

React Hooks

对函数型组件进行增强

目录

1. React Hooks 介绍
2. React Hooks 使用
3. 自定义 Hook
4. React 路由 Hooks
5. React Hooks 原理分析

1. React Hooks 介绍

1.1 React Hooks 是用来做什么的

对函数型组件进行增强, 让函数型组件可以存储状态, 可以拥有处理副作用的能力.

让开发者在不使用类组件的情况下, 实现相同的功能.

类组件的不足

1. React Hooks 介绍

1.2 类组件的不足 (Hooks 要解决的问题)

1. 缺少逻辑复用机制

为了复用逻辑增加无实际渲染效果的组件，增加了组件层级 显示十分臃肿
增加了调试的难度以及运行效率的降低

1. React Hooks 介绍

1.2 类组件的不足 (Hooks 要解决的问题)

2. 类组件经常会变得很复杂难以维护

将一组相干的业务逻辑拆分到了多个生命周期函数中

在一个生命周期函数内存在多个不相干的业务逻辑

1. React Hooks 介绍

1.2 类组件的不足 (Hooks 要解决的问题)

3. 类成员方法不能保证this指向的正确性

useState 钩子函数

2. React Hooks 使用

Hooks 意为钩子, React Hooks 就是一堆钩子函数, React 通过这些钩子函数对函数型组件进行增强, 不同的钩子函数提供了不同的功能.

`useState()`

`useEffects()`

`useReducer()`

`useRef()`

`useCallback()`


`useContext()`

`useMemo()`

2. React Hooks 使用

2.1 useState()

用于为函数组件引入状态



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

function App () {
  const [count, setCount] = useState(0);
  return <div>
    <span>{count}</span>
    <button onClick={() => setCount(count + 1)}>+ 1</button>
  </div>;
}
```

useState 细节

2. React Hooks 使用

2.1 useState()

1. 接收唯一的参数即状态初始值. 初始值可以是任意数据类型.
2. 返回值为数组. 数组中存储状态值和更改状态值的方法. 方法名称约定以set开头, 后面加上状态名称.
3. 方法可以被调用多次. 用以保存不同状态值.
4. 参数可以是一个函数, 函数返回什么, 初始状态就是什么, 函数只会被调用一次, 用在初始值是动态值的情况.

设置状态方法的使用细节

2. React Hooks 使用

2.1 useState()

设置状态值方法的参数可以是一个值也可以是一个函数

设置状态值方法的方法本身是异步的

useReducer 钩子函数

2. React Hooks 使用

2.2 useReducer()

useReducer是另一种让函数组件保存状态的方式.

2. React Hooks 使用

2.2 useReducer()



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

function reducer (state, action) {
  switch (action.type) {
    case 'increment':
      return state + 1
  }
}

function App () {
  const [count, dispatch] = useReducer(reducer, 0)
  return <div>
    <span>{count}</span>
    <button onClick={() => dispatch({type: 'increment'})}>+1</button>
  </div>;
}
```

useContext 钩子函数

2. React Hooks 使用

2.3 useContext()

在跨组件层级获取数据时简化获取数据的代码。



```
import { createContext, useContext } from 'react';
const countContext = createContext();
function App () {
  return <countContext.Provider value={100}>
    <Foo />
  </countContext.Provider>
}
function Foo () {
  const count = useContext(countContext);
  return <div>{count}</div>
}
```

useEffect 钩子函数执行分析

2. React Hooks 使用

2.4 useEffect()

让函数型组件拥有处理副作用的能力. 类似生命周期函数.

2. React Hooks 使用

2.4 useEffect()

1. useEffect 执行时机

可以把 `useEffect` 看做 `componentDidMount`, `componentDidUpdate` 和 `componentWillUnmount` 这三个函数的组合.

