六、新特性Hooks应用

前面在编写组件的时候，我们常用的方式有类组件与函数组件，那么我们知道函数组件中是没有状态state,和相应的生命周期函数的，而Hook 让你在不编写 class 组件的情况下使用 state，生命周期 以及其他的 React 特性， *Hook* 是 React 16.8 的新增特性 。

所以，有了Hook以后，我们只能说整个React中只有类组件与函数组件，而不能说状态组件和无状态组件。

1、Hook简单使用

下面我们首先定义一个状态组件，实现简单的计算功能

```
import React from 'react';
class Example extends React.Component{
    constructor(props){
        super(props);
        this.state={
            count: 0
        };
    }
    render(){
        return(
        <div>
            <p>{this.state.count}</p>
            <button onClick={()=>(this.setState({count:this.state.count+1}))}>单击</button>
        </div>
        )
    }
}
export default Example
```

下面我们使用Hook来改写上面的程序，前面说过Hook可以让函数组件使用状态，那么下面我们来体验一下。

```
import React from 'react';
import {useState} from 'react'
function HookExample(){
  const [count,setCount]=useState(0);
    return (
        <div>
            <p>{count}</p>
            <button onClick={()=>setCount(count+1)}>单击</button>
        </div>
    )
}
export default HookExample
```

通过上面的案例，可以看出使用Hook让整个程序变得非常简单，HookExample就是一个函数，但是这个函数有自己的状态count,同时还可以更新自己的状态(setCount).【这两个名字可以随意命名】这个函数之所以有状态，就是因为注入了 useState这个Hook,useState这个Hook让函数变成了一个有状态的函数。

除了`useState`这个Hook外，还有很多别的Hook，比如`useEffect`提供了类似于`componentDidMount`等生命周期钩子的功能，`useContext`提供了上下文（context）的功能等等。

Hooks本质上就是一类特殊的函数，它们可以为你的函数型组件（function component）注入一些特殊的功能 。

2、React为什么需要Hooks

2.1 传统应用难以复用

我们都知道react都核心思想就是，将一个页面拆成一堆独立的，可复用的组件，并且用自上而下的以单向数据流的形式将这些组件串联起来。但假如你在大型的工作项目中用react，你会发现你的项目中实际上很多react组件冗长且难以复用。尤其是那些写成class的组件，它们本身包含了状态（state），所以复用这类组件就变得很麻烦。

为了解决组件复用的问题，我们常用的是使用高阶组件来解决。

高阶组件就是指：一个函数接收一个组件作为参数，经过一系列加工后，返回一个新的组件，例如下面的代码

```
const withUser = WrappedComponent => {
  const user = sessionStorage.getItem("user");
  return props => <WrappedComponent user={user} {...props} />;
};

const UserPage = props => (
  <div class="user-container">
    <p>My name is {props.user}!</p>
  </div>
);

export default withUser(UserPage);
```

由于sessionStorage经常使用，所以可以将其封装到一个高阶组件中，这样任何一个组件需要使用sessionStorage，那么只需要使用withUser这个高阶组件包装一下就可以了。

但是高阶组件这种用法带来的问题，就是代码层级关系比较复杂，与hook应用相比，层级嵌套太多，代码不够简洁。

2.2 生命周期函数中逻辑太乱

我们通常希望一个函数只做一件事情，但是我们的生命周期函数里通常做了很多的事情。例如，我们要在componentDidMount中发起ajax请求获取数据，还要绑定一些事件的监听操作等等。这样当项目变得复杂以后，这一块的代码也会变得非常的不直观。

2.3 this指向的问题

我们使用class来创建组件的时候，还有一件比较麻烦的事情，就是this的指向问题。为了保证this的指向正确，我们要经常写这样的代码 this.handleClick = this.handleClick.bind(this) 或者是 <button onClick={() => this.handleClick(e)}> 。一旦我们不小心忘记了绑定this,那么会出现很多的bug。

还有一件事件比较麻烦，就是在前面我们讲过为了更好的复用和提高性能我们建议尽量把组件写成无状态组件的形式，但是由于需求的变动，要求这个组件必须有自己的状态，那么我们有的把function改写成class,这种情况，我们前面也已经遇到过，非常的麻烦。

那么以上几点，就是我们在开发React应用中经常遇到的问题，那么通过新增的Hooks，就可以解决上面的问题。

下面，我们先来讲解一下State Hook的应用。

3、State Hook

在前面，我们已经写过一个State Hook的应用了，如下代码

```
import React from 'react';
import {useState} from 'react'
function HookExample(){
  const [count,setCount]=useState(0);
    return (
        <div>
            <p>{count}</p>
            <button onClick={()=>setCount(count+1)}>单击</button>
        </div>
    )
}
export default HookExample
```

