传入的参数、甚至是 console.log()，所以 ajax 操作，修改 dom 都是算作副作用

3.常用的钩子

React Hooks的意思是，组件尽量写成纯函数，如果需要外部功能和副作用，就用钩子把外部代码“钩”进来。React Hooks就是那些钩子。

- useState（）
- useEffect（）
- useReducer （）
- useContext（）
- useRef ()
- useMemo （）
- useCallback（）

useState():状态钩子

useState()用于为函数组件引入状态（state）。纯函数不能有状态，所以把状态放在钩子里面。

```javascript 复制代码
import React, { useState } from 'react';

function Example() {

    // 声明一个叫 “count” 的 state 变量。

    const [count, setCount] = useState(0);

    return (

      <div>

        <p>You clicked {count} times</p>

        <button onClick={() => setCount(count + 1)}>Click me</button>

    </div>

);

}
```

useEffect（）：副作用钩子

useEffect（）用来引入具有副作用的操作，最常见的就是向服务器请求数据。以前，放在componentDidMount里面的代码，现在可以放在useEffect()。

```scss 复制代码
useEffect(() => { Async Action },[dependencies])
```

上面用法中，`useEffect()`接受两个参数。第一个参数是一个函数，异步操作的代码放在里面。第二个参数是一个数组，用于给出 Effect 的依赖项，只要这个数组发生变化，`useEffect()`就会执行。第二个参数可以省略，这时每次组件渲染时，就会执行`useEffect()`。

```javascript 复制代码
import React , { useState , useEffect } from "react";

function DemoUseEffect () {

    const [count,setCount] = useState(0);

    const [num,setNum] = useState(0);

    useEffect(()=>{ document.title = You clicked ${count} times; }); //state发生变化的时候都会执行

    useEffect(()=>{ setNum(()=> num + 1);},[]) //只有页面第一进来的时候执行。

    return (
        <div>
            <div>count:{count}</div>
            <div>num:{num}</div>
            <button onClick={()=>setCount(count + 1)}>click</button>
        </div>)
 }
```

useReducer():action钩子

React 本身不提供状态管理功能，通常需要使用外部库。这方面最常用的库是 Redux。

Redux 的核心概念是，组件发出 action 与状态管理器通信。状态管理器收到 action 以后，使用 Reducer 函数算出新的状态，Reducer 函数的形式是`(state, action) => newState`。

`useReducers()`钩子用来引入 Reducer 功能。

下面是`useReducer()`的基本用法，它接受 Reducer 函数和状态的初始值作为参数，返回一个数组。数组的第一个成员是状态的当前值，第二个成员是发送 action 的`dispatch`函数。

```scss 复制代码
const [state, dispatch] = useReducer(reducer, initialState);
```

举例

```javascript
import React, { useReducer } from 'react';

function reducer(state,action){

    switch (action.type){

        case  'add':

            return {count:state.count+ 1 }

        case 'reduce':

           return {count:state.count- 1 }

         case 'reset':

            return {count:action.initialCount}

        default :

            return state

    }

}

export default function DemoUseReducer ({initialCount}) {

    const [state,dispatch] = useReducer(reducer,initialCount)

     return (

            <>

                <div>count:{state.count}</div>

                <button onClick={()=>{dispatch({type:'add'})}}>+</button>

                <button onClick={()=>{dispatch({type:'reduce'})}}>-</button>

                <button onClick={()=>{dispatch({type:'reset',initialCount})}}>reset</button>

            </>

)}
```

useContext():共享状态钩子

如果需要在组件之间共享状态，可以使用useConttext()

一般用useReducer+useContent 来实现redux。

举例：

```javascript
import React, { useReducer } from  'react';

//第一步创建myContent

const myContent = React.createContext();

    function reducer(state,action){

    switch (action.type){

        case 'add':

            return {count:state.count+ 1}

        case 'reduce' :

            return {count:state.count- 1 }

        case 'reset' :

            return {count:action.playload}

        default :

            return state

    }

}

//第二步利用myCotent提供的Provider来给其子组件绑定value

function ContextProvider (props) {

    const [state,dispatch] = useReducer(reducer,{count: 1 })

      return (

        <myContent.Provider value={{state,dispatch}}>

            {props.children}
```

```
        </myContent.Provider>

)}
export {myContent,reducer,ContextProvider}
```

然后在子组件中获取外层组件的值

```javascript 复制代码
import React, { useContext } from  'react';

import {myContent} from './useContextDemo.js'

function ContentChildren () {

    //第三步，通过useContent来接受外层传递过来的value

    const {state,dispatch} = useContext(myContent);

       return (

            <>`

            <div>count:{state.count}</div>

            <button onClick={()=>{dispatch({type:'add'})}}>+</button>

            <button onClick={()=>{dispatch({type:'reduce'})}}>-</button>

            <button onClick={()=>{dispatch({type:'reset',playload: 1})}}>reset</button>

            </>

)}
export  default ContentChildren
```

useRef()

