P01:React Hooks介绍和环境搭建

React Hooks 简介

2018年底FaceBook的React小组推出Hooks以来，所有的React的开发者都对它大为赞赏。`React Hooks`就是用函数的形式代替原来的继承类的形式，并且使用预函数的形式管理`state`，有Hooks可以不再使用类的形式定义组件了。这时候你的认知也要发生变化了，原来把组件分为有状态组件和无状态组件，有状态组件用类的形式声明，无状态组件用函数的形式声明。那现在所有的组件都可以用函数来声明了。

我们这里先不说Hooks有什么好处，就算说了，你也不可能完全理解，好像我王婆卖瓜自卖自夸一样，所以先学习，学过几节课后，我们再来总结`React Hooks`的好处。

使用`create-react-app`创建项目

这里我在D盘新建一个`ReactHooksDemo`的文件夹，然后在文件夹中用`create-react-app`创建一个demo01的项目。我们这些动作全部在命令提示符（我习惯叫终端）中进行。

```
D: // 进入D盘
mkdir ReactHooksDemo
cd ReactHooksDemo
create-react-app demo01
```

只留`/src/index.js`文件，然后把里边的代码删减成下面的样子:

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(<App />, document.getElementById('root'));
```

这样就算开发环境搭建完成了，接下来我们对比一下原始的写法和现在有了React Hooks的写法。

React Hooks 编写形式对比

先来写一个最简单的有状体组件，点我们点击按钮时，点击数量不断增加。

原始写法:

```
import React, { Component } from 'react';

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

```
    render() {
        return (
            <div>
                <p>You clicked {this.state.count} times</p>
                <button onClick={this.addCount.bind(this)}>Chlick me</button>
            </div>
        );
    }
    addCount(){
        this.setState({count:this.state.count+1})
    }
}

export default Example;
```

React Hooks 写法：

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

从这两个程序的对比上可以看出Hooks本质上就是一类特殊的函数，他们可以为你的函数型组件（function component）注入一些特殊的功能。这听起来有点像以前React中的`Mixins`差不多哦。其实是由很多不同，hooks的目的就是让你不再写`class`，让`function`一统江湖。

P02:useState 的介绍和多状态声明

useState的介绍

`useState`是react自带的一个hook函数，它的作用是用来声明状态变量。

那我们从三个方面来看`useState`的用法，分别是声明、读取、使用（修改）。这三个方面掌握了，你基本也就会使用`useState`了.

先来看一下声明的方式，上节课的代码如下：

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

这种方法是ES6语法中的数组解构，这样看起来代码变的简单易懂。现在ES6的语法已经在工作中频繁使用，所以如果你对ES6的语法还不熟悉，我觉的有必要拿出2天时间学习一下。如果不写成数组解构，上边的语法要写成下面的三行:

```
let _useState = userState(0)
let count = _useState[0]
let setCount = _useState[1]
```

`useState`这个函数接收的参数是状态的初始值(Initial state)，它返回一个数组，这个数组的第0位是当前的状态值，第1位是可以改变状态值的方法函数。所以上面的代码的意思就是声明了一个状态变量为count，并把它的初始值设为0，同时提供了一个可以改变`count`的状态值的方法函数。
这时候你已经会声明一个状态了，接下来我们看看如何读取状态中的值。

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

你可以发现，我们读取是很简单的。只要使用`{count}`就可以，因为这时候的count就是JS里的一个变量，想在`JSX`中使用，值用加上`{}`就可以。
最后看看如果改变`State`中的值,看下面的代码:

```
<buttononClick={()=>{setCount(count+1)}}>click mebutton>
```

直接调用setCount函数，这个函数接收的参数是修改过的新状态值。接下来的事情就交给`React`,他会重新渲染组件。`React`自动帮助我们记忆了组件的上一次状态值，但是这种记忆也给我们带来了一点小麻烦，但是这种麻烦你可以看成规则，只要准守规则，就可以愉快的进行编码。

多状态声明的注意事项

比如现在我们要声明多个状态，有年龄（age）、性别(sex)和工作(work)。代码可以这么写.

```
import React, { useState } from 'react';
function Example2(){
    const [ age , setAge ] = useState(18)
    const [ sex , setSex ] = useState('男')
    const [ work , setWork ] = useState('前端程序员')
    return (
        <div>
            <p>JSPang 今年:{age}岁</p>
            <p>性别:{sex}</p>
            <p>工作是:{work}</p>
```

```
11
12          </div>
13      )
14  }
15  export default Example2;
```

其实细心的小伙伴一定可以发现，在使用`useState`的时候只赋了初始值，并没有绑定任何的`key`,那React是怎么保证这三个useState找到它自己对应的state呢?

