Hooks学习总结

Created and last modified by 王文磊 on Oct 09, 2021

1. Hooks背景

1.1 React组件本质

Model（state，props） --> View（DOM）

> 将UI的展现看成是一个函数的执行过程。Model是参数，执行结果是DOM树，也就是View。由React实现Model发生变化函数重新执行，生成新的DOM树，React再以最优的方式更新到浏览器。

1.2 Class组件不是React设计模型下的最优

一、 Class的继承特性并没有被使用

React的组件之间是不会相互继承的

二、 实例化的特性

React中的UI都由状态驱动，很少会在外部去调用一个类实例（组件）的方法。

1.3 Hooks的诞生

函数更加符合 state => view 的映射关系，可函数没有实例化的对象也就是没有 state，也没有生命周期方法。所以需要一个方法实现：*创建一个函数之外的空间在多次执行之间保存状态，并且检测其变化，从而能够触发函数组件的重新渲染。*

在React中，这个机制就是 Hooks。

> **把某个目标结果钩到某个可能会变化的数据源或者事件源上，那么当被钩到的数据源或时间发生变化时，产生这个目标结果的代码会重新执行，产生更新后的结果。**

对于函数组件，这个结果就是最终的DOM树；

1.4 Hooks带来的逻辑复用

Hooks 中被钩的对象，不仅可以是某个独立的数据源，也可以是另一个 Hook 执行的结果。

例：有多个组件需要在用户调整浏览器窗口大小的时候重新调整布局，那么我们将监听的逻辑抽离成一个公共模块供多个组件复用。

一、Class 组件实现（HOC），定义一个高阶组件，负责监听窗口大小的变化，并将变化后的值作为props传给下一个组件：

```
import React from 'react';

const withWindowSize = Component => {
  // 产生一个高阶组件 WrappedComponent，只包含监听窗口大小的逻辑
  class WrappedComponent extends React.PureComponent {
    constructor(props) {
      super(props);
      this.state = {
        size: this.getSize()
      };
    }
    componentDidMount() {
      window.addEventListener("resize", this.handleResize);
    }
    componentWillUnmount() {
      window.removeEventListener("resize", this.handleResize);
    }
    getSize() {
      return window.innerWidth > 1000 ? "large" : "small";
    }
    handleResize = ()=> {
      const currentSize = this.getSize();
      this.setState({
        size: this.getSize()
      });
    }
    render() {
      // 将窗口大小传递给真正的业务逻辑组件
      return <Component size={this.state.size} />;
    }
  }
  return WrappedComponent;
};

class MyComponent extends React.Component{
  render() {
    const { size } = this.props;
    if (size === "small") return <SmallComponent />;
    else return <LargeComponent />;
  }
}
```

```
// 使用 withWindowSize 产生高阶组件，用于产生 size 属性传递给真正的业务组件
export default withWindowSize(MyComponent);
```

缺点：

1. 代码难理解，不直观。
2. 会增加很多额外的组件节点。

二、函数组件 + hooks实现：

```
import React from 'react';

const {useState, useEffect} = React;

const getSize = () => {
  return window.innerWidth > 1000 ? "large" : "small";
}
const useWindowSize = () => {
  const [size, setSize] = useState(getSize());
  useEffect(() => {
  const handler = () => {
      setSize(getSize())
   };
    window.addEventListener('resize', handler);
    return () => {
      window.removeEventListener('resize', handler);
   };
 }, []);

  return size;
};

const Demo = () => {
  const size = useWindowSize();
  if (size === "small") return <SmallComponent />;
  else return <LargeComponent />;
};
```

窗口大小被 hooks 封装成了组件外部的一个可绑定的数据源。当窗口大小发生变化时，使用这个 Hook 的组件就都会重新渲染。

1.5 Hooks有助于关注分离

Hooks 能够让针对同一个业务逻辑的代尽可能聚合在一块儿。这是过去在 Class 组件中很难做到的。因为在 Class 组件中，你不得不把同一个业务逻辑的代码分散在不同的生命周期中。

Class 对比 使用 Hooks 的函数组件：

可以看到，在 Class 组件中，代码是从技术角度组织在一起的，例如在 cdm 中去做一些初始化的事情。而在函数组件中，代码是从业务角度组织在一起的，相关代码能够出现在集中的地方，从而更容易理解和维护。

2. 基础 Hooks 使用

> 不要将需求映射到某个生命周期中，直接考虑在 Hooks 中去实现

2.1 useState：让函数组件具有维持状态的能力

```
const [state, changeState] = useState(initialState)
```

实现：在一个函数组件的多次渲染之间，这个 state 是共享的。

etc：

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

function Example() {
  // 创建一个保存 count 的 state，并给初始值 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>
        +
      </button>
    </div>
  );
}
```

用法总结：

1. useState(initialState) 的参数 initialState 是创建 state 的初始值，它可以是任意类型，比如数字、对象、数组等等。
2. useState() 的返回值是一个有着两个元素的数组。第一个数组元素用来读取 state 的值，第二个则是用来设置这个 state 的值。在这里要注意的是，state 的变量（例子中的 count）是只读的，所以我们必须通过第二个数组元素 setCount 来设置它的值。
3. 如果要创建多个 state，那么我们就需要多次调用 useState。

> state中永远不要保存可以通过计算得到的值：
>
> 1. 从 props 传递过来的值。有时候 props 传递过来的值无法直接使用，而是要通过一定的计算后再在 UI 上展示，比如说排序。那么我们要做的就是每次用的时候，都重新排序一下，或者利用某些 cache 机制，而不是将结果直接放到 state 里。
> 2. 从 URL 中读到的值。比如有时需要读取 URL 中的参数，把它作为组件的一部分状态。那么我们可以在每次需要用的时候从 URL 中读取，而不是读出来直接放到 state 里。
> 3. 从 cookie、localStorage 中读取的值。通常来说，也是每次要用的时候直接去读取，而不是读出来后放到 state 里。

2.2 useEffect：执行副作用

> 副作用：指执行一段和当前执行结果无关的代码，即执行结果不影响渲 染出来的 UI。

useEffect(callback, dependencies)

第一个参数为要执行的函数 callback，第二个是可选的依赖项数组dependencies。

> useEffect 是每次组件 render 完后判断依赖并执行

etc：

