**类组件和函数组件之间的区别是啥？**

函数组件的性能比类组件的性能要高，类组件使用的时候要实例化，而函数组件直接执行函数取返回结果即可。

函数组件没有this 没有生命周期 没有state状态

**state 和 props 区别是啥？**

state 是组件自己管理数据，控制自己的状态，可变；  
props 是外部传入的数据参数，不可变；

**在构造函数调用 super 并将 props 作为参数传入的作用是啥？super(props);**

在调用 super() 方法之前，子类构造函数无法使用this引用，ES6 子类也是如此。将 props 参数传递给 super() 调用的主要原因是在子构造函数中能够通过this.props来获取传入的 props。

**什么是 React Context?**

Context 通过组件树提供了一个传递数据的方法，从而避免了在每一个层级手动的传递 props 属性

export const Context = React.createContext();

export const Provider = Context.Provider;

export const Consumer = Context.Consumer;

**如何避免组件的重新渲染？**

React.memo():这可以防止不必要地重新渲染函数组件

React.memo()可接受2个参数，第一个参数为纯函数的组件，第二个参数用于对比props

function Child({seconds}){

  console.log('I am rendering');

  return (

    <div>I am update every {seconds} seconds</div>

  )

};

function areEqual(prevProps, nextProps) {

  if(prevProps.seconds===nextProps.seconds){

    return true

  }else {

    return false

  }

}

export default React.memo(Child,areEqual)

PureComponent:这可以防止不必要地重新渲染类组件

export default class a extends PureComponent {}

**什么是纯函数？**

纯函数是不依赖并且不会在其作用域之外修改变量状态的函数。本质上，纯函数始终在给定相同参数的情况下返回相同结果。

**当调用setState时，React render 是如何工作的？**

（1）会将传入的参数对象与组件当前的状态合并，重新构建 React 元素树并且重新渲染整个 UI 界面；  
（3）在得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染；  
**区分Real DOM和Virtual DOM**

1. 更新缓慢。 1. 更新更快。

2. 可以直接更新 HTML。 2. 无法直接更新 HTML。  
3. 如果元素更新，则创建新DOM。 3. 如果元素更新，则更新 JSX 。  
4. DOM操作代价很高。 4. DOM 操作非常简单。  
5. 消耗的内存较多。 5. 很少的内存消耗。

**React组件生命周期的阶段是什么？**

初始渲染阶段：这是组件即将开始其生命之旅并进入 DOM 的阶段。  
更新阶段：一旦组件被添加到 DOM，它只有在 prop 或状态发生变化时才可能更新和重新渲染。这些只发生在这个阶段。  
卸载阶段：这是组件生命周期的最后阶段，组件被销毁并从 DOM 中删除。

**什么是useReducer？**

**之前的**

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

**const** [step, setStep] = useState(1);

useEffect(() => {

**const** id = setInterval(() => {

setCount(c => c + step); *// 依赖其他state来更新*

}, 1000);

**return** () => clearInterval(id);

*// 为了保证setCount中的step是最新的，*

*// 我们还需要在deps数组中指定step*

}, [step]);

**现在的** 当状态更新逻辑比较复杂的时候，就考虑使用useReducer

import React, { useReducer, useEffect } from "react";

**const** [state, dispatch] = useReducer(reducer, initialState);

**直接通过**dispatch改变 state

const initialState = {

count: 0,

step: 1,

};

function reducer(state, action) {

const { count, step } = state;

**if** (action.type === 'tick') {

return { count: count + step, step };

} **else** **if** (action.type === 'step') {

return { count, step: action.step };

} **else** {

**throw** new Error();

}

}

**useRef用法**

**传统用法 赋值给button 就可以通过赋值拿到button对应的DOM对象**

const timerID = useRef();

    useEffect(() => {

      timerID.current = setInterval(()=>{

          setCount(count => count + 1);

      }, 1000);

    }, []);

    useEffect(()=>{

      if(count > 5){

          clearInterval(timerID.current);

      }

    });

    return (

      <>

        <button ref={timerID} onClick={() => {setCount(count + 1)}}>Count: {count}</button>

      </>

    );

**redux中间件种类和各自的作用**

**redux-thunk 中间件  改造store.dispatch,解决异步操作**

异步操作的第一种解决方案就是，写出一个返回函数的 Action Creator，然后使用redux-thunk中间件改造store.dispatch

**redux-promise 中间件  使得store.dispatch方法可以接受 Promise 对象作为参数**

**redux-logger中间件 日志中间件**