

1. React Hooks

- 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
- 如果你在编写函数组件并意识到需要向其添加一些 state，以前的做法是必须将其转化为 class。现在你可以在现有的函数组件中使用 Hook

2. 解决的问题

- 在组件之间复用状态逻辑很难,可能要用到render props和高阶组件, React 需要为共享状态逻辑提供更好的原生途径, Hook 使你在无需修改组件结构的情况下复用状态逻辑
- 复杂组件变得难以理解, Hook 将组件中相互关联的部分拆分成更小的函数(比如设置订阅或请求数据)
- 难以理解的 class,包括难以捉摸的 `this`

3. 注意事项

- 只能在函数最外层调用 Hook。不要在循环、条件判断或者子函数中调用。
- 只能在 React 的函数组件中调用 Hook。不要在其他 JavaScript 函数中调用

4. 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

```
const [state, setState] = useState(initialState);
```

4.1 计数器

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

```

class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    };
  }
  render() {
    return (
      <div>
        <p>{this.state.number}</p>
        <button onClick={() => this.setState({ number: this.state.n
umber + 1 })}>
          +
        </button>
      </div>
    );
  }
}
function Counter2(){
  const [number,setNumber] = useState(0);
  return (
    <>
      <p>{number}</p>
      <button onClick={()=>setNumber(number+1)}>+</button>
    </>
  )
}
export default Counter2;

```

4.2 每次渲染都是独立的闭包

- 每一次渲染都有它自己的 Props and State
- 每一次渲染都有它自己的事件处理函数
- alert会“捕获”我点击按钮时候的状态。
- 我们的组件函数每次渲染都会被调用，但是每一次调用中number值都是常量，并且它被赋予了当前渲染中的状态值
- 在单次渲染的范围内，props和state始终保持不变

```

function Counter2(){
  const [number,setNumber] = useState(0);
  function alertNumber(){
    setTimeout(()=>{
      alert(number);
    },3000);
  }
  return (
    <>

```

```
    <p>{number}</p>
    <button onClick={()=>setNumber(number+1)}>+</button>
    <button onClick={alertNumber}>alertNumber</button>
  </>
)
}
```

4.3 函数式更新

- 如果新的 state 需要通过使用先前的 state 计算得出，那么可以将函数传递给 setState。该函数将接收先前的 state，并返回一个更新后的值

```
function Counter2(){
  const [number,setNumber] = useState(0);
  let numberRef = useRef(number);
  numberRef.current = number;
  function alertNumber(){
    setTimeout(()=>{
      alert(numberRef.current);
    },3000);
  }
+  function lazy(){
+    setTimeout(()=>{
+      setNumber(number+1);
+    },3000);
+  }
+  function lazyFunc(){
+    setTimeout(()=>{
+      setNumber(number=>number+1);
+    },3000);
+  }
  return (
    <>
    <p>{number}</p>
    <button onClick={()=>setNumber(number+1)}>+</button>
    <button onClick={lazy}>lazy</button>
    <button onClick={lazyFunc}>lazyFunc</button>
    <button onClick={alertNumber}>alertNumber</button>
  </>
)
}
```

4.4 惰性初始 state

- initialState 参数只会在组件的初始渲染中起作用，后续渲染时会被忽略

- 如果初始 state 需要通过复杂计算获得，则可以传入一个函数，在函数中计算并返回初始的 state，此函数只在初始渲染时被调用
- 与 class 组件中的 setState 方法不同，useState 不会自动合并更新对象。你可以用函数式的 setState 结合展开运算符来达到合并更新对象的效果

```
function Counter3(){
  const [{name,number},setValue] = useState(()=>{
    return {name:'计数器',number:0};
  });
  return (
    <>
      <p>{name}:{number}</p>
      <button onClick={()=>setValue({number:number+1})}>+</button>
    </>
  )
}
```

4.5 性能优化

4.5.1 Object.is

- 调用 State Hook 的更新函数并传入当前的 state 时，React 将跳过子组件的渲染及 effect 的执行。（React 使用 Object.is 比较算法 来比较 state。）

```
function Counter4(){
  const [counter,setCounter] = useState({name:'计数器',number:0});
  console.log('render Counter')
  return (
    <>
      <p>{counter.name}:{counter.number}</p>
      <button onClick={()=>setCounter({...counter,number:counter.number+1})}>+</button>
      <button onClick={()=>setCounter(counter)}>-</button>
    </>
  )
}
```

