link: null
title: 珠峰架构师成长计划
description: src\react-dom.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=76 sentences=162, words=1531

1. React Hooks

- Hook可以让你在不编写 `class` 的情况下使用 `state` 以及其他的 React 特性

2. useState

- useState 就是一个 Hook
- 通过在函数组件里调用它来给组件添加一些内部 state,React 会在重复渲染时保留这个 state
- useState 会返回一对值：当前状态和一个让你更新它的函数，你可以在事件处理函数中或其他一些地方调用这个函数。它类似 class 组件的 this.setState，但是它不会把新的 state 和旧的 state 进行合并
- useState 唯一的参数就是初始 state
- 返回一个 state，以及更新 state 的函数
    - 在初始渲染期间，返回的状态 (state) 与传入的第一个参数 (initialState) 值相同
    - setState 函数用于更新 state。它接收一个新的 state 值并将组件的一次重新渲染加入队列

2.1 计数器

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

function App(){
  const[number,setNumber]=React.useState(0);
  let handleClick = ()=> setNumber(number+1)
  return (
    <div>
      <p>{number}p>
      <button onClick={handleClick}>+button>
    div>
  )
}

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

2.2 src\react-dom.js

src\react-dom.js

```
+let hookStates = [];
+let hookIndex = 0;
+let scheduleUpdate;
+function render(vdom, container) {
+    mount(vdom,container);
+    scheduleUpdate = ()=>{
+      hookIndex = 0;
+      compareTwoVdom(container,vdom,vdom);
+    }
+}
+export function useState(initialState){
+    hookStates[hookIndex] = hookStates[hookIndex]||initialState;
+    let currentIndex = hookIndex;
+    function setState(newState){
+      let newState = typeof action === 'function' ? action(oldState) : action;
+      hookStates[currentIndex] = newState;
+      scheduleUpdate();
+    }
+    return [hookStates[hookIndex++],setState];
+}
```

2.3 src\react.js

src\react.js

```
+import * as hooks from './react-dom';

const React = {
    createElement,
    Component,
    PureComponent,
    createRef,
    createContext,
    cloneElement,
    memo,
+    ...hooks
};
export default React;
```

3.useCallback+useMemo

- 把内联回调函数及依赖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赖项改变时才会更新
- 把创建函数和依赖项数组作为参数传入 `useMemo`，它仅会在某个依赖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

3.1 src\index.js

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

let  Child = ({data,handleClick})=>{
  console.log('Child render');
  return (
     <button onClick={handleClick}>{data.number}button>
  )
}
Child = React.memo(Child);

function App(){
  console.log('App render');
  const[name,setName]=React.useState('zhufeng');
  const[number,setNumber]=React.useState(0);
  let data = React.useMemo(()=>({number}),[number]);
  let handleClick = React.useCallback(()=> setNumber(number+1),[number]);
  return (
    <div>
      <input type="text" value={name} onChange={event=>setName(event.target.value)}/>
      <Child data={data} handleClick={handleClick}/>
    div>
  )
}

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

3.2 src\react-dom.js

src\react-dom.js

```
let hookStates = [];
let hookIndex = 0;
let scheduleUpdate;
function render(vdom,container){
   mount(vdom,container);
   scheduleUpdate = ()=>{
    hookIndex = 0;
    compareTwoVdom(container,vdom,vdom);
   }
}
export function useState(initialState){
    hookStates[hookIndex] = hookStates[hookIndex]||initialState;
    let currentIndex = hookIndex;
    function setState(newState){
      if(typeof newState
      hookStates[currentIndex]=newState;
      scheduleUpdate();
    }
    return [hookStates[hookIndex++],setState];
  }
+export  function useMemo(factory,deps){
+    if(hookStates[hookIndex]){
+      let [lastMemo,lastDeps] = hookStates[hookIndex];
+      let same = deps.every((item,index)=>item === lastDeps[index]);
+      if(same){
+        hookIndex++;
+        return lastMemo;
+      }else{
+        let newMemo = factory();
+        hookStates[hookIndex++]=[newMemo,deps];
+        return newMemo;
+      }
+    }else{
+      let newMemo = factory();
+      hookStates[hookIndex++]=[newMemo,deps];
+      return newMemo;
+    }
+}
+export function useCallback(callback,deps){
+    if(hookStates[hookIndex]){
+      let [lastCallback,lastDeps] = hookStates[hookIndex];
+      let same = deps.every((item,index)=>item === lastDeps[index]);
+      if(same){
+        hookIndex++;
+        return lastCallback;
+      }else{
+        hookStates[hookIndex++]=[callback,deps];
+        return callback;
+      }
+    }else{
+      hookStates[hookIndex++]=[callback,deps];
+      return callback;
+    }
+}
const ReactDOM =  {
    render
};
export default ReactDOM;
```