下面我们再来分析一下上面的代码。

3.1 声明一个状态变量

```
import React from 'react';
import {useState} from 'react'
function HookExample(){
  const [count,setCount]=useState(0);
}
```

useState 是react自带的一个Hook函数，它的作用就是用来声明状态变量。useState这个函数接收的参数就是状态的初始值，它返回了一个数组，这个数组的第一项是当前的状态值，第二项是可以改变状态值的方法函数。

所以上面代码的含义就是：声明了一个状态变量count, 把它的初始值设置为了0，同时提供了一个可以更改count这个状态的函数 setCount.

3.2 读取状态值

```
<p>{count}</p>
```

通过这行代码非常的简单，因为count这个状态就是一个单纯的变量，我们以后再也不需要写成

```
{this.state.count}
```

3.3 更新状态

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

当单击按钮的时候，调用setCount这个函数，这个函数接收的参数是修改过的新状态的值。接下来的事情就交给react, react将会重新渲染我们当前所创建的HookExample这个组件，并且使用的是更新后的新状态。

这样每次单击这个按钮的时候，状态不断的更新。

以上就是关于State Hook的基本应用(到这里是一节)

3.4 多个状态处理

在这里，你可能会问，在一个组件中，是否可以通过useState设置多个状态呢？答案是可以的。

如下代码所示：

```
import React from 'react';
import {useState} from 'react'
function HookExample(){
  const [count,setCount]=useState(0);
  const [userName,setUserName]=useState('张三')
  const [txt,setTxt]=useState([{text:'Hello'},{text:'aaa'}])
    return (
        <div>
            <p>{count}</p>
            <p>{userName}</p>
    <p>{txt[0].text}-----{txt[1].text}</p>
            <button onClick={()=>setCount(count+1)}>单击</button>
            <button onClick={()=>setUserName('李四')}>修改用户名</button>
            <button onClick={()=>setTxt([{text: '你好'},{text:'bbbb'}])}>修改文本
</button>

        </div>
    )
}
export default HookExample
```

useState接收的初始值没有规定一定要是string/number/boolean这种简单数据类型，它完全可以接收对象或者数组作为参数 .

同时，通过HookExample函数，我们可以看到，useState无论调用多少次，相互之间都是独立的。

那么问题是，react是怎样保证多个useState是相互独立的呢?

3.5 React 是怎样保证多个useState是相互独立的

React 是根据useState出现的顺序来确定的。如下程序：

```
import React from 'react';
import {useState} from 'react'
let countData=0;

function HookExample(){
    let count,setCount;
    let userName,setUserName;
```

```
    countData+=1;
    // 偶数的情况
    if(countData%2===0){
        [count,setCount]=useState(0);
        [userName,setUserName]=useState('张三')
    }else{
        [userName,setUserName]=useState('张三')
        [count,setCount]=useState(0);
    }
    return (
        <div>
            <p>{count}</p>
            <p>{userName}</p>
             <button onClick={()=>setCount(count+1)}>单击</button>
            <button onclick={()=>setUserName('李四')}>修改用户名</button>

        </div>
    )

}
export default HookExample
```

执行上面的程序，会出现如下的错误：

```
React Hook "useState" is called conditionally. React Hooks must be called in the exact same order in every component render
```

在每个组件中，必须以完全相同的顺序调用React hook

所以，通过上面的案例，我们知道不能在if...else语句中使用useState中。

3.6 useState默认值处理

在通过ReactDOM.render方法渲染组件的时候，可以传递值，那么在这里假设给Example这个组件传递一个

defaultCount属性，来作为useState的初始值。

```
ReactDOM.render(<Example defaultCount={10} />, document.getElementById('root'));
```

```
import React from 'react';
import {useState} from 'react'
function HookExample(props){
    const defaultCount=props.defaultCount||0;
    const [count,setCount]=useState(defaultCount);
    return (
        <div>
            <p>{count}</p>
            <button onClick={()=>setCount(count+1)}>单击</button>
        </div>
    )
}
export default HookExample
```

通过上面的代码可以发现，我们是通过props来接收传递过来的数据，然后把接收到的数据作为了useState方法的参数，赋值给了状态count.当我们运行上面的代码的时候，是没有问题的。但是，仔细观察一下，这里会发现每次渲染该组件的时候，获取初始值的这行代码const defaultCount=props.defaultCount||0; 都会执行。如果这块内容的代码比较复杂的话，会影响性能。既然是初始化的代码，我们希望只执行一次，那么应该怎样进行处理呢?

