

React Hooks解析

王红元 coderwhy

目录

content



- 1 认识和体验Hooks
- 2 State/Effect
- 3 Context/Reducer
- 4 Callback/Memo
- 5 Ref/LayoutEffect
- 6 自定义Hooks使用

为什么需要Hook?

- *Hook* 是 React 16.8 的新增特性，它可以让我们在**不编写class的情况下使用state以及其他的React特性**（比如生命周期）。
- 我们先来思考一下class组件相对于函数式组件有什么优势？比较常见的是下面的优势：
- class组件可以**定义自己的state**，用来**保存组件自己内部的状态**；
 - 函数式组件不可以，因为函数每次调用都会产生新的临时变量；
- class组件有**自己的生命周期**，我们可以在**对应的生命周期中完成自己的逻辑**；
 - 比如在componentDidMount中发送网络请求，并且该生命周期函数只会执行一次；
 - 函数式组件在学习hooks之前，如果在函数中发送网络请求，意味着每次重新渲染都会重新发送一次网络请求；
- class组件可以**在状态改变时只会重新执行render函数以及我们希望重新调用的生命周期函数componentDidUpdate等**；
 - 函数式组件在重新渲染时，整个函数都会被执行，似乎没有什么地方可以只让它们调用一次；
- 所以，在Hook出现之前，对于上面这些情况我们通常都会编写class组件。

Class组件存在的问题

■ 复杂组件变得难以理解：

- 我们在最初编写一个class组件时，往往逻辑比较简单，并不会非常复杂。但是随着业务的增多，我们的class组件会变得越来越复杂；
- 比如componentDidMount中，可能就会包含大量的逻辑代码：包括网络请求、一些事件的监听（还需要在componentWillUnmount中移除）；
- 而对于这样的class实际上非常难以拆分：因为它们的逻辑往往混在一起，强行拆分反而会造成过度设计，增加代码的复杂度；

■ 难以理解的class：

- 很多人发现学习ES6的class是学习React的一个障碍。
- 比如在class中，我们必须搞清楚this的指向到底是谁，所以需要花很多的精力去学习this；
- 虽然我认为前端开发人员必须掌握this，但是依然处理起来非常麻烦；

■ 组件复用状态很难：

- 在前面为了一些状态的复用我们需要通过高阶组件；
- 像我们之前学习的redux中connect或者react-router中的withRouter，这些高阶组件设计的目的是为了状态的复用；
- 或者类似于Provider、Consumer来共享一些状态，但是多次使用Consumer时，我们的代码就会存在很多嵌套；
- 这些代码让我们不管是编写和设计上来说，都变得非常困难；

Hook的出现

■ Hook的出现，可以解决上面提到的这些问题；

■ 简单总结一下hooks：

- 它可以让我们在不编写class的情况下使用state以及其他的React特性；

- 但是我们可以由此延伸出非常多的用法，来让我们前面所提到的问题得到解决；

■ Hook的使用场景：

- Hook的出现基本可以代替我们之前所有使用class组件的地方；

- 但是如果是一个旧的项目，你并不需要直接将所有的代码重构为Hooks，因为它完全向下兼容，你可以渐进式的来使用它；

- Hook只能在函数组件中使用，不能在类组件，或者函数组件之外的地方使用；

■ 在我们继续之前，请记住 Hook 是：

- 完全可选的：你无需重写任何已有代码就可以在一些组件中尝试 Hook。但是如果你不想，你不必现在就去学习或使用 Hook。

- 100% 向后兼容的：Hook 不包含任何破坏性改动。

- 现在可用：Hook 已发布于 v16.8.0。

Class组件和Functional组件对比

```
import * as React from "react";
import { Card, Row, Input, Text } from "../components";
import ThemeContext from "../ThemeContext";

export default class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Harry",
      surname: "Potter",
      width: window.innerWidth
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleResize = this.handleResize.bind(this);
    this.handleSurnameChange = this.handleSurnameChange.bind(this);
  }

  componentDidMount() {
    window.addEventListener("resize", this.handleResize);
    document.title = this.state.name + ' ' + this.state.surname;
  }

  componentDidUpdate() {
    document.title = this.state.name + ' ' + this.state.surname;
  }

  componentWillUnmount() {
    window.removeEventListener("resize", this.handleResize);
  }

  handleChange(name) {
    this.setState({ name });
  }

  handleSurnameChange(surname) {
    this.setState({ surname });
  }

  handleResize() {
    this.setState({ width: window.innerWidth });
  }

  render() {
    let { name, surname, width } = this.state;
    return (
      <ThemeContext.Consumer>
        {theme => (
          <Card theme={theme}>
            <Row label="Name">
              <Input value={name} onChange={this.handleChange} />
            </Row>
            <Row label="Surname">
              <Input value={surname} onChange={this.handleSurnameChange} />
            </Row>
            <Row label="Width">
              <Text>{width}</Text>
            </Row>
          </Card>
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

class组件

