### 1. React类组件的生命周期函数

组件的生命周期可分成三个状态：  
  
Mounting：已插入真实 DOM  
Updating：正在被重新渲染  
Unmounting：已移出真实 DOM  
生命周期的方法有：  
  
componentWillMount 在渲染前调用,在客户端也在服务端。  
  
componentDidMount : 在第一次渲染后调用，只在客户端。之后组件已经生成了对应的DOM结构，可以通过this.getDOMNode()来进行访问。 如果你想和其他JavaScript框架一起使用，可以在这个方法中调用setTimeout, setInterval或者发送AJAX请求等操作(防止异步操作阻塞UI)。  
  
componentWillReceiveProps 在组件接收到一个新的 prop (更新后)时被调用。这个方法在初始化render时不会被调用。  
  
shouldComponentUpdate 返回一个布尔值。在组件接收到新的props或者state时被调用。在初始化时或者使用forceUpdate时不被调用。  
可以在你确认不需要更新组件时使用。  
  
componentWillUpdate在组件接收到新的props或者state但还没有render时被调用。在初始化时不会被调用。  
  
componentDidUpdate 在组件完成更新后立即调用。在初始化时不会被调用。  
  
componentWillUnmount在组件从 DOM 中移除之前立刻被调用。

### 2. React中的Context(上下文) 是什么 怎么使用

1. React.Context上下文就是解决React父子组件之间公共数据传输的一个APi 减少组件之间公共数据的重复传参 和 多级嵌套传参的问题 但是又不想使用Redux等太复杂的库时使用Context上下文用来管理全局状态

import React from 'react'  
  
// 创建 Context 填入默认值（任何一个 js 变量）  
const ThemeContext = React.createContext('light')  
  
// 底层组件 - 函数是组件  
function ThemeLink (props) {  
 // const theme = this.context // 会报错。函数式组件没有实例，即没有 this  
  
 // 函数式组件可以使用 Consumer  
 return <ThemeContext.Consumer>  
 { value => <p>link's theme is {value}</p> }  
 </ThemeContext.Consumer>  
}  
  
// 底层组件 - class 组件  
class ThemedButton extends React.Component {  
 // 指定 contextType 读取当前的 theme context。  
 // static contextType = ThemeContext // 也可以用 ThemedButton.contextType = ThemeContext  
 render() {  
 const theme = this.context // React 会往上找到最近的 theme Provider，然后使用它的值。  
 return <div>  
 <p>button's theme is {theme}</p>  
 </div>  
 }  
}  
ThemedButton.contextType = ThemeContext // 指定 contextType 读取当前的 theme context。  
  
// 中间的组件再也不必指明往下传递 theme 了。  
function Toolbar(props) {  
 return (  
 <div>  
 <ThemedButton />  
 <ThemeLink />  
 </div>  
 )  
}  
  
class App extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 theme: 'light'  
 }  
 }  
 render() {  
 return <ThemeContext.Provider value={this.state.theme}>  
 <Toolbar />  
 <hr/>  
 <button onClick={this.changeTheme}>change theme</button>  
 </ThemeContext.Provider>  
 }  
 changeTheme = () => {  
 this.setState({  
 theme: this.state.theme === 'light' ? 'dark' : 'light'  
 })  
 }  
}  
  
export default App

### 3. React Hooks常用APi

1. useState

useState 的使用非常简单，我们从 React 中拿到 useState 后，只需要在使用的地方直接调用 useState 函数就可以， useState 会返回一个数组，第一个值是我们的 state， 第二个值是一个函数，用来修改该 state 的，那么这里为什么叫 count 和 setCount？一定要叫这个吗，这里使用了 es6 的解构赋值，所以你可以给它起任何名字：如updateCount, doCount、any thing，当然，为了编码规范，所以建议统一使用一种命名规范，尤其是第二个值

function App () {  
 const [ count, setCount ] = useState(0)  
 return (  
 <div>  
 点击次数: { count }   
 <button onClick={() => { setCount(count + 1)}}>点我</button>  
 </div>  
 )  
}

我们在使用 useState 的第二个参数时，我们想要获取上一轮该 state 的值的话，只需要在 useState 返回的第二个参数，也就是我们上面的例子中的 setCount 使用时，传入一个参数，该函数的参数就是上一轮的 state 的值

setCount((count => count + 1)

1. useEffect

Effect Hook 可以让你在函数组件中执行副作用操作，这里提到副作用，什么是副作用呢，就是除了状态相关的逻辑，比如网络请求，监听事件，查找 dom

function App () {  
 const [ count, setCount ] = useState(0)  
   
 useEffect(() => {  
 document.title = count  
 })  
   
 return (  
 <div>  
 页面名称: { count }   
 <button onClick={() => { setCount(count + 1 )}}>点我</button>  
 </div>  
 )  
}

useEffect如何模拟生命周期组件加载组件销毁

useEffect(() => {  
 // 相当于 componentDidMount  
 console.log('add resize event')  
 window.addEventListener('resize', onChange, false)  
   
 return () => {  
 // 相当于 componentWillUnmount  
 window.removeEventListener('resize', onChange, false)  
 }  
 }, [])  
   
 useEffect(() => {  
 // 相当于 componentDidUpdate  
 document.title = count  
 })

1. useContext ```jsx

// 创建一个 context const Context = createContext(0)

// 组件一, Consumer 写法 class Item1 extends Component { render () { return ( <Context.Consumer> { (count) => (

{count}

) } </Context.Consumer> ) } } // 组件二, contextType 写法 class Item2 extends Component { static contextType = Context render () { const count = this.context return (

{count}

)

} } // 组件一, useContext 写法 function Item3 () { const count = useContext(Context); return (

