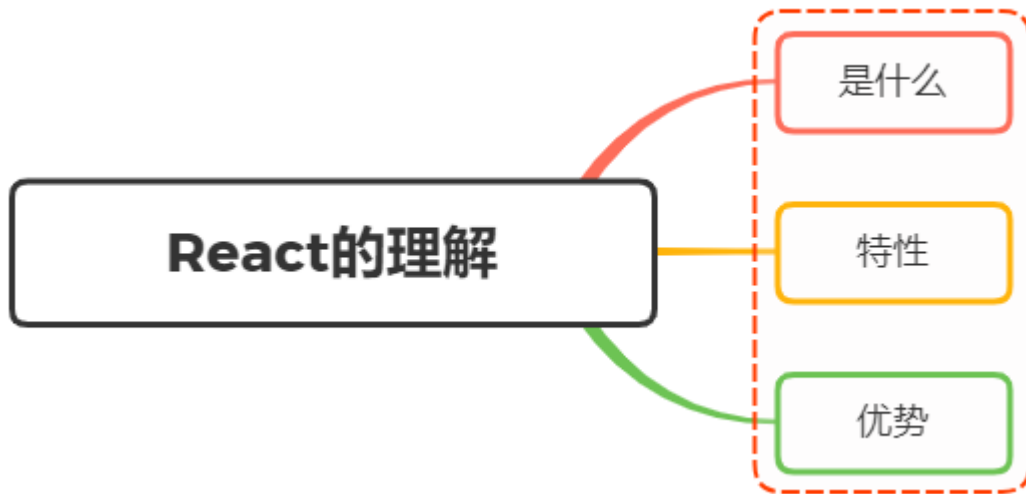


面试官：说说对React的理解？有哪些特性？



一、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

二、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

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

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：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

三、优势

通过上面的初步了解，可以感受到React存在的优势：

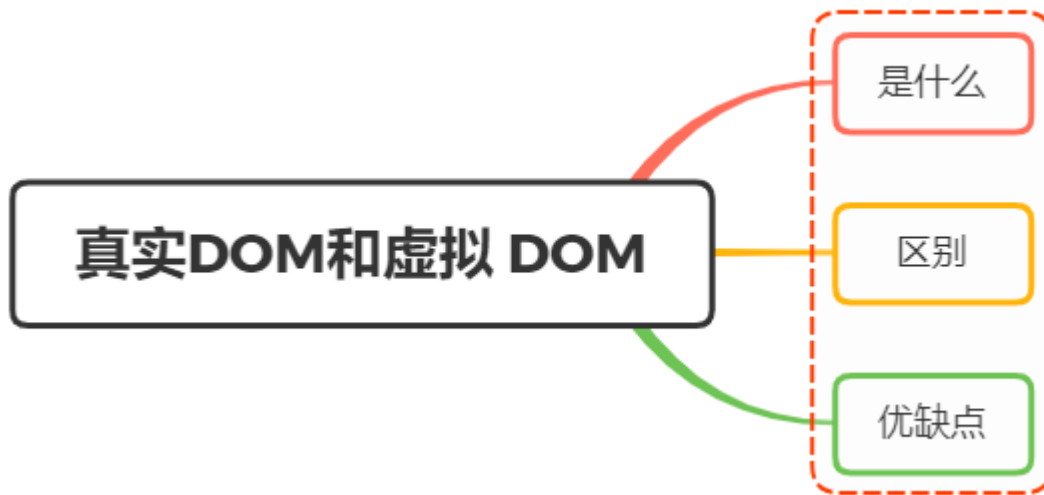
- 高效灵活
- 声明式的设计，简单使用
- 组件式开发，提高代码复用率
- 单向响应的数据流会比双向绑定的更安全，速度更快

参考文献

- <https://segmentfault.com/a/1190000015924762>

- <https://react.docschina.org/>

面试官：说说 Real DOM和 Virtual DOM 的区别？优缺点？



一、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

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

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

```
▼ Object 
  $$typeof: Symbol(react.element)
  key: null
  ▶ props: {children: "Hello World"}
  ref: null
  type: "h1"
  _owner: null
  ▶ _store: {validated: false}
  _self: undefined
  ▶ _source: {fileName: "E:\\Users\\u
  ▶ __proto__: Object
```

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

二、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”

拿[以前文章](#)举过的例子：

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

三、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

参考文献

- <https://juejin.cn/post/6844904052971536391>
- <https://www.html.cn/qa/other/22832.html>