答案是：React是根据useState出现的顺序来确定的

比如我们把代码改成下面的样子：

```
1  import React, { useState } from 'react';
2
3  let showSex = true
4  function Example2(){
5      const [ age , setAge ] = useState(18)
6      if(showSex){
7          const [ sex , setSex ] = useState('男')
8          showSex=false
9      }
10
11     const [ work , setWork ] = useState('前端程序员')
12     return (
13         <div>
14             <p>JSPang 今年:{age}岁</p>
15             <p>性别:{sex}</p>
16             <p>工作是:{work}</p>
17
18         </div>
19     )
20 }
21 export default Example2;
```

这时候控制台就会直接给我们报错，错误如下：

```
1  React Hook "useState" is called conditionally. React Hooks must be called in the
```

意思就是useState不能在`if...else...`这样的条件语句中进行调用，必须要按照相同的顺序进行渲染。如果你还是不理解，你可以记住这样一句话就可以了：**就是React Hooks不能出现在条件判断语**

句中，因为它必须有完全一样的渲染顺序。

P03:useEffect代替常用生命周期函数

在用Class制作组件时，经常会用生命周期函数，来处理一些额外的事情（副作用：和函数业务主逻辑关联不大，特定时间或事件中执行的动作，比如Ajax请求后端数据，添加登录监听和取消登录，手动修改DOM等等）。在React Hooks中也需要这样类似的生命周期函数，比如在每次状态（State）更新时执行，它为我们准备了useEffect。从这节课开始来认识一下这个useEffect函数。

用Class的方式为计数器增加生命周期函数

为了让你更好的理解useEffect的使用，先用原始的方式把计数器的Demo增加两个生命周期函数componentDidMount和componentDidUpdate。分别在组件第一次渲染后在浏览器控制台打印出计数器结果和在每次计数器状态发生变化后打印出结果。代码如下：

```
import React, { Component } from 'react';

class Example3 extends Component {
    constructor(props) {
        super(props);
        this.state = { count:0 }
    }


    componentDidMount(){
        console.log(`ComponentDidMount=>You clicked ${this.state.count} times`)
    }
    componentDidUpdate(){
        console.log(`componentDidUpdate=>You clicked ${this.state.count} times`)
    }

    render() {
        return (
            <div>
                <p>You clicked {this.state.count} times</p>
                <button onClick={this.addCount.bind(this)}>Chlick me</button>
            </div>
        );
    }
    addCount(){
        this.setState({count:this.state.count+1})
    }
```

```
28 }
29
30 export default Example3;
```

这就是在不使用Hooks情况下的写法，那如何用Hooks来代替这段代码，并产生一样的效果那。

用useEffect函数来代替生命周期函数

在使用React Hooks的情况下，我们可以使用下面的代码来完成上边代码的生命周期效果，代码如下（修改了以前的diamond）：记得要先引入useEffect后，才可以正常使用。

```
1  import React, { useState , useEffect } from 'react';
2  function Example(){
3      const [ count , setCount ] = useState(0);
4      //---关键代码---------start-------
5      useEffect(()=>{
6          console.log(`useEffect=>You clicked ${count} times`)
7      })
8      //---关键代码---------end-------
9
10     return (
11         <div>
12             <p>You clicked {count} times</p>
13             <button onClick={()=>{setCount(count+1)}}>click me</button>
14         </div>
15     )
16 }
17 export default Example;
```

写完后，可以到浏览器中进行预览一下，可以看出跟class形式的生命周期函数是完全一样的，这代表第一次组件渲染和每次组件更新都会执行这个函数。那这段代码逻辑是什么？我们梳理一下:首先，我们生命了一个状态变量count,将它的初始值设为0，然后我们告诉react，我们的这个组件有一个副作用。给useEffecthook传了一个匿名函数，这个匿名函数就是我们的副作用。在这里我们打印了一句话，当然你也可以手动的去修改一个DOM元素。当React要渲染组件时，它会记住用到的副作用，然后执行一次。等Reat更新了State状态时，它再一词执行定义的副作用函数。

useEffect两个注意点

1. React首次渲染和之后的每次渲染都会调用一遍useEffect函数，而之前我们要用两个生命周期函数分别表示首次渲染(componentDidMonut)和更新导致的重新渲染(componentDidUpdate)。
2. useEffect中定义的函数的执行不会阻碍浏览器更新视图，也就是说这些函数时异步执行的，而componentDidMonut和componentDidUpdate中的代码都是同步执行的。个人认为这个有好处也有坏处

