React@张天禹

React 概述、React 基本语法、React 脚手架、React Ajax、React Router、Redux、React 补充

> 附：React 代码测试环境

React 概述

什么是 React?

1. 介绍：React 是一个将**数据**渲染为 **HTML 视图**的开源 JavaScript 库。
2. 原生 JavaScript 的缺点
   a. 操作 DOM 繁琐、效率低
   b. 直接操作 DOM，浏览器会进行大量的**重绘重排**
   c. 没有**组件化**编码方案，代码复用率低
3. React 的特点
   a. 采用**组件化模式、声明化编码**，提高了开发效率及组件复用率
   b. React Native 中可以使用 React 语法进行**移动端开发**
   c. 使用**虚拟 DOM** 以及 **Diff 算法**，尽量减少与真实 DOM 的交互

Hello React

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="UTF-8" />
    <title>Hello World</title>
    <!-- React 核心库。此时全局有 React 对象。 -->
    <script src="https://unpkg.com/react@18/umd/react.development.js"></script>
    <!-- 用于支持 React 操作 DOM 的扩展库。此时全局有 ReactDOM 对象。 -->
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js">
</script>

    <!-- babel，除了可以将 JS 代码从 ES6 转为 ES5 之外，这里还可以将 JSX 代码转换为
    JS 代码 -->
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

```
</head>

<body>
    <!-- 准备好一个容器 -->
    <div id="root"></div>
    <!-- 这里一定要写 text/babel，表示 script 中的代码为 JSX 代码，需要解析为 JS 代码 -->
    <script type="text/babel">

        const VDOM = <h1>Hello, React!</h1>; // 虚拟 DOM
        const container = document.getElementById('root'); // 容器
        ReactDOM.render(VDOM, container); // 将虚拟 DOM 渲染到页面中的容器

    </script>
</body>

</html>
```

> React 使用 `ReactDOM.render(virtualDOM, containerDOM)` 将虚拟 DOM 在页面的真实 DOM 中渲染并显示。
>
> **最新**的渲染虚拟 DOM 的方式如下，
>
> ```
> const root = ReactDOM.createRoot(containerDOM);
> root.render(virtualDOM)
> ```

虚拟 DOM Vs. 真实 DOM

- 虚拟 DOM 本质是 JavaScript 中的一般对象。
- 虚拟 DOM 比较“轻”，真实 DOM 比较“重”。虚拟 DOM 是在 React 内部中使用，因此无需真实 DOM 上的那么多属性。
- 虚拟 DOM 最终会被 React 转化为真实 DOM，并呈现在页面上。

JSX 语法规则

1. 介绍：JSX 全称是 JavaScript XML，是 React 定义的一种类似于 XML 的 JavaScript 扩展语法。JSX 的本质是 `React.createElement(component, props, ...children)` 方法的语法糖，用于简化虚拟 DOM 的创建。
2. 举例：虚拟 DOM（JSX 标签）的创建

```
var element = <h1>Hello JSX!</h1> // element 是一个 JavaScript 对象，表示虚拟
```

```
DOM
```

3. JSX 语法规则
   a. 定义 JSX 标签时，**不要写引号**。
   b. JSX 标签中使用 `{}` **嵌入 JavaScript 表达式**。

   > 区分表达式和语句：简单来说，**一个表达式会产生一个值**。
   >
   > - 表达式：`a`、`a+b`、`demo(1)`、`arr.map()` 等等
   > - 语句：`if(){}`、`for(){}`、`switch(){case}` 等等
   >
   > **React 会自动遍历数组中的所有元素**，并将其呈现在页面上。如果要通过这种方式进行条件渲染来生成一组 JSX 标签，则一定要加上 `key` 属性。

   c. JSX 标签通过 `className` 指定**类名**。
   d. JSX 标签通过 `style={{key:value}}` 的方式指定**内联样式**。
   e. JSX 标签**只能有一个根标签**，其中 `<></>` 是一个特殊的标签，可以作为根标签。
   f. JSX 标签**必须闭合**。
   g. JSX 标签的首字母**以小写开头，React 会将该标签转换为 HTML 同名元素**，如果不存在对应的同名元素，则报错；**以大写开头，React 将该标签看作 React 组件**，如果没有定义对应的组件，则报错。

模块化与组件化

1. 模块：向外提供特定功能的 Javascript 程序，一般就是一个 Javascript 文件。有利于**复用** Javascript，**简化** Javascript 的编写，提高 Javascript 运行**效率**。
2. 组件：用于实现局部功能效果的代码和资源的集合(html+css+js+image 等)。有利于**复用**代码，**简化**项目，提高运行**效率**。

React 开发者工具

Chrome 浏览器扩展：**React Developer Tools**。安装后 F12 控制台出现两个新的选项卡 Components 和 Profiler。Component 用于查看 React 项目中的组件，Profiler 则与项目性能有关。

React 基本语法

组件的两种写法

1. **Function** 组件

```
1     <script type="text/babel">
```

```
        function MyComponent1() {
            return <h2>我是 Function 组件</h2>
        }

        const container1 = document.getElementById('root1'); // 容器
        ReactDOM.render(<MyComponent1 />, container1); // 将虚拟 DOM 渲染到页面中的容器
    </script>
```

2. **Class** 组件

```
    <script type="text/babel">
        class MyComponent2 extends React.Component {
            render(){
                return <h2>我是 Class 组件</h2>
            }
        }

        const container2 = document.getElementById('root2'); // 容器
        ReactDOM.render(<MyComponent2 />, container2); // 将虚拟 DOM 渲染到页面中的容器
    </script>
```

3. 注意
   a. 组件名**首字母必须大写**。
   b. **Function** 组件中的 `this` 是 `undefined`。这是因为 babel 编译 JSX 后开启了严格模式。函数的返回值就是虚拟 DOM。
   c. **Class** 组件中的 `this` 是**组件实例对象**。其中的 `render` 方法存在于组件实例对象的原型上，用于返回虚拟 DOM。

组件的三大属性（Class 组件）

State

1. 介绍：每个 Class 组件实例对象都有一个 `state` 属性，其值是一个对象。组件被称作“状态机”，**通过更新组件的 `state` 属性可以触发组件的重新渲染**，从而更新对应的页面显示。
   a. 定义 **state**：每个 Class 组件实例对象都有一个 `state` 属性。
      - 完整写法

```
class MyComponent extends React.Component {
```

```
    constructor(props) {
        super(props);
        this.state = obj;
    }
    // ...
}
```

- 简化写法

```
class MyComponent extends React.Component {
    state = obj
    // ...
}
```

b. 更新 **state**：每个 Class 组件实例对象都有一个 `setState` 方法。本职是合并操作。`state` 更新 => `render` 调用 => 视图更新。

- 完整写法

```
class MyComponent extends React.Component {
    constructor(props) {
        // ...
        this.handleClick= this.handleClick.bind(this); // 黄色方法在实例上，绿色方法在原型上
    }
    handleClick() { // 原型上的方法
        this.setState(newObj)
    }
}
```

- 简化写法

```
class MyComponent extends React.Component {
    handleClick = () => { // 实例上的方法
        this.setState({ isHot: !this.state.isHot })
    }
}
```

2. 案例：天气切换

a. 完整写法

```
<script type="text/babel">
    class Weather extends React.Component {
        // 1. 初始化状态 2. 修正事件响应函数的 this => 调用一次
        constructor(props) {
            super(props);
            // 初始化状态
            this.state = { isHot: false };
            // 修正 this：将一个绑定了 this 的函数放在实例上，其 this 永远指向组件实例对象
            // "=" 前边的 changeIsHot 在实例对象上，后边的 changeIsHot 在原型对象上
            this.changeIsHot = this.changeIsHot.bind(this);
        }
        // 事件响应函数（在原型上） => click 几次，调用几次
        changeIsHot() {
            // 更新状态，会触发视图的更新
            this.setState({ isHot: !this.state.isHot })
        }
        // 渲染函数（在原型上），返回虚拟 DOM => 调用 1 + n 次，初始化视图时调用一次，每次状态改变时会触发视图更新，也会调用一次
        render() {
            return (
                <h1 onClick={this.changeIsHot}
                    style={{ cursor: "pointer" }}>
                    今天天气很{this.state.isHot ? "炎热" : "凉爽"}
                </h1>
            )
        }
    }

    const container = document.getElementById('root'); // 容器
    ReactDOM.render(<Weather />, container); // 将虚拟 DOM 渲染到页面中的容器

</script>
```

b. 简化写法（Better）

```
<script type="text/babel">
    class Weather extends React.Component {
        // 初始化状态
        state = { isHot: false };
        // 事件响应函数（在实例上），此时由于箭头函数没有自己的 this，因此该事件响应函数的 this 就是实例对象
```

```
        changeIsHot = () => {
            this.setState({ isHot: !this.state.isHot })
        }
        // 渲染函数（在实例上）
        render() {
            return (
                <h1 onClick={this.changeIsHot}
                    style={{ cursor: "pointer" }}>
                    今天天气很{this.state.isHot ? "炎热" : "凉爽"}
                </h1>
            )
        }
    }

    const container = document.getElementById('root'); // 容器
    ReactDOM.render(<Weather />, container); // 将虚拟 DOM 渲染到页面中的容器

</script>
```

3. 注意事项
   a. React 中将原生的 `onxxx` 事件绑定属性重写了一份，为 `onXxx` 的形式，如 `onclick => onClick`、`onblur => onBlur` 等。
   b. 类中的 `constructor` 和 `render` 方法中的 `this` 都是组件实例对象，但是将自定义的方法作为事件响应函数时，函数内部的 `this` 是 `undefined`。
      - `constructor` 中的 `this` 指向组件实例对象 => 构造函数 的 `this` 指向其创建的实例。
      - `render` 是由 React 根据组件实例对象调用的 => 以**实例.方法**的形式调用的函数，其中的 `this` 指向调用它的实例。
      - 事件绑定函数作为 `onXxx` 属性的回调，是直接调用的 => 直接调用的函数，其中的 `this` 指向 `window`。又因为类中默认开启了严格模式，因此 `this` 值为 `undefined`。
   c. **事件响应函数 `this` 的修正（bind）**：假设我们已经在类的**原型**上定义了 `handleClick` 方法，那么可以在构造器中添加一行代码。此时**实例身上和其原型身上**各有一个 `handleClick` 方法，并且实例对象上的 `handleClick` 方法的 `this` 永远指向实例对象，无论以何种方式调用。

```
class Component{
    constructor(props){
        // ...
```

```
        this.handleClick = this.handleClick.bind(this); // 将原型上的 handleClick 方法的 this 修改后添加到实例对象身上，此时原型上的 handleClick 方法没有改变
        // ...
    }
    handleClick(){ // 原型身上的方法
        // ...
    }
}
```

d. **事件响应函数 `this` 的修正（箭头函数）**：直接定义一个在**实例**身上的箭头函数，此时由于箭头函数没有自己的 `this`，因此无论以何种方式调用，其 `this` 都指向其上下文，即组件实例对象。

```
class Component{

    handleClick => (){ // 实例身上的方法
        // ...
    }
}
```

Props

1. 介绍：每个 Class 组件实例对象都有一个 `props` 属性，React 将所有组件标签的属性都保存在 `props` 中。其中，`props` 中的所有属性都是**只读**的。
2. 使用
   a. 基本使用：传 `props` & 读 `props`

```
class Person extends React.Component {
    render() {
        const { name, gender, age } = this.props; // 解析 props
        return <></>
    }
}

ReactDOM.render(<Person name="Tom" gender="女" age="18" />, document.getElementById('root1')); // 传入 props

const jack = { name: "Jack", gender: "男", age: 17 };
ReactDOM.render(<Person {...jack} />, document.getElementById('root2')); // 批量传递 props
```

