前端一面必会react面试题（附答案）

前言：最近接触到一种新的（对我个人而言）状态管理方式，它没有采用现有的开源库，如redux、mobx等，也没有使用传统的useContext，而是用useState + useEffect写了一个发布订阅者模式进行状态管理，这一点对我来说感觉比较新奇，以前从没接触过这种写法，于是决定研究一下目前比较常用的状态管理方式。
ps：这里谈到的状态管理是指全局状态管理，局部的使用useState即可

状态管理方式

目前比较常用的状态管理方式有hooks、redux、mobx三种，下面我将详细介绍一下这三类的使用方法以及分析各自的优缺点，以供各位进行参考。

Hooks状态管理

用hooks进行状态管理主要有两种方式：

- useContext+useReducer
- useState+useEffect

useContext+useReducer

使用方法

1.创建store和reducer以及全局context

src/store/reducer.ts

javascript 复制代码

```
import React from "react";
// 初始状态
export const state = {
  count: 0,
```

```
  name: "ry",
};

// reducer 用于修改状态
export const reducer = (state, action) => {
  const { type, payload } = action;
  switch (type) {
    case "ModifyCount":
      return {
        ...state,
        count: payload,
      };
    case "ModifyName":
      return {
        ...state,
        name: payload,
      };
    default: {
      return state;
    }
  }
};

export const GlobalContext = React.createContext(null);
```

2.根组件通过 Provider 注入 context

src/App.tsx

javascript 复制代码

```
import React, { useReducer } from "react";
import './index.less'
import { state as initState, reducer, GlobalContext} from './store/reducer'
import Count from './components/Count'
import Name from './components/Name'

export default function () {
  const [state, dispatch] = useReducer(reducer, initState);

  return (
    <div>
      <GlobalContext.Provider value={{state, dispatch}}>
        <Count />
        <Name />
      </GlobalContext.Provider>
    </div>
```

```
  )
}
```

3.在组件中使用

src/components/Count/index.tsx

javascript 复制代码

```javascript
import { GlobalContext } from "@/store/reducer";
import React, { FC, useContext } from "react";

const Count: FC = () => {
  const ctx = useContext(GlobalContext)
  return (
    <div>
      <p>count:{ctx.state.count}</p>
      <button onClick={() => ctx.dispatch({ type: "ModifyCount", payload: ctx.state.count+1 })}>+1</
    </div>
  );
};

export default Count;
```

参考 前端进阶面试题详细解答

src/components/Name/index.tsx

javascript 复制代码

```javascript
import { GlobalContext } from "@/store/reducer";
import React, { FC, useContext } from "react";

const Name: FC = () => {
  const ctx = useContext(GlobalContext)
  console.log("NameRerendered")
  return (
    <div>
      <p>name:{ctx.state.name}</p>
    </div>
  );
};

export default Name;
```

useState+useEffect

使用方法

1.创建state和reducer

src/global-states.ts

javascript 复制代码

```javascript
// 初始state
let globalState: GlobalStates = {
  count: 0,
  name: 'ry'
}

// reducer
export const modifyGlobalStates = (
  operation: GlobalStatesModificationType,  payload: any
) => {
  switch (operation) {
    case GlobalStatesModificationType.MODIFY_COUNT:
      globalState = Object.assign({}, globalState, { count: payload })
      break
    case GlobalStatesModificationType.MODIFY_NAME:
      globalState = Object.assign({}, globalState, { name: payload })
      break
  }
  broadcast()
}
```

src/global-states.type.ts

typescript 复制代码

```typescript
export interface GlobalStates {
  count: number;
  name: string;
}

export enum GlobalStatesModificationType {
  MODIFY_COUNT,
  MODIFY_NAME
}
```

