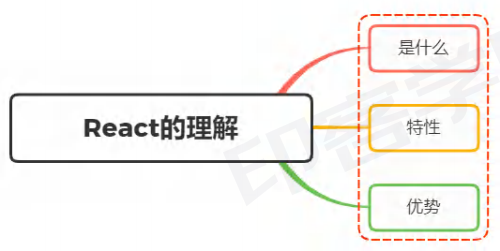
印客学院独家整理 , 盗版必究

React面试真题 (31题)



**1.** 说说对 React 的理解? 有哪些特性?



**1.1.** 是什么

React, 用于构建用户界面的 Javascript 库 , 只提供了 UI 层面的解决方案

遵循组件设计模式 、声明式编程范式和函数式编程概念 , 以使前端应用程序更高效 使用虚拟 DOM  来有效地操作 DOM  · 遵循从高阶组件到低阶组件的单向数据流

帮助我们将界面成了各个独立的小块  每  个块就是组件 · 这些组件之间可以组合 · 嵌套  构成整体页 面

react  类组件使用 -个名为 render()  的方法或者函数组件 ,return 接收输入的数据并返回需

要展示的内容

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1 | class HelloMessage extends React.Component { | |  |
| 2 | |  | | --- | |  | | render() { |  |
| 3 | return <div>Hello {this.props.name}</div>; | |  |
| 4 | } | |  |
| 5  6  7  8  9  10 | }  ReactDOM.render(  <HelloMessage name="Taylor" />,  document.getElementById("hello-example") ); | |  |

上述这种类似 XML  形式就是 ,JSX 最终会被 babel  编译为合法的 JS  语句调用 被传入的数据可在组件中通过 this.props  在 render()  访问

**1.2.** 特性



React  特性有很多 , 如 :

o JSX 语法

。 单向数据绑定

• 虚拟 DOM • 声明式编程  component

着重介绍下声明式编程及 component

**1.2.1.** 声明式编程

声明式编程是一种编程范式 , 它关注的是你要做什么 , 而不是如何做

它表达逻辑而不显式地定义步骤 。这意味着我们需要根据逻辑的计算来声明要显示的组件 如实现一个标记的地图 :

通过命令式创建地图 、创建标记 、以及在地图上添加的标记的步骤如下:

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 | // 创建地图 |  |
| 2 | const map = new Map.map(document.getElementById("map"), { |  |
| 3 | zoom: 4, |  |
| 4 | center: { lat, lng }, |  |
| 5 | }); |  |
| 6 |  |  |
| 7 | // 创建标记 |  |
| 8 | const marker = new Map.marker({ |  |
| 9 | position: { lat, lng }, |  |
| 10 | title: "Hello Marker", |  |
| 11 | }); |  |
| 12 |  |  |
| 13 | // 地图上添加标记 |  |
| 14 | marker.setMap(map); |  |

而用 React  实现上述功能则如下 :

|  |  |
| --- | --- |
| JSX | LO 复制代码 |
| 1 <Map zoom={4} center={(lat, lng)}> |  |
| 2 <Marker position={(lat, lng)} title={"Hello Marker"} /> |  |
| 3 </Map> |  |

声明式编程方式使得 React  组件很容易使用 , 最终的代码简单易于维护

**1.2.2.** component

在 React  中 , 一切皆为组件 。通常将应用程序的整个逻辑分解为小的单个部分 。 我们将每个单独的 部分称为组件

组件可以是  个函数或者是  个类 , 接受数据输入 , 处理它并返回在 UI  中呈现的 React  元素 函数式组件如下 :

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | JSX | COL 复制代码 |
| 1 | const Header = () => { |  |  |
| 2 | return ( |  |  |
| 3 | <Jumbotron style={{ backgroundColor: "orange" }}> |  |  |
| 4 | <h1>TODO App</h1> |  |  |
| 5 | </Jumbotron> |  |  |
| 6 | ); |  |  |
| 7 | }; |  |  |

印客学院独家整理 , 盗版必究

类组件 (有状态组件) 如下 :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | |  | JSX | LO 复制代码 |
| 1 | class Dashboard extends React.Component | | { |  |  |
| 2 | |  | | --- | |  | | constructor(props) { |  |  |  |
| 3 | super(props); | |  |  |  |
| 4 |  | |  |  |  |
| 5 | this.state = {}; | |  |  |  |
| 6 | } | |  |  |  |
| 7 | |  | | --- | |  | | render() { |  |  |  |
| 8 | return ( | |  |  |  |
| 9 | <div className="dashboard"> | |  |  |  |
| 10 | <ToDoForm /> | |  |  |  |
| 11 | <ToDolist /> | |  |  |  |
| 12 | </div> | |  |  |  |
| 13 | ); | |  |  |  |
| 14 | } | |  |  |  |
| 15 | } | |  |  |  |

一个组件该有的特点如下 :

 可组合: 每个组件易于和其它组件一起使用 , 或者嵌套在另 一个组件内部 。 可重用 : 每个组件都是具有独立功能的 , 它可以被使用在多个 UI 场景

 可维护 : 每个小的组件仅仅包含自身的逻辑 , 更容易被理解和维护

**1.3.** 优势



通过上面的初步了解 , 可以感受到 React  存在的优势:

。 高效灵活

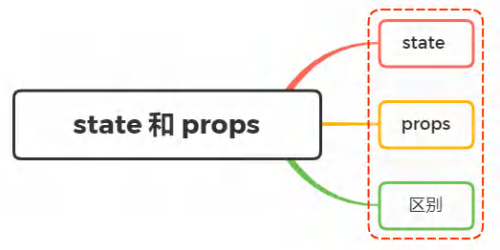
 声明式的设计 , 简单使用

• 组件式开发 , 提高代码复用率

 单向响应的数据流会比双向绑定的更安全 , 速度更快

**2.** state 和 props 有什么区别?

印客学院独家整理 , 盗版必究



**2.1.** state

 constru

-个组件的显示形态可以由数据状态和外部参数所决定 , 而数据状态就是 ,state 般在 ctor  中初始化

当需要修改里面的值的状态需要通过调用 setState  来改变 , 从而达到更新组件内部数据的作用 , 并

且重新调用组件 render  方法 , 如下面的例子 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | JS | X LO 复制代码 |
| 1 | class Button extends React.Component { | | | |  |  |
| 2 | |  | | --- | |  | | constructor() { | | |  |  |
| 3 | super(); | | | |  |  |
| 4 | |  | | --- | |  | | this.state = { | | |  |  |
| 5 | count: 0, | | |  |  |  |
| 6 | }; | | |  |  |  |
| 7 | } | | |  |  |  |
| 8 |  | | | |  |  |
| 9 | |  | | --- | |  | | updateCount() { |  | |  |  |
| 10 | |  | | --- | |  | | this.setState((prevState, props) => { | | |  |  |
| 11 | return { count: prevState.count + 1 }; | | | |  |  |
| 12 | }); | | | |  |  |
| 13 | } | | | |  |  |
| 14 |  | | | |  |  |
| 15 | |  | | --- | |  | | render() { | | |  |  |
| 16 | return ( | | | |  |  |
| 17 | <button onClick={() => this.updateCount()}> | | | |  |  |
| 18 | Clicked {this.state.count} times | | | |  |  |
| 19 | </button> | | | |  |  |
| 20 | ); | | | |  |  |
| 21 | } | | | |  |  |
| 22 | } | | | |  |  |

setState  还可以接受第二个参数 , 它是  个函数 , 会在 setState  调用完成并且组件开始重新渲

染时被调用 , 可以用来监听渲染是否完成 

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | | Javascript | COL 复制代码 |
| 1 | this.setState( | | | |  | | |
| 2 | |  | | --- | |  | | { |  | | | | |
| 3 | name: "JS每日 一题 ", | | |  | | | |
| 4 | }, | | | | | | |
| 5  6 | () => console.log("setState finished") ); | | | | | | |

**2.2.** props

 React  的核心思想就是组件化思想 , 页面会被切分成  些独立的 、可复用的组件

组件从概念上看就是  个函数  可以接受  个参数作为输入值 , 这个参数就是 ,props 所以可以把 

props  理解为从外部传入组件内部的数据

react  具有单向数据流的特性 , 所以他的主要作用是从父组件向子组件中传递数据

印客学院独家整理 , 盗版必究

 props  除了可以传字符串 , 数字 , 还可以传递对象 , 数组甚至是回调函数 , 如下 :

|  |  |
| --- | --- |
|  | JSX LO 复制代码 |
| 1  2  3  4  5  6  7 | class Welcome extends React.Component { render() {   |  | | --- | |  | |  |   return <h1>Hello {this.props.name}</h1>;  } }  const element = <Welcome name="Sara" onNameChanged={this.handleName} />; |

上述 name  属性与 onNameChanged 方法都能在子组件的 props  变量中访问

在子组件中 , props  在内部不可变的 , 如果想要改变它看 , 只能通过外部组件传入新的 props  来 重新渲染子组件 , 否则子组件的 props  和展示形式不会改变

**2.3.** 区别

相同点 :

 两者都是 Javascript 对象 。 两者都是用于保存信息

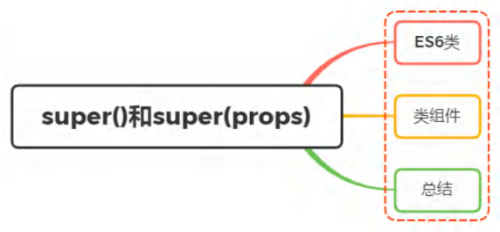
 props 和 state 都能触发渲染更新 区别 :

o props 是外部传递给组件的 , 而 state 是在组件内被组件自己管理的 , 一般在 constructor 中初始 化

 props 在组件内部是不可修改的 , 但 state 在组件内部可以进行修改  state 是多变的 、可以修改

**3.** super() 和 super(props) 有什么区别?

印客学院独家整理 , 盗版必究



**3.1.** ES6 类

在 ES6  中 , 通过 extends  关键字实现类的继承 , 方式如下 :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | Javascript | LO 复制代码 |
| 1 | class sup { | | | |  |  |
| 2 | |  | | --- | |  | | constructor(name) { | | |  |  |
| 3 | this.name = name; | | | |  |  |
| 4 | } | | | |  |  |
| 5 |  | | | |  |  |
| 6 | |  | | --- | |  | | printName() { | | |  |  |
| 7 | console.log(this.name); | | | |  |  |
| 8 | } | | | |  |  |
| 9 | } | | | |  |  |
| 10 |  | | | |  |  |
| 11 | class sub extends sup { | | |  |  |  |
| 12 | |  | | --- | |  | | constructor(name, age) { | |  |  |  |
| 13 | super(name); // super代表的事父类的构造函数 | | | |  |  |
| 14 | this.age = age; | |  | |  |  |
| 15 | } | | | |  |  |
| 16 |  | | | |  |  |
| 17 | |  | | --- | |  | | printAge() { | | |  |  |
| 18 | console.log(this.age); | | | |  |  |
| 19 | } | | | |  |  |
| 20 | } | | | |  |  |
| 21 |  | | | |  |  |
| 22 | let jack = new sub("jack", 20); | | | |  |  |
| 23 | jack.printName(); //输出 : jack | | | |  |  |
| 24 | jack.printAge(); //输出 : 20 | | | |  |  |

在上面的例子中 , 可以看到通过 super  关键字实现调用父类 , super  代替的是父类的构建函数 , 使用 super(name)  相当于调用 sup.prototype.constructor.call(this,name) 

印客学院独家整理 , 盗版必究

如果在子类中不使用 ,super 关键字 , 则会引发报错 , 如下 :



报错的原因是 子类是没有自己的 this  对象的 , 它只能继承父类的 this  对象 , 然后对其进行加工

this  super()

 super() 

对象

如果先调用 ,this 再初始化



this

super()  

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Javascript | LO 复制代码 |
| 1  class sub extends sup {  2  constructor(name, age) {  3 this.age = age;  4 super(name); // super代表的事父类的构造函数 5 } | | | |
| 6 } | | | |



this

所以在子类

中 , 必须先代用

才能引用

super 



constructor



**3.2.** 类组件

在

nstructor 



React

中 , 类组件是基于 ES6  的规范实现的 , 继承 ,React.Component 因此如果用到 

this

就必须写

super() 

才初始化



t  内部也会将其定义在组件实例中 

这时候  在调用 super()  的时候  我们  般都需要传入 props  作为参数  如果不传进去  

|  |  |  |  |
| --- | --- | --- | --- |
| v Javascript B 复制代码 | | | |
| 1 | // React 内部 |  |  |
| 2 | const instance = new YourComponent(props); | | |
| 3 | instance.props = props; | | |



this.props

React

, 在

所以无论有没有

都是可以使用的 , 这是

render 中

constructor

自动附带的 , 是可以不写的 :



|  |  |
| --- | --- |
|  |  |
| 1  class HelloMessage extends React.Component { |  |
| 2  render() { |  |
| 3 return <div>nice to meet you! {this.props.name}</div>;  4 } 5 } |  |

 super(props) 

super() 



React

 this.props 

props 



undefined

super 

this.props

|  |  |
| --- | --- |
|  | |
| 1  2 | class Button extends React.Component {  constructor(props) { |
| 3 | super(); //  props |
| 4 | console.log(props); // {} |
| 5 | console.log(this.props); // undefined |
| 6 | // ... |
| 7 | } |
| 8 | } |



props 

this.props



|  |
| --- |
|  |
| 1  class Button extends React.Component { 2  constructor(props) {  3 super(props); //  props  4 console.log(props); // {}  5 console.log(this.props); // {} |
| 6 // ...  7 } 8 } |

**3.3.** 





React



 super 

rps 

ES6



super 

constructor



  props React   porps    po

印客学院独家整理 , 盗版必究

 super()  

this.props

super() 

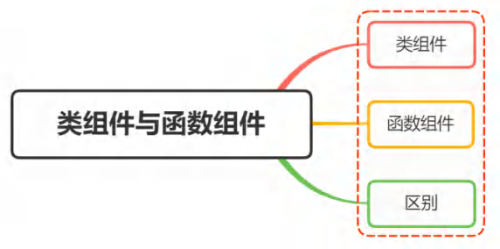
和构造函数结束之间仍是 

ed 

在

**4.** 说说对React中类组件和函数组件的理解? 有什么区

别?



**4.1.** 类组件

类组件 · 顾名思义  也就是通过使用 ES6  类的编写形式去编写组件  该类必须继承 React.Componen

t 



this.props

如果想要访问父组件传递过来的参数 , 在组件中必须实现 render  方法 , 在

的方式去访问

可通过



React

对象 , 如下 :

return 中返回

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | JS | X LO 复制代码 |
| 1 | class Welcome extends React.Component { |  |  |
| 2 | constructor(props) { |  |  |
| 3 | super(props) |  |  |
| 4 | } |  |  |
| 5 | render() { |  |  |
| 6 | return <h1>Hello, {this.props.name}</h1> |  |  |
| 7 | } |  |  |
| 8 | } |  |  |

**4.2.** 函数组件

印客学院独家整理 , 盗版必究

函数组件 , 顾名思义 , 就是通过函数编写的形式去实现-个 React  组件 , 是 React  中定义组件最简 单的方式

|  |  |  |
| --- | --- | --- |
|  | JSX CO | 复制代码 |
| 1  function Welcome(props) { |  |  |
| 2 return <h1>Hello, {props.name}</h1>; |  |  |
| 3 } |  |  |

props  用于接收父组件传递过来的参数

函数第一个参数为





**4.3.**

区别

组件 , 其区别主要分成以下几大方向 :

针对两种



React

编写形式 状态管理

生命周期

。 调用方式

 获取渲染的值

**4.3.1.** 编写形式

两者最明显的区别在于编写形式的不同 , 同 一种功能的实现可以分别对应类组件和函数组件的编写形式 函数组件:

Hello

|  |  |  |  |
| --- | --- | --- | --- |
| ·    JS 。复制代码 | | | |
| 1  2 | |  | | --- | |  | | function Welcome(props) {  h1 | h1 |
| 3 | }  return < > , {props.name}</ >; | | |

类组件:

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JS | | X LO 复制代码 |
| 1 | class Welcome extends React.Component { | |  |
| 2 | |  | | --- | |  | | constructor(props) { |  |
| 3  4  5  6  7 | |  | | --- | |  | | super(props)  }  render() {  return <h1>Hello, {this.props.name}</h1> } |  |
| 8 | } | |  |



**4.3.2.** 状态管理

 hooks  出来之前 , 函数组件就是无状态组件 , 不能保管组件的状态 , 不像类组件中调 用

在

setState

如果想要管理

useState

state 状态 , 可以使用

如下:

|  |  |  |
| --- | --- | --- |
|  | JSX | LO 复制代码 |
| 1 | const FunctionalComponent = () => { |  |
| 2 | const [count, setCount] = React.useState(0); |  |
| 3 |  |  |
| 4 | return ( |  |
| 5  6  7  8  9 | <div>  <p>count: {count}</p >  <button onClick={() => setCount(count + 1)}>Click</ </div>  ); | button> |
| 10 | }; |  |

hooks  情况下 , -般如果函数组件调用

在使用

,state 则需要创建一个类组件或者

state 提升到

 props  对象传递到子组件

你的父组件中 , 然后通过

**4.3.3.** 生命周期

React.Component

在函数组件中 , 并不存在生命周期 , 这是因为这些生命周期钩子都来自于继承的

所以 , 如果用到生命周期 , 就只能使用类组件

但是函数组件使用 useEffect  也能够完成替代生命周期的作用 , 这里给出 -个简单的例子 :

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | JS | X LO 复制代码 |
| 1  const FunctionalComponent = () => { |  |  |
| 2  useEffect(() => { |  |  |
| 3 console.log("Hello"); |  |  |
| 4 }, []); |  |  |
| 5 return <h1>Hello, World</h1>; |  |  |
| 6 }; |  |  |

 componentDidMount 

生命周期

上述简单的例子对应类组件中的

return 一个函数 , 则

如果在

useEffect 回调函数中

return 函数会在组件卸载的时候执行 ,

正

 componentWillUnmount 

如

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | |  |  | JS | X LO 复制代码 |
| 1 | const FunctionalComponent = () | | => | { |  |  |
| 2 | React.useEffect(() => { | |  |  |  |  |
| 3 | |  | | --- | |  | | return () => { |  |  |  |  |
| 4 | console.log("Bye"); | |  |  |  |  |
| 5 | }; | |  |  |  |  |
| 6 | }, []); | |  |  |  |  |
| 7 | return <h1>Bye, World</h1>; | |  |  |  |  |
| 8 | }; | |  |  |  |  |

**4.3.4.** 调用方式

如果是一个函数组件 , 调用则是执行函数即可:

|  |
| --- |
| v JSX LO 复制代码 |
| 1 // 你的代码  2  function SayHi() {  6 const result = SayHi( ) // » <p>Hello React</p >  3 return <p>Hello, React</p >  4 }  5 // React内部 |
| props , |

如果是  个类组件 , 则需要将组件进行实例化 , 然后调用实例对象的 render  方法:

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1 | // 你的代码 | |  |
| 2 | class SayHi extends React.Component { | |  |
| 3  4  5  6  7  8  9 | |  | | --- | |  | | render() {  return <p>Hello, React</p >  } }  // React内部  const instance = new SayHi(props) // » SayHi {}  const result = instance.render() // » <p>Hello, React</p > |  |

**4.3.5.** 获取渲染的值

首先给出 一个示例 函数组件对应如下:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX | | | | COL 复制代码 |
| 1 | function ProfilePage(props) { | | | |  |
| 2 | |  | | --- | |  | | | const showMessage = () => { | |  |
| 3 | alert( 'Followed ' + props.user); | | | |  |
| 4 | } | | | |  |
| 5  6  7  8  9  10  11  12  13 | |  | | --- | |  | | } | | const handleClick = () => {  setTimeout(showMessage, 3000);  }  return (  <button onClick={handleClick}>Follow</button> ) |  |

类组件对应如下 :

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | JSX LO 复制代码 | |
| 1 | class ProfilePage extends React.Component { | |
| 2 | |  | | --- | |  | | showMessage() { |
| 3  4  5  6  7  8  9  10  11  12 |  | alert( 'Followed ' + this.props.user); }  handleClick() {  setTimeout(this.showMessage.bind(this), 3000); }  render() {  onClick={this.handleClick.bind(this)}>Follow</button>  return <button  } |
| 13 | } | |

两者看起来实现功能是一致的 , 但是在类组件中 , 输出 this.props.user  , Props  在 React  中 是不可变的所以它永远不会改变 , 但是 this  总是可变的 , 以便您可以在 render  和生命周期函数 中读取新版本

因此 , 如果我们的组件在请求运行时更新 。 this.props  将会改变 。 showMessage  方法从 " 最

新" 的 props  中读取 user 

而函数组件 , 本身就不存在 this  , props  并不发生改变 , 因此同样是点击 , alert  的内容仍旧是 之前的内容

**4.3.6.** 小结



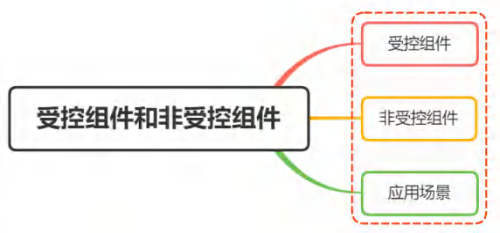
两种组件都有各自的优缺点

函数组件语法更短 、更简单 , 这使得它更容易开发 、理解和测试

而类组件也会因大量使用 this  而让人感到困惑

**5.** 说说对受控组件和非受控组件的理解? 应用场景?

印客学院独家整理 , 盗版必究



**5.1.** 受控组件

受控组件 , 简单来讲 , 就是受我们控制的组件 , 组件的状态全程响应外部数据 举个简单的例子 :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | JSX CO | | | | | 复制代码 |
| 1 | class TestComponent extends React.Component { | | | | |  |
| 2 | |  | | --- | |  | | | constructor (props) { | | |  |
| 3 | super(props); | | | | |  |
| 4 | this.state = { username: 'lindaidai' }; | | | | |  |
| 5  6  7  8  9 | |  | | --- | |  | | } | | }  render () {  return <input  } | name="username" value={this.state.username} /> |  |

这时候当我们在输入框输入内容的时候  会发现输入的内容并无法显示出来  也就是 input  标签是-

个可读的状态 



this.state.username



value

 this.state.u

当用户输入新的内容时 ,

所控制住。

这是因为

被

sername  并不会自动更新 , 这样的话 input  内的内容也就不会变了

如果想要解除被控制 , 可以为 input  标签设置 onChange  事件 , 输入的时候触发事件函数 , 在函数 内部实现 state  的更新 , 从而导致 input  框的内容页发现改变

因此 , 受控组件我们 一般需要初始状态和 一个状态更新事件函数

**5.2.** 非受控组件

非受控组件 , 简单来讲 , 就是不受我们控制的组件

印客学院独家整理 , 盗版必究

一般情况是在初始化的时候接受外部数据 , 然后自己在内部存储其自身状态 当需要时 , 可以使用 ref  查询 DOM  并查找其当前值 , 如下 :

|  |  |  |
| --- | --- | --- |
|  |  | JSX COL 复制代码 |
| 1 import React, { Component } from 'react'; 2  3- export class UnControll extends Component {  4  constructor (props) {  5 super(props);  6 this.inputRef = React.createRef(); 7 }  8  handleSubmit = (e) => {  9 console.log( '我们可以获得input内的值为 ', this.inputRef.current.value);  10 e.preventDefault(); 11 }  12  render () {  13 return (  14 <form onSubmit={e => this.handleSubmit(e)}>  15 <input defaultValue="lindaidai" ref={this.inputRef} /> | | |
| 16 <input type="submit" value="提交 " />  17 </form> 18 )  19 } 20 } | | |

**5.3.** 应用场景

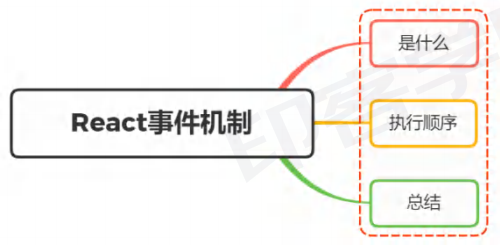
大部分时候推荐使用受控组件来实现表单 , 因为在受控组件中 , 表单数据由 React  组件负责处理

如果选择非受控组件的话 , 控制能力较弱 , 表单数据就由 DOM  本身处理 , 但更加方便快捷 , 代码量少 针对两者的区别 , 其应用场景如下图所示 :

印客学院独家整理 , 盗版必究



**6.** 说说React的事件机制?



**6.1.** 是什么

印客学院独家整理 , 盗版必究

React  基于浏览器的事件机制自身实现了  套事件机制 , 包括事件注册 、事件的合成 、事件冒泡 、事 件派发等

React 中这套事件机制被称之为合成事件

在

**6.1.1.** 合成事件 (syntheticEvent)

DOM

React 模拟原生

事件所有能力的 一个事件对象 , 即浏览器原生事件的跨浏览器包装

合成事件是

器

 W3C  规范来定义合成事件 , 兼容所有浏览器 , 拥有与浏览器原生事件相同的接口 , 例如 :

根据

|  |  |
| --- | --- |
| v JS | X LO 复制代码 |
| 1 const button = <button onClick={handleClick}>按钮</button> |  |

如果想要获得原生 DOM  事件 , 可以通过 e.nativeEvent  属性获取

|  |  |
| --- | --- |
| v Javascript | LO 复制代码 |
| 1 const handleClick = (e) => console.log(e.nativeEvent);;  2 const button = <button onClick={handleClick}>按钮</button> |  |

从上面可以看到 React  事件和原生事件也非常的相似 , 但也有一定的区别 :

• 事件名称命名方式不同

|  |  |
| --- | --- |
|  | JSX COL 复制代码 |
| 1 | // 原生事件绑定方式 |
| 2 | <button onclick="handleClick()">按钮命名</button> |
| 3 |  |
| 4 | // React 合成事件绑定方式 |
| 5 | const button = <button onClick={handleClick}>按钮命名</button> |

• 事件处理函数书写不同

|  |  |
| --- | --- |
| v JS | X LO 复制代码 |
| 1 // 原生事件 事件处理函数写法 |  |
| 2 <button onclick="handleClick()">按钮命名</button> |  |
| 3 |  |
| 4 // React 合成事件 事件处理函数写法 |  |
| 5 const button = <button onClick={handleClick}>按钮命名</button> |  |

印客学院独家整理 , 盗版必究

虽然 onclick  看似绑定到 DOM  元素上 , 但实际并不会把事件代理函数直接绑定到真实的节点上 , 而 是把所有的事件绑定到结构的最外层 , 使用 一个统一的事件去监听