```
function BlogView({ id }) {
  // 设置一个本地 state 用于保存 blog 内容
  const [blogContent, setBlogContent] = useState(null);

  useEffect(() => {
    // useEffect 的 callback 要避免直接的 async 函数，需要封装一下
    const doAsync = async () => {
      // 当 id 发生变化时，将当前内容清除以保持一致性
      setBlogContent(null);
      // 发起请求获取数据
      const res = await fetch(`/blog-content/${id}`);
      // 将获取的数据放入 state
```

```
      setBlogContent(await res.text());
    };
    doAsync();
  }, [id]); // 使用 id 作为依赖项，变化时则执行副作用

  // 如果没有 blogContent 则认为是在 loading 状态
  const isLoading = !blogContent;
  return <div>{isLoading ? "Loading..." : blogContent}</div>;
}
```

1. 没有依赖项：

```
// dependencies为undefined
useEffect(() => {
  // 每次 render 完一定执行
  console.log('re-rendered');
})
```

2. 依赖项为空数组：

```
// dependencies为空数组
useEffect(() => {
  // 组件首次渲染时执行，等价于 class 组件中的 componentDidMount
  console.log('re-rendered');
}, [])
```

除了这些机制之外，useEffect 还***允许你返回一个函数，用于在组件销毁的时候做一些清理的操作。***比如移除事件的监听。这个机制就几乎等价于类组件中的 componentWillUnmount。举个例子，在组件中，我们需要监听窗口的大小变化，以便做一些布局上的调整：

```
// 允许callback中提供一个回调函数，用于清理等操作，等同于 class 中 componentWillUnmount
const [size, setSize] = useState({});
useEffect(() => {
  // 窗口大小变化事件处理函数
  const handler = () => {
    setSize(getSize());
  };
  // 监听 resize 事件
  window.addEventListener('resize', handler);

  // 返回一个 callback 在组件销毁时调用
  return () => {
    // 移除 resize 事件
    window.removeEventListener('resize', handler);
  };
}, []);
```

useEffect 四种执行时机总结：

1. 每次 render 后执行：不提供第二个依赖项参数。比如useEffect(() => {})。
2. 仅第一次 render 后执行：提供一个空数组作为依赖项。比如useEffect(() => {}, [])。
3. 第一次以及依赖项发生变化后执行：提供依赖项数组。比如useEffect(() => {}, [deps])。
4. 组件 unmount 后执行：返回一个回调函数。比如useEffect() => { return () => {} }, [])。

2.3 Hooks 的依赖

那么在定义依赖项时，我们需要注意以下三点：

1. 依赖项中定义的变量一定是会在回调函数中用到的，否则声明依赖项其实是没有意义的。
2. 依赖项一般是一个常量数组，而不是一个变量。因为一般在创建 callback 的时候，你其实非常清楚其中要用到哪些依赖项了。
3. React 会使用浅比较来对比依赖项是否发生了变化，所以要特别注意数组或者对象类型。如果你是每次创建一个新对象，即使和之前的值是等价的，也会被认为是依赖项发生了变化。这是一个刚开始使用 Hooks 时很容易导致 Bug 的地方。例如下面的代码：

```
function Sample() {
  // 这里在每次组件执行时创建了一个新数组
  const todos = [{ text: 'Learn hooks.'}];
  useEffect(() => {
    console.log('Todos changed.');
  }, [todos]);
}
```

代码的原意可能是在 todos 变化的时候去产生一些副作用，但是这里的 todos 变量是在函数内创建的，实际上每次都产生了一个新数组。所以在作为依赖项的时候进行引用的比较，实际上被认为是发生了变化的。

2.4 掌握 Hooks 的使用规则

1. 只能在函数组件的顶级作用域使用：
    a. 所有 Hook 必须要被执行到。
    b. 必须按顺序执行。
2. 只能在函数组件或者其他 Hooks 中使用。（如果一定要在 Class 组件中使用，有一个通用的机制：利用高阶组件的模式，将 Hooks 封装成高阶组件，从而让类组件使用。）

```
import React from 'react';
import { useWindowSize } from '../hooks/useWindowSize';
export const withWindowSize = (Comp) => {
  return props => {
    const windowSize = useWindowSize();
    return <Comp windowSize={windowSize} {...props} />;
 };
};

// 那么我们就可以通过如下代码来使用这个高阶组件：

import React from 'react';
import { withWindowSize } from './withWindowSize';

class MyComp {
  render() {
    const { windowSize } = this.props;
    // ...
 }
}

// 通过 withWindowSize 高阶组件给 MyComp 添加 windowSize 属性
export default withWindowSize(MyComp);

// 这样，通过 withWindowSize 这样一个高阶组件模式，你就可以把 useWindowSize 的结果作为属性，传递给需要使用窗口大小的类组件，这样就可以实现在 Class 组件
```

2.5 useCallback：缓存回调函数

函数组件每次 UI 变化都要重新执行整个函数。**每次创建新函数的方式会让接收事件处理函数的组件需要重新渲染**

```
useCallback(fn, deps);
```

fn：定义的回调函数

deps：依赖的变量数组

只有当某个依赖变量发生变化时，才会重新声明 fn 这个回调函数。

2.6 useMemo：缓存计算的结果

