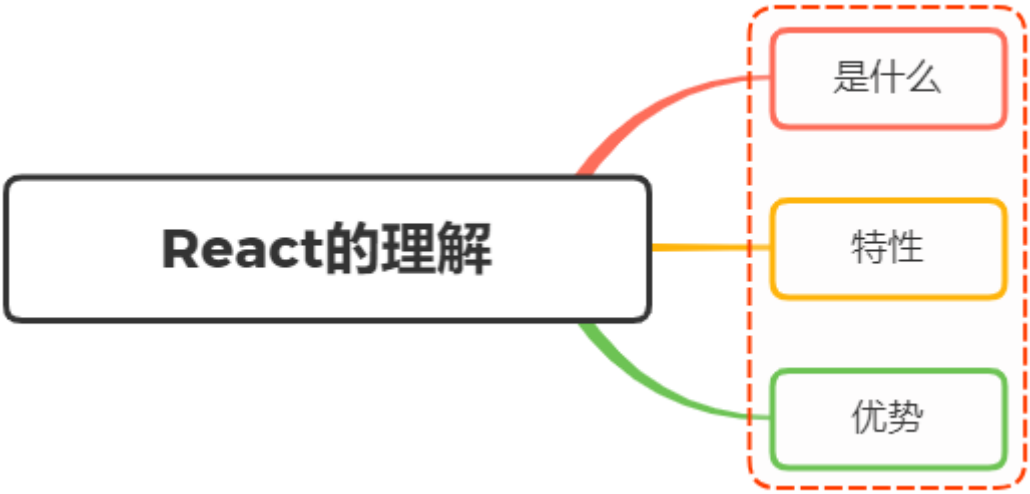
React面试真题（31题）

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，每一个块就是组件，这些组件之间可以组合、嵌套，构成整体页面

`react` 类组件使用一个名为 `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

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
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：

- 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：

```
// 创建地图
const map = new Map.map(document.getElementById("map"), {
  zoom: 4,
  center: { lat, lng },
});
// 创建标记
const marker = new Map.marker({
  position: { lat, lng },
  title: "Hello Marker",
});
// 地图上添加标记
marker.setMap(map);
```

而用 `React` 实现上述功能则如下：

```
<Map zoom={4} center={(lat, lng)}>
  <Marker position={(lat, lng)} title={"Hello Marker"} />
</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

组件可以是一个函数或者是一个类，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：

```
const Header = () => {
  return (
    <Jumbotron style={{ backgroundColor: "orange" }}>
      <h1>TODO App</h1>
    </Jumbotron>
  );
};
```

类组件（有状态组件）如下：

```
class Dashboard extends React.Component {
  constructor(props) {
    super(props);
    this.state = {};
  }
  render() {
    return (
      <div className="dashboard">
        <ToDoForm />
        <ToDolist />
      </div>
    );
  }
}
```

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?

2.1. state

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`，一般在 `constructor` 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：

```
class Button extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
  }
  updateCount() {
    this.setState((prevState, props) => {
      return { count: prevState.count + 1 };
    });
  }
  render() {
    return (
      <button onClick={() => this.updateCount()}>
```

```
        Clicked {this.state.count} times
      </button>
    );
  }
}
```

`setState` 还可以接受第二个参数，它是一个函数，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

```
this.setState(
  {
    name: "JS每日一题",
  },
  () => console.log("setState finished")
);
```

2.2. props

`React` 的核心思想就是组件化思想，页面会被切分成一些独立的、可复用的组件

组件从概念上看就是一个函数，可以接受一个参数作为输入值，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

`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：

```
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：

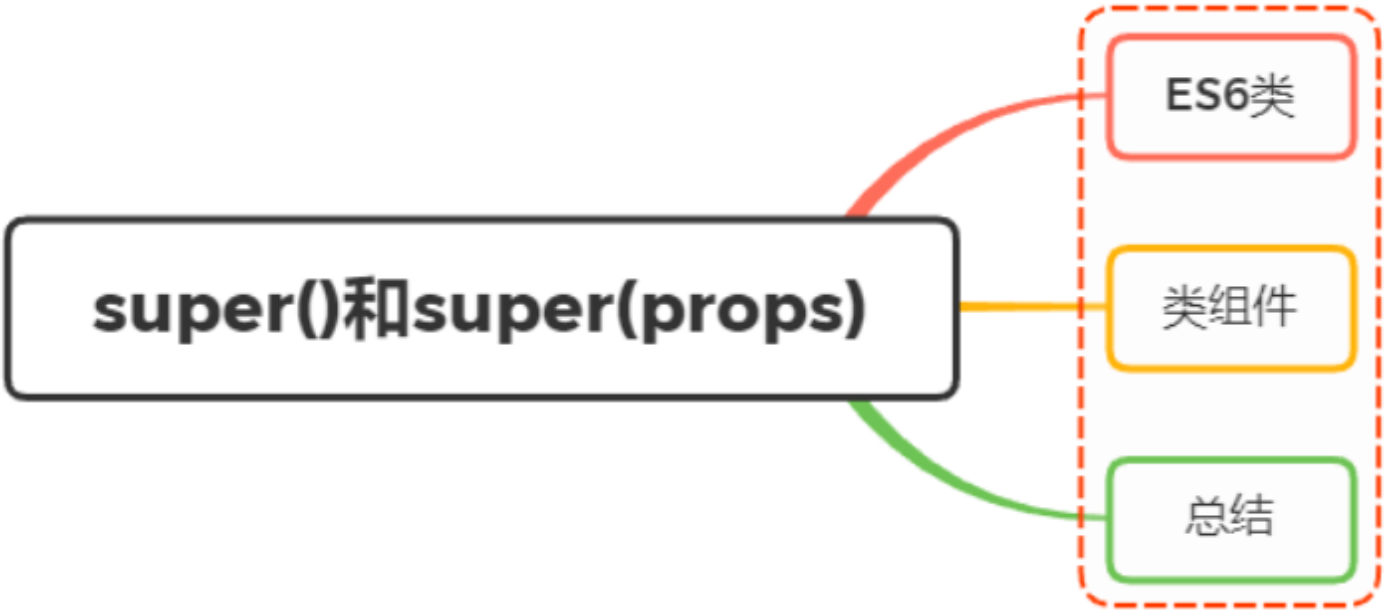
- 两者都是 JavaScript 对象

- 两者都是用于保存信息
- props 和 state 都能触发渲染更新

区别：

- 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?



3.1. ES6 类

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

```
class sup {
  constructor(name) {
    this.name = name;
  }
  printName() {
    console.log(this.name);
  }
}
class sub extends sup {
  constructor(name, age) {
    super(name); // super代表的事父类的构造函数
    this.age = age;
  }
```

```
  printAge() {
    console.log(this.age);
  }
}
let jack = new sub("jack", 20);
jack.printName(); //输出 : jack
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)`

如果在子类中不使用 `super` ，关键字，则会引发报错，如下：

```
ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor
```

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

而 `super()` 就是将父类中的 `this` 对象继承给子类的，没有 `super()` 子类就得不到 `this` 对象

如果先调用 `this` ，再初始化 `super()` ，同样是禁止的行为

```
class sub extends sup {
  constructor(name, age) {
    this.age = age;
    super(name); // super代表的事父类的构造函数
  }
}
```

所以在子类 `constructor` 中，必须先代用 `super` 才能引用 `this`

3.2. 类组件

在 `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 `constructor` 就必须写 `super()` 才初始化 `this`

这时候，在调用 `super()` 的时候，我们一般都需要传入 `props` 作为参数，如果不传进去，`React` 内部也会将其定义在组件实例中

```
// React 内部
const instance = new YourComponent(props);
instance.props = props;
```

所以无论有没有 `constructor`，在 `render` 中 `this.props` 都是可以使用的，这是 `React` 自动附带的，是可以不写的：

```
class HelloMessage extends React.Component {
  render() {
    return <div>nice to meet you! {this.props.name}</div>;
  }
}
```

但是也不建议使用 `super()` 代替 `super(props)`

因为在 `React` 会在类组件构造函数生成实例后再给 `this.props` 赋值，所以在不传递 `props` 在 `super` 的情况下，调用 `this.props` 为 `undefined`，如下情况：

```
class Button extends React.Component {
  constructor(props) {
    super(); // 没传入 props
    console.log(props);      //  {}
    console.log(this.props); //  undefined
    // ...
  }
}
```

而传入 `props` 的则都能正常访问，确保了 `this.props` 在构造函数执行完毕之前已被赋值，更符合逻辑，如下：

```
class Button extends React.Component {
  constructor(props) {
    super(props); // 没传入 props
    console.log(props);      //  {}
    console.log(this.props); //  {}
    // ...
  }
}
```

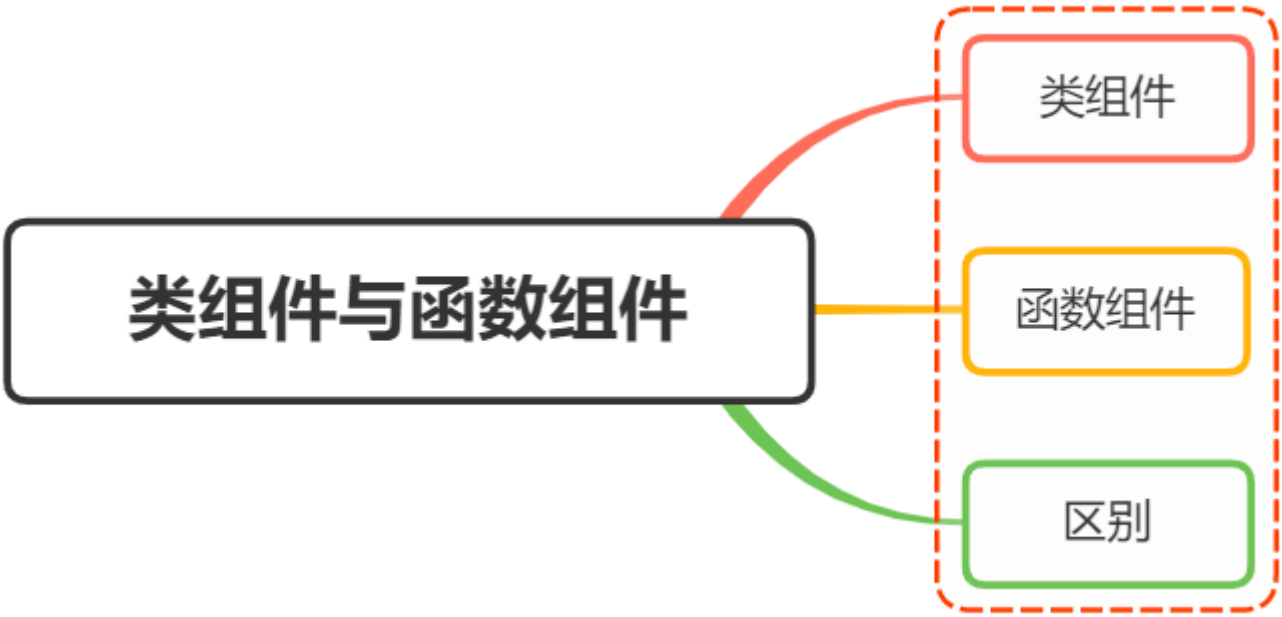
3.3. 总结

在 `React` 中，类组件基于 `ES6`，所以在 `constructor` 中必须使用 `super`

在调用 `super` 过程，无论是否传入 `props`，`React` 内部都会将 `porps` 赋值给组件实例 `porps` 属性中

如果只调用了 `super()`，那么 `this.props` 在 `super()` 和构造函数结束之间仍是 `undefined`

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.Component`

如果想要访问父组件传递过来的参数，可通过 `this.props` 的方式去访问

在组件中必须实现 `render` 方法，在 `return` 中返回 `React` 对象，如下：

```
class Welcome extends React.Component {
  constructor(props) {
    super(props)
  }
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

4.2. 函数组件

函数组件，顾名思义，就是通过函数编写的形式去实现一个 `React` 组件，是 `React` 中定义组件最简单的方式

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

函数第一个参数为 `props` 用于接收父组件传递过来的参数

4.3. 区别

针对两种 `React` 组件，其区别主要分成以下几大方向：

- 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：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

类组件：

```
class Welcome extends React.Component {
  constructor(props) {
    super(props)
  }
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

4.3.2. 状态管理

在 `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 `setState`

如果想要管理 `state` 状态，可以使用 `useState` ，如下：

```
const FunctionalComponent = () => {
    const [count, setCount] = React.useState(0);
    return (
        <div>
            <p>count: {count}</p >
            <button onClick={() => setCount(count + 1)}>Click</button>
        </div>
    );
};
```

在使用 `hooks` 情况下，一般如果函数组件调用 `state` ，则需要创建一个类组件或者 `state` 提升到你的父组件中，然后通过 `props` 对象传递到子组件

4.3.3. 生命周期

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 `React.Component`

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，这里给出一个简单的例子：

```
const FunctionalComponent = () => {
    useEffect(() => {
        console.log("Hello");
    }, []);
    return <h1>Hello, World</h1>;
};
```

上述简单的例子对应类组件中的 `componentDidMount` 生命周期

如果在 `useEffect` 回调函数中 `return` 一个函数，则 `return` 函数会在组件卸载的时候执行，正如 `componentWillUnmount`

```
const FunctionalComponent = () => {
  React.useEffect(() => {
    return () => {
      console.log("Bye");
```

```
    };
  }, []);
  return <h1>Bye, World</h1>;
};
```

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：

```
// 你的代码
function SayHi() {
    return <p>Hello, React</p >
}
// React内部
const result = SayHi(props) // » <p>Hello, React</p >
```

如果是一个类组件，则需要将组件进行实例化，然后调用实例对象的 `render` 方法：

```
// 你的代码
class SayHi extends React.Component {
    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：

```
function ProfilePage(props) {
  const showMessage = () => {
    alert('Followed ' + props.user);
  }
  const handleClick = () => {
    setTimeout(showMessage, 3000);
  }
  return (
```

```
    <button onClick={handleClick}>Follow</button>
  )
}
```

类组件对应如下：

```
class ProfilePage extends React.Component {
  showMessage() {
    alert('Followed ' + this.props.user);
  }
  handleClick() {
    setTimeout(this.showMessage.bind(this), 3000);
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Follow</button>
  }
}
```

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

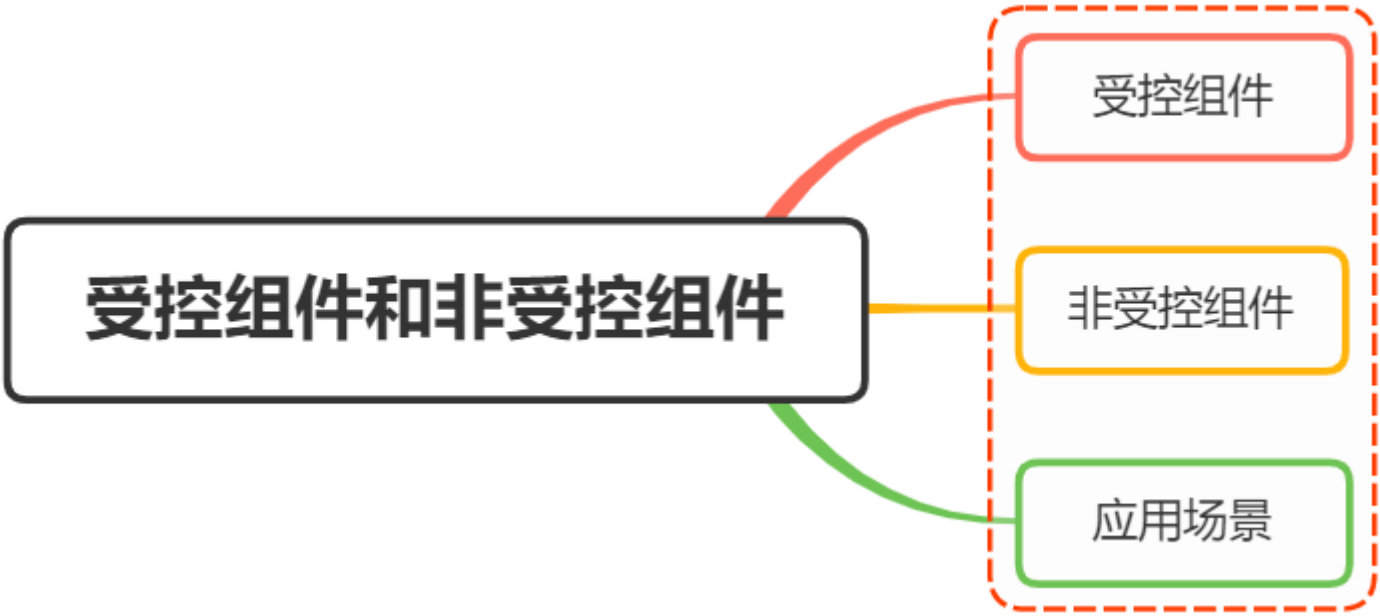
4.3.6. 小结

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？



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：

```
class TestComponent extends React.Component {
  constructor (props) {
    super(props);
    this.state = { username: 'lindaidai' };
  }
  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` 标签是一个可读的状态

这是因为 `value` 被 `this.state.username` 所控制住。当用户输入新的内容时，`this.state.username` 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

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：

```
import React, { Component } from 'react';
export class UnControll extends Component {
  constructor (props) {
    super(props);
    this.inputRef = React.createRef();
  }
  handleSubmit = (e) => {
    console.log('我们可以获得input内的值为', this.inputRef.current.value);
    e.preventDefault();
  }
  render () {
    return (
      <form onSubmit={e => this.handleSubmit(e)}>
        <input defaultValue="lindaidai" ref={this.inputRef} />
        <input type="submit" value="提交" />
      </form>
    )
  }
}
```

5.3. 应用场景

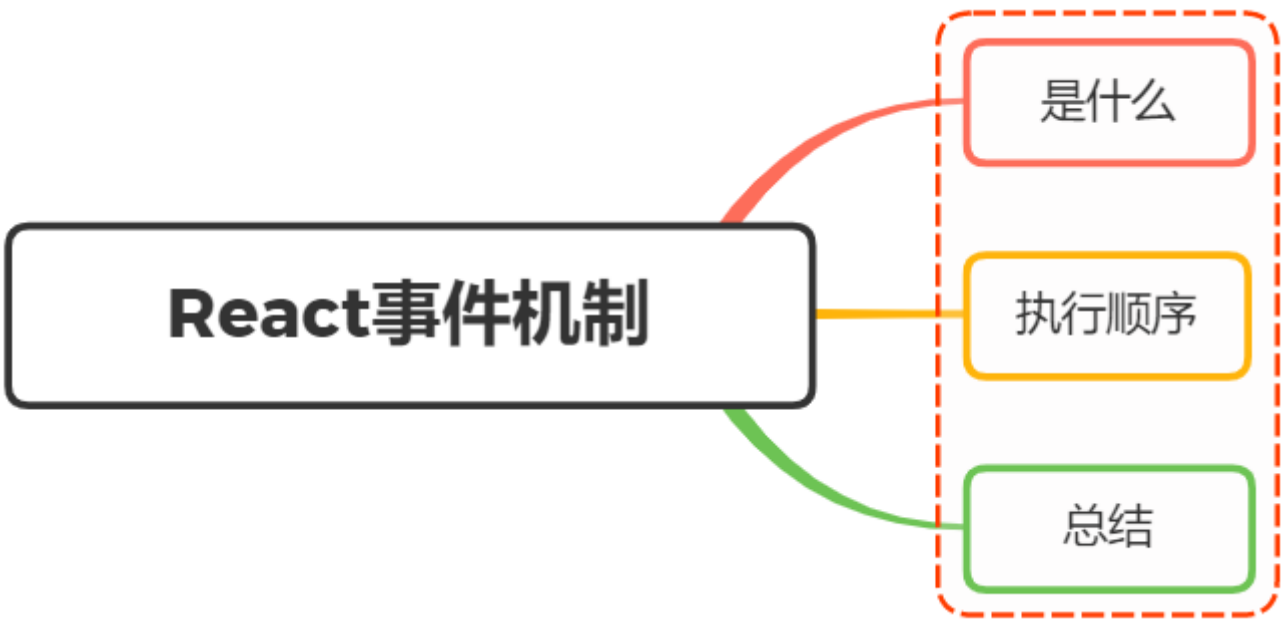
大部分时候推荐使用受控组件来实现表单，因为在受控组件中，表单数据由 `React` 组件负责处理

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：

特征	不受控制	受控
一次性取值（例如，提交时）	✅	✅
提交时验证	✅	✅
即时现场验证	❌	✅
有条件地禁用提交按钮	❌	✅
强制输入格式	❌	✅
一个数据的多个输入	❌	✅
动态输入	❌	✅

6. 说说React的事件机制?



6.1. 是什么

`React` 基于浏览器的事件机制自身实现了一套事件机制，包括事件注册、事件的合成、事件冒泡、事件派发等

在 `React` 中这套事件机制被称之为合成事件

6.1.1. 合成事件（SyntheticEvent）

合成事件是 `React` 模拟原生 `DOM` 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器

根据 `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：

```
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

```
const handleClick = (e) => console.log(e.nativeEvent);;
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

```
// 原生事件绑定方式
<button onclick="handleClick()">按钮命名</button>

// React 合成事件绑定方式
const button = <button onClick={handleClick}>按钮命名</button>
```

- 事件处理函数书写不同

```
// 原生事件 事件处理函数写法
<button onclick="handleClick()">按钮命名</button>

// React 合成事件 事件处理函数写法
const button = <button onClick={handleClick}>按钮命名</button>
```

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，可以看看下面一个例子：