这个事件监听器上维持了 一个映射来保存所有组件内部的事件监听和处理函数 。当组件挂载或卸载时 , 只是在这个统一的事件监听器上插入或删除一些对象

当事件发生时 , 首先被这个统一 的事件监听器处理 , 然后在映射里找到真正的事件处理函数并调用 。这 样做简化了事件处理和回收机制 , 效率也有很大提升



**6.2.** 执行顺序

关于 React  合成事件与原生事件执行顺序 , 可以看看下面-个例子 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JS | | | | X LO 复制代码 |
| 1 | import React from 'react'; | | | |  |
| 2 | class App extends React.Component{ | | | |  |
| 3 |  | | | |  |
| 4 | |  | | --- | |  | | constructor(props) { | |  |  |
| 5 | super(props); | | |  |  |
| 6 | this.parentRef = React.createRef(); | | | |  |
| 7 | this.childRef = React.createRef(); | | | |  |
| 8 | } | | | |  |
| 9 | |  | | --- | |  | | componentDidMount() { |  | |  |
| 10 | console.log("React componentDidMount  "); | | | |  |
| 11 | |  | | --- | |  | | this.parentRef.current?.addEventListener("click", () => | | | { |
| 12 | console.log("原生事件 : 父元素 DOM 事件监听  "); | | | |  |
| 13 | }); | | | |  |
| 14 | this.childRef.current?.addEventListener("click", () => { | | | |  |
| 15 | console.log("原生事件 : 子元素 DOM 事件监听! "); | | | |  |
| 16 | }); | | | |  |
| 17 | |  | | --- | |  | | document.addEventListener("click", (e) => { | | |  |
| 18 | console.log("原生事件 : document DOM 事件监听! "); | | | |  |
| 19 | }); | | | |  |
| 20 | } | | | |  |
| 21 | |  | | --- | |  | | parentClickFun = () => { | | |  |
| 22 | console.log("React 事件 : 父元素事件监听! "); | | | |  |
| 23 | }; | | | |  |
| 24 | |  | | --- | |  | | childClickFun = () => { | | |  |
| 25 | console.log("React 事件 : 子元素事件监听! "); | | | |  |
| 26 | }; | | | |  |
| 27 | |  | | --- | |  | | render() { | | |  |
| 28 | return ( | | | |  |
| 29 | <div ref={this.parentRef} onClick={this.parentClickFun | | | | }> |
| 30 | <div ref={this.childRef} onClick={this.childClickFun | | | | }> |
| 31 | 分析事件执行顺序 | | | |  |
| 32 | </div> | | | |  |
| 33 | </div> | | | |  |
| 34 | ); | | | |  |
| 35 | } | | | |  |
| 36 | } | | | |  |
| 37 | export default App; | | | |  |

输出顺序为 :

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
| v LaTe | | | X LO 复制代码 |
| 1 原生事件 : 子元素 DOM 事件监听! | | |  |
| 2 原生事件 : 父元素 DOM 事件监听! | | |  |
| 3  4  5 | React 事件 : 子元素事件监听! React 事件 : 父元素事件监听!  原生事件 : document DOM 事件监听! |  |  |

可以得出以下结论:

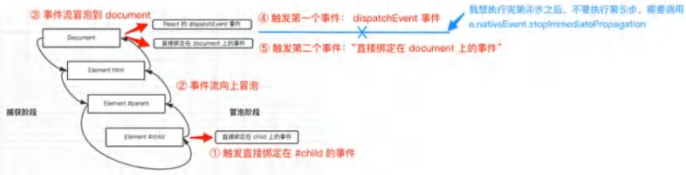
o React 所有事件都挂载在 document 对象上

当真实 DOM 元素触发事件 , 会冒泡到 document 对象后 , 再处理 React 事件

所以会先执行原生事件 , 然后处理 React 事件

最后真正执行 document 上挂载的事件

对应过程如图所示:



所以想要阻止不同时间段的冒泡行为 , 对应使用不同的方法 , 对应如下:

阻止合成事件间的冒泡 , 用e.stoppropagation()

阻止合成事件与最外层 document 上的事件间的冒泡 ,

用

e. nativeEvent.stoplmmediatepropagation()

阻止合成事件与除最外层document上的原生事件上的冒泡 , 通过判断e.targEt来避免

|  |  |
| --- | --- |
| v Javascript | LO 复制代码 |
| 1  document.body.addEventListener( 'click', e => { |  |
| 2  if (e.target && e.target.matches( 'div.code')) { |  |
| 3 return; |  |
| 4 } |  |
| 5 this.setState({ active: false, }); }); |  |
| 6 } |  |

**6.3.** 总结

印客学院独家整理 , 盗版必究

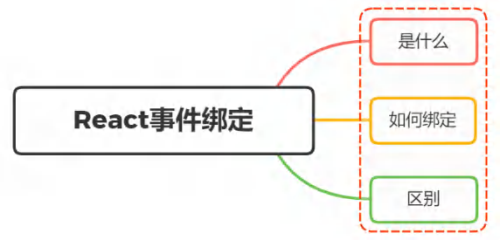
React  事件机制总结如下 :

 React 上注册的事件最终会绑定在document这个 DOM 上 , 而不是 React 组件对应的 DOM(减少 内存开销就是因为所有的事件都绑定在 document 上 , 其他节点没有绑定事件)

 React 自身实现了 一套事件冒泡机制 , 所以这也就是为什么我们 event. stoppropagation()无效的原 因 。

 React 通过队列的形式 , 从触发的组件向父组件回溯 , 然后调用他们 JSX 中定义的 callback  React 有 一套自己的合成事件 syntheticEvent

**7.** React事件绑定的方式有哪些? 区别?



**7.1.** 是什么



onClick

 react  应用中 , 事件名都是用小驼峰格式进行书写 , 例如

在

onclick 要改写成

最简单的事件绑定如下:

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1 | class ShowAlert extends React.Component { | |  |
| 2 | |  | | --- | |  | | showAlert() { |  |
| 3 | console.log("Hi"); | |  |
| 4 | } | |  |
| 5 |  | |  |
| 6 | |  | | --- | |  | | render() { |  |
| 7 | return <button onClick={this.showAlert}>show</button>; | |  |
| 8 | } | |  |
| 9 | } | |  |

印客学院独家整理 , 盗版必究

从上面可以看到 , 事件绑定的方法需要使用 {}  包住

上述的代码看似没有问题 , 但是当将处理函数输出代码换成 console.log(this)  的时候 , 点击按 钮 , 则会发现控制台输出 undefined 

**7.2.** 如何绑定

为了解决上面正确输出 this  的问题 , 常见的绑定方式有如下:  render方法中使用bind

 render方法中使用箭头函数 o constructor中bind

• 定义阶段使用箭头函数绑定

**7.2.1.** render方法中使用bind

如果使用 -个类组件 , 在其中给某个组件/元素  个 onClick  属性 , 它现在并会自定绑定其 this  到

当前组件 , 解决这个问题的方法是在事件函数后使用 .bind(this)  将 this  绑定到当前组件中

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1  2  3  4  5  6  7  8  9  10 |  | class App extends React.Component { handleClick() {  console.log( 'this > ', this); }  render() { return (  <div onClick={this.handleClick.bind(this)}>test</div> )  } } |  |

这种方式在组件每次 render  渲染的时候 · 都会重新进行 bind  的操作 · 影响性能





**7.2.2.** render方法中使用箭头函数

通过 ES6  的上下文来将 this  的指向绑定给当前组件 , 同样再每  次 render  的时候都会生成新的

方法 , 影响性能

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | JSX | | | | | LO 复制代码 |
| 1 | class App extends React.Component { | | | | |  |
| 2 | |  | | --- | |  | | | handleClick() { | | |  |
| 3  4  5  6  7  8  9  10 | |  | | --- | |  | | } | | console.log( 'this }  render() { return (  <div onClick={e )  } | > ', this);  => this.handleClick(e)}>test</div> |  |



**7.2.3.** constructor中bind



constructor 中预先

render 操作中重复绑定

在

bind  当前组件 , 可以避免在

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX | | | | COL 复制代码 |
| 1 | class App extends React.Component { | | | |  |
| 2 | |  | | --- | |  | | | constructor(props) { | |  |
| 3 | super(props); | | | |  |
| 4 | this.handleClick = this.handleClick.bind(this); | | | |  |
| 5 | } | | | |  |
| 6  7  8  9  10  11  12  13  14 |  | } | | handleClick() {  console.log( 'this > ', this); }  render() { return (  <div onClick={this.handleClick}>test</div> )  } |  |



**7.2.4.** 定义阶段使用箭头函数绑定

跟上述方式三-样 , 能够避免在 render  操作中重复绑定 , 实现也非常的简单 , 如下:

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1 | class App extends React.Component { | |  |
| 2 | |  | | --- | |  | | constructor(props) { |  |
| 3  4  5  6  7  8  9  10  11  12 |  | super(props); }  handleClick = () => {  console.log( 'this > ', this);  }  render() { return (  <div onClick={this.handleClick}>test</div> )  } |  |
| 13 | } | |  |



**7.3.** 区别

上述四种方法的方式 , 区别主要如下 :

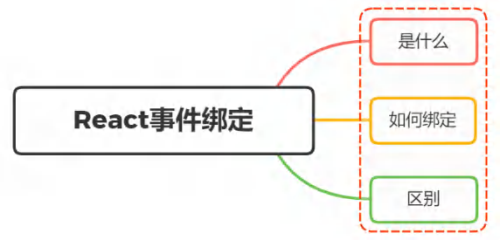
• 编写方面 : 方式一 、方式二写法简单 , 方式三的编写过于冗杂

。 性能方面 : 方式一和方式二在每次组件render的时候都会生成新的方法实例 , 性能问题欠缺 。若该 函数作为属性值传给子组件的时候 , 都会导致额外的渲染 。而方式三 、方式四只会生成一个方法实

例

综合上述 , 方式四是最优的事件绑定方式

**8.** React构建组件的方式有哪些? 区别?



印客学院独家整理 , 盗版必究

**8.1.** 是什么

组件就是把图形 、非图形的各种逻辑均抽象为 一个统一的概念 (组件) 来实现开发的模式

React  中 , -个类 、一个函数都可以视为 -个组件

在

降低整个系统的耦合度 , 在保持接口不变的情况下 , 我们可以替换不同的组件快速完成需求 , 例如 输入框 , 可以替换为 日历 、时间 、范围等组件作具体的实现

调试方便 , 由于整个系统是通过组件组合起来的 , 在出现问题的时候

可以用排除法直接移除组

件 , 或者根据报错的组件快速定位问题 , 之所以能够快速定位 , 是因为每个组件之间低耦合 , 职责 单一 , 所以逻辑会比分析整个系统要简单

并且组件在系统中是被复用的 , 所以对代码进行优化可

提高可维护性 , 由于每个组件的职责单

获得系统的整体升级

**8.2.** 如何构建

 目React前来讲 , 组件的创建主要分成了

在

种方式 :

 函数式创建

 通过 React. createclass 方法创建 • 继承 REact. component 创建

**8.2.1.** 函数式创建

React Hooks  出来之前 , 函数式组件可以视为无状态组件 , 只负责根据传入的

来展示视

在

props 

状态的操作

图 , 不涉及对



state



React 中 , 通过函数简单创建组件的示例如下:

大多数组件可以写为无状态组件 , 通过简单组合构建其他组件 在

|  |  |  |
| --- | --- | --- |
|  | JSX CO | 复制代码 |
| 1  function HelloComponent(props, /\* context \*/) { |  |  |
| 2 return <div>Hello {props.name}</div> |  |  |
| 3 } |  |  |



**8.2.2.** 通过 React.createclass 方法创建

React.createClass  是react刚开始推荐的创建组件的方式 , 目前这种创建方式已经不怎么用了

印客学院独家整理 , 盗版必究

像上述通过函数式创建的组件的方式 , 最终会通过 babel  转化成 React.createClass  这种形式 , 转化成如下 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| v JSX LO 复制代码 | | | | |
| 1 | function HelloComponent(props) /\* context \*/{ | | |  |
| 2 | return React.createElement( | |  | |
| 3 | "div", | |  |  |
| 4 | null, | |  |  |
| 5 | "Hello ", | | | |
| 6 | props.name |  | | |
| 7 | ); |  | | |
| 8 | } | | | |

由于上述的编写方式过于冗杂 , 目前基本上不使用上



**8.2.3.** 继承 React.component 创建



React.Component

 react hooks  出来之前 , 有状态的组件只能通过继承

同样在

这种形式进行创建

有状态的组件也就是组件内部存在维护的数据 , 在类创建的方式中通过 this.state  进行访问 当调用 this.setState  修改组件的状态时 , 组价会再次会调用 render()  方法进行重新渲染 通过继承 React.Component  创建一个时钟示例如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | JSX | | LO 复制代码 |
| 1 | class Timer extends React.Component { | | |  | |  |
| 2 | |  | | --- | |  | | constructor(props) { | |  | |  |
| 3 | super(props); | | |  | |  |
| 4 | this.state = { seconds: 0 }; | | |  | |  |
| 5 | } | | |  | |  |
| 6 |  | | |  |  |  |
| 7 | |  | | --- | |  | | tick() { | |  | |  |
| 8 | |  | | --- | |  | | this.setState(state => ({ | |  | |  |
| 9 | seconds: state.seconds + 1 | | |  | |  |
| 10 | })); | |  |  | |  |
| 11 | } | | |  | |  |
| 12 |  | | |  | |  |
| 13 | |  | | --- | |  | | componentDidMount() { | |  | |  |
| 14 | this.interval = setInterval(() => | | | this.tick(), 1000); | |  |
| 15 | } | | |  | |  |
| 16 |  | | |  | |  |
| 17 | |  | | --- | |  | | componentWillUnmount() { | |  | |  |
| 18 | clearInterval(this.interval); | | |  | |  |
| 19 | } | | |  | |  |
| 20 |  | | |  | |  |
| 21 | |  | | --- | |  | | render() { | |  | |  |
| 22 | return ( | | |  | |  |
| 23 | <div> | | |  | |  |
| 24  25  26  27  28 | Seconds: {this.state.seconds}  </div> );  } } | | |  | |  |



**8.3.** 区别



由于 React.createClass  创建的方式过于冗杂 , 并不建议使用

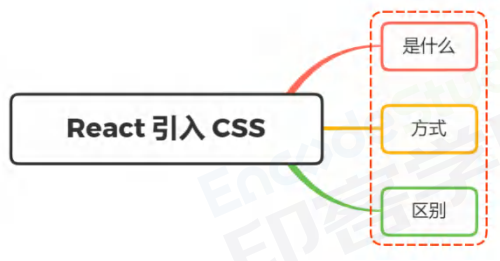
而像函数式创建和类组件创建的区别主要在于需要创建的组件是否需要为有状态组件:  对于一些无状态的组件创建 , 建议使用函数式创建的方式

 由于 react hooks  的出现 , 函数式组件创建的组件通过使用 hooks  方法也能使之成为有状态 组件 , 再加上目前推崇函数式编程 , 所以这里建议都使用函数式的方式来创建组件

在考虑组件的选择原则上 , 能用无状态组件则用无状态组件

印客学院独家整理 , 盗版必究

**9.** 说说react中引入css的方式有哪几种? 区别?



**9.1.** 是什么

组件式开发选择合适的 css  解决方案尤为重要

通常会遵循以下规则 :

。 可以编写局部CSS , 不会随意污染其他组件内的原生;

 可以编写动态的CSS , 可以获取当前组件的 一些状态 , 根据状态的变化生成不同的CSS样式; • 支持所有的CSS特性 : 伪类 、动画 、媒体查询等;

• 编写起来简洁方便 、最好符合一 贯的CSS风格特点 在这-方面 , vue  使用 css  起来更为简洁:

 通过 style 标签编写样式

 scoped 属性决定编写的样式是否局部有效 o lang 属性设置预处理器

 内联样式风格的方式来根据最新状态设置和改变CSS

而在 react  中 , 引入 CSS  就不如 Vue  方便简洁 , 其引入 css  的方式有很多种 , 各有利弊

**9.2.** 方式

常见的 CSS  引入方式有以下 : • 在组件内直接使用

• 组件中引入 . CSS 文件

印客学院独家整理 , 盗版必究

• 组件中引入 . module. Css 文件 o CSS in JS

**9.2.1.** 在组件内直接使用

直接在组件中书写 css  样式 , 通过 style  属性直接引入 , 如下 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | Javascript | LO 复制代码 |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import React, { Component } from "react";  const div1 = {  width: "300px",  margin: "30px auto",  backgroundColor: "#44014C", //驼峰法  minHeight: "200px",  boxSizing: "border-box" };  class Test extends Component {  constructor(props, context) { | |  | |
| super(props); }  13  14  15  render() { return ( <div>  16  17  18  19  20  21  22  23  24  <div  style={div1}>123</div>  style={{backgroundColor:"red"}}>  <div  </div> );  } }  25  26  default Test;  export | | | | |

上面可以看到 , css  属性需要转换成驼峰写法 这种方式优点:

 内联样式, 样式之间不会有冲突

。 可以动态获取当前state中的状态 缺点 :

 写法上都需要使用驼峰标识

印客学院独家整理 , 盗版必究  某些样式没有提示

 大量的样式, 代码混乱

。 某些样式无法编写(比如伪类/伪元素)



**9.2.2.** 组件中引入CSS文件

css

将 css  单独写在-个 App.css  文件 :

文件中 ,

然后在组件中直接引入

|  |  |  |
| --- | --- | --- |
| CSS LO 复制代码 | | |
| 1  .title {  2 color: red;  3 font-size: 20px; 4 }  5  6  .desc {  7 color: green; |  |  |
| 8 text-decoration: underline;  9 } | | |

组件中引入 :

|  |  |
| --- | --- |
|  | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | React, { PureComponent } from 'react'; Home from './Home';  import import import  './App.css';  export default class App extends PureComponent { render() {   |  | | --- | |  | |  |   return (  <div className="app">  <h2 className="title">我是App的标题</h2>  <p className="desc">我是App中的  段文字描述</p > <Home/>  </div> |
| 15 ) 16 } 17 } | |

这种方式存在不好的地方在于样式是全局生效 , 样式之间会互相影响

印客学院独家整理 , 盗版必究

**9.2.3.** 组件中引入 .module.Css 文件

将 css  文件作为 -个模块引入 , 这个模块中的所有 ,css 只作用于当前组件 。不会影响当前组件的后 代组件



 modules:true 

webpack 

配置文件中

这种方式是

特工的方案 ,

只需要配置

即可

webpack 

|  |  |  |  |
| --- | --- | --- | --- |
| LO 复制代码  JSX | | | |
| 1  2  3  4  5  6  7  8  9  10  11 | import React, { PureComponent } from 'react'; import Home from './Home';  import './App.module.css';  export default class App extends PureComponent { render() {   |  | | --- | |  | |  |   return (  <div className="app">  <h2 className="title">我是App的标题</h2> | |  |
| 13 <Home/> 14 </div>  12 <p className="desc">我是App中的  段文字描述</p >  15 )  16 } 17 } | |  | |

这种方式能够解决局部作用域问题 , 但也有一定的缺陷 :

。 引用的类名 , 不能使用连接符( .xxx-xx) , 在 Javascript 中是不识别的

 所有的 classNamE 都必须使用 {style. className} 的形式来编写

 不方便动态来修改某些样式 , 依然需要使用内联样式的方式;

**9.2.4.** CSS in JS

Css-in-JS, 是指 一种模式 , 其中 CSS  由 JavaScript  生成而不是在外部文件中定义 此功能并不是 React 的 一部分 , 而是由第三方库提供 , 例如 :

o styled-components

o emotion

o glamorous

下面主要看看 styled-components  的基本使用

印客学院独家整理 , 盗版必究

本质是通过函数的调用 , 最终创建出 一个组件 :

。 这个组件会被自动添加上一个不重复的class

 styled-components会给该class添加相关的样式

基本使用如下:

创建一个 style.js  文件用于存放样式组件 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | Javascript | LO 复制代码 |
| 1  2  3  4  5  6  7  8  9  10 | export const SelfLink = styled.div` height: 50px;  border: 1px solid red; color: yellow;  `  ;  export const SelfButton = styled.div`  height: 150px; width: 150px;  color: ${props => props.color}; | |  | |
| 11 background-image: url(${props => props.s rc});  12 background-size: 150px 150px; 13 `; | | | | |

引入样式组件也很简单:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX LO 复制代码 | | | | |
| 1 | import React, { Component } from "react"; | | | | |
| 2 |  | | | | |
| 3 | import { SelfLink, SelfButton } from "./style"; | | | | |
| 4 |  | | | | |
| 5 | class Test extends Component { | | |  | |
| 6 | |  | | --- | |  | | constructor(props, context) { | |  | |
| 7 | super(props); | | |  |  |
| 8 | } | | | | |
| 9 |  | | | | |
| 10 | |  | | --- | |  | | render() { |  | | |
| 11 | return ( | | | | |
| 12 | <div> | | | | |
| 13 | <SelfLink title="People's Republic of China">app.js</SelfLink> | | | | |
| 14 | <SelfButton color="palevioletred" style={{ color: "pink" }} s rc={fi  st}> | | | | |
| 15 | SelfButton | | | | |
| 16 | </SelfButton> | | | | |
| 17 | </div> | | | | |
| 18 | ); | | | | |
| 19 | } | | | | |
| 20 | } | | | | |
| 21 |  | | | | |
| 22 | export default Test; | | | | |

**9.3.**

区别





css 该方式编写方便 , 容易能够根据状态修改样式属性 , 但是大量的演示编写

通过上面四种样式的引入 , 可以看到 : 在组件内直接使用

容易导致代码混乱

组件中引入 .CSS 文件符合我们日常的编写习惯 , 但是作用域是全局的 , 样式之间会层叠 引入 . module. Css 文件能够解决局部作用域问题 , 但是不方便动态修改样式

需要使用内联的方式

进行样式的编写

通过CSS in js 这种方法 , 可以满足大部分场景的应用 , 可以类似于预处理器

样样式嵌套 、定义 、

修改状态等

css , 并没有一个绝对的答案 , 可以根据各自情况选择合适的方案

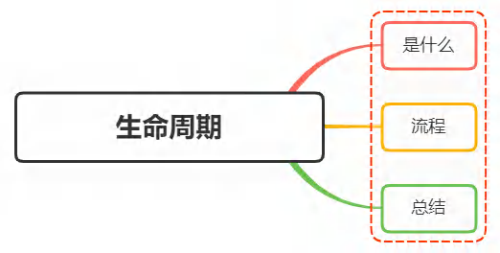
至于使用

react 用哪种方案引入

**10.** 说说 React 生命周期有哪些不同阶段? 每个阶段对

印客学院独家整理 , 盗版必究

应的方法是?



**10.1.** 是什么

生命周期 ( )LifeCycle 的概念应用很广泛 , 特别是在经济 、环境 、技术 、社会等诸多

领域经常出

现 , 其基本涵义可以通俗地理解为 "从摇篮到坟墓" ( )Cradle-to-Grave 的整个过程

跟 一Vue样 , React  整个组件生命周期包括从创建 、初始化数据 、编译模板 、挂载Dom一渲染  更 新→渲染 、卸载等  系列过程

**10.2.** 流程



这里主要讲述 react16.4  之后的生命周期 , 可以分成三个阶段:

。 创建阶段 • 更新阶段 。 卸载阶段

**10.2.1.** 创建阶段

创建阶段主要分成了以下几个生命周期方法:

o constructor

o getDerivedstateFromprops o render

 componentDidMount

印客学院独家整理 , 盗版必究

[**10.2.1.1**](10.2.1.1)**.** constructor

实例过程中自动调用的方法 , 在方法内部通过 super  关键字获取来自父组件的 props  在该方法中 , 通常的操作为初始化 state  状态或者在 this  上挂载方法 



**10.2.2.** getDerivedstateFromprops  

该方法是新增的生命周期方法 , 是一个静态的方法 , 因此不能访问到组件的实例

执行时机: 组件创建和更新阶段 , 不论是 props  变化还是 state  变化 , 也会调用

在每次 render  方法前调用 , 第一个参数为即将更新的 ,props 第二个参数为上-个状态 的 ,state 可以比较 props  和 state  来加  些限制条件 , 防止无用的state更新

该方法需要返回 -个新的对象作为新的 state  或者返回 null  表示 state  状态不需要更新

**10.2.3.** render

类组件必须实现的方法 , 用于渲染 DOM  结构 , 可以访问组件 state  与 prop  属性 注意: 不要在 render  里面 ,setState 否则会触发死循环导致内存崩溃

**10.2.4.** componentDidMount

组件挂载到真实 DOM  节点后执行 , 其在 render  方法之后执行 此方法多用于执行一些数据获取 , 事件监听等操作

**10.2.5.** 更新阶段

该阶段的函数主要为如下方法 :

 getDerivedstateFromprops o shouldcomponentupdate

o render

o getsnapshotBeforeupdate o componentDidupdate

**10.2.6.** getDerivedstateFromprops

该方法介绍同上

印客学院独家整理 , 盗版必究

**10.3.** shouldcomponentupdate

用于告知组件本身基于当前的 props  和 state  是否需要重新渲染组件 , 默认情况返回 true  执行时机: 到新的props或者state时都会调用 , 通过返回true或者fase告知组件更新与否

一般情况 , 不建议在该周期方法中进行深层比较 , 会影响效率

同时也不能调用 setState  · 否则会导致无限循环调用更新 

**10.3.1.** render

介绍如上

**10.3.2.** getsnapshotBeforeupdate

该周期函数在 render  后执行 , 执行之时 DOM  元素还没有被更新

该方法返回的-个 Snapshot  值 , 作为 componentDidUpdate  第三个参数传入

|  |  |  |
| --- | --- | --- |
|  | JSX COL 复制代码 | |
| 1 | getSnapshotBeforeUpdate(prevProps, prevState) { | |
| 2 | console.log( '#enter getSnapshotBeforeUpdate'); | |
| 3  4  5  6  7  8 | |  | | --- | |  | | return 'foo';  }  componentDidUpdate(prevProps, prevState, snapshot) {  console.log( '#enter componentDidUpdate snapshot = ', snapshot); } |

此方法的目的在于获取组件更新前的 一些信息 , 比如组件的滚动位置之类的 , 在组件更新后可以根据这 些信息恢复一些UI视觉上的状态

**10.3.3.** componentDidupdate

执行时机: 组件更新结束后触发

 state  的变化做相应的操作 , 如获取数据 , 修改

在该方法中 , 可以根据前后的

DOM 样式

和

props 

等

**10.3.4.** 卸载阶段

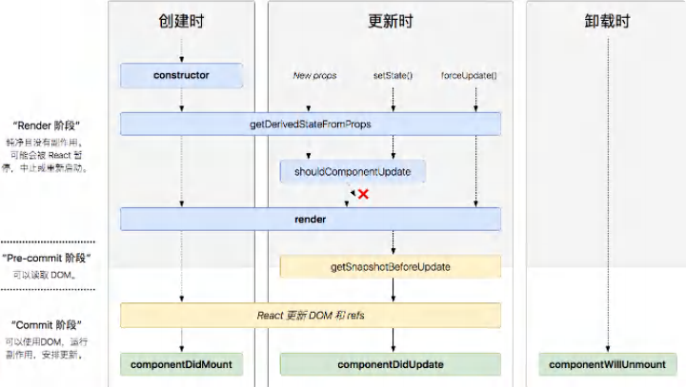
印客学院独家整理 , 盗版必究

**10.4.** componentwillunmount

此方法用于组件卸载前 , 清理一些注册是监听事件 , 或者取消订阅的网络请求等 一旦一个组件实例被卸载 , 其不会被再次挂载 , 而只可能是被重新创建

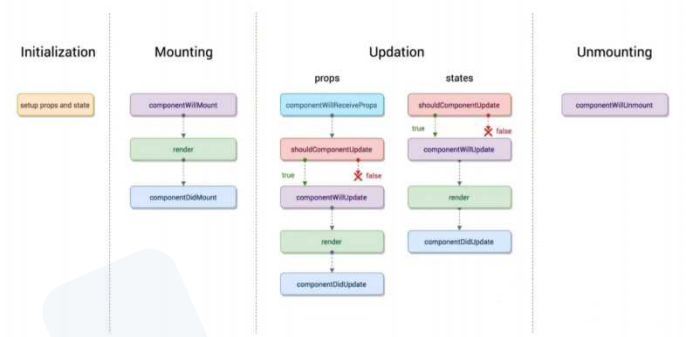
**10.5.** 总结

新版生命周期整体流程如下图所示 :