<code>useEffect(() => {})</code>	<code>=></code>	<code>componentDidMount</code> , <code>componentDidUpdate</code>
<code>useEffect(() => {}, [])</code>	<code>=></code>	<code>componentDidMount</code>
<code>useEffect(() => () => {})</code>	<code>=></code>	<code>componentWillUnmount</code>

useEffect 使用方式

2. React Hooks 使用

2.4 useEffect()

2. useEffect 使用方法

1. 为window对象添加滚动事件
2. 设置定时器让count数值每隔一秒增加1

2. React Hooks 使用

2.4 useEffect()

3. useEffect 解决的问题

1. 按照用途将代码进行分类 (将一组相干的业务逻辑归置到了同一个副作用函数中)
2. 简化重复代码, 使组件内部代码更加清晰

useEffect 数据监测

2. React Hooks 使用

2.4 useEffect()

4. 只有指定数据发生变化时触发effect



```
useEffect(() => {  
    document.title = count;  
}, [count]);
```

useEffect 结合异步函数

2. React Hooks 使用

2.4 useEffect()

5. useEffect 结合异步函数

useEffect中的参数函数不能是异步函数, 因为useEffect函数要返回清理资源的函数, 如果是异步函数就变成了返回Promise



```
useEffect(() => {  
  (async () => {  
    await axios.get()  
  })()  
})
```

useMemo 钩子函数

2. React Hooks 使用

2.5 useMemo()

useMemo 的行为类似Vue中的计算属性, 可以监测某个值的变化, 根据变化值计算新值.

useMemo 会缓存计算结果. 如果监测值没有发生变化, 即使组件重新渲染, 也不会重新计算. 此行为可以有助于避免在每个渲染上进行昂贵的计算.



```
import { useMemo } from 'react';

const result = useMemo(() => {
  // 如果count值发生变化此函数重新执行
  return result;
}, [count])
```

memo 方法

2. React Hooks 使用

2.6 memo 方法

性能优化, 如果本组件中的数据没有发生变化, 阻止组件更新. 类似类组件中的 PureComponent 和 shouldComponentUpdate



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

function Counter () {
  return <div></div>;
}

export default memo(Counter);
```

useCallback 钩子函数

2. React Hooks 使用

2.7 useCallback()

性能优化, 缓存函数, 使组件重新渲染时得到相同的函数实例.



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

function Counter () {
  const [count, setCount] = useState(0);
  const resetCount = useCallback(() => setCount(0), [setCount]);
  return <div>
    <span>{count}</span>
    <button onClick={() => setCount(count + 1)}>+1</button>
    <Test resetCount={resetCount}/>
  </div>
}
```

2. React Hooks 使用

2.7 useCallback()

性能优化, 缓存函数, 使组件重新渲染时得到相同的函数实例.



```
import { memo } from 'react';

function Test (props) {
  console.log('Test re-render');
  return <div>
    Test
    <button onClick={props.resetCount}>reset</button>
  </div>
}
export default memo(Test);
```

useRef 钩子函数获取 DOM 元素

2. React Hooks 使用

2.8 useRef()

2.8.1 获取DOM元素对象



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

function App () {
  const username = useRef();
  const handler = () => console.log(username); // {current: input}
  return <input ref={username} onChange={handler}/>
}
```

useRef 钩子函数保存数据

2. React Hooks 使用

2.8 useRef()

2.8.2 保存数据 (跨组件周期)

即使组件重新渲染, 保存的数据仍然还在. 保存的数据被更改不会触发组件重新渲染.

自定义 Hook 函数

3. 自定义 Hook

自定义 Hook 是标准的封装和共享逻辑的方式.

自定义 Hook 是一个函数, 其名称以 use 开头.

自定义 Hook 其实就是逻辑和内置 Hook 的组合.

自定义 Hook 函数实例

React 路由 Hooks

4. React 路由 Hooks

4.1 react-router-dom 路由提供的钩子函数



```
import { useHistory, useLocation, useRouteMatch, useParams } from 'react-router-dom';
```

- ▼ `{history: {...}, location: {...}, match: {...}, staticContext: undefined}`
 - ▶ `history`: `{length: 6, action: "PUSH", location: {...}, createHref: f, push: ...}`
 - ▶ `location`: `{pathname: "/home", search: "", hash: "", state: undefined, key...`
 - ▶ `match`: `{path: "/home", url: "/home", isExact: true, params: {...}}`
`staticContext`: `undefined`
 - ▶ `__proto__`: `Object`

实现 useState 钩子函数

实现 useEffect 钩子函数

完成首页获取商品列表的redux流程

拉勾教育

— 互联网人实战大学 —



下载「拉勾教育App」
获取更多内容