吧，比如我们要根据页面的大小，然后绘制当前弹出窗口的大小，如果时异步的就不好操作了。

P04:useEffect 实现 componentWillUnmount生命周期函数

在写React应用的时候，在组件中经常用到componentWillUnmount生命周期函数（组件将要被卸载时执行）。比如我们的定时器要清空，避免发生内存泄漏;比如登录状态要取消掉，避免下次进入信息出错。所以这个生命周期函数也是必不可少的，这节课就来用useEffect来实现这个生命周期函数,并讲解一下useEffect容易踩的坑。

useEffect解绑副作用

学习React Hooks 时，我们要改掉生命周期函数的概念（人往往有先入为主的毛病，所以很难改掉），因为Hooks叫它副作用，所以componentWillUnmount也可以理解成解绑副作用。这里为了演示用useEffect来实现类似componentWillUnmount效果，先安装React-Router路由,进入项目根本录，使用npm进行安装。

```
npm install--save react-router-dom
```

然后打开Example.js文件，进行改写代码，先引入对应的React-Router组件。

```
import { BrowserRouter as Router, Route, Link } from"react-router-dom"
```

在文件中编写两个新组件，因为这两个组件都非常的简单，所以就不单独建立一个新的文件来写了。

```
function Index() {
    return <h2>JSPang.com</h2>;
}

function List() {
    return <h2>List-Page</h2>;
}
```

有了这两个组件后，接下来可以编写路由配置，在以前的计数器代码中直接增加就可以了。

```
return (
    <div>
        <p>You clicked {count} times</p>
        <button onClick={()=>{setCount(count+1)}}>click me</button>

        <Router>
            <ul>
                <li> <Link to="/">首页</Link> </li>
                <li><Link to="/list/">列表</Link> </li>
```

```
            </ul>
            <Route path="/" exact component={Index} />
            <Route path="/list/" component={List} />
        </Router>
    </div>
)
```

然后到浏览器中查看一下，看看组件和路由是否可用。如果可用，我们现在可以调整`useEffect`了。在两个新组件中分别加入`useEffect()`函数:

```
function Index() {
    useEffect(()=>{
        console.log('useEffect=>老弟，你来了！Index页面')
        )
    return <h2>JSPang.com</h2>;
}

function List() {
    useEffect(()=>{
        console.log('useEffect=>老弟，你来了！List页面')
    })

    return <h2>List-Page</h2>;
}
```

这时候我们点击`Link`进入任何一个组件，在浏览器中都会打印出对应的一段话。这时候可以用**返回一个函数的形式进行解绑**，代码如下:

```
function Index() {
    useEffect(()=>{
        console.log('useEffect=>老弟你来了！Index页面')
        return ()=>{
            console.log('老弟，你走了!Index页面')
        }
    })
    return <h2>JSPang.com</h2>;
  }
```

这时候你在浏览器中预览，我们仿佛实现了componentWillUnmount方法。但这只是好像实现了，当点击计数器按钮时，你会发现老弟，你走了!Index页面，也出现了。这到底是怎么回事那？其实每次状态发生变化，useEffect都进行了解绑。

useEffect的第二个参数

那到底要如何实现类似componentWillUnmount的效果那?这就需要请出useEffect的第二个参数，它是一个数组，数组中可以写入很多状态对应的变量，意思是当状态值发生变化时，我们才进行解绑。但是当传空数组[]时，就是当组件将被销毁时才进行解绑，这也就实现了componentWillUnmount的生命周期函数。

```
function Index() {
    useEffect(()=>{
        console.log('useEffect=>老弟你来了! Index页面')
        return ()=>{
            console.log('老弟，你走了!Index页面')
        }
    },[])
    return <h2>JSPang.com</h2>;
}
```

为了更加深入了解第二个参数的作用，把计数器的代码也加上useEffect和解绑方法，并加入第二个参数为空数组。代码如下：

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

    useEffect(()=>{
        console.log(`useEffect=>You clicked ${count} times`)

        return ()=>{
            console.log('====================')
        }
    },[])

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={()=>{setCount(count+1)}}>click me</button>

```

```
            <Router>
                <ul>
                    <li> <Link to="/">首页</Link> </li>
                    <li><Link to="/list/">列表</Link> </li>
                </ul>
                <Route path="/" exact component={Index} />
                <Route path="/list/" component={List} />
            </Router>
        </div>
    )
}
```

这时候的代码是不能执行解绑副作用函数的。但是如果我们想每次count发生变化，我们都进行解绑，只需要在第二个参数的数组里加入count变量就可以了。代码如下：

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

    useEffect(()=>{
        console.log(`useEffect=>You clicked ${count} times`)

        return ()=>{
            console.log('====================')
        }
    },[count])

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={()=>{setCount(count+1)}}>click me</button>

            <Router>
                <ul>
                    <li> <Link to="/">首页</Link> </li>
                    <li><Link to="/list/">列表</Link> </li>
                </ul>
                <Route path="/" exact component={Index} />
                <Route path="/list/" component={List} />
            </Router>
```

```
        </div>
    )
}
```

这时候只要count状态发生变化，都会执行解绑副作用函数，浏览器的控制台也就打印出了一串==========。

这节课学完我们就对useEffect函数有了一个比较深入的了解，并且可以通过useEffect实现生命周期函数了，也完成了本节课学习的目的，现在用React Hooks这种函数的方法编写组件，对比以前用Class编写组件几乎一样了。但这并不是Hooks的所有东西，它还有一些让我们惊喜的新特性。这节就到这里了，下节课我们继续讲解React Hooks。

P05:useContext 让父子组件传值更简单

有了useState和useEffect已经可以实现大部分的业务逻辑了，但是React Hooks中还是有很多好用的Hooks函数的，比如useContext和useReducer。

在用类声明组件时，父子组件的传值是通过组件属性和props进行的，那现在使用方法(Function)来声明组件，已经没有了constructor构造函数也就没有了props的接收，那父子组件的传值就成了一个问题。React Hooks 为我们准备了useContext。这节课就学习一下useContext，它可以帮助我们跨越组件层级直接传递变量，实现共享。需要注意的是useContext和redux的作用是不同的，一个解决的是组件之间值传递的问题，一个是应用中统一管理状态的问题，但通过和useReducer的配合使用，可以实现类似Redux的作用。

这就好比玩游戏时有很多英雄，英雄的最总目的都是赢得比赛，但是作用不同，有负责输出的，有负责抗伤害的，有负责治疗的。

Context的作用就是对它所包含的组件树提供全局共享数据的一种技术。

createContext 函数创建context

直接在src目录下新建一个文件Example4.js,然后拷贝Example.js里的代码，并进行修改，删除路由部分和副作用的代码，只留计数器的核心代码就可以了。

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

function Example4(){
    const [ count , setCount ] = useState(0);
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={()=>{setCount(count+1)}}>click me</button>
        </div>
    )
}
export default Example4;
```

然后修改一下index.js让它渲染这个Example4.js组件，修改的代码如下。

```
import React from 'react';
import ReactDOM from 'react-dom';
import Example from './Example4'
ReactDOM.render(<Example />, document.getElementById('root'));
```

之后在Example4.js中引入createContext函数，并使用得到一个组件，然后在return方法中进行使用。先看代码，然后我再解释。

```
import React, { useState , createContext } from 'react';
//===关键代码
const CountContext = createContext()

function Example4(){
    const [ count , setCount ] = useState(0);

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={()=>{setCount(count+1)}}>click me</button>
            {/*======关键代码 */}
            <CountContext.Provider value={count}>
            </CountContext.Provider>

        </div>
    )
}
export default Example4;
```

这段代码就相当于把count变量允许跨层级实现传递和使用了（也就是实现了上下文），当父组件的count变量发生变化时，子组件也会发生变化。接下来我们就看看一个React Hooks的组件如何接收到这个变量。

useContext 接收上下文变量

已经有了上下文变量，剩下的就时如何接收了，接收这个直接使用useContext就可以，但是在使用前需要新进行引入useContext（不引入是没办法使用的）。

```
import React, { useState , createContext , useContext } from'react';
```

引入后写一个Counter组件，只是显示上下文中的count变量代码如下：

```
function Counter(){
    const count = useContext(CountContext)  //一句话就可以得到count
    return (<h2>{count}</h2>)
}
```

得到后就可以显示出来了，但是要记得在的闭合标签中,代码如下。