旧的生命周期流程图如下 :

印客学院独家整理 , 盗版必究



通过两个图的对比 , 可以发现新版的生命周期减少了以下三种方法: o componentwillMount

o componentwillReceiveprops

o componentwillupdate

其实这三个方法仍然存在 , 只是在前者加上了 UNSAFE\_  前缀 ,

,UNSAFE\_componentWillMount 并不像字面意思那样表示不安全 , 而是表示这些生命周期的代

如

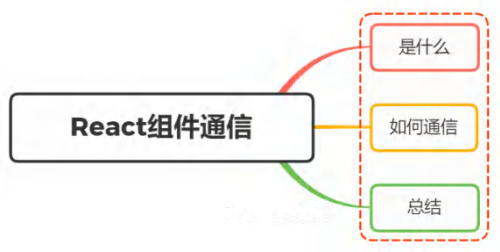
react 版本可能废除

码可能在未来的

同时也新增了两个生命周期函数: o getDerivedstateFromprops o getsnapshotBeforeupdate

**11.** React中组件之间如何通信?

印客学院独家整理 , 盗版必究



**11.1.** 是什么

我们将组件间通信可以拆分为两个词 :

• 组件 • 通信

组件是 vue  中最强大的功能之- , 同样组件化是 React  的核心思想

相比 vue  , React  的组件更加灵活和多样 , 按照不同的方式可以分成很多类型的组件

而通信指的是发送者通过某种媒体以某种格式来传递信息到收信者以达到某个目的 , 广义上 , 任何信息

的交通都是通信

组件间通信即指组件通过某种方式来传递信息以达到某个目的

**11.2.** 如何通信

组件传递的方式有很多种 , 根据传送者和接收者可以分为如下 :

 父组件向子组件传递

。 子组件向父组件传递

。 兄弟组件之间的通信

 父组件向后代组件传递  非关系组件传递

**11.2.1.** 父组件向子组件传递

由于 React  的数据流动为单向的 , 父组件向子组件传递是最常见的方式

印客学院独家整理 , 盗版必究

父组件在调用子组件的时候 , 只需要在子组件标签内传递参数 , 子组件通过 props  属性就能接收父组 件传递过来的参数

|  |
| --- |
| v JSX LO 复制代码 |
| function EmailInput(props) { return (  1  2  3  4  5  6  7  <label>  Email: <input value={props.email} /> </label>  ); }  8 |
| 9 const element = <EmailInput email="123124132@163.com" />; |



**11.2.2.** 子组件向父组件传递

子组件向父组件通信的基本思路是 , 父组件向子组件传一个函数 , 然后通过这个函数的回调 , 拿到子组 件传过来的值

父组件对应代码如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | JS | | | | | X LO 复制代码 |
| 1 | class Parents extends Component { | | | | |  |
| 2 | |  | | --- | |  | | constructor() { | | | |  |
| 3 | super(); | | | | |  |
| 4 | |  | | --- | |  | | this.state = { | | |  |  |
| 5 | price: 0 | | |  | |  |
| 6 | }; | | | | |  |
| 7 | } | | |  | |  |
| 8 |  | | | | |  |
| 9 | |  | | --- | |  | | getItemPrice(e) { |  | | |  |
| 10 | |  | | --- | |  | | this.setState({ |  | | |  |
| 11 | price: e | | | | |  |
| 12 | }); | | | | |  |
| 13 | } | | | | |  |
| 14 |  | | | | |  |
| 15 | |  | | --- | |  | | render() { | | | |  |
| 16 | return ( | | | | |  |
| 17 | <div> | | | | |  |
| 18 | <div>price: {this.state.price}</div> | | | | |  |
| 19 | {/\* 向子组件中传入-个函数 \*/} | | | | |  |
| 20 | <Child getPrice={this.getItemPrice.bind(this)} /> | | | | |  |
| 21 | </div> | | | | |  |
| 22 | ); | | | | |  |
| 23 | } | | | | |  |
| 24 | } | |  | | |  |

子组件对应代码如下 :

|  |
| --- |
| ·  JS  复制代码 |
| 1 class Child extends Component {  11 <button onClick={this.clickGoods.bind(this, 1000)}>goods2</button>   |  | | --- | |  | |  |   2 clickGoods(e) {  3 // 在此函数中传入值  4 this.props.getPrice(e); 5 }  6  7 render() {  8 return ( 9 <div>  10 <button onClick={this.clickGoods.bind(this, 100)}>goods1</button> |
| 12 </div> 13 );  14 } 15 } |

印客学院独家整理 , 盗版必究

**11.2.3.** 兄弟组件之间的通信

如果是兄弟组件之间的传递 , 则父组件作为中间层来实现数据的互通 , 通过使用父组件传递

|  |  |  |
| --- | --- | --- |
| JS, ca 复制代码 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Parent extends React.Component {   |  | | --- | |  | |  |   constructor(props) { super(props)  this.state = {count: 0}  }  setCount = () => {  this.setState({count: this.state.count + 1}) }  render() { return (  <div>  <SiblingA  count={this.state.count} /> | |
| 16  15  17  18  19  20  21 | onClick={this.setCount}  <SiblingB  />  </div> );  } } |  |





**11.2.4.** 父组件向后代组件传递 

父组件向后代组件传递数据是一件最普通的事情 , 就像全局数据一样

使用 context  提供了组件之间通讯的一种方式 , 可以共享数据 , 其他数据都能读取对应的数据 通过使用 React.createContext  创建一个 context 

|  |  |
| --- | --- |
| v Javascript | LO 复制代码 |
| 1 const PriceContext = React.createContext( 'price') |  |

 context  创建成功后 , 其下存在 Provider  组件用于创建数据源 , Consumer  组件用于接收数

据 , 使用实例如下:

Provider  组件通过 value  属性用于给后代组件传递数据:

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JS |  | LO 复制代码 |
| 1 <PriceContext.Provider value={100}> |  |  | |
| 2 </PriceContext.Provider> |  |  | |



如果想要获取 Provider  传递的数据 , 可以通过 Consumer  组件或者或者使用 contextType  属性

接收  对应分别如下  

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | JSX LO 复制代码 |
| 1  2  3  4  5 | class MyClass extends React.Component { static contextType = PriceContext;  render() {  let price = this.context; /\* 基于这个值进行渲染工作 \*/ | | |
| 6 } 7 } | | | |



组件:

Consumer

|  |  |  |  |
| --- | --- | --- | --- |
| v JS | | | X LO 复制代码 |
| 1 <PriceContext.Consumer> | | |  |
| 2 { /\*这里是-个函数\*/ } | | |  |
| 3 { | | |  |
| 4  5  6 | price => <div>price: {price}</div> }  </PriceContext.Consumer> |  |  |



**11.2.5.** 非关系组件传递



如果组件之间关系类型比较复杂的情况 , 建议将数据进行-个全局资源管理 , 从而实现通信 , 例如  red 。ux 关于 redux  的使用后续再详细介绍

**11.3.** 总结

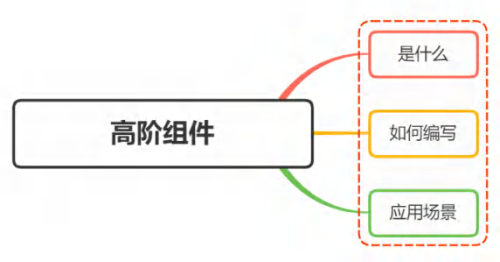
由于 React  是单向数据流 , 主要思想是组件不会改变接收的数据 , 只会监听数据的变化 , 当数据发生

变化时它们会使用接收到的新值 , 而不是去修改已有的值

因此 , 可以看到通信过程中 , 数据的存储位置都是存放在上级位置中

印客学院独家整理 , 盗版必究

**12.** 说说对高阶组件的理解? 应用场景?



**12.1.** 是什么

高阶函数 (Higher-order function) , 至少满足下列 一个条件的函数

• 接受一个或多个函数作为输入 。 输出 一个函数

在 React  中 ,

并

高阶组件即接受一个或多个组件作为参数并且返回 一个组件 , 本质也就是一个函数 ,

不是一个组件

|  |  |  |
| --- | --- | --- |
|  |  | JSX LO 复制代码 |
|  | | |
| 1 const EnhancedComponent = highOrderComponent(WrappedComponent); | | |

上述代码中 · 该函数接受  个组件 WrappedComponent  作为参数 · 返回加工过的新组件

omponent 

高阶组件的这种实现方式 · 本质上是  个装饰者设计模式





**12.2.** 如何编写

最基本的高阶组件的编写模板如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1 | import React, { Component } from 'react'; | |  |
| 2 |  | |  |
| 3  4  5  6  7  8  9  10 |  | export default (WrappedComponent) => {  return class EnhancedComponent extends  Component {  // do something  render() {  return <WrappedComponent />; }  } } |  |

 WrappedComponent  做一些你想要的操作 (比如操作 props , 提取 state , 给原始组件包裹其他元素等) , 从而加工出想要的组件

通过对传入的原始组件

EnhancedComponent

把通用的逻辑放在高阶组件中 , 对组件实现一致的处理 , 从而实现代码的复用

所以 , 高阶组件的主要功能是封装并分离组件的通用逻辑 , 让通用逻辑在组件间更好地被复用 但在使用高阶组件的同时 , 一般遵循一些约定 , 如下 :

 props 保持一致

 你不能在函数式 (无状态) 组件上使用 ref 属性 , 因为它没有实例  不要以任何方式改变原始组件 wrappedcomponent

• 透传不相关 props 属性给被包裹的组件 wrappedcomponent  不要再 render() 方法中使用高阶组件

 使用 compose 组合高阶组件  包装显示名字以便于调试

这里需要注意的是 , 高阶组件可以传递所有的 ,props 但是不能传递 ref 

如果向 -个高阶组件添加 refe  引用 , 那么 ref  指向的是最外层容器组件实例的 , 而不是被包裹的组 件 , 如果需要传递 refs  的话 , 则使用 ,React.forwardRef 如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX CO | | | | 复制代码 |
| 1 | function withLogging(WrappedComponent) { | | | |  |
| 2 | |  | | --- | |  | | class Enhance extends WrappedComponent { | | |  |
| 3 | |  | | --- | |  | | componentWillReceiveProps() { | | |  |
| 4 | console.log( 'Current props', this.props); | | |  |  |
| 5 | console.log( 'Next props', nextProps); | | | |  |
| 6 | } | | |  |  |
| 7 | |  | | --- | |  | | render() { |  |  |  |
| 8 | const {forwardedRef, ...rest} = this.props; | | | |  |
| 9 | // 把 forwardedRef 赋值给 ref | | | |  |
| 10 | return <WrappedComponent {...rest} ref={forwardedRef} | | | | />; |
| 11 | } | | | |  |
| 12 | }; | | | |  |
| 13 |  | | | |  |
| 14 | // React.forwardRef 方法会传入 props 和 ref 两个参数给其回调函数 | | | |  |
| 15 | // 所以这边的 ref 是由 React.forwardRef 提供的 | | | |  |
| 16 | |  | | --- | |  | | function forwardRef(props, ref) { | | |  |
| 17 | return <Enhance {...props} forwardRef={ref} /> | | | |  |
| 18 | } | | | |  |
| 19 |  | | | |  |
| 20 | return React.forwardRef(forwardRef); | | | |  |
| 21 | } | | | |  |
| 22 | const EnhancedComponent = withLogging(SomeComponent); | | | |  |



**12.3.** 应用场景



通过上面的了解 , 高阶组件能够提高代码的复用性和灵活性 , 在实际应用中 , 常常用于与核心业务无关 但又在多个模块使用的功能 , 如权限控制 、 日志记录 、数据校验 、异常处理 、统计上报等

举个例子 , 存在一个组件 , 需要从缓存中获取数据 , 然后渲染 。一般情况 , 我们会如下编写 :

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1 | import React, { Component } from 'react' | |  |
| 2 |  | |  |
| 3  4  5  6  7  8  9  10  11  12 |  | class MyComponent extends Component {  componentWillMount() {  let data = localStorage.getItem( 'data'); this.setState({data});  }  render() {  return <div>{this.state.data}</div> } |  |
| 13 | } | |  |

上述代码当然可以实现该功能 , 但是如果还有其他组件也有类似功能的时候 , 每个组件都需要重复写 

mponentWillMount 

中的代码 , 这明显是冗杂的

下面就可以通过高价组件来进行改写 , 如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX LO 复制代码 | | | | |
| 1 | import React, { Component } from 'react' | | | | |
| 2 |  | | | | |
| 3 | function withPersistentData(WrappedComponent) { | | | | |
| 4 | |  | | --- | |  | | return class extends Component { | | |  |
| 5 | |  | | --- | |  | | componentWillMount() { | | |  |
| 6 | let data = localStorage.getItem( 'data'); | | | |  |
| 7 | this.setState({data}); | | |  |  |
| 8 | } | | | | |
| 9 |  | | | | |
| 10 | |  | | --- | |  | | render() { |  | | |
| 11 | // 通过{...this.props} 把传递给当前组件的属性继续传递给被包装的组件WrappedCo | | | | |
| 12 | mponent  return <WrappedComponent data={this.state.data} {...this.props} /> | | | | |
| 13 | } | | | | |
| 14 | } | | | | |
| 15 | } | | | | |
| 16 |  | | | | |
| 17 | class MyComponent2 extends Component { | | | | |
| 18 | |  | | --- | |  | | render() { | | | |
| 19 | return <div>{this.props.data}</div> | | | | |
| 20 | } | | | | |
| 21 | } | | | | |
| 22 |  | | | | |
| 23 | const MyComponentWithPersistentData = withPersistentData(MyComponent2) | | | | |

再比如组件渲染性能监控 , 如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | JSX | | | | | | LO 复制代码 |
| 1 | class Home extends React.Component { | | | | | |  |
| 2 | |  | | --- | |  | | render() { | | | | |  |
| 3 | return (<h1>Hello World.</h1>); | | | | | |  |
| 4 | } | | |  | | |  |
| 5 | } |  | | | | |  |
| 6 | function withTiming(WrappedComponent) { | | | |  |  |  |
| 7 | |  | | --- | |  | | return class extends WrappedComponent { | | | |  |  |
| 8 | |  | | --- | |  | | constructor(props) { | | | | |  |
| 9 | super(props); | |  | | | |  |
| 10 | this.start = 0; | | | | | |  |
| 11 | this.end = 0; | | | | | |  |
| 12 | } | | | | | |  |
| 13 | |  | | --- | |  | | componentWillMount() { | | | | |  |
| 14 | super.componentWillMount && super.componentWillMou | | | | | | nt(); |
| 15 | this.start = Date.now(); | | | | | |  |
| 16 | } | | | | | |  |
| 17 | |  | | --- | |  | | componentDidMount() { | | | | |  |
| 18 | super.componentDidMount && super.componentDidMount | | | | | | (); |
| 19 | this.end = Date.now(); | | | | | |  |
| 20 | console.log(`${WrappedComponent.name} 组件渲染时间为 | | | | | | ${this.end |
|  | - this.start} ms`); | | | | | |  |
| 21 | } | | | | | |  |
| 22 | |  | | --- | |  | | render() { | | | | |  |
| 23  24  25  26  27  28 | return super.render(); }  }; } | | | | | |  |
| export default withTiming(Home); | | |  | | |

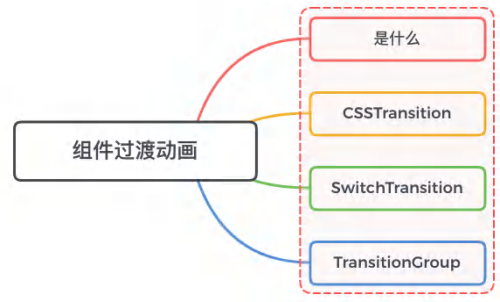




**13.**

在react中组件间过渡动画如何实现?

印客学院独家整理 , 盗版必究



**13.1.** 是什么

在日常开发中 , 页面切换时的转场动画是比较基础的一个场景

当 一个组件在显示与消失过程中存在过渡动画 , 可以很好的增加用户的体验

 react-motio

react-transition-group

在 react  中实现过渡动画效果会有很多种选择 , 如

n  ,Animated 以及原生的 CSS  都能完成切换动画

**13.2.** 如何实现

react-transition-group

enter

 ent

其为元素添加

是  种很好的解决方案 ,

在

中 ,



exit-active

exit



react

er-active 

这  系列勾子



可以帮助我们方便的实现组件的入场和离场动画 其主要提供了三个主要的组件 :

 CSSTransition : 在前端开发中 , 结合 Css 来完成过渡动画效果  switchTransition : 两个组件显示和隐藏切换时 , 使用该组件

 TransitionGroup: 将多个动画组件包裹在其中 , 一般用于列表中元素的动画

**13.2.1.** CSSTransition

其实现动画的原理在于 , 当 CSSTransition  的 in  属性置为 true  时 , CSSTransition  首先会 给其子组件加上 xxx-enter  、 xxx-enter-active  的 class  执行动画

印客学院独家整理 , 盗版必究

当动画执行结束后 , 会移除两个 ,class 并且添加 -enter-done  的 class 

所以可以利用这  点 , 通过 css  的 transition  属性 , 让元素在两个状态之间平滑过渡 , 从而得到 相应的动画效果

in

false

CSSTransition

xxx-exit和  xxx-exit-

会给子组件加上

属性置为

时,

当

active 

 -enter-don



class

然后开始执行动画 ,

当动画结束后 , 移除两个

然后添加

的

e 

class



的

class

如下例子 :

|  |  |  |
| --- | --- | --- |
|  |  | JSX COL 复制代码 |
| 1 export default class App2 extends React.PureComponent { 2  3 state = {show: true};  4  5 onToggle = () => this.setState({show: !this.state.show}); 6  7 render() {  8 const {show} = this.state; | | |
| 9 return (  10 <div className={ 'container'}>  11 <div className={'square-wrapper'}>  12 <CSSTransition  13 in={show}  14 timeout={500}  15 classNames={ 'fade'}  16 unmountOnExit={true} 17 >  18 <div className={'square'} />  19 </CSSTransition> 20 </div>  21 <Button onClick={this.onToggle}>toggle</Button>  22 </div> 23 );  24 } 25 } | | |

对应 css  样式如下:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | | | CSS | LO 复制代码 |
| 1 | .fade-enter { | | |  |  |
| 2 | opacity: 0; | | |  |  |
| 3 | transform: translateX(100%); | | |  |  |
| 4 | } | |  |  |  |
| 5 |  | | |  |  |
| 6 | .fade-enter-active { | |  |  |  |
| 7 | opacity: 1; | |  |  |  |
| 8 | transform: translateX(0); | | |  |  |
| 9 | transition: all 500ms; | |  |  |  |
| 10 | } |  | |  |  |
| 11 |  | | |  |  |
| 12 | .fade-exit { | | |  |  |
| 13 | opacity: 1; | | |  |  |
| 14 | transform: translateX(0); | | |  |  |
| 15 | } | | |  |  |
| 16 |  | | |  |  |
| 17 | .fade-exit-active { | | |  |  |
| 18 | opacity: 0; | | |  |  |
| 19 | transform: translateX(-100%); | | |  |  |
| 20 | transition: all 500ms; | | |  |  |
| 21 | } | | |  |  |

**13.2.2.** switchTransition

 SwitchTransition  可以完成两个组件之间切换的炫酷动画

比如有-个按钮需要在 on  和 off  之间切换 , 我们希望看到 on  先从左侧退出 , off  再从右侧进入

SwitchTransition  中主要有-个属性 ,mode 对应两个值: . in-out: 表示新组件先进入 , 旧组件再移除;

o out-in : 表示就组件先移除 , 新组建再进入

SwitchTransition  组件里面要有 ,CSSTransition 不能直接包裹你想要切换的组件

里面的 CSSTransition  组件不再像以前那样接受 in  属性来判断元素是何种状态 , 取而代之的是 k

ey  属性

下面给出 一个按钮入场和出场的示例 , 如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | JSX LO 复制代码 | | | | | |
| 1 | import { SwitchTransition, CSSTransition } from "react-transition-group"; | | | | | |
| 2 |  | | | | | |
| 3 | export default class SwitchAnimation extends PureComponent { | | | | | |
| 4 | |  | | --- | |  | | | constructor(props) { | | |  |
| 5 | super(props); | | | | |  |
| 6 |  | | | | | |
| 7 | |  | | --- | |  | | | this.state = { | |  |  |
| 8 | isOn: true | | | | | |
| 9 | } | | | |  | |
| 10 | } | | | | | |
| 11 |  | | | | | |
| 12 | |  | | --- | |  | | | render() { | | | |
| 13 | const {isOn} = this.state; | | | | | |
| 14 |  | | | | | |
| 15 | return ( | | | | | |
| 16 | <SwitchTransition mode="out-in"> | | | | | |
| 17 | <CSSTransition classNames="btn" | | | | | |
| 18 | timeout={500} | | | | | |
| 19 | key={isOn ? "on" : "off"}> | | | | | |
| 20 | { | | | | | |
| 21 | <button onClick={this.btnClick.bind(this)}> | | | | | |
| 22 | {isOn ? "on": "off"} | | | | | |
| 23 | </button> | | | | | |
| 24  25  26  27  28  29  30  31  32  33 | |  | | --- | |  | | } | | }  </CSSTransition> </SwitchTransition>  ) }  btnClick() {  this.setState({isOn: !this.state.isOn}) } | | |



文件对应如下:

css

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | CSS | LO 复制代码 |
| 1 | .btn-enter { | | | |  |  |
| 2 | transform: translate(100%, 0); | | | |  |  |
| 3 | opacity: 0; | | | |  |  |
| 4 | } | |  | |  |  |
| 5 |  | | | |  |  |
| 6 | .btn-enter-active { | |  | |  |  |
| 7 | transform: translate(0, 0); | | |  |  |  |
| 8 | opacity: 1; | | | |  |  |
| 9 | transition: all 500ms; | |  | |  |  |
| 10 | } |  | | |  |  |
| 11 |  | | | |  |  |
| 12 | .btn-exit { | | | |  |  |
| 13 | transform: translate(0, 0); | | | |  |  |
| 14 | opacity: 1; | | | |  |  |
| 15 | } | | | |  |  |
| 16 |  | | | |  |  |
| 17 | .btn-exit-active { | | | |  |  |
| 18 | transform: translate(-100%, 0); | | | |  |  |
| 19 | opacity: 0; | | | |  |  |
| 20 | transition: all 500ms; | | | |  |  |
| 21 | } | | | |  |  |

**13.2.3.** TransitionGroup

当有  组动画的时候 , 就可将这些 CSSTransition  放入到 -个 TransitionGroup 中来完成动画 同样 CSSTransition  里面没有 in  属性 , 用到了 key  属性

TransitionGroup  在感知 children  发生变化的时候 , 先保存移除的节点 , 当动画结束后才真正 移除

其处理方式如下 :

 插入的节点 , 先渲染dom , 然后再做动画

 删除的节点 , 先做动画 , 然后再删除dom 如下 :

印客学院独家整理 , 盗版必究

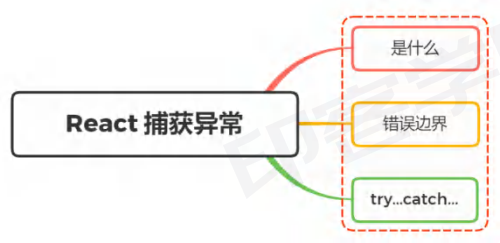
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX LO 复制代码 | | | | |
| 1 | import React, { PureComponent } from 'react' | | | | |
| 2 | import { CSSTransition, TransitionGroup } from 'react-transition-group'; | | | | |
| 3 |  | | | | |
| 4 | - export default class GroupAnimation extends PureComponent { | | | | |
| 5 | |  | | --- | |  | | constructor(props) { | |  | |
| 6 | super(props); | | |  | |
| 7 |  | | | |  |
| 8 | |  | | --- | |  | | this.state = { | | | |
| 9 | friends: [] | |  | | |
| 10 | } | |  | | |
| 11 | } | | | | |
| 12 |  | | | | |
| 13 | |  | | --- | |  | | render() { | | | |
| 14 | return ( | | | | |
| 15 | <div> | | | | |
| 16 | <TransitionGroup> | | | | |
| 17 | { | | | | |
| 18 | |  | | --- | |  | | this.state.friends.map((item, index) => { | | | |
| 19 | return ( | | | | |
| 20 | <CSSTransition classNames="friend" timeout={300} key={inde x}> | | | | |
| 21 | <div>{item}</div> | | | | |
| 22 | </CSSTransition> | | | | |
| 23 | ) | | | | |
| 24 | }) | | | | |
| 25 | } | | | | |
| 26 | </TransitionGroup> | | | | |
| 27 | <button onClick={e => this.addFriend()}>+friend</button> | | | | |
| 28 | </div> | | | | |
| 29 | ) | | | | |
| 30 | } | | | | |
| 31 |  | | | | |
| 32 | |  | | --- | |  | | addFriend() { | | | |
| 33 | |  | | --- | |  | | this.setState({ | | | |
| 34 | friends: [...this.state.friends, "code rwhy"] | | | | |
| 35 | }) | | | | |
| 36 | } | | | | |
| 37 | } | | | | |

对应 css  如下:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | CSS | LO 复制代码 |
| 1 | .friend-enter { | | | |  |  |
| 2 | transform: translate(100%, 0); | | | |  |  |
| 3 | opacity: 0; | | | |  |  |
| 4 | } | |  | |  |  |
| 5 |  | | | |  |  |
| 6 | .friend-enter-active { | |  | |  |  |
| 7 | transform: translate(0, 0); | | |  |  |  |
| 8 | opacity: 1; | | | |  |  |
| 9 | transition: all 500ms; | | | |  |  |
| 10 | } |  | | |  |  |
| 11 |  | | | |  |  |
| 12 | .friend-exit { | | | |  |  |
| 13 | transform: translate(0, 0); | | | |  |  |
| 14 | opacity: 1; | | | |  |  |
| 15 | } | | | |  |  |
| 16 |  | | | |  |  |
| 17 | .friend-exit-active { | | | |  |  |
| 18 | transform: translate(-100%, 0); | | | |  |  |
| 19 | opacity: 0; | | | |  |  |
| 20 | transition: all 500ms; | | | |  |  |
| 21 | } | | | |  |  |

**14.** 说说你在React项目是如何捕获错误的?



**14.1.** 是什么

错误在我们日常编写代码是非常常见的

印客学院独家整理 , 盗版必究

React

举个例子 , 在 react  项目中去编写组件内 JavaScript  代码错误会导致 的内部状态被破

坏 , 导致整个应用崩溃 , 这是不应该出现的现象

作为 -个框架 , react  也有自身对于错误的处理的解决方案



**14.2.** 如何做



为了解决出现的错误导致整个应用崩溃的问题 , react16  引用了错误边界新的概念

错误边界是一种 React  组件 , 这种组件可以捕获发生在其子组件树任何位置的 错

JavaScript 

误 , 并打印这些错误 , 同时展示降级 ,UI 而并不会渲染那些发生崩溃的子组件树

错误边界在渲染期间 、生命周期方法和整个组件树的构造函数中捕获错误 形成错误边界组件的两个条件 :

。 使用了 static getDerivedstateFromError() • 使用了 componentDidcatch()

 static getDerivedStateFromError() 

抛出错误后 , 请使用

渲染备用 UI , 使用 

DidCatch()  打印错误信息 , 如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX | | | | LO 复制代码 |
| 1 | class ErrorBoundary extends React.Component { | | | |  |
| 2 | |  | | --- | |  | | constructor(props) { | | |  |
| 3 | super(props); | | | |  |
| 4 | this.state = { hasError: false }; | | |  |  |
| 5 | } | | |  |  |
| 6 |  | | | |  |
| 7 | |  | | --- | |  | |  | |  |  |
| 8 | // 更新 state 使下-次渲染能够显示降级后的 UI  static getDerivedStateFromError(error)  { | | | |  |
| 9 | return { hasError: true }; | | | |  |
| 10 | } | |  | |  |
| 11 |  | | | |  |
| 12 | |  | | --- | |  | | componentDidCatch(error, errorInfo) { | | |  |
| 13 | // 你同样可以将错误日志上报给服务器 | | | |  |
| 14 | logErrorToMyService(error, errorInfo); | | | |  |
| 15 | } | | | |  |
| 16 |  | | | |  |
| 17 | |  | | --- | |  | | render() { | | |  |
| 18 | |  | | --- | |  | | if (this.state.hasError) { | | |  |
| 19 | // 你可以自定义降级后的 UI 并渲染 | | | |  |
| 20 | return <h1>Something went wrong.</h1>; | | | |  |
| 21 | } | | | |  |
| 22 |  | | | |  |
| 23 | return this.props.children; | | | |  |
| 24  25 | } } | |  | |  |

然后就可以把自身组件的作为错误边界的子组件 , 如下:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | JSX | LO 复制代码 |
| 1 <ErrorBoundary> |  |  |  |
| 2 <MyWidget /> | |  |  |
| 3 </ErrorBoundary> | |  |  |

下面这些情况无法捕获到异常 :

• 事件处理

 异步代码

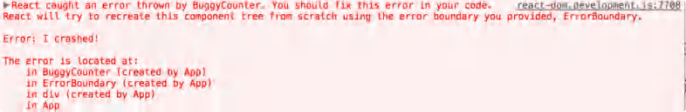
。 服务端渲染

。 自身抛出来的错误

在 react 16  版本之后 , 会把渲染期间发生的所有错误打印到控制台

印客学院独家整理 , 盗版必究

除了错误信息和 Javascript 栈外 , React 16 还提供了组件栈追踪 。现在你可以准确地查看发生在组件 树内的错误信息:



可以看到在错误信息下方文字中存在一个组件栈 , 便于我们追踪错误

对于错误边界无法捕获的异常 , 如事件处理过程中发生问题并不会捕获到 , 是因为其不会在渲染期间触 发 , 并不会导致渲染时候问题

js 



这种情况可以使用

语法 , 如下:

的

try...catch...