```
import React, { useState, useContext, useEffect } from "react";
import { Card, Row, Input, Text } from "../components";
import ThemeContext from "../ThemeContext";

export default function Greeting(props) {
  let theme = useContext(ThemeContext);

  let [name, setName] = useState("Harry");
  let [surname, setSurname] = useState("Potter");
  useEffect(() => {
    document.title = name + ' ' + surname;
  });

  let [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    let handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  });

  return (
    <Card theme={theme}>
      <Row label="Name">
        <Input value={name} onChange={setName} />
      </Row>
      <Row label="Surname">
        <Input value={surname} onChange={setSurname} />
      </Row>
      <Row label="Width">
        <Text>{width}</Text>
      </Row>
    </Card>
  );
}
```

函数式组件结合hooks

计数器案例对比

■ 我们通过一个计数器案例，来对比一下class组件和函数式组件结合hooks的对比：

```
import React, { PureComponent } from 'react'

export default class Counter01 extends PureComponent {
  constructor(props) {
    super(props);

    this.state = {
      counter: 0
    }
  }

  render() {
    console.log("111");
    return (
      <div>
        <h2>当前计数: {this.state.counter}</h2>
        <button onClick={e => this.increment()}>+1</button>
        <button onClick={e => this.decrement()}>-1</button>
      </div>
    )
  }

  increment() {
    this.setState({counter: this.state.counter + 1});
  }

  decrement() {
    this.setState({counter: this.state.counter - 1});
  }
}
```

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

export default function Counter2() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>当前计数: {count}</h2>
      <button onClick={e => setCount(count + 1)}>+1</button>
      <button onClick={e => setCount(count - 1)}>-1</button>
    </div>
  )
}
```

■ 你会发现上面的代码差异非常大：

- 函数式组件结合hooks让整个代码变得非常简洁
- 并且再也不用考虑this相关的问题；

■ 那么我们来研究一下核心的一段代码代表什么意思：

□ useState来自react，需要从react中导入，它是一个hook；

✓ 参数：初始化值，如果不设置为undefined；

✓ 返回值：数组，包含两个元素；

➢ 元素一：当前状态的值（第一调用为初始化值）；

➢ 元素二：设置状态值的函数；

□ 点击button按钮后，会完成两件事情：

✓ 调用setCount，设置一个新的值；

✓ 组件重新渲染，并且根据新的值返回DOM结构；

Tip:

➢ Hook指的类似于useState、useEffect这样的函数

➢ Hooks是对这类函数的统称；

■ 相信通过上面的一个简单案例，你已经会喜欢上Hook的使用了。

□ Hook 就是 JavaScript 函数，这个函数可以帮助你 钩入 (hook into) React State以及生命周期等特性；

■ 但是使用它们会有两个额外的规则：

□ 只能在函数最外层调用 Hook。不要在循环、条件判断或者子函数中调用。

□ 只能在 React 的函数组件中调用 Hook。不要在其他 JavaScript 函数中调用。

认识useState

■ State Hook的API就是 useState，我们在前面已经进行了学习：

- useState会帮助我们定义一个 state变量，useState 是一种新方法，它与 class 里面的 this.state 提供的功能完全相同。
 - ✓ 一般来说，在函数退出后变量就会“消失”，而 state 中的变量会被 React 保留。
- useState接受唯一一个参数，在第一次组件被调用时使用来作为初始化值。（如果没有传递参数，那么初始化为undefined）。
- useState的返回值是一个数组，我们可以通过数组的解构，来完成赋值会非常方便。
 - ✓ https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