面试官：说说 React 生命周期有哪些不同阶段？每个阶段对应的方法是？



一、是什么

在[以前文章](#)中，我们了解到生命周期定义

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

二、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

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

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

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

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

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

更新阶段

该阶段的函数主要为如下方法：

- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `getSnapshotBeforeUpdate`
- `componentDidUpdate`

getDerivedStateFromProps

该方法介绍同上

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

render

介绍如上

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

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

卸载阶段

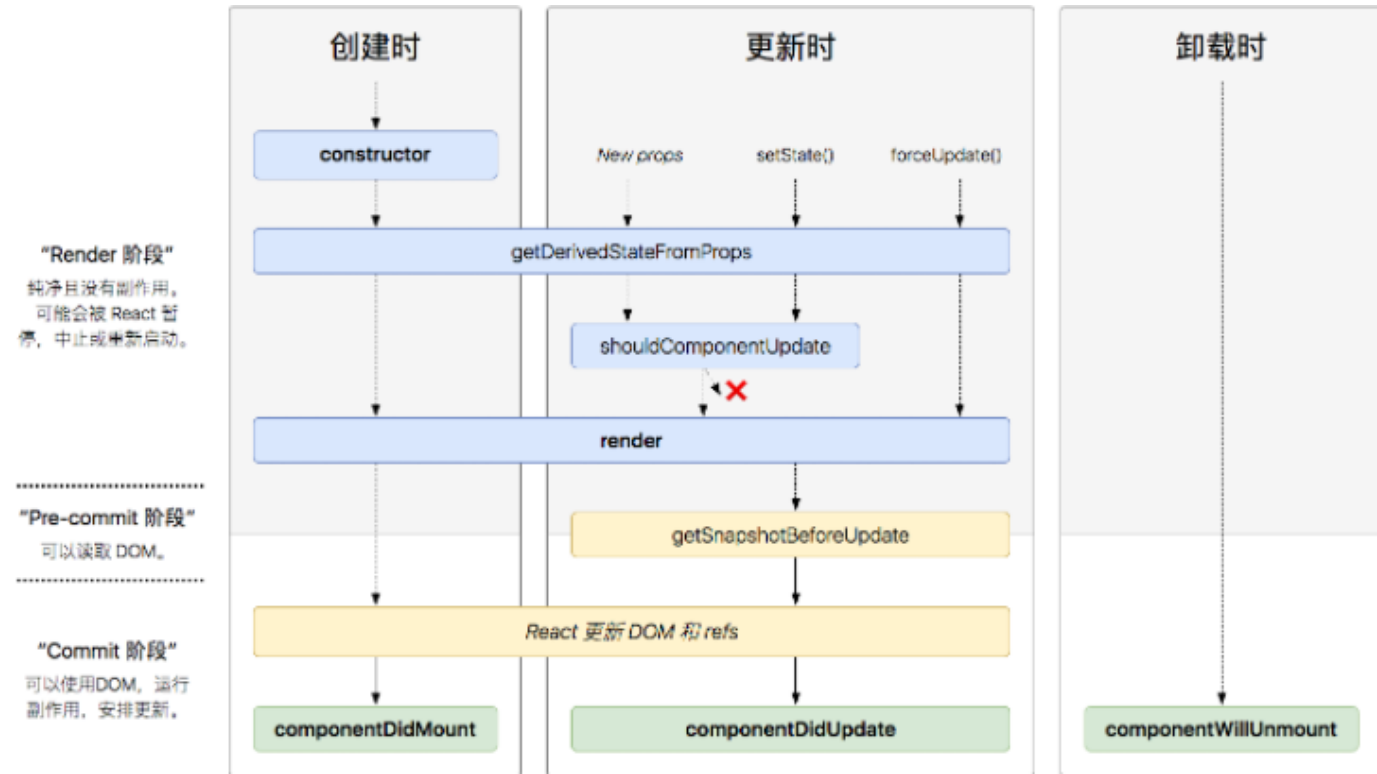
componentWillUnmount

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

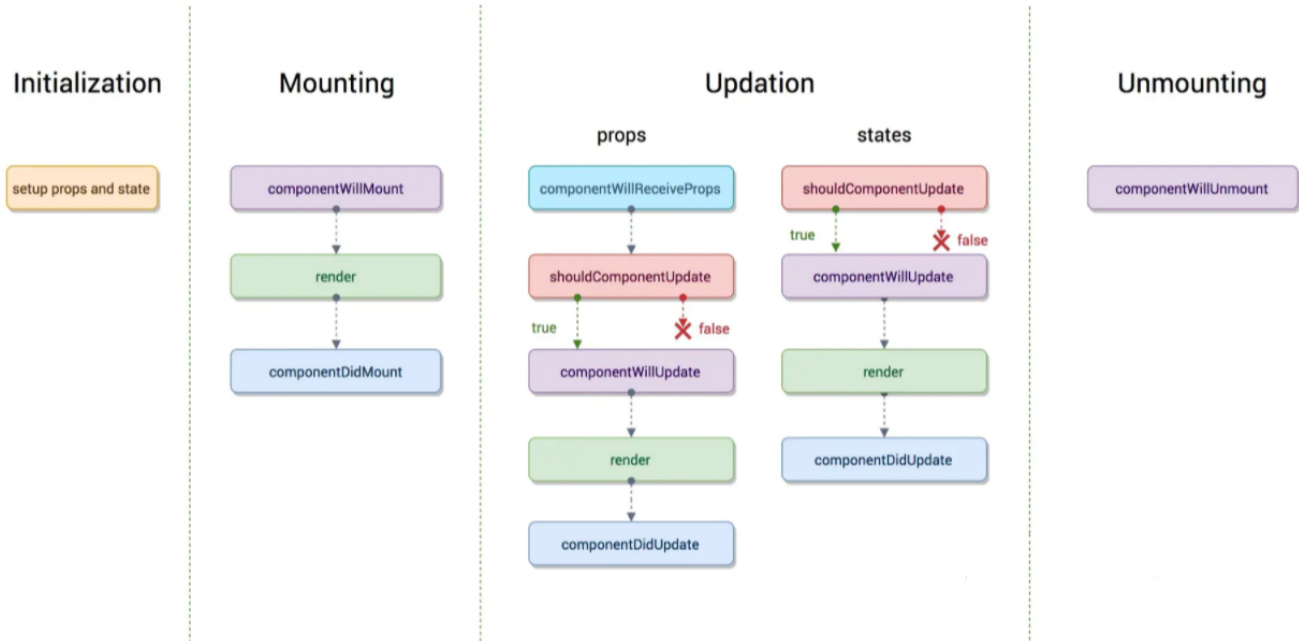
一旦一个组件实例被卸载，其不会被再次挂载，而只可能是被重新创建

三、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：

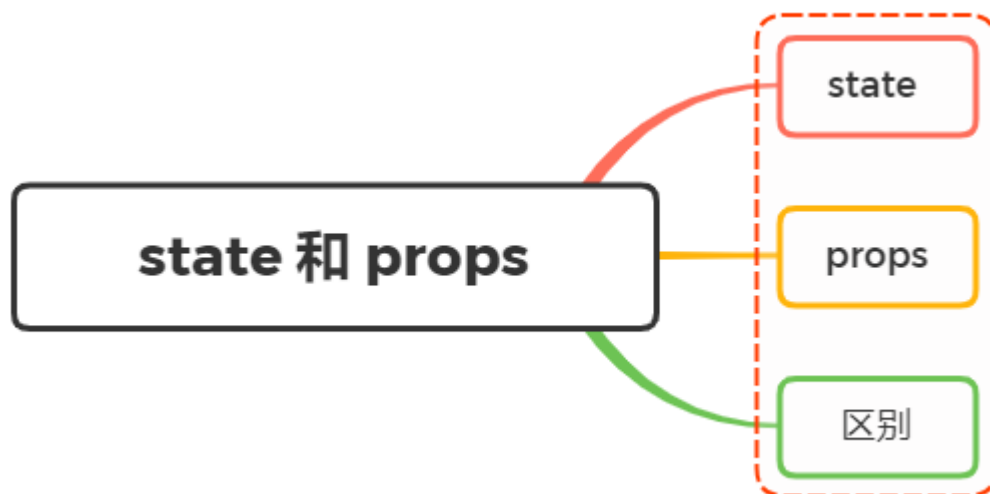
- getDerivedStateFromProps

- `getSnapshotBeforeUpdate`

参考文献

- <https://github.com/pomelovico/keep/issues/23>
- <https://segmentfault.com/a/1190000020268993>