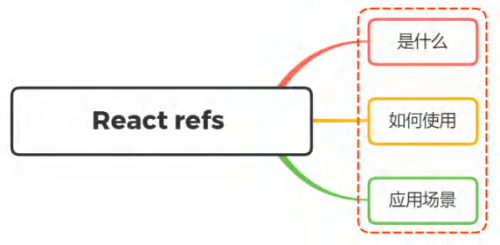
|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX CO | | 复制代码 |
| 1 | class MyComponent extends React.Component { | |  |
| 2 | |  | | --- | |  | | constructor(props) { |  |
| 3 | super(props); | |  |
| 4 | this.state = { error: null }; | |  |
| 5 | this.handleClick = this.handleClick.bind(this); | |  |
| 6 | } | |  |
| 7 |  | |  |
| 8 | |  | | --- | |  | | handleClick() { |  |
| 9 | |  | | --- | |  | | try { |  |
| 10 | // 执行操作 如有错误则会抛出 | |  |
| 11 | |  | | --- | |  | | } catch (error) { |  |
| 12 | this.setState({ error }); | |  |
| 13 | } | |  |
| 14 | } | |  |
| 15 |  | |  |
| 16 | |  | | --- | |  | | render() { |  |
| 17 | |  | | --- | |  | | if (this.state.error) { |  |
| 18 | return <h1>Caught an error.</h1> | |  |
| 19 | } | |  |
| 20 | return <button onClick={this.handleClick}>Click Me</button> | |  |
| 21 | } | |  |
| 22 | } | |  |

除此之外还可以通过监听 onerror  事件

|  |  |
| --- | --- |
| v Javascript | LO 复制代码 |
| 1 window.addEventListener( 'error', function(event) { ... }) |  |

印客学院独家整理 , 盗版必究

**15.** 说说对React refs 的理解? 应用场景?



**15.1.** 是什么

Refs  在计算机中称为弹性文件系统 (英语 : Resilient File system , 简称ReFS)

 Reac

 React  中的 Refs  提供了  种方式 , 允许我们访问 DOM  节点或在 render  方法中创建的 t  元素

本质为 ReactDOM.render()  返回的组件实例 , 如果是渲染组件则返回的是组件实例 , 如果渲

dom

则返回的是具体的 dom  节点

染

**15.2.** 如何使用



创建 ref  的形式有三种 :

传入字符串 , 使用时通过 this. refs. 传入的字符串的格式获取对应的元素

传入对象 , 对象是通过 React. createRef() 方式创建出来 , 使用时获取到创建的对象中存在 current 属性就是对应的元素

传入函数 , 该函数会在 DOM 被挂载时进行回调 , 这个函数会传入一个 元素对象 , 可以自己保存 , 使用时 , 直接拿到之前保存的元素对象即可

传入hook, hook是通过 useRef() 方式创建 , 使用时通过生成hook对象的 current 属性就是对应的 元素

**15.2.1.** 传入字符串

只需要在对应元素或组件中 ref  属性

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX | | | | LO 复制代码 |
| 1 | class MyComponent extends React.Component { | | | |  |
| 2 | |  | | --- | |  | | | constructor(props) { | |  |
| 3  4  5  6  7  8  9 | |  | | --- | |  | | } | | super(props);  this.myRef = React.createRef(); }  render() {  return <div ref="myref" />; } |  |

访问当前节点的方式如下 :

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 this.refs.myref.innerHTML = "hello"; |  |  |

**15.2.2.** 传入对象

refs  通过 React.createRef()  创建 , 然后将 ref  属性添加到 React  元素中 , 如下 :

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1  2  3  4  5  6  7  8  9 |  | class MyComponent extends React.Component {  constructor(props) { super(props);  this.myRef = React.createRef(); }  render() {  return <div ref={this.myRef} />; }  } |  |

当 ref  被传递给 render  中的元素时  对该节点的引用可以在 ref  的 current  属性中访问

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 const node = this.myRef.current; |  |  |



**15.2.3.** 传入函数





ref

|  |  |
| --- | --- |
|  | |
| 1 | class MyComponent extends React.Component { |
| 2 | constructor(props) { |
| 3 | super(props); |
| 4 | this.myRef = React.createRef(); |
| 5 | } |
| 6 | render() { |
| 7 | return <div ref={element => this.myref = element} />; |
| 8 | } |
| 9 | } |

 ref

|  |
| --- |
|  |
| 1 const node = this.myref |

**15.2.4.** 





ref



 React.createRef   



useRef



|  |
| --- |
|  |
| 1  function App(props) {  2 const myref = useRef()  3 return ( 4 <>  5 <div ref={myref}></div>  6 </> |
| 7 ) 8 } |

 ref hook current

|  |
| --- |
|  |
| 1 const node = myref.current; |



 r

 ref 

 ref 

 HTML 

ef 

印客学院独家整理 , 盗版必究

注意的是 , 不能在函数组件上使用 ref  属性 , 因为他们并没有实例

**15.3.** 应用场景

在某些情况下 , 我们会通过使用 refs  来更新组件 , 但这种方式并不推荐 更 情况我们是通过  prop

s  与 state  的方式进行去重新渲染子元素

过多使用 ,refs 会使组件的实例或者是 DOM  结构暴露 , 违反组件封装的原则

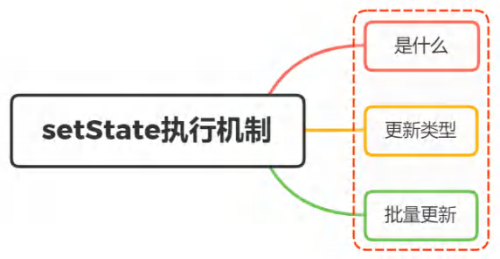
例如 , 避免在 Dialog  组件里暴露 open()  和 close()  方法 , 最好传递 isOpen  属性 但下面的场景使用 refs  非常有用 :

 对Dom元素的焦点控制 、 内容选择 、控制  对Dom元素的内容设置及媒体播放

。 对Dom元素的操作和对组件实例的操作 • 集成第三方 DOM 库



**16.** 说说 React中的setstate执行机制



**16.1.** 是什么



一个组件的显示形态可以由数据状态和外部参数所决定 , 而数据状态就是

state

当需要修改里面的值的状态需要通过调用 setState  来改变 , 从而达到更新组件内部数据的作用 如下例子

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX Co 复制代码 | | | | |
| 1 | import React, { Component } from 'react' | | | | |
| 2 |  | | | | |
| 3 | export default class App extends Component { | | | | |
| 4 | |  | | --- | |  | | constructor(props) { | | |  |
| 5 | super(props); | | | |  |
| 6 |  | | | | |
| 7 | |  | | --- | |  | | this.state = { | |  |  |
| 8 | message: "Hello World" | | | | |
| 9 | } | |  | | |
| 10 | } | | | | |
| 11 |  | | | | |
| 12 | |  | | --- | |  | | render() { | | | |
| 13 | return ( | | | | |
| 14 | <div> | | | | |
| 15 | <h2>{this.state.message}</h2> | | | | |
| 16 | <button onClick={e => this.changeText()}>面试官系列</button | | | | |
|  | > | | | | |
| 17 | </div> | | | | |
| 18 | ) | | | | |
| 19 | } | | | | |
| 20 |  | | | | |
| 21 | |  | | --- | |  | | changeText() { | | | |
| 22 | |  | | --- | |  | | this.setState({ | | | |
| 23  24  25  26 | message: "JS每日 一题 " })  } } | | | |  |

通过点击按钮触发 onclick  事件 , 执行 this.setState  方法更新 state  状态  然后重新执行  r

ender  函数 , 从而导致页面的视图更新 如果直接修改 state  的状态 , 如下 :

|  |  |  |
| --- | --- | --- |
|  | JS | X LO 复制代码 |
| 1  changeText() { |  |  |
| 2 this.state.message = "你好啊 ,李银河 "; |  |  |
| 3 } |  |  |

我们会发现页面并不会有任何反应 , 但是 state  的状态是已经发生了改变

 Pr

这是因为 React  并不像 vue2  中调用 Object.defineProperty  数据响应式或者 Vue3  调用 oxy  监听数据的变化

必须通过 setState  方法来告知 react  组件 state  已经发生了改变

印客学院独家整理 , 盗版必究

关于 state  方法的定义是从 React.Component  中继承 , 定义的源码如下:

|  |  |  |
| --- | --- | --- |
|  | Javascript B 复制代码 | |
| 1  2  3  4  5  6  7  8  9  10 | |  | | --- | |  | | Component.prototype.setState = function(partialState, callback) { invariant(  typeof partialState === 'object' ||  typeof partialState === 'function' || partialState == null,  'setState(...): takes an object of state variables to update or a ' +  'function which returns an object of state variables. ', );  this.updater.enqueueSetState(this, partialState, callback, 'setState'); }; |

 setState  第一个参数可以是  个对象 , 或者是  个函数 , 而第二个参数是  个回调 函数 , 用于可以实时的获取到更新之后的数据

从上面可以看到

**16.2.** 更新类型

在使用 setState  更新数据的时候 , setState  的更新类型分成: • 异步更新

 同步更新

**16.2.1.** 异步更新

先举出 一个例子 :

|  |  |  |
| --- | --- | --- |
|  | JSX LO 复制代码 | |
| 1 | changeText() { | |
| 2 | |  | | --- | |  | | this.setState({ |
| 3 | message: "你好啊 " | |
| 4 | }) | |
| 5  6 | console.log(this.state.message); // Hello World } | |

从上面可以看到 , 最终打印结果为 ,Helloworld 并不能在执行完 setState  之后立马拿到最新 的 state  的结果

如果想要立刻获取更新后的值 , 在第二个参数的回调中更新后会执行

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX | | | | LO 复制代码 |
| 1 | changeText() { | | | |  |
| 2 | |  | | --- | |  | | | this.setState({ | |  |
| 3  4  5  6  7 | |  | | --- | |  | | } | | message: "你好啊 " }, () => {  console.log(this.state.message); // 你好啊 }); |  |



**16.2.2.** 同步更新



setTimeout 中更新的例子 :

同样先给出 一个在

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | |  | JSX CO | 复制代码 |
| 1 | changeText() { | |  |  |  |
| 2 | |  | | --- | |  | | setTimeout(() => { |  |  |  |
| 3 | |  | | --- | |  | | this.setState({ |  |  |  |
| 4 | message: "你好啊 | |  |  |  |
| 5 | }); | |  |  |  |
| 6 | console.log(this.state.message); // | | 你好啊 |  |  |
| 7 | }, 0); | |  |  |  |
| 8 | } | |  |  |  |

上面的例子中 , 可以看到更新是同步 再来举  个原生 DOM  事件的例子 :

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | JS | X COL 复制代码 |
| 1 | componentDidMount() { |  |  |
| 2 | const btnEl = document.getElementById("btn"); |  |  |
| 3 | btnEl.addEventListener( 'click', () => { |  |  |
| 4 | this.setState({ |  |  |
| 5 | message: "你好啊 ,李银河 " |  |  |
| 6 | }); |  |  |
| 7 | console.log(this.state.message); // 你好啊 ,李银河 |  |  |
| 8 | }) |  |  |
| 9 | } |  |  |

**16.2.3.** 小结

• 在组件生命周期或React合成事件中 , setstate是异步

印客学院独家整理 , 盗版必究

• 在setTimeout或者原生dom事件中 , setstate是同步

**16.2.4.** 批量更新

同样先给出 一个例子 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | JSX | LO 复制代码 |
| 1  2  3  4  5  6  7  8  9  10  11  12 | handleClick = () => {   |  | | --- | |  | |  |   this.setState({  count: this.state.count + 1, })  console.log(this.state.count) // 1  this.setState({  count: this.state.count + 1, })  console.log(this.state.count) // 1  this.setState({ | | | |
| 13 count: this.state.count + 1, 14 })  15 console.log(this.state.count) // 1 16 } | | | | |

点击按钮触发事件 , 打印的都是 1 , 页面显示 count  的值为 2

对同 -个值进行多次 setState  , 结果

的批量更新策略会对其进行覆盖 

取最后一次的执行

setState

上述的例子 , 实际等价于如下 :

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 | Object.assign( | |
| 2 | previousState, | |
| 3 | {index: state.count+ 1}, | |
| 4 | {index: state.count+ 1}, | |
| 5 | ... | |
| 6 | ) | |

由于后面的数据会覆盖前面的更改 , 所以最终只加了 一次

如果是下-个 state  依赖前-个 state  的话 , 推荐给 -setState个参数传入- 个 ,function 如下:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | | JS | | X B 复制代码 |
| 1 | onClick = () => { | |  | |  |
| 2 | |  | | --- | |  | | this.setState((prevState, props) | => { | |  |
| 3  4  5  6  7 | |  | | --- | |  | | return {count: prevState.count });  this.setState((prevState, props) return {count: prevState.count  }); | + 1};  => { + 1}; |  |  |
| 8 | } | |  | |  |

dom 事件中 , 由于是同步的操作 , 所以并不会进行覆盖现象

而在

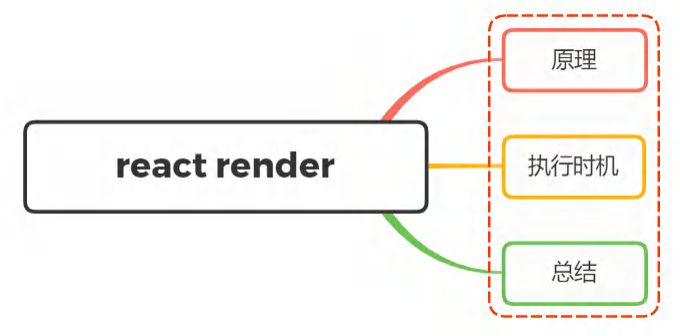
setTimeout 或者原生





**17.** 说说React render方法的原理? 在什么时候会被触

发?



**17.1.** 原理

首先 , render  函数在 react  中有两种形式:

在类组件中 , 指的是 render  方法 :

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
| v JSX | | | LO 复制代码 |
| 1  class Foo extends React.Component { | | |  |
| 2  render() { | | |  |
| 3  4  5 | return <h1> Foo </h1>; }  } |  |  |

在函数组件中 , 指的是函数组件本身 :

jsx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | Javascript | LO 复制代码 |
| 1 | function Foo() { | | | | | |
| 2  3 | return  } | <h1> | Foo | </h1>; | | |

babel 编译后就会转化成我们熟悉的

js 

jsx 通过

在

render  中 , 我们会编写

如

格式 ,

下:

|  |  |  |  |
| --- | --- | --- | --- |
|  | JS | | X LO 复制代码 |
| 1 | return ( | |  |
| 2 | <div className= 'cn'> | |  |
| 3 | <Header> hello </Header> | |  |
| 4 | <div> start </div> | |  |
| 5  6  7 | Right Reserve  </div> ) |  |  |



编译后:

babel

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | Javascript | | LO 复制代码 |
| 1 | return ( | | |  | |  |
| 2 | React.createElement( | | |  | |  |
| 3 | 'div', | | |  | |  |
| 4 | |  | | --- | |  | | { | |  | |  |
| 5 | className : 'cn' | | |  | |  |
| 6 | }, | | |  |  |  |
| 7 | React.createElement( | | |  |  |  |
| 8 | Header, | | |  | |  |
| 9 | null, | |  |  | |  |
| 10 | 'hello' | | |  | |  |
| 11 | ), | | |  | |  |
| 12 | React.createElement( | | |  | |  |
| 13 | 'div', | | |  | |  |
| 14 | null, | | |  | |  |
| 15 | 'start' | | |  | |  |
| 16 | ), | | |  | |  |
| 17 | 'Right Reserve' | | |  | |  |
| 18 | ) | | |  | |  |
| 19 | ) | | |  | |  |

从名字上来看 , createElement  方法用来元素的

在 react  中 , 这个元素就是虚拟 DOM  树的节点 , 接收三个参数 :

 type : 标签

o attributes: 标签属性 , 若无则为null o children : 标签的子节点

这些虚拟 DOM  树最终会渲染成真实 DOM 

在 render  过程中 , React  将新调用的 render  函数返回的树与旧版本的树进行比较 , 这  步是 决定如何更新 DOM  的必要步骤 , 然后进行 diff  比较 , 更新 DOM  树

**17.2.** 触发时机

render  的执行时机主要分成了两部分 :

• 类组件调用 setstate 修改状态

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JS | | | | X LO 复制代码 |
| 1 | class Foo extends React.Component { | | | |  |
| 2 | state = { count: 0 }; | | | |  |
| 3 |  | | | |  |
| 4 | |  | | --- | |  | | increment = () => { | |  |  |
| 5 | const { count } = this.state; | | |  |  |
| 6 |  | | | |  |
| 7 | const newCount = count < 10 ? count + 1 : count; | | | |  |
| 8 |  | | | |  |
| 9 | this.setState({ count: newCount }); | | | |  |
| 10 | }; | |  | |  |
| 11 |  | | | |  |
| 12 | |  | | --- | |  | | render() { | | |  |
| 13 | const { count } = this.state; | | | |  |
| 14 | console.log("Foo render"); | | | |  |
| 15 |  | | | |  |
| 16 | return ( | | | |  |
| 17 | <div> | | | |  |
| 18 | <h1> {count} </h1> | | | |  |
| 19 | <button onClick={this.increment}>Increment</button> | | | |  |
| 20 | </div> | | | |  |
| 21 | ); | | | |  |
| 22 | } | | | |  |
| 23 | } | | | |  |

点击按钮则调用 setState  方法 , 无论 count  发生变化辩护 , 控制台都会输出 ,Foorender 证 明 render  执行了

 函数组件通过 useState hook  修改状态

印客学院独家整理 , 盗版必究

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | JS | X LO 复制代码 |
| 1 | function Foo() { | |  |  |
| 2 | const [count, setCount] = useState(0); | |  |  |
| 3 |  | |  |  |
| 4 | |  | | --- | |  | | function increment() { |  |  |
| 5 | const newCount = count < 10 ? count + 1 : count; | |  |  |
| 6 | setCount(newCount); | |  |  |
| 7 | } | |  |  |
| 8 |  | |  |  |
| 9 | console.log("Foo render"); | |  |  |
| 10 |  | |  |  |
| 11 | return ( | |  |  |
| 12 | <div> | |  |  |
| 13 | <h1> {count} </h1> | |  |  |
| 14 | <button onClick={increment}>Increment</button> | |  |  |
| 15 | </div> | |  |  |
| 16 | ); | |  |  |
| 17 | } | |  |  |

函数组件通过 useState  这种形式更新数据 , 当数组的值不发生改变了 , 就不会触发 render 

• 类组件重新渲染

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Javascript | | | | LO 复制代码 |
| 1 | class App extends React.Component { | | | |  |
| 2 | state = { name: "App" }; | | | |  |
| 3 | |  | | --- | |  | | render() { | | |  |
| 4 | return ( | | |  |  |
| 5 | <div className="App"> | | |  |  |
| 6 | <Foo /> | | |  |  |
| 7 | <button onClick={() => this.setState({ name: "App" })}> | | | |  |
| 8 | Change name | | | |  |
| 9 | </button> | |  | |  |
| 10 | </div> | |  | |  |
| 11 | ); | | | |  |
| 12 | } | | | |  |
| 13 | } | | | |  |
| 14 |  | | | |  |
| 15 | function Foo() { | | | |  |
| 16 | console.log("Foo render"); | | | |  |
| 17 |  | | | |  |
| 18 | return ( | | | |  |
| 19 | <div> | | | |  |
| 20 | <h1> Foo </h1> | | | |  |
| 21 | </div> | | | |  |
| 22 | ); | | | |  |
| 23 | } | | | |  |



App

 rend

Change name

Foo 具体实现是什么 , 都会被重新

按钮 , 不管

组件内的

只要点击了

er 渲染

函数组件重新渲染

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | JS | X LO 复制代码 |
| 1 | function App(){ | | | |  |  |
| 2 | const [name,setName] = useState( 'App') | | | |  |  |
| 3 |  | | | |  |  |
| 4 | return ( | | | |  |  |
| 5 | <div className="App"> | | |  |  |  |
| 6 | <Foo /> | | | |  |  |
| 7 | <button onClick={() => setName("aaa")}> | | | |  |  |
| 8 | { name } | | | |  |  |
| 9 | </button> | |  | |  |  |
| 10 | </div> |  | | |  |  |
| 11 | ) | | | |  |  |
| 12 | } | | | |  |  |
| 13 |  | | | |  |  |
| 14 | function Foo() { | | | |  |  |
| 15 | console.log("Foo render"); | | | |  |  |
| 16 |  | | | |  |  |
| 17 | return ( | | | |  |  |
| 18 | <div> | | | |  |  |
| 19 | <h1> Foo </h1> | | | |  |  |
| 20 | </div> | | | |  |  |
| 21 | ); | | | |  |  |
| 22 | } | | | |  |  |

可以发现 , 使用 useState  来更新状态的时候 , 只有首次会触发 ,Foorender 后面并不会导致 

o render 



**17.3.** 总结

render  函数里面可以编写 ,JSX 转化成 createElement  这种形式 , 用于生成虚拟 ,DOM 最终 转化成真实 DOM 

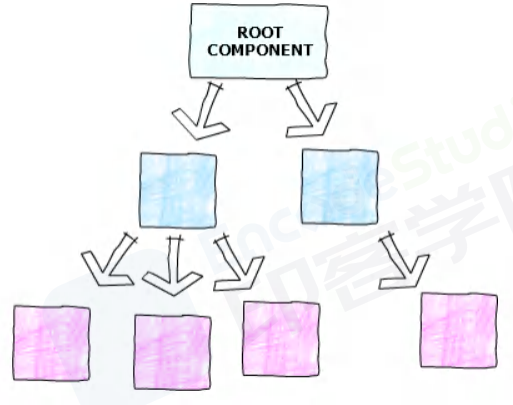
在 React  中 , 类组件只要执行了 setState  方法 , 就一定会触发 render  函数执行 , 函数组件 使用 useState  更改状态不  定导致重新 render 

组件的 props  改变了 , 不  定触发 render  函数的执行 , 但是如果 props  的值来自于父组件或 者祖先组件的 state 

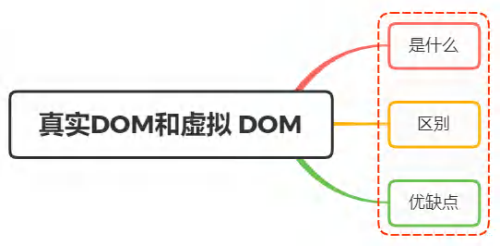
在这种情况下 , 父组件或者祖先组件的 state  发生了改变 , 就会导致子组件的重新渲染

所以 , -旦执行了 setState  就会执行 render  方法 , useState  会判断当前值有无发生改变确定 是否执行 render  方法 , -旦父组件发生渲染 , 子组件也会渲染

印客学院独家整理 , 盗版必究



**18.** 说说 Real DOM 和 virtual DOM 的区别? 优缺点?



**18.1.** 是什么

Real DOM, 真实 ,DOM 意思为文档对象模型 , 是一个结构化文本的抽象 , 在页面渲染出的每一个结

点都是  个真实 DOM  结构 , 如下 :

印客学院独家整理 , 盗版必究



 ,VirtualDom 本质上是以 JavaScript  对象形式存在的对 DOM  的描述

创建虚拟 目DOM 的就是为了更好将虚拟的节点渲染到页面视图中 , 虚拟 DOM  对象的节点与真实  D

OM 

照应

的属性 — —

JSX

在 React  中 , 的 DOM  结构

是其一大特性 , 可以让你在

XML  的方式去直接声明界面

JS  中通过使用

|  |  |
| --- | --- |
|  | JSX COL 复制代码 |
| 1  2 | // 创建 h1 标签 , 右边千万不能加引号  const vDom = <h1>Hello World</h1>; |
| 3 | // 找到 <div id="root"></div> 节点 |
| 4 | const root = document.getElementById("root"); |
| 5 | // 把创建的 h1 标签渲染到 root 节点上 |
| 6 | ReactDOM.render(vDom, root); |

上述中 , ReactDOM.render()  用于将你创建好的虚拟 DOM  节点插入到某个真实节点上 , 并渲染 到页面上

 JSX  实际是一种语法糖 , 在使用过程中会被 babel  进行编译转化成 JS  代码 , 上述 VDOM  转 化为如下:

|  |  |  |
| --- | --- | --- |
|  | JSX | LO 复制代码 |
| 1 const vDom = React.createElement( |  |  |
| 2 'h1' |  |  |
| 3 { className: 'hClass', id: 'hId ' }, |  |  |
| 4 'hello world' |  |  |
| 5 ) |  |  |