4. useReducer

- 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
- 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

4.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
function reducer(state={number:0}, action) {
  switch (action.type) {
    case 'ADD':
      return {number: state.number + 1};
    case 'MINUS':
      return {number: state.number - 1};
    default:
      return state;
  }
}

function Counter(){
    const [state, dispatch] = React.useReducer(reducer,{number:0});
    return (
        <div>
          Count: {state.number}
          <button onClick={() => dispatch({type: 'ADD'})}>+button>
          <button onClick={() => dispatch({type: 'MINUS'})}>-button>
        div>
    )
}
ReactDOM.render(
  <Counter/>,
  document.getElementById('root')
);
```

4.2 src\react-dom.js

src\react-dom.js

```
+export function useReducer(reducer, initialState) {
+    hookStates[hookIndex] = hookStates[hookIndex] || initialState;
+    let currentIndex = hookIndex;
+    function dispatch(action) {
+        //1.获取老状态
+        let oldState = hookStates[currentIndex];
+        //如果有reducer就使用reducer计算新状态
+        if (reducer) {
+            let newState = reducer(oldState, action);
+            hookStates[currentIndex] = newState;
+        } else {
+            //判断action是不是函数，如果是传入老状态，计算新状态
+            let newState = typeof action === 'function' ? action(oldState) : action;
+            hookStates[currentIndex] = newState;
+        }
+        scheduleUpdate();
+    }
+    return [hookStates[hookIndex++], dispatch];
+}
const ReactDOM =  {
    render
};
export default ReactDOM;
```

5. useContext

- 接收一个 context 对象（React.createContext 的返回值）并返回该 context 的当前值
- 当前的 context 值由上层组件中距离当前组件最近的 `<mycontext.provider></mycontext.provider>` 的 value prop 决定
- 当组件上层最近的 `<mycontext.provider></mycontext.provider>` 更新时，该 Hook 会触发重渲染，并使用最新传递给 `MyContext provider` 的 context value 值
- useContext(MyContext) 相当于 class 组件中的 `static contextType = MyContext` 或者 `<mycontext.consumer></mycontext.consumer>`
- useContext(MyContext) 只是让你能够读取 context 的值以及订阅 context 的变化。你仍然需要在上层组件树中使用 `<mycontext.provider></mycontext.provider>` 来为下层组件提供 context

5.1 src\index.js

src\index.js

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

const CounterContext = React.createContext();

function reducer(state, action) {
  switch (action.type) {
    case 'add':
      return {number: state.number + 1};
    case 'minus':
      return {number: state.number - 1};
    default:
      return state;
  }
}
function Counter(){
  let {state,dispatch} = React.useContext(CounterContext);
  return (
      <div>
        <p>{state.number}p>
        <button onClick={() => dispatch({type: 'add'})}>+button>
        <button onClick={() => dispatch({type: 'minus'})}>-button>
      div>
  )
}
function App(){
    const [state, dispatch] = React.useReducer(reducer, {number:0});
    return (
        <CounterContext.Provider value={{state,dispatch}}>
          <Counter/>
        CounterContext.Provider>
    )
}

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

5.2 src\react-dom.js

src\react-dom.js

```
+function useContext(context){
+  return context._currentValue;
+}
```

6. useEffect

- 在函数组件主体内（这里指在 React 渲染阶段）改变 DOM、添加订阅、设置定时器、记录日志以及执行其他包含副作用的操作都是不被允许的，因为这可能会产生莫名其妙的 bug 并破坏 UI 的一致性
- 使用 useEffect 完成副作用操作。赋值给 useEffect 的函数会在组件渲染到屏幕之后执行。你可以把 effect 看作从 React 的纯函数式世界通往命令式世界的逃生通道
- useEffect 就是一个 Effect Hook，给函数组件增加了操作副作用的能力。它跟 class 组件中的 `componentDidMount`、 `componentDidUpdate` 和 `componentWillUnmount` 具有相同的用途，只不过被合并成了一个 API
- 该 Hook 接收一个包含命令式、且可能有副作用代码的函数

6.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
function Counter() {
    const [number, setNumber] = React.useState(0);
    React.useEffect(() => {
        console.log('开启一个新的定时器')
        const $timer = setInterval(() => {
            setNumber(number => number + 1);
        }, 1000);
        return () => {
            console.log('销毁老的定时器');
            clearInterval($timer);
        }
    });
    return (
        <p>{number}p>
    )
}
ReactDOM.render(<Counter />, document.getElementById('root'));
```