> 注意：这里的 `...` 不是 Javascript 中的展开运算符，是 React 中用于批量传递 props 的一种语法。

b. 指定规则：限制 `props` 的必要性、类型、默认值

```
class Person extends React.Component {
    render() {
        const { name, gender, age } = this.props;
        return <></>
    }
}
// 对 props 的类型、必要性进行限制
Person.propTypes = {
    name: PropTypes.string.isRequired, // name 属性是字符串类型，其必传
    age: PropTypes.number, // age 属性是数值类型
    gender: PropTypes.string, // gender 属性是字符串类型
    speak: PropTypes.func // speak 属性是函数类型
}
// 对 props 的默认值进行指定
Person.defaultProps = {
    age: 18,
    gender: "男"
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Person name="Tom" gender="女" age={19} />)
```

> 注意：必须引入 `PropTypes` 对象才能对 `props` 进行限制！！！

```
<!-- prop-types 用于对组件的 props 进行限制。此时全局有 PropTypes 对象。 -->
<script src="./assets/prop-types.js"></script>
```

c. 指定规则的简写形式

```
class Person extends React.Component {
    // 对 props 的类型、必要性进行限制
    static propTypes = {
        name: PropTypes.string.isRequired, // name 属性是字符串类型，其必传
        age: PropTypes.number, // age 属性是数值类型
        gender: PropTypes.string, // gender 属性是字符串类型
        speak: PropTypes.func // speak 属性是函数类型
```

```
    }
    // 对 props 的默认值进行指定
    static defaultProps = {
        age: 18,
        gender: "男"
    }
    render() {
        const { name, gender, age } = this.props;
        return <></>
    }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Person name="Tom" gender="女" age={19} />)
```

d. Function 组件的 props 属性：函数接收的参数就是 props

```
function Person(props) {
    const { name, gender, age } = props;
    return <></>
}
// 对 props 的类型、必要性进行限制
Person.propTypes = {
    name: PropTypes.string.isRequired, // name 属性是字符串类型，其必传
    age: PropTypes.number, // age 属性是数值类型
    gender: PropTypes.string, // gender 属性是字符串类型
    speak: PropTypes.func // speak 属性是函数类型
}
// 对 props 的默认值进行指定
Person.defaultProps = {
    age: 18,
    gender: "男"
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Person name="Tom" gender="女" age={19} />)
```

3. 注意事项：对于 Class 组件，如果要使用构造函数，则一定要有以下内容，否则无法在构造函数中通过 `this.props` 的方式访问到传入的 `props`。同时标签体内容作为标签的 `children` 属性传递给标签，即 `this.props.children`。

```
constructor(props){
    super(props);
```

```
3 }
```

Refs

1. 介绍：组件标签可以通过 `ref` 属性标识自己，其取值总共有三种情况，**字符串 `ref`、回调函数 `ref`、 `createRef`**。
2. 字符串 `ref`：通过 `ref="domRef"`，此时组件实例对象上的固有的 `refs` 属性中会增加一个键值对 `{"domRef": 真实 DOM}`。此时可以通过 `this.refs.domRef` 的方式去引用相应的真实 DOM。

> 注意：由于效率问题，字符串 ref 的方式将被禁用，不被推荐！

```
<script type="text/babel">
    class Demo extends React.Component {
        handleClick = () => {
            const { input1 } = this.refs;
            alert(input1.value);
        }

        handleBlur = ()=>{
            const { input2 } = this.refs;
            alert(input2.value);
        }

        render() {
            return (
                <>
                    <input ref="input1" type="text" placeholder="点击按钮提示数据" />
                    <button onClick={this.handleClick}>点我提示左侧数据</button>
                    <input ref="input2" onBlur={this.handleBlur} type="text" placeholder="失去焦点提示数据" />
                </>
            )

        }
    }

    const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(<Demo />)
</script>
```

3. 回调函数 `ref`：通过 `ref={function}`，可以将相应的真实 DOM 添加到组件实例对象的 this 身上。回调函数接收一个参数，表示相应的真实 DOM。
   a. 如果回调函数是以**内联函数**的方式定义，则 `render` 再次调用时，其会执行两次。第一次传入参数为 `null`，第二次传入参数为 DOM 元素。这是因为每次渲染需要创建一个新的函数实例，因此 React 需要先传入 null 清空旧的 `ref`，再传入 DOM 元素设置新的 `ref`。
   b. 如果回调函数是**实例对象上的函数**，则不会出现这种问题，即只会在首次组件渲染时执行一次。
   c. 总而言之，内联函数形式的回调执行 `1 + 2n` 次，实例函数形式的回调执行 `1` 次。

```
<script type="text/babel">
    class Demo extends React.Component {
        state = { isHot: true }

        handleClick = () => {
            alert(this.input1.value);
        }

        handleBlur = () => {
            alert(this.input2.value);
        }

        saveInput1 = (currentNode) => {
            this.input1 = currentNode;
            console.log("@Input1Ref", currentNode);
        }

        changeIsHot = () => {
            const { isHot } = this.state;
            this.setState({ isHot: !isHot });
        }

        render() {
            return (
                <>
                    <h2 onClick={this.changeIsHot}>今天天气真{this.state.isHot ? "炎热" : "凉爽"}</h2>
                    <input ref={this.saveInput1} type="text" placeholder="点击按钮提示数据" />
                    <button onClick={this.handleClick}>点我提示左侧数据</button>
                    <input
                        ref={currentNode => { this.input2 = currentNode; console.log("@Input2Ref", currentNode) }}
                        onBlur={this.handleBlur}
```

```
                        type="text"
                        placeholder="失去焦点提示数据" />
                </>
            )

        }
    }

    const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(<Demo />)
</script>
```

4. `createRef`：通过 `ref={container}`，可以将相应的真实 DOM 添加到由 `React.createRef()` 创建的容器中。然后可以通过 `container.current` 的方式访问到该真实 DOM。
   a. `React.createRef()` 创建一个容器，用于存储 `ref` 所标识的 DOM 节点。
   b. 一个容器只能存储一个 DOM 节点。
   c. 通过容器访问到 DOM 的一般步骤为
      i. 创建容器 `const container = React.createRef()`
      ii. 将 DOM 节点存储到容器中 `<MyComponent ref={container} />`
      iii. 访问容器中的 DOM 节点 `container.current`

```
<script type="text/babel">
    class Demo extends React.Component {
        container1 = React.createRef();
        container2 = React.createRef();

        handleClick = () => {
            alert(this.container1.current.value);
        }

        handleBlur = () => {
            alert(this.container2.current.value);
        }

        render() {
            return (
                <>
                    <input ref={this.container1} type="text" placeholder="点击按钮提示数据" />
```

```
                    <button onClick={this.handleClick}>点我提示左侧数据</button>
                    <input ref={this.container2} onBlur={this.handleBlur} type="text" placeholder="失去焦点提示数据" />
                </>
            )

        }
    }

    const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(<Demo />)
</script>
```

事件处理

React 通过 `onXxx` 属性指定事件响应函数，这是因为

1. React 使用的是自定义事件，而不是原生的 DOM 事件，其做了一些额外处理
2. React 将事件委托给组件最外层元素，然后可以通过 `event.target` 得到发生事件的 DOM 元素

表单提交

在 React 中，受控组件是指其状态由 React 控制和管理（通常通过 `state` 和 `props`），而非受控组件则是指其状态由 DOM 自己管理（通常通过 `ref` 直接访问 DOM 元素）。

非受控组件

以下示例中，`Login` 是一个**非受控组件**，其要使用的状态 `username` 和 `password` 都是由 DOM 管理的（借助 `ref`）。

```
<script type="text/babel">
    class Login extends React.Component {
        handleSubmit = (event) => {
            event.preventDefault();
            const username = this.username.value;
            const password = this.password.value;
            alert(`username=${username}&password=${password}`)
        }

        render() {
            return (
                <form onSubmit={this.handleSubmit}>
                    用户名 <input type="text" ref={currentNode => this.username = currentNode} name="username" /> <br />
```

```
                    密码 <input type="password" ref={currentNode => this.password = currentNode} name="password" /> <br />
                    <button>登录</button>
                </form>
            )
        }
    }

    const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(<Login />)
</script>
```

受控组件

以下示例中，`Login` 是一个**受控组件**，其要使用的状态 `username` 和 `password` 都是由 React 管理的（借助 `state` 和 `onChange`）。

```
<script type="text/babel">
    class Login extends React.Component {
        state = { username: "", password: "" }

        handleSubmit = (event) => {
            event.preventDefault();
            const { username, password } = this.state;
            alert(`username=${username}&password=${password}`)
        }

        handleUsernameChange = (event) => {
            this.setState({ username: event.target.value })
        }

        handlePasswordChange = (event) => {
            this.setState({ password: event.target.value })
        }

        render() {
            return (
                <form onSubmit={this.handleSubmit}>
                    用户名 <input type="text" onChange={this.handleUsernameChange} name="username" /> <br />
                    密码 <input type="password" onChange={this.handlePasswordChange} name="password" /> <br />
                    <button>登录</button>
                </form>
            )
```

```
        }
    }

    const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(<Login />)
</script>
```

受控组件的改进

对于以下两个 `input` 元素，如果要在数据变化（`onChange`）时更新其 `state`，则需要两个对应的事件响应函数。但是如果任意多个字段对应的 input 元素呢？难道任意多个事件响应函数吗？这里有两种解决方式，函数柯里化和箭头函数。

```
用户名 <input type="text" onChange={this.handleUsernameChange} name="username" /> <br />
密码 <input type="password" onChange={this.handlePasswordChange} name="password" /> <br />
```

函数柯里化

函数柯里化是将一个需要多个参数的函数转换成一系列每次只接受一个参数的函数。

```
const saveFormData(dataType) = {
    return (event) => {
        this.setState({[dataType]: event.target.value});
    }
}
```

```
用户名 <input type="text" onChange={this.saveFormData("username")} name="username" /> <br />
密码 <input type="password" onChange={this.saveFormData("password")} name="password" /> <br />
```

箭头函数

```
const saveFormData(dataType, value) = {
    this.setState({[dataType]: value});
}
```

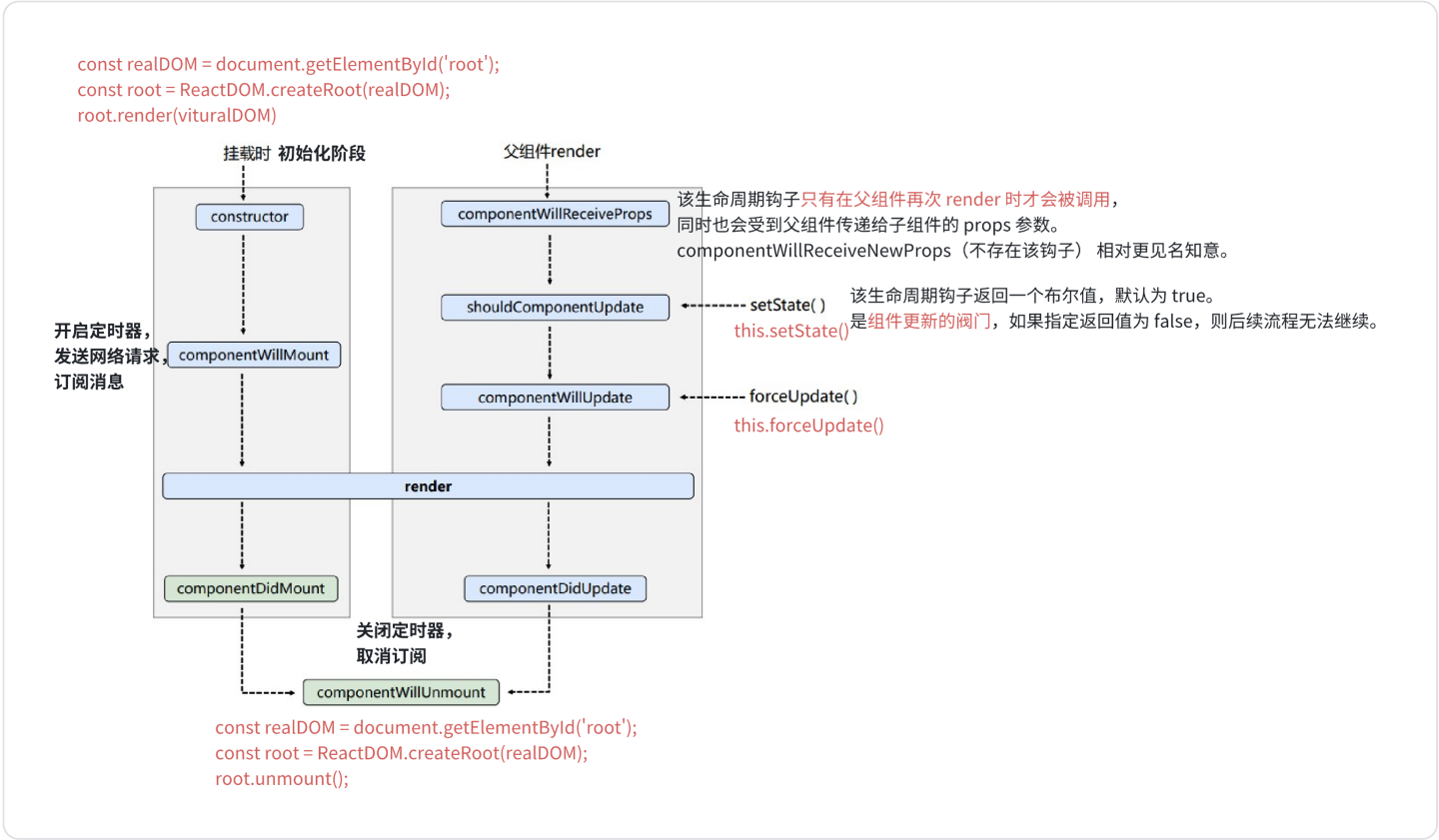
```
1 用户名 <input type="text" onChange={event => this.saveFormData("username",
  event.target.value)} name="username" /> <br />
2 密码 <input type="password" onChange={event => this.saveFormData("password",
  event.target.value)} name="password" /> <br />
```