```
useMemo(fn, deps)
```

fn：产生所需数据的一个计算函数

deps：依赖的变量数组

如果某个数据是通过其他数据计算得到的，那么只有当用到的数据，也就是依赖的数据发生变化的时候，才需要重新计算

```
//...
// 使用 userMemo 缓存计算的结果
const usersToShow = useMemo(() => {
    if (!users) return null;
    return users.data.filter((user) => {
      return user.first_name.includes(searchKey));
    }
 }, [users, searchKey]);
//...
```

优点：

1. 复杂计算时提升性能
2. 避免子组件的重复渲染

使用 useMemo 实现 useCallback：

```
 const myEventHandler = useMemo(() => {
   // 返回一个函数作为缓存结果
   return () => {
     // 在这里进行事件处理
   }
 }, [dep1, dep2]);
```

useMemo 与 useCallback 本质上做了同一件事：建立了一个绑定某个结果到依赖数据的关系。只有当依赖变了，这个结果才需要被重新得到。

2.7 useRef：在多次渲染之间共享数据

函数组件无法像类组件一样通过实例的属性去保存一些数据

```
const myRefContainer = useRef(initialValue);
myRefContainer.current = ...
```

目的：

1. 存储跨渲染的数据
2. 保存某个 DOM 节点的引用

特点：存储的数据一般和 UI 的渲染无关，这也是 useRef 和 useState 的区别

2.8 useContext：定义全局状态

提供了一个强大的机制，让 React 应用具备定义全局的响应式数据的能力。

```
const value = useContext(MyContext);
```

使用 React.createContext API 创建一个 Context

```
const MyContext = React.createContext(initialValue);
```

结合 useState 让 Context 变为动态的全局状态，即实现数据绑定

```
const themes = {
  light: {
    foreground: "#000000",
```

```
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

function App() {
  // 使用 state 来保存 theme 从而可以动态修改
  const [theme, setTheme] = useState("light");

  // 切换 theme 的回调函数
  const toggleTheme = useCallback(() => {
    setTheme((theme) => (theme === "light" ? "dark" : "light"));
  }, []);

  return (
    // 使用 theme state 作为当前 Context
    <ThemeContext.Provider value={themes[theme]}>
      <button onClick={toggleTheme}>Toggle Theme</button>
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

使用 Context 需要注意的两点：

1. 会让调试变得困难，因为你很难跟踪某个 Context 的变化究竟是如何产生的。
2. 让组件的复用变得困难，因为一个组件如果使用了某个 Context，它就必须确保被用到的地方一定有这个 Context 的 Provider 在其父组件的路径上。

3. 生命周期

3.1 忘掉Class组件的生命周期

Class 组件和函数组件是两种实现 React 应用的方式，虽然它们是等价的，但是开发的思想有很大不同。如果你是从 Class 组件转换到 Hooks 的方式，那么很重要的一点就是，你要学会忘掉 Class 组件中的生命周期概念，千万不要将原来习惯的 Class 组件开发方式映射到函数组。

一个用于显示博客文章的组件接收一个文章的 id 作为参数，然后根据这个 id 从服务器端获取文章的内容并显示出来。那么当 id 变化的时候，你就需要检测到这个变化，并重新发送请求，显示在界面上。

在 Class 组件中，你通常要用如下的代码实现：

```
class BlogView extends React.Component {
  // ...
  componentDidMount() {
    // 组件第一次加载时去获取 Blog 数据
    fetchBlog(this.props.id);
  }
  componentDidUpdate(prevProps) {
    if (prevProps.id !== this.props.id) {
      // 当 Blog 的 id 发生变化时去获取博客文章
      fetchBlog(this.props.id);
    }
  }
  // ...
}
```

可以看到，在 Class 组件中，需要在两个生命周期方法中去实现副作用，一个是首次加载，另外一个则是每次 UI 更新后。而在函数组件中不再有生命周期的概念，而是提供了 useEffect 这样一个 Hook 专门用来执行副作用，因此，只需下面的代码即可实现同样的功能：

```
function BlogView({ id }) {
  useEffect(() => {
    // 当 id 变化时重新获取博客文章
    fetchBlog(id);
  }, [id]); // 定义了依赖项 id
}
```

> 在函数组件中你要思考的方式永远是：当某个状态发生变化时，我要做什么，而不再是在 Class 组件中的某个生命周期方法中我要做什么。

3.2 重新思考组件的生命周期

3.2.1 构造函数

构造函数的本质，其实就是：在所有其它代码执行之前的一次性初始化工作。在函数组件中，因为没有生命周期的机制，那么转换一下思路，其实我们要实现的是：一次性的代码执行。

虽然没有直接的机制可以做到这一点，但是利用 useRef 这个 Hook，我们可以实现一个 useSingleton 这样的一次性执行某段代码的自定义 Hook，代码如下：

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

// 创建一个自定义 Hook 用于执行一次性代码
function useSingleton(callback) {
  // 用一个 called ref 标记 callback 是否执行过
  const called = useRef(false);
  // 如果已经执行过，则直接返回
  if (called.current) return;
  // 第一次调用时直接执行
  callBack();
  // 设置标记为已执行过
  called.current = true;
}
```

从而在一个函数组件中，可以调用这个自定义 Hook 来执行一些一次性的初始化逻辑：

```
import useSingleton from './useSingleton';

const MyComp = () => {
  // 使用自定义 Hook
  useSingleton(() => {
    console.log('这段代码只执行一次');
 });

  return (
    <div>My Component</div>
 );
};
```

在日常开发中，是无需去将功能映射到传统的生命周期的构造函数的概念，而是要从函数的角度出发，去思考功能如何去实现。

3.2.3 三种常用的生命周期方法

在函数组件中，这几个生命周期方法可以统一到 useEffect 这个 Hook，正如 useEffect 的字面含义，它的作用就是触发一个副作用，即在组件每次 render 之后去执行。