```
import  React  from 'react';
class App extends React.Component{
  constructor(props) {
    super(props);
    this.parentRef = React.createRef();
    this.childRef = React.createRef();
  }
  componentDidMount() {
    console.log("React componentDidMount! ");
    this.parentRef.current?.addEventListener("click", () => {
      console.log("原生事件：父元素 DOM 事件监听! ");
    });
    this.childRef.current?.addEventListener("click", () => {
      console.log("原生事件：子元素 DOM 事件监听! ");
    });
    document.addEventListener("click", (e) => {
      console.log("原生事件：document DOM 事件监听! ");
    });
  }
  parentClickFun = () => {
    console.log("React 事件：父元素事件监听! ");
  };
  childClickFun = () => {
    console.log("React 事件：子元素事件监听! ");
  };
  render() {
    return (
      <div ref={this.parentRef} onClick={this.parentClickFun}>
        <div ref={this.childRef} onClick={this.childClickFun}>
          分析事件执行顺序
        </div>
      </div>
    );
  }
}
export default App;
```

输出顺序为：

```
原生事件：子元素 DOM 事件监听！
原生事件：父元素 DOM 事件监听！
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：

- 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.stopImmediatePropagation()
- 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

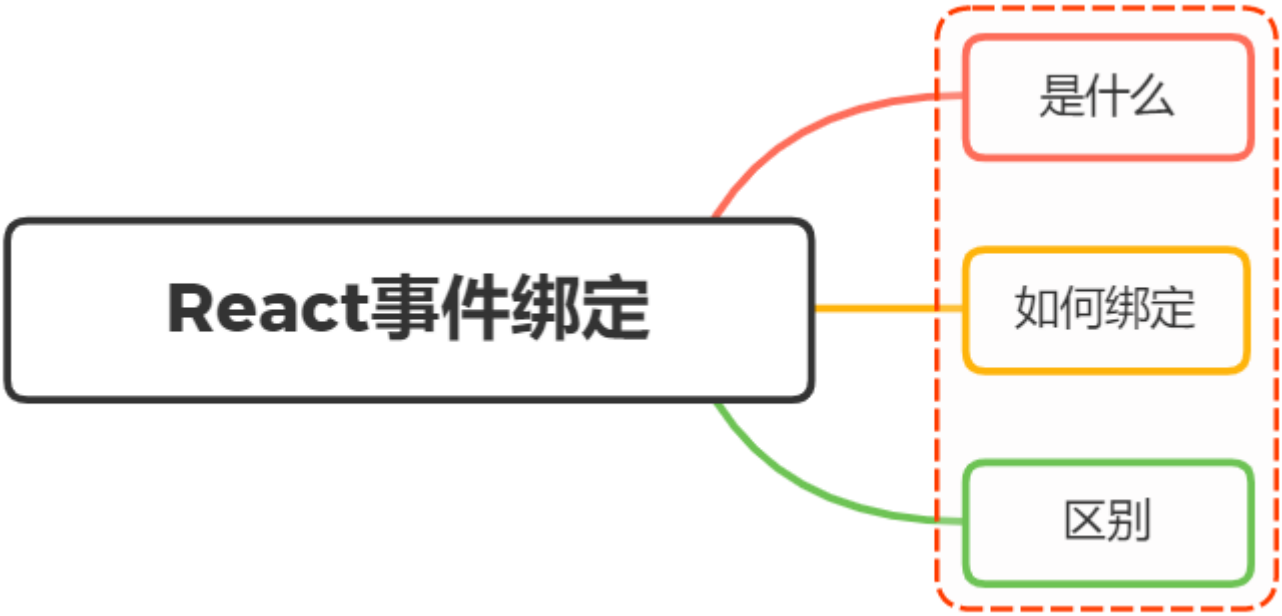
```
document.body.addEventListener('click', e => {

    if (e.target && e.target.matches('div.code')) {

        return;

    }

    this.setState({   active: false,    });   });
}
```

6.3. 总结

`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

在 `react` 应用中，事件名都是用小驼峰格式进行书写，例如 `onclick` 要改写成 `onClick`

最简单的事件绑定如下：

```
class ShowAlert extends React.Component {
  showAlert() {
    console.log("Hi");
  }
  render() {
    return <button onClick={this.showAlert}>show</button>;
  }
}
```

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
- 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

如果使用一个类组件，在其中给某个组件/元素一个 `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

```
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，同样再每一次 `render` 的时候都会生成新的方法，影响性能

```
class App extends React.Component {
  handleClick() {
    console.log('this > ', this);
  }
  render() {
    return (
```

```
      <div onClick={e => this.handleClick(e)}>test</div>
    )
  }
}
```

7.2.3. constructor中bind

在 `constructor` 中预先 `bind` 当前组件，可以避免在 `render` 操作中重复绑定

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  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

跟上述方式三一样，能够避免在 `render` 操作中重复绑定，实现也非常的简单，如下：

```
class App extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log('this > ', this);
  }
  render() {
    return (
      <div onClick={this.handleClick}>test</div>
    )
  }
}
```

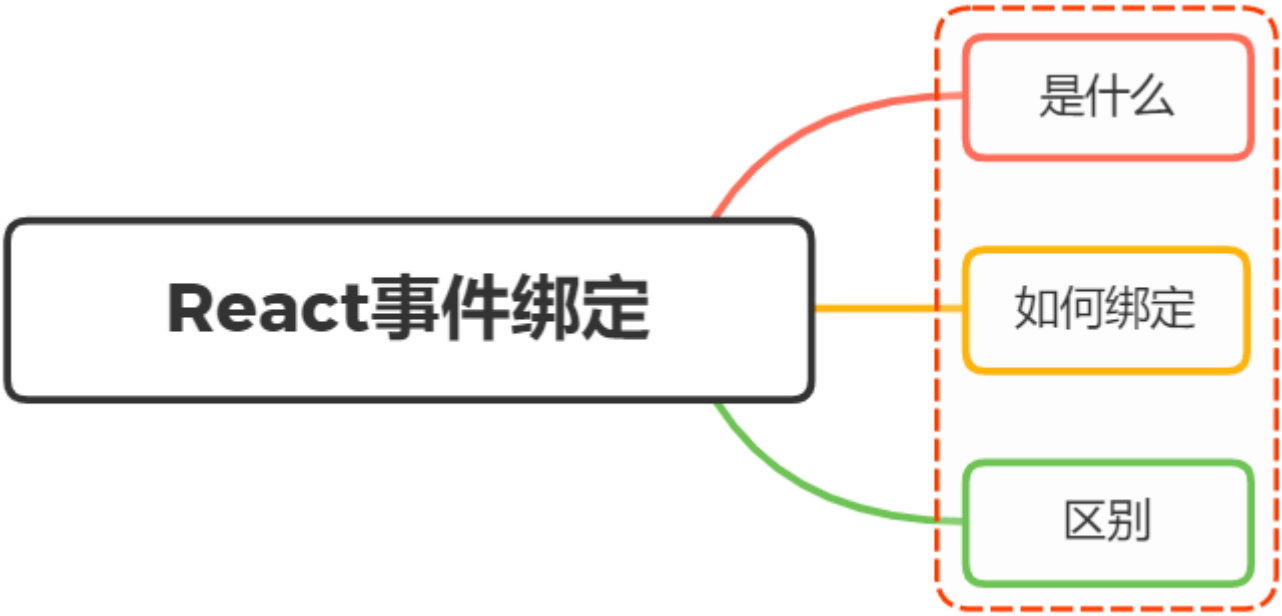
7.3. 区别

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？



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

在 `React` 中，一个类、一个函数都可以视为一个组件

- 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
- 提高可维护性，由于每个组件的职责单一，并且组件在系统中是被复用的，所以对代码进行优化可获得系统的整体升级

8.2. 如何构建

在 `React` 目前来讲，组件的创建主要分成了三种方式：

- 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

在 `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 `props` 来展示视图，不涉及对 `state` 状态的操作

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 `React` 中，通过函数简单创建组件的示例如下：

```
function HelloComponent(props, /* context */) {
  return <div>Hello {props.name}</div>
}
```

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

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：

```
function HelloComponent(props) /* context */{
  return React.createElement(
    "div",
    null,
    "Hello ",
    props.name
  );
}
```

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

同样在 `react hooks` 出来之前，有状态的组件只能通过继承 `React.Component` 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：

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }
  tick() {
    this.setState(state => ({
      seconds: state.seconds + 1
    }));
  }
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }
  componentWillUnmount() {
    clearInterval(this.interval);
  }
  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}
```

8.3. 区别

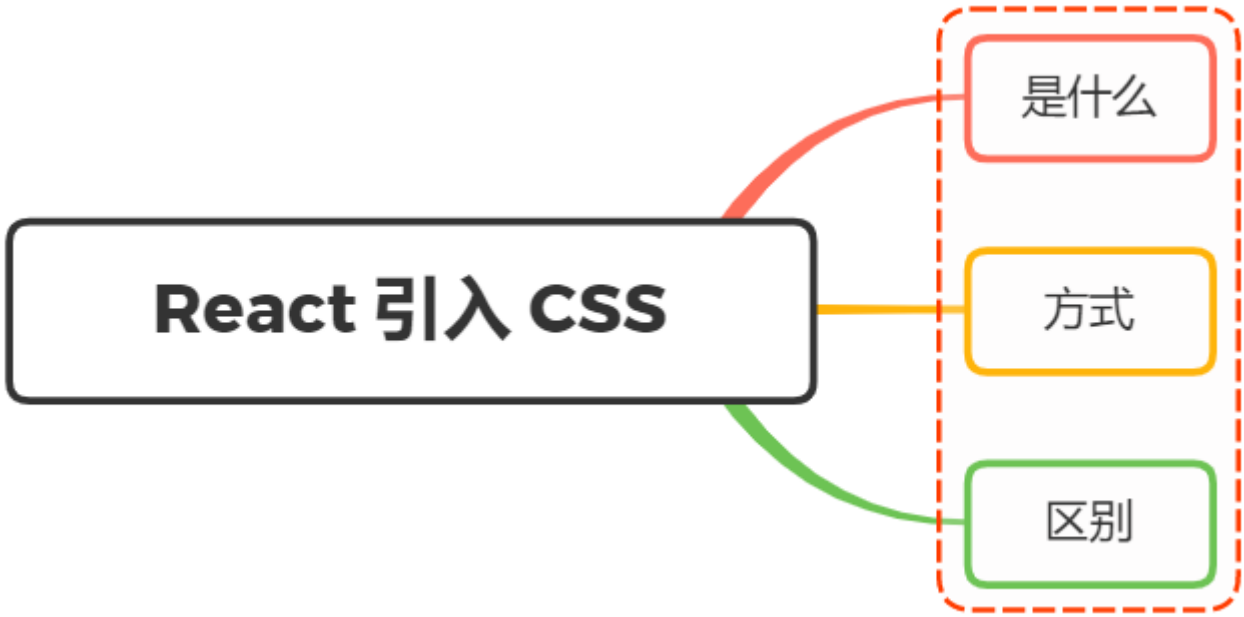
由于 `React.createClass` 创建的方式过于冗杂，并不建议使用

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

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

在这一方面， `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
- 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
- 组件中引入 .module.css 文件
- 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：

```
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);
  }
  render() {
    return (
     <div>
       <div style={div1}>123</div>
       <div style={{backgroundColor:"red"}}>
     </div>
    );
  }
}
export default Test;
```

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
- 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

将 `css` 单独写在一个 `css` 文件中，然后在组件中直接引入

`App.css` 文件：

```
.title {
  color: red;
  font-size: 20px;
}
.desc {
  color: green;
  text-decoration: underline;
}
```

组件中引入：

```
import React, { PureComponent } from 'react';
import Home from './Home';
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中的一段文字描述</p >
        <Home/>
      </div>
    )
  }
}
```

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

9.2.3. 组件中引入 .module.css 文件

将 `css` 文件作为一个模块引入，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

这种方式是 `webpack` 特工的方案，只需要配置 `webpack` 配置文件中 `modules:true` 即可

```
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>
        <p className="desc">我是App中的一段文字描述</p >
        <Home/>
      </div>
    )
  }
}
```

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：

- styled-components
- emotion
- glamorous

下面主要看看 `styled-components` 的基本使用

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：

```
export const SelfLink = styled.div
  height: 50px;   border: 1px solid red;   color: yellow;
;
export const SelfButton = styled.div  height: 150px;   width: 150px;   color:
${props => props.color};   background-image: url(${props => props.src});
background-size: 150px 150px;;
```

引入样式组件也很简单：

```
import React, { Component } from "react";
import { SelfLink, SelfButton } from "./style";
class Test extends Component {
  constructor(props, context) {
    super(props);
  }
  render() {
    return (
     <div>
       <SelfLink title="People's Republic of China">app.js</SelfLink>
       <SelfButton color="palevioletred" style={{ color: "pink" }} src={fist}>
          SelfButton
        </SelfButton>
     </div>
    );
  }
}
export default Test;
```

9.3. 区别

通过上面四种样式的引入，可以看到：

- 在组件内直接使用 `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一样样式嵌套、定义、修改状态等

至于使用 `react` 用哪种方案引入 `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

10. 说说 React 生命周期有哪些不同阶段？每个阶段对应的方法是？

10.1. 是什么

生命周期 `（Life Cycle）` 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 `Vue` 一样， `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→渲染、更新→渲染、卸载等一系列过程

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：

- constructor
- getDerivedStateFromProps
- render
- componentDidMount

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` ，第二个参数为上一个状态的 `state` ，可以比较 `props` 和 `state` 来加一些限制条件，防止无用的state更新

该方法需要返回一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
- shouldComponentUpdate
- render
- getSnapshotBeforeUpdate
- componentDidUpdate

10.2.6. getDerivedStateFromProps

该方法介绍同上

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false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

该方法返回的一个 `Snapshot` 值，作为 `componentDidUpdate` 第三个参数传入

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('#enter getSnapshotBeforeUpdate');
    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

在该方法中，可以根据前后的 `props` 和 `state` 的变化做相应的操作，如获取数据，修改 `DOM` 样式等

10.3.4. 卸载阶段

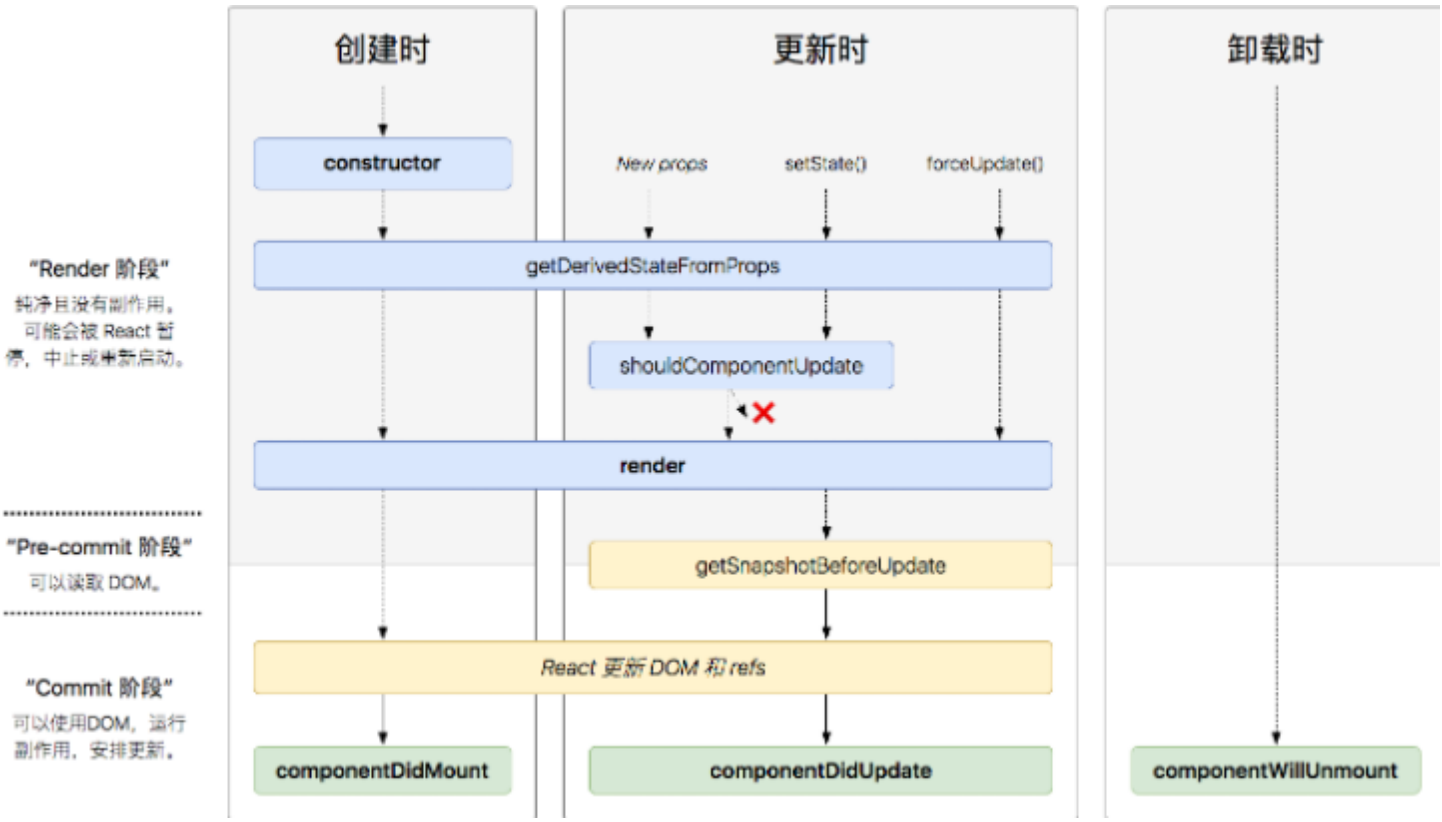
10.4. componentWillUnmount

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

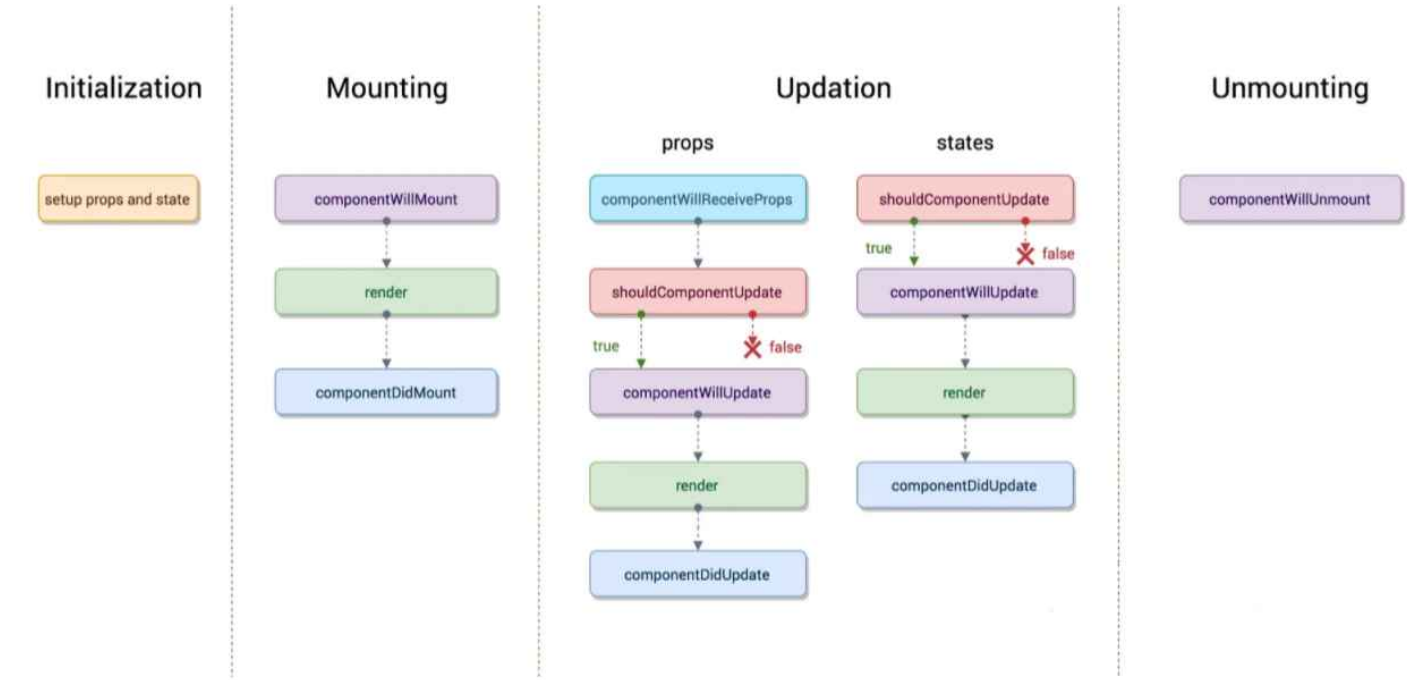
一旦一个组件实例被卸载，其不会被再次挂载，而只可能是被重新创建

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：



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：

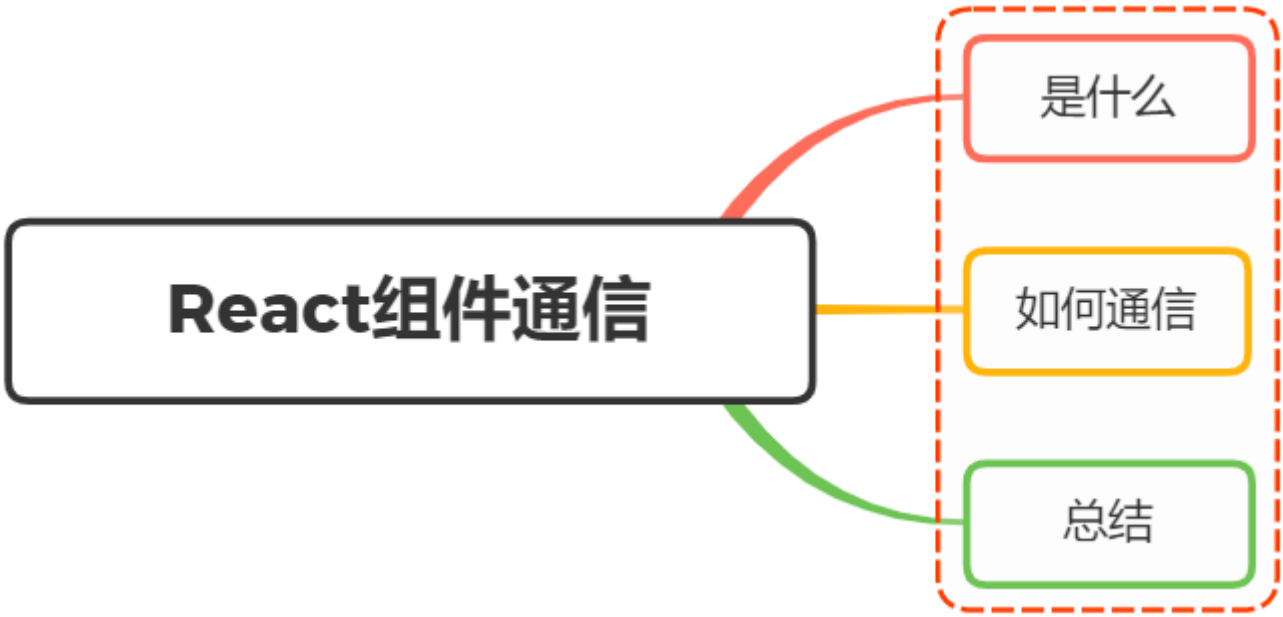
- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