- `useRef` 返回一个可变的 ref 对象，其 `.current` 属性被初始化为传入的参数（`initialValue`）。
- 返回的 ref 对象在组件的整个生命周期内保持不变。
- 当更新 current 值时并不会 re-render，这是与 useState 不同的地方
- 更新useRef 是 side effect (副作用)，所以一般写在 useEffect 或 event handler 里

```ini 复制代码
const refContainer = useRef(initialValue);
```

举例1：

```javascript 复制代码
import React, { useRef } from  'react';

function DemoUseRef () {

        const inputEl = useRef();

        const focus = ()=>{ f.current.focus();}

return (

        <div>

            <input type="text" ref={inputEl}></input>

            <button onClick={()=>{ focus() }}>focus</button>

        </div>
        )
}

export default DemoUseRef
```

小结：通过useRef定义个inputEl变量，在input 元素上定义ref={inputEl},这样通过inputEl.current就可以获取到input Dom 元素，选中则调用下focus函数即可

举例2:

```javascript 复制代码
import React, { useRef } from "react";

// 定义一个全局变量

let like = 0;

const LikeButton: React.FC = () => {

        let likeRef = useRef(0);

        function handleAlertClick() {

        setTimeout(() => {

        alert(`you clicked on ${like}`);
```

```
        alert(`you clicked on ${likeRef.current}`);
        }, 3000);
}
return (
        <p>
            <button
            onClick={() => { like = ++like; likeRef.current = likeRef.current + 1;}}>
            点赞
            </button>
            <button onClick={handleAlertClick}>Alert</button>
        </p>
);
};
export default LikeButton;
```

小结：

- useRef 是定义在实例基础上的，如果代码中有多个相同的组件，每个组件的 ref 只跟组件本身有关，跟其他组件的 ref 没有关系。
- 组件前定义的 global 变量，是属于全局的。如果代码中有多个相同的组件，那这个 global 变量在全局是同一个，他们会互相影响。

useMemo()

```scss
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```


返回一个memoized值。把"创建"函数和依懒项数组作为参数传入useMemo，它仅会在某个依懒项改变时才重新计算memoized值。这种优化有助于避免在每次渲染时都进行高开销的计

算。记住，传入useMemo的函数会在渲染期间执行。请不要在这个函数内部执行与渲染无关的操作，诸如副作用这类的操作属于useEffect的适用范畴，而不是useMemo。

举例：

```javascript 复制代码
import React, { useMemo, useState } from  'react';

import { Button, Input } from  'antd';

function DemoUseMemo() {

    const [count, setCount] = useState( 1 );

    const [val, setValue] = useState('');

    const expensive = useMemo(() => {

        console.log('compute');

        let sum = 0;

        for(let i = 0; i < count * 100; i++) {

            sum += i;

        }

        return sum;

    }, [count]);


    return
        <div>

            <h4>{count}-{expensive}</h4>

            <div>{val}</div>

            <div>

            <Button className='m-b-10' type="primary" onClick={() => setCount(count + 1 )}>conut++</

            <Input  value={val} onChange={event => setValue(event.target.value)} />

            </div>

        </div>;
```

```
}


export default DemoUseMemo
```

useCallback()

```scss
const memoizedCallback = useCallback(() => {doSomething(a, b);},[a, b]);
```

复制代码

返回一个memoized回调函数。

把内联回调函数及依懒项数组作为参数传入useCallback,它将返回该回调函数的memoized版本，该回调函数仅在某个依懒项改变时才会更新。当你把回调函数传递给经过优化的并使用引用相等性去避免非必要渲染(例如shouldComponentUpdate)的子组件时，它将非常有用。

举例：

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

import{ Button, Input } from 'antd';

importReact, { useState, useCallback, useEffect } from 'react';

function DemoUseCallBack() {

    const[count, setCount] = useState(1);

    const[val, setVal] = useState('');


    constcallback = useCallback(() => {returncount;}, [count]);

    return

        <div>

            <h4>{count}</h4>

            <Child callback={callback}/>

            <div>
```

复制代码

```
                <Button onClick={() => setCount(count + 1)}>+</Button>

                <Input value={val} onChange={event => setVal(event.target.value)}/>

            </div>

        </div>;

}

function Child({ callback }) {

        const[count, setCount] = useState(() => callback());

        useEffect(() => { setCount(callback());}, [callback]);

        return  <div>{count}</div>

}

export default DemoUseCallBack`
```

useMemo和useCallback的总结：

useCallback和useMemo的参数跟useEffect一致，他们之间最大的区别有是useEffect会用于处理副作用，而前两个hooks不能。

useMemo和useCallback都会在组件第一次渲染的时候执行，之后会在其依赖的变量发生改变时再次执行；并且这两个hooks都返回缓存的值，useMemo返回缓存的变量，useCallback返回缓存的函数。