具体的处理过程如下

```
import React from 'react';
import {useState} from 'react'
function HookExample(props){
    const [count,setCount]=useState(()=>{
        console.log('init');
        return props.defaultCount||0;
    });
    return (
        <div>
            <p>{count}</p>
            <button onClick={()=>setCount(count+1)}>单击</button>
        </div>
    )
}
export default HookExample
```

将接收初始值的这行代码放到useState的回调函数中，并且将其返回，那么返回的值会赋值给状态count,并且回调函数中的代码只会执行一次。

这就是对useState默认值的处理。

3.7 State Hook案例

3.7.1 展示新闻信息

```
import React from 'react';
import {useState} from 'react'
import './style.css'
function TodoList() {
    const [todos,setTodos]=useState([{
        id:1,
        content:'体育新闻',
        isCompleted: true
    },{
        id:2,
        content:'国际新闻',
        isCompleted: true
    }])
    return(
        <div className="app">
            <form className="todo-list">
                <ul>
                    {todos.map((todo)=>(
                          <li className="todo" key={todo.id}>
                              <div className="checkbox">
                                    <input type="checkbox"  value={todo.content}
onClick={()=>setTodos()} />
```

```
                                </div>
                                {todo.content}
                            </li>

                    ))}
                </ul>
                <input type="text"/>
                <button>添加</button>

            </form>
        </div>
    )

}
export default TodoList
```

用到的样式是：

```
body {
    background-color: #282c34;
    min-height: 100vh;
  }

  .app {
    padding-top: 10rem;
  }

  .header {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
  }

  .logo {
    animation: App-logo-spin infinite 20s linear;
    height: 20vmin;
    pointer-events: none;
  }

  .todo-list {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    color: white;
  }

  .todo {
    display: flex;
    align-items: center;
    margin: 1rem 0;
  }

  .todo-is-completed .checkbox {
    color: #000;
    background: #fff;
```

```
}

.todo-is-completed input {
  text-decoration: line-through;
}

.checkbox {
  width: 18px;
  height: 18px;
  border-radius: 50%;
  margin-right: 10px;
  cursor: pointer;
  font-size: 10px;
  display: flex;
  justify-content: center;
  align-items: center;
  transition: background-color .2s ease-in-out;
  border: 1px solid #fff;
}

.checkbox:hover {
  background: rgba(255, 255, 255, 0.25);
  border: 1px solid rgba(255, 255, 255, 0);
}

.checkbox-checked {
  color: #000;
  background: #fff;
}

ul {
  list-style: none;
  padding: 0;
  line-height: 2rem;
  width: 500px;
}

input {
  border: 1px solid;
  background: transparent;
  color: white;
  font-size: 1.4rem;
  outline: none;
  width: 300px;
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

3.7.2 添加新闻信息

添加一个新的useState，用来关联文本框，inputValue这个变量默认值为空字符

```
  const [inputValue, valueChange] = useState("");
```

关联文本框

```
      <input type="text" value={inputValue} onChange=
{(e)=>valueChange(e.target.value)} />
```

```

```

当在文本框中输入值的时候，会触发onChange，执行valueChange这个方法，这时获取文本框中用户输入的值，赋值给了inputValue这个变量。

下面给按钮绑定单击事件。

```
       <button onClick={(e)=>handleClick(e)}>添加</button>
```

handleClick方法完成信息的添加。

```
// 添加信息
    function handleClick(e){
        e.preventDefault();
         setTodos([...todos,
{id:Date.now(),content:inputValue,isCompleted:true}]);
       valueChange("");//清空文本框中的值。
```

在handleClick方法中，调用setTodos方法，向todos数组中添加用户输入的数据。

然后执行valueChange("")清空文本框，执行valueChange("")时给inputValue这个字符串赋值了一个空字符串，而inputValue与文本框的value属性绑定在了一起，所以文本框被清空了。

完整代码如下：

```
import React from 'react';
import {useState} from 'react'
import './style.css'
function TodoList() {
    //完成添加
    const [inputValue, valueChange] = useState("");
    const [todos,setTodos]=useState([{
        id:1,
        content:'体育新闻',
        isCompleted: true
    },{
        id:2,
        content:'国际新闻',
        isCompleted: true
    }])
// 添加信息
    function handleClick(e){
        e.preventDefault();
```

博学谷 www.boxuegu.com