其实这三个方法仍然存在，只是在前者加上了 `UNSAFE_` 前缀，如 `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码可能在未来的 `react` 版本可能废除

同时也新增了两个生命周期函数：

- getDerivedStateFromProps
- getSnapshotBeforeUpdate

11. React中组件之间如何通信?



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` 中最强大的功能之一，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

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

```
function EmailInput(props) {
  return (
    <label>
      Email: <input value={props.email} />
    </label>
  );
}
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：

```
class Parents extends Component {
  constructor() {
    super();
    this.state = {
      price: 0
    };
  }
  getItemPrice(e) {
    this.setState({
      price: e
    });
  }
  render() {
    return (
      <div>
        <div>price: {this.state.price}</div>
        {/* 向子组件中传入一个函数  */}
        <Child getPrice={this.getItemPrice.bind(this)} />
      </div>
    );
```

```
  }
}
```

子组件对应代码如下：

```
class Child extends Component {
  clickGoods(e) {
    // 在此函数中传入值
    this.props.getPrice(e);
  }
  render() {
    return (
      <div>
        <button onClick={this.clickGoods.bind(this, 100)}>goods1</button>
        <button onClick={this.clickGoods.bind(this, 1000)}>goods2</button>
      </div>
    );
  }
}
```

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

```
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}
        />
        <SiblingB
          onClick={this.setCount}
        />
      </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`

```
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：

```
<PriceContext.Provider value={100}>
</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

```
class MyClass extends React.Component {
  static contextType = PriceContext;
  render() {
    let price = this.context;
    /* 基于这个值进行渲染工作 */
  }
}
```

`Consumer` 组件：

```
<PriceContext.Consumer>
    { /
*这里是一个函数*
/ }
    {
        price => <div>price: {price}</div>
```

```
    }
</PriceContext.Consumer>
```

11.2.5. 非关系组件传递

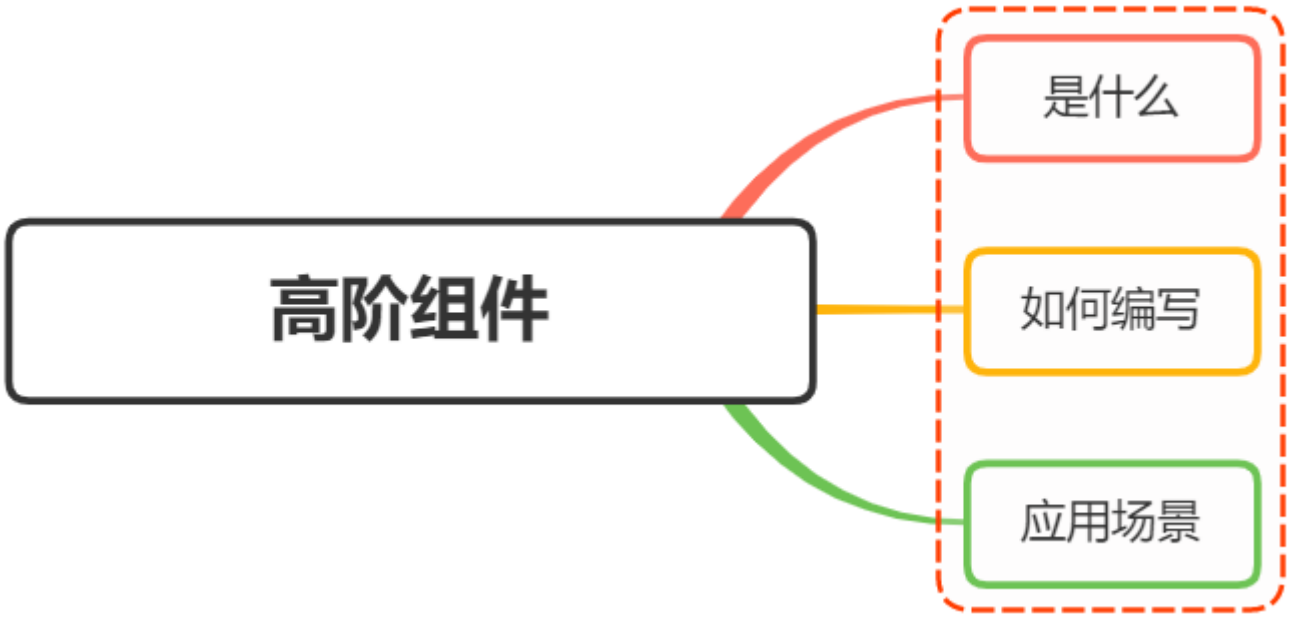
如果组件之间关系类型比较复杂的情况，建议将数据进行一个全局资源管理，从而实现通信，例如 `redux` 。关于 `redux` 的使用后续再详细介绍

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

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，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，并不是一个组件

```
const EnhancedComponent = highOrderComponent(WrappedComponent);
```

上述代码中，该函数接受一个组件 `WrappedComponent` 作为参数，返回加工过的新组件 `EnhancedComponent`

高阶组件的这种实现方式，本质上是一个装饰者设计模式

12.2. 如何编写

最基本的高阶组件的编写模板如下：

```
import React, { Component } from 'react';
export default (WrappedComponent) => {
  return class EnhancedComponent extends Component {
    // do something
    render() {
      return <WrappedComponent />;
    }
  }
}
```

通过对传入的原始组件 `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 `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`

如果向一个高阶组件添加 `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：

```
function withLogging(WrappedComponent) {
```

```
    class Enhance extends WrappedComponent {
        componentWillReceiveProps() {
            console.log('Current props', this.props);
            console.log('Next props', nextProps);
        }
        render() {
            const {forwardedRef, ...rest} = this.props;
            // 把 forwardedRef 赋值给 ref
            return <WrappedComponent {...rest} ref={forwardedRef} />;
        }
    };
    // React.forwardRef 方法会传入 props 和 ref 两个参数给其回调函数
    // 所以这边的 ref 是由 React.forwardRef 提供的
    function forwardRef(props, ref) {
        return <Enhance {...props} forwardRef={ref} />
    }
    return React.forwardRef(forwardRef);
}
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：

```
import React, { Component } from 'react'
class MyComponent extends Component {
  componentWillMount() {
      let data = localStorage.getItem('data');
      this.setState({data});
  }

  render() {
    return <div>{this.state.data}</div>
  }
}
```

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 `componentWillMount` 中的代码，这明显是冗余的

下面就可以通过高价组件来进行改写，如下：

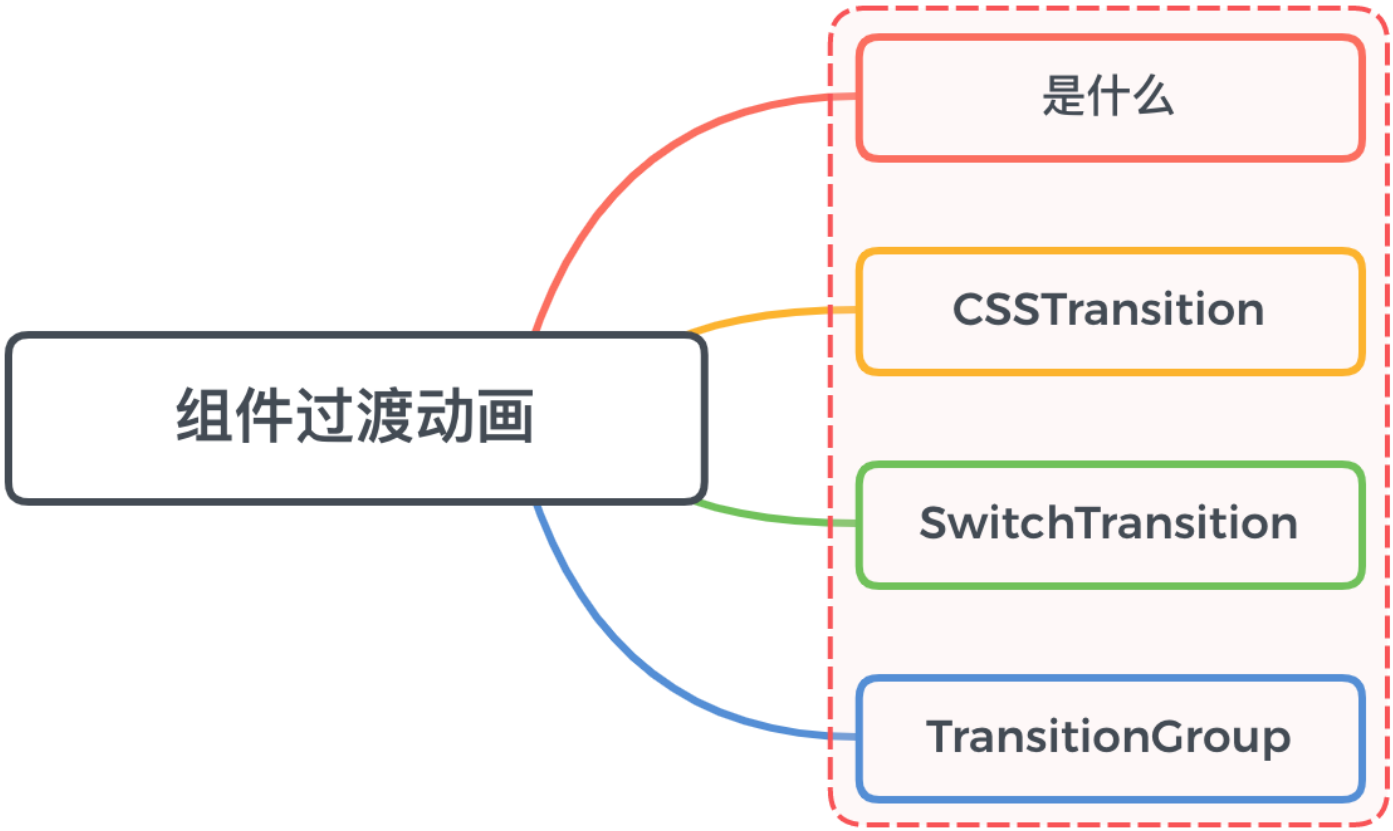
```
import React, { Component } from 'react'
function withPersistentData(WrappedComponent) {
  return class extends Component {
    componentWillMount() {
      let data = localStorage.getItem('data');
        this.setState({data});
    }

    render() {
      // 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 WrappedComponent
      return <WrappedComponent data={this.state.data} {...this.props} />
    }
  }
}
class MyComponent2 extends Component {

  render() {
    return <div>{this.props.data}</div>
  }
}
const MyComponentWithPersistentData = withPersistentData(MyComponent2)
```

再比如组件渲染性能监控，如下：

```
class Home extends React.Component {
    render() {
        return (<h1>Hello World.</h1>);
    }
}
function withTiming(WrappedComponent) {
    return class extends WrappedComponent {
        constructor(props) {
            super(props);
            this.start = 0;
            this.end = 0;
        }
        componentWillMount() {
            super.componentWillMount && super.componentWillMount();
            this.start = Date.now();
        }
        componentDidMount() {
            super.componentDidMount && super.componentDidMount();
            this.end = Date.now();
            console.log(
```

```
21 ${WrappedComponent.name} 组件渲染时间为 ${this.end - this.start} ms
22 );
23         }
24         render() {
25             return super.render();
26         }
27     };
28 }
29 export default withTiming(Home);
```

13. 在react中组件间过渡动画如何实现?



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

在 `react` 中实现过渡动画效果会有很多种选择，如 `react-transition-group` ， `react-motion` ， `Animated` ，以及原生的 `CSS` 都能完成切换动画

13.2. 如何实现

在 `react` 中， `react-transition-group` 是一种很好的解决方案，其为元素添加 `enter` ， `enter-active` ， `exit` ， `exit-active` 这一系列勾子

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

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

所以可以利用这一点，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

当 `in` 属性置为 `false` 时， `CSSTransition` 会给子组件加上 `xxx-exit` 和 `xxx-exit-active` 的 `class` ，然后开始执行动画，当动画结束后，移除两个 `class` ，然后添加 `-enter-done` 的 `class`

如下例子：

```
export default class App2 extends React.PureComponent {
  state = {show: true};
  onToggle = () => this.setState({show: !this.state.show});
  render() {
    const {show} = this.state;
    return (
      <div className={'container'}>
        <div className={'square-wrapper'}>
          <CSSTransition
            in={show}
            timeout={500}
            classNames={'fade'}
            unmountOnExit={true}
          >
            <div className={'square'} />
          </CSSTransition>
        </div>
        <Button onClick={this.onToggle}>toggle</Button>
      </div>
    );
  }
}
```

对应 `css` 样式如下：

```
.fade-enter {
  opacity: 0;
  transform: translateX(100%);
}
.fade-enter-active {
  opacity: 1;
  transform: translateX(0);
  transition: all 500ms;
}
.fade-exit {
  opacity: 1;
  transform: translateX(0);
}
.fade-exit-active {
  opacity: 0;
  transform: translateX(-100%);
  transition: all 500ms;
}
```

13.2.2. SwitchTransition

`SwitchTransition` 可以完成两个组件之间切换的炫酷动画

比如有一个按钮需要在 `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` 中主要有一个属性 `mode` ，对应两个值：

- in-out：表示新组件先进入，旧组件再移除；
- 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 `key` 属性

下面给出一个按钮入场和出场的示例，如下：

```
import { SwitchTransition, CSSTransition } from "react-transition-group";
export default class SwitchAnimation extends PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      isOn: true
    }
```

```
  }
  render() {
    const {isOn} = this.state;
    return (
      <SwitchTransition mode="out-in">
        <CSSTransition classNames="btn"
                       timeout={500}
                       key={isOn ? "on" : "off"}>
          {
          <button onClick={this.btnClick.bind(this)}>
            {isOn ? "on": "off"}
          </button>
        }
        </CSSTransition>
      </SwitchTransition>
    )
  }
  btnClick() {
    this.setState({isOn: !this.state.isOn})
  }
}
```

`css` 文件对应如下：

```
.btn-enter {
  transform: translate(100%, 0);
  opacity: 0;
}
.btn-enter-active {
  transform: translate(0, 0);
  opacity: 1;
  transition: all 500ms;
}
.btn-exit {
  transform: translate(0, 0);
  opacity: 1;
}
.btn-exit-active {
  transform: translate(-100%, 0);
  opacity: 0;
  transition: all 500ms;
}
```

13.2.3. TransitionGroup

当有一组动画的时候，就可将这些 `CSSTransition` 放入到一个 `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：

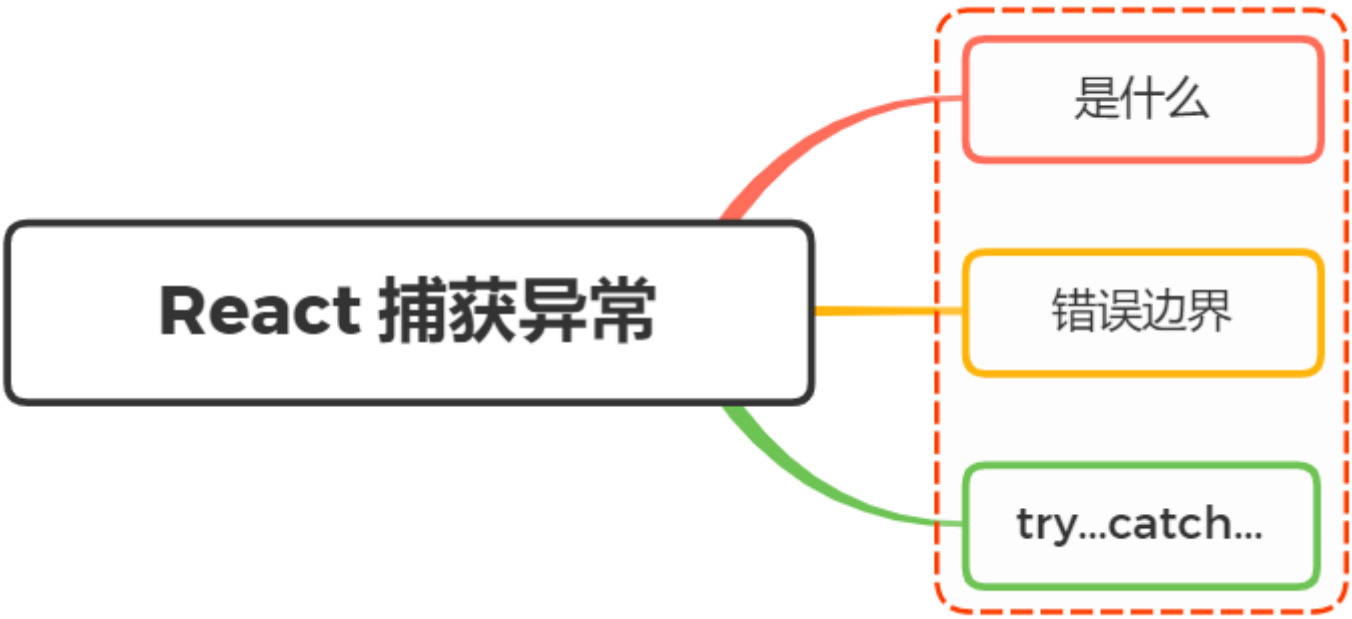
```
import React, { PureComponent } from 'react'
import { CSSTransition, TransitionGroup } from 'react-transition-group';
export default class GroupAnimation extends PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      friends: []
    }
  }
  render() {
    return (
      <div>
        <TransitionGroup>
          {
            this.state.friends.map((item, index) => {
              return (
                <CSSTransition classNames="friend" timeout={300} key={index}>
                  <div>{item}</div>
                </CSSTransition>
              )
            })
          }
        </TransitionGroup>
        <button onClick={e => this.addFriend()}>+friend</button>
      </div>
    )
  }
  addFriend() {
    this.setState({
      friends: [...this.state.friends, "coderwhy"]
    })
  }
```

```
}
```

对应 `css` 如下：

```
.friend-enter {
    transform: translate(100%, 0);
    opacity: 0;
}
.friend-enter-active {
    transform: translate(0, 0);
    opacity: 1;
    transition: all 500ms;
}
.friend-exit {
    transform: translate(0, 0);
    opacity: 1;
}
.friend-exit-active {
    transform: translate(-100%, 0);
    opacity: 0;
    transition: all 500ms;
}
```

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

举个例子，在 `react` 项目中去编写组件内 `JavaScript` 代码错误会导致 `React` 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

作为一个框架， `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 `JavaScript` 错误，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()

抛出错误后，请使用 `static getDerivedStateFromError()` 渲染备用 UI ，使用 `componentDidCatch()` 打印错误信息，如下：

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染能够显示降级后的 UI
    return { hasError: true };
  }
  componentDidCatch(error, errorInfo) {
    // 你同样可以将错误日志上报给服务器
    logErrorToMyService(error, errorInfo);
  }
  render() {
    if (this.state.hasError) {
      // 你可以自定义降级后的 UI 并渲染
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

然后就可以把自身组件的作为错误边界的子组件，如下：

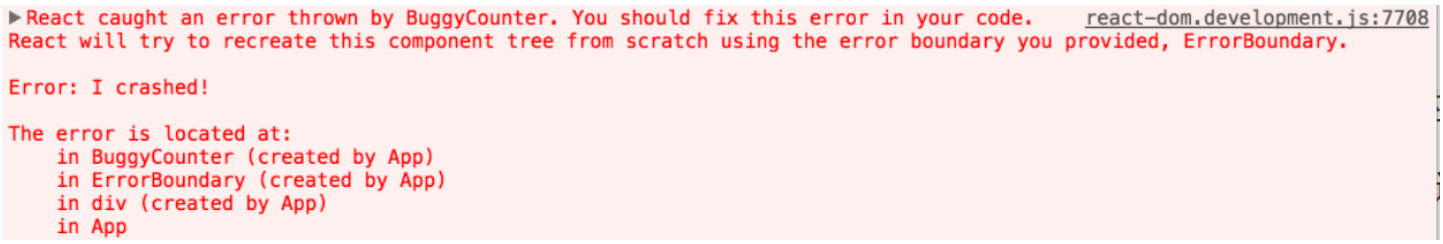
```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