可以看到 · JSX  就是为了简化直接调用 React.createElement()  方法 :



• 第一个参数是标签名 , 例如 h1 、 span 、table …

。 第二个参数是个对象 , 里面存着标签的 一些属性 , 例如 id 、class 等

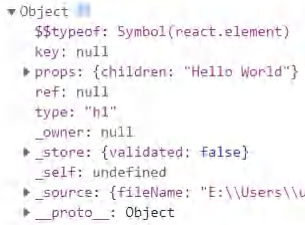
。 第三个参数是节点中的文本



VDOM

通过 ,console.log(VDOM) 则能够得到虚拟 消息

印客学院独家整理 , 盗版必究



所以可以得到 , JSX  通过 babel  的方式转化成 React.createElement  执行 , 返回值是  个 对象 , 也就是虚拟 DOM 

**18.2.** 区别

两者的区别如下 :

 虚拟 DOM 不会进行排版与重绘操作 , 而真实 DOM 会频繁重排与重绘

• 虚拟 DOM 的总损耗是 "虚拟 DOM 增删改+真实 DOM 差异增删改+排版与重绘" , 真实 DOM 的总 损耗是 " 真实 DOM 完全增删改+排版与重绘"

传统的原生 api  或 jQuery  去操作 DOM  时 , 浏览器会从构建 DOM  树开始从头到尾执行  遍

流程  

当你在 — 次操作时  需要更新 。个 DOM  节点  浏览器没这么智能  收到第 个更新 DOM  请求后  并不知道后续还有 9 次更新操作 , 因此会马上执行流程 , 最终执行 10 次流程

而通过 ,VNode 同样更新 10 个 DOM  节点 , 虚拟 DOM  不会立即操作 ,DOM 而是将这 10 次更 新的 diff  内容保存到本地的 -个 js  对象中 , 最终将这个 js  对象一次性 attach  到 DOM  树上 , 避免大量的无谓计算

**18.3.** 优缺点

真实 DOM  的优势 :

。 易用 缺点 :

 效率低 , 解析速度慢 , 内存占用量过高

。 性能差 : 频繁操作真实 DOM , 易于导致重绘与回流

印客学院独家整理 , 盗版必究

使用虚拟 DOM  的优势如下 :

 简单方便: 如果使用手动操作真实 DOM  来完成页面 , 繁琐又容易出错 , 在大规模应用下维护起 来也很困难

 性能方面 : 使用 virtual DOM , 能够有效避免真实 DOM 数频繁更新 , 减少多次引起重绘与回流 ,

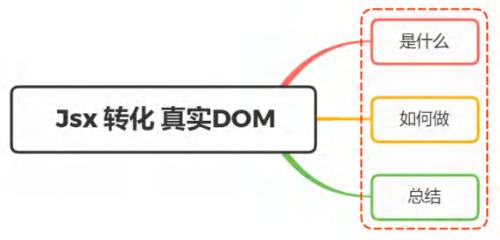
提高性能

 跨平台 : REact 借助虚拟 DOM , 带来了跨平台的能力 , 一套代码多端运行 缺点 :

• 在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化

 首次渲染大量 DOM 时 , 由于多了 一层虚拟 DOM 的计算 , 速度比正常稍慢

**19.** 说说React Jsx转换成真实DOM过程?



**19.1.** 是什么



React

react  通过将组件编写的 JSX  映射到屏幕 , 以及组件中的状态发生了变化之后 会将这些

「变化」 更新到屏幕上

在前面文章了解中 , JSX  通过 babel  最终转化成 React.createElement  这种形式 , 例如 :

|  |  |  |
| --- | --- | --- |
|  | JSX CO | 复制代码 |
| 1 <div> |  |  |
| 2 < img s rc="avatar.png" className="profile" /> |  |  |
| 3 <Hello /> |  |  |
| 4 </div> |  |  |

印客学院独家整理 , 盗版必究 会被 bebel  转化成如下:

|  |  |  |
| --- | --- | --- |
|  | JSX LO 复制代码 | |
| 1 | React.createElement( | |
| 2 | "div", | |
| 3 | null, | |
| 4 | |  | | --- | |  | | React.createElement("img", { |
| 5 | s rc: "avatar.png", | |
| 6 | className: "profile" | |
| 7 | }), | |
| 8 | React.createElement(Hello, null) | |
| 9 | ); | |

在转化过程中 , babel  在编译时会判断 JSX 中组件的首字母:

 当首字母为小写时 , 其被认定为原生 DOM  标签 , createElement  的第一个变量被编译为字 符串

 当首字母为大写时 , 其被认定为自定义组件 , createElement 的第一个变量被编译为对象 最终都会通过 RenderDOM.render(...)  方法进行挂载 , 如下:

|  |  |
| --- | --- |
| v JS | X LO 复制代码 |
| 1 ReactDOM.render(<App />, document.getElementById("root")); |  |



**19.2.** 过程

 react  中 , 节点大致可以分成四个类别 : 原生标签节点

在

文本节点 函数组件 类组件

如下所示 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX | | | | LO 复制代码 |
| 1 | class ClassComponent extends Component { | | | |  |
| 2 | |  | | --- | |  | | static defaultProps = { | | |  |
| 3 | color: "pink" | | | |  |
| 4 | }; | |  | |  |
| 5 | |  | | --- | |  | | render() { |  | |  |
| 6 | return ( | |  |  |  |
| 7 | <div className="border"> | |  |  |  |
| 8 | <h3>ClassComponent</h3> | | | |  |
| 9 | <p className={this.props.color}>{this.props.name}</p > | | | |  |
| 10 | </div> | | | |  |
| 11 | ); | | | |  |
| 12 | } | | | |  |
| 13 | } | | | |  |
| 14 |  | | | |  |
| 15 | function FunctionComponent(props) { | | | |  |
| 16 | return ( | | | |  |
| 17 | <div className="border"> | | | |  |
| 18 | FunctionComponent | | | |  |
| 19 | <p>{props.name}</p > | | | |  |
| 20 | </div> | | | |  |
| 21 | ); | | | |  |
| 22 | } | | | |  |
| 23 |  | | | |  |
| 24 | const jsx = ( | | | |  |
| 25 | <div className="border"> | | | |  |
| 26 | <p>xx</p > | | | |  |
| 27  28  29  30  31 | < a href=" ">xxx</ a>  <FunctionComponent name="函数组件 " />  <ClassComponent name="类组件 " color="red" /> </div>  ); | | | |  |

React.createElement 这种形式

这些类别最终都会被转化成



props

 chi

React.createElement 其被调用时会传入标签类型

, 标签属性

及若干子元素



type

 ,ldren 作用是生成  个虚拟

Dom 对象 , 如下所示:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Javascript | | | | | LO 复制代码 |
| 1 | function createElement(type, config, ...children) { | | | | |  |
| 2 | |  | | --- | |  | | | if (config) { | | |  |
| 3 | delete config. self; | | | | |  |
| 4 | delete config. source; | | | |  |  |
| 5 | } | | | |  |  |
| 6 | // ! 源码中做了详细处理 比如过滤掉key ref等 | | | | |  |
| 7 | |  | | --- | |  | | | const props = { |  |  |  |
| 8 | ...config, | | | | |  |
| 9 | children: children.map(child => | | | | |  |
| 10 | typeof child === "object" ? child : createTextNode(child) | | | | |  |
| 11 | ) | | | | |  |
| 12 | }; | | | | |  |
| 13 | |  | | --- | |  | | | return { | | |  |
| 14 | type, | | | | |  |
| 15 | props | | | | |  |
| 16 | }; | | | | |  |
| 17 | } | | | | |  |
| 18 | function createTextNode(text) { | | | | |  |
| 19 | |  | | --- | |  | | | return { | | |  |
| 20 | type: TEXT, | | | | |  |
| 21 | |  | | --- | |  | | | props: { | | |  |
| 22 | children: [], | | | | |  |
| 23 | nodeValue: text | | | | |  |
| 24  25  26  27  28  29 | |  | | --- | |  | | } };  }  export default {  createElement  }; | |  | |  |

createElement 会根据传入的节点信息进行一个判断:

。 如果是原生标签节点 , type 是字符串 , 如div 、 span  如果是文本节点 , type就没有 , 这里是 TEXT

 如果是函数组件 , type 是函数名  如果是类组件 , type 是类名

虚拟 DOM 会通过 ReactDOM.render 进行渲染成真实 DOM , 使用方法如下:

|  |  |  |
| --- | --- | --- |
|  | JS | X LO 复制代码 |
| 1 ReactDOM.render(element, container [, callback]) |  |  |

印客学院独家整理 , 盗版必究

当首次调用时 , 容器节点里的所有 DOM  元素都会被替换 , 后续的调用则会使用 React  的 diff  算法进行高效的更新

如果提供了可选的回调函数 ,callback 该回调将在组件被渲染或更新之后被执行

render  大致实现方法如下:  



印客学院独家整理 , 盗版必究

v Javascript B 复制代码



1  function render(vnode, container) {

2 console.log("vnode", vnode); // 虚拟DOM对象 3 // vnode \_> node

4 const node = createNode(vnode, container);

5 container.appendChild(node); 6 }

7

8 // 创建真实DOM节点

9  function createNode(vnode, parentNode) {

10 let node = null;

11 const {type, props} = vnode; 12  if (type === TEXT) {

13 node = document.createTextNode(""); 14  } else if (typeof type === "string") {

15 node = document.createElement(type); 16  } else if (typeof type === "function") {

17 node = type.isReactComponent

18 ? updateClassComponent(vnode, parentNode)

19 : updateFunctionComponent(vnode, parentNode);

20  } else {

21 node = document.createDocumentFragment(); 22 }

23 reconcileChildren(props.children, node);

24 updateNode(node, props);

25 return node; 26 }

27

28 // 遍历下子vnode, 然后把子vnode->真实DOM节点 , 再插入父node中 29  function reconcileChildren(children, node) {

30  for (let i = 0; i < children.length; i++) {

31 let child = children [i];

32  if (Array.isArray(child)) {

33  for (let j = 0; j < child.length; j++) {

34 render(child [j], node);

35 }

36  } else {

37 render(child, node);

38 }

39 }

40 }

41  function updateNode(node, nextVal) {

42 Object.keys(nextVal)

43 .filter(k => k !== "children")

44  .forEach(k => {

45  if (k.slice(0, 2) === "on") {

印客学院独家整理 , 盗版必究

46

let eventName = k.slice(2).toLocaleLowerCase();

node.addEventListener(eventName, nextVal [k]); } else {

node [k] = nextVal [k];

47

48

49

50

} });

51

52

}

53

54

// 返回真实dom节点 // 执行函数

function updateFunctionComponent(vnode, parentNode) {

const {type, props} = vnode; let vvnode = type(props);

const node = createNode(vvnode, parentNode); return node;

55

56

57

58

59

60

61

}

// 返回真实dom节点

// 先实例化 , 再执行render函数

function updateClassComponent(vnode, parentNode) { const {type, props} = vnode;

62

63

64

65

66

67

let cmp = new type(props); const vvnode = cmp.render();

const node = createNode(vvnode, parentNode); return node;

68

69

70

71

}

export default {

render

72

73

74

};





**19.3.** 总结



react

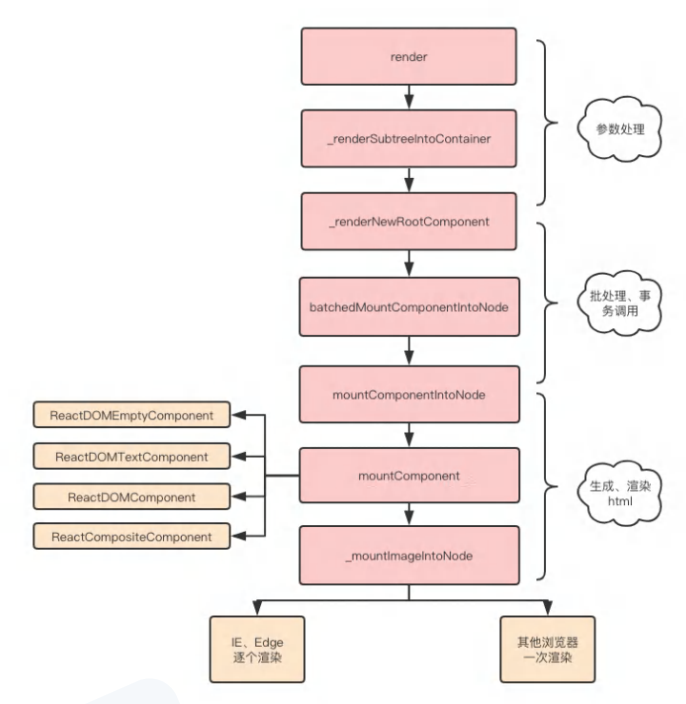
Dom 整体流程如下图所示 :

Dom 转化成真实

在

源码中 , 虚拟

印客学院独家整理 , 盗版必究



其渲染流程如下所示:

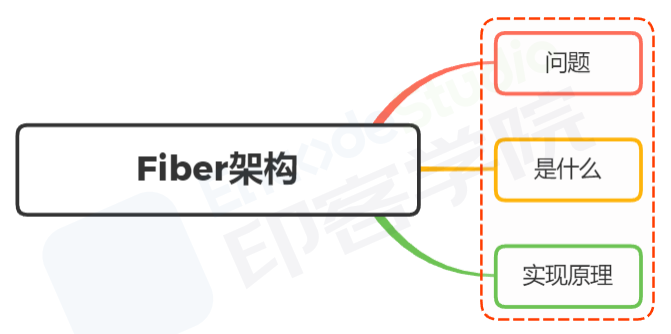
。 使用React. createElement或JS 编写React组件 实际上所有的 JS 代码最后都会转换成 React. createElement(.) , Babel帮助我们完成了这个转换的过程。

o createElement函数对key和ref等特殊的props进行处理 , 并获取defaultprops对默认props进行赋 值 , 并且对传入的孩子节点进行处理 , 最终构造成一个虚拟DOM对象

 ReactDOM. render将生成好的虚拟DOM渲染到指定容器上 , 其中采用了批处理 、事务等机制并且对 特定浏览器进行了性能优化 , 最终转换为真实DOM

印客学院独家整理 , 盗版必究

**20.** 说说对Fiber架构的理解? 解决了什么问题?



**20.1.** 问题

JavaScript  引擎和页面渲染引擎两个线程是互斥的 , 当其中 -个线程执行时 , 另 -个线程只能挂起 等待

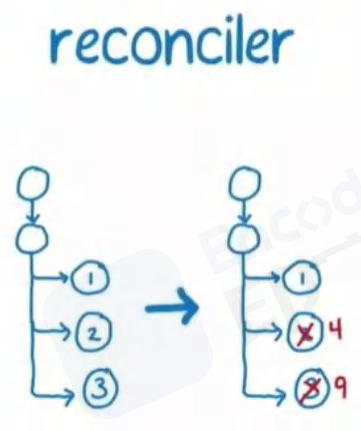
如果 JavaScript  线程长时间地占用了主线程 , 那么渲染层面的更新就不得不长时间地等待 , 界面长 时间不更新 , 会导致页面响应度变差 , 用户可能会感觉到卡顿

而这也正是 React 15  的 Stack Reconcile r  所面临的问题 , 当 React  在渲染组件时 , 从开

始到渲染完成整个过程是一气呵成的 , 无法中断

如果组件较大 , 那么 js  线程会-直执行 , 然后等到整棵 VDOM  树计算完成后 , 才会交给渲染的线程 这就会导致一些用户交互 、动画等任务无法立即得到处理 , 导致卡顿的情况

印客学院独家整理 , 盗版必究



**20.2.** 是什么

React Fiber 是 Facebook 花费两年余时间对 React 做出的 一个重大改变与优化 , 是对 React 核心算法 的 一次重新实现 。 从Facebook在 React conf 2017 会议上确认 , React Fiber 在React 16 版本发布

在 react  中 , 主要做了以下的操作 :

 为每个增加了优先级 , 优先级高的任务可以中断低优先级的任务 。然后再重新 , 注意是重新执行优 先级低的任务

。 增加了异步任务 , 调用requestldlecallback api , 浏览器空闲的时候执行

o dom diff树变成了链表 , 一个dom对应两个fiber ( -个链表) , 对应两个队列 , 这都是为找到被中 断的任务 , 重新执行

从架构角度来看 , Fiber  是对 React  核心算法 (即调和过程) 的重写

从编码角度来看 , Fiber  是 React  内部所定义的 一种数据结构 , 它是 Fiber  树结构的节点单 位 , 也就是 React 16  新架构下的虚拟 DOM 

-个 fiber  就是一个 JavaScript  对象 , 包含了元素的信息 、该元素的更新操作队列 、类型 , 其数 据结构如下 :

印客学院独家整理 , 盗版必究

v Javascript B 复制代码



1 type Fiber = {

2 // 用于标记fiber的WorkTag类型 , 主要表示当前fiber代表的组件类型如FunctionCompone

nt ClassComponent等

3 tag: WorkTag,

4 // ReactElement里面的key

5 key: null | string,

6 // ReactElement.type, 调用 `createElement`的第  个参数

7 elementType: any,

8 // The resolved function/class/ associated with this fiber.

9 // 表示当前代表的节点类型

10 type: any,

11 // 表示当前FiberNode对应的element组件实例

12 stateNode: any,

13

14 // 指向他在Fiber节点树中的`parent`, 用来在处理完这个节点之后向上返回

15 return: Fiber | null, 16 // 指向自己的第  个子节点

17 child: Fiber | null,

18 // 指向自己的兄弟结构 , 兄弟节点的 return指向同 -个父节点

19 sibling: Fiber | null,

20 index: number, 21

22 ref: null | (((handle: mixed) => void) & { \_stringRef: ?string }) | RefO bject,

23

24 // 当前处理过程中的组件props对象

25 pendingProps: any,

26 // 上-次渲染完成之后的props

27 memoizedProps: any,

28

29 // 该Fiber对应的组件产生的Update会存放在这个队列里面

30 updateQueue: UpdateQueue<any> | null,

31

32 // 上-次渲染的时候的state

33 memoizedState: any,

34

35 // -个列表 , 存放这个Fiber依赖的context

36 firstContextDependency: ContextDependency<mixed> | null, 37

38 mode: TypeOfMode, 39

40 // Effect

41 // 用来记录Side Effect

42 effectTag: SideEffectTag,

43

印客学院独家整理 , 盗版必究

// 单链表用来快速查找下-个side effect

nextEffect: Fiber | null,

// 子树中第  个side effect

firstEffect: Fiber | null,

// 子树中最后-个side effect

lastEffect: Fiber | null,

// 代表任务在未来的哪个时间点应该被完成 , 之后版本改名为 lanes

expirationTime: ExpirationTime,

// 快速确定子树中是否有不在等待的变化

childExpirationTime: ExpirationTime,

// fiber的版本池  即记录fiber更新过程 , 便于恢复

alternate: Fiber | null, }



44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

**20.3.** 如何解决

Fiber  把渲染更新过程拆分成多个子任务 , 每次只做  小部分 , 做完看是否还有剩余时间 , 如果有继 续下一个任务; 如果没有 , 挂起当前任务 , 将时间控制权交给主线程 , 等主线程不忙的时候在继续执行

即可以中断与恢复 , 恢复后也可以复用之前的中间状态 , 并给不同的任务赋予不同的优先级 , 其中每个

任务更新单元为 React Element  对应的 Fiber  节点 实现的上述方式的是 requestIdleCallback  方法

 window.requestIdleCallback()  方法将在浏览器的空闲时段内调用的函数排队 。这使开发者能 够在主事件循环上执行后台和低优先级工作 , 而不会影响延迟关键事件 , 如动画和输入响应

首先 React 中任务切割为多个步骤 , 分批完成 。在完成一部分任务之后 , 将控制权交回给浏览器 , 让浏 览器有时间再进行页面的渲染 。等浏览器忙完之后有剩余时间 , 再继续之前 React 未完成的任务 , 是一 种合作式调度。



该实现过程是基于 Fiber  节点实现 , 作为静态的数据结构来说 , 每个 Fiber  节点对应-个  Reac

 ,telement 保存了该组件的类型 (函数组件/类组件/原生组件等等) 、对应的 DOM 节点等信息。

作为动态的工作单元来说 , 每个 Fiber  节点保存了本次更新中该组件改变的状态 、要执行的工作。 每个 Fiber 节点有个对应的 , 多Reactelement个 Fiber  节点根据如下三个属性构建一颗树 :

印客学院独家整理 , 盗版必究

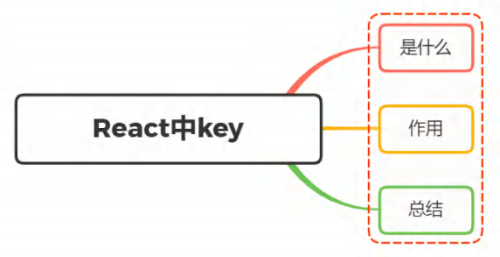
|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Javascript | LO 复制代码 |
| 1 | // 指向父级Fiber节点 |  |  |
| 2 | this.return = null |  |  |
| 3 | // 指向子Fiber节点 |  |  |
| 4 | this.child = null |  |  |
| 5 | // 指向右边第  个兄弟Fiber节点 |  |  |
| 6 | this.sibling = null |  |  |

通过这些属性就能找到下 一个执行目标





**21.** React中的key有什么作用?



**21.1.** 是什么

首先 , 先给出 react  组件中进行列表渲染的-个示例 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | | | JS | X LO 复制代码 |
| 1 | const data = [ | | |  |  |
| 2 | { id: 0, name: 'abc' }, | | |  |  |
| 3 | { id: 1, name: 'def' }, | | |  |  |
| 4 | { id: 2, name: 'ghi' }, | | |  |  |
| 5 | { id: 3, name: 'jkl' } | |  |  |  |
| 6 | ]; | | |  |  |
| 7 |  | | |  |  |
| 8 | const ListItem = (props) => { | | |  |  |
| 9 | return <li>{props.name}</li>; | | |  |  |
| 10 | }; |  | |  |  |
| 11 |  | | |  |  |
| 12 | const List = () => { | | |  |  |
| 13 | return ( | | |  |  |
| 14 | <ul> | | |  |  |
| 15 | {data.map((item) => ( | | |  |  |
| 16 | <ListItem name={item.name}></ListItem> | | |  |  |
| 17 | ))} | | |  |  |
| 18 | </ul> | | |  |  |
| 19 | ); | | |  |  |
| 20 | }; | | |  |  |

然后在输出就可以看到 react  所提示的警告信息:

根据意思就可以得到渲染列表的每  个子元素都应该需要  个唯 的 key  值  id key 

|  |  |  |
| --- | --- | --- |
|  | LaTe | X COL 复制代码 |
| 1 Each child in a list should have a unique "key" prop. | |  |

|  |
| --- |
|  |
| 1  const List = () => {  2 return (  3 <ul>  4 {data.map((item) => (  5 <ListItem name={item.name} key={item.id}></ListItem>  6 ))} 7 </ul> |
| 8 ); 9 }; |

印客学院独家整理 , 盗版必究

**21.2.** 作用

跟 一Vue样 , React  也存在 Diff  算法 , 而元素 key  属性的作用是用于判断元素是新创建的还

是被移动的元素 , 从而减少不必要的元素渲染

因此 key  的值需要为每  个元素赋予  个确定的标识   

如果列表数据渲染中 · 在数据后面插入  条数据  key  作用并不大 如下 : 

|  |  |  |
| --- | --- | --- |
|  |  | JSX LO 复制代码 |
| 1  this.state = {  2 numbers: [111,222,333] 3 }  4  5  insertMovie() {  6 const newMovies = [...this.state.numbers, 444]; 7- this.setState({  8 movies: newMovies  9 }) 10 } | | |
| 11  12 <ul> 13 {  14 this.state.movies.map((item, index) => {  15 return <li>{item}</li> 16 })  17 } 18 </ul> | | |

前面的元素在 diff  算法中 · 前面的元素由于是完全相同的 并不会产生删除创建操作 · 在最后  个比



   key 

较的时候 · 则需要插入到新的 DOM  树中   



下面再来看看在前面插入数据时 , 使用 key  与不使用 key  的区别 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | Javascript | LO 复制代码 |
| 1 | insertMovie() { | |  |  |
| 2 | const newMovies = [000 ,...this.state.numbers]; | |  |  |
| 3 | |  | | --- | |  | | this.setState({ |  |  |
| 4 | movies: newMovies | |  |  |
| 5 | }) | |  |  |
| 6 | } | |  |  |

印客学院独家整理 , 盗版必究

当拥有 key  的时候 , react  根据 key  属性匹配原有树上的子元素以及最新树上的子元素 , 像上述情 况只需要将000元素插入到最前面位置

当没有 key  的时候 , 所有的 li  标签都需要进行修改

同样 · 并不是拥有 key  值代表性能越高 · 如果说只是文本内容改变了 , 不写 key  反而性能和效率更高 主要是因为不写 key  是将所有的文本内容替换  下 · 节点不会发生变化 

而写 key  则涉及到了节点的增和删  发现旧 key  不存在了 , 则 将其删除  新 key  在之前没有 , 则插 入 , 这就增加性能的开销

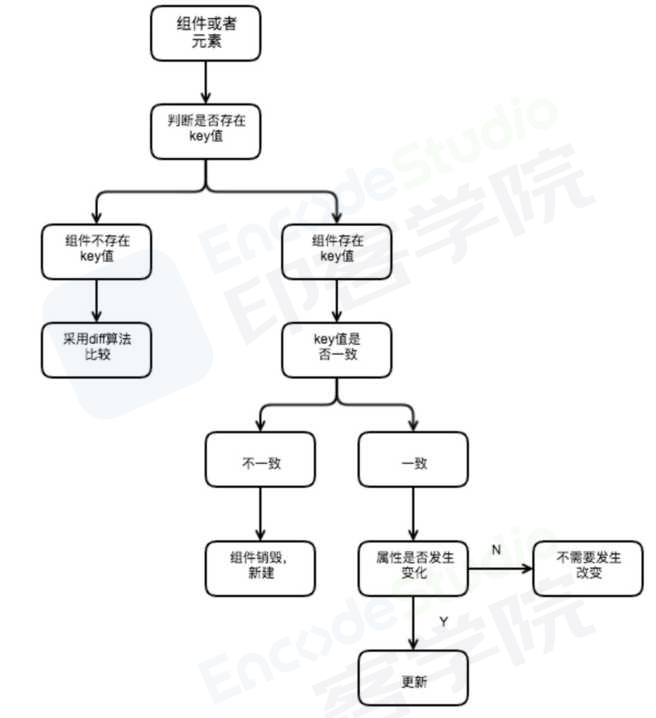
**21.3.** 总结

良好使用 key  属性是性能优化的非常关键的  步 , 注意事项为 :  key 应该是唯一 的

 key不要使用随机值 (随机数在下一次 render 时 , 会重新生成一个数字) • 使用 index 作为 key值 , 对性能没有优化

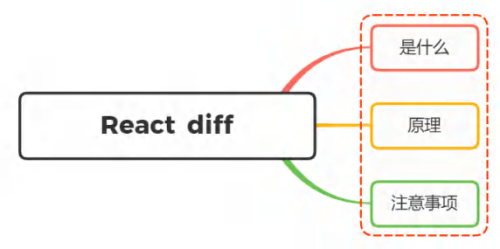
react  判断 key  的流程具体如下图 :