```
          setTodos([...todos,
{id:Date.now(),content:inputValue,isCompleted:true}]);
        valueChange("");//清空文本框中的值。

    }
    return(
        <div className="app">
            <form className="todo-list">
                <ul>
                    {todos.map((todo)=>(
                          <li className="todo" key={todo.id}>
                              <div className="checkbox">
                                    <input type="checkbox"  value=
{todo.content}/>

                              </div>
                              {todo.content}
                          </li>

                    ))}
                </ul>
                <input type="text" value={inputValue} onChange=
{(e)=>valueChange(e.target.value)} />
               <button onClick={(e)=>handleClick(e)}>添加</button>

            </form>
        </div>
    )

}
export default TodoList
```

3.7.3 删除新闻信息

首先给map循环添加索引，并且给li绑定一个单击的事件。

```
                <ul>
                    {todos.map((todo,index)=>(
                          <li className="todo" key={todo.id} onClick=
{()=>handleDelete(index)}>
                              <div className="checkbox">
                                    <input type="checkbox"  value=
{todo.content}/>

                              </div>
                              {todo.content}
                          </li>

                    ))}
                </ul>
```

下面定义handleDelete方法

```
    // 删除信息
     function handleDelete(index){
       setTodos(todos => todos.slice(0, index))
     }
```

在handleDelete方法中调用setTodos方法删除todos数组中指定的项。

通过以上的案例，发现使用State Hook确实是非常的简单。

3.8 useState原理分析

伪代码如下：

```
const MyReact=(function(){
    //开辟一个存储hooks的空间
    let hooks=[];
    //指针从0开始
    let currentHook=0;
    return {
        useState(initValue){
            //hooks[0]='lisi'
            hooks[currentHook]=hooks[currentHook]||initValue;
            // setStateHookIndex=0;
            const setStateHookIndex=currentHook;
            // setState方法，修改状态。
            const setState=newState=>(hooks[setStateHookIndex]=newState)

            //这里要考虑的是要多次使用useState,所以索引要加1.
            return [hooks[currentHook++],setState]
        },
        render(Component){

        }

    }
})()
```

useState是来自react包，所以这里我们定义一个MyReact

4、Effect Hook

4.1 Effect Hook基本使用

下面，我们实现一个比较简单的案例。在前面的计算器案例的基础上，添加了一个更新文档标题的功能。

按照以前的做法。

（下面的代码只写，生命周期的代码）

```
import React from 'react';
class Example extends React.Component {
```

```
    constructor(props) {
      super(props);
      this.state = {
        count: 0
      };
    }

    componentDidMount() {
      document.title = `You clicked ${this.state.count} times`;
    }

    componentDidUpdate() {
      document.title = `You clicked ${this.state.count} times`;
    }

    render() {
      return (
        <div>
          <p> {this.state.count} </p>
          <button onClick={() => this.setState({ count: this.state.count + 1
})}>
           单击
          </button>
        </div>
      );
    }
  }
  export default  Example
```

下面，我们换另外一种做法

```
import React,{ useState, useEffect } from 'react';

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

  // 类似于componentDidMount 和 componentDidUpdate:
  useEffect(() => {
    // 更新文档的标题
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p> {count} </p>
      <button onClick={() => setCount(count + 1)}>
      单击
      </button>
    </div>
  );
}
export default Example
```

在最开始的时候，导入useEffect. 然后在useEffect中指定回调函数。

useEffect的作用类似于类似于componentDidMount 和 componentDidUpdate。

使用起来更加的简单方便。

通过这个案例，我们可以总结出以下几点：

第一：React首次渲染和之后的每次渲染都会调用一遍useEffect中的函数，而之前我们需要用两个生命周期函数来分别表示首次渲染(componentDidMount)，和之后的更新导致的重新渲染（componentDidUpdate）.

第二：useEffect中定义的函数的执行不会阻碍浏览器更新视图，也就是说这些函数是异步执行的。

第三：在useEffect中可以直接访问count state,我们不需要特殊的API来读取它，它已经保存在函数的作用域中。

在前面已经讲过，每次渲染都会调用一遍useEffect函数，那么在某些情况下是非常消耗性能的，那么应该怎样解决这个问题呢?