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

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

这种情况可以使用 `js` 的 `try...catch...` 语法，如下：

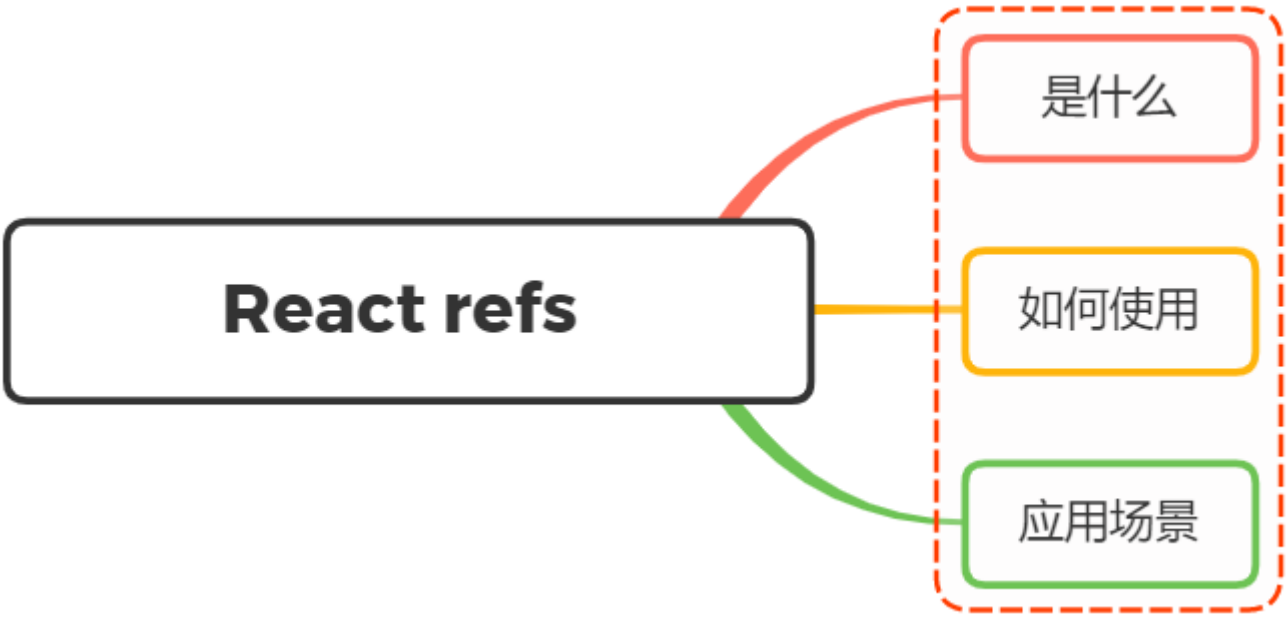
```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    try {
      // 执行操作，如有错误则会抛出
    } catch (error) {
      this.setState({ error });
    }
  }
  render() {
    if (this.state.error) {
```

```
16        return <h1>Caught an error.</h1>
17      }
18      return <button onClick={this.handleClick}>Click Me</button>
19    }
20  }
```

除此之外还可以通过监听 `onerror` 事件

```
1 window.addEventListener('error', function(event) { ... })
```

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）

`React` 中的 `Refs` 提供了一种方式，允许我们访问 `DOM` 节点或在 `render` 方法中创建的 `React` 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，如果渲染 `dom` 则返回的是具体的 `dom` 节点

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

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref="myref" />;
  }
}
```

访问当前节点的方式如下：

```
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：

```
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

```
const node = this.myRef.current;
```

15.2.3. 传入函数

当 `ref` 传入为一个函数的时候，在渲染过程中，回调函数参数会传入一个元素对象，然后通过实例将对象进行保存

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={element => this.myref = element} />;
  }
}
```

获取 `ref` 对象只需要通过先前存储的对象即可

```
const node = this.myref
```

15.2.4. 传入hook

通过 `useRef` 创建一个 `ref` ，整体使用方式与 `React.createRef` 一致

```
function App(props) {
  const myref = useRef()
  return (
    <>
      <div ref={myref}></div>
    </>
  )
}
```

获取 `ref` 属性也是通过 `hook` 对象的 `current` 属性

```
1 const node = myref.current;
```

上述三种情况都是 `ref` 属性用于原生 `HTML` 元素上，如果 `ref` 设置的组件为一个类组件的时候，`ref` 对象接收到的是组件的挂载实例

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，更多情况我们是通过 `props` 与 `state` 的方式进行去重新渲染子元素

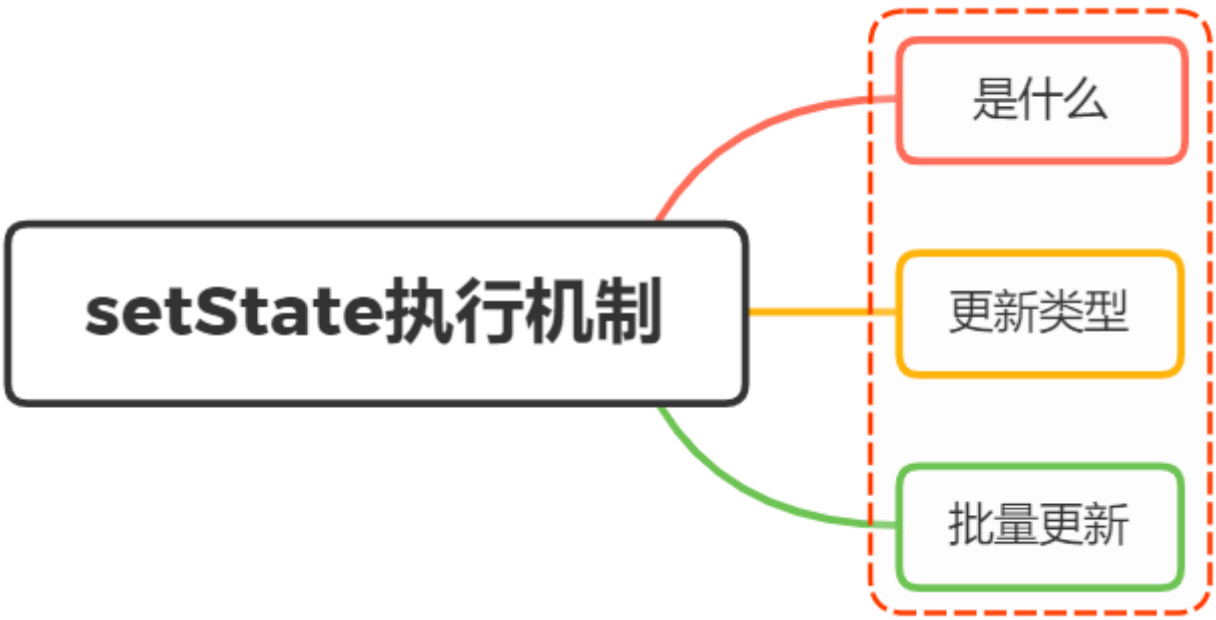
过多使用 `refs` ，会使组件的实例或者是 `DOM` 结构暴露，违反组件封装的原则

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

```
import React, { Component } from 'react'
export default class App extends Component {
    constructor(props) {
        super(props);
        this.state = {
            message: "Hello World"
        }
    }
    render() {
        return (
            <div>
                <h2>{this.state.message}</h2>
                <button onClick={e => this.changeText()}>面试官系列</button>
            </div>
        )
    }
    changeText() {
        this.setState({
            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 `render` 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：

```
changeText() {
    this.state.message = "你好啊,李银河";
}
```

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

这是因为 `React` 并不像 `vue2` 中调用 `Object.defineProperty` 数据响应式或者 `Vue3` 调用 `Proxy` 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

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

```
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');
};
```

从上面可以看到 `setState` 第一个参数可以是一个对象，或者是一个函数，而第二个参数是一个回调函数，用于可以实时的获取到更新之后的数据

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：

```
changeText() {
  this.setState({
    message: "你好啊"
  })
  console.log(this.state.message); // Hello World
}
```

从上面可以看到，最终打印结果为 `Hello world` ，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

```
changeText() {
  this.setState({
    message: "你好啊"
  }, () => {
    console.log(this.state.message); // 你好啊
  });
}
```

16.2.2. 同步更新

同样先给出一个在 `setTimeout` 中更新的例子：

```
changeText() {
  setTimeout(() => {
    this.setState({
      message: "你好啊
    });
    console.log(this.state.message); // 你好啊
  }, 0);
}
```

上面的例子中，可以看到更新是同步

再来举一个原生 `DOM` 事件的例子：

```
componentDidMount() {
  const btnEl = document.getElementById("btn");
  btnEl.addEventListener('click', () => {
    this.setState({
      message: "你好啊,李银河"
    });
    console.log(this.state.message); // 你好啊,李银河
  })
}
```

16.2.3. 小结

- 在组件生命周期或React合成事件中，setState是异步
- 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：

```
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({
        count: this.state.count + 1,
    })
    console.log(this.state.count) // 1
}
```

点击按钮触发事件，打印的都是 1，页面显示 `count` 的值为 2

对同一个值进行多次 `setState`， `setState` 的批量更新策略会对其进行覆盖，取最后一次的执行结果

上述的例子，实际等价于如下：

```
Object.assign(
  previousState,
  {index: state.count+ 1},
  {index: state.count+ 1},
  ...
)
```

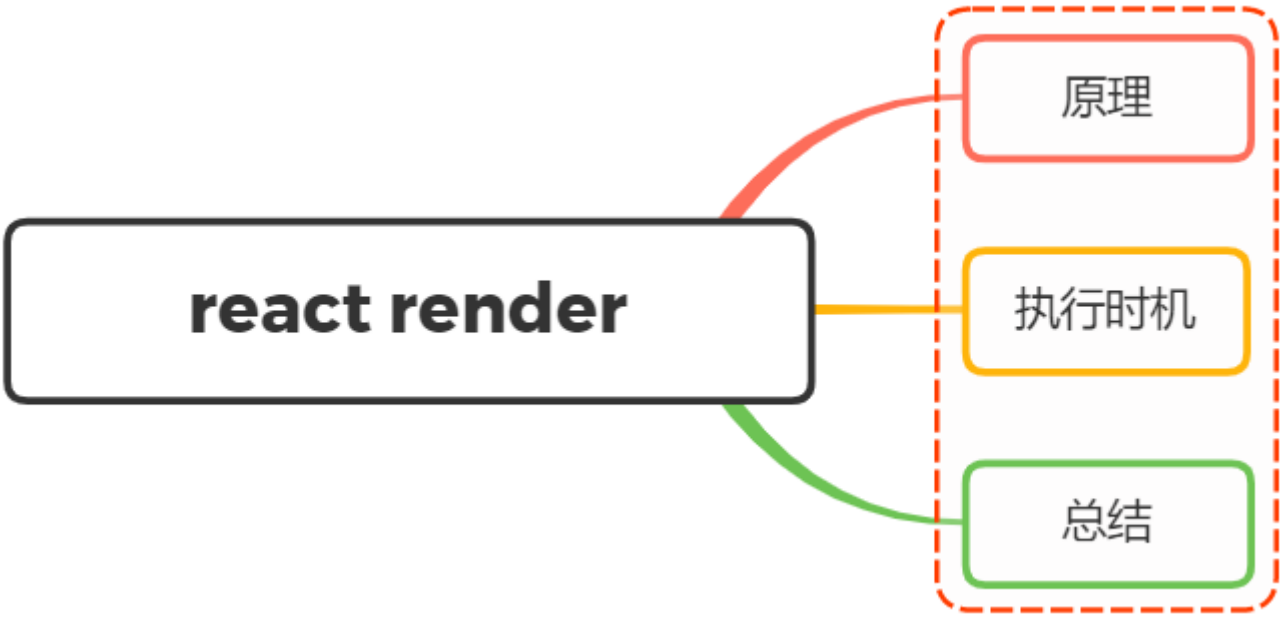
由于后面的数据会覆盖前面的更改，所以最终只加了一次

如果是下一个 `state` 依赖前一个 `state` 的话，推荐给 `setState` 一个参数传入一个 `function`，如下：

```
onClick = () => {
    this.setState((prevState, props) => {
      return {count: prevState.count + 1};
    });
    this.setState((prevState, props) => {
      return {count: prevState.count + 1};
    });
}
```

而在 `setTimeout` 或者原生 `dom` 事件中，由于是同步的操作，所以并不会进行覆盖现象

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：

```
class Foo extends React.Component {
    render() {
        return <h1> Foo </h1>;
    }
}
```

在函数组件中，指的是函数组件本身：

```
function Foo() {
    return <h1> Foo </h1>;
}
```

在 `render` 中，我们会编写 `jsx`， `jsx` 通过 `babel` 编译后就会转化成我们熟悉的 `js` 格式，如下：

```
return (
  <div className='cn'>
    <Header> hello </Header>
    <div> start </div>
    Right Reserve
```

```
    </div>
)
```

`babel` 编译后：

```
return (
  React.createElement(
    'div',
    {
      className : 'cn'
    },
    React.createElement(
      Header,
      null,
      'hello'
    ),
    React.createElement(
      'div',
      null,
      'start'
    ),
    'Right Reserve'
  )
)
```

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
- attributes：标签属性，若无则为null
- 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，这一步是决定如何更新 `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

```
class Foo extends React.Component {
  state = { count: 0 };
  increment = () => {
    const { count } = this.state;
    const newCount = count < 10 ? count + 1 : count;
    this.setState({ count: newCount });
  };
  render() {
    const { count } = this.state;
    console.log("Foo render");
    return (
      <div>
        <h1> {count} </h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

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 `Foo render` ，证明 `render` 执行了

- 函数组件通过 `useState hook` 修改状态

```
function Foo() {
  const [count, setCount] = useState(0);
  function increment() {
    const newCount = count < 10 ? count + 1 : count;
    setCount(newCount);
  }
  console.log("Foo render");

  return (
    <div>
      <h1> {count} </h1>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

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

```
class App extends React.Component {
  state = { name: "App" };
  render() {
    return (
      <div className="App">
        <Foo />
        <button onClick={() => this.setState({ name: "App" })}>
          Change name
        </button>
      </div>
    );
  }
}
function Foo() {
  console.log("Foo render");
  return (
    <div>
      <h1> Foo </h1>
    </div>
  );
}
```

只要点击了 `App` 组件内的 `Change name` 按钮，不管 `Foo` 具体实现是什么，都会被重新 `render` 渲染

- 函数组件重新渲染

```
function App(){
    const [name,setName] = useState('App')
    return (
        <div className="App">
            <Foo />
            <button onClick={() => setName("aaa")}>
                { name }
            </button>
      </div>
    )
}
function Foo() {
  console.log("Foo render");
  return (
    <div>
      <h1> Foo </h1>
    </div>
  );
```

```
19 }
```

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

17.3. 总结

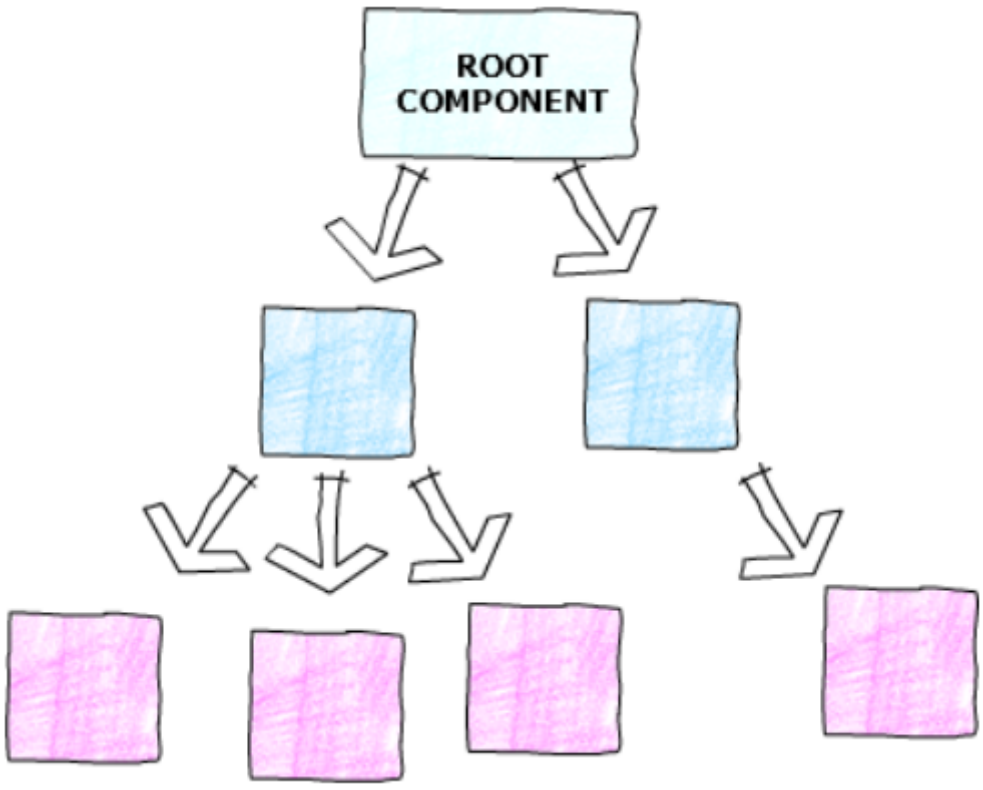
`render` 函数里面可以编写 `JSX` ，转化成 `createElement` 这种形式，用于生成虚拟 `DOM` ，最终转化成真实 `DOM`

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` 更改状态不一定导致重新 `render`

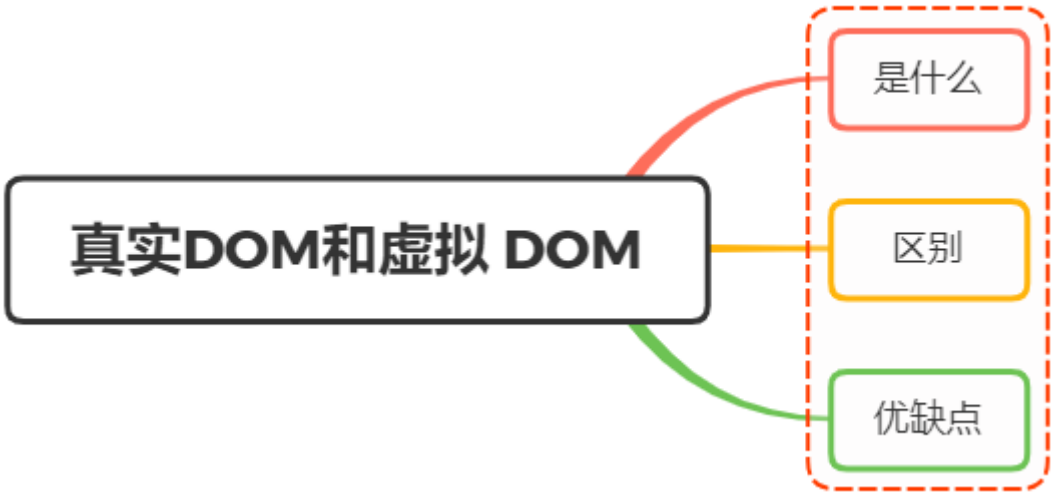
组件的 `props` 改变了，不一定触发 `render` 函数的执行，但是如果 `props` 的值来自于父组件或者祖先组件的 `state`

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，一旦执行了 `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，一旦父组件发生渲染，子组件也会渲染



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一个真实 `DOM` 结构，如下：

```
▼<div id="root">
    <h1>Hello World</h1>
  </div>
```

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

创建虚拟 `DOM` 目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 `DOM` 的属性一一照应

在 `React` 中，`JSX` 是其一大特性，可以让你在 `JS` 中通过使用 `XML` 的方式去直接声明界面的 `DOM` 结构

```
// 创建 h1 标签，右边千万不能加引号
const vDom = <h1>Hello World</h1>;
// 找到 <div id="root"></div> 节点
const root = document.getElementById("root");
// 把创建的 h1 标签渲染到 root 节点上
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：

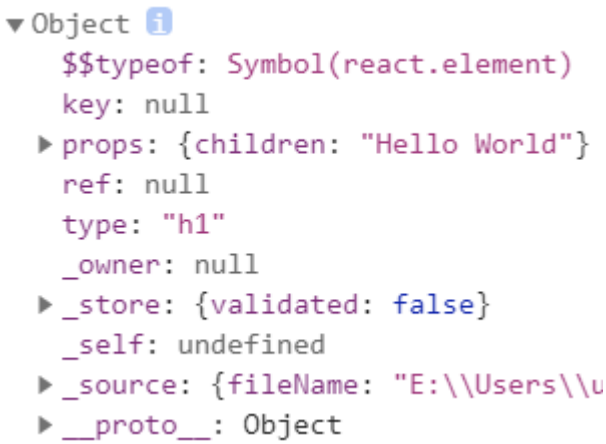
```
const vDom = React.createElement(
```

```
  'h1',
  { className: 'hClass', id: 'hId' },
  'hello world'
)
```

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

通过 `console.log(VDOM)`，则能够得到虚拟 `VDOM` 消息



所以可以得到，`JSX` 通过 `babel` 的方式转化成 `React.createElement` 执行，返回值是一个对象，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一遍流程