印客学院独家整理 , 盗版必究



**22.** 说说React diff的原理是什么?

印客学院独家整理 , 盗版必究



**22.1.** 是什么

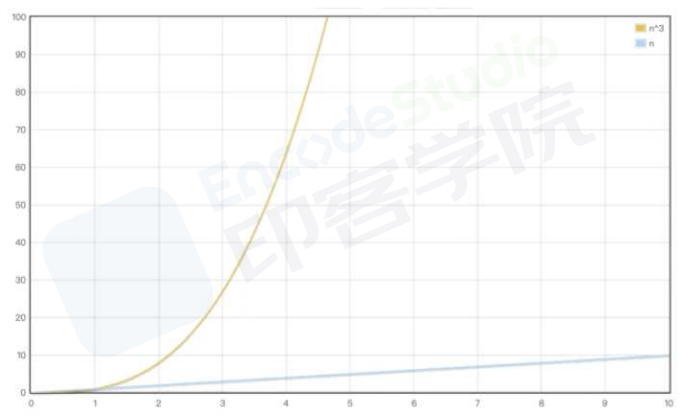
跟 Vue   致 , React  通过引入 Virtual DOM  的概念 , 极大地避免无效的 Dom  操作 ,

使我们的页

面的构建效率提到了极大的提升

而 diff  算法就是更高效地通过对比新旧 Virtual DOM  来找出真正的 Dom  变化之处

传统diff算法通过循环递归对节点进行依次对比 , 效率低下 , 算法复杂度达到 o(n  3) , react  将算法 进行-个优化 , 复杂度 ,O(n) 两者效率差距如下图 :



印客学院独家整理 , 盗版必究

**22.2.**

原理



react 中

diff 算法主要遵循三个层级的策略 :

tree层级

conponent 层级

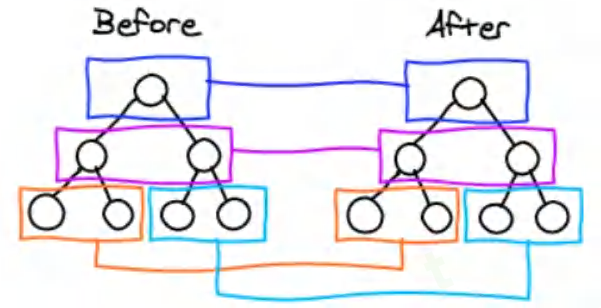
elemEnt 层级



**22.2.1.** tree层级

DOM 节点跨层级的操作不做优化 ,

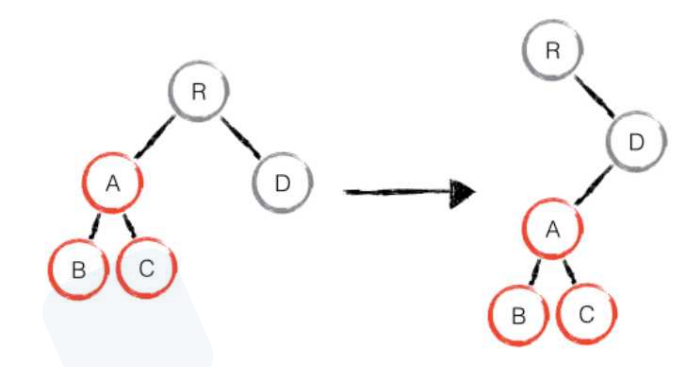
只会对相同层级的节点进行比较



没有移动操作 , 如下图 :

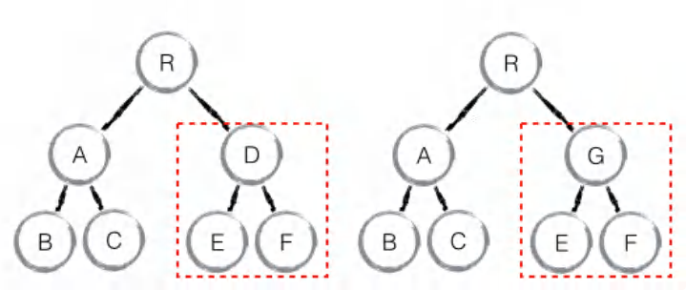
只有删除 、创建操作 ,

印客学院独家整理 , 盗版必究



react  发现新树中 , R节点下没有了A, 那么直接删除A , 在D节点下创建A以及下属节点 上述操作中 , 只有删除和创建操作

**22.2.2.** conponent层级



diff  运算 , 如果不是  个类的组件 , 那么直接删除这个组件下

的所有子节点 , 创建新的

如果是同 一个类的组件 , 则会继续往下

当 component D  换成了 component G  后 , 即使两者的结构非常类似 , 也会将 D  删除再重新创 建 G 

印客学院独家整理 , 盗版必究

**22.2.3.** element层级

对于比较同 -层级的节点们 , 每个节点在对应的层级用唯一的 key  作为标识

 REMOVE\_NO

MOVE\_EXISTING 

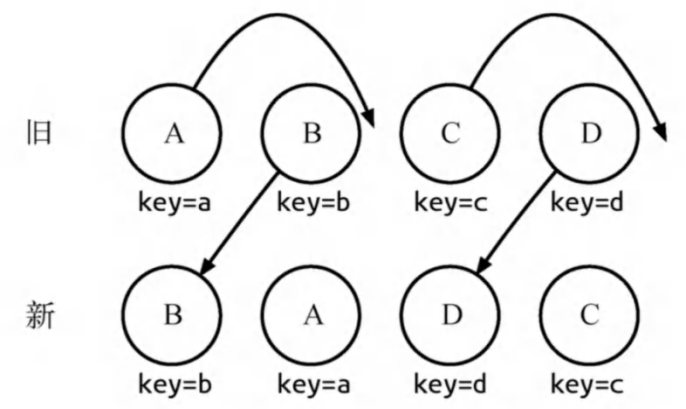
提供了 3 种节点操作 , 分别为 (INSERT\_MARKUP插入)

(移动)和

DE 

(删除)

如下场景 :



通过 key  可以准确地发现新旧集合中的节点都是相同的节点 , 因此无需进行节点删除和创建 , 只需要 将旧集合中节点的位置进行移动 , 更新为新集合中节点的位置

流程如下表:



印客学院独家整理 , 盗版必究

o index: 新集合的遍历下标。

o oldlndex: 当前节点在老集合中的下标

o maxIndex: 在新集合访问过的节点中 , 其在老集合的最大下标

如果当前节点在新集合中的位置比老集合中的位置靠前的话 , 是不会影响后续节点操作的 , 这里这时候 被动字节不用动

操作过程中只比较oldlndex和maxIndex , 规则如下:

 当oldlndex>maxIndex时 , 将oldlndex的值赋值给maxIndex  当oldlndex=maxIndex时 , 不操作

。 当oldndex<maxIndex时 , 将当前节点移动到index的位置 diff  过程如下 :

 节点B: 此时 maxIndex=0 , oldlndex=1 ; 满足 maxIndex< oldlndex , 因此B节点不动 , 此时 maxIndex= Math. max(oldlndex, maxIndex) , 就是1

 节点A: 此时maxIndEX=1 , oldlndex=0; 不满足maxIndex< oldlndex , 因此A节点进行移动操作 , 此时maxIndex= Math. max(oldlndex, maxIndex) , 还是1

 节点D: 此时maxIndex=1 , oldlndex=3; 满足maxIndex< oldlndex , 因此D节点不动 , 此时 maxIndex= Math. max(oldlndex, maxIndex) , 就是3

 节点c: 此时maxIndex=3 , oldlndex=2; 不满足maxIndex< oldlndex , 因此c节点进行移动操作 ,

当前已经比较完了

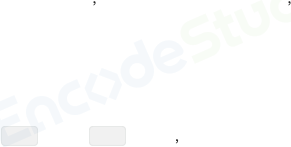


当ABCD节点比较完成后 , 过程还没完 还会整体遍历老集合中节点 看有没有没用到的节点 ,

有的话, 就删除

diff

**22.3.** 注意事项

对于简单列表渲染而言 , 不使用 key 比使用 key 的性能 例如 : 将一个[1 , 2,3 ,4,5] , 渲染成如下的样子 :

|  |  |  |
| --- | --- | --- |
|  | HTM | L LO 复制代码 |
| 1 <div>1</div> |  |  |
| 2 <div>2</div> |  |  |
| 3 <div>3</div> |  |  |
| 4 <div>4</div> |  |  |
| 5 <div>5</div> |  |  |

后续更改成1 , 3 , 2 , 5 ,4) , 使用 key  与不使用 key  作用如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | HTML | LO 复制代码 |
| 1 | 1.加key |  |  |
| 2 | <div key= '1'>1</div> <div key= '1'>1</div> |  |  |
| 3 | <div key= '2'>2</div> <div key= '3'>3</div> |  |  |
| 4 | <div key= '3'>3</div> ========> <div key= '2'>2</div> |  |  |
| 5 | <div key= '4'>4</div> <div key= '5'>5</div> |  |  |
| 6 | <div key= '5'>5</div> <div key= '4'>4</div> |  |  |
| 7 |  |  |  |
| 8 |  |  |  |
| 9 | 2.不加key |  |  |
| 10 | <div>1</div> <div>1</div> |  |  |
| 11 | <div>2</div> <div>3</div> |  |  |
| 12 | <div>3</div> ========> <div>2</div> |  |  |
| 13 | <div>4</div> <div>5</div> |  |  |
| 14 | <div>5</div> <div>4</div> |  |  |
| 15 | 操作: 修改第1个到第5个节点的innerText |  |  |

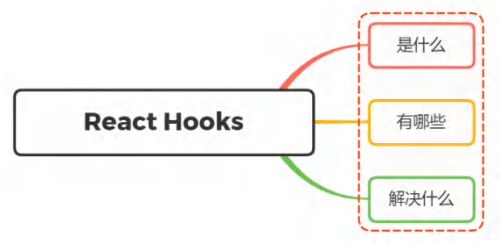
如果我们对这个集合进行增删的操作改成[1 ,3,2,5 ,6]

|  |  |  |
| --- | --- | --- |
|  | HTML | LO 复制代码 |
| 1 | 1.加key |  |
| 2 | <div key= '1'>1</div> <div key= '1'>1</div> |  |
| 3 | <div key= '2'>2</div> <div key= '3'>3</div> |  |
| 4 | <div key= '3'>3</div> ========> <div key= '2'>2</div> |  |
| 5 | <div key= '4'>4</div> <div key= '5'>5</div> |  |
| 6 | <div key= '5'>5</div> <div key= '6'>6</div> |  |
| 7 | 操作: 节点2移动至下标为2的位置 新增节点6至下标为4的位置 删除节点4 |  |
| 8 |  |  |
| 9 | 2.不加key |  |
| 10 | <div>1</div> <div>1</div> |  |
| 11 | <div>2</div> <div>3</div> |  |
| 12 | <div>3</div> ========> <div>2</div> |  |
| 13 | <div>4</div> <div>5</div> |  |
| 14 | <div>5</div> <div>6</div> |  |
| 15 | 操作: 修改第1个到第5个节点的innerText |  |

由于 dom  节点的移动操作开销是比较昂贵的 , 没有 key  的情况下要比有 key  的性能更好

**23.** 说说对React HooKs的理解? 解决了什么问题?

印客学院独家整理 , 盗版必究



**23.1.** 是什么

state

 Hook  是 React 16 .8 的新增特性 。它可以让你在不编写 class  的情况下使用 以及其他

的 React  特性

至于为什么引入 ,hook 官方给出的动机是解决长时间使用和维护 react  过程中常遇到的问题 , 例

如 :

• 难以重用和共享组件中的与状态相关的逻辑

 逻辑复杂的组件难以开发与维护 , 当我们的组件需要处理多个互不相关的 local state 时 , 每个生命

周期函数中可能会包含着各种互不相关的逻辑在里面

 类组件中的this增加学习成本 , 类组件在基于现有工具的优化上存在些许问题

 由于业务变动 , 函数组件不得不改为类组件等等 

在以前 , 函数组件也被称为无状态的组件 , 只负责渲染的

些工作

因此 , 现在的函数组件也可以是有状态的组件 , 内部也可以维护自身的状态以及做一些逻辑方面的处理

**23.2.** 有哪些

Hooks  让我们的函数组件拥有了类组件的特性 , 例如组件内的状态 、生命周期

上面讲到 ,

hooks 有如下 :

最常见的

usestate

o useEffect 。 其他

印客学院独家整理 , 盗版必究

**23.2.1.** usestate

首先给出 一个例子 , 如下 :

|  |  |
| --- | --- |
| Javascrpt ca 复制代码 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import React, { useState } from 'react';  function Example() {  // 声明 -个叫 "count" 的 state 变量  const [count, setCount] = useState(0);  return ( <div>  <p>You clicked {count} times</p >  <button onClick={() => setCount(count + 1)}> Click me  </button> </div>  ); |
| 15 } | |

在函数组件中通过 useState  实现函数内部维护 ,state 参数为 state  默认的值 , 返回值是  个 数组 , 第一个值为当前的 ,state 第二个值为更新 state  的函数

该函数组件等价于的类组件如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Javascript B 复制代码 | | | | |
| 1 | class Example extends React.Component { | | | | |
| 2 | |  | | --- | |  | | constructor(props) { | | | |
| 3 | super(props); | | | | |
| 4 | |  | | --- | |  | | this.state = { | |  | |
| 5 | count: 0 | | | | |
| 6 | }; | | |  |  |
| 7 | } | | |  |  |
| 8 |  | | | | |
| 9 | |  | | --- | |  | | render() { |  | | |
| 10 | return ( | |  | | |
| 11 | <div> | | | | |
| 12 | <p>You clicked {this.state.count} times</p > | | | | |
| 13 | <button onClick={() => this.setState({ count: this.state.count + 1  })}> | | | | |
| 14 | Click me | | | | |
| 15 | </button> | | | | |
| 16 | </div> | | | | |
| 17 | ); | | | | |
| 18 | } | | | | |
| 19 | } | | | | |

从上述两种代码分析 , 可以看出两者区别 :



this.state.count 的方式获取

state声明方式 : 在函数组件中通过 usestate 直接获取 , 类组件通过constructor 构造函数中设置 state读取方式 : 在函数组件中直接使用变量 , 类组件通过



this 指向不明确的情况

state更新方式 : 在函数组件中通过 setcount 更新 , 类组件通过this.setstate() 总的来讲 , usestate 使用起来更为简洁 , 减少了

**23.2.2.** useEffect



可以让我们在函数组件中进行一些带有副作用的操作

useEffect

同样给出 一个计时器示例 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Javascript B 复制代码 | | | | |
| 1 | class Example extends React.Component { | | | | |
| 2 | |  | | --- | |  | | constructor(props) { | | | |
| 3 | super(props); | | | | |
| 4 | |  | | --- | |  | | this.state = { | |  | |
| 5 | count: 0 | | | | |
| 6 | }; | | |  |  |
| 7 | } | | |  |  |
| 8 |  | | | | |
| 9 | |  | | --- | |  | | componentDidMount() { |  | | |
| 10 | document.title = `You clicked ${this.state.count} times`; | | | | |
| 11 | } | | | | |
| 12 | |  | | --- | |  | | componentDidUpdate() { | | | |
| 13 | document.title = `You clicked ${this.state.count} times`; | | | | |
| 14 | } | | | | |
| 15 |  | | | | |
| 16 | |  | | --- | |  | | render() { | | | |
| 17 | return ( | | | | |
| 18 | <div> | | | | |
| 19 | <p>You clicked {this.state.count} times</p > | | | | |
| 20 | <button onClick={() => this.setState({ count: this.state.count + 1 | | | | |
|  | })}> | | | | |
| 21 | Click me | | | | |
| 22 | </button> | | | | |
| 23  24  25  26 | </div> );  } } | |  | | |



useEffect 后 , 则能够将相同的逻辑抽离出来 , 这是类组件不具备的方法

从上面可以看见 , 组件在加载和更新阶段都执行同样操作 而如果使用

useEffect 示例如下 :

对应的

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | JSX | LO 复制代码 |
| 1 | import React, { useState, useEffect } from 'react'; |  |
| 2 | function Example() { |  |
| 3 | const [count, setCount] = useState(0); |  |
| 4 |  |  |
| 5 | useEffect(() => { document.title = `You clicked ${count} | times`; }); |
| 6 | return ( |  |
| 7 | <div> |  |
| 8 | <p>You clicked {count} times</p > |  |
| 9 | <button onClick={() => setCount(count + 1)}> |  |
| 10 | Click me |  |
| 11 | </button> |  |
| 12 | </div> |  |
| 13 | ); |  |
| 14 | } |  |

useEffect  第一个参数接受-个回调函数 , 默认情况下 , useEffect  会在第一次渲染和更新之后 都会执行 , 相当于在 componentDidMount  和 componentDidUpdate  两个生命周期函数中执行回

调

如果某些特定值在两次重渲染之间没有发生变化 , 你可以跳过对 effect 的调用 , 这时候只需要传入第二 个参数 , 如下:

|  |  |  |
| --- | --- | --- |
| v Javascript | | LO 复制代码 |
| 1  useEffect(() => { | |  |
| 2 document.title = `You clicked ${count} times`; | |  |
| 3 }, [count]); // 仅在 count 更改时更新 |  |  |

上述传入第二个参数后 , 如果 count  的值是 5  而且我们的组件重渲染的时候 count  还是等

于 5  · React 将对前  次渲染的 [5] 和后  次渲染的 [5]  进行比较 · 如果是相等则跳过

ts  执行

回调函数中可以返回 -个清除函数 , 这是 effect  可选的清除机制 , 相当于类组件中 

lUnmount  生命周期函数 , 可做  些清除副作用的操作 , 如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX LO 复制代码 | | |
| 1 | useEffect(() => { | | |
| 2 | |  | | --- | |  | | | function handleStatusChange(status) { |
| 3  4  5  6  7  8  9  10 | |  | | --- | |  | | setIsOnline(status.isOnline); }  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange); return () => {  ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusC hange);  }; }); | |



useEffect

 componentWi

相当于

所以 ,

和



componentDidMount 

, componentDidUpdate

llUnmount  这三个生命周期函数的组合

**23.2.3.** 其它 hooks

在组件通信过程中可以使用 useContext  , refs  学习中我们也用到了 useRef  获取 DOM  结构 … 还有很多额外的 ,hooks 如 :

o useReducer

o usecallback o useMemo

o USEREf





**23.3.** 解决什么

通过对上面的初步认识 , 可以看到 hooks  能够更容易解决状态相关的重用的问题: 。 每调用useHooK— 次都会生成一份独立的状态

 通过自定义hook能够更好的封装我们的功能

编写 hooks  为函数式编程 , 每个功能都包裹在函数中 , 整体风格更清爽 , 更优雅

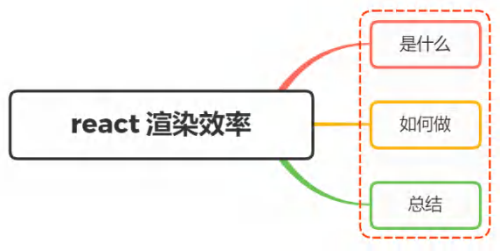
 hooks  的出现 , 使函数组件的功能得到了扩充 , 拥有了类组件相似的功能  在我们  常使用中  使用 

hooks  能够解决大多数问题 , 并且还拥有代码复用机制 , 因此优先考虑 hooks 

**24.** 说说你是如何提高组件的渲染效率的? 在React中如

印客学院独家整理 , 盗版必究

何避免不必要的render?



**24.1.** 是什么

react  基于虚拟 DOM  和高效 Diff  算法的完美配合 , 实现了对 DOM  最小粒度的更新 , 大多数 情况下 , React  对 DOM  的渲染效率足以我们的业务日常

复杂业务场景下 , 性能问题依然会困扰我们 。此时需要采取一些措施来提升运行性能 , 避免不必要的渲 染则是业务中常见的优化手段之

**24.2.** 如何做

在之前文章中 , 我们了解到 render  的触发时机 , 简单来讲就是类组件通过调用 setState  方法 , 就会导致 ,render 父组件-旦发生 render  渲染 , 子组件一定也会执行 render  渲染

从上面可以看到 , 父组件渲染导致子组件渲染 , 子组件并没有发生任何改变 , 这时候就可以从避免无谓

的渲染 , 具体实现的方式有如下 : o shouldcomponentupdate  purecomponent

o React. memo

**24.2.1.** shouldcomponentupdate

通过 shouldComponentUpdate  生命周期函数来比对 state  和 ,props 确定是否要重新渲染 默认情况下返回 true  表示重新渲染 , 如果不希望组件重新渲染 , 返回 false  即可

印客学院独家整理 , 盗版必究

**24.2.2.** purecomponent

跟 shouldComponentUpdate  原理基本  致 , 通过对

props

和 state  的浅比较结果来实现 

houldComponentUpdate 

, 源码大致如下 :

|  |  |
| --- | --- |
| v Javascript B 复制代码 | |
| 1 | if (this.\_compositeType === CompositeTypes.PureClass) { |
| 2 | shouldUpdate = !shallowEqual(prevProps, nextProps) || ! shallowEqual(in st.state, nextState); |
| 3 | } |

 shallowEqual 

对应方法大致如下:

印客学院独家整理 , 盗版必究

v Javascript B 复制代码



1 const hasOwnProperty = Object.prototype.hasOwnProperty; 2

3 /\*\*

4 \* is 方法来判断两个值是否是相等的值 , 为何这么写可以移步 MDN 的文档

5 \* https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Globa l\_Objects/Object/is

6 \*/

|  |
| --- |
|  |
|  |

7 function is(x: mixed, y: mixed): boolean {

8 if (x === y) {

9 return x !== 0 || y !== 0 || 1 / x === 1 / y; 10 } else {

11 return x !== x && y !== y; 12 }

13 } 14

15 function shallowEqual(objA: mixed, objB: mixed): boolean { 16 // 首先对基本类型进行比较

17 if (is(objA, objB)) {

18 return true; 19 }

20

21 if (typeof objA !== 'object' || objA === null ||

22 typeof objB !== 'object' || objB === null) {

23 return false; 24 }

25

26 const keysA = Object.keys(objA);

27 const keysB = Object.keys(objB); 28

29 // 长度不相等直接返回false

30 if (keysA.length !== keysB.length) {

31 return false; 32 }

33

34 // key相等的情况下 , 再去循环比较

35 for (let i = 0; i < keysA.length; i++) {

36 if (

37 !hasOwnProperty.call(objB, keysA[i]) ||

38 !is(objA[keysA[i]], objB[keysA[i]])

39 ) {

40 return false;

41 } 42 } 43

44 return true;

印客学院独家整理 , 盗版必究

45 }

当对象包含复杂的数据结构时 , 对象深层的数据已改变却没有触发 render 

注意: 在 react  中 , 是不建议使用深层次结构的数据

**24.2.3.** React. memo

React.memo  用来缓存组件的渲染 , 避免不必要的更新 , 其实也是  个高阶组件  与  PureCompone

nt +分类似 · 但不同的是 · React.memo 只能用于函数组件

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | JS | X LO 复制代码 |
| 1 | import { memo } from 'react'; |  |  |
| 2 |  |  |  |
| 3 | function Button(props) { |  |  |
| 4 | // Component code |  |  |
| 5 | } |  |  |
| 6 |  |  |  |
| 7 | export default memo(Button); |  |  |

如果需要深层次比较 , 这时候可以给 memo  第二个参数传递比较函数

|  |  |
| --- | --- |
| JSX | LO 复制代码 |
| function arePropsEqual(prevProps, nextProps) { // your code  1  2  3 4 5 6  return prevProps === nextProps; }  export default memo(Button, arePropsEqual); |  |



**24.3.** 总结

在实际开发过程中 , 前端性能问题是一个必须考虑的问题 随着业务的复杂 遇到性能问题的概率也在

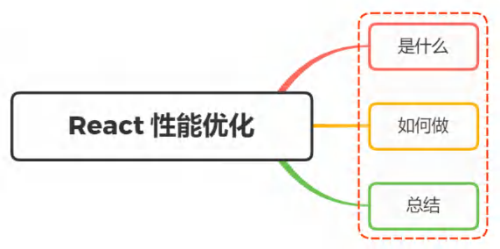
增高

除此之外 , 建议将页面进行更小的颗粒化 , 如果 个过大 当状态发生修改的时候 , 就会导致整个大组

件的渲染 , 而对组件进行拆分后 , 粒度变小了 , 也能够减少子组件不必要的渲染

**25.** 说说 React 性能优化的手段有哪些?

印客学院独家整理 , 盗版必究

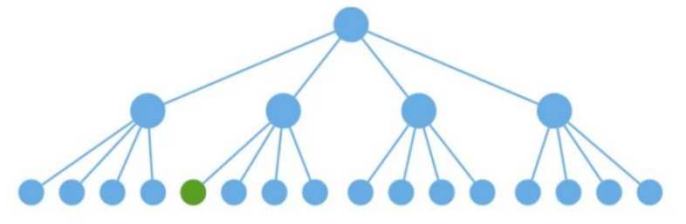


**25.1.** 是什么

React  凭借 virtual DOM  和 diff  算法拥有高效的性能 , 但是某些情况下 , 性能明显可以进一步 提高

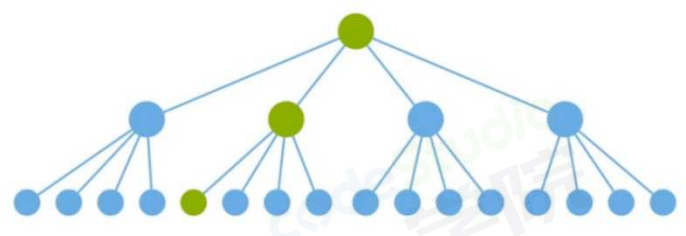
在前面文章中 , 我们了解到类组件通过调用 setState  方法 , 就会导致 ,render 父组件-旦发生

render  渲染 , 子组件一定也会执行 render  渲染 当我们想要更新一个子组件的时候 , 如下图绿色部分:



理想状态只调用该路径下的组件 render  :

印客学院独家整理 , 盗版必究



react 的默认做法是调用所有组件的

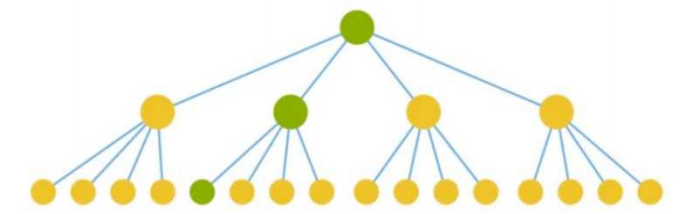
但是

(黄色部分) ,

,render 再对生成的虚拟

DOM 进行对比

如不变则不进行更新



从上图可见 , 黄色部分 diff  算法对比是明显的性能浪费的情况



**25.2.** 如何做

在React中如何避免不必要的render中 , 我们了解到如何避免不必要的 render  来应付上面的问题 , 主



PureComponent

这三种形式这里

要手段是通过

、 React.memo ,

shouldComponentUpdate

就不再复述

除此之外 , 常见性能优化常见的手段有如下:  避免使用内联函数

 使用 React Fragments 避免额外标记 。 使用 Immutable

。 懒加载组件

。 事件绑定方式 。 服务端渲染

印客学院独家整理 , 盗版必究

**25.2.1.** 避免使用内联函数

如果我们使用内联函数 , 则每次调用 render  函数时都会创建一个新的函数实例 , 如下:

|  |
| --- |
| JS, ca 复制代码 |
| 1 import React from "react";  2   |  | | --- | |  | |  |   3 export default class InlineFunctionComponent extends React.Component {  4 render() {  5 return ( 6 <div>  7 <h1>Welcome Guest</h1>  8 <input type="button" onClick={(e) => { this.setState({inputValue:  e.target.value}) }} value="Click For Inline Function" />  9 </div> 10 ) |
| 11 } 12 } |