■ FAQ：为什么叫 useState 而不叫 createState？

- “create” 可能不是很准确，因为 state 只在组件首次渲染的时候被创建。
- 在下一次重新渲染时，useState 返回给我们当前的 state。
- 如果每次都创建新的变量，它就不是 “state” 了。
- 这也是 Hook 的名字总是以 use 开头的一个原因。

■ 当然，我们也可以在一個组件中定义多个变量和复杂变量（数组、对象）

认识Effect Hook

■ 目前我们已经通过hook在函数式组件中定义state，那么类似于生命周期这些呢？

- Effect Hook 可以让你来完成一些类似于class中生命周期的功能；
- 事实上，类似于网络请求、手动更新DOM、一些事件的监听，都是React更新DOM的一些副作用（Side Effects）；
- 所以对于完成这些功能的Hook被称之为 Effect Hook；

■ 假如我们现在有一个需求：页面的title总是显示counter的数字，分别使用class组件和Hook实现：

- 案例代码不再贴出

■ useEffect的解析：

- 通过useEffect的Hook，可以告诉React需要在渲染后执行某些操作；
- useEffect要求我们传入一个回调函数，在React执行完更新DOM操作之后，就会回调这个函数；
- 默认情况下，无论是第一次渲染之后，还是每次更新之后，都会执行这个 回调函数；

需要清除Effect

■ 在class组件的编写过程中，某些副作用的代码，我们需要在componentWillUnmount中进行清除：

- 比如我们之前的事件总线或Redux中手动调用subscribe；
- 都需要在componentWillUnmount有对应的取消订阅；
- Effect Hook通过什么方式来模拟componentWillUnmount呢？

■ useEffect传入的回调函数A本身可以有一个返回值，这个返回值是另外一个回调函数B：

```
type EffectCallback = () => (void | (() => void | undefined));
```

■ 为什么要在 effect 中返回一个函数？

- 这是 effect 可选的清除机制。每个 effect 都可以返回一个清除函数；
- 如此可以将添加和移除订阅的逻辑放在一起；
- 它们都属于 effect 的一部分；

■ React 何时清除 effect？

- React 会在组件更新和卸载的时候执行清除操作；
- 正如之前学到的，effect 在每次渲染的时候都会执行；

使用多个Effect

- 使用Hook的其中一个目的就是解决class中生命周期经常将很多的逻辑放在一起的问题：
 - 比如网络请求、事件监听、手动修改DOM，这些往往都会放在componentDidMount中；
- 使用Effect Hook，我们可以将它们分离到不同的useEffect中：
 - 代码不再给出
- Hook 允许我们按照代码的用途分离它们，而不是像生命周期函数那样：
 - React 将按照 effect 声明的顺序依次调用组件中的每一个 effect；

Effect性能优化

■ 默认情况下，useEffect的回调函数会在每次渲染时都重新执行，但是这会导致两个问题：

- 某些代码我们只是希望执行一次即可，类似于componentDidMount和componentWillUnmount中完成的事情；（比如网络请求、订阅和取消订阅）；
- 另外，多次执行也会导致一定的性能问题；

■ 我们如何决定useEffect在什么时候应该执行和什么时候不应该执行呢？

- useEffect实际上有两个参数：
- 参数一：执行的回调函数；
- 参数二：该useEffect在哪些state发生变化时，才重新执行；（受谁的影响）

■ 案例练习：

- 受count影响的Effect；

■ 但是，如果一个函数我们不希望依赖任何的内容时，也可以传入一个空的数组 []：

- 那么这里的两个回调函数分别对应的就是componentDidMount和componentWillUnmount生命周期函数了；

useContext的使用

■ 在之前的开发中，我们要在组件中使用共享的Context有两种方式：

- 类组件可以通过 `类名.contextType = MyContext` 方式，在类中获取context；
- 多个Context或者在函数式组件中通过 `MyContext.Consumer` 方式共享context；

■ 但是多个Context共享时的方式会存在大量的嵌套：

- Context Hook允许我们通过Hook来直接获取某个Context的值；

```
export default function ContextHook() {  
  const user = useContext(UserContext);  
  const theme = useContext(ThemeContext);  
  console.log(user);  
  console.log(theme);  
  
  return (  
    <div>  
      ContextHook  
    </div>  
  )  
}
```

■ 注意事项：

- 当组件上层最近的 `<MyContext.Provider>` 更新时，该 Hook 会触发重新渲染，并使用最新传递给 `MyContext provider` 的 context value 值。