当你在一次操作时，需要更新 10 个 `DOM` 节点，浏览器没这么智能，收到第一个更新 `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` 内容保存到本地的一个 `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

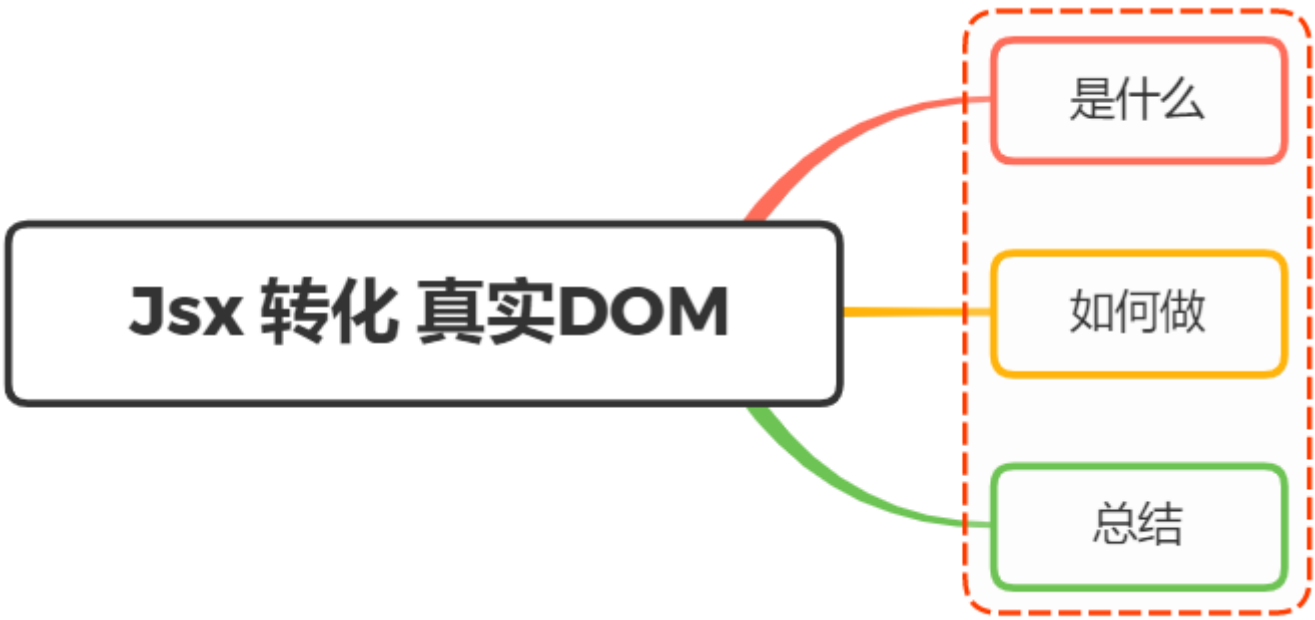
使用虚拟 `DOM` 的优势如下：

- 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` 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 `React` 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
<div>
  < img src="avatar.png" className="profile" />
  <Hello />
</div>
```

会被 `bebel` 转化成如下：

```jsx
React.createElement(
  "div",
  null,
  React.createElement("img", {
    src: "avatar.png",
    className: "profile"
  }),
  React.createElement(Hello, null)
);
```

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：

```jsx
ReactDOM.render(<App />,  document.getElementById("root"));
```

19.2. 过程

在 `react` 中，节点大致可以分成四个类别：

- 原生标签节点
- 文本节点
- 函数组件
- 类组件

如下所示：

```jsx
class ClassComponent extends Component {
  static defaultProps = {
```

```
    color: "pink"
  };
  render() {
    return (
      <div className="border">
        <h3>ClassComponent</h3>
        <p className={this.props.color}>{this.props.name}</p >
      </div>
    );
  }
}
function FunctionComponent(props) {
  return (
    <div className="border">
      FunctionComponent
      <p>{props.name}</p >
    </div>
  );
}
const jsx = (
  <div className="border">
    <p>xx</p >
    < a href=" ">xxx</ a>
    <FunctionComponent name="函数组件" />
    <ClassComponent name="类组件" color="red" />
  </div>
);
```

这些类别最终都会被转化成 `React.createElement` 这种形式

`React.createElement` 其被调用时会传入标签类型 `type` ，标签属性 `props` 及若干子元素 `children` ，作用是生成一个虚拟 `Dom` 对象，如下所示：

```
function createElement(type, config, ...children) {
    if (config) {
        delete config.__self;
        delete config.__source;
    }
    // ! 源码中做了详细处理，比如过滤掉key、ref等
    const props = {
        ...config,
        children: children.map(child =>
   typeof child === "object" ? child : createTextNode(child)
  )
    };
```

```
    return {
        type,
        props
    };
}
function createTextNode(text) {
    return {
        type: TEXT,
        props: {
            children: [],
            nodeValue: text
        }
    };
}
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：

```
ReactDOM.render(element, container[, callback])
```

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：

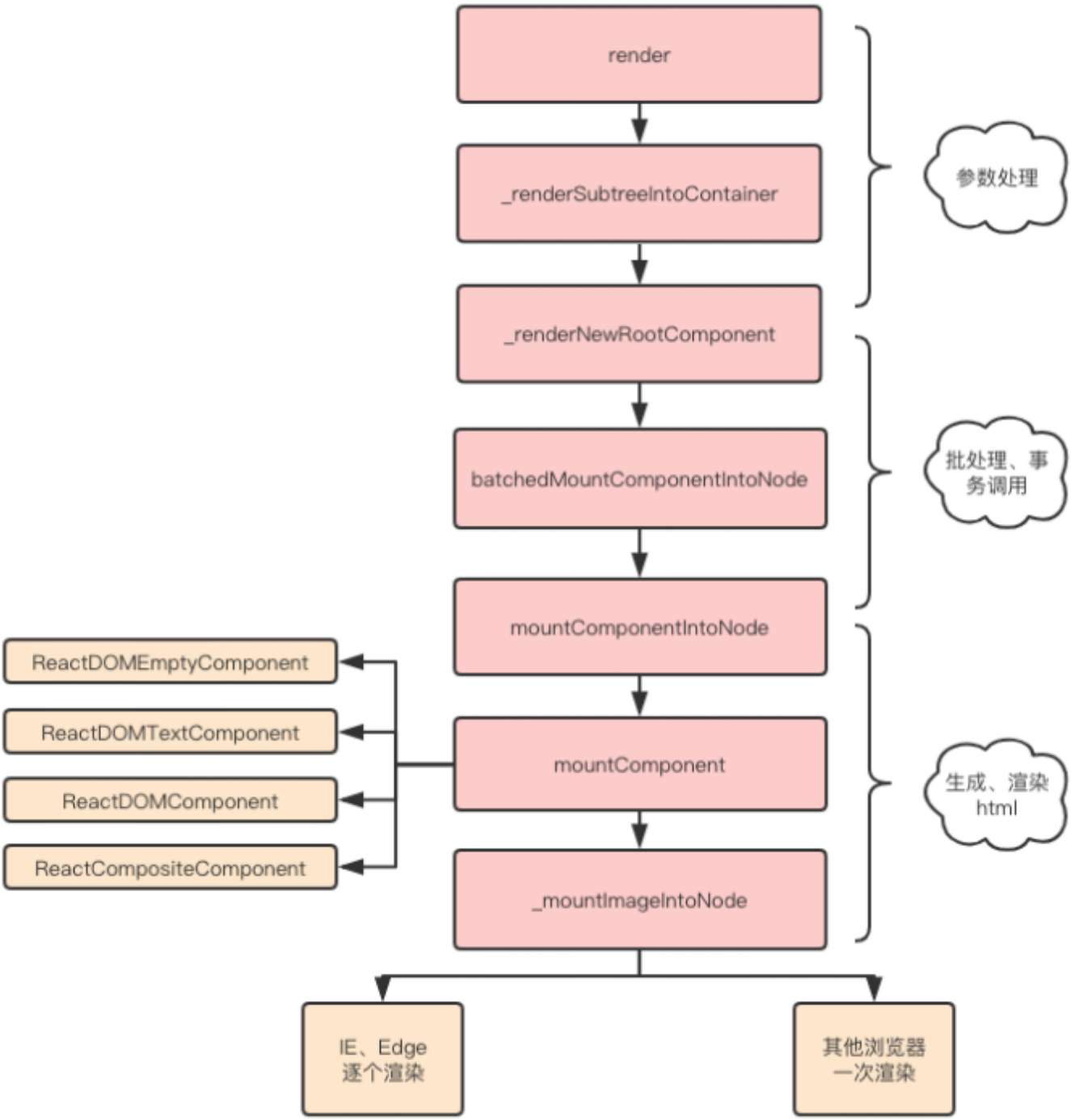
```
function render(vnode, container) {
    console.log("vnode", vnode); // 虚拟DOM对象
    // vnode _> node
    const node = createNode(vnode, container);
    container.appendChild(node);
```

```
}
// 创建真实DOM节点
function createNode(vnode, parentNode) {
    let node = null;
    const {type, props} = vnode;
    if (type === TEXT) {
        node = document.createTextNode("");
    } else if (typeof type === "string") {
        node = document.createElement(type);
    } else if (typeof type === "function") {
        node = type.isReactComponent
            ? updateClassComponent(vnode, parentNode)
        : updateFunctionComponent(vnode, parentNode);
    } else {
        node = document.createDocumentFragment();
    }
    reconcileChildren(props.children, node);
    updateNode(node, props);
    return node;
}
// 遍历下子vnode，然后把子vnode->真实DOM节点，再插入父node中
function reconcileChildren(children, node) {
    for (let i = 0; i < children.length; i++) {
        let child = children[i];
        if (Array.isArray(child)) {
            for (let j = 0; j < child.length; j++) {
                render(child[j], node);
            }
        } else {
            render(child, node);
        }
    }
}
function updateNode(node, nextVal) {
    Object.keys(nextVal)
        .filter(k => k !== "children")
        .forEach(k => {
        if (k.slice(0, 2) === "on") {
            let eventName = k.slice(2).toLocaleLowerCase();
            node.addEventListener(eventName, nextVal[k]);
        } else {
            node[k] = nextVal[k];
        }
    });
}
// 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;
}
// 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;
    let cmp = new type(props);
    const vvnode = cmp.render();
    const node = createNode(vvnode, parentNode);
    return node;
}
export default {
    render
};
```

19.3. 总结

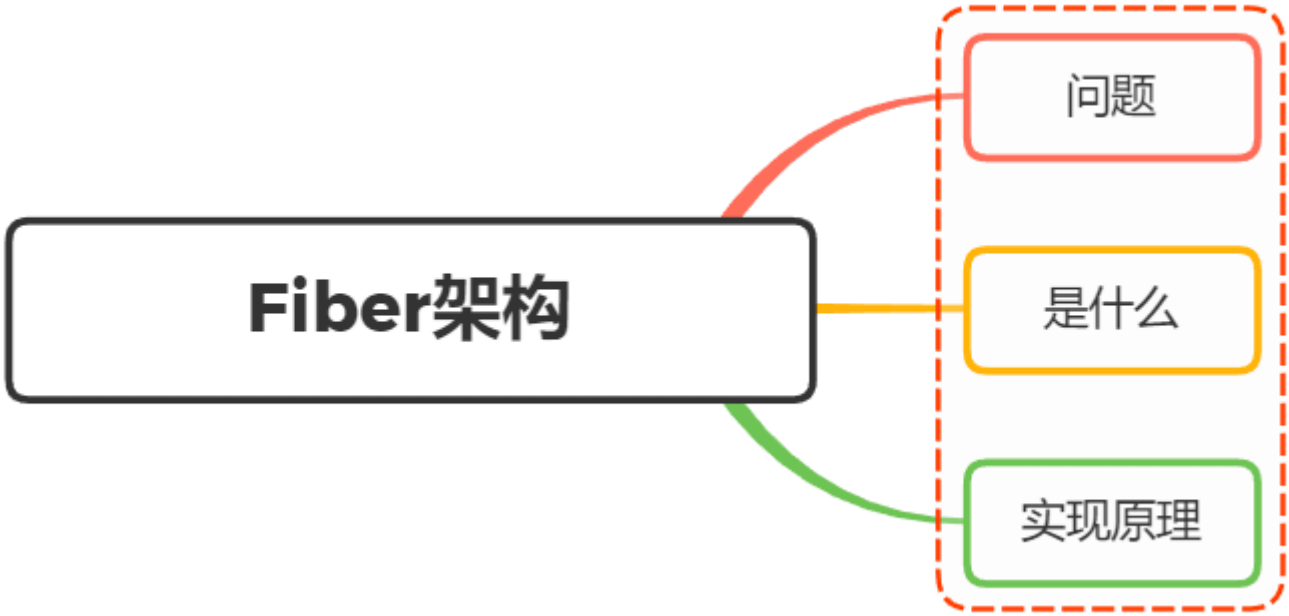
在 `react` 源码中，虚拟 `Dom` 转化成真实 `Dom` 整体流程如下图所示：



其渲染流程如下所示：

- 使用React.createElement或JSX编写React组件，实际上所有的 JSX 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。
- 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

20. 说说对Fiber架构的理解？解决了什么问题？



20.1. 问题

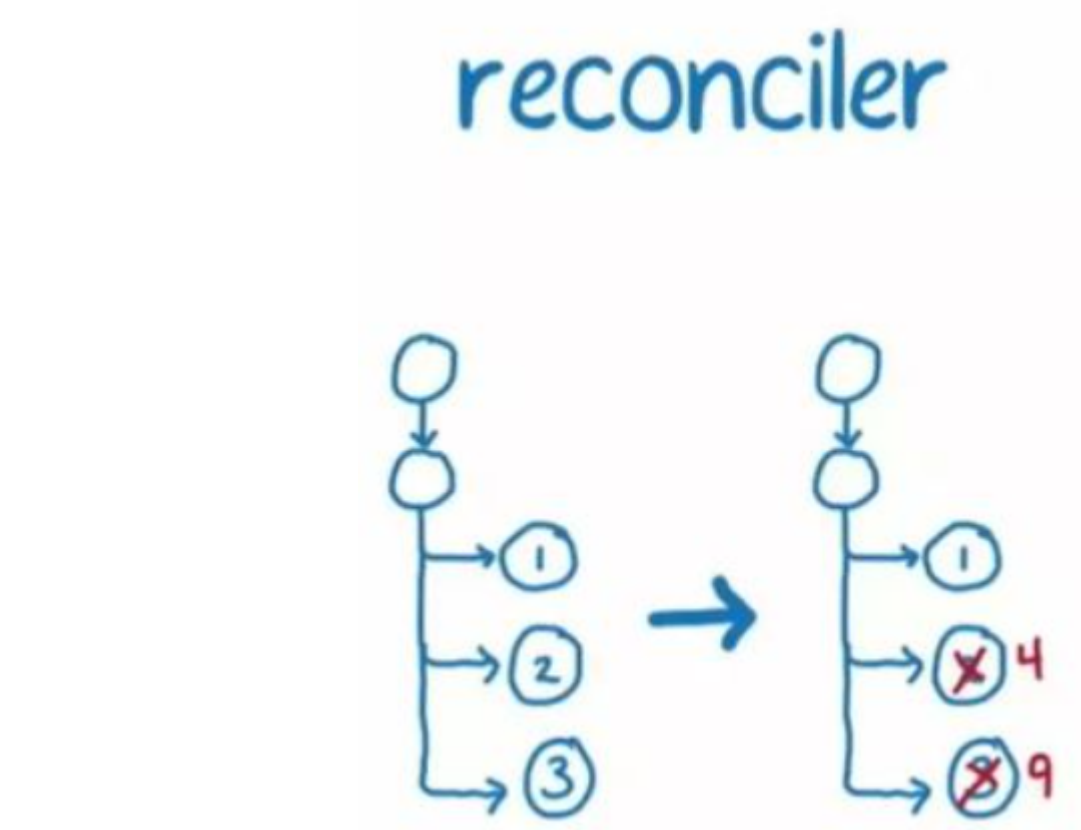
`JavaScript` 引擎和页面渲染引擎两个线程是互斥的，当其中一个线程执行时，另一个线程只能挂起等待

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 Reconciler` 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` 线程会一直执行，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



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requestIdleCallback api，浏览器空闲的时候执行
- dom diff树变成了链表，一个dom对应两个fiber（一个链表），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`

一个 `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：

```
type Fiber = {
  // 用于标记fiber的WorkTag类型，主要表示当前fiber代表的组件类型如FunctionComponent、ClassComponent等
  tag: WorkTag,
  // ReactElement里面的key
```

```
  key: null | string,
  // ReactElement.type，调用
createElement
的第一个参数
  elementType: any,
  // The resolved function/class/ associated with this fiber.
  // 表示当前代表的节点类型
  type: any,
  // 表示当前FiberNode对应的element组件实例
  stateNode: any,
  // 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
  return: Fiber | null,
  // 指向自己的第一个子节点
  child: Fiber | null,
  // 指向自己的兄弟结构，兄弟节点的return指向同一个父节点
  sibling: Fiber | null,
  index: number,
  ref: null | (((handle: mixed) => void) & { _stringRef: ?string }) |
RefObject,
  // 当前处理过程中的组件props对象
  pendingProps: any,
  // 上一次渲染完成之后的props
  memoizedProps: any,
  // 该Fiber对应的组件产生的Update会存放在这个队列里面
  updateQueue: UpdateQueue<any> | null,
  // 上一次渲染的时候的state
  memoizedState: any,
  // 一个列表，存放这个Fiber依赖的context
  firstContextDependency: ContextDependency<mixed> | null,
  mode: TypeOfMode,
  // Effect
  // 用来记录Side Effect
  effectTag: SideEffectTag,
  // 单链表用来快速查找下一个side effect
  nextEffect: Fiber | null,
  // 子树中第一个side effect
  firstEffect: Fiber | null,
  // 子树中最后一个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,
```

```
51 }
```

20.3. 如何解决

`Fiber` 把渲染更新过程拆分成多个子任务，每次只做一小部分，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` 节点对应一个 `React element`，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。

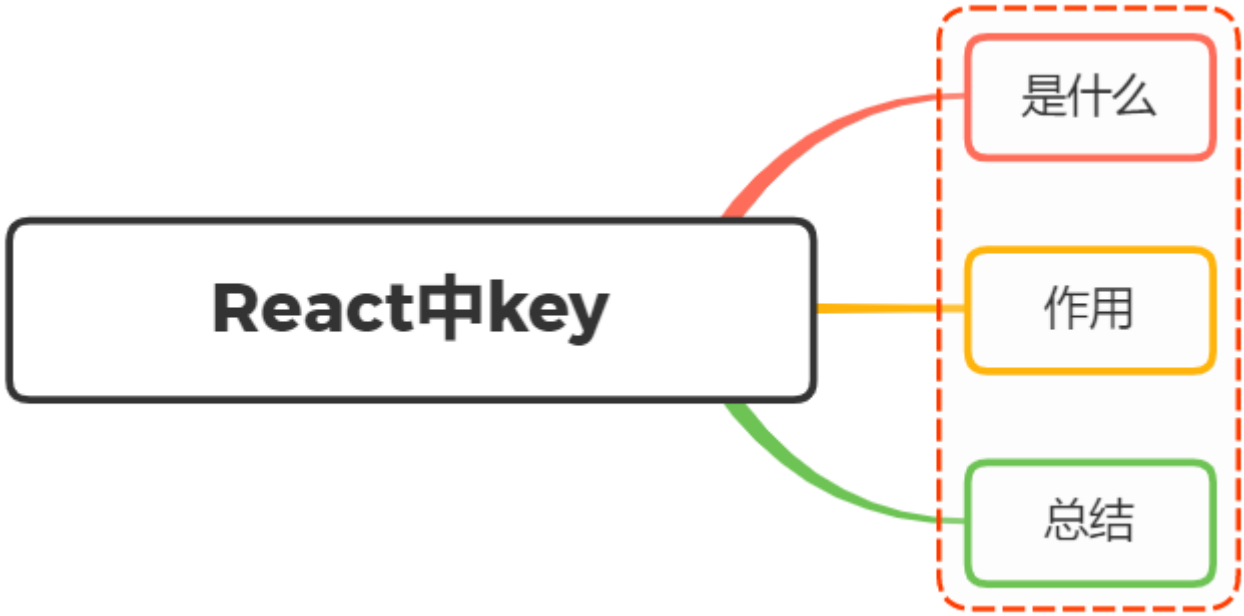
作为动态的工作单元来说，每个 `Fiber` 节点保存了本次更新中该组件改变的状态、要执行的工作。

每个 Fiber 节点有个对应的 `React element`，多个 `Fiber` 节点根据如下三个属性构建一颗树：

```
// 指向父级Fiber节点
this.return = null
// 指向子Fiber节点
this.child = null
// 指向右边第一个兄弟Fiber节点
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一个示例：

```
const data = [
  { id: 0, name: 'abc' },
  { id: 1, name: 'def' },
  { id: 2, name: 'ghi' },
  { id: 3, name: 'jkl' }
];
const ListItem = (props) => {
  return <li>{props.name}</li>;
};
const List = () => {
  return (
    <ul>
      {data.map((item) => (
        <ListItem name={item.name}></ListItem>
      ))}
    </ul>
  );
};
```

然后在输出就可以看到 `react` 所提示的警告信息：

```
Each child in a list should have a unique "key" prop.
```

根据意思就可以得到渲染列表的每一个子元素都应该需要一个唯一的 `key` 值

在这里可以使用列表的 `id` 属性作为 `key` 值以解决上面这个警告