我们应该在组件内部创建一个函数 , 并将事件绑定到该函数本身 。这样每次调用 render  时就不会创

建单独的函数实例 , 如下 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | JSX LO 复制代码 | | | |
| 1 | import React from "react"; | | | |
| 2 |  | | | |
| 3 | - export default class InlineFunctionComponent extends React.Component { | | | |
| 4 |  | | | |
| 5 | |  | | --- | |  | | setNewStateData = (event) => { |  |  |
| 6 | |  | | --- | |  | | this.setState({ | | |
| 7 | inputValue: e.target.value | | | |
| 8 | }) | | | |
| 9 | } | | | |
| 10 |  | | | |
| 11 | |  | | --- | |  | | render() { | | |
| 12 | return ( | | | |
| 13 | <div> | | | |
| 14 | <h1>Welcome Guest</h1> | | | |
| 15 | <input type="button" onClick={this.setNewStateData} value="Click F or Inline Function" /> | | | |
| 16 | </div> | | | |
| 17 | ) | | | |
| 18 | } | | | |
| 19 | } | | | |

印客学院独家整理 , 盗版必究

**25.2.2.** 使用 React Fragments 避免额外标记

用户创建新组件时 , 每个组件应具有单个父标签 。父级不能有两个标签 , 所以顶部要有一个公共标签 , 所以我们经常在组件顶部添加额外标签 div 

这个额外标签除了充当父标签之外 , 并没有其他作用 , 这时候则可以使用 fragement  其不会向组件引入任何额外标记 , 但它可以作为父级标签的作用 , 如下所示 :



|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | JS,  a 复制代码 |
| 1 | export default class NestedRoutingComponent extends React.Component { | | |
| 2 | |  | | --- | |  | | render() { | |
| 3 | return ( | | |
| 4 | <> | | |
| 5 | <h1>This is the Header Component</h1> | | |
| 6 | <h2>Welcome To Demo Page</h2> | | |
| 7 | </> | | |
| 8 | ) | | |
| 9 | } | | |
| 10 | } | | |

**25.2.3.** 事件绑定方式

在事件绑定方式中 , 我们了解到四种事假绑定的方式

从性能方面考虑 , 在 render  方法中使用 bind  和 render  方法中使用箭头函数这两种形式在每次组 件 render  的时候都会生成新的方法实例 , 性能欠缺

而 constructor  中 bind  事件与定义阶段使用箭头函数绑定这两种形式只会生成  个方法实例 , 性

能方面会有所改善

**25.2.4.** 使用 Immutable

我们了解到使用 Immutable  可以给 React  应用带来性能的优化 , 主要体现在减少渲染的次数

在做 react  性能优化的时候 , 为了避免重复渲染 , 我们会在 shouldComponentUpdate()  中做对 比 , 当返回 true  执行 render  方法

Immutable  通过 is  方法则可以完成对比 , 而无需像一样通过深度比较的方式比较

**25.2.5.** 懒加载组件

印客学院独家整理 , 盗版必究

从工程方面考虑 , webpack  存在代码拆分能力 , 可以为应用创建多个包 , 并在运行时动态加载 , 减少 初始包的大小

而在 react  中使用到了 Suspense  和 lazy  组件实现代码拆分功能 , 基本使用如下:

|  |
| --- |
| JS, a 复制代码 |
| 1 const johanComponent = React.lazy(() => import(/\* webpackChunkName: "johanC omponent" \*/ './myAwesome.component'));  6 </React >  2  3 export const johanAsyncComponent = props => (  4 <React.Suspense fallback={<Spinner />}>  5 <johanComponent {...props} /> |
| 7 );  .Suspense |



**25.2.6.** 服务端渲染

采用服务端渲染端方式 , 可以使用户更快的看到渲染完成的页面

服务端渲染 , 需要起-个 node  服务 , 可以使用

express 

、 koa  等 , 调用 react  的

 renderToS

tring  方法 , 将根组件渲染成字符串 , 再输出到响应中 例如 :

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 | import { renderToString } from "react-dom/server"; |  |
| 2 | import MyPage from "./MyPage"; |  |
| 3 | app.get("/", (req, res) => { |  |
| 4 | res.write("<!DOCTYPE html><html><head><title>My Page</title></ >"); | head><body |
| 5 | res.write("<div id='content'>"); |  |
| 6 | res.write(renderToString(<MyPage/>)); |  |
| 7 | res.write("</div></body></html>"); |  |
| 8 | res.end(); |  |
| 9 | }); |  |

客户端使用render方法来生成HTML

|  |  |
| --- | --- |
| v JSX | LO 复制代码 |
| 1 import ReactDOM from 'react-dom'; |  |
| 2 import MyPage from "./MyPage"; |  |
| 3 ReactDOM.render(<MyPage />, document.getElementById( 'app')); |  |

印客学院独家整理 , 盗版必究

**25.2.7.** 其他

除此之外 , 还存在的优化手段有组件拆分 、合理使用 hooks  等性能优化手段



**25.3.** 总结



通过上面初步学习 , 我们了解到 react  常见的性能优化可以分成三个层面 :

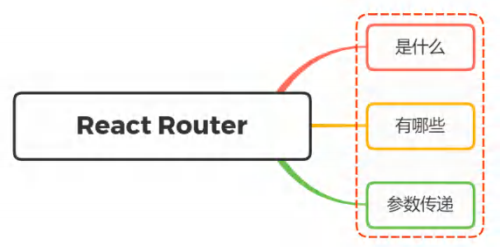
。 代码层面  工程层面

。 框架机制层面

通过这三个层面的优化结合 , 能够使基于 react  项目的性能更上  层楼

**26.** 说说你对React Router的理解? 常用的Router组件

有哪些?



**26.1.** 是什么

react-router  等前端路由的原理大致相同 , 可以实现无刷新的条件下切换显示不同的页面

路由的本质就是页面的 URL  发生改变时 , 页面的显示结果可以根据 URL  的变化而变化 , 但是页面不会 刷新

因此 , 可以通过前端路由可以实现单页(SPA)应用 react-router  主要分成了几个不同的包 :

印客学院独家整理 , 盗版必究

o react-router: 实现了路由的核心功能

 react-router-dom : 基于 react-router, 加入了在浏览器运行环境下的 一些功能

 react-router-native: 基于 react-router , 加入了 react-native 运行环境下的 一些功能 o react-router-config: 用于配置静态路由的工具库

**26.2.** 有哪些

这里主要讲述的是 react-router-dom  的常用 ,API 主要是提供了  些组件: o BrowserRouter 、 HashRouter

o Route

o Link 、 NavLink o switch

o redirect

**26.2.1.** BrowserRouter 、HashRouter

Router  中包含了对路径改变的监听 , 并且会将相应的路径传递给子组件

history 



模式 ,

模式

BrowserRouter 是

HashRouter

使用两者作为最顶层组件包裹其他组件

印客学院独家整理 , 盗版必究

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | JS | | | X LO 复制代码 |
| 1 | import { BrowserRouter as Router } from "react-router-dom"; | | |  |
| 2 |  | | |  |
| 3 | export default function App() { | | |  |
| 4 | return ( | |  |  |
| 5 | <Router> | |  |  |
| 6 | <main> | |  |  |
| 7 | <nav> | |  |  |
| 8 | <ul> | | |  |
| 9 | <li> |  | |  |
| 10 | < a href=" ">Home</ a> | | |  |
| 11 | </li> | | |  |
| 12 | <li> | | |  |
| 13 | < a href="/about">About</ a> | | |  |
| 14 | </li> | | |  |
| 15 | <li> | | |  |
| 16 | < a href="/contact">Contact</ a> | | |  |
| 17 | </li> | | |  |
| 18 | </ul> | | |  |
| 19 | </nav> | | |  |
| 20 | </main> | | |  |
| 21 | </Router> | | |  |
| 22 | ); | | |  |
| 23 | } | | |  |

**26.2.2.** Route

Route  用于路径的匹配 , 然后进行组件的渲染 , 对应的属性如下:  path 属性 : 用于设置匹配到的路径

o component 属性: 设置匹配到路径后 , 渲染的组件 o render 属性: 设置匹配到路径后 , 渲染的内容

o exact 属性: 开启精准匹配 , 只有精准匹配到完全一致的路径 , 才会渲染对应的组件

印客学院独家整理 , 盗版必究

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | JSX LO 复制代码 | | | |
| 1 | import { BrowserRouter as Router, Route } from "react-router-dom"; | | | |
| 2 |  | | | |
| 3 | export default function App() { | | | |
| 4 | return ( | |  | |
| 5 | <Router> | |  | |
| 6 | <main> | |  |  |
| 7 | <nav> | |  |  |
| 8 | <ul> | | | |
| 9 | <li> |  | | |
| 10 | < a href="/">Home</ a> | | | |
| 11 | </li> | | | |
| 12 | <li> | | | |
| 13 | < a href="/about">About</ a> | | | |
| 14 | </li> | | | |
| 15 | <li> | | | |
| 16 | < a href="/contact">Contact</ a> | | | |
| 17 | </li> | | | |
| 18 | </ul> | | | |
| 19 | </nav> | | | |
| 20 | <Route path="/" render={() => <h1>Welcome!</h1>} /> | | | |
| 21 | </main> | | | |
| 22 | </Router> | | | |
| 23 | ); | | | |
| 24 | } |  | | |



**26.2.3.** Link、NavLink

href

通常路径的跳转是使用 Link  组件 , 最终会被渲染成 a  元素 , 其中属性 to  代替 a  标题的 属

性

NavLink  是在 Link  基础之上增加了  些样式属性 , 例如组件被选中时 , 发生样式变化 , 则可以设 置 NavLink  的 -下属性 :

 activestyle : 活跃时 (匹配时) 的样式

 activeclassName: 活跃时添加的class 如下 :

|  |  |  |
| --- | --- | --- |
| v Javascript |  | 复制代码 |
| 1 <NavLink to="/" exact activeStyle={{color: "red"}}>首页</NavLink> | |  |
| 2 <NavLink to="/about" activeStyle={{color: "red"}}>关于</NavLink> | |  |
| 3 <NavLink to="/profile" activeStyle={{color: "red"}}>我的</NavLink> | |  |

印客学院独家整理 , 盗版必究

如果需要实现 js  实现页面的跳转 , 那么可以通过下面的形式 :



通过 Route  作为顶层组件包裹其他组件后, 页面组件就可以接收到 -些路由相关的东西 , 比如  props.

history 

|  |  |
| --- | --- |
|  | JSX COL 复制代码 |
| 1 | const Contact = ({ history }) => ( |
| 2 | <Fragment> |
| 3 | <h1>Contact</h1> |
| 4 | <button onClick={() => history.push("/")}>Go to home</button> |
| 5 | <FakeText /> |
| 6 | </Fragment> |
| 7 | ) |

 props  中接收到的 history  对象具有 些方便的方法  如 ·goBack ·goForward

push

**26.2.4.** redirect

用于路由的重定向 , 当这个组件出现时 , 就会执行跳转到对应的 to  路径中 , 如下例子:

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 | const About = ({ |  |
| 2  3  4  5  6  7  8  9  10  11  12 | match: {  params: { name }, },  }) => (  // props.match.params.name  <Fragment>  {name !== "tom" ? <Redirect to="/" /> : null} <h1>About {name}</h1>  <FakeText /> </Fragment>  ) |  |

上述组件当接收到的路由参数 name  不等于 tom  的时候 , 将会自动重定向到首页



**26.2.5.** switch



swich  组件的作用适用于当匹配到第一个组件的时候 , 后面的组件就不应该继续匹配 如下例子 :

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | JSX | LO 复制代码 |
| 1 | <Switch> |  |
| 2 | <Route exact path="/" component={Home} /> |  |
| 3  4  5  6  7 | <Route path="/about" component={About} />  <Route path="/profile" component={Profile} /> <Route path="/:userid" component={User} />  <Route component={NoMatch} />  </Switch> |  |

 switch  组件进行包裹 除了 一些路由相关的组件之外 ,

如果不使用

hooks

react-router 还提供一些

, 如下 :

useHistory

useparams

useLocation

**26.2.6.** useHistory

useHistory  可以让组件内部直接访问

history 

, 无须通过 props  获取

|  |  |  |
| --- | --- | --- |
|  | Javascript B 复制代码 | |
| 1  2  3  4  5  6  7  8  9  10  11 | |  | | --- | |  | | import { useHistory } from "react-router-dom";  const Contact = () => {  const history = useHistory(); return (  <Fragment>  <h1>Contact</h1>  <button onClick={() => history.push("/")}>Go </Fragment>  to home</button>  ); }; |



**26.2.7.** useparams

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | JSX | LO 复制代码 |
| 1 | const About = () => { |  |
| 2 | const { name } = useParams(); |  |
| 3  4  5  6  7  8  9  10  11 | return (  // props.match.params.name  <Fragment>  {name !== "John Doe" ? <Redirect to="/" /> : null} <h1>About {name}</h1>  <Route component={Contact} /> </Fragment>  ); }; |  |



**26.2.8.** useLocation

 useLocation  会返回当前

URL  的 location  对象

|  |  |  |
| --- | --- | --- |
|  | JSX | LO 复制代码 |
| 1 | import { useLocation } from "react-router-dom"; |  |
| 2 |  |  |
| 3 | const Contact = () => { |  |
| 4 | const { pathname } = useLocation(); |  |
| 5  6  7  8  9  10  11  12 | return (  <Fragment>  <h1>Contact</h1>  <p>Current URL: {pathname}</p > </Fragment>  ); }; |  |

**26.3.** 参数传递

这些路由传递参数主要分成了三种形式 :

• 动态路由的方式 o search传递参数 o to传入对象

**26.3.1.** 动态路由

印客学院独家整理 , 盗版必究

动态路由的概念指的是路由中的路径并不会固定

 path  Route  /detail/:id  

可以匹配到该

 /detail/abc 



/detail/123

都



Route

|  |  |
| --- | --- |
|  | JSX LO 复制代码 |
| 1 | <NavLink to="/detail/abc123">详情</NavLink> |
| 2 |  |
| 3 | <Switch> |
| 4 | ... 其他Route |
| 5 | <Route path="/detail/:id" component={Detail}/> |
| 6 | <Route component={NoMatch} /> |
| 7 | </Switch> |

获取参数方式如下:

|  |  |  |
| --- | --- | --- |
|  | JS | X LO 复制代码 |
| 1 console.log(props.match.params.xxx) |  |  |

**26.3.2.** search传递参数

在跳转的路径中添加了 一些query参数;

|  |  |
| --- | --- |
|  | JSX LO 复制代码 |
| 1 | <NavLink to="/detail2?name=why&age=18">详情2</NavLink> |
| 2 |  |
| 3 | <Switch> |
| 4 | <Route path="/detail2" component={Detail2}/> |
| 5 | </Switch> |

获取形式如下:

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 console.log(props.location.search) |  |  |



**26.3.3.** to传入对象

传递方式如下:

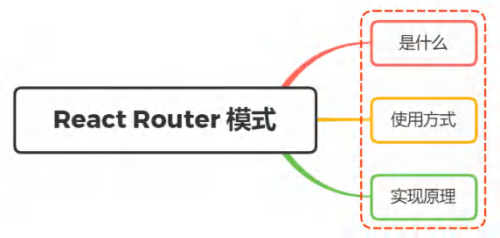
印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | JS | X B 复制代码 |
| 1 | <NavLink to={{ |  |  |
| 2 | pathname: "/detail2", |  |  |
| 3  4  5  6  7 | query: {name: "kobe", state: {height: 1.98, search: "?apikey=123"  }}> 详情2 | age: 30},  address: "洛杉矶 "}, |  |
| 8 | </NavLink> |  |  |

获取参数的形式如下 :

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 console.log(props.location) |  |  |

**27.** 说说React Router有几种模式? 实现原理?



**27.1.** 是什么

在单页应用中 , -个 web  项目只有-个 html  页面 , -旦页面加载完成之后 , 就不用因为用户的操作 而进行页面的重新加载或者跳转 , 其特性如下 :

。 改变 url 且不让浏览器像服务器发送请求

 在不刷新页面的前提下动态改变浏览器地址栏中的URL地址 其中主要分成了两种模式:

 hash 模式: 在url后面加上# , 如http://127 . 0 . 0 . 1:5500/home/#/page1

印客学院独家整理 , 盗版必究

 history 模式 : 允许操作浏览器的曾经在标签页或者框架里访问的会话历史记录

**27.2.** 使用

React Router  对应的 hash  模式和 history  模式对应的组件为 :

o HashRouter

o BrowserRouter

这两个组件的使用都分的简单 , 作为最顶层组件包裹其他组件 , 如下所示

|  |  |  |
| --- | --- | --- |
|  |  | JSX LO 复制代码 |
| 1 // 1.import { BrowserRouter as Router } from "react-router-dom"; 2 // 2.import { HashRouter as Router } from "react-router-dom";  3  4 import React from 'react';  5 import {  6 BrowserRouter as Router, | | |
| 8 Switch,  7 // HashRouter as Router  9 Route,  10 } from "react-router-dom";  11 import Home from './pages/Home';  12 import Login from './pages/Login';  13 import Backend from './pages/Backend';  14 import Admin from './pages/Admin'; 15  16  17  function App() {  18 return (  19 <Router>  20 <Route path="/login" component={Login}/>  21 <Route path="/backend" component={Backend}/>  22 <Route path="/admin" component={Admin}/>  23 <Route path="/" component={Home}/>  24 </Router> 25 );  26 } 27  28 export default App; | | |

**27.3.** 实现原理

印客学院独家整理 , 盗版必究

路由描述了 URL  与 UI  之间的映射关系 , 这种映射是单向的 , 即 URL 变化引起 UI 更新 (无需刷新 页面)

下面以 hash  模式为例子 , 改变 hash  值并不会导致浏览器向服务器发送请求 , 浏览器不发出请求 , 也就不会刷新页面

 hash  值改变 , 触发全局 window  对象上的 hashchange  事件 。所以 hash 模式路由就是利 用 hashchange  事件监听 URL  的变化  从而进行 DOM  操作来模拟页面跳转

 react-router  也是基于这个特性实现路由的跳转 下面以 HashRouter  组件分析进行展开 :

**27.4.** HashRouter

HashRouter  包裹了整应用 ,

通过 window.addEventListener('hashChange',callback)  监听 hash  值的变化 , 并传递给

其嵌套的组件

然后通过 context  将 location  数据往后代组件传递 , 如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | JSX | | | | LO 复制代码 |
| 1 | import React, { Component } from 'react'; | | | |  |
| 2 | import { Provider } from './context' | | | |  |
| 3 | // 该组件下Api提供给子组件使用 | | | |  |
| 4 | class HashRouter extends Component { | | |  |  |
| 5 | |  | | --- | |  | | constructor() { | |  |  |
| 6 | super() | |  |  |  |
| 7 | |  | | --- | |  | | this.state = { |  |  |  |
| 8 | |  | | --- | |  | | location: { | | |  |
| 9 | pathname: window.location.hash.slice(1) || '/' | | | |  |
| 10 | } | | | |  |
| 11 | } | | | |  |
| 12 | } | | | |  |
| 13 | // url路径变化 改变location | | | |  |
| 14 | |  | | --- | |  | | componentDidMount() { | | |  |
| 15 | window.location.hash = window.location.hash || '/' | | | |  |
| 16 | |  | | --- | |  | | window.addEventListener( 'hashchange', () => { | | |  |
| 17 | |  | | --- | |  | | this.setState({ | | |  |
| 18 | |  | | --- | |  | | location: { | | |  |
| 19 | ...this.state.location, | | | |  |
| 20 | pathname: window.location.hash.slice(1) || '/' | | | |  |
| 21 | } | | | |  |
| 22 | }, () => console.log(this.state.location)) | | | |  |
| 23 | }) | | | |  |
| 24  25  26  27  28  29  30  31  32  33  34  35  36  37 | }  render() {   |  | | --- | |  | |  |   let value = {  location: this.state.location }  return (  <Provider value={value}> {  this.props.children  }  </Provider>  ); }  } | | | |  |
| 38 |  | | | |  |
| 39 | export default HashRouter; | | | |  |



**27.4.1.** Router

印客学院独家整理 , 盗版必究

 Router  组件主要做的是通过 BrowserRouter  传过来的当前值 , 通过 props  传进来的 path  与  context  传进来的 pathname  进行匹配 , 然后决定是否执行渲染组件

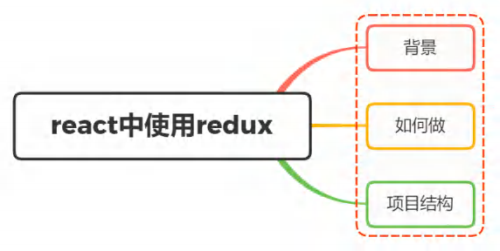
|  |  |
| --- | --- |
| v Javascript B 复制代码 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import React, { Component } from 'react'; import { Consumer } from './context'  const { pathToRegexp } = require("path-to-regexp");  class Route extends Component { render() {   |  | | --- | |  | |  |   return (  <Consumer>  {  state => {  console.log(state)  let {path, component: Component} = this.props let pathname = state.location.pathname  let reg = pathToRegexp(path, [], {end: false}) // 判断当前path是否包含pathname  if(pathname.match(reg)) { |
| return <Component></Component> }  16  17  18  19  20  21  22  23  24  25  return null  } }  </Consumer> );  } }  export default Route; | |



**28.** 你在React项目中是如何使用Redux的? 项目结构是

如何划分的?

印客学院独家整理 , 盗版必究



**28.1.** 背景

在前面文章了解中 , 我们了解到 redux  是用于数据状态管理 , 而 react  是  个视图层面的库

如果将两者连接在一起 , 可以使用官方推荐 react-redux  库 , 其具有高效且灵活的特性 react-redux  将组件分成 :

• 容器组件: 存在逻辑处理

 UI 组件 : 只负责现显示和交互 , 内部不处理逻辑 , 状态由外部控制

通过 redux  将整个应用状态存储到 store  中 , 组件可以派发 dispatch  其他组件通过订阅 store  中的状态 state  来更新自身的视图

store

行为 action  给



**28.2.** 如何做

使用 react-redux  分成了两大核心: o provider

o connection

**28.2.1.** provider

在 redux  中存在-个 store  用于存储 ,state 如果将这个 store  存放在顶层元素中 , 其他组件

都被包裹在顶层元素之上

那么所有的组件都能够受到 redux  的控制 , 都能够获取到 redux  中的数据 使用方式如下

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
| v Javascript | | LO 复制代码 |
| 1 <Provider store = {store}> | |  |
| 2 <App /> | |  |
| 3 <Provider> |  |  |

**28.2.2.** connection

 connect  方法将 store  上的 getState  和 dispatch  包装成组件的 props  导入 conect  如下:

|  |  |  |
| --- | --- | --- |
|  | Javascript | COL 复制代码 |
| 1 import { connect } from "react-redux"; |  |  |

用法如下:

|  |  |
| --- | --- |
| v Javascript | LO 复制代码 |
| 1 connect(mapStateToProps, mapDispatchToProps)(MyComponent) |  |

可以传递两个参数:

mapstateToprops

mapDispatchToprops

**28.2.3.** mapstateToprops

redux 中的数据映射到

react 中的

中去

把

props 

如下

|  |  |  |  |
| --- | --- | --- | --- |
|  | JSX | | LO 复制代码 |
| 1 | const mapStateToProps = (state) => { | |  |
| 2 | |  | | --- | |  | | return { |  |
| 3 | // prop : state.xxx | 意思是将state中的某个数据映射到props中 | |  |
| 4 | foo: state.bar | |  |
| 5 | } | |  |
| 6 | } | |  |

组件内部就能够通过 props  获取到 store  中的数据

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | plain Text | LO 复制代码 |
| 1 | class Foo extends Component { |  |
| 2 | constructor(props){ |  |
| 3  4  5  6  7  8  9  10  11  12 | super(props);  }  render(){ return(  // 这样子渲染的其实就是state.bar的数据了 <div>this.props.foo</div>  )  }  }  connect()(Foo)  Foo = |  |
| 13 | export default Foo |  |



**28.2.4.** mapDispatchToprops

将 redux  中的 dispatch  映射到组件内部的

中

props 

|  |  |  |
| --- | --- | --- |
|  | JSX LO 复制代码 | |
| 1 | const mapDispatchToProps = (dispatch) => { // 默认传递参数就是dispatch | |
| 2 | |  | | --- | |  | | return { |
| 3  4  5  6  7  8  9 | onClick: () => { dispatch({   |  | | --- | |  | |  |   type: 'increatment'  });  } };  } | |



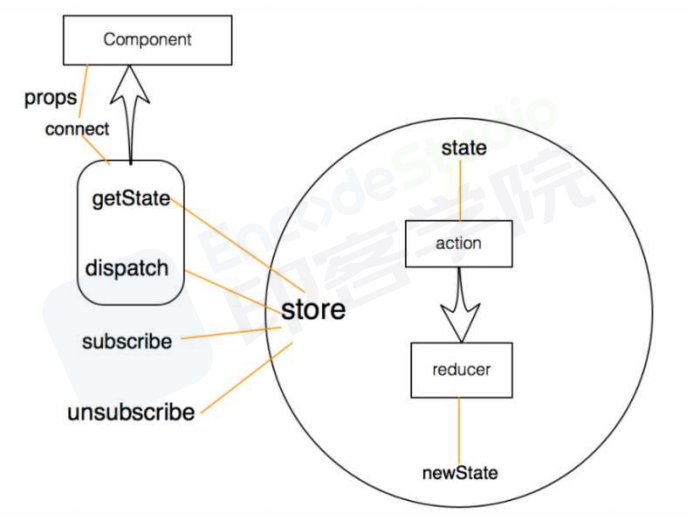
印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | Javascript B 复制代码 | | |
| 1 | class Foo extends Component { | | |
| 2 | |  | | --- | |  | | | constructor(props){ |
| 3  4  5  6  7  8  9  10  11  12 | |  | | --- | |  | | super(props); }  render(){ return(  <button onClick = {this.props.onClick}>点击increase</button> )  } }  Foo = connect()(Foo); | |
| 13 | export default Foo; | | |

**28.3.** 小结

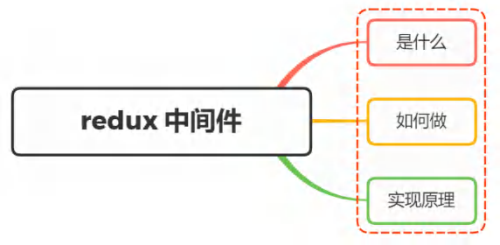
整体流程图大致如下所示:

印客学院独家整理 , 盗版必究



**29.** 说说对Redux中间件的理解? 常用的中间件有哪

些? 实现原理?