- 很多人看到useReducer的第一反应应该是redux的某个替代品，其实并不是。
- useReducer仅仅是useState的一种替代方案：
 - 在某些场景下，如果state的处理逻辑比较复杂，我们可以通过useReducer来对其进行拆分；
 - 或者这次修改的state需要依赖之前的state时，也可以使用；

```
export function counterReducer(state, action) {  
  switch(action.type) {  
    case "increment":  
      return {...state, counter: state.counter + 1}  
    case "decrement":  
      return {...state, counter: state.counter - 1}  
    default:  
      return state;  
  }  
}
```

```
import { counterReducer } from '../reducer/counter'  
  
export default function Home() {  
  const [state, dispatch] = useReducer(counterReducer, {counter: 100});  
  
  return (  
    <div>  
      <h2>当前计数: {state.counter}</h2>  
      <button onClick={e => dispatch({type: "increment"})}>+1</button>  
      <button onClick={e => dispatch({type: "decrement"})}>-1</button>  
    </div>  
  )  
}
```

- 数据是不会共享的，它们只是使用了相同的counterReducer的函数而已。
- 所以，useReducer只是useState的一种替代品，并不能替代Redux。

■ useCallback实际的目的是为了进行性能的优化。

■ 如何进行性能的优化呢？

- useCallback会返回一个函数的 memoized（记忆的） 值；
- 在依赖不变的情况下，多次定义的时候，返回的值是相同的；

```
const memoizedCallback = useCallback(  
  () => {  
    doSomething(a, b);  
  },  
  [a, b],  
);
```

■ 案例

- 案例一：使用useCallback和不使用useCallback定义一个函数是否会带来性能的优化；
- 案例二：使用useCallback和不使用useCallback定义一个函数传递给子组件是否会带来性能的优化；

■ 通常使用useCallback的目的是不希望子组件进行多次渲染，并不是为了函数进行缓存；



useMemo

- **useMemo**实际的目的是为了进行性能的优化。

- 如何进行性能的优化呢？

- **useMemo**返回的也是一个 memoized（记忆的） 值；

- 在依赖不变的情况下，多次定义的时候，返回的值是相同的；

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- **案例：**

- 案例一：进行大量的计算操作，是否有必须要每次渲染时都重新计算；

- 案例二：对子组件传递相同内容的对象时，使用**useMemo**进行性能的优化

- **useRef**返回一个ref对象，返回的ref对象再组件的整个生命周期保持不变。

- **最常用的ref是两种用法：**

- 用法一：引入DOM（或者组件，但是需要是class组件）元素；

- 用法二：保存一个数据，这个对象在整个生命周期中可以保存不变；

```
const refContainer = useRef(initialValue);
```

- **案例：**

- 案例一：引用DOM

- 案例二：使用ref保存上一次的某一个值

useImperativeHandle

- **useImperativeHandle并不是特别好理解，我们一点点来学习。**
- **我们先来回顾一下ref和forwardRef结合使用：**
 - 通过forwardRef可以将ref转发到子组件；
 - 子组件拿到父组件中创建的ref，绑定到自己的某一个元素中；
- **forwardRef的做法本身没有什么问题，但是我们是将子组件的DOM直接暴露给了父组件：**
 - 直接暴露给父组件带来的问题是某些情况的不可控；
 - 父组件可以拿到DOM后进行任意的操作；
 - 但是，事实上在上面的案例中，我们只是希望父组件可以操作的focus，其他并不希望它随意操作；
- **通过useImperativeHandle可以暴露固定的操作：**
 - 通过useImperativeHandle的Hook，将传入的ref和useImperativeHandle第二个参数返回的对象绑定到了一起；
 - 所以在父组件中，使用 inputRef.current时，实际上使用的是返回的对象；
 - 比如我调用了 focus函数，甚至可以调用 printHello函数；

```
const MyInput = forwardRef(function (props, ref) {  
  const inputRef = useRef()  
  useImperativeHandle(ref, () => ({  
    focus() {  
      inputRef.current.focus()  
    }  
  }))  
  
  return <input ref={inputRef} type="text" />  
})
```