```
const List = () => {
  return (
    <ul>
      {data.map((item) => (
        <ListItem name={item.name} key={item.id}></ListItem>
      ))}
    </ul>
  );
};
```

21.2. 作用

跟 `Vue` 一样， `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` 的值需要为每一个元素赋予一个确定的标识

如果列表数据渲染中，在数据后面插入一条数据， `key` 作用并不大，如下：

```
this.state = {
    numbers:[111,222,333]
}
insertMovie() {
  const newMovies = [...this.state.numbers, 444];
  this.setState({
    movies: newMovies
  })
}
<ul>
    {
        this.state.movies.map((item, index) => {
            return <li>{item}</li>
        })
    }
</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，在最后一个比较的时候，则需要插入到新的 `DOM` 树中

因此，在这种情况下，元素有无 `key` 属性意义并不大

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：

```
insertMovie() {
  const newMovies = [000 ,...this.state.numbers];
  this.setState({
    movies: newMovies
  })
}
```

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一下，节点不会发生变化

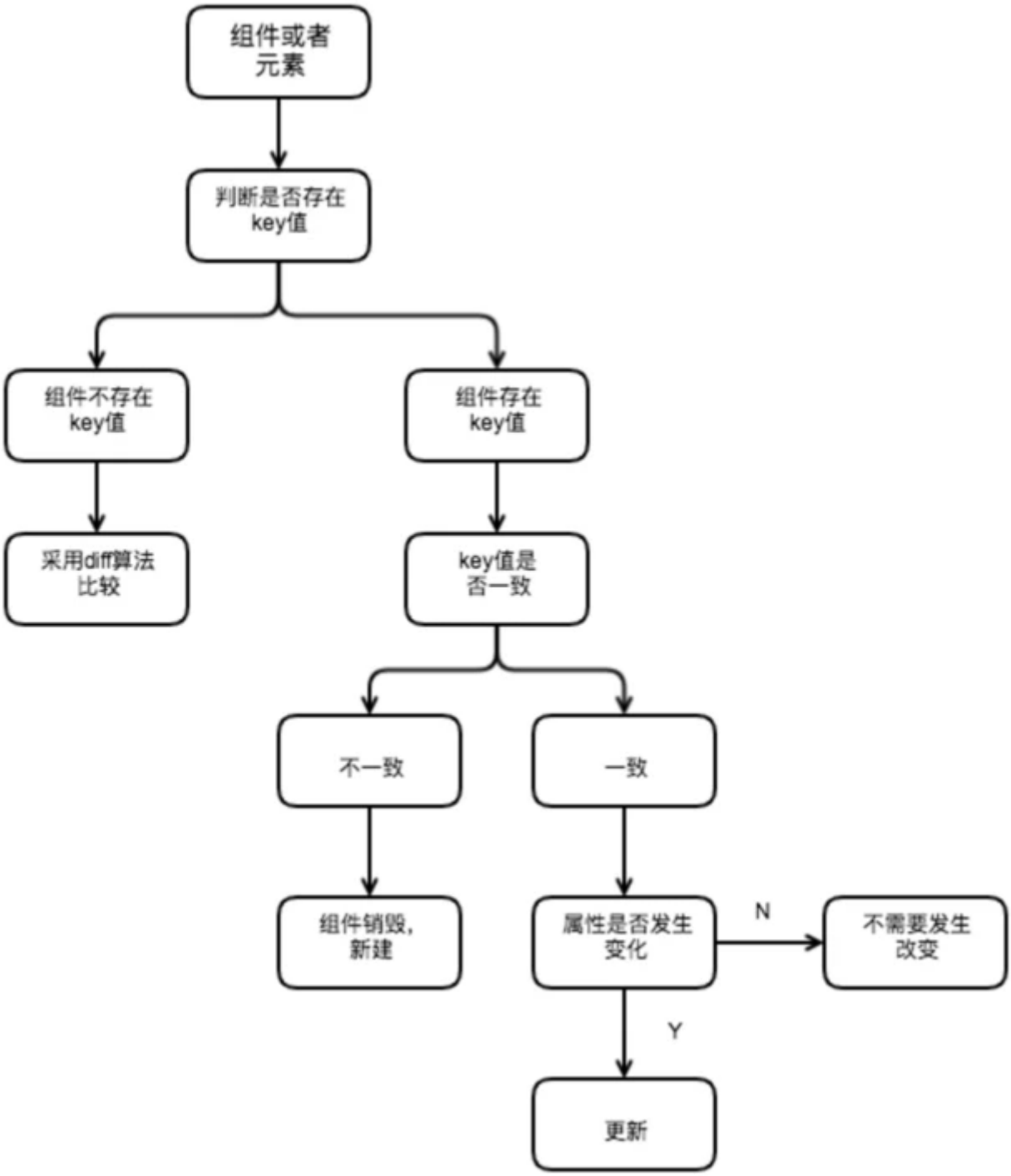
而写 `key` 则涉及到了节点的增和删，发现旧 `key` 不存在了，则将其删除，新 `key` 在之前没有，则插入，这就增加性能的开销

21.3. 总结

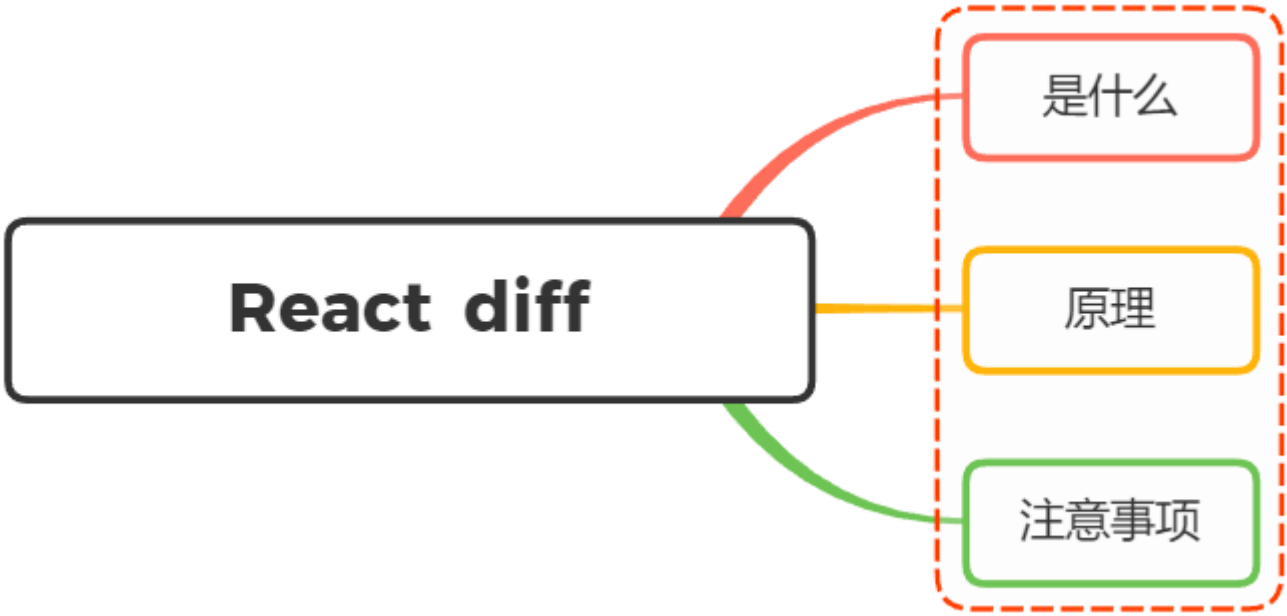
良好使用 `key` 属性是性能优化的非常关键的一步，注意事项为：

- 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：



22. 说说React diff的原理是什么？

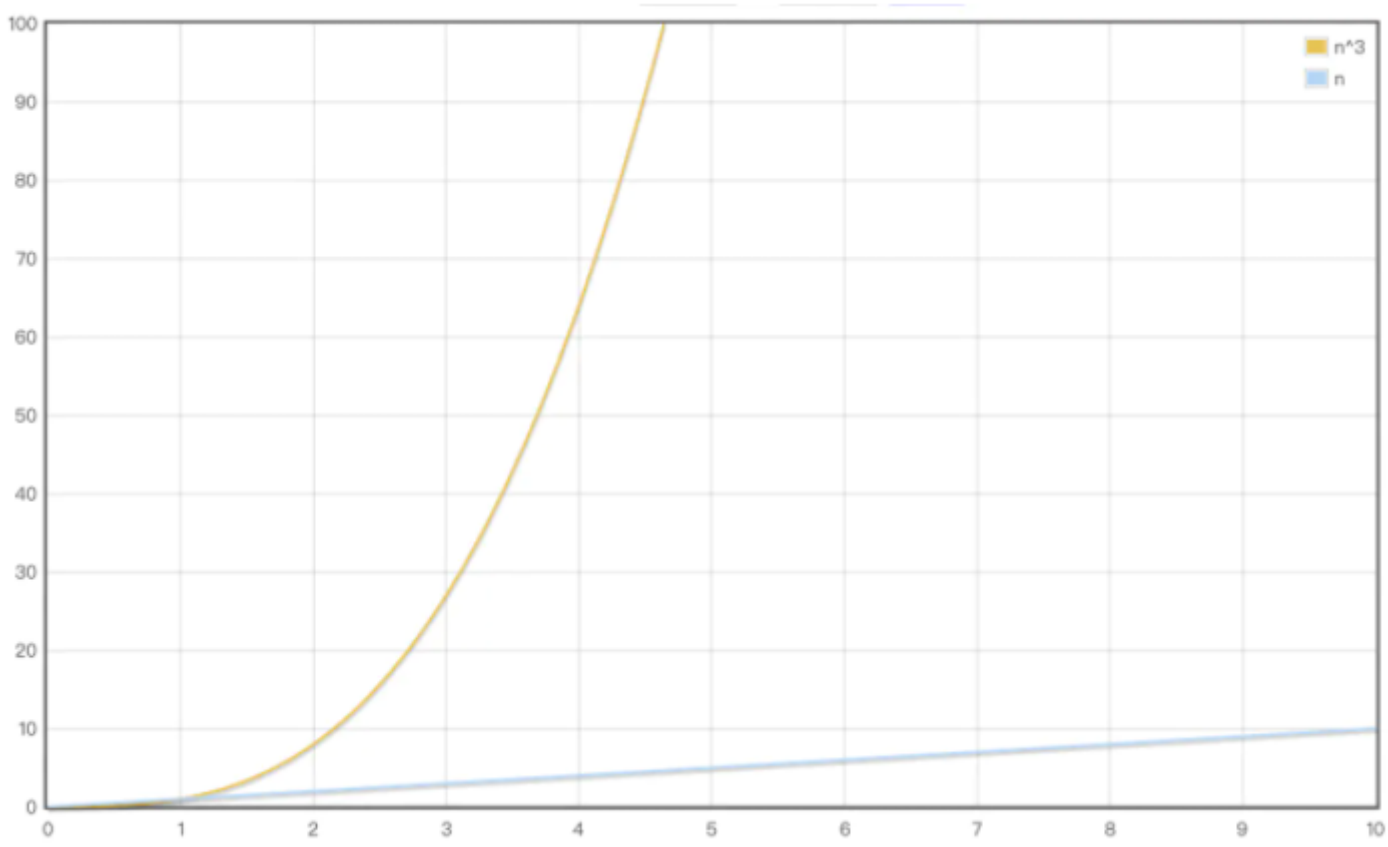


22.1. 是什么

跟 `Vue` 一致， `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` 将算法进行一个优化，复杂度 `O(n)` ，两者效率差距如下图：



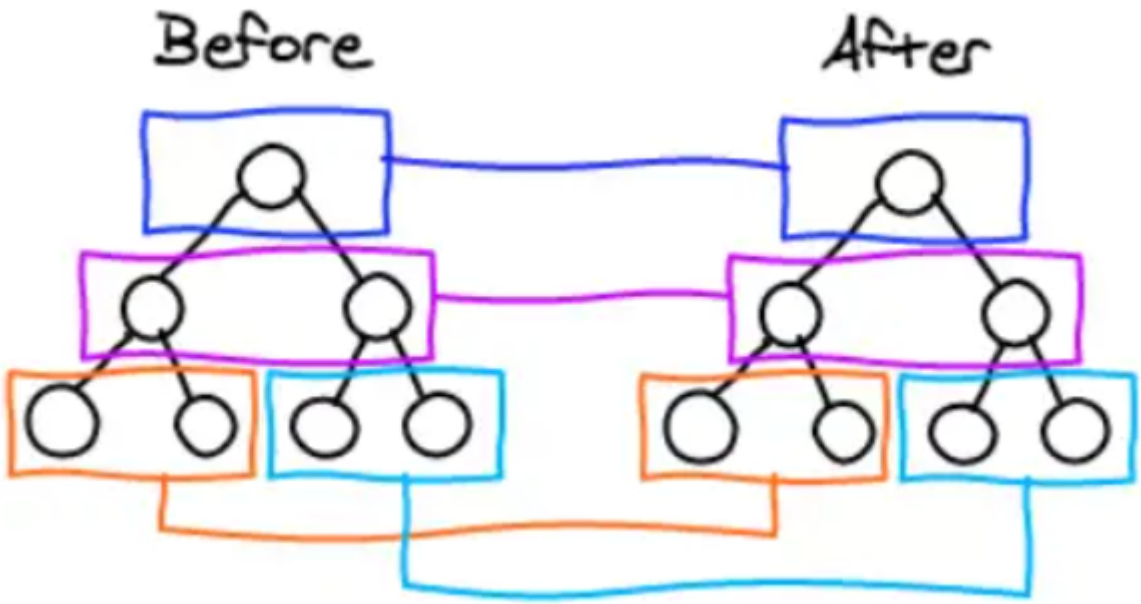
22.2. 原理

`react` 中 `diff` 算法主要遵循三个层级的策略：

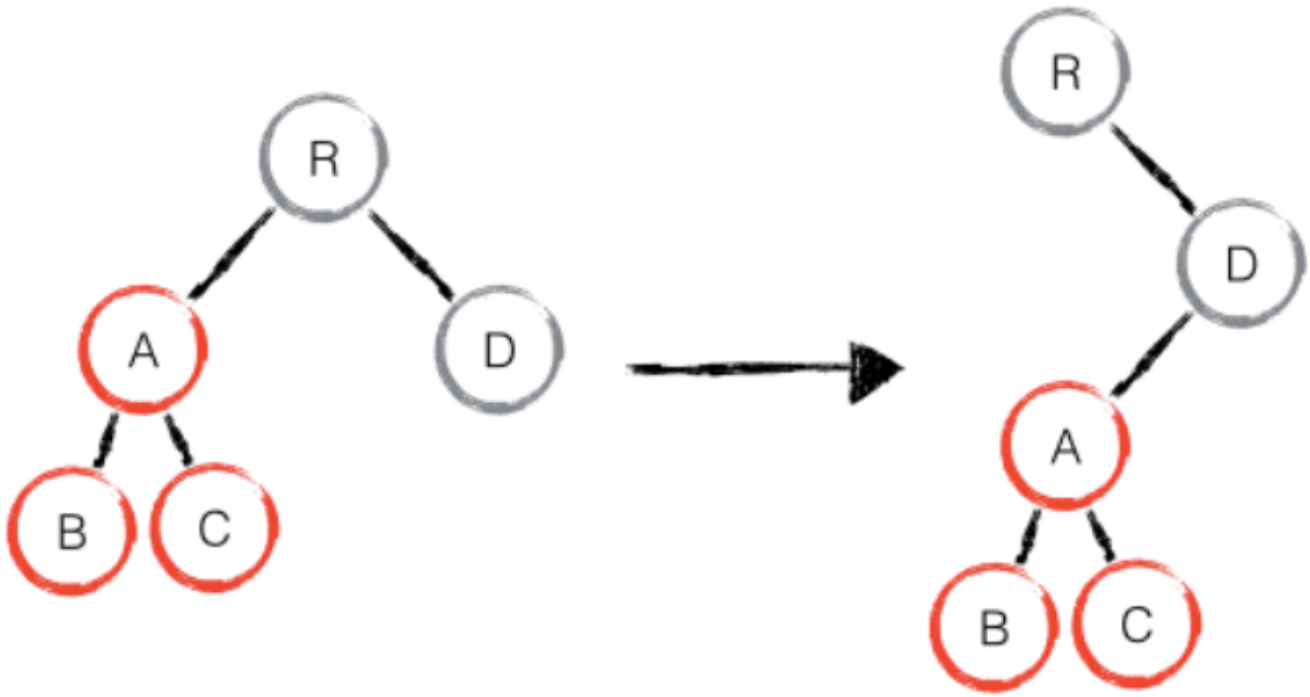
- tree层级
- conponent 层级
- element 层级

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



只有删除、创建操作，没有移动操作，如下图：

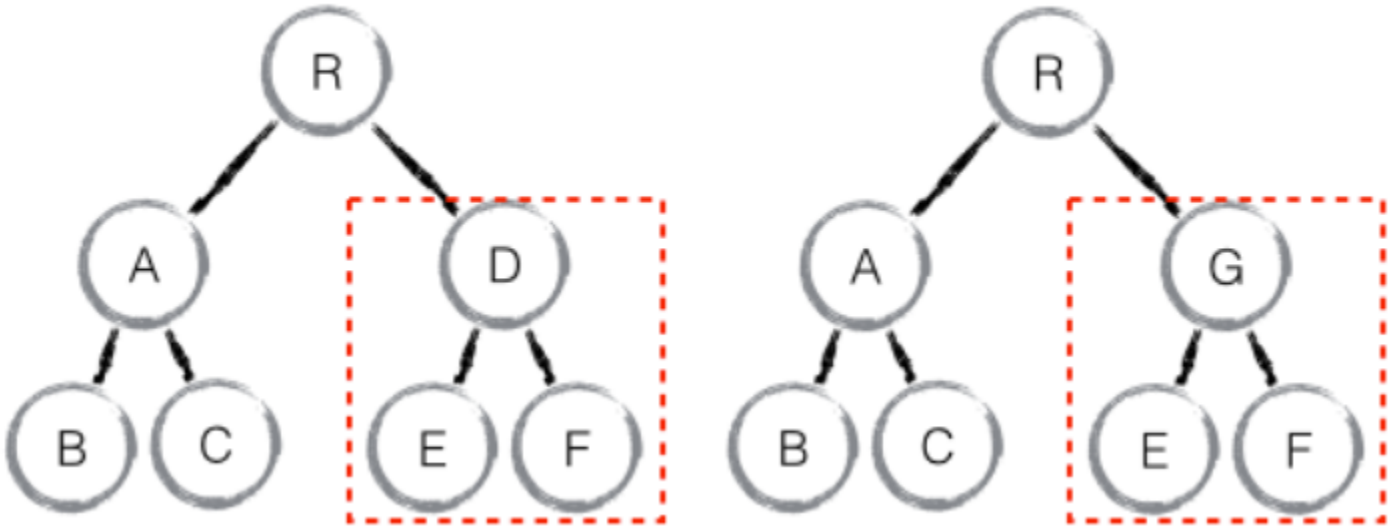


`react` 发现新树中，R节点下没有了A，那么直接删除A，在D节点下创建A以及下属节点

上述操作中，只有删除和创建操作

22.2.2. conponent层级

如果是同一个类的组件，则会继续往下 `diff` 运算，如果不是一个类的组件，那么直接删除这个组件下的所有子节点，创建新的



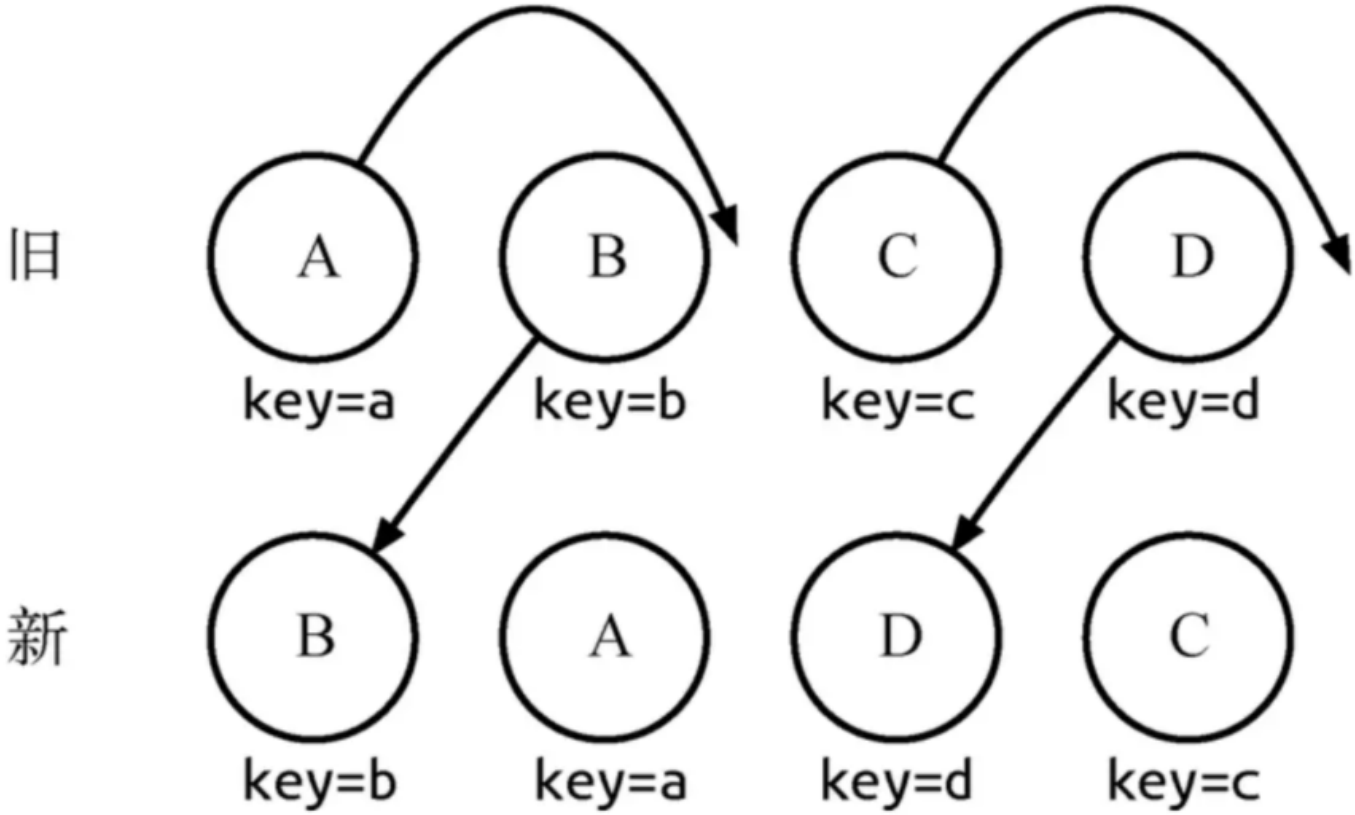
当 `component D` 换成了 `component G` 后，即使两者的结构非常类似，也会将 `D` 删除再重新创建 `G`