组件的生命周期

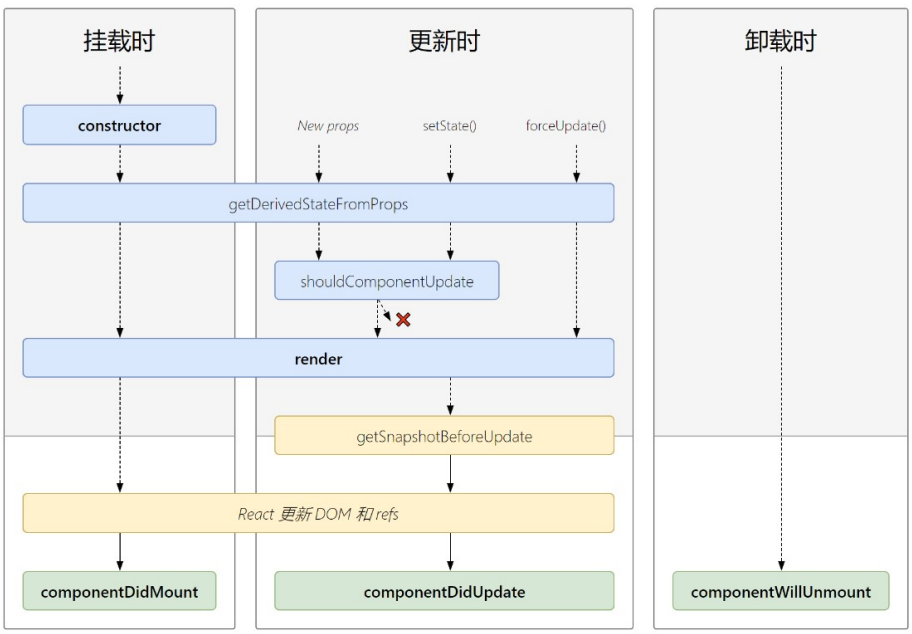
生命周期：又称生命周期钩子、生命周期函数，会在特定的时刻调用。生命周期函数在 Class 组件中定义，React 会在合适的时机调用对应的钩子。

旧生命周期

1. 初始化（当前组件 `render`）：`constructor()` → `componentWillMount()` → `render()` → `componentDidMount()`
2. 更新
   a. 父组件 `render`：`componentWillReceiveProps()` → `shouldComponentUpdate()` → `componentWillUpdate()` → `render()` → `componentDidUpdate()`
   b. 当前组件 `setState`：`shouldComponentUpdate()` → `componentWillUpdate()` → `render()` → `componentDidUpdate()`
   c. 当前组件 `forceUpdate`：`componentWillUpdate()` → `render()` → `componentDidUpdate()`
3. 卸载（当前组件 `unmount`）：`componentWillUnmount()`



新生命周期



1. 新旧生命周期对比
   - 旧的生命周期钩子中的 `componentWillMount`、`componentWillReceiveProps`、`componentWillUpdate` 即将被**废弃**，如果要使用，需要加上 `UNSAFE_` 前缀。
   - 新的生命周期钩子中**新增**了 `getDerivedStateFromProps`、`getSnapshotBeforeUpdate`。
   - 其他流程类似。
2. 新增的生命周期钩子

a. `getDerivedStateFromProps`：罕见使用，用于根据 `props` 派生 `state`，即返回值作为新的 `state`

   i. 函数声明 `static getDerivedStateFromProps(props, state){ return obj | null }`

   ii. 注意事项

      1. 该函数是类方法，需要使用 `static` 修饰，不能声明为实例方法。
      2. 该函数接收两个参数，`props` 表示通过标签属性传入的参数，`state` 表示组件实例对象的状态。
      3. 该函数要么返回 `null`，表示什么都不做；要么返回一个状态对象，作为最新的 `state` 渲染组件。

b. `getSnapshotBeforeUpdate`：罕见使用，用于在更新完成前生成快照，即在最近一次渲染输出之前调用，使得组件能在发生更改之前从 DOM 中捕获一些信息，并将其返回值作为参数传递给 `componentDidUpdate`。

   i. 函数声明：`getSnapshotBeforeUpdate(prevProps, prevState){return snapshot | null}`

   > `componentDidUpdate` 接收的参数为 `componentDidUpdate(prevProps, prevState)`。这里的 `prevPros` 表示上一次的 `props`，`prevState` 表示上一次的 `state`。
   >
   > 相应的，`componentDidUpdate` 接收的参数为 `componentDidUpdate(prevProps, prevState, snapshot)`。

   ii. 应用场景：获取滚动位置，当内容增加时内容不自动滚动，转而滚动条滚动。

```
<script type="text/babel">
    class NewsList extends React.Component {
        state = { newsArr: [] }

        /* 组件挂载成功 */
        componentDidMount() {
            /* 模拟每秒生成一篇新闻 */
            this.timer = setInterval(() => {
                const { newsArr } = this.state;
                const news = "新闻" + (newsArr.length + 1);
                this.setState({ newsArr: [news, ...newsArr] });
            }, 1000)
        }

        /* 组件将要卸载 */
        componentWillUnmount() {
            /* 清除定时器 */
```

```
            clearInterval(this.timer);
        }

        /* 组件更新，已渲染但未提交到页面 */
        getSnapshotBeforeUpdate(prevProps, prevState) {
            /* 生成快照(未更新时的 scrollHeight)，避免滚动条滚动 */
            return { lastScrollHeight: this.newsDOM.scrollHeight };
        }

        /* 组件更新成功 */
        componentDidUpdate(prevProps, prevState, snapshot) {
            /*
                this.newsDOM.scrollHeight 表示更新后的元素内容总高度（包括溢出的不可见内容）
                this.newsDOM.scrollTop 表示更新后的元素内容顶部到可视区域顶部的距离，可用于读取或设置滚动条的当前位置
            */
            this.newsDOM.scrollTop += this.newsDOM.scrollHeight - snapshot.lastScrollHeight;
        }

        render() {
            return (
                <ul style={{
                    width: 100,
                    height: 200,
                    backgroundColor: "pink",
                    overflow: "scroll"
                }}
                    ref={currentNode => this.newsDOM = currentNode} >
                    {this.state.newsArr.map((news, index) => <li key={index}>{news}</li>)}
                </ul>
            )
        }
    }

    const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(<NewsList />)
</script>
```

DOM Diff 算法

1. 虚拟 DOM 中 `key` 的作用：`key` 是虚拟 DOM 的**标识**，在更新显示时起着极其重要的作用。**当组件的状态数据发生变化时，React 会根据新数据生成新的虚拟 DOM，随后进行新虚拟 DOM 与旧虚拟 DOM 的 Diff 比较**。规则如下，

    a. If 旧虚拟 DOM 中找到了与新虚拟 DOM 相同的 `key`
        i. If 虚拟 DOM 中的内容没变，Then 复用之前的真实 DOM
        ii. Else Then 生成新的真实 DOM，替换之前的真实 DOM
    b. Else Then 生成新的真实 DOM，并渲染到页面
2. 选择 `key={index}` 可能会引发的问题
    a. If 对数据进行破坏顺序的操作，如逆序添加、逆序删除等，Then 产生没必要的真实 DOM 更新，此时界面效果没问题，但效率低
    b. If 结构中包含输入类 DOM，Then 产生错误的 DOM 更新，此时界面会有问题

    > 如果仅对用于渲染列表，则使用 `index` 作为 `key` 是没有问题的。
3. 选择 `key` 的策略：最好使用每条数据的**唯一标识**作为 `key`，如 `id`、手机号、身份证号等唯一值。如果只是简单的展示数据，也可以使用 `index`。

React 脚手架

脚手架

1. xxx 脚手架：用于帮助程序员**快速创建一个基于 xxx 库的模板项目**的工具。其中包含了所需要的配置、下载了所有相关依赖、并可以直接运行一个简单效果。
2. create-react-app：React 提供的一个用于创建 React 项目的脚手架库。项目的整体架构为 react + webpack + es6 + eslint + ...
3. 使用脚手架的项目特点：模块化、组件化、工程化

使用 create-react-app

Step 1. 全局安装 create-react-app

```
1 npm i -g create-react-app
```

Step 2. 在指定目录下创建项目

```
1 create-react-app <project-name>
```

Step 3. 进入项目文件夹并启动项目

```
1 cd <project-name>
```

```
npm start
```

> 项目常用命令
>
> `npm start` 启动项目
>
> `npm run build` 打包项目
>
> `npm run test` 测试项目
>
> `npm run eject` 暴露隐藏的依赖文件，如 `webpack.config.js`

脚手架的项目结构

1. 项目结构

```
public 静态资源文件夹
-- favicon.ico 网站页签图标
-- index.html 主页面
-- logo192.png、logo512.png logo 图
-- manifest.json 应用加壳的配置文件
-- robots.txt 爬虫协议文件
src 项目源代码文件夹
-- App.css App 组件的样式
-- App.js App 组件
-- App.test.js 测试 App 组件的文件
-- index.css 全局样式
-- index.js 入口文件
-- logo.svg logo 图
-- reportWebVitals.js 页面性能分析文件，需要 web-vitals 库的支持
-- setupTests.js 组件单元测试文件，需要 jest-dom 库的支持
```

2. `public/index.html`

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <!-- %PUBLIC_URL% 表示 public 文件夹的路径 -->
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <!-- 表示开启理想视口，用于移动端网页的适配 -->
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <!-- 表示配置浏览器页签 + 地址栏的颜色，仅支持安卓手机浏览器 -->
  <meta name="theme-color" content="#000000" />
```

```
  <!-- 表示网页描述性信息 -->
  <meta name="description" content="Web site created using create-react-app" />
  <!-- 表示添加网页到手机主屏幕后的图标，仅支持苹果 Safari 浏览器 -->
  <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
  <!-- 表示应用加壳时的配置文件，表示给网页应用套上一层壳，使其可以被下载 -->
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
  <title>React App</title>
</head>

<body>
  <!-- 如果浏览器不支持 JavaScript，则显示该标签中的内容 -->
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
</body>