4.5.2 减少渲染次数

- 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

```
function Child({onButtonClick,data}){
```

```

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

```

5. 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

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

```

const initialState = 0;

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {number: state.number + 1};
    case 'decrement':
      return {number: state.number - 1};
    default:
      throw new Error();
  }
}

function init(initialState){
  return {number:initialState};
}

function Counter(){
  const [state, dispatch] = useReducer(reducer, initialState,init);
  return (

```

```

    <>
      Count: {state.number}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    >

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

```

6. 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> 的 value prop 决定
- 当组件上层最近的 <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>`
- 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> 来为下层组件提供 context

```

const CounterContext = React.createContext();

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {number: state.number + 1};
    case 'decrement':
      return {number: state.number - 1};
    default:
      throw new Error();
  }
}

function Counter(){
  let {state,dispatch} = useContext(CounterContext);
  return (
    <>
      <p>{state.number}</p>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </>
  )
}

function App(){
  const [state, dispatch] = useReducer(reducer, {number:0});
  return (

```

```
    <CounterContext.Provider value={{state,dispatch}}>
      <Counter/>
    </CounterContext.Provider>
  )
}
```

7. effect

- 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

```
useEffect(didUpdate);
```

7.1 通过class实现修标题

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    };
  }

  componentDidMount() {
    document.title = `你点击了${this.state.number}次`;
  }

  componentDidUpdate() {
    document.title = `你点击了${this.state.number}次`;
  }

  render() {
    return (
      <div>
        <p>{this.state.number}</p>
      </div>
    );
  }
}
```

```

      <button onClick={() => this.setState({ number: this.state.numbe
r + 1 })}>
        +
      </button>
    </div>
  );
}
}

```

在这个 class 中，我们需要在两个生命周期函数中编写重复的代码，这是因为很多情况下，我们希望在组件加载和更新时执行同样的操作。我们希望它在每次渲染之后执行，但 React 的 class 组件没有提供这样的方法。即使我们提取出一个方法，我们还是要在两个地方调用它。useEffect会在第一次渲染之后和每次更新之后都会执行

7.2 通过effect实现

```

import React, {Component, useState, useEffect} from 'react';
import ReactDOM from 'react-dom';
function Counter(){
  const [number, setNumber] = useState(0);
  // 相当于 componentDidMount 和 componentDidUpdate:
  useEffect(() => {
    // 使用浏览器的 API 更新页面标题
    document.title = `你点击了${number}次`;
  });
  return (
    <>
      <p>{number}</p>
      <button onClick={() => setNumber(number+1)}>+</button>
    </>
  )
}
ReactDOM.render(<Counter />, document.getElementById('root'));

```

每次我们重新渲染，都会生成新的 effect，替换掉之前的。某种意义上讲，effect 更像是渲染结果的一部分 —— 每个 effect 属于一次特定的渲染。

7.3 清除副作用

- 副作用函数还可以通过返回一个函数来指定如何清除副作用
- 为防止内存泄漏，清除函数会在组件卸载前执行。另外，如果组件多次渲染，则在执行下一个 effect 之前，上一个 effect 就已被清除


```
function Counter(){
  const [number,setNumber] = useState(0);
  // 相当于componentDidMount 和 componentDidUpdate
  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>
    </>
  )
}
```

7.4 跳过 Effect 进行性能优化

- 如果某些特定值在两次重渲染之间没有发生变化，你可以通知 React 跳过对 effect 的调用，只要传递数组作为 useEffect 的第二个可选参数即可
- 如果想执行只运行一次的 effect（仅在组件挂载和卸载时执行），可以传递一个空数组（[]）作为第二个参数。这就告诉 React 你的 effect 不依赖于 props 或 state 中的任何值，所以它永远都不需要重复执行

```
function Counter(){
  const [number,setNumber] = useState(0);
  // 相当于componentDidMount 和 componentDidUpdate
  useEffect(() => {
    console.log('开启一个新的定时器')
    const $timer = setInterval(()=>{
      setNumber(number=>number+1);
    },1000);
  },[]);
  return (
    <>
    <p>{number}</p>
    </>
  )
}
```

7.5 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