{ count }

) }

function App () { const [ count, setCount ] = useState(0) return (

点击次数: { count }   
 <button onClick={() => { setCount(count + 1)}}>点我</button>  
 <Context.Provider value={count}>  
 <Item1></Item1>  
 <Item2></Item2>  
 <Item3></Item3>  
 </Context.Provider>  
</div>  
)

} ```

1. useMemo

useMemo 是什么呢，它跟 memo 有关系吗 ，说白了 memo 就是函数组件的 PureComponent，用来做性能优化的手段，useMemo 也是，useMemo 在我的印象中和 Vue 的 computed 计算属性类似，都是根据依赖的值计算出结果，当依赖的值未发生改变的时候，不触发状态改变

function App () {  
 const [ count, setCount ] = useState(0)  
 const add = useMemo(() => {  
 return count + 1  
 }, [count])  
 return (  
 <div>  
 点击次数: { count }  
 <br/>  
 次数加一: { add }  
 <button onClick={() => { setCount(count + 1)}}>点我</button>  
 </div>  
 )  
}

上面的例子中，useMemo 也支持传入第二个参数，用法和 useEffect 类似

不传数组，每次更新都会重新计算 空数组，只会计算一次 依赖对应的值，当对应的值发生变化时，才会重新计算(可以依赖另外一个 useMemo 返回的值) 需要注意的是，useMemo 会在渲染的时候执行，而不是渲染之后执行，这一点和 useEffect 有区别，所以 useMemo 不建议有 副作用相关的逻辑

同时，useMemo 可以作为性能优化的手段，但不要把它当成语义上的保证，将来，React 可能会选择“遗忘”以前的一些 memoized 值，并在下次渲染时重新计算它们

1. useCallback

useCallback 是什么呢，可以说是 useMemo 的语法糖，能用 useCallback 实现的，都可以使用 useMemo, 在 react 中我们经常面临一个子组件渲染优化的问题， 尤其是在向子组件传递函数props时，每次 render 都会创建新函数，导致子组件不必要的渲染，浪费性能，这个时候，就是 useCallback 的用武之地了，useCallback 可以保证，无论 render 多少次，我们的函数都是同一个函数，减小不断创建的开销

const onClick = useMemo(() => {  
 return () => {  
 console.log('button click')  
 }  
}, [])  
   
const onClick = useCallback(() => {  
 console.log('button click')  
}, [])

useCallback 的第二个参数和useMemo一样，没有区别

1. useRef

useRef 有什么作用呢，其实很简单，总共有两种用法

获取子组件的实例(只有类组件可用) 在函数组件中的一个全局变量，不会因为重复 render 重复申明， 类似于类组件的 this.xxx

// 使用 ref 子组件必须是类组件  
class Children extends PureComponent {  
 render () {  
 const { count } = this.props  
 return (  
 <div>{ count }</div>  
 )  
 }  
}  
   
function App () {  
 const [ count, setCount ] = useState(0)  
 const childrenRef = useRef(null)  
 // const   
 const onClick = useMemo(() => {  
 return () => {  
 console.log('button click')  
 console.log(childrenRef.current)  
 setCount((count) => count + 1)  
 }  
 }, [])  
 return (  
 <div>  
 点击次数: { count }  
 <Children ref={childrenRef} count={count}></Children>  
 <button onClick={onClick}>点我</button>  
 </div>  
 )  
}

使用useRef来保存state的值

function App () {  
 const [ count, setCount ] = useState(0)  
 const timer = useRef(null)  
 let timer2   
   
 useEffect(() => {  
 let id = setInterval(() => {  
 setCount(count => count + 1)  
 }, 500)  
   
 timer.current = id  
 timer2 = id  
 return () => {  
 clearInterval(timer.current)  
 }  
 }, [])  
   
 const onClickRef = useCallback(() => {  
 clearInterval(timer.current)  
 }, [])  
   
 const onClick = useCallback(() => {  
 clearInterval(timer2)  
 }, [])  
   
 return (  
 <div>  
 点击次数: { count }  
 <button onClick={onClick}>普通</button>  
 <button onClick={onClickRef}>useRef</button>  
 </div>  
 )  
}

当我们们使用普通的按钮去暂停定时器时发现定时器无法清除，因为 App 组件每次 render，都会重新申明一次 timer2, 定时器的 id 在第二次 render 时，就丢失了，所以无法清除定时器，针对这种情况，就需要使用到 useRef，来为我们保留定时器 id，类似于 this.xxx，这就是 useRef 的另外一种用法

1. useReducer

useReducer 是什么呢，它其实就是类似 redux 中的功能，相较于 useState，它更适合一些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 等等的特定场景， useReducer 总共有三个参数

1. 第一个参数是 一个 reducer，就是一个函数类似 (state, action) => newState 的函数，传入 上一个 state 和本次的 action
2. 第二个参数是初始 state，也就是默认值，是比较简单的方法
3. 第三个参数是惰性初始化，这么做可以将用于计算 state 的逻辑提取到 reducer 外部，这也为将来对重置 state 的 action 做处理提供了便利

function reducer(state, action) {  
 switch (action.type) {  
 case 'increment':  
 return {count: state.count + 1};  
 case 'decrement':  
 return {count: state.count - 1};  
 default:  
 throw new Error();  
 }  
}  
   
function App() {  
 const [state, dispatch] = useReducer(reducer, {  
 count: 0  
 });  
 return (  
 <>  
 点击次数: {state.count}  
 <button onClick={() => dispatch({type: 'increment'})}>+</button>  
 <button onClick={() => dispatch({type: 'decrement'})}>-</button>  
 </>  
 );  
}