```
  useEffect(() => {
    // 更新文档的标题
    document.title = `You clicked ${count} times`;
  },[count]);
```

用第二个参数来告诉react只有当这个参数的值发生改变时，才执行对应的函数 。第二个参数是可选的，但是要用的话，对应的类型是数组。**也就是数组中的每一项都不变的情况下，useEffect才不会执行。**

通过useEffect，可以在函数组件中使用生命周期的函数，而不需要在将其转换成类组件。

4.2 使用 useEffect 获取数据

useEffect类似于生命周期函数componentDidMount，所以我们可以在useEffect中构建异步请求，来获取数据。

如下所示

```
import React,{ useState, useEffect } from 'react';
function Example(){
    const [posts,setPosts]=useState([]);
    useEffect(()=>{
      fetch('http://localhost:8081/api/getProductList')
      .then(res=>res.json())
      .then(json=>(
            setPosts(json.message.map(item=>item))
          )
       )

    })
  return(
      <ul>

          {posts.map(post=>(
              <li key={post.id}>{post.title}</li>
          ))}
      </ul>
  )
}
export default Example
```

虽然上面的代码可以实现异步请求，但是问题是只要组件重新渲染一次，那么就会重新发送异步请求。所以上面的代码是有问题的，可以通过控制台进行监视。（useEffect执行时，它获取数据并更新状态。然后，一旦状态更新，组件将重新呈现，这将再次触发`useEffect`，这就是问题所在。）

那么应该怎样解决这个问题呢?

可以给useEffect添加第二个参数就可以。

`useEffect`所依赖的唯一变量是`setPosts`。因此，咱们应该在这里传递数组`[setPosts]`。这里每一次渲染的时候，setPosts都是一样的（操作的数据是一样的） 所以，不会在每次渲染的时候重新创建它，因为useEffect只会执行一次。

```
import React,{ useState, useEffect } from 'react';
function Example(){
    const [posts,setPosts]=useState([]);
    useEffect(()=>{
      fetch('http://localhost:8081/api/getProductList')
      .then(res=>res.json())
      .then(json=>(
            setPosts(json.message.map(item=>item))
          )
       )

    },[setPosts])// 这里传递了第二个参数
  return(
      <ul>

          {posts.map(post=>(
              <li key={post.id}>{post.title}</li>
          ))}
      </ul>
  )
}
export default Example
```

4.3 useEffect原理分析

伪代码如下

```
let _deps; // _deps 记录 useEffect 上一次的 依赖

function useEffect(callback, depArray) {
  const hasNoDeps = !depArray; // 如果 dependencies 不存在
  const hasChangedDeps = _deps
    ? !depArray.every((el, i) => el === _deps[i]) // 两次的 dependencies（传递过来的依赖与上一次的依赖是否相同） 是否完全相等,如果相等，没有变化，返回false,如果有变化返回true.
    : true;
  /* 如果 dependencies 不存在，或者 dependencies 有变化*/
  if (hasNoDeps || hasChangedDeps) {
    callback();
    _deps = depArray;
  }
}
```

5、Context Hook

5.1 回顾Context的应用

Context:能够运行**数据跨越组件层级进行数据的传递。**

也就是说Context是一种在应用程序中深入传**递数据的方式**，而无需手动一个一个在多个父子孙之间传递props.

下面我们来演示一下使用Context的使用（这块内容在前面的课程中也已经讲解过了）：

示例代码如下所示：

```
import React from 'react'
const NumberContext=React.createContext();
// Context中有两个对象{Provider(向子组件中传递数据),Consumer(接收传递过来的数据)}
function Example(){
    return(
        // 使用NumberContext中的Provider为子组件传递数据
        <NumberContext.Provider value={30}>
            <div>
                <Display/>
            </div>
        </NumberContext.Provider>
    )
}
function Display(){
    return (
        // 使用Consumer接收Context中传递过来的数据
    <NumberContext.Consumer>
        {(value)=>(<div>传递过来的数据是{value}</div>)}
    </NumberContext.Consumer>
    )
}
export default Example
```

5.2 useContext的使用

下面使用 Context Hook的方式改造上面的代码。

```
import React,{useContext} from 'react'
const NumberContext=React.createContext();

function Example(){
    return(
        // 使用NumberContext中的Provider为子组件传递数据
        <NumberContext.Provider value={60}>
            <div>
                <Display/>
            </div>
        </NumberContext.Provider>
    )
```

```
}
function Display(){
    //调用useContext的时候，需要传递的是React.createContext创建的上下文对象。
    const value=useContext(NumberContext);
return <div>数据为:{value}</div>
}
export default Example
```

通过以上的代码，可以发现使用useContext是非常简单的，而且在子组件中不需要在使用Consumer.

5.3 嵌套的Consumers

我们在使用Context进行数据传递的时候，经常会遇到嵌套多个组件的情况，也就是从多个嵌套的组件中接收数据。如下伪代码所示 ,接收两个嵌套组件传递过来的值。

