```jsx
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：

```javascript
// 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：

```jsx
<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：

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

类组件（有状态组件）如下：

JSX | 复制代码

```jsx
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：

JSX | 复制代码

```jsx
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

JavaScript | 复制代码

```javascript
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：

JSX | 复制代码

```jsx
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

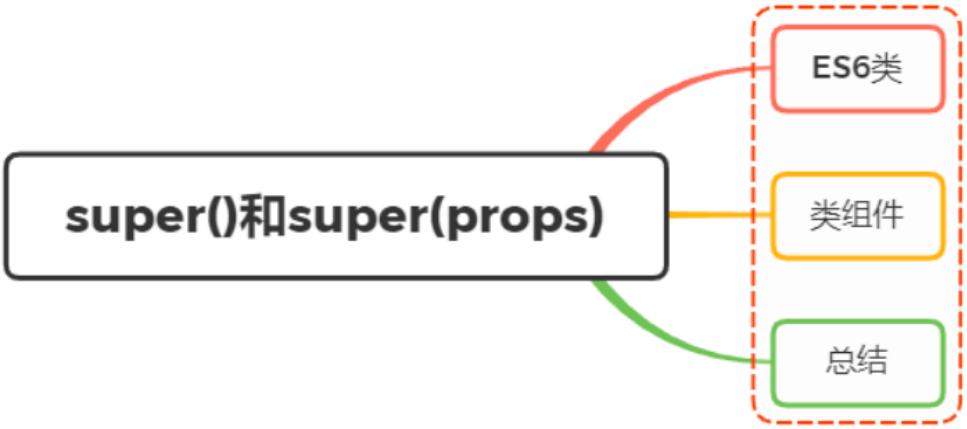
相同点：

- 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：

JavaScript | 复制代码

```
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

JavaScript 复制代码

```
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

JavaScript 复制代码

```
// 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：

```jsx
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：

```jsx
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：

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

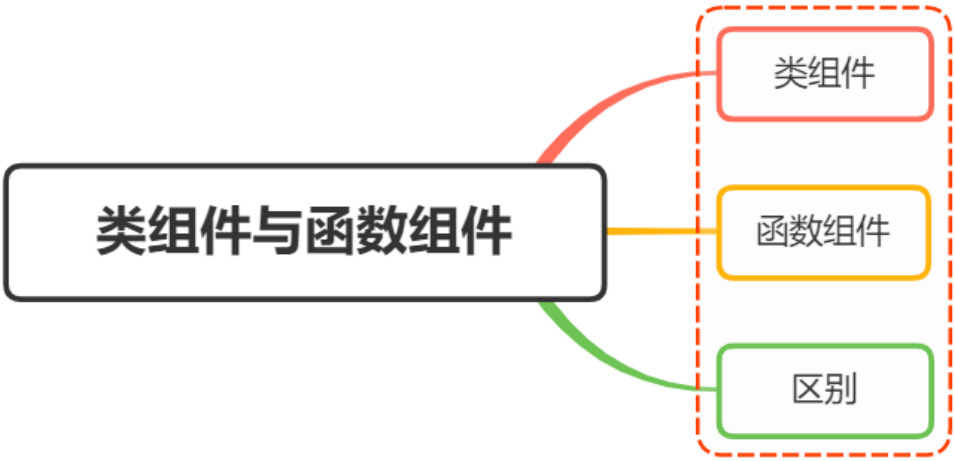
3.3. 总结

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：

JSX | 复制代码

```jsx
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

```jsx
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：

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

类组件：

```jsx
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：

```jsx
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：

```jsx
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`

```jsx
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：

```jsx
// 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：

```jsx
// 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：

```jsx
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：

```jsx
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

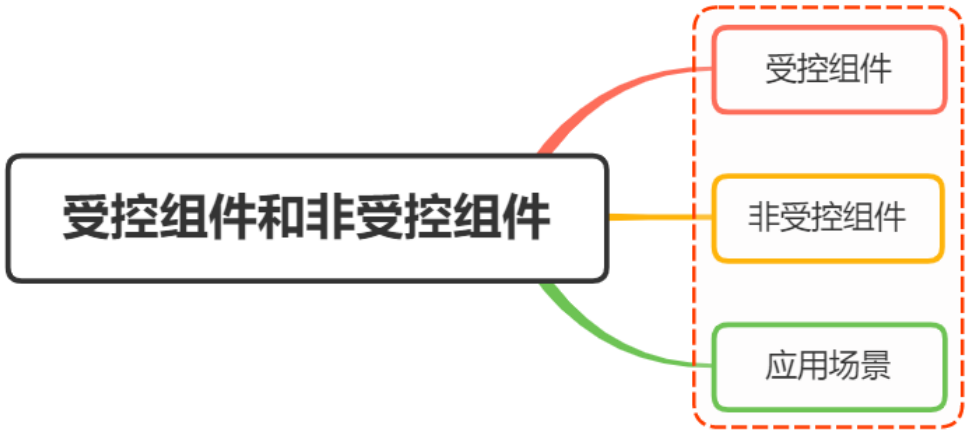
4.3.6. 小结

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：

```jsx
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：

JSX | 复制代码

```jsx
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

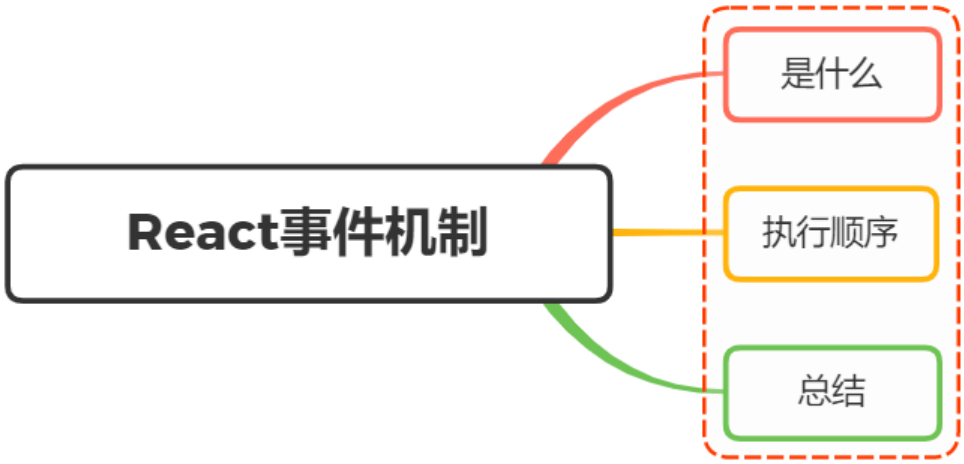
大部分时候推荐使用受控组件来实现表单，因为在受控组件中，表单数据由 `React` 组件负责处理

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：

JSX 复制代码

```jsx
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

JavaScript 复制代码

```javascript
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

JSX 复制代码

```jsx
// 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

JSX 复制代码

```jsx
// 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：

JSX | 复制代码

```jsx
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: