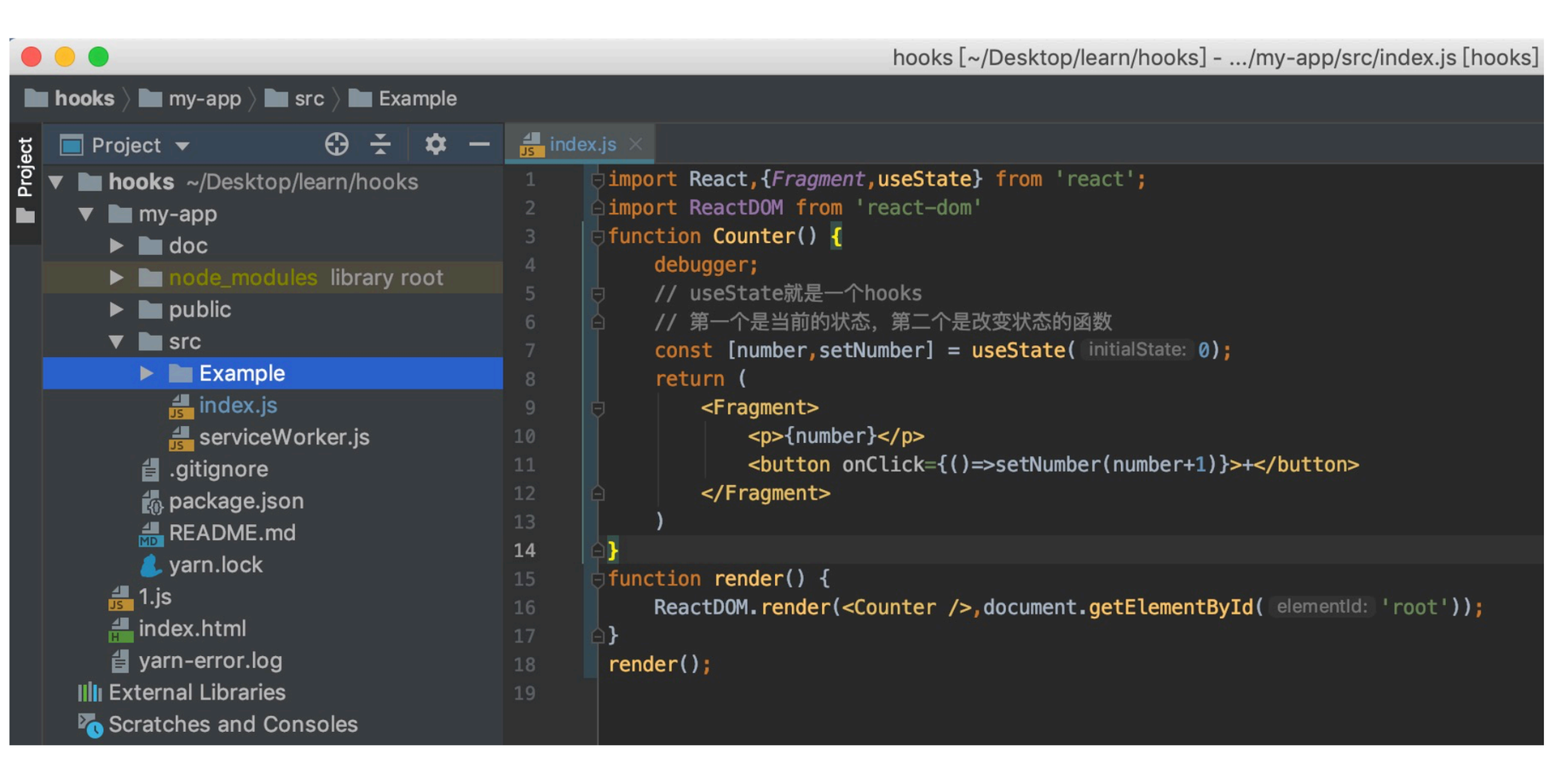
react hook用法和原理实现

1、hook的简介
2、useState的简版实现
3、userReducer简介和实现
4、多个useState同时调用
5、useEffect简介与实现

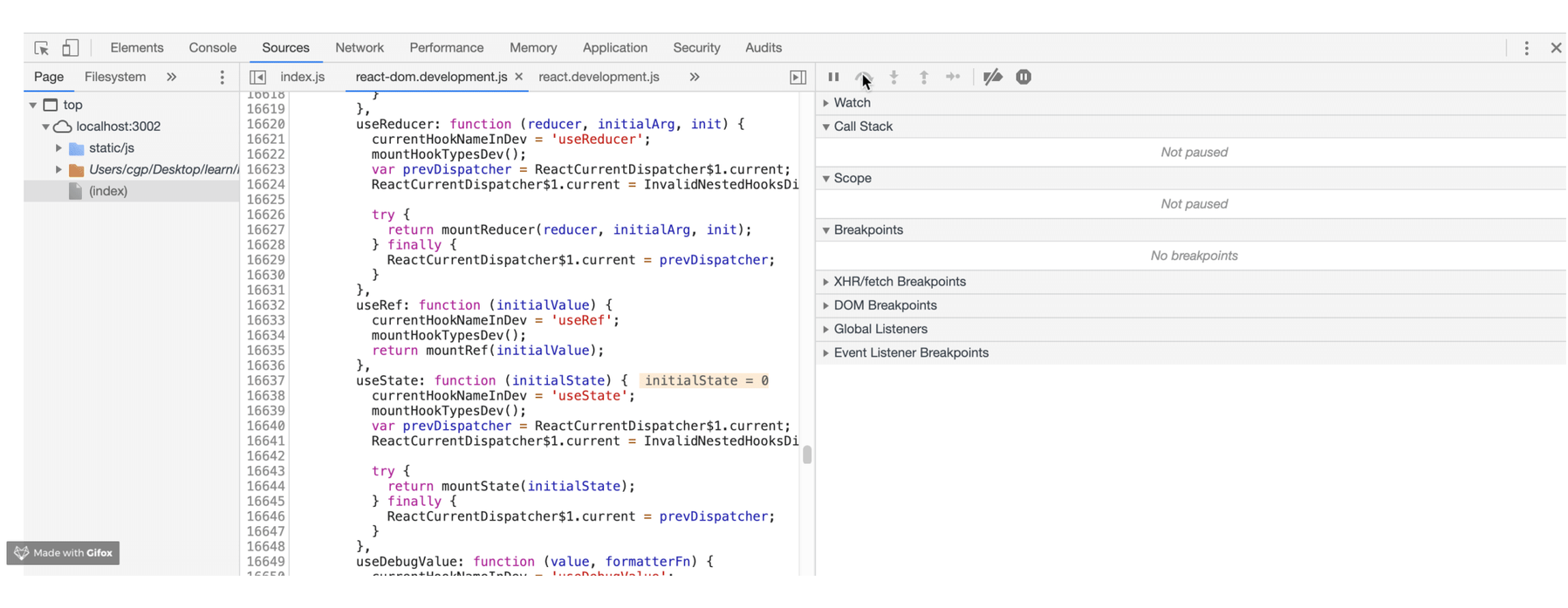
1、hook的简介

> 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。

直接贴code,hook实现简版计数器



简单debugger下代码



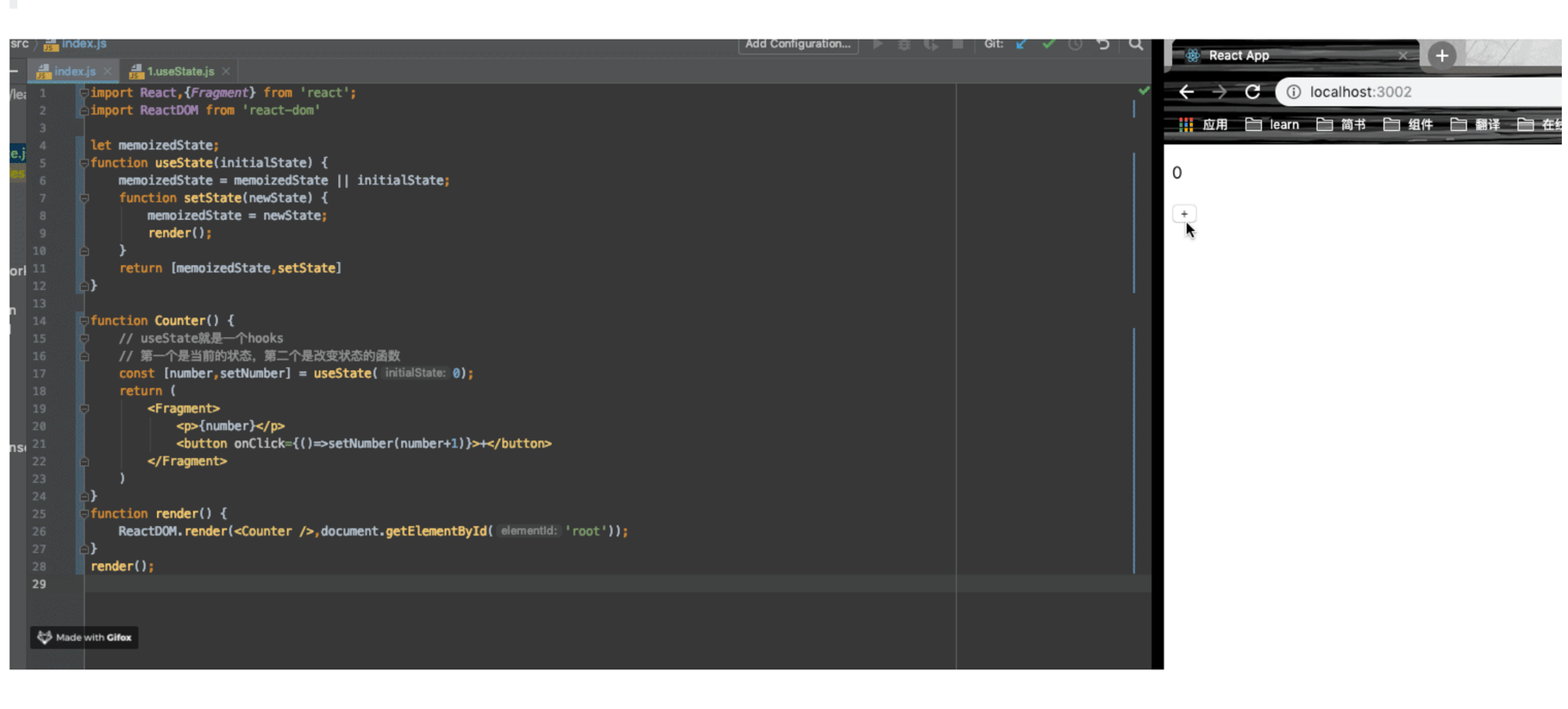
- 我们可以看到hook是用memorizedState来保存状态

2、useState的简版实现

- 核心作用是给函数组件增加了一个保持状态的功能

```
let memoizedState;  //声明memoizedState
function useState(initialState) {
    memoizedState = memoizedState || initialState;
    function setState(newState) {
        memoizedState = newState; //设置状态时候把新状态赋值给memoizedState
        render(); //重新render
    }
    return [memoizedState,setState]
}
```

> 从简版的实现来看，还是很容易理解，测试下效果



3、userReducer简介和实现

3.1、userReducer简介

- useState 的替代方案。它接收一个形如 (state, action) => newState 的 reducer，并返回当前的 state 以及与其配套的 dispatch 方法。（如果你熟悉 Redux 的话，就已经知道它如何工作了。）
- 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。

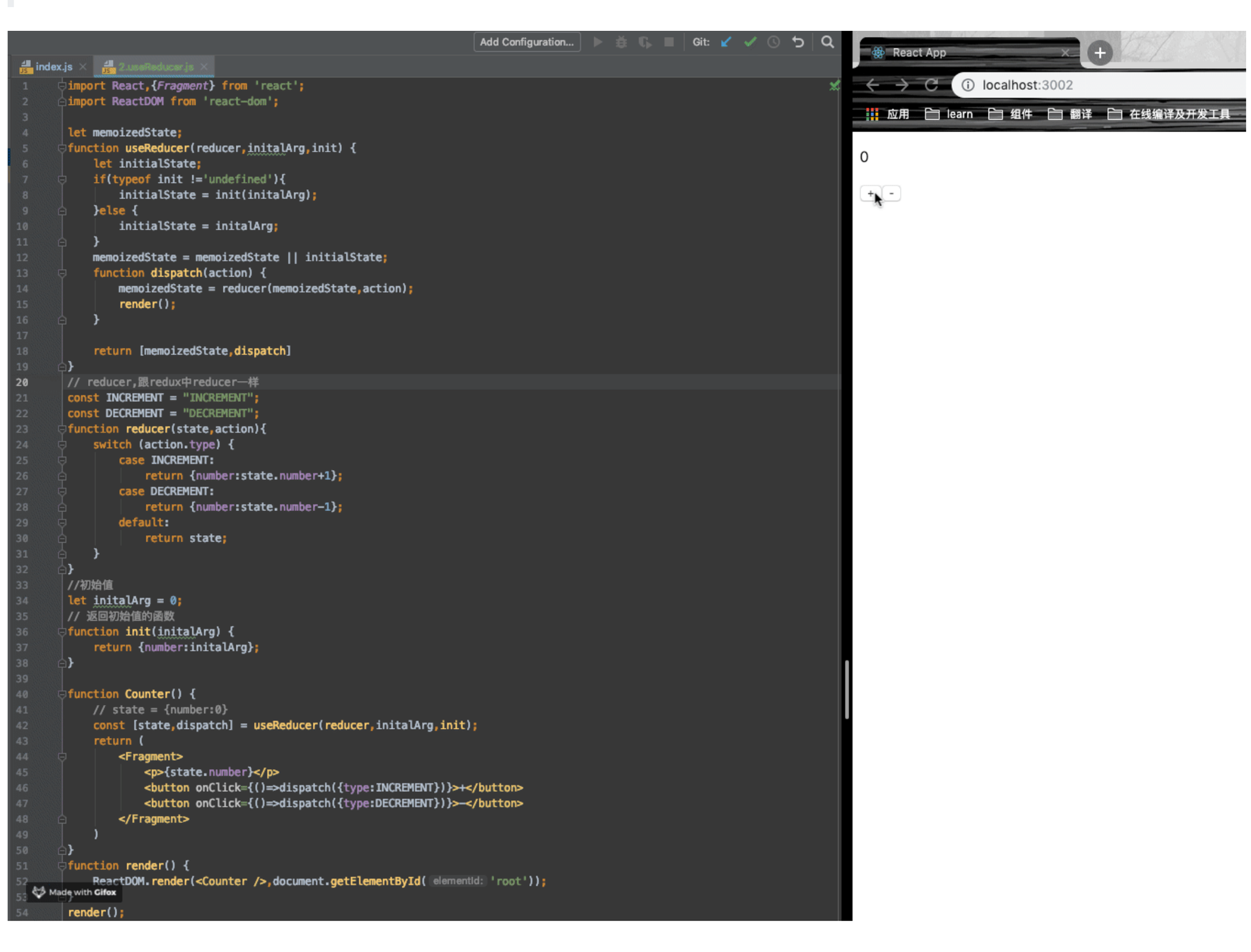
> 用userReducer实现计数器，可接收3个参数，reducer | initalArg(初始值) | init (定义初始值的函数)

```
import React,{Fragment,useReducer} from 'react';
import ReactDOM from 'react-dom';
// reducer,跟redux中reducer一样
const INCREMENT = "INCREMENT";
const DECREMENT = "DECREMENT";
function reducer(state,action){
    switch (action.type) {
        case INCREMENT:
            return {number:state.number+1};
        case DECREMENT:
            return {number:state.number-1};
        default:
            return state;
    }
}
//初始值
let initalArg = 0;
// 返回初始值的函数
function init(initalArg) {
    return {number:initalArg};
}

function Counter() {
    // state = {number:0}
    const [state,dispatch] = useReducer(reducer,initalArg,init);
    return (
        <Fragment>
            <p>{state.number}</p>
            <button onClick={()=>dispatch({type:INCREMENT})}>+</button>
            <button onClick={()=>dispatch({type:DECREMENT})}>-</button>
        </Fragment>
    )
}
function render() {
    ReactDOM.render(<Counter />,document.getElementById('root'));
}
render();
```