面试官：state 和 props有什么区别？



一、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'))
```

二、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

三、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：

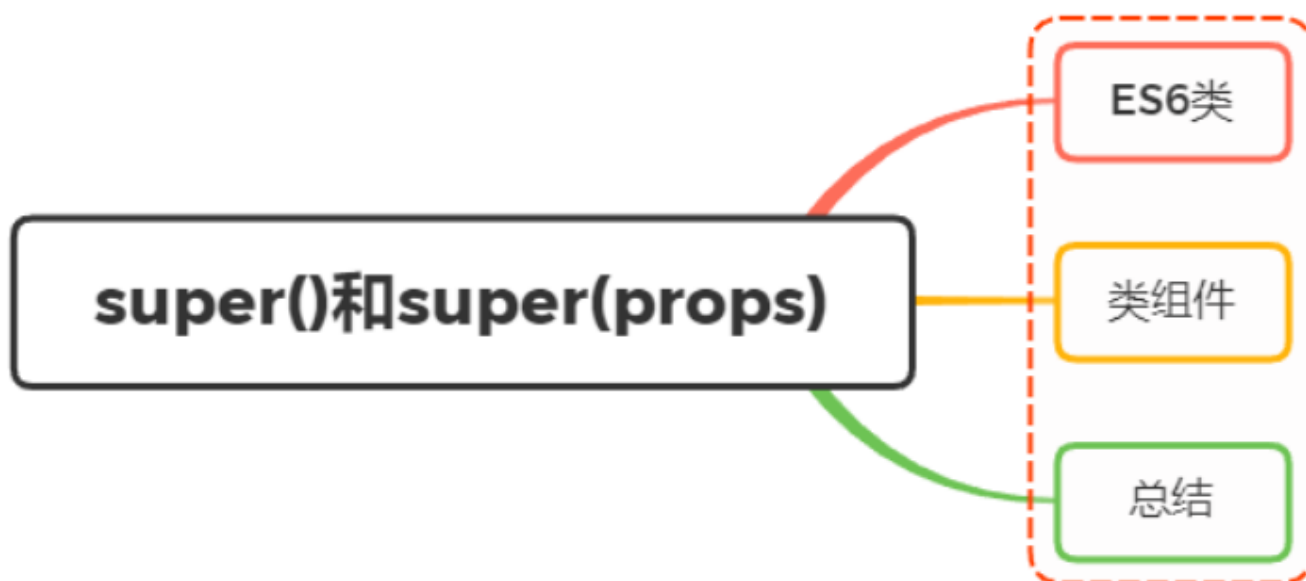
- `props` 是外部传递给组件的，而 `state` 是在组件内被组件自己管理的，一般在 `constructor` 中初始化
- `props` 在组件内部是不可修改的，但 `state` 在组件内部可以进行修改
- `state` 是多变的、可以修改

参考文献

- <https://lucybain.com/blog/2016/react-state-vs-pros/>

- <https://juejin.cn/post/6844904009203974158>

面试官：super()和super(props)有什么区别？



一、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`代替的是父类的构造函数，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`

二、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); // {}
    // ...
  }
}
```

三、总结

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

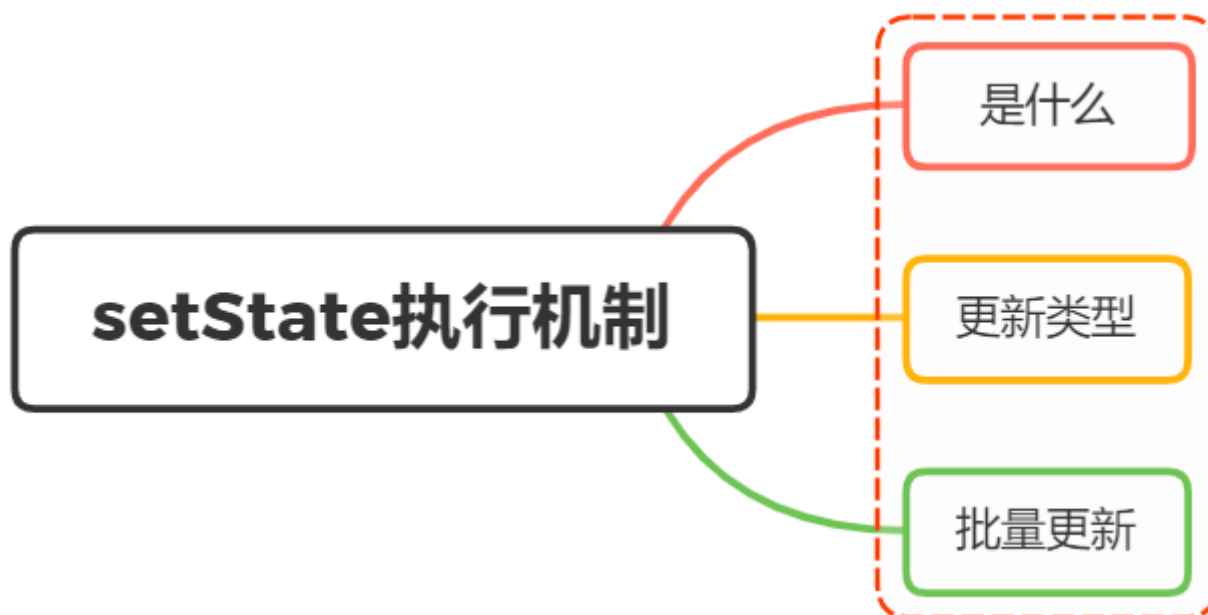
在调用`super`过程，无论是否传入`props`，`React`内部都会将`props`赋值给组件实例`props`属性中

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`

参考文献

- <https://overreacted.io/zh-hans/why-do-we-write-super-props/>
- <https://segmentfault.com/q/1010000008340434>

面试官：说说 React 中的 setState 执行机制



一、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

二、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

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  
  });  
}
```

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  
  })  
}
```