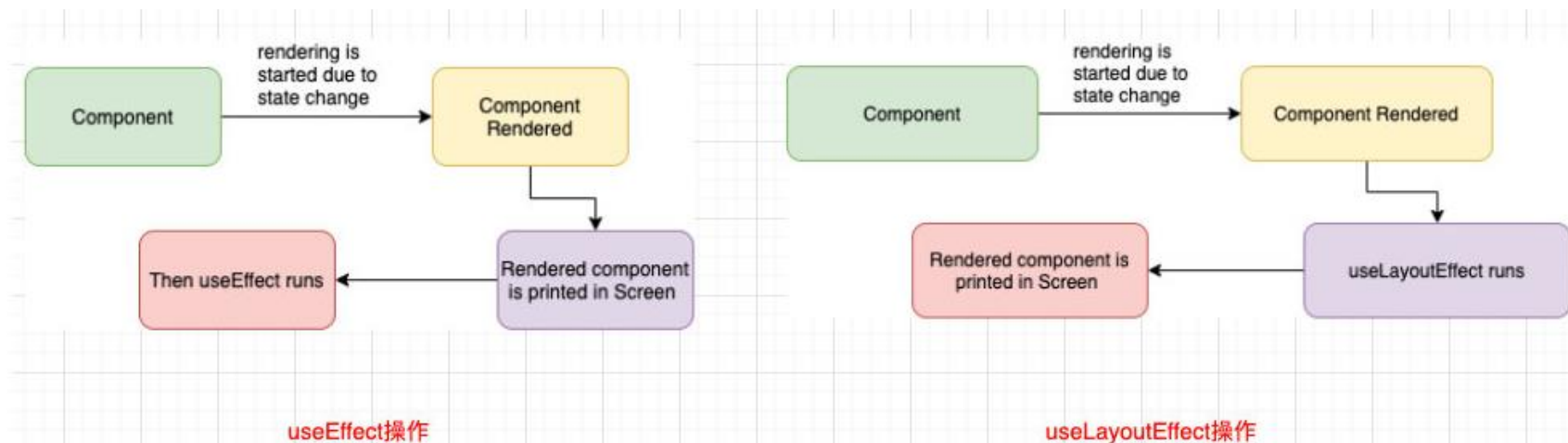
useLayoutEffect

■ useLayoutEffect看起来和useEffect非常的相似，事实上他们也只有一点区别而已：

- useEffect会在渲染的内容更新到DOM上后执行，不会阻塞DOM的更新；
- useLayoutEffect会在渲染的内容更新到DOM上之前执行，会阻塞DOM的更新；

■ 如果我們希望在某些操作发生之后再更新DOM，那么应该将这个操作放到useLayoutEffect。

■ 案例：useEffect和useLayoutEffect的对比



■ 官方更推荐使用useEffect而不是useLayoutEffect。

自定义Hook

■ 自定义Hook本质上只是一种函数代码逻辑的抽取，严格意义上来说，它本身并不算React的特性。

■ 需求：所有的组件在创建和销毁时都进行打印

□ 组件被创建：打印 组件被创建了；

□ 组件被销毁：打印 组件被销毁了；

```
export default function CustomHookDemo() {  
  useEffect(() => {  
    console.log(`组件被创建了`);  
    return () => {  
      console.log(`组件被销毁了`);  
    }  
  }, [])  
  
  return (  
    <div>  
      <h2>CustomHookDemo</h2>  
    </div>  
  )  
}
```

```
function Home(props) {  
  useLoggingLife();  
  return <h2>Home</h2>  
}
```

```
function Profile(props) {  
  useLoggingLife();  
  return <h2>Profile</h2>  
}
```

```
function useLoggingLife(name) {  
  useEffect(() => {  
    console.log(`${name}组件被创建了`);  
    return () => {  
      console.log(`${name}组件被销毁了`);  
    }  
  }, [])  
}
```

自定义Hook练习

■ 需求一：Context的共享

```
function useUserToken() {  
  const user = useContext(UserContext);  
  const token = useContext(TokenContext);  
  
  return [user, token];  
}
```

■ 需求二：获取滚动位置

```
function useScrollPosition() {  
  const [scrollPosition, setScrollPosition] = useState(0);  
  
  useEffect(() => {  
    const handleScroll = () => {  
      setScrollPosition(window.scrollY);  
    }  
    document.addEventListener("scroll", handleScroll);  
  
    return () => {  
      document.removeEventListener("scroll", handleScroll);  
    }  
  }, [])  
  
  return scrollPosition;  
}
```