```
function HeaderBar() {
  return (
    <CurrentUser.Consumer>
      {user =>
        <Notifications.Consumer>
          {notifications =>
            <header>
              Welcome back, {user.name}!
              You have {notifications.length} notifications.
            </header>
          }
        }
    </CurrentUser.Consumer>
  );
}
```

通过上面的伪代码，可以发现整个的结构比较复杂，那么对应的使用useContext来实现的话就比较简单了，如下伪代码所示：

```
function HeaderBar() {
  const user = useContext(CurrentUser);
  const notifications = useContext(Notifications);

  return (
    <header>
      Welcome back, {user.name}!
      You have {notifications.length} notifications.
    </header>
  );
}
```

整体的代码结构非常的清晰，使用也非常的简单。

6、Memo Hook

6.1 memo的应用

当父组件引入子组件的情况下，往往会造成组件之间的一些不必要的浪费。如下案例：

```
import React,{useState} from 'react';
const Child=(porps)=>{
    console.log('子组件');
    return(
        <div>
            我是一个子组件
        </div>
    )
}
function Example(){
    const[count,setCount] =useState(0);
    return(
        <div>
            <button onClick={(e)=>{setCount(count+1)}}>加1</button>
            <p>count:{count}</p>
            <Child></Child>
        </div>
    )
}
export default Example
```

每次单击按钮进行计算的时候，Child这个子组件也会被渲染执行。

为了解决这个问题，可以使用memo. 如下代码所示

```
import React,{useState,memo} from 'react';
const Child=(porps)=>{
    console.log('子组件');
    return(
        <div>
            我是一个子组件
        </div>
    )
}
// 创建memo
const ChildMemo=memo(Child)
function Example(){
    const[count,setCount] =useState(0);
    return(
        <div>
            <button onClick={(e)=>{setCount(count+1)}}>加1</button>
            <p>count:{count}</p>
            // 加载memo封装的组件
            <ChildMemo></ChildMemo>
        </div>
    )
}
export default Example
```

通过memo把Child进行封装后，每次单击按钮的时候，Child这个子组件就不会在被执行、而是仅仅在第一次被加载的时候被执行。

博学谷 www.boxuegu.com

6.2 useMemo应用

如果给子组件传递数状态据，发现memo又不起作用了。

```
import React,{useState,memo} from 'react';
const Child=(porps)=>{

    function changeName(name) {
        console.log('111');
        return name+'改变了name'
    }
    const otherName =changeName(porps.name);
    return (
        <div>
           <div>{otherName}</div>
    <div>{porps.children}</div>
        </div>
    )
}
// 创建memo
const ChildMemo=memo(Child)
function Example(){
    const[count,setCount] =useState(0);
    const[name,setName]=useState('子组件')
    return(
        <div>
            <button onClick={()=>{setCount(count+1)}}>加1</button>
            <p>count:{count}</p>
            <button onClick={()=>{setName(new Date().getTime())}}>改变姓名
</button>
    <ChildMemo name={name}>{count}</ChildMemo>
        </div>
    )
}
export default Example
```

以上的案例中，给ChildMemo传递状态时，memo不起作用了。

当我们单击父组件的按钮的时候，子组件的name和children都会发生变化.

注意console.log的输出。

也就是说，不管我们是改变name或者是count的值，我们发现changeName这个方法都会被调用。

是不是意味着，我们本来只想修改count的值，但是由于name并没有发生变化，所以无需执行对应的changeName方法。但是，我们发现它却执行了，这是不是意味着性能的损耗，做了无用功。

下面我们使用useMemo进行优化

```
import React,{useState,useMemo} from 'react';
const Child=(porps)=>{

    function changeName(name) {
        console.log('111');
        return name+'改变了name'
    }
    //使用useMemo(只有name属性的值发生了变化，才会执行changeName方法)
    const otherName = useMemo(()=>changeName(porps.name),[porps.name]);
```

```
    return (
        <div>
           <div>{otherName}</div>
    <div>{porps.children}</div>
        </div>
    )
}
// 创建memo
const ChildMemo=Child;
function Example(){
    const[count,setCount] =useState(0);
    const[name,setName]=useState('子组件')
    return(
        <div>
            <button onClick={()=>{setCount(count+1)}}>加1</button>
            <p>count:{count}</p>
            <button onClick={()=>{setName(new Date().getTime())}}>改变姓名
</button>
    <ChildMemo name={name}>{count}</ChildMemo>
        </div>
    )
}
export default Example
```

这时，发现单击count计算按钮的时候，发现changeName这个方法没有被调用。但是单击改变name属性值按钮的时候，changeName被调用了，所以，我们可以使用useMemo这个方法来避免无用方法的调用。

7、Ref Hook

7.1 作为DOM引用

我们在前面的课程中，讲解过ref的用法，它可以用来获取组件实例对象或者是DOM对象。而useRef这个Hooks这个函数，也具有这样的功能，如下代码所示：