22.2.3. element层级

对于比较同一层级的节点们，每个节点在对应的层级用唯一的 `key` 作为标识

提供了 3 种节点操作，分别为 `INSERT_MARKUP` (插入)、 `MOVE_EXISTING` (移动)和 `REMOVE_NODE` (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：

index	节点	oldIndex	maxIndex	操作
0	B	1	0	oldIndex(1)>maxIndex(0),maxIndex=oldIndex,maxIndex变为1
1	A	0	1	oldIndex(0)<maxIndex(1),节点A移动至index(1)的位置
2	D	3	1	oldIndex(3)>maxIndex(1),maxIndex=oldIndex,maxIndex变为3
3	C	2	3	oldIndex(2)<maxIndex(3),节点C移动至index(3)的位置

- index： 新集合的遍历下标。
- oldIndex：当前节点在老集合中的下标
- 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oldIndex和maxIndex，规则如下：

- 当oldIndex>maxIndex时，将oldIndex的值赋值给maxIndex

- 当oldIndex=maxIndex时，不操作
- 当oldIndex<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，oldIndex=1；满足 maxIndex< oldIndex，因此B节点不动，此时 maxIndex= Math.max(oldIndex, maxIndex)，就是1
- 节点A：此时maxIndex=1，oldIndex=0；不满足maxIndex< oldIndex，因此A节点进行移动操作，此时maxIndex= Math.max(oldIndex, maxIndex)，还是1
- 节点D：此时maxIndex=1, oldIndex=3；满足maxIndex< oldIndex，因此D节点不动，此时 maxIndex= Math.max(oldIndex, maxIndex)，就是3
- 节点C：此时maxIndex=3，oldIndex=2；不满足maxIndex< oldIndex，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，`diff` 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

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：

```
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
<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：

```
1.加key
<div key='1'>1</div>             <div key='1'>1</div>

<div key='2'>2</div>             <div key='3'>3</div>

<div key='3'>3</div>  ========>  <div key='2'>2</div>

<div key='4'>4</div>             <div key='5'>5</div>

<div key='5'>5</div>             <div key='4'>4</div>

操作：节点2移动至下标为2的位置，节点4移动至下标为4的位置。
```

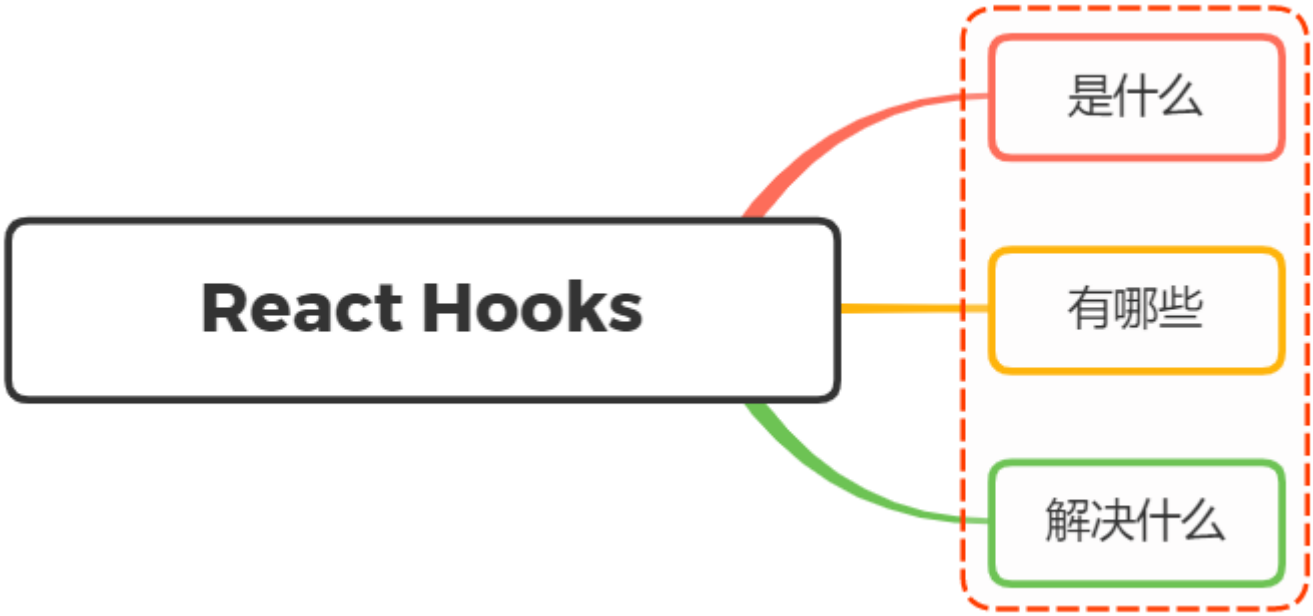
```
2.不加key
<div>1</div>              <div>1</div><div>2</div>              <div>3</div>
<div>3</div>  ========>   <div>2</div><div>4</div>              <div>5</div>
<div>5</div>              <div>4</div>操作：修改第1个到第5个节点的innerText
```

如果我们对这个集合进行增删的操作改成[1,3,2,5,6]

```
1.加key
<div key='1'>1</div>              <div key='1'>1</div>

<div key='2'>2</div>              <div key='3'>3</div>

<div key='3'>3</div>  ========>   <div key='2'>2</div>

<div key='4'>4</div>              <div key='5'>5</div>

<div key='5'>5</div>              <div key='6'>6</div>

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。
2.不加key
<div>1</div>              <div>1</div>

<div>2</div>              <div>3</div>

<div>3</div>  ========>   <div>2</div>

<div>4</div>              <div>5</div>

<div>5</div>              <div>6</div>
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？



23.1. 是什么

`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 `state` 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，只负责渲染的一些工作

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

上面讲到， `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

最常见的 `hooks` 有如下：

- useState
- useEffect
- 其他

23.2.1. useState

首先给出一个例子，如下：

```
import React, { useState } from 'react';
function Example() {
  // 声明一个叫 "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>
  );
}
```

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，返回值是一个数组，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：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p >
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

从上述两种代码分析，可以看出两者区别：

- 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.state.count` 的方式获取
- 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 `this` 指向不明确的情况

23.2.2. useEffect

`useEffect` 可以让我们在函数组件中进行一些带有副作用的操作

同样给出一个计时器示例：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  componentDidMount() {
    document.title =
You clicked ${this.state.count} times
;
  }
  componentDidUpdate() {
    document.title =
You clicked ${this.state.count} times
;
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p >
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

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` 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

对应的 `useEffect` 示例如下：

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {    document.title =
You clicked ${count} times
;  });
  return (
    <div>
      <p>You clicked {count} times</p >
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

`useEffect` 第一个参数接受一个回调函数，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：

```
useEffect(() => {
  document.title =
You clicked ${count} times
;
}, [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 将对前一次渲染的 `[5]` 和后一次渲染的 `[5]` 进行比较，如果是相等则跳过 `effects` 执行

回调函数中可以返回一个清除函数，这是 `effect` 可选的清除机制，相当于类组件中 `componentwillUnmount` 生命周期函数，可做一些清除副作用的操作，如下：

```
useEffect(() => {
    function handleStatusChange(status) {
        setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
```

```
7         ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
  handleStatusChange);
8     };
9 });
```

所以， `useEffect` 相当于 `componentDidMount` ， `componentDidUpdate` 和 `componentWillUnmount` 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：

- useReducer
- useCallback
- useMemo
- useRef

23.3. 解决什么

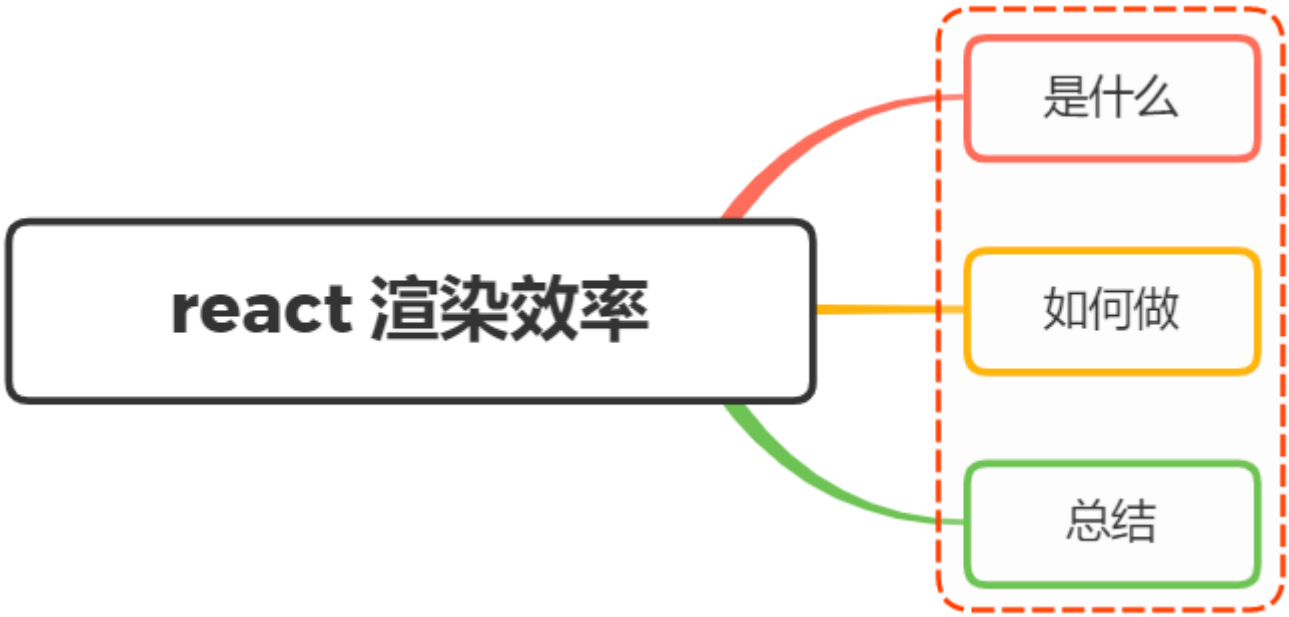
通过对上面的初步认识，可以看到 `hooks` 能够更容易解决状态相关的重用的问题：

- 每调用useHook一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，在我们日常使用中，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一

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` ，父组件一旦发生 `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：

- shouldComponentUpdate
- PureComponent
- 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

24.2.2. PureComponent

跟 `shouldComponentUpdate` 原理基本一致，通过对 `props` 和 `state` 的浅比较结果来实现 `shouldComponentUpdate` ，源码大致如下：

```
if (this._compositeType === CompositeTypes.PureClass) {
    shouldUpdate = !shallowEqual(prevProps, nextProps) || !
shallowEqual(inst.state, nextState);
}
```

`shallowEqual` 对应方法大致如下：

- const hasOwnProperty = Object.prototype.hasOwnProperty;
  /**
  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
  https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/is
  */
  function is(x: mixed, y: mixed): boolean {
  if (x === y) {
   return x !== 0 || y !== 0 || 1 / x === 1 / y;
  } else {
   return x !== x && y !== y;
  }
  }
  function shallowEqual(objA: mixed, objB: mixed): boolean {
   // 首先对基本类型进行比较
   if (is(objA, objB)) {
    return true;
   }
   if (typeof objA !== 'object' || objA === null ||
      typeof objB !== 'object' || objB === null) {
    return false;
   }
   const keysA = Object.keys(objA);
   const keysB = Object.keys(objB);
   // 长度不相等直接返回false
   if (keysA.length !== keysB.length) {
    return false;
   }
   // key相等的情况下，再去循环比较
   for (let i = 0; i < keysA.length; i++) {
    if (
     !hasOwnProperty.call(objB, keysA[i]) ||
     !is(objA[keysA[i]], objB[keysA[i]])
) {
     return false;
    }
   }
   return true;
  }

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，其实也是一个高阶组件，与 `PureComponent` 十分类似。但不同的是， `React.memo` 只能用于函数组件

```
import { memo } from 'react';
function Button(props) {
  // Component code
}
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

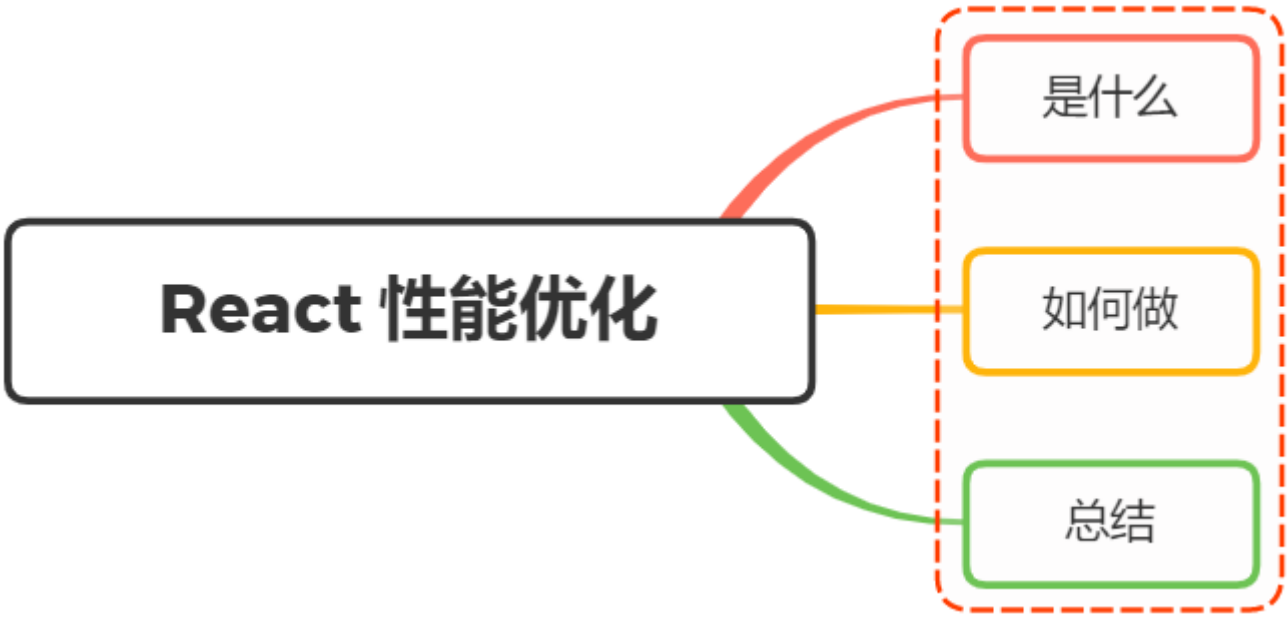
```
function arePropsEqual(prevProps, nextProps) {
  // your code
  return prevProps === nextProps;
}
export default memo(Button, arePropsEqual);
```

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，如果一个过大，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?

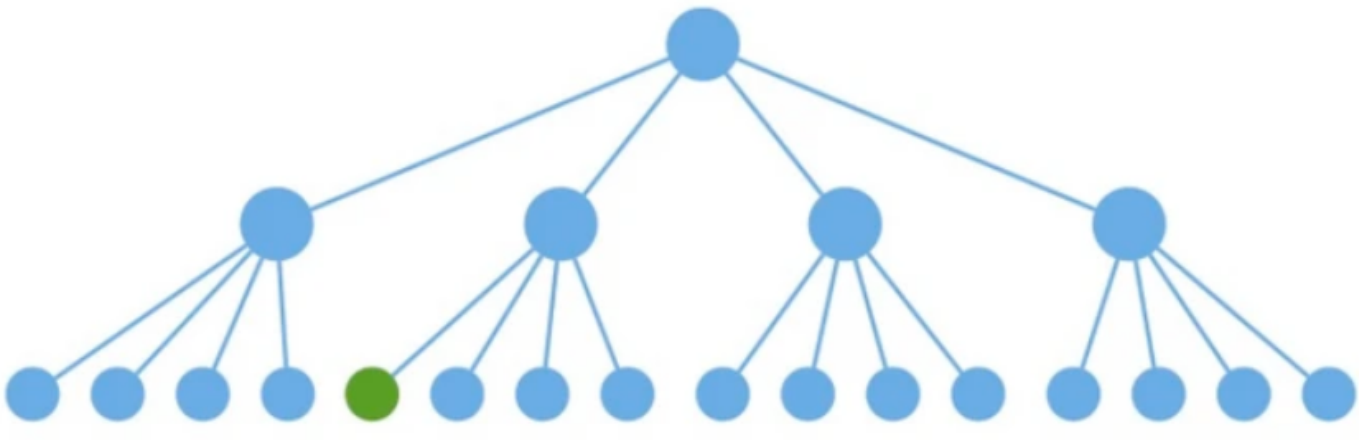


25.1. 是什么

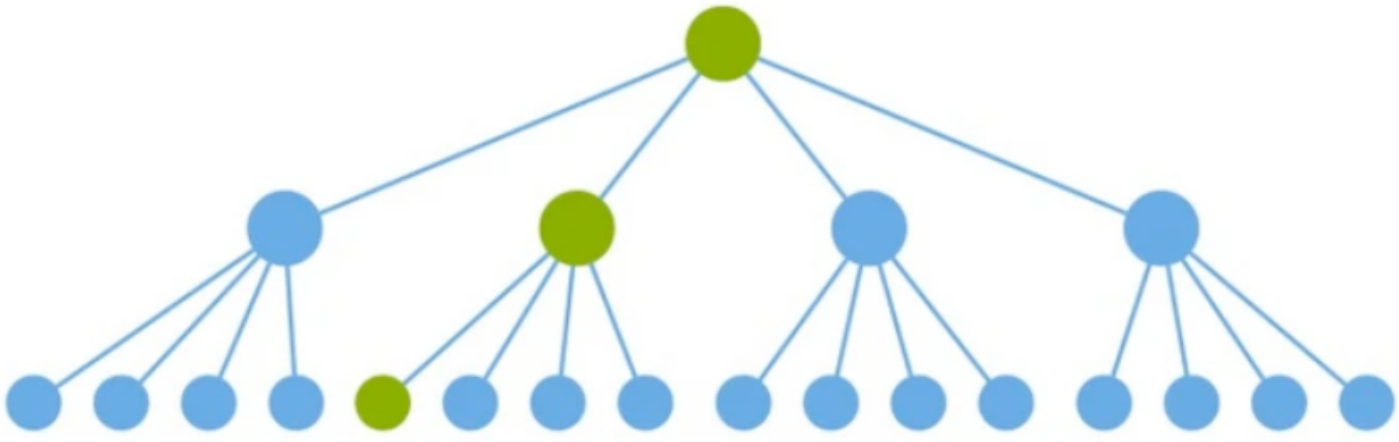
`React` 凭借 `virtual DOM` 和 `diff` 算法拥有高效的性能，但是某些情况下，性能明显可以进一步提高

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` ，父组件一旦发生 `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` ：



但是 `react` 的默认做法是调用所有组件的 `render` ，再对生成的虚拟 `DOM` 进行对比（黄色部分），如不变则不进行更新

图片 加载失败

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，主要手段是通过 `shouldComponentUpdate` 、 `PureComponent` 、 `React.memo` ，这三种形式这里就不再复述

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

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：

```
import React from "react";
export default class InlineFunctionComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>Welcome Guest</h1>
        <input type="button" onClick={(e) => { this.setState({inputValue:
e.target.value}) }} value="Click For Inline Function" />
      </div>
    )
  }
}
```

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：

```
import React from "react";
export default class InlineFunctionComponent extends React.Component {

  setNewStateData = (event) => {
    this.setState({
      inputValue: e.target.value
    })
  }

  render() {
    return (
      <div>
        <h1>Welcome Guest</h1>
        <input type="button" onClick={this.setNewStateData} value="Click For
Inline Function" />
      </div>
    )
  }
}
```

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：

```
export default class NestedRoutingComponent extends React.Component {
    render() {
        return (
            <>
                <h1>This is the Header Component</h1>
                <h2>Welcome To Demo Page</h2>
            </>
        )
    }
}
```

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一个方法实例，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

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：

```
const johanComponent = React.lazy(() => import(/* webpackChunkName:
"johanComponent" */ './myAwesome.component'));
export const johanAsyncComponent = props => (
  <React.Suspense fallback={<Spinner />}>
    <johanComponent {...props} />
  </React.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，需要起一个 `node` 服务，可以使用 `express` 、 `koa` 等，调用 `react` 的 `renderToString` 方法，将根组件渲染成字符串，再输出到响应中

例如：