3.2、简版实现原理

```
let memoizedState; //声明记忆的状态
function useReducer(reducer,initalArg,init) {
    let initialState;
    // init如果没有传值，initalArg为默认的初始状态。如果传值，初始值函数处理后作为初始状态
    if(typeof init !='undefined'){
        initialState = init(initalArg);
    }else {
        initialState = initalArg;
    }
    memoizedState = memoizedState || initialState;
    function dispatch(action) {
        memoizedState = reducer(memoizedState,action);
        render();
    }

    return [memoizedState,dispatch]
}
```

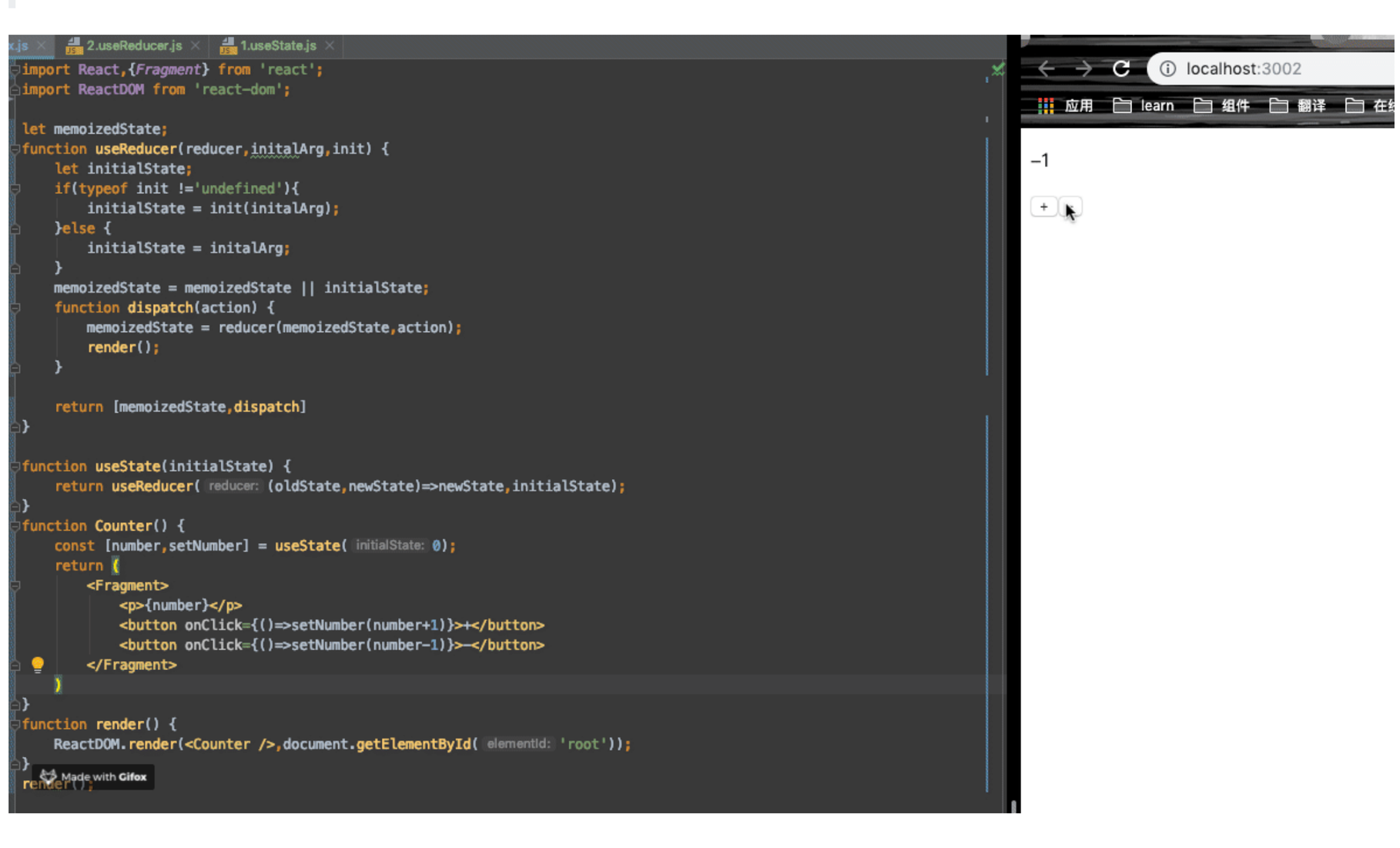
> 运行结果



3.3、useReducer是useState的内部实现，重写useState实现

```
let memoizedState;
function useReducer(reducer,initalArg,init) {
    let initialState;
    if(typeof init !='undefined'){
        initialState = init(initalArg);
    }else {
        initialState = initalArg;
    }
    memoizedState = memoizedState || initialState;
    function dispatch(action) {
        memoizedState = reducer(memoizedState,action);
        render();
    }

    return [memoizedState,dispatch]
}

function useState(initialState) {
    // 主要是reducer实现，把新状态赋值过去
    return useReducer((oldState,newState)=>newState,initialState);
}
```