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

三、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) => {
```

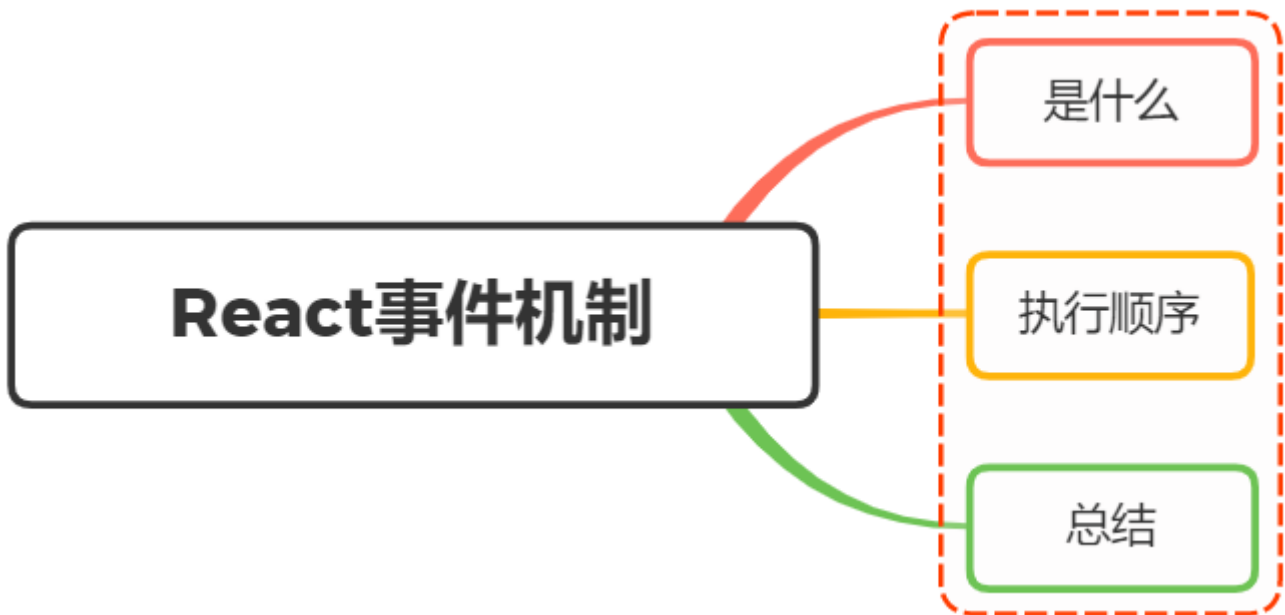
```
    return {count: prevState.count + 1};  
  });  
}
```

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

参考文献

- <https://juejin.cn/post/6844903667426918408>
- <https://juejin.cn/post/6844903636749778958>
- <https://segmentfault.com/a/1190000039077904>

面试官：说说React的事件机制？



一、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。

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

二、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, });  
})
```