印客学院独家整理 , 盗版必究

**29.1.** 是什么

中间件 (Middleware) 是介于应用系统和系统软件之间的 一类软件 , 它使用系统软件所提供的基础服务

(功能) , 衔接网络上应用系统的各个部分或不同的应用 , 能够达到资源共享 、功能共享的目的

那么如果需要支持异步操作 , 或者支持错误处理 、 日志监控 , 这个过程就可以用上中间件

Redux  中 , 中间件就是放在就是在 dispatch  过程 , 在分发 action  进行拦截处理 , 如下图 :



其本质上-个函数 , 对 store.dispatch  方法进行了改造 , 在发出 Action  和执行 Reduce r  这 两步之间 , 添加了其他功能

**29.2.** 常用的中间件

有很多优秀的 redux  中间件 , 如 : o redux-thunk: 用于异步操作 o redux-logger: 用于日志记录

上述的中间件都需要通过 applyMiddlewares  进行注册 , 作用是将所有的中间件组成-个数组 , 依次 执行

然后作为第二个参数传入到 createStore  中

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
| v Javascript | | | LO 复制代码 |
| 1 const store = createStore( | | |  |
| 2 reduce r, | | |  |
| 3  4 | applyMiddleware(thunk, logger) ); |  |  |



**29.2.1.** redux-thunk

redux-thunk  是官网推荐的异步处理中间件

 dispatch(action)  action    JavaScript 

redux-thunk  中间件会判断你当前传进来的数据类型 , 如果是  个函数 , 将会给函数传入参数值 (dispatch , getstatE)

 dispatch函数用于我们之后再次派发action

o getstate函数考虑到我们之后的 一些操作需要依赖原来的状态 , 用于让我们可以获取之前的一些状 态

 dispatch 

|  |
| --- |
|  |
|  |
|  |

|  |  |
| --- | --- |
|  | Javascript B 复制代码 |
| 1 | const getHomeMultidataAction = () => { |
| 2 | return (dispatch) => { |
| 3 | axios.get("http://xxx.xx.xx.xx/test").then(res => { |
| 4 | const data = res.data.data; |
| 5 | dispatch(changeBannersAction(data.banner.list)); |
| 6 | dispatch(changeRecommendsAction(data.recommend.list)); |
| 7 | }) |
| 8 | } |
| 9 | } |



**29.2.2.** redux-logger

 redux-logger 

如果想要实现一个日志功能 , 则可以使用现成的

印客学院独家整理 , 盗版必究

|  |  |
| --- | --- |
| v Javascript | LO 复制代码 |
| 1 |  |
| 2 import { applyMiddleware, createStore } from 'redux'; |  |
| 3 import createLogger from 'redux-logger';  4 const logger = createLogger(); 5  6 const store = createStore(  7 reduce r,  8 applyMiddleware(logger) 9 ); |  |

这样我们就能简单通过中间件函数实现a志记录的信息 

**29.3.** 实现原理



首先看看

的源码



applyMiddlewares

|  |  |  |
| --- | --- | --- |
|  | Javascript B 复制代码 | |
| 1 | export default function applyMiddleware(...middlewares) { | |
| 2 | |  | | --- | |  | | return (createStore) => (reduce r, preloadedState, enhancer) => { |
| 3 | var store = createStore(reduce r, preloadedState, enhancer); | |
| 4 | var dispatch = store.dispatch; | |
| 5 | var chain = []; | |
| 6 |  | |
| 7 | |  | | --- | |  | | var middlewareAPI = { |
| 8 | getState: store.getState, | |
| 9 | dispatch: (action) => dispatch(action) | |
| 10 | }; | |
| 11 | chain = middlewares.map(middleware => middleware(middlewareAPI)); | |
| 12 | dispatch = compose(...chain)(store.dispatch); | |
| 13 |  | |
| 14 | return {...store, dispatch} | |
| 15 | } | |
| 16 | } | |

所有中间件被放进了 -个数组 ,chain 然后嵌套执行 , 最后执行 。store.dispatch 可以看到 , 中 间件内部 ( )middlewareAPI 可以拿到 getState  和 dispatch  这两个方法

在上面的学习中 , 我们了解到了 redux-thunk  的基本使用

内部会将 dispatch  进行-个判断 , 然后执行对应操作 , 原理如下 :

印客学院独家整理 , 盗版必究

|  |  |  |  |
| --- | --- | --- | --- |
|  | Javascript | | LO 复制代码 |
| 1 | function patchThunk(store) { | |  |
| 2 | let next = store.dispatch; | |  |
| 3  4  5  6  7  8  9  10  11  12 |  | function dispatchAndThunk(action) {  if (typeof action === "function") {  action(store.dispatch, store.getState); } else {  next(action);  } }  store.dispatch = dispatchAndThunk; |  |
| 13 | } | |  |

实现一个日志输出的原理也非常简单 , 如下 :

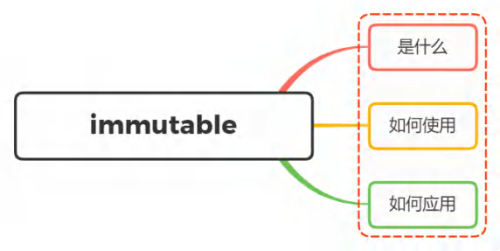
|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 | let next = store.dispatch; |  |
| 2 |  |  |
| 3 | function dispatchAndLog(action) { |  |
| 4 | console.log("dispatching:", addAction(10)); |  |
| 5 | next(addAction(5)); |  |
| 6  7  8  9 | console.log("新的state:", store.getState()); }  store.dispatch = dispatchAndLog; |  |



**30.** 说说你对immutable的理解? 如何应用在react项目

中?

印客学院独家整理 , 盗版必究



**30.1.** 是什么

Immutable, 不可改变的 , 在计算机中 , 即指一旦创建 , 就不能再被更改的数据

对 Immutable  对象的任何修改或添加删除操作都会返回-个新的 Immutable  对象 Immutable  实现的原理是 Persistent Data Structure  (持久化数据结构) :

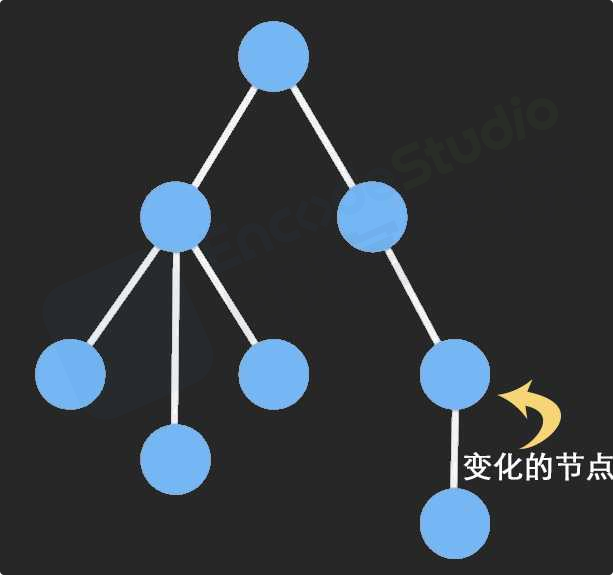
 用 一种数据结构来保存数据

 当数据被修改时 , 会返回 一个对象 , 但是新的对象会尽可能的利用之前的数据结构而不会对内存造 成浪费

也就是使用旧数据创建新数据时 , 要保证旧数据同时可用且不变 , 同时为了避免 deepCopy  把所有节

点都复制 一遍带来的性能损耗 , Immutable  使用了 (StructuralSharing结构共享) 如果对象树中 一个节点发生变化 , 只修改这个节点和受它影响的父节点 , 其它节点则进行共享 如下图所示 :

印客学院独家整理 , 盗版必究



**30.2.** 如何使用

使用 Immutable  对象最主要的库是 immutable.js 

immutable.js 是一个完全独立的库 , 无论基于什么框架都可以用它

其出现场景在于弥补 Javascript 没有不可变数据结构的问题 , 通过 structural sharing来解决的性能问题

内部提供了  套完整的 persistent Data structure , 还有很多易用的数据类型 , 如 、



L

Collection

、 Map Set Record ,Seq 其中 :

ist 

o List: 有序索引集 , 类似 Javascript 中的 Array

 Map: 无序索引集 , 类似 Javascript 中的 object  set: 没有重复值的集合

主要的方法如下 :

印客学院独家整理 , 盗版必究

o fromJS() : 将一个js数据转换为Immutable类型的数据

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 const obj = Immutable.fromJS({a: '123',b: '234'}) |  |  |

toJS() : 将一个Immutable数据转换为JS类型的数据 is() : 对两个对象进行比较

|  |
| --- |
| v Javascript B 复制代码 |
| 1 import { Map, is } from 'immutable'  2 const map1 = Map({ a: 1, b: 1, c: 1 })  3 const map2 = Map({ a: 1, b: 1, c: 1 })  4 map1 === map2 //false  5 Object.is(map1, map2) // false |
| 6 is(map1, map2) // true |

o gEt(key) : 对数据或对象取值

o getIn([J) : 对嵌套对象或数组取值 , 传参为数组 , 表示位置

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1  2  3  4  5  6  7 | let abs = Immutable.fromJS({a: {b:2}}); abs.getIn( [ 'a', 'b']) // 2  abs.getIn( [ 'a', 'c']) // 子级没有值  let arr = Immutable.fromJS( [1 ,2, 3, {a: 5}]); arr.getIn( [3, 'a']); // 5  arr.getIn( [3, 'c']); // 子级没有值 |  |

如下例子 : 使用方法如下 :

|  |  |
| --- | --- |
|  | Javascript B 复制代码 |
| 1  2 | import Immutable from 'immutable'; foo = Immutable.fromJS({a: {b: 1}}); |
| 3 | bar = foo.setIn( [ 'a', 'b'], 2); // 使用 setIn 赋值 |
| 4 | console.log(foo.getIn( [ 'a', 'b'])); // 使用 getIn 取值 打印 1 |
| 5 | console.log(foo === bar); // 打印 false |

如果换到原生的 js  , 则对应如下:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| v Javascript | | | | LO 复制代码 |
| 1 let foo = {a: {b: 1}}; | | | |  |
| 2 let bar = foo; | | | |  |
| 3  4  5 | bar.a.b = 2;  console.log(foo.a.b); // 打印 console.log(foo === bar); // | 2  打印 true |  |  |



**30.3.** 在React中应用



使用 Immutable  可以给 React  应用带来性能的优化 , 主要体现在减少渲染的次数

在做 react  性能优化的时候 , 为了避免重复渲染 , 我们会在 shouldComponentUpdate()  中做对 比 , 当返回 true  执行 render  方法

Immutable  通过 is  方法则可以完成对比 , 而无需像一样通过深度比较的方式比较

在使用 redux  过程中也可以结合 ,Immutable 不使用 Immutable  前修改-个数据需要做  个深 拷贝

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | JS | X LO 复制代码 |
| 1 | import '\_ ' from 'lodash'; | |  |  |
| 2 |  | |  |  |
| 3 | const Component = React.createClass({ | |  |  |
| 4 | |  | | --- | |  | | getInitialState() { |  |  |
| 5 | |  | | --- | |  | | return { |  |  |
| 6 | data: { times: 0 } | |  |  |
| 7 | } | |  |  |
| 8 | }, | |  |  |
| 9 | |  | | --- | |  | | handleAdd() { |  |  |
| 10 | let data = \_.cloneDeep(this.state.data); | |  |  |
| 11 | data.times = data.times + 1; | |  |  |
| 12 | this.setState({ data: data }); | |  |  |
| 13 | } | |  |  |
| 14 | } | |  |  |

使用 Immutable 后:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | JSX | | | LO 复制代码 |
| 1 | getInitialState() { | | |  |
| 2 | |  | | --- | |  | | | return { |  |
| 3  4  5  6  7  8  9  10 | |  | | --- | |  | | data: Map({ times: 0 }) }  },  handleAdd() {  this.setState({ data: this.state.data.update( 'times', v => v  // 这时的 times 并不会改变  console.log(this.state.data.get( 'times')); } | | + 1) }); |



同理 , 在

中也可以将数据进行

fromJS 处理



redux

印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Javascript B 复制代码 | | | | |
| 1 | import \* as constants from './constants' | | | | |
| 2 | import {fromJS} from 'immutable' | | | | |
| 3 | const defaultState = fromJS({ //将数据转化成immutable数据 | | | | |
| 4 | home:true, | | |  | |
| 5 | focused:false, | | |  | |
| 6 | mouseIn:false, | | |  | |
| 7 | list: [], | | |  |  |
| 8 | page:1, | | | | |
| 9 | totalPage:1 | |  | | |
| 10 | }) | | | | |
| 11 | export default(state=defaultState,action)=>{ | | | | |
| 12 | |  | | --- | |  | | switch(action.type){ | | | |
| 13 | case constants.SEARCH\_FOCUS: | | | | |
| 14 | return state.set( 'focused',true) //更改immutable数据 | | | | |
| 15 | case constants.CHANGE\_HOME\_ACTIVE: | | | | |
| 16 | return state.set( 'home',action.value) | | | | |
| 17 | case constants.SEARCH\_BLUR: | | | | |
| 18 | return state.set( 'focused',false) | | | | |
| 19 | case constants.CHANGE\_LIST: | | | | |
| 20 | // return state.set('list',action.data).set('totalPage',actio n.totalPage) | | | | |
| 21 | //merge效率更高 执行-次改变多个数据 | | | | |
| 22 | |  | | --- | |  | | return state.merge({ | | | |
| 23 | list:action.data, | | | | |
| 24 | totalPage:action.totalPage | | | | |
| 25 | }) | | | | |
| 26 | case constants.MOUSE\_ENTER: | | | | |
| 27 | return state.set( 'mouseIn',true) | | | | |
| 28 | case constants.MOUSE\_LEAVE: | | | | |
| 29 | return state.set( 'mouseIn',false) | | | | |
| 30 | case constants.CHANGE\_PAGE: | | | | |
| 31 | return state.set( 'page',action.page) | | | | |
| 32 | default: | | | | |
| 33 | return state | | | | |
| 34 | } | | | | |
| 35 | } | | | | |

**31.** 说说React服务端渲染怎么做? 原理是什么?

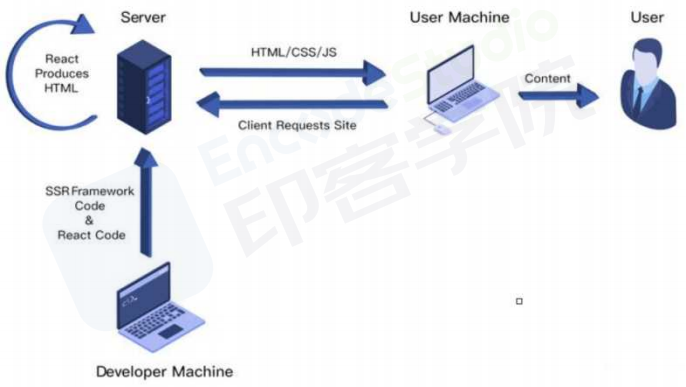
印客学院独家整理 , 盗版必究



**31.1.** 是什么

,Server-SideRendering 简称 ,SSR 意为服务端渲染

指由服务侧完成页面的 HTML  结构拼接的页面处理技术 , 发送到浏览器 , 然后为其绑定状态与事件 , 成为完全可交互页面的过程



其解决的问题主要有两个:

印客学院独家整理 , 盗版必究

 SEO , 由于搜索引擎爬虫抓取工具可以直接查看完全渲染的页面  加速首屏加载 , 解决首屏白屏问题

**31.2.** 如何做

在 react  中 , 实现 SSR  主要有两种形式: • 手动搭建一个 SSR 框架

 使用成熟的SSR 框架 , 如 Next.JS

这里主要以手动搭建一个 SSR  框架进行实现

 HTM

首先通过 express  启动-个 app.js  文件  用于监听3ooo端a的请求  当请求根录时  返回 ,L 如下 :

|  |  |  |  |
| --- | --- | --- | --- |
|  | Javascript | | LO 复制代码 |
| 1 | const express = require( 'express') | |  |
| 2 | const app = express() | |  |
| 3 | app.get( '/', (req,res) => res.send(` | |  |
| 4 | <html> | |  |
| 5 | <head> | |  |
| 6 | <title>ssr demo</title> | |  |
| 7 | </head> | |  |
| 8 | <body> | |  |
| 9 | Hello world | |  |
| 10  11  12  13  14 | </body> </html>  `))  app.listen(3000, () => console.log( 'Exampleapp  app.js | listening on port | 3000!')) |



然后再服务器中编写

react

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | JSX | LO 复制代码 |
| 1 | import React from 'react' |  |  |
| 2 |  |  |  |
| 3 | const Home = () =>{ |  |  |
| 4 |  |  |  |
| 5 | return <div>home</div> |  |  |
| 6 |  |  |  |
| 7 | } |  |  |
| 8 |  |  |  |
| 9 | export default Home |  |  |

印客学院独家整理 , 盗版必究

为了让服务器能够识别 ,JSX 这里需要使用 webpakc  对项目进行打包转换, 创建一个配置文件

pack.server.js 

并进行相关配置 , 如下 :

' '

React  组件解析成 

|  |
| --- |
| v Javascript B 复制代码 |
| 1 const path = require( 'path') //node的path模块  2 const nodeExternals = require( 'webpack-node-externals') 3  4 module.exports = {  5 target: 'node',  6 mode: 'development', //开发模式  7 entry: './app.js', //入。  8 output: { //打包出口  9 filename: 'bundle.js', //打包后的文件名  10 path:path.resolve(\_\_dirname, 'build') //存放到根目录的build文件夹 11 },  12 externals: [nodeExternals()], //保持node中 require的引用方式  13 module: {  14 rules: [{ //打包规则  15 test: /\.js?$/, //对所有js文件进行打包 |
| exclude: /node\_modules/,//不打包node\_modules中的js文件 options: {   |  | | --- | |  | |  |   loader: babel-loader , //使用babel-loader进行打包  16  17  18  19  20  21  22  presets: [ 'react', 'stage-0', [ 'env', {  //loader时额外的打包规则 ,对react,JSX  ES6进  行转换  targets: {  2versions']  browsers: [ 'last  //对主流浏览器最近两个  版本进行兼容  23  } }]]  24  25  } }]  26  27  }  28  } |



接着借助

react-dom 提供了服务端渲染的

方法 , 负责把

renderToString

tml 

印客学院独家整理 , 盗版必究

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Javascript B 复制代码 | | | |
| 1 | import express from 'express' | | | |
| 2 | import React from 'react'//引入React以支持JSX的语法 | | | |
| 3 | import { renderToString } from 'react-dom/server'//引入renderToString方法 | | | |
| 4 | import Home from './src/containers/Home' | | |  |
| 5 |  | | | |
| 6 | const app= express() | |  |  |
| 7 | const content = renderToString(<Home/>) | | |  |
| 8 | app.get( '/',(req,res) => res.send(` | | | |
| 9 | <html> |  | | |
| 10 | <head> |  | | |
| 11 | <title>ssr demo</title> | | | |
| 12 | </head> | | | |
| 13 | <body> | | | |
| 14 | ${content} | | | |
| 15 | </body> | | | |
| 16 | </html> | | | |
| 17 | `)) | | | |
| 18 |  | | | |
| 19 | app.listen(3001, () => console.log( 'Exampleapp listening on port 3001!')) | | | |

上面的过程中 , 已经能够成功将组件渲染到了页面上

但是像一些事件处理的方法 , 是无法在服务端完成 , 因此需要将组件代码在浏览器中再执行一遍 , 这种 服务器端和客户端共用 一套代码的方式就称之为同构

重构通俗讲就是 一套React代码在服务器上运行一遍 , 到达浏览器又运行一遍 :

 服务端渲染完成页面结构

 浏览器端渲染完成事件绑定

浏览器实现事件绑定的方式为让浏览器去拉取 JS  文件执行 , 让 JS  代码来控制 , 因此需要引入  scri

pt  标签

通过 script  标签为页面引入客户端执行的 react  代码 , 并通过 express  的 static  中间件为 

s  文件配置路由 , 修改如下:

印客学院独家整理 , 盗版必究

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Javascript B 复制代码 | | | |
| 1 | import express from 'express' | | | |
| 2 | import React from 'react'//引入React以支持JSX的语法 | | | |
| 3 | import { renderToString } from'react-dom/server'//引入renderToString方法 | | | |
| 4 | import Home from './src/containers/Home' | | |  |
| 5 |  | | | |
| 6 | const app = express() |  | |  |
| 7 | app.use(express.static( 'public')); | |  |  |
| 8 | //使用express提供的static中间件 ,中间件会将所有静态文件的路由指向public文件夹 | | | |
| 9 | const content = renderToString(<Home/>) | | | |
| 10 |  | | | |
| 11 | app.get( '/',(req,res)=>res.send(` | | | |
| 12 | <html> | | | |
| 13 | <head> | | | |
| 14 | <title>ssr demo</title> | | | |
| 15 | </head> | | | |
| 16 | <body> | | | |
| 17 | ${content} | | | |
| 18 | <script s rc="/index.js"></script> | | | |
| 19 | </body> | | | |
| 20 | </html> | | | |
| 21 | `)) | | | |
| 22 |  | | | |
| 23 | app.listen(3001, () =>console.log( 'Example app listening on port 3001!')) | | | |

 webpack.client.js 

react

代码 , 新建

然后再客户端执行以下

作为客户端React代码

webpack 

的

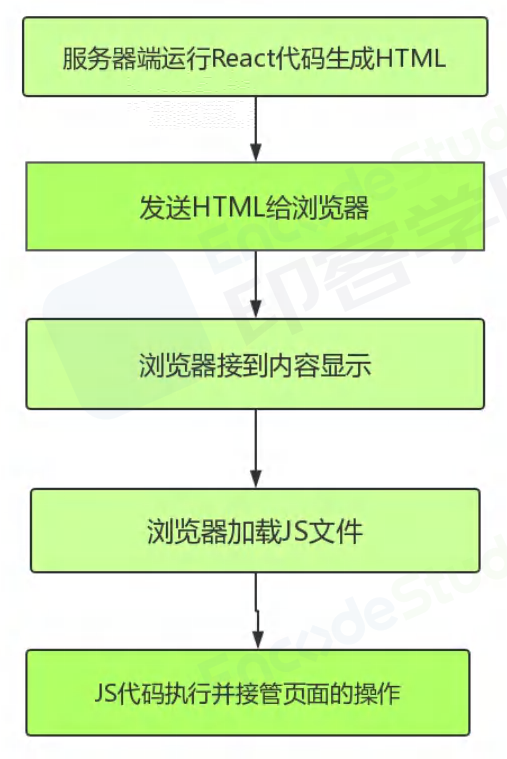
印客学院独家整理 , 盗版必究

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| v Javascript B 复制代码 | | | | | |
| 1 const path = require( 'path') //node的path模块 | | | | | |
| 2 | | | | | |
| 3  module.exports = { | | | | | |
| 4 mode: 'development', //开发模式 | | | | | |
| 5 entry: './src/client/index.js', //入a | | | | | |
| 6  output: { //打包出 | | | | | |
| 7 filename: 'index.js', | | | |  | //打包后的文件名 |
| 8 path:path.resolve( dirname, 'public') //存放到根目录的build文件夹 | | | | | |
| 9 }, | | |  | | |
| 10 | | module: { |  | | |
| 11  rules: [{ //打包规则 | | | | | |
| 12 test: /\.js?$/, //对所有js文件进行打包 | | | | | |
| 13 loader: 'babel-loader', //使用babel-loader进行打包 | | | | | |
| 14 exclude: /node\_modules/, //不打包node\_modules中的js文 | | | | | |
| 件 | | | | | |
| 15  options: { | | | | | |
| 16  presets: [ 'react', 'stage-0', [ 'env', { | | | | | |
| 17 //loader时额外的打包规则 ,这里对react,JSX进行转换 | | | | | |
| 18  targets: { | | | | | |
| 19 browsers: [ 'last 2versions'] //对主流浏览器最近两个 | | | | | |
| 版本进行兼容 | | | | | |
| 20 } | | | | | |
| 21 }]] | | | | | |
| 22  23  24  25 | } }]  } } | |  | | |

 react  服务端渲染 , 过程对应如下图 :

这种方法就能够简单实现首页的

印客学院独家整理 , 盗版必究



在做完初始渲染的时候 , 一个应用会存在路由的情况 , 配置信息如下 :

印客学院独家整理 , 盗版必究

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 | import React from 'react' //引入React以支持JSX |  |
| 2 | import { Route } from 'react-router-dom' //引入路由 |  |
| 3  4  5  6  7  8  9 | import Home from './containers/Home' //引入Home组件  export default (  <div>  <Route path="/" exact component={Home}></Route> </div>  ) |  |

 index.js 

引用路由信息 , 如下:

然后可以通过

|  |  |  |
| --- | --- | --- |
|  | Javascript | LO 复制代码 |
| 1 | import React from 'react' |  |
| 2 | import ReactDom from 'react-dom' |  |
| 3 | import { BrowserRouter } from'react-router-dom' |  |
| 4 | import Router from '../Routers' |  |
| 5 |  |  |
| 6 | const App= () => { |  |
| 7 | return ( |  |
| 8 | <BrowserRouter> |  |
| 9 | {Router} |  |
| 10  11  12  13  14 | </BrowserRouter>  ) }  ReactDom.hydrate(<App/>, document.getElementById( 'root')) |  |

Route 组件外面包裹着一层

这时候控制台会存在报错信息 , 原因在于每个

div , 但服务端返回的代

div

码中并没有这个



解决方法只需要将路由信息在服务端执行一遍 , 使用使用

来替

StaticRouter



context 进行参数传递

代

, 通过

BrowserRouter

印客学院独家整理 , 盗版必究

|  |  |
| --- | --- |
|  | Javascript B 复制代码 |
| 1 | import express from 'express' |
| 2 | import React from 'react'//引入React以支持JSX的语法 |
| 3 | import { renderToString } from 'react-dom/server'//引入renderToString方法 |
| 4 | import { StaticRouter } from 'react-router-dom' |
| 5 | import Router from '../Routers' |
| 6 |  |
| 7 | const app = express() |
| 8 | app.use(express.static( 'public')); |
| 9 | //使用express提供的static中间件 ,中间件会将所有静态文件的路由指向public文件夹 |
| 10 |  |
| 11 | app.get( '/',(req,res)=>{ |
| 12 | const content = renderToString(( |
| 13 | //传入当前path |
| 14 | //context为必填参数 ,用于服务端渲染参数传递 |
| 15 | <StaticRouter location={req.path} context={{}}> |
| 16 | {Router} |
| 17 | </StaticRouter> |
| 18 | )) |
| 19 | res.send(` |
| 20 | <html> |
| 21 | <head> |
| 22 | <title>ssr demo</title> |
| 23 | </head> |
| 24 | <body> |
| 25 | <div id="root">${content}</div> |
| 26 | <script s rc="/index.js"></script> |
| 27 | </body> |
| 28 | </html> |
| 29 | `) |
| 30 | }) |
| 31 |  |
| 32 |  |
| 33 | app.listen(3001, () => console.log( 'Exampleapp listening on port 3001!')) |

这样也就完成了路由的服务端渲染 

**31.3.** 原理

整体 react  服务端渲染原理并不复杂 , 具体如下:

node server  接收客户端请求 , 得到当前的请求 url  路径 , 然后在已有的路由表内查找到对应的 组件 , 拿到需要请求的数据 , 将数据作为 props  、 context  或者 store  形式传入组件

印客学院独家整理 , 盗版必究

然后基于 react  内置的服务端渲染方法 renderToString()  把组件渲染为 html  字符串在把最 终的 html  进行输出前需要将数据注入到浏览器端

浏览器开始进行渲染和节点对比 , 然后执行完成组件内事件绑定和 一些交互 , 浏览器重用了服务端输出

 html  节点 , 整个流程结束

的