> 验证一下



4、多个useState同时调用

> 当一个组件调用多个useState时，此时我们需要用数组来保存多个初始值

4.1、多个useState使用的示例demo

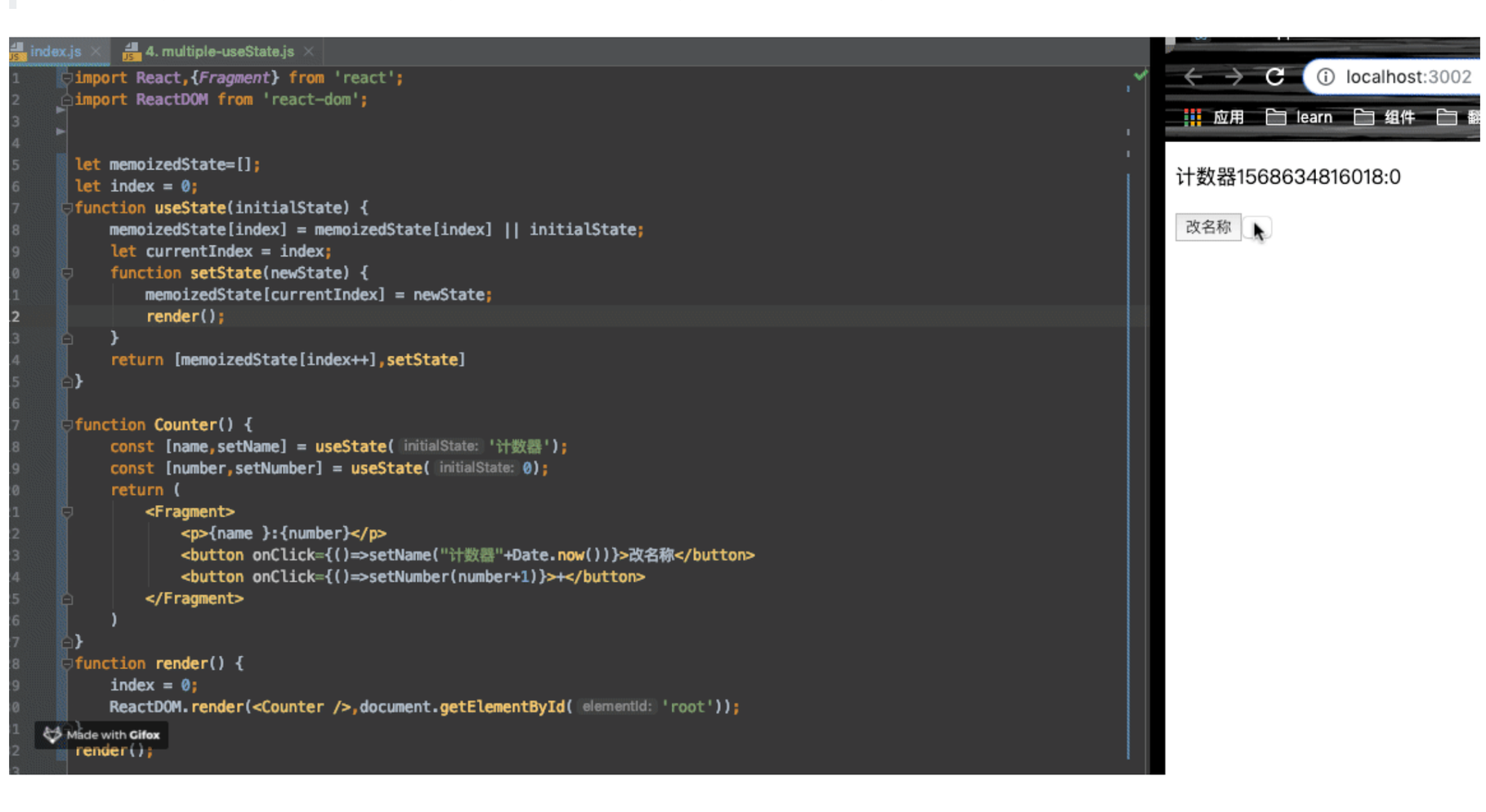
- 两个按钮，一个改变name，一个改变number

4.2、实现原理

- 之前都是使用一个useState，当多个useState时候，需要用数组保存所有的初始状态
- 需要用index记录当前的索引
- 每次render时候，index索引需要回复初始值

```
import React,{Fragment} from 'react';
import ReactDOM from 'react-dom';

// 数组保存memoizedState
let memoizedState=[];
// 记录索引
let index = 0;
function useState(initialState) {
    memoizedState[index] = memoizedState[index] || initialState;
    // 缓存当前索引，因为每次render，index索引会重置为0
    let currentIndex = index;
    function setState(newState) {
        memoizedState[currentIndex] = newState;
        render();
    }
    return [memoizedState[index++],setState]
}

function Counter() {
    const [name,setName] = useState('计数器');
    const [number,setNumber] = useState(0);
    return (
        <Fragment>
            <p>{name }:{number}</p>
            <button onClick={()=>setName("计数器"+Date.now())}>改名称</button>
            <button onClick={()=>setNumber(number+1)}>+</button>
        </Fragment>
    )
}
function render() {
    // 每次render，把index回复初始值
    index = 0;
    ReactDOM.render(<Counter />,document.getElementById('root'));
}
render();
```

> 看下效果，源码是用链表实现的，此处我们用数组，逻辑是差不多，容易理解



5、useEffect简介与实现

5.1、简介

- useEffect 给函数组件添加了操作副作用的能力， 比如事件的订阅与取消、定时器的设置与清空
- 类似于在类组件生命周期 componentDidMount、componentWillUnmount做的事情

> 计数器变化后，实现打印一句log的示例
> 参考链接

```
import React,{Fragment,useState} from 'react';
import ReactDOM from 'react-dom';

function Counter() {
    const [name,setName] = useState('计数器');
    const [number,setNumber] = useState(0);
    useEffect(()=>{
        // 订阅
        console.log("订阅状态")
    },[number,name]);
    return (
        <Fragment>
            <p>{name }:{number}</p>
            <button onClick={()=>setName("计数器"+Date.now())}>改名称</button>
            <button onClick={()=>setNumber(number+1)}>+</button>
        </Fragment>
    )
}
function render() {
    ReactDOM.render(<Counter />,document.getElementById('root'));
}
render();
```