```
useEffect(() => {
  // componentDidMount + componentDidUpdate
  console.log('这里基本等价于 componentDidMount + componentDidUpdate');
  return () => {
    // componentWillUnmount
    console.log('这里基本等价于 componentWillUnmount');
 }
}, [deps])
```

这个写法并没有完全等价于传统的这几个生命周期方法。主要有两个原因：

1. useEffect(callback) 这个 Hook 接收的 callback，只有在依赖项变化时才被执行。而传统的 componentDidUpdate 则一定会执行。这样来看，Hook 的机制其实
2. callback 返回的函数（一般用于清理工作）在下一次依赖项发生变化以及组件销毁之前执行，而传统的 componentWillUnmount 只在组件销毁时才会执行。

假设当文章 id 发生变化时，我们不仅需要获取文章，同时还要监听某个事件，这样在有新的评论时获得通知，就能显示新的评论了。这时候的代码结构如下：

```
import React, { useEffect } from 'react';
import comments from './comments';

function BlogView({ id }) {
  const handleCommentsChange = useCallback(() => {
    // 处理评论变化的通知
 }, []);
  useEffect(() => {
    // 获取博客内容
    fetchBlog(id);
    // 监听指定 id 的博客文章的评论变化通知
    const listener = comments.addListener(id, handleCommentsChange);

    return () => {
      // 当 id 发生变化时，移除之前的监听
      comments.removeListener(listener);
   };
 }, [id, handleCommentsChange])
}
```

useEffect 接收的返回值是一个回调函数，这个回调函数不只是会在组件销毁时执行，而且是每次 Effect 重新执行之前都会执行，用于清理上一次 Effect 的执行结果。

理解这一点非常重要。useEffect 中返回的回调函数，只是清理当前执行的 Effect 本身。这其实是更加语义化的，因此你不用将其映射到 componentWillUnmount，它也完全不等价于 componentWillUnmount。你只需记住它的作用就是用于清理上一次 Effect 的结果就行了，这样在实际的开发中才能够使用得更加自然和合理。

4. 四个典型使用场景

在遇到一个功能开发的需求时，首先问自己一个问题：这个功能中的哪些逻辑可以抽出来成为独立的 Hooks?

这么问的目的，是为了让我们尽可能地把业务逻辑拆成独立的 Hooks ，这样有助于实现代码的模块化和解耦，同时也方便后面的维护。

4.1 如何创建自定义 Hooks？（抽取业务逻辑）

自定义 Hooks 的两个特点：

1. 名字一定是以 **use 开头的函数**，这样 React 才能够知道这个函数是一个 Hook；
2. 函数内部**一定调用了其它的 Hooks**，可以是内置的 Hooks，也可以是其它自定义 Hooks。这样才能够让组件刷新，或者去产生副作用。

以之前计数器为例，将业务逻辑提取出来成为一个 Hook：

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

function useCounter() {
  // 定义 count 这个 state 用于保存当前数值
  const [count, setCount] = useState(0);
  // 实现加 1 的操作
  const increment = useCallback(() => setCount(count + 1), [count]);
  // 实现减 1 的操作
  const decrement = useCallback(() => setCount(count - 1), [count]);
  // 重置计数器
  const reset = useCallback(() => setCount(0), []);

  // 将业务逻辑的操作 export 出去供调用者使用
  return { count, increment, decrement, reset };
}
```

有了这个 Hook，我们就可以在组件中使用它，比如下面的代码：

```
import React from 'react';

function Counter() {
  // 调用自定义 Hook
  const { count, increment, decrement, reset } = useCounter();

  // 渲染 UI
  return (
    <div>
      <button onClick={decrement}> - </button>
      <p>{count}</p>
      <button onClick={increment}> + </button>
      <button onClick={reset}> reset </button>
    </div>
 );
}
```

在这段代码中，我们把原来在函数组件中实现的逻辑提取了出来，成为一个单独的 Hook，**一方面能让这个逻辑得到重用，另外一方面也能让代码更加语义化，并且易于理解和维护。**

4.2 封装通用逻辑：useAsync

在日常 UI 的开发中，有一个最常见的需求：发起异步请求获取数据并显示在界面上。在这个过程中，我们不仅要关心请求正确返回时，UI 会如何展现数据；还需要处理请求出错，以及关注 Loading 状态在 UI 上如何显示。

从 Server 端获取用户列表，并显示在界面上：

```
import React from "react";

export default function UserList() {
  // 使用三个 state 分别保存用户列表，loading 状态和错误状态
  const [users, setUsers] = React.useState([]);
  const [loading, setLoading] = React.useState(false);
  const [error, setError] = React.useState(null);

  // 定义获取用户的回调函数
  const fetchUsers = async () => {
    setLoading(true);
    try {
      const res = await fetch("https://reqres.in/api/users/");
      const json = await res.json();
      // 请求成功后将用户数据放入 state
      setUsers(json.data);
   } catch (err) {
      // 请求失败将错误状态放入 state
      setError(err);
   }
    setLoading(false);
 };

  return (
    <div className="user-list">
      <button onClick={fetchUsers} disabled={loading}>
       {loading ? "Loading..." : "Show Users"}
```

```
        </button>
        {error &&
          <div style={{ color: "red" }}>Failed: {String(error)}</div>
        }
        <br />
        <ul>
         {users && users.length > 0 &&
           users.map((user) => {
             return <li key={user.id}>{user.first_name}</li>;
           })}
        </ul>
      </div>
  );
}
```