三、总结

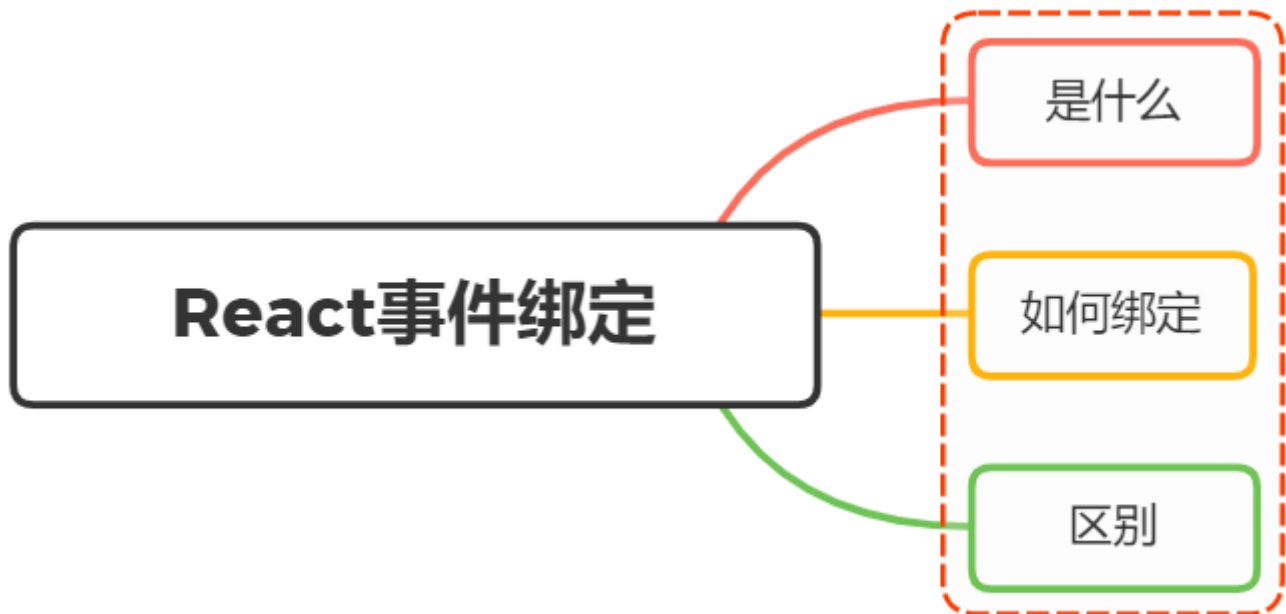
React事件机制总结如下：

- 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

参考文献

- <https://zh-hans.reactjs.org/docs/events.html>
- https://segmentfault.com/a/1190000015725214?utm_source=sf-similar-article
- <https://segmentfault.com/a/1190000038251163>

面试官：React事件绑定的方式有哪些？区别？



一、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`

二、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

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

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>
    )
  }
}
```

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>
    )
  }
}
```

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>
    )
  }
}
```

三、区别

上述四种方法的方式，区别主要如下：

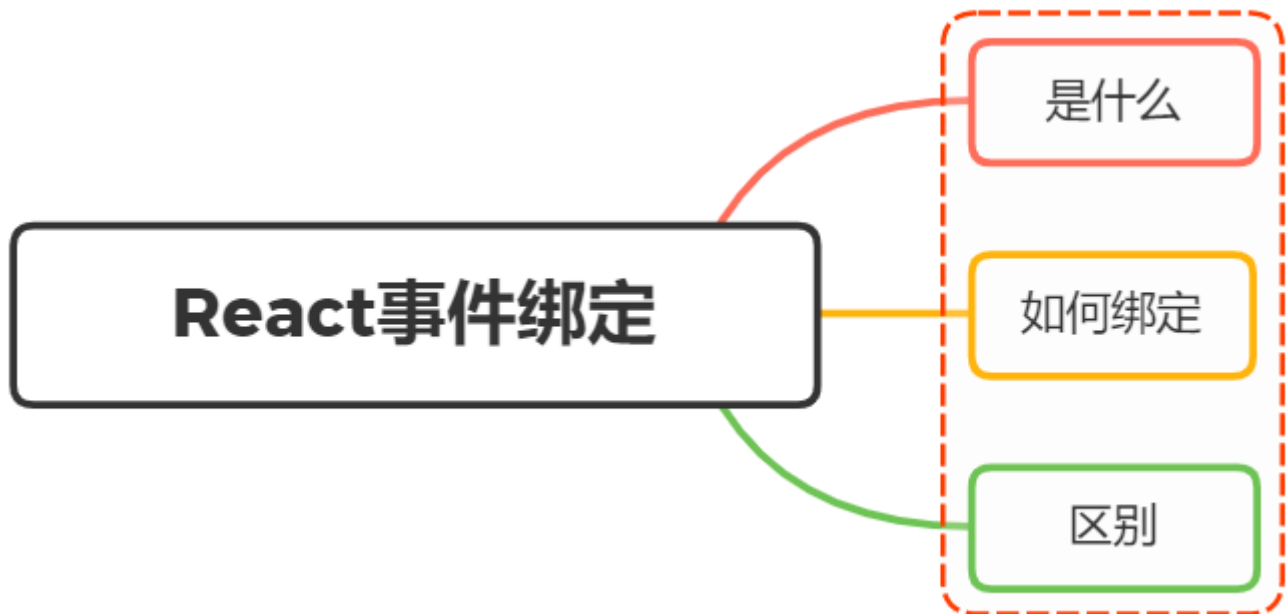
- 编写方面：方式一、方式二写法简单，方式三的编写过于冗杂
- 性能方面：方式一和方式二在每次组件`render`的时候都会生成新的方法实例，性能问题欠缺。若该函数作为属性值传给子组件的时候，都会导致额外的渲染。而方式三、方式四只会生成一个方法实例

综合上述，方式四是最优的事件绑定方式

参考文献

- <https://segmentfault.com/a/1190000011317515>
- <https://vue3js.cn/interview/>

面试官：React构建组件的方式有哪些？区别？



一、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

在[之前文章](#)中，我们了解到组件所存在的优势：

- 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

二、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

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>  
}
```