6.2 src\react-dom.js

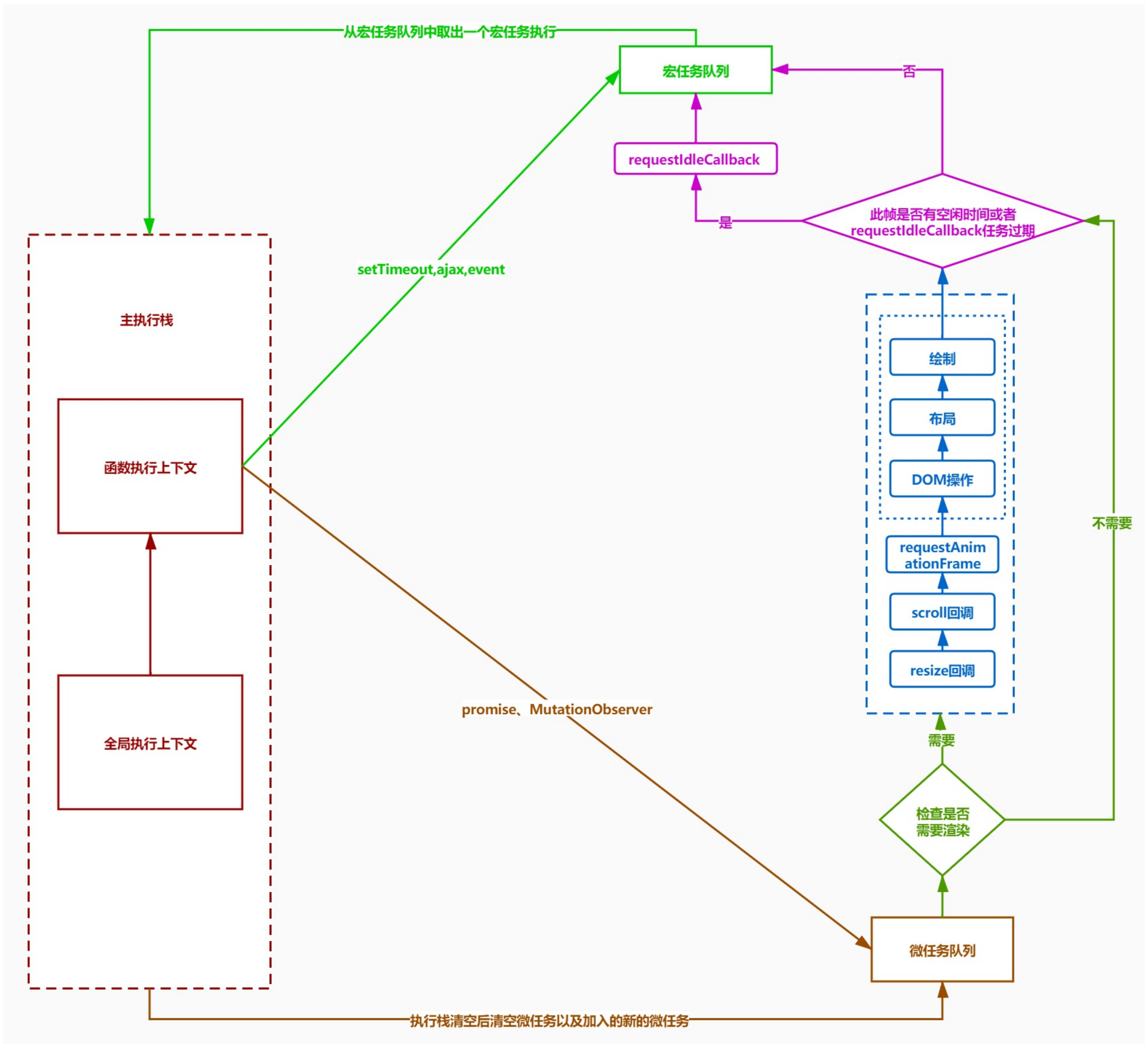
src\react-dom.js

```
+export function useEffect(callback,dependencies){
+  let currentIndex = hookIndex;
+  if(hookStates[hookIndex]){
+      let [destroy,lastDeps] = hookStates[hookIndex];
+      let same = dependencies&&dependencies.every((item,index)=>item === lastDeps[index]);
+      if(same){
+        hookIndex++;
+      }else{
+        destroy&&destroy();
+        setTimeout(()=>{
+            hookStates[currentIndex]=[callback(),dependencies];
+        });
+        hookIndex++;
+      }
+  }else{
+    setTimeout(()=>{
+        hookStates[currentIndex]=[callback(),dependencies];
+    });
+    hookIndex++;
+  }
+}
const ReactDOM =  {
    render
};
export default ReactDOM;
```