5.2、简版实现

- useEffect第二个参数为依赖项，即当依赖项改变时，才会触发回调

> 简版实现

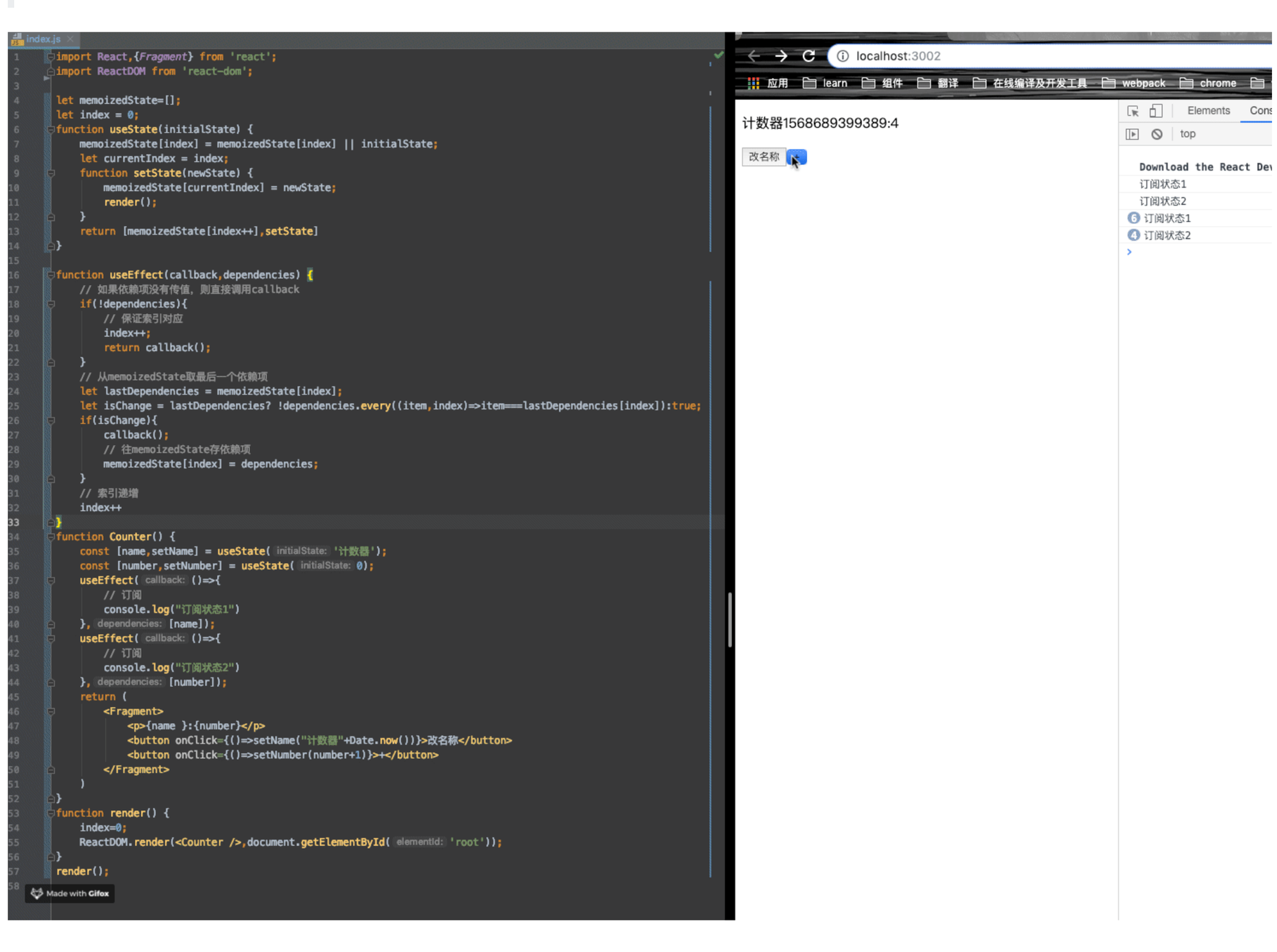
```
// 记录最后依赖项
let lastDependencies;
function useEffect(callback,dependencies) {
    // 如果依赖项没有传值，则直接调用callback
    if(!dependencies) return callback();
    /* 1、首次渲染isChange为true，把初始的依赖项赋给lastDependencies
    *  2、再次渲染时候，把lastDependencies和dependencies做对比，当不完全相等时，才触发回调
    */
    let isChange = lastDependencies? !dependencies.every((item,index)=>item===lastDependencies[
    if(isChange){
        callback();
        lastDependencies = dependencies;
    }
}
```

5.3、当有多个useEffect时，如何实现

- 统一把lastDependencies放到useState中的memoizedState中

```
let memoizedState=[];
let index = 0;
function useState(initialState) {
    memoizedState[index] = memoizedState[index] || initialState;
    let currentIndex = index;
    function setState(newState) {
        memoizedState[currentIndex] = newState;
        render();
    }
    return [memoizedState[index++],setState]
}

function useEffect(callback,dependencies) {
    console.log('dependencies',dependencies);
    // 如果依赖项没有传值，则直接调用callback
    if(!dependencies){
        // 保证索引对应
        index++;
        return callback();
    }
    // 从memoizedState取最后一个依赖项
    let lastDependencies = memoizedState[index];
    let isChange = lastDependencies? !dependencies.every((item,index)=>item===lastDependencies[
    if(isChange){
        callback();
        // 往memoizedState存依赖项
        memoizedState[index] = dependencies;
    }
    // 索引递增
    index++
}
```

> 验证下效果



后续继续补充