```
<CountContext.Provider value={count}>
    <Counter />
</CountContext.Provider>
```

P06:useReducer介绍和简单使用

上节课学习了useContext函数，那这节课开始学习一下useReducer，因为他们两个很像，并且合作可以完成类似的Redux库的操作。在开发中使用useReducer可以让代码具有更好的可读性和可维护性，并且会给测试提供方便。

reducer到底是什么?

为了更好的理解useReducer，所以先要了解JavaScript里的Redcuer是什么。它的兴起是从Redux广泛使用开始的，但不仅仅存在Redux中，可以使用冈的JavaScript来完成Reducer操作。那reducer其实就是一个函数，这个函数接收两个参数，一个是状态，一个用来控制业务逻辑的判断参数。我们举一个最简单的例子。

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

上面的代码就是Reducer，你主要理解的就是这种形式和两个参数的作用，一个参数是状态，一个参数是如何控制状态。

useReducer的使用

了解reducer的含义后，就可以讲useReducer了，它也是React hooks提供的函数，可以增强我们的Reducer，实现类似Redux的功能。我们新建一个Example5.js的文件，然后用useReducer实现计数器的加减双向操作。

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

function ReducerDemo(){
    const [ count , dispatch ] =useReducer((state,action)=>{
        switch(action){
            case 'add':
                return state+1
            case 'sub':
                return state-1
            default:
                return state
        }
    },0)
    return (
       <div>
           <h2>现在的分数是{count}</h2>
           <button onClick={()=>dispatch('add')}>Increment</button>
           <button onClick={()=>dispatch('sub')}>Decrement</button>
       </div>
    )

}

export default ReducerDemo
```

这段代码是useReducer的最简单实现了，这时候可以在浏览器中实现了计数器的增加减少。
修改index.js文件，让ReducerDemo组件起作用。

```
import React from 'react';
import ReactDOM from 'react-dom';
import Example from './Example5'


ReactDOM.render(<Example />, document.getElementById('root'));
```

P07:useReducer代替Redux小案例-1

使用useContext和useReducer是可以实现类似Redux的效果，并且一些简单的个人项目，完全可以用下面的方案代替Redux，这种做法要比Redux简单一些。因为useContext和useReducer在前两节课已经学习过了，所以我们这节课把精力就放在如何模拟出Redux的效果。

本节课程参考了掘金上缪宇的文章 ， 文章地址:https://juejin.im/post/5ceb37c851882520724c7504

理论上的可行性

我们先从理论层面看看替代Redux的可能性，其实如果你对两个函数有所了解，只要我们巧妙的结合，这种替代方案是完全可行的。

useContext：可访问全局状态，避免一层层的传递状态。这符合Redux其中的一项规则，就是状态全局化，并能统一管理。

useReducer：通过action的传递，更新复杂逻辑的状态，主要是可以实现类似Redux中的Reducer部分，实现业务逻辑的可行性。

经过我们在理论上的分析是完全可行的，接下来我们就用一个简单实例来看一下具体的实现方法。那这节课先实现useContext部分（也就是状态共享），下节再继续讲解useReducer部分（控制业务逻辑）。

编写基本UI组件

既然是一个实例，就需要有些界面的东西，小伙伴们不要觉的烦。在/src目录下新建一个文件夹Example6，有了文件夹后，在文件夹下面建立一个showArea.js文件。代码如下：

```
import React from 'react';
function ShowArea(){

    return (<div style={{color:'blue'}}>字体颜色为blue</div>)

}
export default ShowArea
```

显示区域写完后，新建一个Buttons.js文件，用来编写按钮，这个是两个按钮，一个红色一个黄色。先不写其他任何业务逻辑。

```
import React from 'react';

function Buttons(){
    return (
        <div>
            <button>红色</button>
```

```
            <button>黄色</button>
        </div>
    )
}

export default Buttons
```

然后再编写一个组合他们的`Example6.js`组件，引入两个新编写的组件`ShowArea`和`Buttons`，并用`<Fragment>`标签给包裹起来。

```
import React, { useReducer } from 'react';
import ShowArea from './ShowArea';
import Buttons from './Buttons';


function Example6(){
    return (
        <div>
                <ShowArea />
                <Buttons />
        </div>
    )
}

export default Example6
```

这步做完，需要到`/src`目录下的`index.js`中引入一下`Example6.js`文件，引入后React才能正确渲染出刚写的UI组件。

```
import React from 'react';
import ReactDOM from 'react-dom';
import Example from './Example6/Example6'


ReactDOM.render(<Example />, document.getElementById('root'));
```