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

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，组件会再次调用`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>  
  );  
}
```

三、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：

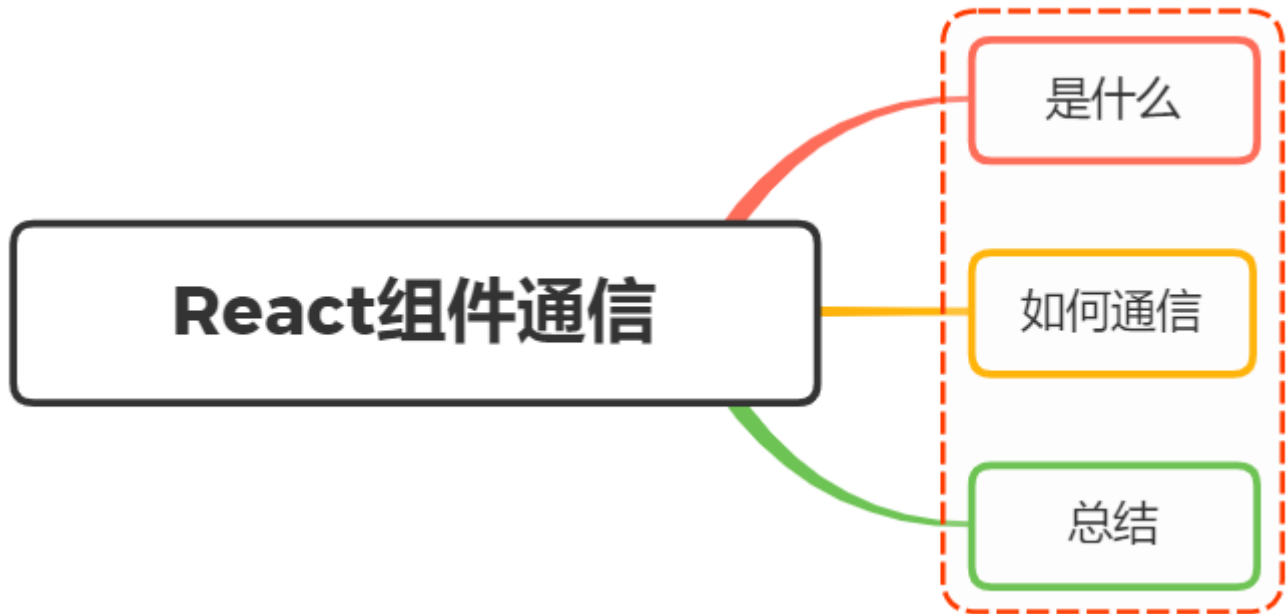
- 对于一些无状态的组件创建，建议使用函数式创建的方式
- 由于`react hooks`的出现，函数式组件创建的组件通过使用`hooks`方法也能使之成为有状态组件，再加上目前推崇函数式编程，所以这里建议都使用函数式的方式来创建组件

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

参考文献

- <https://react.docschina.org/>

面试官：React中组件之间如何通信？



一、是什么

我们将组件间通信可以拆分为两个词：

- 组件
- 通信

回顾[Vue系列](#)的文章，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受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

二、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

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" />;
```

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>
    );
  }
}
```

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>
    );
  }
}
```

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>
```

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

三、总结

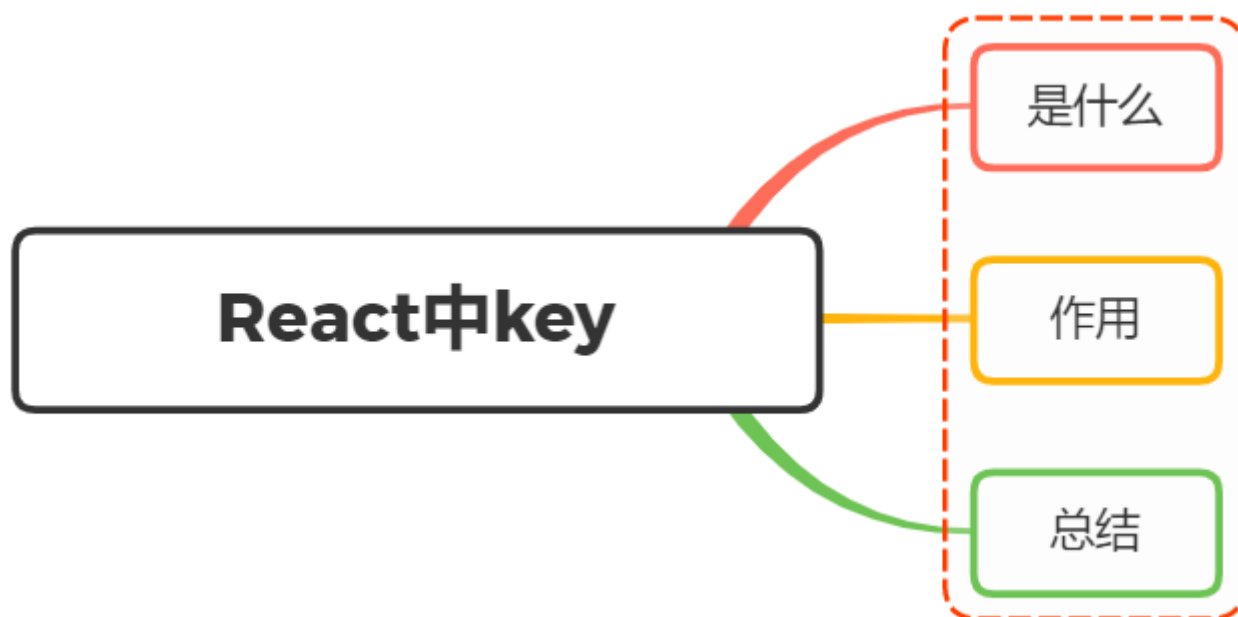
由于`React`是单向数据流，主要思想是组件不会改变接收的数据，只会监听数据的变化，当数据发生变化时它们会使用接收到的新值，而不是去修改已有的值

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

参考文献

- <https://react.docschina.org/docs/context.html>

面试官：React中的key有什么作用？



一、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>
  );
};
```