2.写一个发布订阅模式，让组件订阅globalState

src/global-states.ts

typescript 复制代码

```typescript
import { useState, useEffect } from 'react'
import {
  GlobalStates,
  GlobalStatesModificationType
} from './global-states.type'

let listeners = []

let globalState: GlobalStates = {
  count: 0,
  name: 'ry'
}
// 发布，所有订阅者收到消息，执行setState重新渲染
const broadcast = () => {
  listeners.forEach((listener) => {
    listener(globalState)
  })
}

export const modifyGlobalStates = (
  operation: GlobalStatesModificationType,  payload: any
) => {
  switch (operation) {
    case GlobalStatesModificationType.MODIFY_COUNT:
      globalState = Object.assign({}, globalState, { count: payload })
      break
    case GlobalStatesModificationType.MODIFY_NAME:
      globalState = Object.assign({}, globalState, { name: payload })
      break
  }
  // 状态改变即发布
  broadcast()
}

// useEffect + useState实现发布订阅
export const useGlobalStates = () => {
  const [value, newListener] = useState(globalState)

  useEffect(() => {
    // newListener是新的订阅者
    listeners.push(newListener)
    // 组件卸载取消订阅
    return () => {
      listeners = listeners.filter((listener) => listener !== newListener)
```

```
    }
  })

  return value
}
```

3.组件中使用

src/App.tsx

typescript 复制代码

```
import React from 'react'
import './index.less'
import Count from './components/Count'
import Name from './components/Name'

export default function () {
  return (
    <div>
      <Count />
      <Name />
    </div>
  )
}
```

src/components/Count/index.tsx

javascript 复制代码

```
import React, { FC } from 'react'
import { useGlobalStates, modifyGlobalStates } from '@/store/global-states'
import { GlobalStatesModificationType } from '@/store/global-states.type'

const Count: FC = () => {
  // 调用useGlobalStates()即订阅globalStates()
  const { count } = useGlobalStates()
  return (
    <div>
      <p>count:{count}</p>
      <button
        onClick={() =>
          modifyGlobalStates(            GlobalStatesModificationType.MODIFY_COUNT,            count
    </div>
  )
}

export default Count
```

src/components/Name/index.tsx

```javascript
import React, { FC } from 'react'
import { useGlobalStates } from '@/store/global-states'

const Count: FC = () => {
  const { name } = useGlobalStates()
  console.log('NameRerendered')
  return (
    <div>
      <p>name:{name}</p>
    </div>
  )
}

export default Count
```

优缺点分析

由于以上两种都是采用hooks进行状态管理，这里统一进行分析，

优点

- 代码比较简洁，如果你的项目比较简单，只有少部分状态需要提升到全局，大部分组件依旧通过本地状态来进行管理。这时，使用 hookst进行状态管理就挺不错的。杀鸡焉用牛刀。

缺点

- 两种hooks管理方式都有一个很明显的缺点，会产生大量的无效rerender，如上例中的Count和Name组件，当state.count改变后，Name组件也会rerender，尽管他没有使用到state.count。这在大型项目中无疑是效率比较低的。

Redux状态管理

使用方法：

1.引入redux

typescript 复制代码

```
yarn add redux react-redux @types/react-redux redux-thunk
```

2.新建reducer

在src/store/reducers文件夹下新建addReducer.ts（可建立多个reducer）

typescript 复制代码

```
import * as types from '../action.types'
import { AnyAction } from 'redux'

// 定义参数接口
export interface AddState {
  count: number
  name: string
}

// 初始化state
let initialState: AddState = {
  count: 0,
  name: 'ry'
}

// 返回一个reducer
export default (state: AddState = initialState, action: AnyAction): AddState => {
  switch (action.type) {
    case types.ADD:
      return { ...state, count: state.count + action.payload }
    default:
      return state
  }
}
```

在src/stores文件夹下新建action.types.ts
主要用于声明action类型

typescript 复制代码

```
export const ADD = 'ADD'
export const DELETE = 'DELETE'
```

3.合并reducer

在src/store/reducers文件夹下新建index.ts

typescript 复制代码

```typescript
import { combineReducers, ReducersMapObject, AnyAction, Reducer } from 'redux'
import addReducer, { AddState } from './addReducer'

// 如有多个reducer则合并reducers，模块化
export interface CombinedState {
  addReducer: AddState
}
const reducers: ReducersMapObject<CombinedState, AnyAction> = {
  addReducer
}
const reducer: Reducer<CombinedState, AnyAction> = combineReducers(reducers)

export default reducer
```