编写颜色共享组件`color.js`

有了UI组件后，就可以写一些业务逻辑了，这节课我们先实现状态共享，这个就是利用useContext。建立一个color.js文件，然后写入下面的代码。

```
import React, { createContext } from 'react';

export const ColorContext = createContext({})

export const Color = props=>{
    return (
        <ColorContext.Provider value={{color:"blue"}}>
            {props.children}
        </ColorContext.Provider>
    )
}
```

代码中引入了createContext用来创建共享上下文ColorContext组件，然后我们要用{props.children}来显示对应的子组件。
有了这个组件后，我们就可以把Example6.js进行改写，让她可以共享状态。

```
import React, { useReducer } from 'react';
import ShowArea from './ShowArea';
import Buttons from './Buttons';
import { Color } from './color';   //引入Color组件

function Example6(){
    return (
        <div>
            <Color>
                <ShowArea />
                <Buttons />
            </Color>

        </div>
    )
}

export default Example6
```

然后再改写showArea.js文件，我们会引入useContext和在color.js中声明的ColorContext，让组件可以接收全局变量。

```
import React , { useContext } from 'react';
import { ColorContext } from './color';

function ShowArea(){
    const {color} = useContext(ColorContext)
    return (<div style={{color:color}}>字体颜色为{color}</div>)

}

export default ShowArea
```

这时候就通过useContext实现了状态的共享，可以到浏览器中看一下效果。

P08:useReducer代替Redux小案例-2

用useContext实现了Redux状态共享的能力，这节课看一下如何使用useReducer来实现业务逻辑的控制。

在color.js中添加Reducer

颜色（state）管理的代码我们都放在了color.js中，所以在文件里添加一个reducer，用于处理颜色更新的逻辑。先声明一个reducer的函数，它就是JavaScript中的普通函数，在讲useReducer的时候已经详细讲过了。有了reducer后，在Color组件里使用useReducer,这样Color组件就有了那个共享状态和处理业务逻辑的能力，跟以前使用的Redux几乎一样了。之后修改一下共享状态。我们来看代码：

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

export const ColorContext = createContext({})

export const UPDATE_COLOR = "UPDATE_COLOR"

const reducer= (state,action)=>{
    switch(action.type){
        case UPDATE_COLOR:
            return action.color
        default:
            return state
    }
}
```

```
15
16
17 export const Color = props=>{
18     const [color,dispatch]=useReducer(reducer,'blue')
19     return (
20         <ColorContext.Provider value={{color,dispatch}}>
21             {props.children}
22         </ColorContext.Provider>
23     )
24 }
```

注意，这时候我们共享出去的状态变成了color和dispatch,如果不共享出去dispatch，你是没办法完成按钮的相应事件的。

通过dispatch修改状态

目前程序已经有了处理共享状态的业务逻辑能力，接下来就可以在`buttons.js`使用`dispatch`来完成按钮的相应操作了。先引入`useContext`、`ColorContext`和`UPDATE_COLOR`，然后写`onClick`事件就可以了。代码如下:

```
1  import React ,{useContext} from 'react';
2  import {ColorContext,UPDATE_COLOR} from './color'
3
4  function Buttons(){
5      const { dispatch } = useContext(ColorContext)
6      return (
7          <div>
8              <button onClick={()=>{dispatch({type:UPDATE_COLOR,color:"red"})}}>红色
9              <button onClick={()=>{dispatch({type:UPDATE_COLOR,color:"yellow"})}}>
10         </div>
11     )
12 }
13
14 export default Buttons
```

这样代码就编写完成了，用`useContext`和`useReducer`实现了Redux的效果，这个代码编写过程比Redux要简单。

P09:useMemo优化React Hooks程序性能

`useMemo`主要用来解决使用React hooks产生的无用渲染的性能问题。使用function的形式来声明组件，失去了`shouldCompnentUpdate`（在组件更新之前）这个生命周期，也就是说我们没有办法通过组件

更新前条件来决定组件是否更新。而且在函数组件中，也不再区分mount和update两个状态，这意味着函数组件的每一次调用都会执行内部的所有逻辑，就带来了非常大的性能损耗。useMemo和useCallback都是解决上述性能问题的，这节课先学习useMemo.

性能问题展示案例

先编写一下刚才所说的性能问题，建立两个组件,一个父组件一个子组件，组件上由两个按钮，一个是小红，一个是志玲，点击哪个，那个就像我们走来了。在/src文件夹下，新建立一个Example7的文件夹，在文件夹下建立一个Example7.js文件.然后先写第一个父组件。

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

function Example7(){
    const [xiaohong , setXiaohong] = useState('小红待客状态')
    const [zhiling , setZhiling] = useState('志玲待客状态')
    return (
        <>
            <button onClick={()=>{setXiaohong(new Date().getTime())}}>小红</buttor
            <button onClick={()=>{setZhiling(new Date().getTime()+',志玲向我们走来了
            <ChildComponent name={xiaohong}>{zhiling}</ChildComponent>
        </>
    )
}
```