自定义Hook练习

■ 需求三: localStorage数据存储

```
function useLocalStorage(key) {  
  const [data, setData] = useState(() => {  
    return JSON.parse(window.localStorage.getItem(key))  
  });  
  
  useEffect(() => {  
    window.localStorage.setItem(key, JSON.stringify(data));  
  }, [data]);  
  
  return [data, setData];  
}
```

■ 在之前的redux开发中，为了让组件和redux结合起来，我们使用了react-redux中的connect：

- 但是这种方式必须使用高阶函数结合返回的高阶组件；
- 并且必须编写：mapStateToProps和 mapDispatchToProps映射的函数；

■ 在Redux7.1开始，提供了Hook的方式，我们再也不需要编写connect以及对应的映射函数了

■ **useSelector**的作用是将state映射到组件中：

- 参数一：将state映射到需要的数据中；
- 参数二：可以进行比较来决定是否组件重新渲染；（后续讲解）

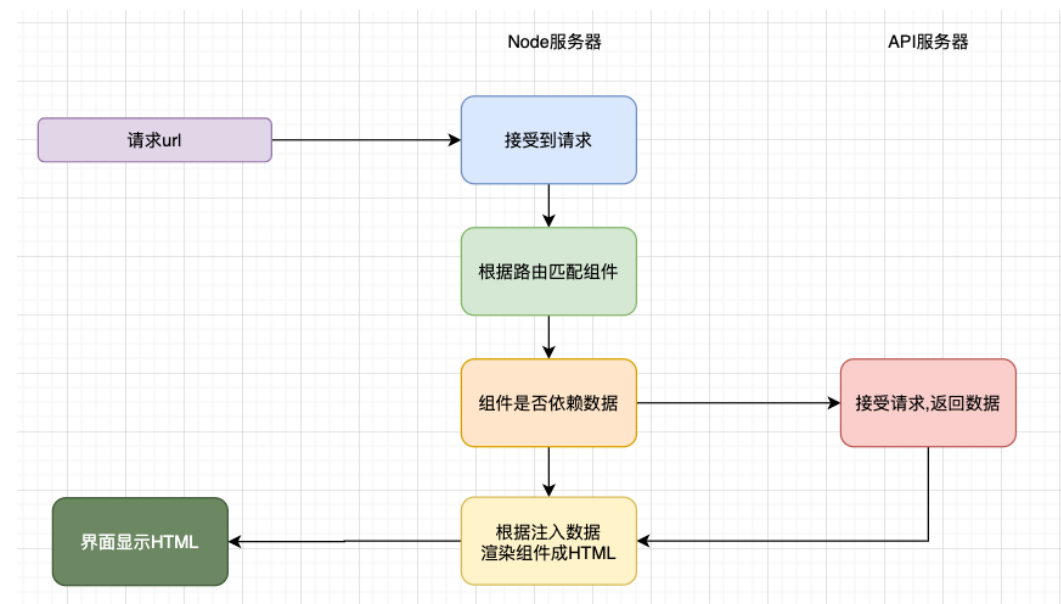
■ **useSelector**默认会比较我们返回的两个对象是否相等；

- 如何比较呢？ `const refEquality = (a, b) => a === b`；
- 也就是我们必须返回两个完全相等的对象才可以不引起重新渲染；

■ **useDispatch**非常简单，就是直接获取dispatch函数，之后在组件中直接使用即可；

■ 我们还可以通过**useStore**来获取当前的store对象；

- 官方的解释：useld 是一个用于生成横跨服务端和客户端的稳定的唯一 ID 的同时避免 hydration 不匹配的 hook。
- 这里有一个词叫hydration，要想理解这个词，我们需要理解一些服务器端渲染（SSR）的概念。
- 什么是SSR？
 - SSR（**Server Side Rendering，服务端渲染**），指的是页面在服务器端已经生成了完成的HTML页面结构，不需要浏览器解析；
 - 对应的是CSR（**Client Side Rendering，客户端渲染**），我们开发的SPA页面通常依赖的就是客户端渲染；
- 早期的服务端渲染包括PHP、JSP、ASP等方式，但是在目前前后端分离的开发模式下，前端开发人员不太可能再去学习PHP、JSP等技术来开发网页；
- 不过我们可以借助于Node来帮助我们执行JavaScript代码，提前完成页面的渲染；



■ 什么是同构?