```
import React, { useState, useEffect, useMemo, useRef } from 'react';

export default function App(props){
  const [count, setCount] = useState(0);

  const doubleCount = useMemo(() => {
    return 2 * count;
  }, [count]);

  const couterRef = useRef();

  useEffect(() => {
    document.title = `The value is ${count}`;
    console.log(couterRef.current);
  }, [count]);

  return (
    <>
```

```
      <button ref={couterRef} onClick={() => {setCount(count + 1)}}>Count:
{count}, double: {doubleCount}</button>
    </>
  );
}
```

上面的代码中使用useRef创建了couterRef对象，并且将其赋值给了button的ref属性，通过访问couterRef.current就可以访问到button对应的DOM对象了。

7.2 作为实例属性

```
import React, { useState, useEffect, useMemo, useRef } from 'react';

export default function App(props){
  const [count, setCount] = useState(0);

  const doubleCount = useMemo(() => {
    return 2 * count;
  }, [count]);

  const timerID = useRef();

  useEffect(() => {
    timerID.current = setInterval(()=>{
        setCount(count => count + 1);
    }, 1000);
  }, []);//[]：空数组表示只有第一次渲染的时候执行useEffect,后面不在执行，因为空数组内容没有
任何的变化。在第一次执行的时候创建了定时器。

  useEffect(()=>{
      if(count > 10){

          clearInterval(timerID.current);//useRef类似于类的实例属性，所以这里
timerID.current返回的与前面的是同一个对象。
      }
  });

  return (
    <div>
      <button  onClick={() => {setCount(count + 1)}}>Count: {count}, double:
{doubleCount}</button>
    </div>
  );
}
```

8、Reducer Hook

什么是reducer?

关于reducer的概念，在前面讲解Redux的课程中，我们已经讲解过。简单的说，reducer是一个函数

```
(state,action)=>newState
```

该函数接收当前应用的state和触发的动作action,计算并返回最新的state. 如下面的一段伪代码

```
    function countReducer(state, action) {
        switch(action.type) {
            case 'add':
                return state + 1;
            case 'sub':
                return state - 1;
            default:
                return state;
        }
    }
```

上面的代码 根据state（当前状态）和action（触发的动作加、减）参数，计算返回newState 。

Reducer Hook是通过useReducer方法来创建。

基本的语法格式：

```
    const [state, dispatch] = useReducer(reducer, initState);
```

useReducer是useState的替代方案，它接收一个形式如 (state,atcion)=>newState的reducer,并且返回当前的state以及与其配套的dispatch方法。 在某些场景下，`useReducer` 会比 `useState` 更适用，例如 state 逻辑较复杂且包含多个子值，或者下一个 state 依赖于之前的 state 等 。

下面演示一下关于useReducer的基本使用

```
import React from 'react';
import {useReducer} from 'react'
// 对应useReducer的初始状态。
const initState={count:0}
function reducer(state, action) {
    switch (action.type) {
        case 'Add':
            return {count:state.count+1};
         case 'Sub':
             return {count:state.count-1};
          default:
              throw new Error();
    }
}

function Example(){
    // 返回值：最新的state和dispatch函数
    const[state,dispatch] =useReducer(reducer,initState);
    return(
        <div>
            {/* useReducer会根据dispatch的action，返回最终的state，并触发rerender */}
            count:{state.count}
             {/* dispatch 用来接收一个 action参数[reducer中的action]，用来触发reducer
函数，更新最新的状态 */}
             <button onClick={() => dispatch({type: 'Add'})}>加法</button>
               <button onClick={() => dispatch({type: 'Sub'})}>减法</button>
```

```
        </div>
    )
}
export default Example
```

前面说过useReducer是useState的替代方案,下面我们通过一段伪代码来体会一下

这个案例是一个登录的案例，如果我们最开始使用useState来完成，那么对应的形式如下：