在处理这类请求的时候，模式都是类似的，通常都会遵循下面步骤：

1. 创建 data, loading, error 这 3 个 state;
2. 请求发出后，设置 loading state 为 true;
3. 请求成功后，将返回的数据放到某个 state 中，并将 loading state 设为 false;
4. 请求失败后，设置 error state 为 true，并将 loading state 设为 false。

通过创建一个自定义 Hook，可以很好地将这样的逻辑提取出来，成为一个可重用的模块。比如代码可以这样实现：

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

const useAsync = (asyncFunction) => {
  // 设置三个异步逻辑相关的 state
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  // 定义一个 callback 用于执行异步逻辑
  const execute = useCallback(() => {
    // 请求开始时，设置 loading 为 true，清除已有数据和 error 状态
    setLoading(true);
    setData(null);
    setError(null);
    return asyncFunction()
     .then((response) => {
        // 请求成功时，将数据写进 state，设置 loading 为 false
        setData(response);
        setLoading(false);
     })
     .catch((error) => {
        // 请求失败时，设置 loading 为 false，并设置错误状态
        setError(error);
        setLoading(false);
     });
  }, [asyncFunction]);

  return { execute, loading, data, error };
};
```

那么有了这个 Hook，我们在组件中就只需要关心与业务逻辑相关的部分。比如代码可以简化成这样的形式：

```
import React from "react";
import useAsync from './useAsync';

export default function UserList() {
  // 通过 useAsync 这个函数，只需要提供异步逻辑的实现
  const {
    execute: fetchUsers,
    data: users,
    loading,
    error,
  } = useAsync(async () => {
    const res = await fetch("https://reqres.in/api/users/");
    const json = await res.json();
    return json.data;
  });

  return (
    // 根据状态渲染 UI...
  );
}
```

利用了 Hooks 能够管理 React 组件状态的能力，将一个组件中的某一部分状态独立出来，从而实现了通用逻辑的重用。

4.3 监听浏览器状态：useScroll

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

// 获取横向，纵向滚动条位置
const getPosition = () => {
  return {
    x: document.body.scrollLeft,
    y: document.body.scrollTop,
  };
};
const useScroll = () => {
```

```
    // 定一个 position 这个 state 保存滚动条位置
    const [position, setPosition] = useState(getPosition());
    useEffect(() => {
      const handler = () => {
        setPosition(getPosition(document));
     };
      // 监听 scroll 事件，更新滚动条位置
      document.addEventListener("scroll", handler);
      return () => {
        // 组件销毁时，取消事件监听
        document.removeEventListener("scroll", handler);
     };
   }, []);
    return position;
  };
```

有了这个 Hook，你就可以非常方便地监听当前浏览器窗口的滚动条位置了。比如下面的代码就展示了“返回顶部”这样一个功能的实现：

```
  import React, { useCallback } from 'react';
  import useScroll from './useScroll';

  function ScrollTop() {
    const { y } = useScroll();

    const goTop = useCallback(() => {
      document.body.scrollTop = 0;
   }, []);

    const style = {
      position: "fixed",
      right: "10px",
      bottom: "10px",
   };
    // 当滚动条位置纵向超过 300 时，显示返回顶部按钮
    if (y > 300) {
      return (
        <button onClick={goTop} style={style}>
          Back to Top
        </button>
    );
   }
    // 否则不 render 任何 UI
    return null;
  }
```

通过这个例子，我们看到了如何将浏览器状态变成可被 React 组件绑定的数据源，从而在使用上更加便捷和直观。当然，除了窗口大小、滚动条位置这些状态，还有其它一些数据也可以这样操作，比如 cookies，localStorage， URL，等等。

4.4 拆分复杂组件

“保持每个函数的短小”这样通用的最佳实践，同样适用于函数组件。只有这样，才能让代码始终易于理解和维护。

> **尽量将相关的逻辑做成独立的 Hooks，然后在函数组中使用这些 Hooks，通过参数传递和返回值让 Hooks 之间完成交互。**

拆分逻辑的目的不一定是为了重用，而可以是仅仅为了业务逻辑的隔离。所以在这个场景下，我们不一定要把 Hooks 放到独立的文件中，而是可以和函数组件写在一个文件中。这么做的原因就在于，这些 Hooks 是和当前函数组件紧密相关的，所以写到一起，反而更容易阅读和理解。

看一个例子。设想现在有这样一个需求：我们需要展示一个博客文章的列表，并且有一列要显示文章的分类。同时，我们还需要提供表格过滤功能，以便能够只显示某个分类的文章。

如果按照直观的思路去实现，通常都会把逻辑都写在一个组件里，比如类似下面的代码：

```
  function BlogList() {
    // 获取文章列表...
    // 获取分类列表...
    // 组合文章数据和分类数据...
    // 根据选择的分类过滤文章...

    // 渲染 UI ...
  }
```