</html>
```

3. `src/index.js`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  /* React.StrictMode 为内部组件树启用额外的开发行为和警告 */
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

4. `src/App.js`

```
import './App.css';

function App() {
  return (
      // jsx
  );
}

```

```
9 export default App;
```

VSCODE 插件 - 代码片段提示

1. 插件名：ES7+ React/Redux/React-Native snippets
2. 常用片段提示
   - `rcc` 生成 React Class Component
   - `rfc` 生成 React Function Component

组件化编码流程

Step 1. 拆分组件

Step 2. 实现静态组件

Step 3. 实现动态组件：动态显示初始化数据、用户交互

TodoList 案例

- 拆分组件、实现静态组件时注意 `className` 和 `style` 的写法。
- 动态初始化数据时，如果数据仅某个组件使用，则将数据放在其自身的 `state` 中；如果数据被某些组件使用，则将数据放在其共同的父组件的 `state` 中 —— **状态提升**。
- 父向子通信，父组件使用 props 向子组件直接传递数据；子向父通信，父组件使用 props 向子组件传递一个函数，子组件使用这个函数向父组件传递数据。
- 输入类标签的 `defaultChecked` 和 `defaultValue` 仅在第一次赋值时生效，后续取值改变不会再影响标签的渲染。为了避免这种情况，建议使用 `checked` 和 `value`。
- 安装 `prop-types` 用于 props 的类型控制。
  - 安装 `yarn add -D prop-types`
  - 引入 `import PropTypes from "prop-types"`
  - 使用（类中） `static propTypes = { propName: PropTypes.<dataType>.<isRequired> }`
- **状态在哪里，操作状态的方法就在哪里。**

React Ajax

React 使用 `axios` 发送给 Ajax 请求，使用 `yarn add axios` 安装 `axios`。

`axios` 是对 `xhr` 的封装，React 也可以使用 `fetch` 发送给请求，体现**关注分离**的设计思想。

配置代理的两种方式

1. 方式一：在 `package.json` 中添加以下配置。

```
1  "proxy":"http://localhost:5000" // 表示将所有 ajax 请求转发到 5000 端口
```

   - 作用：假设前端网页部署在 3000 端口，配置代理后，当请求了 3000 端口不存在的资源时，该请求会转发给 5000 端口。需要注意的是，发送请求时会优先匹配前端资源，即 `/public` 文件夹中的资源。
   - 优点：配置简单，前端请求资源时可以不加任何前缀。
   - 缺点：不能配置多个代理。

2. 方式二：创建 `src/setupProxy.js` 并在其中添加以下配置。

```
1 const proxy = require('http-proxy-middleware')
2
3 module.exports = function(app) {
4   app.use(
5     proxy('/api1', { // 所有以 "/api1" 开头的请求按照该代理配置转发到 5000 端口
6       target: 'http://localhost:5000', // 服务器地址，或者说请求转发目标地址
7       changeOrigin: true, // 控制服务器接受到的请求头中 HOST 字段的值，一般设置为 true
8       /* true, 则服务器收到 HOST: localhost:5000; false, 则服务器收到 HOST: localhost: 3000 */
```

```
      pathRewrite: {'^/api1': ''} // 去除用于匹配的请求前缀，确保交给服务器的是正常请求地址
    }),
    proxy('/api2', { // 所有以 "/api2" 开头的请求按照该代理配置转发到 5001 端口
      target: 'http://localhost:5001',
      changeOrigin: true,
      pathRewrite: {'^/api2': ''}
    })
  )
}
```

- 优点：可以配置多个代理，可以灵活的控制请求是否走代理。
- 缺点：配置繁琐，前端请求资源时必须加前缀。

github 搜索案例

组件间通信 - pubsub-js

1. 下载 `yarn add pubsub-js`
2. 使用
   a. 引入 `import PubSub from 'pubsub-js'`
   b. 订阅消息

```
var mySubscriber = function (msg, data) {console.log( msg, data );};
var token = PubSub.subscribe('MY TOPIC', mySubscriber);
```

   c. 发布消息

```
// 异步
PubSub.publish('MY TOPIC', 'hello world!');
// 同步（不推荐）
PubSub.publishSync('MY TOPIC', 'hello world!');
```

   d. 取消订阅

```
// 取消特定消息的订阅
PubSub.unsubscribe(token);
// 取消特定回调的订阅
PubSub.unsubscribe(mySubscriber);、
```

```
// 取消所有消息的订阅
PubSub.clearAllSubscriptions();
```

3. 注意事项
   a. 在 `componentDidMount` 中订阅消息
   b. 在 `componentWillUnmount` 中取消订阅

React Router@5

理解 SPA

1. 解释：SPA（single page web application），即单页 Web 应用中，**一个项目只有一个 HTML 页面**，一旦加载完成就不会进行重新加载或跳转，取而代之的是**通过使用 JS 动态的改变这单个 HTML 页面的内容，模拟多页面的跳转**。
2. 分析
   a. 优点
      i. **页面更加流畅，用户的体验更好**（与用户的交互中不需要重新刷新页面，并且数据的获取也是异步执行的）
      ii. **服务器压力小**
      iii. **前后端分离开发**
   b. 缺点
      i. **初次加载耗时增加**（SPA 是通过 JS 动态改变 HTML 内容实现的，页面本身的 URL 没有改变）
      ii. **需要自行实现导航**（SPA 无法记住用户的操作记录，刷新、前进、后退存在问题）
      iii. **SEO 不友好**（只有一个 URL）

理解路由

1. 路由：一个路由就是一个映射关系 `key-value`，其中 `key` 为路径，`value` 为 `function` 或 `component`。
2. 路由的分类
   a. 后端路由：`value` 是 `function`，通过 `router.<method>(path, function(req, res)=>{xxx})` 注册路由，用于处理客户端提交的请求。

   > node 收到请求 → 根据请求路径找到匹配的路由 → 调用路由的回调函数来处理请求 → 返回响应数据

b. 前端路由：`value` 是 `component`，通过 `<Router path="路径" component={组件}>` 注册路由，用于展示页面内容。

> 浏览器的 `path` 改变 → 路由组件变为对应的 `path` 的组件

前端路由的实现原理

1. 解释：前端路由的实现原理简单来说，就是**在不跳转或者刷新页面的前提下，为 SPA 应用中的每个视图匹配一个特殊的 URL**，之后的刷新、前进、后退等操作均通过这个特殊的 URL 实现。为实现上述要求，需要满足：
   - 改变 URL 且**不会向服务器发起请求**；
   - 可以**监听**到 URL 的变化，并**渲染**与之匹配的视图。
2. 实现方式：主要有 **Hash 路由**和 **History 路由**两种实现方式。

Hash 路由 —— location

1. Hash：即 URL 中 **# 号及其后面的字符。**由于 **URL 中 Hash 值的改变并不会向服务器发起请求**，并且我们也可以**通过 `hashchange` 事件对其改变进行监听。**因此我们就可以**通过改变页面的 Hash 来实现不同视图的匹配与切换。**
2. 相关 API
   a. **设置 Hash** `window.location.hash = 'xxxx';`
   b. **获取 Hash** `let hash = window.location.hash;`
   c. **监听 Hash** `window.addEventListener('hashchange', function(event){xxx}, false);`
3. 路由类（路由器）的设计：**通过键值对的形式保存路由及对应要执行的回调函数，当监听到页面 Hash 发生改变时，根据最新的 Hash 值调用注册好的回调函数，即改变页面。**

```
class Routers{
    constructor(){
        this.routes = {}; // key-value 的形式存储路由
        this.currentUrl = ''; // 当前页面的 Hash 值
        /* load 事件在整个页面及所有依赖资源如样式表和图片都已完成加载时触发。 */
        window.addEventListener('load', this.refresh, false);
        /* hashchange 事件在 URL 的 Hash 值变化时触发。 */
        window.addEventListener('hashchange', this.refresh, false);
    }

    route = (path, callback) => {
        this.routes[path] = callback || function(){}; // 注册路由
    }

```

```
    refresh = () => {
        this.currentUrl = location.hash.slice(1) || '/'; // 获取 Hash 值 - key
        this.routes[this.currentUrl](); // 指定执行的回调函数，渲染页面 - value
    }
}

window.Router = new Routers();
```

4. 路由的注册示例

```
var content = document.querySelector('body');

function changeBgColor(color){
    content.style.background = color;
}

// 添加路由
Router.route('/', () => {
    changeBgColor('yellow');
});
Router.route('/red', () => {
    changeBgColor('red');
});
Router.route('/green', () => {
    changeBgColor('green');
});
Router.route('/blue', () => {
    changeBgColor('blue');
});
```

History 路由 —— history

1. History：即文档的浏览历史，`history` 对象。当 `history` 对象出现变化时，就会触发 `popstate` 事件。调用 `history.pushState` 或者 `history.replaceState` 并不会触发该事件，只有用户点击浏览器倒退按钮和前进按钮，或者调用 `history.back`、`history.forward`、`history.go` 方法时才会触发。也就是说 **`history.pushState()` 和 `history.replaceState()` 都具有在改变页面URL的同时，不刷新页面的能力**，因此也可以用来实现前端路由。

2. 相关 API（H5 新增）

   a. **添加新状态到历史状态栈** `history.pushState(state, title, url);`

b. **用新状态替换当前状态** `history.replaceState(state, title, url);`

c. **获取当前状态** `history.state;`

> 参数解释
>
> - state 表示一个与特定 URL 相关的状态对象，当 popstate 事件触发时，该对象会被传入回调函数。如不使用，可填 null。
> - title 表示新页面标题，被浏览器忽略，可填 null。
> - url 表示新的网址，必须与当前页面处于同一域。

3. 路由类（路由器）的设计

```
class Routers{
  constructor(){
    this.routes = {};
    window.addEventListener('popstate', e => { // 表示网页 URL 变化时
      const path = e.state && e.state.path; // 获取请求路径 - key
      this.routes[path] && this.routes[path](); // 执行响应的回调 - value
    })
  }

  init(path){ // 初始情况执行响应的回调
    history.replaceState({path: path}, null, path);
    this.routes[path] && this.routes[path]();
  }

  route(path, callback){
    this.routes[path] = callback || function(){}; // 注册路由
  }

  go(path){ // 用于解决超链接的路由跳转
    history.pushState({path: path}, null, path);
    this.routes[path] && this.routes[path]();
  }
}

window.Router = new Routers();
```

4. 路由的注册示例

```
function changeBgColor(color){
  content.style.background = color;
}

```

```
Router.route(location.pathname, () => {
  changeBgColor('yellow');
});
Router.route('/red', () => {
  changeBgColor('red');
});
Router.route('/green', () => {
  changeBgColor('green');
});
Router.route('/blue', () => {
  changeBgColor('blue');
});

const content = document.querySelector('body');
Router.init(location.pathname);
```

5. History 路由的几点注意事项
   a. 路由触发的解决方案分析
      i. 点击浏览器的前进或者后退按钮：监听 `popstate` 事件，获取相应路径并执行回调函数
      ii. 点击 `<a/>` 标签：阻止其默认行为，获取其 `href` 属性，手动调用 `history.pushState()`，并执行相应回调。