```
import { renderToString } from "react-dom/server";
import MyPage from "./MyPage";
app.get("/", (req, res) => {
  res.write("<!DOCTYPE html><html><head><title>My Page</title></head><body>");
  res.write("<div id='content'>");

  res.write(renderToString(<MyPage/>));
  res.write("</div></body></html>");
  res.end();
});
```

客户端使用render方法来生成HTML

```
import ReactDOM from 'react-dom';
import MyPage from "./MyPage";
ReactDOM.render(<MyPage />, document.getElementById('app'));
```

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` 项目的性能更上一层楼

26. 说说你对React Router的理解？常用的Router组件有哪些？

图片 加载失败

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：

- 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
- 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` ，主要是提供了一些组件：

- BrowserRouter、HashRouter
- Route
- Link、NavLink
- switch
- 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

`BrowserRouter` 是 `history` 模式， `HashRouter` 模式

使用两者作为最顶层组件包裹其他组件

```
1 import { BrowserRouter as Router } from "react-router-dom";
2 export default function App() {
3   return (
```

```
    <Router>
      <main>
        <nav>
          <ul>
            <li>
              < a href=" ">Home</ a>
            </li>
            <li>
              < a href="/about">About</ a>
            </li>
            <li>
              < a href="/contact">Contact</ a>
            </li>
          </ul>
        </nav>
      </main>
    </Router>
  );
}
```

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

```
import { BrowserRouter as Router, Route } from "react-router-dom";
export default function App() {
  return (
    <Router>
      <main>
        <nav>
          <ul>
            <li>
              < a href="/">Home</ a>
            </li>
            <li>
              < a href="/about">About</ a>
            </li>
            <li>
              < a href="/contact">Contact</ a>
```

```
            </li>
          </ul>
        </nav>
        <Route path="/" render={() => <h1>Welcome!</h1>} />
      </main>
    </Router>
  );
}
```

26.2.3. Link、NavLink

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`href`属性

`NavLink`是在`Link`基础之上增加了一些样式属性，例如组件被选中时，发生样式变化，则可以设置`NavLink`的一下属性：

- activeStyle：活跃时（匹配时）的样式
- activeClassName：活跃时添加的class

如下：

```
<NavLink to="/" exact activeStyle={{color: "red"}}>首页</NavLink>
<NavLink to="/about" activeStyle={{color: "red"}}>关于</NavLink>
<NavLink to="/profile" activeStyle={{color: "red"}}>我的</NavLink>
```

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一些路由相关的东西，比如`props.history`

```
const Contact = ({ history }) => (
  <Fragment>
    <h1>Contact</h1>
    <button onClick={() => history.push("/")}>Go to home</button>
    <FakeText />
  </Fragment>
)
```

`props`中接收到的`history`对象具有一些方便的方法，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：

```
const About = ({
  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：

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/profile" component={Profile} />
  <Route path="/:userid" component={User} />
  <Route component={NoMatch} />
</Switch>
```

如果不使用 `switch` 组件进行包裹

除了一些路由相关的组件之外， `react-router` 还提供一些 `hooks` ，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

```
```

```
import { useHistory } from "react-router-dom";
const Contact = () => {
  const history = useHistory();
  return (
    <Fragment>
      <h1>Contact</h1>
      <button onClick={() => history.push("/")}>Go to home</button>
    </Fragment>
  );
};
```

26.2.7. useParams

```
const About = () => {
  const { name } = useParams();
  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

```
import { useLocation } from "react-router-dom";
const Contact = () => {
  const { pathname } = useLocation();
  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
- search传递参数
- to传入对象

26.3.1. 动态路由

动态路由的概念指的是路由中的路径并不会固定

例如将 `path` 在 `Route` 匹配时写成 `/detail/:id` ，那么 `/detail/abc` 、 `/detail/123` 都可以匹配到该 `Route`

```
<NavLink to="/detail/abc123">详情</NavLink>
<Switch>
    ... 其他Route
    <Route path="/detail/:id" component={Detail}/>
    <Route component={NoMatch} />
</Switch>
```

获取参数方式如下：

```
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；

```
<NavLink to="/detail2?name=why&age=18">详情2</NavLink>
<Switch>
  <Route path="/detail2" component={Detail2}/>
</Switch>
```

获取形式如下：

```
console.log(props.location.search)
```

26.3.3. to传入对象

传递方式如下：

```
<NavLink to={{
    pathname: "/detail2",
    query: {name: "kobe", age: 30},
    state: {height: 1.98, address: "洛杉矶"},
    search: "?apikey=123"
  }}>
  详情2
</NavLink>
```

获取参数的形式如下：

```
console.log(props.location)
```

27. 说说React Router有几种模式？实现原理？

图片 加载失败

27.1. 是什么

在单页应用中，一个 `web` 项目只有一个 `html` 页面，一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
- 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：

- HashRouter
- BrowserRouter

这两个组件的使用都十分的简单，作为最顶层组件包裹其他组件，如下所示

```
// 1.import { BrowserRouter as Router } from "react-router-dom";
// 2.import { HashRouter as Router } from "react-router-dom";
import React from 'react';
import {
  BrowserRouter as Router,
  // HashRouter as Router

  Switch,
  Route,
} from "react-router-dom";
import Home from './pages/Home';
import Login from './pages/Login';
import Backend from './pages/Backend';
import Admin from './pages/Admin';
function App() {
  return (
    <Router>
        <Route path="/login" component={Login}/>
        <Route path="/backend" component={Backend}/>
        <Route path="/admin" component={Admin}/>
        <Route path="/" component={Home}/>
    </Router>
  );
}
export default App;
```

27.3. 实现原理

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：

```
import React, { Component } from 'react';
import { Provider } from './context'
// 该组件下Api提供给子组件使用
class HashRouter extends Component {
  constructor() {
    super()
    this.state = {
      location: {
        pathname: window.location.hash.slice(1) || '/'
      }
    }
  }
  // url路径变化 改变location
  componentDidMount() {
    window.location.hash = window.location.hash || '/'
    window.addEventListener('hashchange', () => {
      this.setState({
        location: {
          ...this.state.location,
          pathname: window.location.hash.slice(1) || '/'
        }
      }, () => console.log(this.state.location))
    })
  }
  render() {
    let value = {
      location: this.state.location
    }
    return (
      <Provider value={value}>
        {
          this.props.children
        }
      </Provider>
    );
  }
}
```

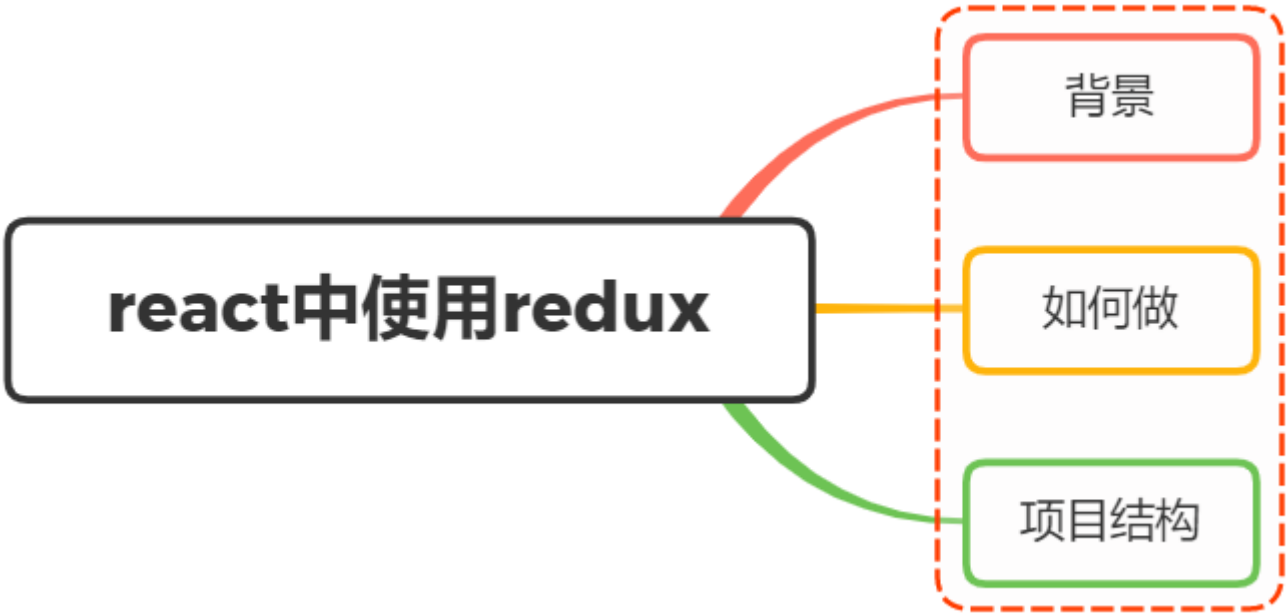
```
38 export default HashRouter;
```

27.4.1. Router

`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

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



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` 是一个视图层面的库

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` 行为 `action` 给 `store`

其他组件通过订阅 `store` 中的状态 `state` 来更新自身的视图

28.2. 如何做

使用 `react-redux` 分成了两大核心：

- Provider
- connection

28.2.1. Provider

在 `redux` 中存在一个 `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

```
<Provider store = {store}>
    <App />
<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：

```
import { connect } from "react-redux";
```

用法如下：

```
connect(mapStateToProps, mapDispatchToProps)(MyComponent)
```

可以传递两个参数：

- mapStateToProps
- mapDispatchToProps

28.2.3. mapStateToProps

把 `redux` 中的数据映射到 `react` 中的 `props` 中去

如下

```
const mapStateToProps = (state) => {
    return {
        // prop : state.xxx  | 意思是将state中的某个数据映射到props中
        foo: state.bar
    }
}
```

组件内部就能够通过 `props` 获取到 `store` 中的数据

```
class Foo extends Component {
    constructor(props){
        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>
        )
    }
}
Foo = connect()(Foo)
export default Foo
```

28.2.4. mapDispatchToProps

将 `redux` 中的 `dispatch` 映射到组件内部的 `props` 中

```
const mapDispatchToProps = (dispatch) => { // 默认传递参数就是dispatch
  return {
    onClick: () => {
      dispatch({
        type: 'increatment'
      });
    }
  };
}
```

```
class Foo extends Component {
    constructor(props){
        super(props);
    }
    render(){
        return(

             <button onClick = {this.props.onClick}>点击increase</button>
        )
    }
}
Foo = connect()(Foo);
export default Foo;
```

28.3. 小结

整体流程图大致如下所示：

图片 加载失败

29. 说说对Redux中间件的理解？常用的中间件有哪些？实现原理？

图片 加载失败

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：

图片 加载失败

其本质上一个函数，对 `store.dispatch` 方法进行了改造，在发出 `Action` 和执行 `Reducer` 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：

- redux-thunk：用于异步操作
- 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一个数组，依次执行

然后作为第二个参数传入到 `createStore` 中

```
const store = createStore(
  reducer,
  applyMiddleware(thunk, logger)
);
```

29.2.1. redux-thunk

`redux-thunk` 是官网推荐的异步处理中间件

默认情况下的 `dispatch(action)` ， `action` 需要是一个 `JavaScript` 的对象

`redux-thunk` 中间件会判断你当前传进来的数据类型，如果是一个函数，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
- 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

所以 `dispatch` 可以写成下述函数的形式：

```
const getHomeMultidataAction = () => {
  return (dispatch) => {
    axios.get("http://xxx.xx.xx.xx/test").then(res => {
      const data = res.data.data;
      dispatch(changeBannersAction(data.banner.list));
      dispatch(changeRecommendsAction(data.recommend.list));
    })
  }
}
```

29.2.2. redux-logger

如果想要实现一个日志功能，则可以使用现成的 `redux-logger`

```

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

```
import createLogger from 'redux-logger';
const logger = createLogger();
const store = createStore(
  reducer,
  applyMiddleware(logger)
);
```

这样我们就能简单通过中间件函数实现日志记录的信息

29.3. 实现原理

首先看看 `applyMiddlewares` 的源码

```
export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, preloadedState, enhancer) => {
    var store = createStore(reducer, preloadedState, enhancer);
    var dispatch = store.dispatch;
    var chain = [];
    var middlewareAPI = {
      getState: store.getState,
      dispatch: (action) => dispatch(action)
    };
    chain = middlewares.map(middleware => middleware(middlewareAPI));
    dispatch = compose(...chain)(store.dispatch);
    return {...store, dispatch}
  }
}
```

所有中间件被放进了一个数组 `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` 进行一个判断，然后执行对应操作，原理如下：

```
function patchThunk(store) {
    let next = store.dispatch;
    function dispatchAndThunk(action) {
        if (typeof action === "function") {
            action(store.dispatch, store.getState);
        } else {
            next(action);
        }
    }
```

```
    store.dispatch = dispatchAndThunk;
}
```

实现一个日志输出的原理也非常简单，如下：

```
let next = store.dispatch;
function dispatchAndLog(action) {
  console.log("dispatching:", addAction(10));
  next(addAction(5));
  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？

图片 加载失败

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一个新的 `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 `Structural Sharing` （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：

图片 加载失败

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

内部提供了一套完整的 Persistent Data Structure，还有很多易用的数据类型，如 `Collection`、`List`、`Map`、`Set`、`Record`、`Seq`，其中：

- 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：

- fromJS()：将一个js数据转换为Immutable类型的数据

```
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

```
import { Map, is } from 'immutable'
const map1 = Map({ a: 1, b: 1, c: 1 })
const map2 = Map({ a: 1, b: 1, c: 1 })
map1 === map2   //false
Object.is(map1, map2) // false
is(map1, map2) // true
```

- get(key)：对数据或对象取值
- getIn([])：对嵌套对象或数组取值，传参为数组，表示位置

```
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：

```
import Immutable from 'immutable';
foo = Immutable.fromJS({a: {b: 1}});
bar = foo.setIn(['a', 'b'], 2);   // 使用 setIn 赋值
console.log(foo.getIn(['a', 'b']));  // 使用 getIn 取值，打印 1
console.log(foo === bar);  //  打印 false
```

如果换到原生的 `js` ，则对应如下：

```
let foo = {a: {b: 1}};
let bar = foo;
bar.a.b = 2;
console.log(foo.a.b);  // 打印 2
console.log(foo === bar);  //  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` 前修改一个数据需要做一个深拷贝

```
import '_' from 'lodash';
const Component = React.createClass({
  getInitialState() {
    return {
      data: { times: 0 }
    }
```

```
  },
  handleAdd() {
    let data = _.cloneDeep(this.state.data);
    data.times = data.times + 1;
    this.setState({ data: data });
  }
}
```

使用 Immutable 后：

```
getInitialState() {
  return {
    data: Map({ times: 0 })
  }
},
  handleAdd() {
    this.setState({ data: this.state.data.update('times', v => v + 1) });
    // 这时的 times 并不会改变
    console.log(this.state.data.get('times'));
  }
```

同理，在 `redux` 中也可以将数据进行 `fromJS` 处理

```
import * as constants from './constants'
import {fromJS} from 'immutable'
const defaultState = fromJS({ //将数据转化成immutable数据
    home:true,
    focused:false,
    mouseIn:false,
    list:[],
    page:1,
    totalPage:1
})
export default(state=defaultState,action)=>{
    switch(action.type){
        case constants.SEARCH_FOCUS:
            return state.set('focused',true) //更改immutable数据
        case constants.CHANGE_HOME_ACTIVE:
            return state.set('home',action.value)
        case constants.SEARCH_BLUR:
            return state.set('focused',false)
        case constants.CHANGE_LIST:
```

```
            // return
    state.set('list',action.data).set('totalPage',action.totalPage)
            //merge效率更高，执行一次改变多个数据
            return state.merge({
                list:action.data,
                totalPage:action.totalPage
            })
        case constants.MOUSE_ENTER:
            return state.set('mouseIn',true)
        case constants.MOUSE_LEAVE:
            return state.set('mouseIn',false)
        case constants.CHANGE_PAGE:
            return state.set('page',action.page)
        default:
            return state
    }
}
```

31. 说说React服务端渲染怎么做？原理是什么？

图片 加载失败

31.1. 是什么

`Server-Side Rendering` ，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

图片 加载失败

其解决的问题主要有两个：

- 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

首先通过 `express` 启动一个 `app.js` 文件，用于监听3000端口的请求，当请求根目录时，返回 `HTML` ，如下：

```
const express = require('express')
const app = express()
app.get('/', (req,res) => res.send(
<html>    <head>        <title>ssr demo</title>    </head>    <body>
Hello world    </body> </html>
))
app.listen(3000, () => console.log('Exampleapp listening on port 3000!'))
```

然后再服务器中编写 `react` 代码，在 `app.js` 中进行应引用

```
import React from 'react'
const Home = () =>{
    return <div>home</div>
}
export default Home
```

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 `webpack.server.js` 并进行相关配置，如下：

```
const path = require('path')    //node的path模块
const nodeExternals = require('webpack-node-externals')
module.exports = {
    target:'node',
    mode:'development',           //开发模式
    entry:'./app.js',             //入口
    output: {                     //打包出口
        filename:'bundle.js',     //打包后的文件名
```

```
        path:path.resolve(__dirname,'build')    //存放到根目录的build文件夹
    },
    externals: [nodeExternals()],  //保持node中require的引用方式
    module: {
        rules: [{                  //打包规则
           test:   /\.js?$/,       //对所有js文件进行打包
           loader:'babel-loader',  //使用babel-loader进行打包
           exclude: /node_modules/,//不打包node_modules中的js文件
           options: {
               presets: ['react','stage-0',['env', {
                                   //loader时额外的打包规则,对react,JSX, ES6进行转换
                    targets: {
                        browsers: ['last 2versions']   //对主流浏览器最近两个版本进行兼容
                    }
               }]]
           }
       }]
    }
}
```

接着借助 `react-dom` 提供了服务端渲染的 `renderToString` 方法，负责把 `React` 组件解析成 `html`

```
import express from 'express'
import React from 'react'//引入React以支持JSX的语法
import { renderToString } from 'react-dom/server'//引入renderToString方法
import Home from'./src/containers/Home'
const app= express()
const content = renderToString(<Home/>)
app.get('/',(req,res) => res.send(
<html>    <head>        <title>ssr demo</title>    </head>    <body>
${content}    </body> </html>
))
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 `script` 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 `js` 文件配置路由，修改如下：

```
import express from 'express'
import React from 'react'//引入React以支持JSX的语法
import { renderToString } from'react-dom/server'//引入renderToString方法
import Home from './src/containers/Home'
const app = express()
app.use(express.static('public'));
//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
 const content = renderToString(<Home/>)
app.get('/',(req,res)=>res.send(
<html>    <head>        <title>ssr demo</title>    </head>    <body>
${content}    <script src="/index.js"></script>    </body> </html>
))
 app.listen(3001, () =>console.log('Example app listening on port 3001!'))
```

然后再客户端执行以下 `react` 代码，新建 `webpack.client.js` 作为客户端React代码的 `webpack` 配置文件如下：

```
const path = require('path')                        //node的path模块
module.exports = {
    mode:'development',                             //开发模式
    entry:'./src/client/index.js',                  //入口
    output: {                                       //打包出口
        filename:'index.js',                        //打包后的文件名
        path:path.resolve(__dirname,'public')       //存放到根目录的build文件夹
    },
    module: {
        rules: [{                                   //打包规则
           test:   /\.js?$/,                        //对所有js文件进行打包
           loader:'babel-loader',                   //使用babel-loader进行打包
           exclude: /node_modules/,                 //不打包node_modules中的js文件
           options: {
               presets: ['react','stage-0',['env', {

                    //loader时额外的打包规则,这里对react,JSX进行转换
                    targets: {
```

```
                        browsers: ['last 2versions']   //对主流浏览器最近两个版本进行兼容
                    }
                }]]
            }
        }]
    }
}
```

这种方法就能够简单实现首页的 `react` 服务端渲染，过程对应如下图：

图片 加载失败

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：

```
import React from 'react'                    //引入React以支持JSX
import { Route } from 'react-router-dom'     //引入路由
import Home from './containers/Home'         //引入Home组件
export default (
    <div>
        <Route path="/" exact component={Home}></Route>
    </div>
)
```

然后可以通过 `index.js` 引用路由信息，如下：

```
import React from 'react'
import ReactDom from 'react-dom'
import { BrowserRouter } from'react-router-dom'
import Router from'../Routers'
const App= () => {
    return (
        <BrowserRouter>
           {Router}
        </BrowserRouter>
```

```
    )
}
ReactDom.hydrate(<App/>, document.getElementById('root'))
```

这时候控制台会存在报错信息，原因在于每个 `Route` 组件外面包裹着一层 `div` ，但服务端返回的代码中并没有这个 `div`

解决方法只需要将路由信息在服务端执行一遍，使用使用 `StaticRouter` 来替代 `BrowserRouter` ，通过 `context` 进行参数传递

```
import express from 'express'
import React from 'react'//引入React以支持JSX的语法
import { renderToString } from 'react-dom/server'//引入renderToString方法
import { StaticRouter } from 'react-router-dom'
import Router from '../Routers'
const app = express()
app.use(express.static('public'));
//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
app.get('/',(req,res)=>{
    const content  = renderToString((
        //传入当前path
        //context为必填参数,用于服务端渲染参数传递
        <StaticRouter location={req.path} context={{}}>
           {Router}
        </StaticRouter>
    ))
    res.send(
   <html>        <head>            <title>ssr demo</title>        </head>
    <body>        <div id="root">${content}</div>        <script src="/index.js">
</script>        </body>    </html>
)
})
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

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的 `html` 节点，整个流程结束