```
const refContainer = useRef(initialValue);
```

7.5.1 useRef

```
function Parent(){
  let [number,setNumber] = useState(0);
  return (
    <>
      <Child/>
      <button onClick={()=>setNumber({number:number+1})}>+</button>
    </>
  )
}
let input;
function Child(){
  const inputRef = useRef();
  console.log('input===inputRef',input===inputRef);
  input = inputRef;
  function getFocus(){
    inputRef.current.focus();
  }
  return (
    <>
      <input type="text" ref={inputRef}/>
      <button onClick={getFocus}>获得焦点</button>
    </>
  )
}
```

7.5.2 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

```
function Child(props,ref){
  return (
    <input type="text" ref={ref}/>
  )
}
Child = forwardRef(Child);
function Parent(){
  let [number,setNumber] = useState(0);
  const inputRef = useRef();
  function getFocus(){
```

```

    inputRef.current.value = 'focus';
    inputRef.current.focus();
  }
  return (
    <>
      <Child ref={inputRef}/>
      <button onClick={()=>setNumber({number:number+1})}>+</button>
      <button onClick={getFocus}>获得焦点</button>
    </>
  )
}

```

7.5.3 useImperativeHandle

- useImperativeHandle 可以让你在使用 ref 时自定义暴露给父组件的实例值
- 在大多数情况下，应当避免使用 ref 这样的命令式代码。useImperativeHandle 应当与 forwardRef 一起使用

```

function Child(props,ref){
  const inputRef = useRef();
  useImperativeHandle(ref,()=>(
    {
      focus(){
        inputRef.current.focus();
      }
    }
  ));
  return (
    <input type="text" ref={inputRef}/>
  )
}
Child = forwardRef(Child);
function Parent(){
  let [number,setNumber] = useState(0);
  const inputRef = useRef();
  function getFocus(){
    console.log(inputRef.current);
    inputRef.current.value = 'focus';
    inputRef.current.focus();
  }
  return (
    <>
      <Child ref={inputRef}/>
      <button onClick={()=>setNumber({number:number+1})}>+</button>
      <button onClick={getFocus}>获得焦点</button>
    </>
  )
}

```

8. useEffect

- 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
- 可以使用它来读取 DOM 布局并同步触发重渲染
- 在浏览器执行绘制之前 `useLayoutEffect` 内部的更新计划将被同步刷新
- 尽可能使用标准的 `useEffect` 以避免阻塞视图更新

```
function LayoutEffect() {  
  const [color, setColor] = useState('red');  
  useLayoutEffect(() => {  
    alert(color);  
    document.getElementById('myDiv').style.background = 'purple';  
  });  
  useEffect(() => {  
    console.log('color', color);  
  });  
  return (  
    <>  
      <div id="myDiv" style={{background:color}}>颜色</div>  
      <button onClick={()=>setColor('red')}>红</button>  
      <button onClick={()=>setColor('yellow')}>黄</button>  
      <button onClick={()=>setColor('blue')}>蓝</button>  
    </>  
  );  
}
```

9. 自定义 Hook

- 有时候我们会想要在组件之间重用一些状态逻辑
- 自定义 Hook 可以让你在不增加组件的情况下达到同样的目的
- Hook 是一种复用状态逻辑的方式，它不复用 state 本身
- 事实上 Hook 的每次调用都有一个完全独立的 state

```
function useNumber(){  
  const [number, setNumber] = useState(0);  
  useEffect(() => {  
    console.log('开启一个新的定时器')  
    const $timer = setInterval(() => {  
      setNumber(number+1);  
    }, 1000);  
    return () => {  
      console.log('销毁老的定时器')  
      clearInterval($timer);  
    }  
  });  
  return number;  
}
```

```
}  
function Counter1(){  
  let number1 = useNumber();  
  return (  
    <>  
      <p>{number1}</p>  
    </>  
  )  
}  
function Counter2(){  
  let number = useNumber();  
  return (  
    <>  
      <p>{number}</p>  
    </>  
  )  
}  
function App(){  
  return <><Counter1/><Counter2/></>  
}
```

10.附录

10.1 浏览器是如何呈现一张页面的

- 解析HTML，并生成一棵DOM tree
- 解析各种样式并结合DOM tree生成一棵Render tree
- 对Render tree的各个节点计算布局信息，比如box的位置与尺寸
- 根据Render tree并利用浏览器的UI层进行绘制