我们要真正把 Hooks 就看成普通的函数，能隔离的尽量去做隔离，从而让代码更加模块化，更易于理解和维护。

针对这样一个功能，我们甚至可以将其拆分成 4 个 Hooks，每一个 Hook 都尽量小，代码如下：

```
  import React, { useEffect, useCallback, useMemo, useState } from "react";
  import { Select, Table } from "antd";
  import _ from "lodash";
  import useAsync from "./useAsync";

  const endpoint = "https://myserver.com/api/";
  const useArticles = () => {
    // 使用上面创建的 useAsync 获取文章列表
    const { execute, data, loading, error } = useAsync(
      useCallback(async () => {
```

```
        const res = await fetch(`${endpoint}/posts`);
        return await res.json();
      }, []),
    );
    // 执行异步调用
    useEffect(() => execute(), [execute]);
    // 返回语义化的数据结构
    return {
      articles: data,
      articlesLoading: loading,
      articlesError: error,
    };
  };
  const useCategories = () => {
    // 使用上面创建的 useAsync 获取分类列表
    const { execute, data, loading, error } = useAsync(
      useCallback(async () => {
        const res = await fetch(`${endpoint}/categories`);
        return await res.json();
      }, []),
    );
    // 执行异步调用
    useEffect(() => execute(), [execute]);

    // 返回语义化的数据结构
    return {
      categories: data,
      categoriesLoading: loading,
      categoriesError: error,
    };
  };
  const useCombinedArticles = (articles, categories) => {
    // 将文章数据和分类数据组合到一起
    return useMemo(() => {
      // 如果没有文章或者分类数据则返回 null
      if (!articles || !categories) return null;
      return articles.map((article) => {
        return {
          ...article,
          category: categories.find(
            (c) => String(c.id) === String(article.categoryId),
          ),
        };
      });
    }, [articles, categories]);
  };
  const useFilteredArticles = (articles, selectedCategory) => {
    // 实现按照分类过滤
    return useMemo(() => {
      if (!articles) return null;
      if (!selectedCategory) return articles;
      return articles.filter((article) => {
        console.log("filter: ", article.categoryId, selectedCategory);
        return String(article?.category?.name) === String(selectedCategory);
      });
    }, [articles, selectedCategory]);
  };

  const columns = [
    { dataIndex: "title", title: "Title" },
    { dataIndex: ["category", "name"], title: "Category" },
  ];

  export default function BlogList() {
    const [selectedCategory, setSelectedCategory] = useState(null);
    // 获取文章列表
    const { articles, articlesError } = useArticles();
    // 获取分类列表
    const { categories, categoriesError } = useCategories();
    // 组合数据
    const combined = useCombinedArticles(articles, categories);
    // 实现过滤
    const result = useFilteredArticles(combined, selectedCategory);

    // 分类下拉框选项用于过滤
    const options = useMemo(() => {
      const arr = _.uniqBy(categories, (c) => c.name).map((c) => ({
        value: c.name,
        label: c.name,
      }));
      arr.unshift({ value: null, label: "All" });
      return arr;
    }, [categories]);

    // 如果出错，简单返回 Failed
    if (articlesError || categoriesError) return "Failed";

    // 如果没有结果，说明正在加载
    if (!result) return "Loading...";

    return (
      <div>
        <Select
          value={selectedCategory}
          onChange={(value) => setSelectedCategory(value)}
          options={options}
          style={{ width: "200px" }}
          placeholder="Select a category"
```

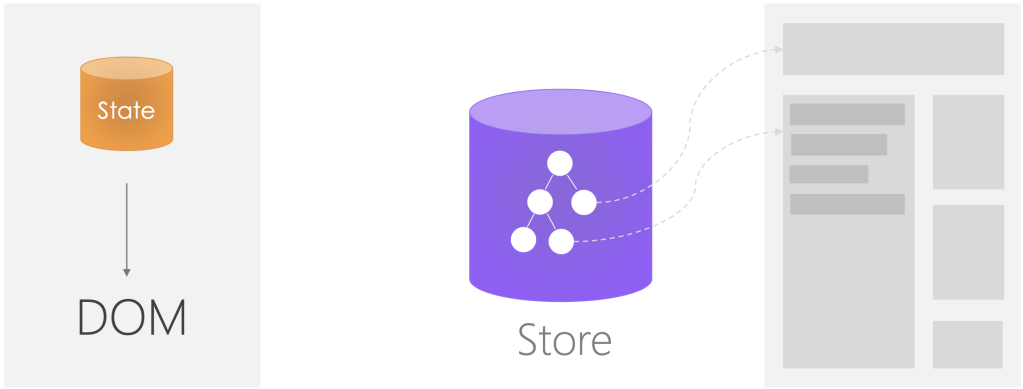
```
        />
        <Table dataSource={result} columns={columns} />
      </div>
    );
  }
```

通过这样的方式，我们就把一个较为复杂的逻辑拆分成一个个独立的 Hook 了，不仅隔离了业务逻辑，也让代码在语义上更加明确。比如说有 useArticles、useCategories 这样与业务相关的名字，就非常易于理解。

5. 函数组件使用 Redux

5.1 Redux 出现的背景

组件级别的 state，和从上而下传递的 props 这两个状态机制，无法满足复杂功能的需要。例如跨层级之间的组件的数据共享和传递。我们可以从下图的对比去理解：



左图是单个 React 组件，它的状态可以用内部的 state 来维护，而且这个 state 在组件外部是无法访问的。而右图则是使用 Redux 的场景，用全局唯一的 Store 维护了整个应用程序的状态。可以说，对于页面的多个组件，都是从这个 Store 来获取状态的，保证组件之间能够共享状态。

从这张对比图，我们可以看到 Redux Store 的两个特点：

1. Redux Store 是全局唯一的。即整个应用程序一般只有一个 Store。
2. Redux Store 是树状结构，可以更天然地映射到组件树的结构，虽然不是必须的。

我们通过把状态放在组件之外，就可以让 React 组件成为更加纯粹的表现层，那么很多对于业务数据和状态数据的管理，就都可以在组件之外去完成。同时这也天然提供了状态共享的能力，有两个场景可以典型地体现出这一点。