3.创建store

在src/stores文件夹下新建index.ts

typescript 复制代码

```typescript
import {
  createStore,
  applyMiddleware,
  StoreEnhancer,
  StoreEnhancerStoreCreator,
  Store
} from 'redux'
import thunk from 'redux-thunk'
import reducer from './reducers'

// 生成store增强器
const storeEnhancer: StoreEnhancer = applyMiddleware(thunk)
const storeEnhancerStoreCreator: StoreEnhancerStoreCreator = storeEnhancer(createStore)

const store: Store = storeEnhancerStoreCreator(reducer)

export default store
```

4.根组件通过 Provider 注入 store

src/index.tsx（用provider将App.tsx包起来）

typescript 复制代码

```typescript
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'
import { Provider } from 'react-redux'
import store from './store'

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

5.在组件中使用

src/somponents/Count/index.tsx

typescript 复制代码

```typescript
import React, { FC } from 'react'
import { connect } from 'react-redux'
import { Dispatch } from 'redux'
import { AddState } from 'src/store/reducers/addReducer'
import { CombinedState } from 'src/store/reducers'
import * as types from '@/store/action.types'

// 声明参数接口
interface Props {
  count: number
  add: (num: number) => void
}

// ReturnType获取函数返回值类型,&交叉类型(用于多类型合并)
// type Props = ReturnType<typeof mapStateToProps> & ReturnType<typeof mapDispatchToProps>

const Count: FC<Props> = (props) => {
  const { count, add } = props
  return (
    <div>
      <p>count: {count}</p>
      <button onClick={() => add(5)}>addCount</button>
    </div>
  )
}

// 这里相当于自己手动做了映射，只有这里映射到的属性变化，组件才会rerender
const mapStateToProps = (state: CombinedState) => ({
  count: state.addReducer.count
```

```
})

const mapDispatchToProps = (dispatch: Dispatch) => {
  return {
    add(num: number = 1) {
      // payload为参数
      dispatch({ type: types.ADD, payload: num })
    }
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(Count)
```

src/somponents/Name/index.tsx

typescript 复制代码

```
import React, { FC } from 'react'
import { connect } from 'react-redux'
import { Dispatch } from 'redux'
import { AddState } from 'src/store/reducers/addReducer'
import { CombinedState } from 'src/store/reducers'
import * as types from '@/store/action.types'

// 声明参数接口
interface Props {
  name: string
}

const Name: FC<Props> = (props) => {
  const { name } = props
  console.log('NameRerendered')
  return (
    <div>
      <p>name: {name}</p>
    </div>
  )
}

// name变化组件才会rerender
const mapStateToProps = (state: CombinedState) => ({
  name: state.addReducer.name
})

// addReducer内任意属性变化组件都会rerender
// const mapStateToProps = (state: CombinedState) => state.addReducer

export default connect(mapStateToProps)(Name)
```

优缺点分析

优点

- 组件会订阅store中具体的某个属性【mapStateToProps手动完成】，只要当属性变化时，组件才会rerender，渲染效率较高
- 流程规范，按照官方推荐的规范和结合团队风格打造一套属于自己的流程。
- 配套工具比较齐全redux-thunk支持异步，redux-devtools支持调试
- 可以自定义各种中间件

缺点

- state+action+reducer的方式不太好理解，不太直观
- 非常啰嗦，为了一个功能又要写reducer又要写action，还要写一个文件定义actionType，显得很麻烦
- 使用体感非常差，每个用到全局状态的组件都得写一个mapStateToProps和mapDispatchToProps，然后用connect包一层，我就简单用个状态而已，咋就这么复杂呢
- 当然还有一堆的引入文件，100行的代码用了redux可以变成120行，不过换个角度来说这也算增加了自己的代码量
- 好像除了复杂也没什么缺点了

Mobx状态管理

常规使用（mobx-react)

使用方法

1.引入mobx