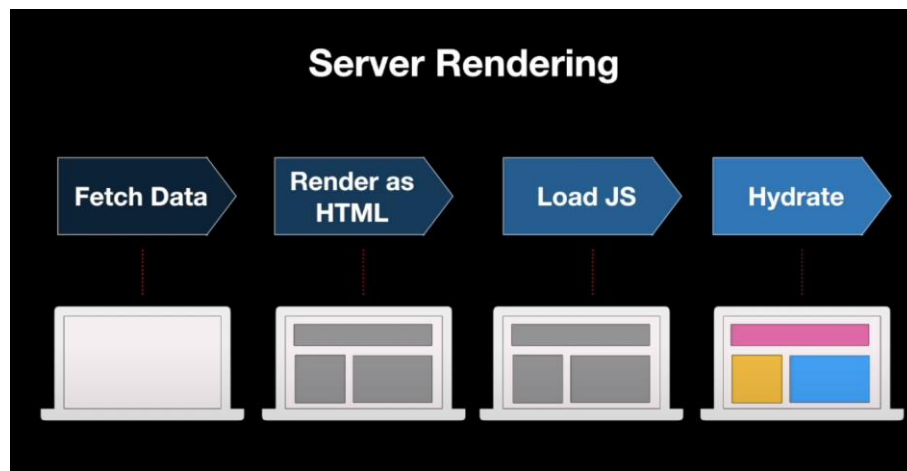
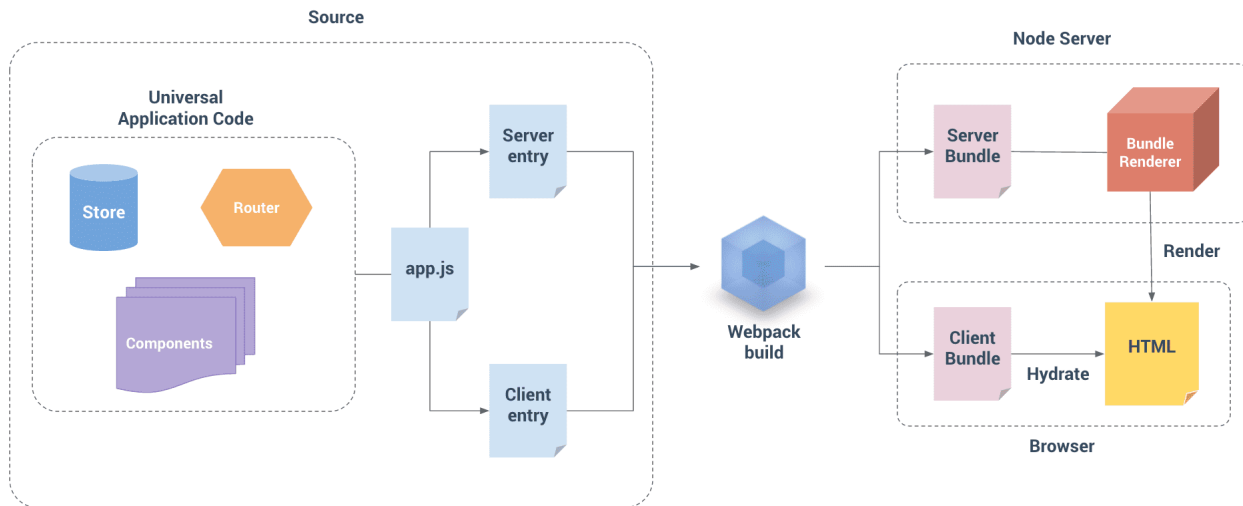
□ 一套代码既可以在服务端运行又可以在客户端运行，这就是同构应用。

■ 同构是一种SSR的形态，是现代SSR的一种表现形式。

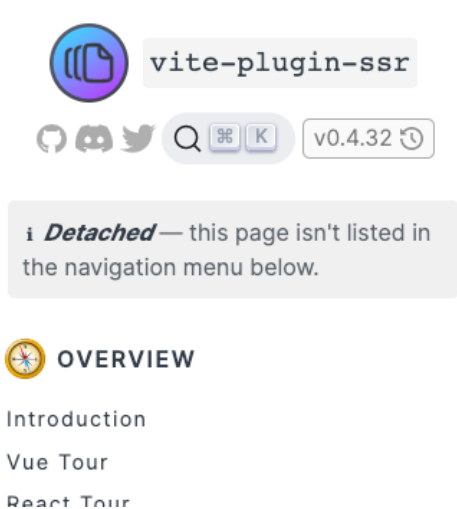
□ 当用户发出请求时，先在服务器通过SSR渲染出首页的内容。

□ 但是对应的代码同样可以在客户端被执行。

□ 执行的目的包括事件绑定等以及其他页面切换时也可以在客户端被渲染；



■ 什么是Hydration? 这里我引入vite-plugin-ssr插件的官方解释。



What is Hydration?