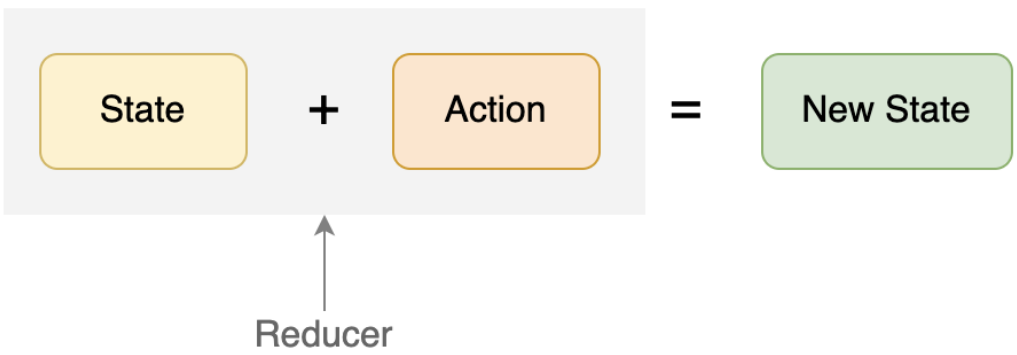
1. 跨组件的状态共享：当某个组件发起一个请求时，将某个 Loading 的数据状态设为 True，另一个全局状态组件则显示 Loading 的状态。
2. 同组件多个实例的状态共享：某个页面组件初次加载时，会发送请求拿回了一个数据，切换到另外一个页面后又返回。这时数据已经存在，无需重新加载。设想如果是本地的组件 state，那么组件销毁后重新创建，state 也会被重置，就还需要重新获取数据。

5.2 理解 Redux 的三个基本概念

State、Action 和 Reducer。

- 其中 State 即 Store，一般就是一个纯 JavaScript Object。
- Action 也是一个 Object，用于描述发生的动作。
- 而 Reducer 则是一个函数，接收 Action 和 State 并作为参数，通过计算得到新的 Store。

它们三者之间的关系可以用下图来表示：



在 Redux 中，所有对于 Store 的修改都必须通过这样一个公式去完成，即通过 Reducer 完成，而不是直接修改 Store。这样的话，一方面可以保证数据的不可变性（Immutable），同时也能带来两个非常大的好处。

1. 可预测性（Predictable）：即给定一个初始状态和一系列的 Action，一定能得到一致的结果，同时这也让代码更容易测试。
2. 易于调试：可以跟踪 Store 中数据的变化，甚至暂停和回放。因为每次 Action 产生的变化都会产生新的对象，而我们可以缓存这些对象用于调试。Redux 的基于浏览器插件的开发工具就是基于这个机制，非常有利于调试。

用Redux实现一个计数器：

```
import { createStore } from 'redux'

// 定义 Store 的初始值
const initialState = { value: 0 }

// Reducer，处理 Action 返回新的 State
function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'counter/incremented':
      return { value: state.value + 1 }
    case 'counter/decremented':
      return { value: state.value - 1 }
    default:
      return state
 }
}

// 利用 Redux API 创建一个 Store，参数就是 Reducer
const store = createStore(counterReducer)

// Store 提供了 subscribe 用于监听数据变化
store.subscribe(() => console.log(store.getState()))

// 计数器加 1，用 Store 的 dispatch 方法分发一个 Action，由 Reducer 处理
const incrementAction = { type: 'counter/incremented' };
store.dispatch(incrementAction);
// 监听函数输出：{value: 1}

// 计数器减 1
const decrementAction = { type: 'counter/decremented' };
store.dispatch(decrementAction)
// 监听函数输出：{value: 0}
```

通过这段代码，我们就用三个步骤完成了一个完整的 Redux 的逻辑：

1. 先创建 Store；
2. 再利用 Action 和 Reducer 修改 Store；
3. 最后利用 subscribe 监听 Store 的变化。

需要注意的是，在 Reducer 中，我们每次都必须返回一个新的对象，确保不可变数据（Immutable）的原则。一般来说，我们可以用延展操作符（Spread Operator）来简单地实现不可变数据的操作，例如：

```
return {
  ...state, // 复制原有的数据结构
  value: state.value + 1, // 变化 value 值使其 + 1
}
```

5.3 如何在 React 中使用 Redux

如何建立 Redux 和 React 的联系：

1. React 组件能够在依赖的 Store 的数据发生变化时，重新 Render；
2. 在 React 组件中，能够在某些时机去 dispatch 一个 action，从而触发 Store 的更新。

要实现这两点，我们需要引入 Facebook 提供的 react-redux 这样一个工具库，工具库的作用就是建立一个桥梁，让 React 和 Redux 实现互通。

在 react-redux 的实现中，为了确保需要绑定的组件能够访问到全局唯一的 Redux Store，利用了 React 的 Context 机制去存放 Store 的信息。通常我们会将这个 Context 作为整个 React 应用程序的根节点。因此，作为 Redux 的配置的一部分，我们通常需要如下的代码：

```
import React from 'react'
import ReactDOM from 'react-dom'

import { Provider } from 'react-redux'
import store from './store'

import App from './App'

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

完成了这样的配置之后，在函数组件中使用 Redux 就非常简单了：利用 react-redux 提供的 useSelector 和 useDispatch 这两个 Hooks。

当 Hooks 用到 Redux 时可变的对象就是 Store，而 useSelector 则让一个组件能够在 Store 的某些数据发生变化时重新 render。

React 中使用 Redux实现计数器：

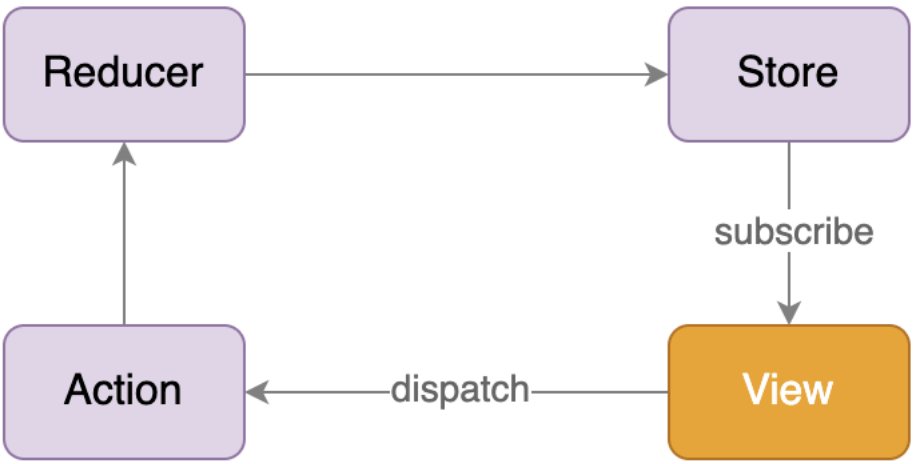
```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'