```
yarn add mobx mobx-react -D
```

typescript 复制代码

2.创建store

在/src/store目录下创建你要用到的store（在这里使用多个store进行演示）
例如：
store1.ts

typescript 复制代码

```typescript
import { observable, action, makeObservable } from 'mobx'

class Store1 {
  constructor() {
    makeObservable(this) //mobx6.0之后必须要加上这一句
  }
  @observable
  count = 0

  @observable
  name = 'ry'

  @action
  addCount = () => {
    this.count += 1
  }
}

const store1 = new Store1()
export default store1
```

store2.ts
这里使用 makeAutoObservable代替了makeObservable，这样就不用对每个state和action进行修饰了（两个方法都可，自行选择）

typescript 复制代码

```typescript
import { makeAutoObservable } from 'mobx'

class Store2 {
  constructor() {
    // mobx6.0之后必须要加上这一句
    makeAutoObservable(this)
  }
  time = 11111111110
}

const store2 = new Store2()
export default store2
```

3.导出store

src/store/index.ts

typescript 复制代码

```typescript
import store1 from './store1'
import store2 from './store2'

export const store = { store1, store2 }
```

4.根组件通过 Provider 注入 store

src/index.tsx（用provider将App.tsx包起来）

typescript 复制代码

```typescript
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'
import store from './store'
import { Provider } from 'mobx-react'

ReactDOM.render(
  <Provider {...store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

5.在组件中使用

src/somponents/Count/index.tsx

typescript 复制代码

```typescript
import React, { FC } from 'react'
import { observer, inject } from 'mobx-react'

// 类组件用装饰器注入，方法如下
// @inject('store1')
// @observer
interface Props {
  store1?: any
}
const Count: FC<Props> = (props) => {
  const { count, addCount } = props.store1
  return (
    <div>
      <p>count: {count}</p>
```

```
      <button onClick={addCount}>addCount</button>
    </div>
  )
}
// 函数组件用Hoc，方法如下（本文统一使用函数组件）
export default inject('store1')(observer(Count))
```

src/components/Name/index.tsx

typescript 复制代码

```
import React, { FC } from 'react'
import { observer, inject } from 'mobx-react'

interface Props {
  store1?: any
}

const Name: FC<Props> = (props) => {
  const { name } = props.store1
  console.log('NameRerendered')
  return (
    <div>
      <p>name: {name}</p>
    </div>
  )
}
// 函数组件用Hoc，方法如下（本文统一使用函数组件）
export default inject('store1')(observer(Name))
```

优缺点分析：

优点：

- 组件会自动订阅store中具体的某个属性，无需手动订阅噢！【下文会简单介绍下原理】只有当订阅的属性变化时，组件才会rerender，渲染效率较高
- 一个store即写state，也写action，这种方式便于理解，并且代码量也会少一些

缺点：

- 当我们选择的技术栈是React+Typescript+Mobx时，这种使用方式有一个非常明显的缺点，引入的store必须要在props的type或interface定义过后才能使用（会增加不少代码量），而且还必须指定这个store为可选的，否则会报错（因为父组件其实没有传递这个

prop给子组件），这样做还可能会致使对store取值时，提示可能为undefined，虽然能够用“!”排除undefined，可是这种作法并不优雅。

最佳实践（mobx+hooks）

使用方法

1.引入mobx

同上

2.创建store

同上

3.导出store（结合useContext）

src/store/index.ts

typescript 复制代码

```typescript
import React from 'react'
import store1 from './store1'
import store2 from './store2'

// 导出store1
export const storeContext1 = React.createContext(store1)
export const useStore1 = () => React.useContext(storeContext1)

// 导出store2
export const storeContext2 = React.createContext(store2)
export const useStore2 = () => React.useContext(storeContext2)
```