When doing SSR our pages are rendered to HTML. But HTML alone is not sufficient to make a page interactive. For example, a page with zero browser-side JavaScript cannot be interactive (there are no JavaScript event handlers to react to user actions such as a click on a button).

To make our page interactive, in addition to render our page to HTML in Node.js, our UI framework (Vue/React/...) also loads and renders the page in the browser. (It creates an internal representation of the page, and then maps the internal representation to the DOM elements of the HTML we rendered in Node.js.)

This process is called *hydration*. Informally speaking: it makes our page interactive/alive/hydrated.

■ 在进行 SSR 时，我们的页面会呈现为 HTML。

- ❑ 但仅 HTML 不足以使页面具有交互性。例如，浏览器端 JavaScript 为零的页面不能是交互式的（没有 JavaScript 事件处理程序来响应用户操作，例如单击按钮）。
- ❑ 为了使我们的页面具有交互性，除了在 Node.js 中将页面呈现为 HTML 之外，我们的 UI 框架（Vue/React/...）还在浏览器中加载和呈现页面。（它创建页面的内部表示，然后将内部表示映射到我们在 Node.js 中呈现的 HTML 的 DOM 元素。）

■ 这个过程称为hydration。

useId的作用

- 我们再来看一遍：useId 是一个用于生成横跨服务端和客户端的稳定的唯一 ID 的同时避免 hydration 不匹配的 hook。
- 所以我们可以得出如下结论：
 - useId是用于react的同构应用开发的，前端的SPA页面并不需要使用它；
 - useId可以保证应用程序在客户端和服务端生成唯一的ID，这样可以有效的避免通过一些手段生成的id不一致，造成 hydration mismatch；

```
const App = memo((props) => {  
  const [count, setCount] = useState(0)  
  const id = useId()  
    
  return (  
    <div>  
      <h2>计数: {count}</h2>  
      <button onClick={e => setCount(count+1)}>+1</button>  
        
      <label htmlFor={id}>  
        用户名:  
        <input id={id} type="text" />  
      </label>  
    </div>  
  )  
})
```

useTransition

■ 官方解释：返回一个状态值表示过渡任务的等待状态，以及一个启动该过渡任务的函数。

□ 事实上官方的说法，还是让人云里雾里，不知所云。

■ useTransition到底是干嘛的呢？它其实在告诉react对于某部分任务的更新优先级较低，可以稍后进行更新。

```
const App = memo((props) => {  
  const [showNames, setShowNames] = useState(fakerNames)  
  const [isPending, startTransition] = useTransition()  
  function valueChange(e) {  
    startTransition(() => {  
      const filterNames = fakerNames.filter(item => item.includes(e.target.value))  
      setShowNames(filterNames)  
    })  
  }  
  
  return (  
    <div>  
      <input type="text" onChange={e => valueChange(e)} />  
      <h2>名单列表{isPending && <h3>loading</h3>}</h2>  
      <ul> ...  
    </ul>  
    </div>  
  )  
})
```

useDeferredValue

- 官方解释：useDeferredValue 接受一个值，并返回该值的新副本，该副本将推迟到更紧急地更新之后。
- 在明白了useTransition之后，我们就会发现useDeferredValue的作用是一样的效果，可以让我们的更新延迟。

```
const App = memo((props) => {  
  const [showNames, setShowNames] = useState(fakerNames)  
  const deferredNames = useDeferredValue(showNames)  
  function valueChange(e) {  
    const filterNames = fakerNames.filter(item => item.includes(e.target.value))  
    setShowNames(filterNames)  
  }  
  ...  
  return (  
    <div>  
      <input type="text" onChange={e => valueChange(e)} />  
      <ul> ...  
    </ul>  
    </div>  
  )  
})
```