7. useLayoutEffect+useRef

- 其函数签名与 `useEffect` 相同，但它会在所有的 `DOM` 变更之后同步调用 effect
- `useEffect`不会阻塞浏览器渲染，而 `useLayoutEffect` 会浏览器渲染
- `useEffect`会在浏览器渲染结束后执行，`useLayoutEffect` 则是在 `DOM` 更新完成后,浏览器绘制之前执行

7.1 事件循环



7.2 src\index.js

src\index.js

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

const Animate = ()=>{
    const ref = React.useRef();
    React.useLayoutEffect(() => {
      ref.current.style.transform = `translate(500px)`;
      ref.current.style.transition = `all 500ms`;
    });
    let style = {
      width: '100px',
      height: '100px',
      borderRadius: '50%',
      backgroundColor: 'red'
    }
    return (
      <div style={style} ref={ref}>div>
    )
}
ReactDOM.render(<Animate/>,document.getElementById('root'));
```

7.3 src\react-dom.js

src\react-dom.js

```
+export function useLayoutEffect(callback,dependencies){
+    let currentIndex = hookIndex;
+    if(hookStates[hookIndex]){
+        let [destroy,lastDeps] = hookStates[hookIndex];
+        let same = dependencies&&dependencies.every((item,index)=>item === lastDeps[index]);
+        if(same){
+          hookIndex++;
+        }else{
+          destroy&&destroy();
+          queueMicrotask(()=>{
+              hookStates[currentIndex]=[callback(),dependencies];
+          });
+          hookIndex++
+        }
+    }else{
+      queueMicrotask(()=>{
+          hookStates[currentIndex]=[callback(),dependencies];
+      });
+       hookIndex++;
+    }
+}
+export function useRef(initialState) {
+    hookStates[hookIndex] =  hookStates[hookIndex] || { current: initialState };
+    return hookStates[hookIndex++];
+}
```

如何获取最新的state值

```
import React from 'react';
import ReactDOM from 'react-dom';
function Counter() {
  let valueRef = React.useRef();
  const [state, setState] = React.useState(0)
  const handleClick = () => {
    let newValue = state + 1;
    valueRef.current = newValue;
    setState(newValue)
    otherFun();
  }
  function otherFun() {
    console.log('state', valueRef.current);
  }
  return (
    <div>
      <p>state:{state}p>
      <button onClick={handleClick}>+button>
    div>
  )
}
ReactDOM.render(<Counter />, document.getElementById('root'));
```

8. forwardRef+useImperativeHandle

- forwardRef将ref从父组件中转发到子组件中的dom元素上,子组件接受props和ref作为参数
- `useImperativeHandle` 可以让你在使用 ref 时自定义暴露给父组件的实例值

8.1 src\index.js

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

function Child(props, ref) {
    const inputRef = React.useRef();
    React.useImperativeHandle(ref, () => (
        {
            focus() {
                inputRef.current.focus();
            }
        }
    ));
    return (
        <input type="text" ref={inputRef} />
    )
}
const ForwardChild = React.forwardRef(Child);
function Parent() {
    let [number, setNumber] = React.useState(0);
    const inputRef = React.useRef();
    function getFocus() {
        console.log(inputRef.current);
        inputRef.current.value = 'focus';
        inputRef.current.focus();
    }
    return (
        <div>
            <ForwardChild ref={inputRef} />
            <button onClick={getFocus}>获得焦点button>
            <p>{number}p>
            <button onClick={() => {
                debugger
                setNumber( number + 1)
            }}>+button>
        div>
    )
}
ReactDOM.render(<Parent/>,document.getElementById('root'));
```

8.2 src\react-dom.js

src\react-dom.js

```
function mountClassComponent(vdom){
+    const {type, props,ref} = vdom;
    const classInstance = new type(props);
+   if(ref){
+       ref.current = classInstance;
+       classInstance.ref = ref;
+   }
    vdom.classInstance=classInstance;
    if(type.contextType){
        classInstance.context = type.contextType.Provider._value;
    }
    if(classInstance.componentWillMount)
       classInstance.componentWillMount();
    classInstance.state = getDerivedStateFromProps(classInstance,classInstance.props,classInstance.state)
    const renderVdom = classInstance.render();
    classInstance.oldRenderVdom=vdom.oldRenderVdom=renderVdom;
    const dom = createDOM(renderVdom);
    if(classInstance.componentDidMount)
      dom.componentDidMount=classInstance.componentDidMount.bind(classInstance);
    return dom;
}

+export function useImperativeHandle(ref,handler){
+    ref.current = handler();
+}
const ReactDOM =  {
    render
};
export default ReactDOM;
```