父组件调用了子组件，子组件我们输出两个姑娘的状态，显示在界面上。代码如下：

```
function ChildComponent({name,children}){
    function changeXiaohong(name){
        console.log('她来了，她来了。小红向我们走来了')
        return name+',小红向我们走来了'
    }

    const actionXiaohong = changeXiaohong(name)
    return (
        <>
            <div>{actionXiaohong}</div>
            <div>{children}</div>
        </>
    )
}
```

然后再导出父组件，让index.js可以渲染。

```
export default Example7
```

这时候你会发现在浏览器中点击志玲按钮，小红对应的方法都会执行，结果虽然没变，但是每次都执行，这就是性能的损耗。目前只有子组件，业务逻辑也非常简单，如果是一个后台查询，这将产生严重的后果。所以这个问题必须解决。当我们点击志玲按钮时，小红对应的changeXiaohong方法不能执行，只有在点击小红按钮时才能执行。

useMemo 优化性能

其实只要使用useMemo，然后给她传递第二个参数，参数匹配成功，才会执行。代码如下：

```
function ChildComponent({name,children}){
    function changeXiaohong(name){
        console.log('她来了，她来了。小红向我们走来了')
        return name+',小红向我们走来了'
    }

    const actionXiaohong = useMemo(()=>changeXiaohong(name),[name])
    return (
        <>
            <div>{actionXiaohong}</div>
            <div>{children}</div>
        </>
    )
}
```

这时在浏览器中点击一下志玲按钮，changeXiaohong就不再执行了。也节省了性能的消耗。案例只是让你更好理解，你还要从程序本身看到优化的作用。好的程序员对自己写的程序都是会进行不断优化的，这种没必要的性能浪费也是绝对不允许的，所以useMemo的使用在工作中还是比较多的。希望小伙伴们可以掌握。

P10:useRef获取DOM元素和保存变量

useRef在工作中虽然用的不多，但是也不能缺少。它有两个主要的作用：

- 用useRef获取React JSX中的DOM元素，获取后你就可以控制DOM的任何东西了。但是一般不建议这样来作，React界面的变化可以通过状态来控制。
- 用useRef来保存变量，这个在工作中也很少能用到，我们有了useContext这样的保存其实意义不大，但是这是学习，也要把这个特性讲一下。

useRef获取DOM元素

界面上有一个文本框，在文本框的旁边有一个按钮，当我们点击按钮时，在控制台打印出input的DOM元素，并进行复制到DOM中的value上。这一切都是通过useRef来实现。
在/src文件夹下新建一个Example8.js文件，然后先引入useRef，编写业务逻辑代码如下:

```
import React, { useRef} from 'react';
function Example8(){
    const inputEl = useRef(null)
    const onButtonClick=()=>{
        inputEl.current.value="Hello ,JSPang"
        console.log(inputEl) //输出获取到的DOM节点
    }
    return (
        <>
            {/*保存input的ref到inputEl */}
            <input ref={inputEl} type="text"/>
            <button onClick = {onButtonClick}>在input上展示文字</button>
        </>
    )
}
export default Example8
```

当点击按钮时，你可以看到在浏览器中的控制台完整的打印出了DOM的所有东西，并且界面上的框的value值也输出了我们写好的Hello ,JSPang。这一切说明我们可以使用useRef获取DOM元素，并且可以通过useRefu控制DOM的属性和值。

useRef保存普通变量

这个操作在实际开发中用的并不多，但我们还是要讲解一下。就是useRef可以保存React中的变量。我们这里就写一个文本框，文本框用来改变text状态。又用useRef把text状态进行保存，最后打印在控制台上。写这段代码你会觉的很绕，其实显示开发中没必要这样写，用一个state状态就可以搞定，这里只是为了展示知识点。
接着上面的代码来写，就没必要重新写一个文件了。先用useState声明了一个text状态和setText函数。然后编写界面，界面就是一个文本框。然后输入的时候不断变化。