export function Counter() {
  // 从 state 中获取当前的计数值
  const count = useSelector(state => state.value)

  // 获得当前 store 的 dispatch 方法
  const dispatch = useDispatch()

  // 在按钮的 click 时间中去分发 action 来修改 store
  return (
    <div>
      <button
        onClick={() => dispatch({ type: 'counter/incremented' })}
      >+</button>
      <span>{count}</span>
      <button
        onClick={() => dispatch({ type: 'counter/decremented' })}
      >-</button>
    </div>
  )
}
```

此外，通过计数器这个例子，我们还可以看到 React 和 Redux 共同使用时的单向数据流：



5.4 使用 Redux 处理异步逻辑

在 Redux 中，处理异步逻辑也常常被称为异步 Action。

对于发送请求获取数据这样一个异步的场景，我们来看看涉及到 Store 数据会有哪些变化：

1. 请求发送出去时：设置 state.pending = true，用于 UI 显示加载中的状态；
2. 请求发送成功时：设置 state.pending = false, state.data = result。即取消 UI 的加载状态，同时将获取的数据放到 store 中用于 UI 的显示。
3. 请求发送失败时：设置 state.pending = false, state.error = error。即取消 UI 的加载状态，同时设置错误的状态，用于 UI 显示错误的内容。

前面提到，任何对 Store 的修改都是由 action 完成的。那么对于一个异步请求，上面的三次数据修改显然必须要三个 action 才能完成。那么假设我们在 React 组件中去做这个发起请求的动作，代码逻辑应该类似如下：

```
function DataList() {
  const dispatch = useDispatch();
  // 在组件初次加载时发起请求
  useEffect(() => {
    // 请求发送时
    dispatch({ type: 'FETCH_DATA_BEGIN' });
    fetch('/some-url').then(res => {
      // 请求成功时
      dispatch({ type: 'FETCH_DATA_SUCCESS', data: res });
   }).catch(err => {
      // 请求失败时
      dispatch({ type: 'FETCH_DATA_FAILURE', error: err });
   })
 }, []);

  // 绑定到 state 的变化
  const data = useSelector(state => state.data);
  const pending = useSelector(state => state.pending);
  const error = useSelector(state => state.error);
```

```
  // 根据 state 显示不同的状态
  if (error) return 'Error.';
  if (pending) return 'Loading...';
  return <Table data={data} />;
}
```

显然，发送请求获取数据并进行错误处理这个逻辑是不可重用的。假设我们希望在另外一个组件中也能发送同样的请求，就不得不将这段代码重新实现一遍。因此，Redux 中提供了 middleware 这样一个机制，让我们可以巧妙地实现所谓异步 Action 的概念。

简单来说，middleware 可以让你提供一个拦截器在 reducer 处理 action 之前被调用。在这个拦截器中，你可以自由处理获得的 action。无论是把这个 action 直接传递到 reducer，或者构建新的 action 发送到 reducer，都是可以的。

从下面这张图可以看到，Middleware 正是在 Action 真正到达 Reducer 之前提供的一个额外处理 Action 的机会：

Redux 中的 Action 不仅仅可以是一个 Object，它可以是任何东西，也可以是一个函数。利用这个机制，Redux 提供了 redux-thunk 这样一个中间件，它如果发现接受到的 action 是一个函数，那么就不会传递给 Reducer，而是执行这个函数，并把 dispatch 作为参数传给这个函数，从而在这个函数中你可以自由决定何时，如何发送 Action。

对于上面的场景，假设我们在创建 Redux Store 时指定了 redux-thunk 这个中间件：

```
import { createStore, applyMiddleware } from 'redux'
import thunkMiddleware from 'redux-thunk'
import rootReducer from './reducer'

const composedEnhancer = applyMiddleware(thunkMiddleware)
const store = createStore(rootReducer, composedEnhancer)
```

那么在我们 dispatch action 时就可以 dispatch 一个函数用于来发送请求，通常，我们会写成如下的结构：

```
function fetchData() {
  return dispatch => {
    dispatch({ type: 'FETCH_DATA_BEGIN' });
    fetch('/some-url').then(res => {
      dispatch({ type: 'FETCH_DATA_SUCCESS', data: res });
   }).catch(err => {
      dispatch({ type: 'FETCH_DATA_FAILURE', error: err });
   })
 }
}

import fetchData from './fetchData';

function DataList() {
  const dispatch = useDispatch();
  // dispatch 了一个函数由 redux-thunk 中间件去执行
  dispatch(fetchData());
}
```

可以看到，通过这种方式，我们就实现了异步请求逻辑的重用。那么这一套结合 redux-thunk 中间件的机制，我们就称之为异步 Action。

所以说异步 Action 并不是一个具体的概念，而可以把它看作是 Redux 的一个使用模式。它通过组合使用同步 Action ，在没有引入新概念的同时，用一致的方式提供了处理异步逻辑的方案。

Like Be the first to like this