```
const ul = document.querySelector('ul');

ul.addEventListener('click', e => {
  if(e.target.tagName.toUpperCase() === 'A'){
    e.preventDefault(); // 阻止默认行为
    Router.go(e.target.getAttribute('href')); // 执行路由回调
  }
})
```

   b. 通过 `history.pushState()` 会**显示“假”的 URL**，同时 history 状态栈会进行改变。
   c. **倘若手动刷新，或输入 URL 直接进入页面的时候，服务端是无法识别这个 URL 的**。所以，如果要应用 History 路由，需要**在服务端增加一个覆盖所有情况的候选资源**，当 URL 匹配不到任何静态资源，则应该返回单页应用的 HTML 文件。

Hash Vs. History

- Hash
  - 优点：兼容性好，无需服务器配合

- 缺点：丑陋、导致锚点功能失效

- History
  - 优点：!(Hash 缺点)
  - 缺点：!(Hash 优点)

react-router-dom@5

1. 解释：`react-router` 用于实现 React 中的前端路由功能，共有三个类别，分别适用于 web、native 和 core。`react-router-dom` 则是适用于 web 的 React 前端路由库。

   > `react-router` 路由核心库，提供许多组件、钩子。
   >
   > `react-router-dom` 包含 react-router 所有内容，以及专门用于 DOM 的组件。
   >
   > `react-router-native` 包含 react-router 所有内容，以及专门用于 ReactNative 的 API。

2. 安装：`yarn add react-router-dom@5`

基本使用 ⭐

`BrowserRouter`、`HashRouter`、`Link`、`Route`

1. 路由跳转的抽象：① **改变 URL 中的路径** ② **监听 URL 中的路径变化，并根据路径渲染不同的路由组件（视图）**。
2. 关键名词：**路由器**（router）和**路由**（route）。**一个路由器可以管理许多路由**，并对 URL 中的路径变化进行监听，以根据路径渲染不同的路由组件（视图）。
3. 相关内置组件
   a. **路由器** `<BrowserRouter></BrowserRouter>` 或 `<HashRouter></HashRouter>`。分别表示两种不同的前端路由器。
   b. **路由切换** `<Link to="路径"></Link>`。`react-router-dom` 最后会把 `Link` 标签变为 `a` 标签。
   c. **路由注册** `<Route path="路径" component="路由组件名"></Route>`。

   > `Link` 和 `Route` 组件必须包裹在 `BrowserRouter` 或 `HashRouter` 中。这样，路由器可以管理所有通过 `Route` 注册的路由。当使用 `Link` 改变 URL 中的路径时，路由器会根据 URL 中的路径渲染与之匹配的 `Route` 组件（视图）。
   >
   > 一般的，会使用 `BrowserRouter` 或 `HashRouter` 包裹 `App` 组件。

一般组件 Vs. 路由组件

区别	一般组件	路由组件
语法	`<Demo />`	

		`<Route path="/demo" component={Demo}>`
存放位置	`src/components`	`src/pages`
props	传入什么，收到什么	`history`、`location`、`match`

> 关于路由组件的 `props` 的进一步说明
>
> - `history` 常用 API 有：`go(n)`、`goBack()`、`goForward()`、`push(path, state)`、`replace(path, state)`
> - `location` 常用 API 有：`pathname`、`search`、`state`
> - `match` 常用 API 有：`params`、`path`、`url`

链接高亮 ⭐

`NavLink`

- `NavLink` 是一个特殊的 `Link` 组件。**当用户点击该组件切换 URL 路径时，我们称该 `NavLink` 是活跃的（active），此时 `NavLink` 会默认添加一个 `active` 类名。**
- 也可以通过 `activeClassName="类名"` 的方式指定 `NavLink` 活跃时添加的类名。
- 使用该标签，可以在路由切换时实现导航组件（即 `NavLink`）的高亮效果。

高效路由匹配 ⭐

`Switch`

- 通常情况下，渲染页面时**会对所有注册的路由进行逐个匹配**，只要匹配成功就会渲染相应路由组件。**通常情况下，`path` 和 `component` 是一一对应的**，因此可以**使用 `Switch` 进行注册路由的单一匹配，只要匹配成功，就停止 `Switch` 组件中其他路由的匹配**，从而提高路由匹配的效率。
- 未使用 `Switch` 组件 Vs. 使用 `Switch` 组件

```
<!-- 对于 /home 路径，页面会渲染 Home 和 Test 组件 -->
<Route path="/about" component={About} />
<Route path="/home" component={Home} />
<Route path="/home" component={Test} />
```

```
<!-- 对于 /home 路径，页面只会渲染 Home 组件（只要匹配一个组件，就停止匹配其他组件）
-->
<Switch>
    <Route path="/about" component={About} />
```

```
    <Route path="/home" component={Home} />
    <Route path="/home" component={Test} />
</Switch>
```

样式丢失问题

- 问题的引入：假设在 `public/index.html` 中通过 `./` 的方式引入样式文件，那么当 URL 不为 `/xxx` 的时候，刷新网页会导致样式的丢失。此时，因为请求的路径

```
<link rel="stylesheet" href="./bootstrap.css">
```

- 问题的解决
  - 方式一：`<link rel="stylesheet" href="/bootstrap.css">`
  - 方式二：`<link rel="stylesheet" href="%PUBLIC_URL%/bootstrap.css">`
  - 方式三：使用 HashRouter，此时路由的切换不会导致 URL 路径的改变。

模糊匹配与严格匹配 ⭐

- 模糊匹配：`<NavLink>` 的 `to` 指定的路径包含 `<Route>` 的 `path` 指定的路径时，会匹配到该路由。
- 严格匹配：`<NavLink>` 的 `to` 指定的路径严格相等 `<Route>` 的 `path` 指定的路径时，会匹配到该路由。
- **开启严格匹配**：`<Route exact path="xxx" component={Xxx}>`

> **不要随便开启严格匹配**，可能会导致二级路由的失效！！！

重定向 ⭐

`Redirect`

- 在进行路由匹配时，如果无法找到匹配的路由，则可以通过 `Redirect` 组件跳转到指定的路由。一般来说，`Redirect` 组件会写在所有 `Route` 组件的最下方。

```
<!-- to 指定要跳转到的路由 -->
<Redirect to="xxx"/>
```

- 示例：`Redirect` + `Route` + `Switch`

```
<Switch>
```

```
    <Route path="/about" component={About} />
    <Route path="/home" component={Home} />
    <Redirect to="/home"/>
</Switch>
```

嵌套路由 ⭐

- 假设 `/home` 路由对应的 `Home` 组件中包含其他路由组件，则称之为子路由。**注册子路由需要写上父路由的 `path` 值**。

```
<!-- Home 组件，假设其有两个子路由 /news 和 /message -->
<!-- 路由切换 -->
<NavLink to="/home/news">News</NavLink>
<NavLink to="/home/message">Message</NavLink>
<!-- 路由注册 -->
<Route path="/home/news" component={News} />
<Route path="/home/message" component={Message} />
```

- 在 React 中路由的注册是有顺序的，当通过 `NavLink` 或 `Link` 实现路由跳转后，React 会**根据路由注册的顺序进行匹配**。假设 App.js 和 Home.js 注册的路由如下，

```
<!-- App.js -->
<Switch>
    <Route path="/about" component={About} />
    <Route path="/home" component={Home} />
    <Redirect to="/home"/>
</Switch>
<!-- Home.js -->
<Switch>
    <Route path="/home/news" component={News} />
    <Route path="/home/message" component={Message} />
</Switch>
```

当路由切换为 `/home/messgae` 时，React 会匹配上 `Home` 和 `Message` 路由组件。

路由参数

params

- 传递：`<Link to="/demo/test/tom/18">点我跳转 Test 组件，传递 params 参数</Link>`

- 声明：`<Route path="/demo/test/:name/:age" component={Test}>`
- 接收：`const {name, age} = this.props.match.params`

search

- 传递：`<Link to="/demo/test/?name=tom&age=18">点我跳转 Test 组件，传递 search 参数</Link>`
- 声明：`<Route path="/demo/test" component={Test}>`
- 接收：`const {name, age} = se(this.props.location.search.slice(1))`

> `search` 参数**无需声明接收**，正常注册路由即可。
>
> `this.props.location.search` 是 urlencoded 编码字符串，需要借助 `querystring` 库解析。`slice(1)` 用于删除编码字符串的第一个字符 `?`。
>
> 也可以使用浏览器自带的 API `URLSearchParams` 处理编码字符串。

state

- 传递：`<Link to={{pathname: "/demo/test", state:{name: "tom", age: 18}}}>点我跳转 Test 组件，传递 params 参数</Link>`
- 声明：`<Route path="/demo/test" component={Test}>`
- 接收：`const {name, age} = this.props.location.state`

> `state` 参数**无需声明接收**，正常注册路由即可。
>
> 使用 BrowserRouter 时，由于 `state` 借助浏览器的 `history` 保存，页面的刷新不会使得 `state` 参数丢失；但使用 HashRouter 时，页面的刷新会导致 `state` 参数的丢失。而 `params` 和 `search` 则不存在刷新参数丢失的情况。

路由跳转的两种模式

- `push` 模式：默认模式，表示将对应路由记录压入历史记录栈。
- `replace` 模式：表示将对应路有记录替换历史记录栈顶的历史记录。通过布尔属性 `replace` 可以开启路由跳转的 `replace` 模式，如 `<Route replace path="/demo/test" component={Test}>`
- **关于历史记录栈**：当 URL 改变时，浏览器会对历史 URL 进行记录，其所使用的数据结构是栈。默认情况下，每次改变 URL 时，浏览器会将新的 URL 压入栈中，这样就可以使用 `back` 和 `go` 等 API 实现历史记录的前进和后退。但是，当开启 `replace` 模式时，浏览器不会简单地将新的 URL 压入栈中，而是将栈顶的 URL 替换为最新的 URL，这样被替换的 URL 将无法通过历史记录访问到。

编程式路由导航

所谓的编程式路由导航，就是指路由组件**借助 `this.props.history` 对象上的 API 来操作路由**。

- `go(n)`：前进或后退 `n` 个历史记录，正数前进，负数后退。
- `goBack()`：后退一个历史记录。
- `goForward()`：前进一个历史记录。
- `push(path, state)`：`push` 模式跳转到 `path` 路由。其中可以通过 `path` 传递 `params` 参数或 `search` 参数；`state` 参数需要作为 `push` 方法的第二个参数进行传递。
- `replace(path, state)`：`replace` 模式跳转到 `path` 路由。其中可以通过 `path` 传递 `params` 参数或 `search` 参数；`state` 参数需要作为 `push` 方法的第二个参数进行传递。

一般组件路由导航

- 引入：对于路由组件，可以通过 `this.props.history` 上的 API 实现编程式路由导航，但是一般组件并没有接受到 `history` 参数，因此无法实现编程式路由导航。可以使用 `react-router-dom` 提供的 **`WithRouter` 函数对一般组件进行加工，使其具有 `history` 参数，从而可以利用 `history` 的 API 实现编程式路由导航**。
- 示例：`WithRouter(一般组件) = 具有导航功能的一般组件`

```
import React, {Component} from "React"
import {WithRouter} from "react-router-dom"