```
   function LoginPage() {
        const [name, setName] = useState(''); // 用户名
        const [pwd, setPwd] = useState(''); // 密码
        const [isLoading, setIsLoading] = useState(false); // 是否展示loading，发送请求中
        const [error, setError] = useState(''); // 错误信息
        const [isLoggedIn, setIsLoggedIn] = useState(false); // 是否登录

        const login = (event) => {
            event.preventDefault();
            setError('');
            setIsLoading(true);
            login({ name, pwd })
                .then(() => {
                    setIsLoggedIn(true);
                    setIsLoading(false);
                })
                .catch((error) => {
                    // 登录失败: 显示错误信息、清空输入框用户名、密码、清除loading标识
                    setError(error.message);
                    setName('');
                    setPwd('');
                    setIsLoading(false);
                });
        }
        return (
            //  返回页面
        )
    }
```

上面Demo我们定义了5个state来描述页面的状态，在login函数中当登录成功、失败时进行了一系列复杂的state设置。可以想象随着需求越来越复杂更多的state加入到页面，更多的setState分散在各处，很容易设置错误或者遗漏，维护这样的老代码更是一个噩梦。

下面使用useReducer来改造上面的案例：

```
    const initState = {
        name: '',
        pwd: '',
        isLoading: false,
        error: '',
        isLoggedIn: false,
    }
    function loginReducer(state, action) {
        switch(action.type) {
            case 'login':
```

```
                return {
                    ...state,
                    isLoading: true,
                    error: '',
                }
            case 'success':
                return {
                    ...state,
                    isLoggedIn: true,
                    isLoading: false,
                }
            case 'error':
                return {
                    ...state,
                    error: action.payload.error,
                    name: '',
                    pwd: '',
                    isLoading: false,
                }
            default:
                return state;
        }
    }
    function LoginPage() {
        const [state, dispatch] = useReducer(loginReducer, initState);
        //获取最新的状态
        const { name, pwd, isLoading, error, isLoggedIn } = state;
        const login = (event) => {
            event.preventDefault();
            dispatch({ type: 'login' });//出现isLoading
            login({ name, pwd })
                .then(() => {
                    dispatch({ type: 'success' });
                })
                .catch((error) => {
                    dispatch({
                        type: 'error'
                        payload: { error: error.message }
                    });
                });
        }
        return (
            //  返回页面
        )
    }
```

乍一看useReducer改造后的代码反而更长了，但很明显第二版有更好的可读性，我们也能更清晰的了解state的变化逻辑。

可以看到login函数现在更清晰的表达了用户的意图，开始登录login、登录success、登录error。LoginPage不需要关心如何处理这几种行为，那是loginReducer需要关心的，表现和业务分离。

另一个好处是所有的state处理都集中到了一起，使得我们对state的变化更有掌控力，同时也更容易复用state逻辑变化代码，比如在其他函数中也需要触发登录error状态，只需要**dispatch({ type: 'error' })**。

通过上面的案例，我们可以发现如果state 逻辑比较复杂，那么对应的就可以考虑使用useReducer.

9、自定义Hook

Hook的自定义，也非常的简单，要注意的一点是所以定义的方法以use开头

```
import React, { useState, useEffect, useRef } from 'react';
// 自定义Hook,必须以use开头
function useCount(defaultValue){
    const [count,setCount] = useState(defaultValue);
    const timerID = useRef();
    useEffect(() => {
        timerID.current = setInterval(()=>{
            setCount(count => count + 1);
        }, 1000);
      }, []);
      useEffect(()=>{
        if(count > 10){
            clearInterval(timerID.current);
        }
    });
    // 将状态，以及对应的方法返回
   return [count,setCount]
}
function Example(){
    const[count,setCount] =useCount(0)
    return(<div>
        <p>{count}</p>
        <button onClick={()=>{setCount(count+1)}}>单击</button>

    </div>)
}
export default Example
```

自定义Hook，类似于函数组件的创建。自定义Hook,可以实现状态逻辑的复用。

10、使用Hook规则

关于Hook的使用规则，在文档中已经有详细的说明： https://react.docschina.org/docs/hooks-rules.html

下面在简单的总结一下：

第一：只在最顶层使用Hook

不要在循环，条件或嵌套函数中调用Hook.

遵守这条规则，能够确保Hook在每一次渲染中按照同样的顺序被调用。

第二、只在React函数中调用Hook

不要在普通的JavaScript函数中调用Hook.

可以在React的函数组件中调用Hook

可以在自定义Hook中调用其他Hook（例如我们前面写的useCount,这个就是自定义的Hook）。

例如下面的就是一个普通的函数

```
funtion test(){

}
```

不要在这种普通的函数中调用Hook,因为我们有可能会在if等条件语句中使用该普通函数，那么这样又导致了被调用的顺序问题。

博学谷 www.boxuegu.com