4.在组件中使用

无需使用Provider注入根组件
src/somponents/Count/index.tsx

typescript 复制代码

```typescript
import React, { FC } from 'react'
import { observer } from 'mobx-react'
import { useStore1 } from '@/store/'

// 类组件可用装饰器，方法如下
// @observer

const Count: FC = () => {
  const { count, addCount } = useStore1()
  return (
    <div>
      <p>count: {count}</p>
      <button onClick={addCount}>addCount</button>
    </div>
  )
}
// 函数组件用Hoc，方法如下（本文统一使用函数组件）
export default observer(Count)
```

src/components/Name/index.tsx

typescript 复制代码

```typescript
import React, { FC } from 'react'
import { observer } from 'mobx-react'
import { useStore1 } from '@/store/'

const Name: FC = () => {
  const { name } = useStore1()
  console.log('NameRerendered')
  return (
    <div>
      <p>name: {name}</p>
    </div>
  )
}

export default observer(Name)
```

优缺点分析：

优点：

- 学习成本少，基础知识非常简单，跟 Vue 一样的核心原理，响应式编程。
- 一个store即写state，也写action，这种方式便于理解

- 组件会自动订阅store中具体的某个属性，只要当属性变化时，组件才会rerender，渲染效率较高
- 成功避免了上一种使用方式的缺点，不用对使用的store进行interface或type声明！
- 内置异步action操作方式
- 代码量真的很少，使用很简单有没有，强烈推荐！

缺点：

- 过于自由：Mobx提供的约定及模版代码很少，这导致开发代码编写很自由，如果不做一些约定，比较容易导致团队代码风格不统一，团队建议启用严格模式！
- 使用方式过于简单

Mobx自动订阅实现原理

基本概念

typescript 复制代码

```
Observable  //被观察者，状态
Observer    //观察者，组件
Reaction    //响应，是一类的特殊的 Derivation，可以注册响应函数，使之在条件满足时自动执行。
```

建立依赖

我们给组件包的一层observer实现了这个功能

typescript 复制代码

```
export default observer(Name)
```

组件每次mount和update时都会执行一遍useObserver函数，useObserver函数中通过reaction.track进行依赖收集，将该组件加到该Observable变量的依赖中（bindDependencies）。

typescript 复制代码

```
// fn = function () { return baseComponent(props, ref);
export function useObserver(fn, baseComponentName) {
    ...
    var rendering;
```

```
    var exception;
    reaction.track(function () {
        try {
            rendering = fn();
        }
        catch (e) {
            exception = e;
        }
    });
    if (exception) {
        throw exception; // re-throw any exceptions caught during rendering
    }
    return rendering;
}
```

reaction.track()

typescript 复制代码

```
 _proto.track = function track(fn) {
    // 开始收集
    startBatch();
    var result = trackDerivedFunction(this, fn, undefined);
    // 结束收集
    endBatch();
  };
```

reaction.track里面的核心内容是trackDerivedFunction

typescript 复制代码

```
function trackDerivedFunction<T>(derivation: IDerivation, f: () => T, context: any) {
       ...
    let result

    // 执行回调f，触发了变量（即组件的参数）的 get，从而获取 dep【收集依赖】
    if (globalState.disableErrorBoundaries === true) {
        result = f.call(context)
    } else {
        try {
            result = f.call(context)
        } catch (e) {
            result = new CaughtException(e)
        }
    }
    globalState.trackingDerivation = prevTracking

    // 给 observable 绑定 derivation
    bindDependencies(derivation)
   ...
```

```
    return result
}
```

触发依赖

Observable（被观察者，状态）修改后，会调用它的set方法，然后再依次执行该Observable之前收集的依赖函数，触发rerender。

组件更新

用组件更新来简单阐述总结一下：mobx的执行原理。

1. observer这个装饰器（也可以是Hoc），对React组件的render方法进行track。
2. 将render方法，加入到各个observable的依赖中。当observable发生变化，track方法就会执行。
3. track中，还是先进行依赖收集，调用forceUpdate去更新组件，然后结束依赖收集。

每次都进行依赖收集的原因是，每次执行依赖可能会发生变化

总结

简单总结了一下目前较为常用的状态管理方式，我个人最喜欢的使用方式是Mobx+Hooks，简单轻量易上手。各位可以根据自己的需求选择适合自己项目的管理方式。