class Header extends Component {
    // ...
}

export default WithRouter(Header);
```

HashRouter Vs. BrowserRouter

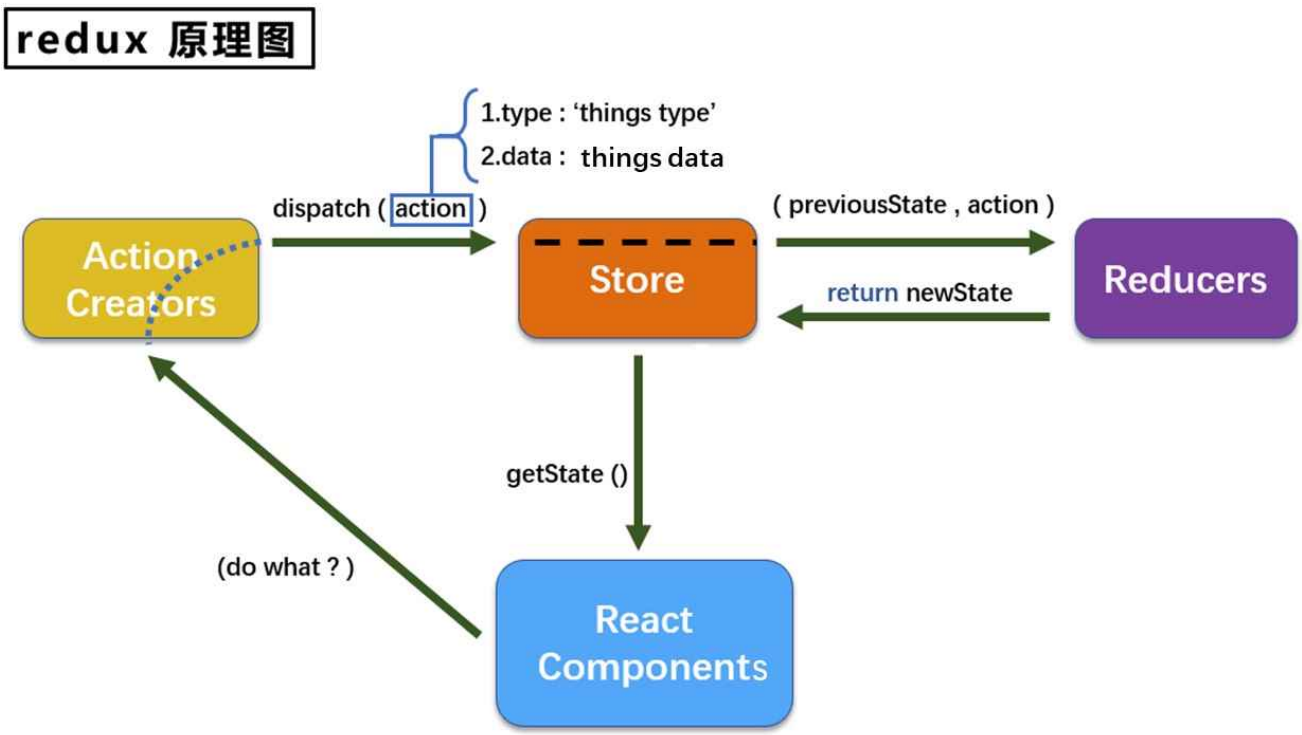
区别	HashRouter	BrowserRouter
底层实现	H5 新增的 history API，不兼容 IE9 及以下版本	URL 的哈希值
URL 形式	没有 #	有 #
刷新对 state 参数的影响	无任何影响	会导致 state 参数的丢失

Redux

Redux 概述

- Redux 是一个专门用于做**状态管理**的 JavaScript 库，可以在 React、Angular、Vue 等项目中使用，一般只与 React 配合使用。通过 `yarn add redux` 安装。
- Redux 集中式管理 React 应用中**多个组件共享的状态**。
- Redux 的使用原则是：能不用就不用！！！

Redux 原理图



action

表示**动作**的**一般对象**，包含两个属性，

- `type`：标识属性，值为字符串，唯一，必要
- `data`：数据属性，值类型任意，可选

reducer

用于**初始化状态和加工状态**的**纯函数**，加工时根据旧 `state` 和 `action` 产生新的 `state`。`store` 会触发 `reducer` 的第一次调用，用于初始化 `state`，其接受到的参数为 `prevState=undefined, action={type: "@@redux/INIT"}`。

store

将 `state`、`action`、`reducer` 联系在一起的对象，通过以下方法得到 `store` 对象，

```
1 import {createStore} from "redux"
```

```
import reducer from "./reducer"
const store = ceateStore(reducer)
```

使用说明

1. 为了方便 `action` 生成，一般会提供 **`actionCreator`** 用于生成相应的 `action`，此时在组件中只需要 `dispatch + actionCreator + data` 即可实现 `action` 的分发。

```
// countAction.js
/* 专为 Count 组件生成 Action 对象 */
import { INCREMENT, DECREMENT } from "./CONSTANT"

export const createIncrementAction = data => ({ type: INCREMENT, data });
export const createDecrementAction = data => ({ type: DECREMENT, data });
```

2. 为了避免 `type` 写错，一般会创建对应的常量，建立 **变量-类型** 映射，此时在 `actionCreator` 或 `reducer` 中只需要引入这些变量即可。

```
// CONSTANT.js
/* 将 Action 对应的 type 保存为常量，避免写错  */
export const INCREMENT = "increment";
export const DECREMENT = "decrement";
```

3. `store` 对象需要**使用指定的 reducer 创建，同时也可以引入中间件**，扩展 `store` 的功能。

```
// store.js
/* 创建了一个 store 对象 */
import { legacy_createStore as createStore, applyMiddleware } from 'redux'
import {thunk} from "redux-thunk"
import countReducer from "./countReducer"

export default createStore(countReducer, applyMiddleware(thunk));
```

4. `reducer` 本质是一个函数，**接收旧 `state` 和 `action`，返回一个新 `state`**。

```
// countReducer.js
/* 定义了一个为 Count 组件服务的 reducer 函数 */
import { INCREMENT, DECREMENT } from "./CONSTANT"
const initState = 0; // 初始值
```

```
function countReducer(prevState = initState, action) {
    const { type, data } = action;
    switch (type) {
        case INCREMENT:
            return prevState + data;
        case DECREMENT:
            return prevState - data;
        default:
            return prevState;
    }
}

export default countReducer;
```

5. 由于 `store` 中 `state` 的更新不会触发页面的渲染，因此可以在 `index.js` 中使用 `store.subscribe`，使得每次 **`state` 的更新都会触发组件的重新渲染**。

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import store from './store';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
/* 每次 store 中状态更新时，会重新从根逐渐开始进行渲染。由于 React-Diffing，不会引起浏览器的大量重绘重排。 */
store.subscribe(
  () => {
    root.render(
      <React.StrictMode>
        <App />
      </React.StrictMode>
    );
  }
)

```

核心 API

- `createStore(reducer)` 创建包含指定 `reducer` 的 `store` 对象。

- `applyMiddleWare(middleWare)` 应用指定 `middleWare` 中间件。
- `combineReducers({"stateName1": reducer1, "stateName2": reducer2})` 合并多个 `reducer` 函数。
- `store`
  - `.getState()` 得到 `state`。
  - `.dispatch(action)` 分发 `action`，触发 `reducer` 调用，产生新的 `state`。
  - `.subscribe(listener)` 注册监听，当产生新的 `state` 时自动调用。

异步 action

- 当使用 `store.dispatch` 发送一个 action 后，store 会立即调用对应的 reducer 处理旧的 state，并生成新的 state，这个过程是同步的。如果希望 state 异步更新，可以使用异步 action。
- **同步 action 是普通的 JavaScript 对象 `{}`，而异步 action 是一个函数**。异步 action 函数返回一个接收 `dispatch` 作为参数的函数，可以在其中分发 action，并执行异步任务。
- Redux 默认无法解析函数类型的 action，需要安装并使用 `redux-thunk` 中间件。
  - 安装：`yarn add redux-thunk`
  - 使用

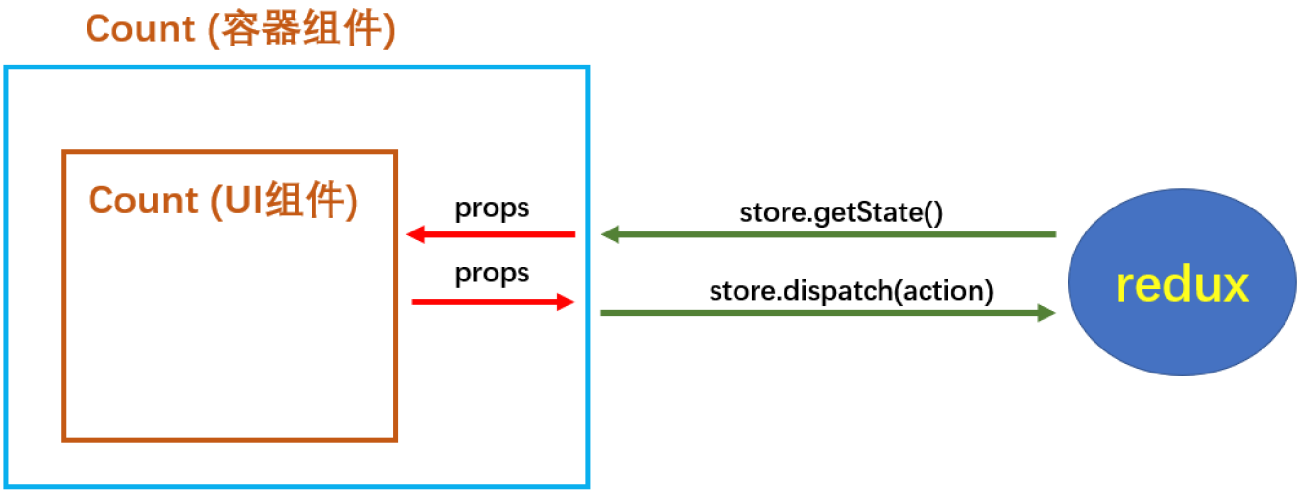
```
1 import { legacy_createStore as createStore, applyMiddleware } from
  'redux'
2 import {thunk} from "redux-thunk"
3 import countReducer from "./countReducer"
4
5 export default createStore(countReducer, applyMiddleware(thunk));
```

react-redux

概述

- react-redux 是一个 react 插件库，用于简化 redux 在 react 应用中的使用。
- 安装：`yarn add react-redux`

原理图



核心思想

react-redux 将所有组件分为 **UI 组件**和**容器组件**两类。 UI 组件包裹在容器组件中，二者之间是**父子关系**。

- UI 组件：负责 UI 呈现；通过 props 接收容器组件传递的数据（Redux 所保存的状态）和函数（用于操作状态的方法）；一般保存在 components 文件夹下。
- 容器组件：负责业务逻辑；直接与 Redux 打交道，使用 Redux 的 API；一般保存在 containers 文件夹下。

> react-redux 可以实现容器组件对状态改变的自动监测，此时无需再手动监测 redux 状态的更新。

基本使用

- 容器组件的**创建**：通过 react-redux 的 `connect` 方法创建容器组件，并将容器组件和 UI 组件连接起来。通过两个函数参数 `mapStateToProps` 、 `mapDispatchToProps` 以 props 的方式给 UI 组件传递状态数据和改变状态数据的方法。

  > 这里不关注 UI 组件，因为 UI 组件直接通过 props 使用容器组件传递过来的数据和方法即可。

```
/* 定义 Count 的容器组件 */
import { connect } from "react-redux" // 引入 connect 函数，创建容器组件，用于连接 UI 组件和 redux
import CountUI from "../../components/Count"; // 引入 Count 的 UI 组件
import { // 引入 actionCreators
    createIncrementAction,
    createDecrementAction,
    createIncrementAsyncAction
} from "../../store/countAction"

/*  connect 函数的第一个参数，用于将 Redux 的 store 维护的 state 映射为 UI 组件的数据 props。
    传递状态。 */
function mapStateToProps(state) {
```

```
    return { count: state };
}

/*  connect 函数的第二个参数，用于将 Redux 的 store 的 dispatch 映射为 UI 组件的方法 props。
    传递操作状态的方法。 */
function mapDispatchToProps(dispatch) {
    return {
        increment: number => dispatch(createIncrementAction(number)),
        decrement: number => dispatch(createDecrementAction(number)),
        incrementAsync: (number, time) => dispatch(createIncrementAsyncAction(number, time))
    }
}

/* 创建并暴露 Count 的容器组件 */
const CountContainer = connect(mapStateToProps, mapDispatchToProps)(CountUI);
export default CountContainer;
```

- 容器组件的**使用**：以标签的形式使用组件，同时通过 `store` 属性传递 `store` 对象，从而建立容器组件和 Redux 之间的关联。

```
import React, { Component } from 'react'
import Count from "./containers/Count"
import store from './store'

export default class App extends Component {
  render() {
    return <Count store={store} />
  }
}
```

使用优化

1. **`mapStateToProps` 的简写形式**，即**对象写法**，对象中的每个 key 是要传递给 UI 组件的方法名，其对应的 value 是一个函数，其返回值是相应的 action。此时，**当在 UI 组件中调用方法时，会创建相应的 action，此时 react-redux 会自动 dispatch 这个 action，进一步实现状态数据的更新**。

```
import { // 引入 actionCreators
    createIncrementAction,
    createDecrementAction,
```

```
    createIncrementAsyncAction
} from "../../store/countAction"

// mapStateToProps 的完整写法，key 是 props 传递给 UI 组件的方法名，value 是实际执行的函数，会 dispatch 一个 action
function mapDispatchToProps(dispatch) {
    return {
        increment: number => dispatch(createIncrementAction(number)),
        decrement: number => dispatch(createDecrementAction(number)),
        incrementAsync: (number, time) => dispatch(createIncrementAsyncAction(number, time))
    }
}

// mapStateToProps 的简化写法，key 是 props 传递给 UI 组件的方法名，value 是实际执行的函数，会创建一个 action，react-redux 会自动 dispatch 这个 action
const mapDispatchToProps = {
    increment: createIncrementAction,
    decrement: createDecrementAction,
    incrementAsync: createIncrementAsyncAction
    }
}
```