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

function Example8(){
    const inputEl = useRef(null)
    const onButtonClick=()=>{
        inputEl.current.value="Hello ,useRef"
        console.log(inputEl)
```

```
    }
    const [text, setText] = useState('jspang')
    return (
        <>
            {/*保存input的ref到inputEl */}
            <input ref={inputEl} type="text"/>
            <button onClick = {onButtonClick}>在input上展示文字</button>
            <br/>
            <br/>
            <input value={text} onChange={(e)=>{setText(e.target.value)}} />

        </>
    )
}

export default Example8
```

这时想每次text发生状态改变，保存到一个变量中或者说是useRef中，这时候就可以使用useRef了。先声明一个textRef变量，他其实就是useRef函数。然后使用useEffect函数实现每次状态变化都进行变量修改，并打印。最后的全部代码如下。

```
import React, { useRef ,useState,useEffect } from 'react';
function Example8(){
    const inputEl = useRef(null)
    const onButtonClick=()=>{
        inputEl.current.value="Hello ,useRef"
        console.log(inputEl)
    }
    //-----------关键代码--------start
    const [text, setText] = useState('jspang')
    const textRef = useRef()

    useEffect(()=>{
        textRef.current = text;
        console.log('textRef.current:', textRef.current)
    })
    //----------关键代码--------------end
    return (
        <>
```

```
            {/*保存input的ref到inputEl */}
            <input ref={inputEl} type="text"/>
            <button onClick = {onButtonClick}>在input上展示文字</button>
            <br/>
            <br/>
            <input value={text} onChange={(e)=>{setText(e.target.value)}} />
        </>
    )
}

export default Example8
```

这时候就可以实现每次状态修改，同时保存到useRef中了。也就是我们说的保存变量的功能。那useRef的主要功能就是获得DOM和变量保存，我们都已经讲过了。你的编码能力有增加了一些，让我们一起加油。

P11: 自定义Hooks函数获取窗口大小

其实自定义Hooks函数和用Hooks创建组件很相似，跟我们平时用JavaScript写函数几乎一模一样，可能就是多了些React Hooks的特性，自定义Hooks函数偏向于功能，而组件偏向于界面和业务逻辑。由于差别不大，所以使用起来也是很随意的。如果是小型项目是可以的，但是如果项目足够复杂，这会让项目结构不够清晰。所以学习自定义Hooks函数还是很有必要的。

编写自定义函数

在实际开发中，为了界面更加美观。获取浏览器窗口的尺寸是一个经常使用的功能，这样经常使用的功能，就可以封装成一个自定义Hooks函数，记住一定要用use开头，这样才能区分出什么是组件，什么是自定义函数。

新建一个文件Example9.js,然后编写一个useWinSize,编写时我们会用到useState、useEffect和useCallback所以先用import进行引入。

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

然后编写函数，函数中先用useState设置size状态，然后编写一个每次修改状态的方法onResize，这个方法使用useCallback，目的是为了缓存方法(useMemo是为了缓存变量)。然后在第一次进入方法时用useEffect来注册resize监听时间。为了防止一直监听所以在方法移除时，使用return的方式移除监听。最后返回size变量就可以了。

```
function useWinSize(){
    const [ size , setSize] = useState({
        width:document.documentElement.clientWidth,
```

```
        height:document.documentElement.clientHeight
```

```
    })

    const onResize = useCallback(()=>{
        setSize({
            width: document.documentElement.clientWidth,
            height: document.documentElement.clientHeight
        })
    },[])
    useEffect(()=>{
        window.addEventListener('resize',onResize)
        return ()=>{
            window.removeEventListener('resize',onResize)
        }
    },[])

    return size;

}functionuseWinSize(){    const [ size , setSize] = useState({        width:docum
```

这就是一个自定义函数，其实和我们以前写的JS函数没什么区别，所以这里也不做太多的介绍。

编写组件并使用自定义函数

自定义Hooks函数已经写好了，可以直接进行使用，用法和JavaScript的普通函数用起来是一样的。直接在Example9组件使用useWinSize并把结果实时展示在页面上。

```
function Example9(){

    const size = useWinSize()
    return (
        <div>页面Size:{size.width}x{size.height}</div>
    )
}

export default Example9
```

之后就可以在浏览器中预览一下结果，可以看到当我们放大缩小浏览器窗口时，页面上的结果都会跟着进行变化。说明自定义的函数起到了作用。