2. **store 的批量传递**，即使用 react-redux 提供的 `Provider` 组件包裹 App 组件，并通过 store 字段传递 store，那么 **App 组件中的所有容器组件将会自动获取 store**，而不再需要手动为每个容器组件提供 store。

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { Provider } from 'react-redux'
import store from './store';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
)
```

3. **容器组件和 UI 组件的合并**，即将 UI 组件定义在容器组件所在文件中，只向外暴露容器组件。

核心 API

- `<Provider store={store}>{/* 需要接收 store 的容器组件 */}</Provider>` 向所有容器组件批量传递 store。
- `connect(mapStateToProps, mapDispatchToProps)(UIComponentName)` 包装 UI 组件生成容器组件。
  - `mapStateToProps(state)`：函数，将 store 管理的 state 映射为 UI 组件的标签属性，表示状态数据
  - `mapDispatchToProps(dispatch)`：函数，根据 store.dispatch 映射为 UI 组件的标签属性，表示操作状态数据的方法

Redux 管理多个 state

假设 Redux 管理 Student 和 Subject 两个状态，因此也应该有相应的 StudentReducer 和 SubjectReducer、StudentActionCreator 和 SubjectActionCreator。下面从文件组织和子 Reducer 合并的角度解释 Redux 如何管理多个 state，也就是说一个 store 如何管理多个 state。

1. 文件组织

```
src
├── store # Redux 相关内容
│   ├── actions # 定义了不同 State 的 ActionCreator
│   │   ├── studentAction.js
│   │   └── subjectAction.js
│   ├── reducers # 定义了不同 State 的 Reducer
│   │   ├── studentReducer.js
│   │   └── subjectReducer.js
│   ├── CONSTANT.js # 定义了所有 State 的 type 的常量映射
│   └── index.js # 合并所有子 Reducer，并创建 store
├── containers # 容器组件，每个 index.js 定义了容器组件和 UI 组件，暴露容器组件
│   ├── Student
│   │   └── index.js
│   ├── Subject
│   │   └── index.js
│   └── index.js
├── index.js # 入口文件，通过 Provider 给所有容器组件提供 store
└── App.js # 根组件

```

2. Reducer 的合并：Redux 中的 `combineReducers` 函数用于将多个子 reducer 合并成一个根 reducer。每个子 reducer 管理不同部分的 state，因此 `combineReducers` 需要一个对象作为参数，这个对象的键（key）用来标识 state 的不同部分，对应的值（value）是具体的子 reducer。最终，根 reducer 管理的 state 是一个对象，可以通过传入的键值来区分不同的子 state。

```
import { combineReducers, createStore } from 'redux';

// 定义子 reducer
const studentReducer = (state = {}, action) => {
  switch (action.type) {
    case 'UPDATE_STUDENT':
      return { ...state, student: action.payload };
    default:
      return state;
  }
};

const subjectReducer = (state = {}, action) => {
  switch (action.type) {
    case 'UPDATE_SUBJECT':
      return { ...state, subject: action.payload };
    default:
      return state;
  }
};

// 使用 combineReducers 合并子 reducer
const rootReducer = combineReducers({
  student: studentReducer,
  subject: subjectReducer,
});

// 创建 Redux store
const store = createStore(rootReducer);

// 输出初始状态
console.log(store.getState()); // { student: {}, subject: {} }

// 分发 action 更新状态
store.dispatch({ type: 'UPDATE_STUDENT', payload: { name: 'Alice' } });
store.dispatch({ type: 'UPDATE_SUBJECT', payload: { name: 'Math' } });

// 输出更新后的状态
console.log(store.getState()); // { student: { student: { name: 'Alice' } }, subject: { subject: { name: 'Math' } } }
```

纯函数和高阶函数

1. 纯函数：满足**同样的输入，必定得到同样的输出**。
   a. 纯函数需要遵守 ①不得改写参数数据 ②不得产生任何副作用，如网络请求、IO ③ 不能调用 `Date.now()` 或 `Math.random()` 等不纯的方法
   b. Redux 的 Reducer 函数必须是一个纯函数，即不得改写 prevState，同时返回一个全新的 newState，Redux 会对此进行一个引用层次的浅比较。
2. 高阶函数：满足**参数或返回值是函数**。

Redux 开发者工具

Step 1 Chrome 浏览器扩展：**Redux DevTools**

Step 2 npm 依赖包：`yarn add redux-devtools-extension`

Step 3 在项目的创建 store 的文件中添加以下代码

```
import {composeWithDevTools} from "redux-devtools-extension"
// ...
const store = createStore(rootReducer, composeWithDevTools()) // 没有使用中间件时
// or
const store = createStore(rootReducer,
composeWithDevTools(applyMiddleWare(thunk))) // 使用中间件时
```

此时在浏览器 F12 控制台出现一个新的选项卡 Redux，用于查看 state、action 等信息。

项目打包

Step 1 `yarn build` 将源代码打包到 `build` 文件夹中

Step 2 `serve buid` 将 `build` 文件夹下的内容部署在本地服务器查看

> serve 是一个工具，将指定目录下的文件部署在本地的服务器上，通过 `yarn add serve -g` 全局安装

React 补充

setState 的两种写法

1. **对象式** `setState(stateChange, [callback])`
   a. `stateChange` 为状态改变对象，用于体现状态的更改
   b. `callback` 是可选的回调函数, 它在状态更新完毕、界面也更新后（render 调用后）才被调

2. **函数式** `setState(updater, [callback])`
   a. `updater` 为返回 stateChange 对象的函数，其类型为 `(state, props) => {}`
   b. `callback` 是可选的回调函数, 它在状态更新完毕、界面也更新后（render 调用后）才被调用
3. 使用说明
   a. 对象式的 setState 是函数式的 setState 的简写形式
   b. 新状态不依赖于原状态时，使用对象式；新状态依赖于原状态时，使用函数式
   c. 如果需要在 setState 执行后获取最新的状态数据，则需要在第二个 callback 函数中读取
   d. **React 中的状态更新是异步的**，其中 setState 是同步方法，但是其引起的 React 对状态的更新是异步的

lazyLoad 懒加载

1. **组件懒加载** `lazy(() => import("组件地址"))`

   > 懒加载又称为动态加载，常用于路由组件，此时路由组件的代码会被分开打包

```
import {lazy} from "react"
const Home = lazy(() => import("./Home")); // 此时该组件没有被加载，使用该组件时加载再开始加载
```

2. **加载中界面** `<Suspense fallback={JSXNode}>注册的路由</Suspense>`

   > 通过 Suspense 组件可以指定通过懒加载得到路由组件前显示的自定义的 Loading 界面，该组件包裹需要懒加载的路由组件，即包裹 Route 组件。

```
import {Suspense} from "react"
<Suspense fallback={<h1>Loading...</h1>}>
    <Switch>
        <Route path="/home" component={Home}/>
        <Redirect to="/home"/>
    </Switch>
</Suspense>
```

3. 使用说明： `lazy` 和 `Suspense` 必须配合使用。

Hooks for RFC

Hook 是 React 16.8.0 版本增加的新特性/新语法，允许在函数组件中使用 state 以及其他的 React 特性。

State Hook

1. 语法 `const [xxx, setXxx] = React.useState(initValue)`
   a. `initValue` 第一次初始化指定的值，会在内部作缓存
   b. `[xxx, setXxx]` 包含 2 个元素的数组, 第 1 个为内部当前状态值, 第 2 个为更新状态值的函数
2. 功能：让函数组件也可以**有 state 状态**, 并进行**状态数据的读写**操作。与类组件不同，函数式组件的 `setXxx` 更新状态是**完全覆盖**的，即 `setXxx({})` 会使状态 `xxx={}`。
3. 状态更新函数 `setXxx()` 的两种写法
   a. **非函数式** `setXxx(newValue)`，参数为非函数值, 直接指定新的状态值, 内部用其覆盖原来的状态值
   b. **函数式** `setXxx(value => newValue)`，参数为函数, 接收原本的状态值, 返回新的状态值, 内部用其覆盖原来的状态值

Effect Hook

1. 语法

```
1 React.useEffect(() => {
2   // 在此可以执行任何带副作用操作
3   return () => { // 在组件卸载前执行
4     // 在此做一些收尾工作, 比如清除定时器/取消订阅等
5   }
6 }, [stateValue]) // 如果指定的是 [], 回调函数只会在第一次 render() 后执行
```

> **useEffect 接收两个参数：**
>
> 1. **副作用函数：** 第一个参数是一个函数，用于执行副作用。
> 2. **依赖项数组：** 第二个参数是一个数组，定义副作用函数的依赖。
>
> **依赖项数组的行为：**
>
> - **不传递依赖项数组：** useEffect 依赖所有的 state 和 props，只要有任何变化，副作用函数都会执行。
> - **依赖项数组为空 []：** useEffect 不依赖任何 state 或 props，副作用函数只在组件首次渲染后执行一次。
> - **包含特定依赖项：** useEffect 仅依赖数组中的 state 或 props，当这些依赖项发生变化时，副作用函数才会执行。
>
> **清理副作用：**

> - **清理函数：** 副作用函数可以返回一个函数，用于清理副作用。当组件卸载或依赖项发生变化时，清理函数会自动执行。

2. 功能：允许在函数组件中执行**副作用**操作，用于模拟类组件中的**生命周期钩子**

> useEffect 可以看做是 `componentDidMount`、`componentDidUpdate`、`componentWillUnmount` 的结合

3. React 副作用：指任何在组件渲染过程中需要执行的、与渲染无关的操作。副作用可能包括但不限于以下几种操作：ajax 请求数据、设置订阅 / 启动定时器、手动更改真实 DOM 等

Ref Hook

1. 语法

```
const refContainer = useRef()
return <input ref={refContianer}></input>
/* 使用 ref={refContianer} 后，可以通过 refContainer.current 的方式操作 input DOM  */
```

2. 功能：允许在函数组件中**存储/查找组件内的标签或任意其它数据，**功能与 `React.createRef()`

Fragment 根标签

`<Fragment></Fragment>` 作为**组件根标签**时，**不会被渲染为真实 DOM**。该标签只接受 key 属性，其简写形式为 `<></>`，此时不允许携带任何属性。

Context 通信

1. 介绍：`React.createContext()` 提供一个**上下文对象**，可以用于**祖先组件和后代组件间通信**。
2. 使用方式
   a. **创建 context 容器** `const XxxContext = React.createContext()`
   b. **祖先组件传递数据**：使用 `XxxContext.Provider` 包裹子组件，并通过 `value` 属性给后代组件传递数据

```
<XxxContext.Provider value={数据}>
  <!-- 子组件 -->
</XxxContext.Provider>
```

   c. **后代组件声明接收**

i. 方式一：使用静态属性声明接收，仅适用于**类组件**

```
static contextType = XxxContext // 声明接收 context
this.context // 读取 context 中的 value 数据
```

ii. 方式二：使用 `XxxContext.Consumer` 声明接收，适用于**函数组件和类组件**

```
<XxxContext.Consumer> // 声明接收 context
    {
        value => ( // 读取 context 中的 value 数据
            // 使用 value 渲染组件
        )
    }
</XxxContext.Consumer>
```

iii. 方式三：使用 `React.useContext` 声明接收，适用于**函数组件**

```
const value = React.useContext(XxxContext); // 声明接收 context 并读取其中的 value 数据
```

PureComponent 性能提升

1. 默认 Component 存在的问题
   a. 执行 `setState({})`，即使**不改变状态**，组件也会调用 `render()` 重新渲染
   b. 当父组件调用 `render()` 重新渲染时，即使**传递给子组件的数据未变化**或**子组件没有使用父组件传递的数据**，子组件也会调用其 `render()` 重新渲染
2. 问题分析及效率提高策略
   a. 原因：生命周期函数 `shouldComponentUpdate()` 总返回 `true`
   b. 策略：只有当组件的 `state` 或 `props` 数据发生变化时，再允许调用 `render()` 重新渲染
3. 针对类组件的解决方式
   a. 方式一：**重写 `shouleComponentUpdate()` 方法**，比较新旧 `state` 或 `props`，有变化则返回 `true`，否则返回 `false`。

```
class Xxx extends React.Component{
    // ...
    shouldComponentUpdate(nextProps, nextState){
```

```
        // nextProps、nextState 表示接下来要变化的目标 props 和 state
        // this.props、this.state 表示当前的 props 和 state
        return this.state.Xxx !== nextState.Xxx || this.props.Xxx !==
  nextProps.Xxx;
    }
    // ...
}
```

b. 方式二：**继承 `React.PureComponent` 定义类组件，** `PureComponent` 中重写了 `shouldComponentUpdate()`，只有 `state` 或 `props` 数据有变化时才返回 `true`，否则返回 `false`。

> **PureComponent 使用注意事项：**
>
> - **浅比较：** PureComponent 通过浅比较（shallow comparison）来确定组件是否需要更新。这意味着它只比较对象的引用，而不比较对象内部的具体值。
> - **不可变数据：** 在更新 state 时，必须生成新的数据，而不是直接修改原有的 state 或其属性。
>
> **具体注意点：**
>
> 1. **新对象：** 更新 state 时，传入的新对象不能与当前 state 引用相同。
> 2. **新属性：** 传入的新对象中的非原始值属性（如对象或数组）不能与当前 state 的同名属性引用相同。

```
// 错误做法：直接修改 state
this.setState(prevState => {
  prevState.user.name = 'John'; // 直接修改原有 state
  return prevState;
});

// 正确做法：创建新对象
this.setState(prevState => ({
  user: {
    ...prevState.user,
    name: 'John' // 创建新对象
  }
}));
```

4. 针对函数式组件的类似解决方式

a. `React.memo` 是一个高阶组件，用于将函数组件包裹起来，**只有在 props 发生变化时才重新渲染**。它类似于 `PureComponent`，但用于函数组件。默认情况下，`React.memo` 会对 props 进行浅比较。如果需要自定义比较逻辑，可以传入一个比较函数作为第二个参数。

```jsx
const MyComponent = React.memo(function MyComponent(props) {
    return <div>{props.name}</div>;
});

const MyComponent = React.memo(function MyComponent(props) {
    return <div>{props.name}</div>;
}, (prevProps, nextProps) => {
    return prevProps.name === nextProps.name;
});
```

b. `useMemo` 钩子可以用来记住**计算结果，只有在依赖项发生变化时才重新计算**。

```jsx
const computedItems = useMemo(() => {
    return items.map(item => item * 2);
}, [items]);
```

c. `useCallback` 钩子可以用来记住**回调函数，只有在依赖项发生变化时才重新创建**。

```jsx
  const handleClick = useCallback(() => {
      console.log('Button clicked');
  }, []);
```

RenderProps 插槽技术

1. **Children Props**：即**组件标签体**的内容作为其 `children` 属性传递给组件。
2. **Render Props**：即组件通过 `render` 属性**传递一个返回组件标签的函数，函数可以携带数据，并将数据作为 `props` 传递给返回的组件标签**。使用 `render` 属性，可以实现向组件内部**动态**传入带数据的结构。

```jsx
// Parent 组件中
render(){
    /* render 属性的值是一个函数，根据传入的 data 数据动态创建 Grant 组件 */
    return <Children render={(data)=><Grant data={data}/>} />
}
// Children 组件中
state = {data: 0}
render(){
    return {this.props.render(data)}
}
```

```
// Grant 组件中
render(){
    const {data} = this.props;
    return <span>Grant 组件收到的 data = {data}</span>
}
```

ErrorBoundary 错误边界

1. **错误边界**：指的是在当前组件中捕**获后代组件错误，渲染备用页面**的机制。

   > 只能捕获**后代组件生命周期**产生的错误

2. 实现方式：`getDerivedStateFromError` + `componentDidCatch`

   a. `getDerivedStateFromError` 生命周期函数，一旦后代组件报错，就会触发该函数，同时**将错误对象传入其中，用于派生新的** `state` 。该生命周期函数是静态函数。

   > 之后在 render 函数中，可以根据 state 的值渲染不同的页面，如果标识错误，则渲染备用页面，没有错误才正常渲染。

   ```
   static getDerivedStateFromError(error){
       return newState;
   }
   ```

   b. `componentDidCatch` 生命周期函数，也是在后代组件报错时执行，常用于**统计页面错误，向后台发送请求**。

3. 注意事项

   a. 错误边界仅仅在**生产环境生效**，不适用于开发环境

   b. 错误边界的使用可以**避免错误的进一步扩散**

组件间通信方式

1. 通信方式：props、消息订阅与发布（如 pubsub）、集中式管理（如 redux）、context
2. 适用情况

   a. 父子组件 → props

   b. 兄弟组件 → 消息订阅与发布、集中式管理

   c. 祖孙组件 → 消息订阅与发布、集中式管理、context（开发用的少，多用于插件的封装）

React Router@6

基本使用

Step1. 使用 `BrowserRouter` 或 `HashRouter` 包裹 App 组件，将整个应用的路由纳入管理。

```
// index.js
import ReactDOM from "react-dom/client"
import {BrowserRouter} from "react-router-dom"
import App from "./App"

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
    <BrowserRouter>
        <App/>
    </BrowserRouter>
)
```

Step2. 使用 `Routes + Route` 注册路由，这里的 `Routes` 替换了 5 版本的 `Switch`。同时通过 `Route + Navigate` 重定向到指定路由。

```
<Routes>
    <Route path="/home" element={<About/>}>
    <Route path="/home" element={<Home/>}>
    <Route path="/" element={<Navigate to="/about"/>}/>
</Routes>
```

> **Routes**
>
> - `Routes` 是强制使用的，即 **`Route` 组件必须被 `Routes` 组件包裹**。
> - 与**废弃**的 `Switch` 类似，`Routes` 也是单一匹配的，即**只要匹任意一个路由成功，就停止匹配 `Routes` 中的其他路由**。
>
> **Route**
>
> - 与 5 版本不同，`Route` 组件**通过 `element` 指定路由组件**，同时传入的是路由组件标签。
> - `Route` 的 `caseSensitive` 布尔属性用于指定**路由匹配时是否区分大小写**。
>
> **Navigate**
>
> - `Navigate` 组件**一旦渲染，就会切换视图**。
> - `Navigate` 组件默认使用 `push` 跳转，可以通过 `replace` 布尔属性指定跳转模式为 `replace`。

Step3. 使用 `NavLink` 切换路由，通过给 className 传递一个函数，实现链接的高亮效果。

```
<NavLink className={({isActive})=> isActive ? "base active" : "base"}
to="\about">
    About
</Navilink>
<NavLink className={({isActive})=> isActive ? "base active" : "base"}
to="\home">
    Home
</Navilink>
```

> **NavLink**
>
> - 通过传入函数参数，可以**动态指定 `NavLink` 的 `className`**。
>   - 参数：对象，包含一个字段 `isActive`，当 `NavLink` 被点击时，取值为 `true`，否则取值为 `false`。
>   - 返回值：字符串，作为 `NavLink` 的 `className` 的值。
> - 如果不传入函数参数，则 React Router 会**自动给被点击的 `NavLink` 添加一个 `active` 类名**。

路由表

1. React Router 一级路由注册的两种方式
   a. `<Routes>` + `<Route>`

```
<Routes>
    <Route path="/home" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="/" element={<Navigate to="/home" />} />
</Routes>
```

   b. `useRoutes` + 路由表

> 路由表是一个包含路由匹配规则的数组，通过 `useRoutes` 可以使用路由表，它返回的是所有注册的路由元素。

```
const routes = [
    { path: "/home",element: <Home />},
    {path: "/about",element: <About />},
    {path: "/",element: <Navigate to="/home" />}
```

```
];

const element = useRoutes(routes);

return { element };
```

2. React Router 嵌套路由注册的两种等价方式

> 假设 Home 组件中嵌套了两个 Message 和 News 路由组件。

a. `<Routes>` + `<Route>`

```
<Route path="/home/news" element={<News/>} />
<Route path="/home/message" element={<Message/>} />
```

b. `useRoutes` + `Outlet` + 路由表

> 通过 `Outlet` 渲染子路由，类似 Vue 中的 `router-view`
>
> 路由表需要新增嵌套规则，同时在 Home 组件中只要使用 `Outlet` 就可以实现子路由的渲染

```
const routes = [
    { path: "/home",element: <Home />, children: [
        {path: "news", element: <News/> },
        {path: "message", element: <Message/> }
    ]},
    { path: "/about",element: <About /> },
    { path: "/",element: <Navigate to="/home" /> }
];
```

```
return { <Outlet/> };
```

3. **多级路由中 `NavLink` 的高亮显示：**给 `NavLink` 组件添加 `end` 属性，可以确保子组件匹配成功时，父组件的导航没有高亮效果。

路由参数

params

- 传递 `<Link to="test/tom/18">点击跳转 Test 组件，并传递 params 参数</Link>`

> 这里的 `to` 属性的值使用了简写形式

- 声明 `{path: "test/:name/:age", element: <Test/>}`

> 这里在路由表中声明了 Test 组件要接受的参数

- 接收 `const {name, age}=useParams()`

> **useParams** Hook 用于获取传递过来的 params 参数

search

- 传递 `<Link to="test/?name=tom&age=18">点击跳转 Test 组件，并传递 search 参数</Link>`
- 声明 `{path: "test", element: <Test/>}`
- 接收

```
1 const [search, setSearch] = useSearchParams();
2 const [name, age] = [search.get("name"), search.get("age")];
```

> **useSearchParams** Hook 用于获取传递过来的 search 参数，该函数返回一个**数组**，第一个元素是 URLSearchParams 对象，可以**使用 `get` 方法解析出 search 参数**，第二个参数是一个方法，用于**设置新的 search 参数**。

state

- 传递 `<Link to="test" state={{name: "tom", age: 18}}>点击跳转 Test 组件，并传递 search 参数</Link>`
- 声明 `{path: "test", element: <Test/>}`
- 接收

```
1 const {state} = useLocation();
2 const {name, age} = state;
```

> **useLocation** Hook 用于获取传递过来的 state 参数，返回值是一个对象，其中的 **state 属性对应的数据就是传递过来的 state 参数**。

编程式路由导航

```
1 import {useNavigate} from "react-router-dom"
2 const navigate = useNavigate(); // 该 Hook 返回的对象可以用于编程式路由导航
```

```
3
4  const showInfo = (person) => {
5      navigate("detail", {
6          replace: false,
7          state: {
8              name: person.name,
9              age: person.age
10         }
11     })
12 }
13 const back = () => navigate(-1);
14 const forward = () => navigate(1);
```

其他 Hooks

- `useInRouterContext()` 检测组件（不管是路由组件还是一般组件）**是否在 Router 的上下文中呈现**，是则返回 `true`，否则返回 `false`。

  > 如果当前组件或其父组件被 `BrowserRouter` 或 `HashRouter` 包裹，则该 Hook 返回值为 `true`，否则 `false`。

- `useNavigationType()` 检测组件的**导航类型**，可取值 `POP、PUSH、REPLACE`。POP 是指直接在浏览器中打开这个路由组件，即刷新页面。
- `useOutlet()` 呈现当前组件中要渲染的**嵌套路由**。注意，如果嵌套路由没有挂载，则输出结果为 `null`。
- `useResolvedPath(url)` **解析**给定 URL 中的 `path、search、hash` 值。