二、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.movies];  
  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

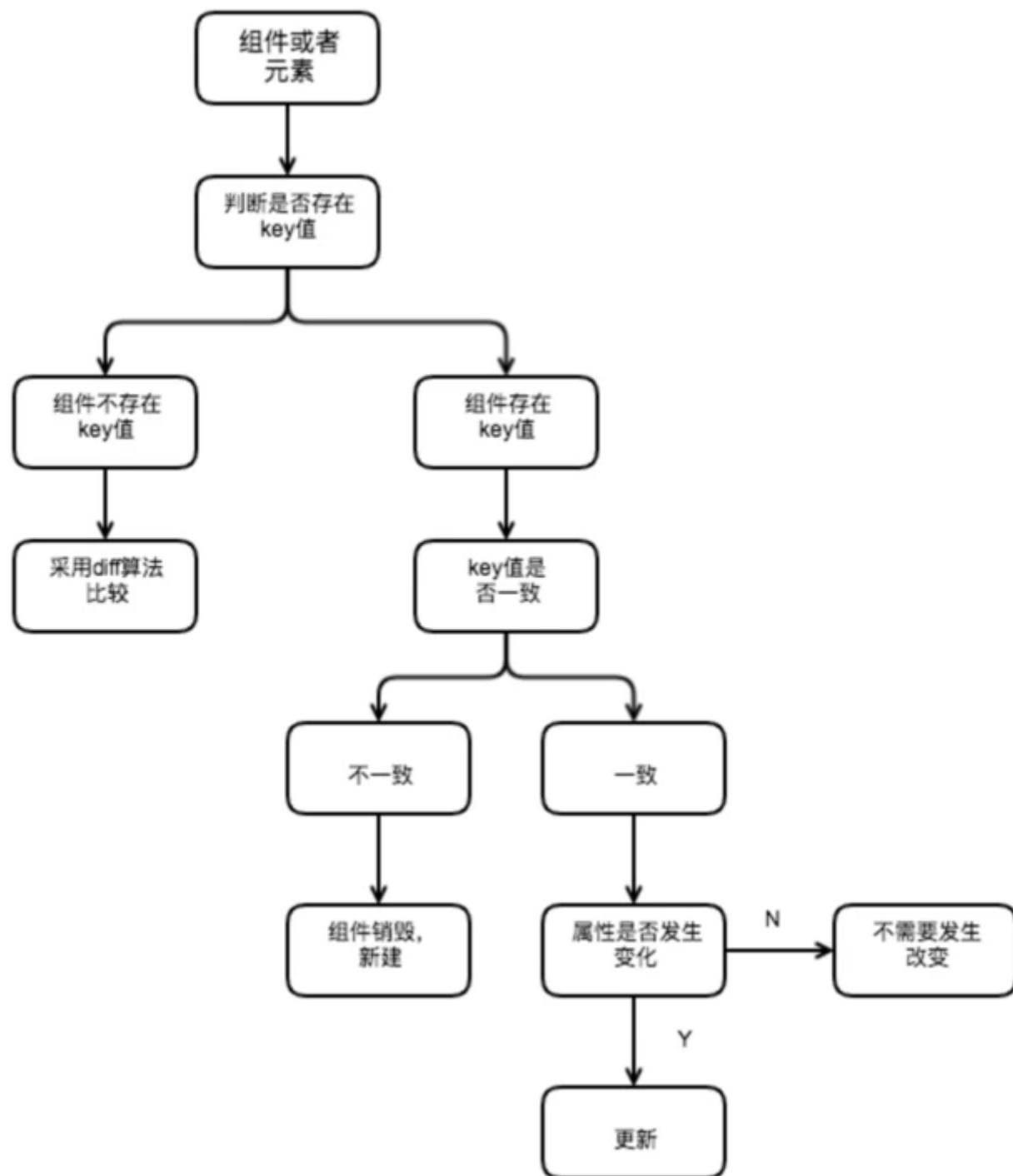
而写key则涉及到了节点的增和删，发现旧key不存在了，则将其删除，新key在之前没有，则插入，这就增加性能的开销

三、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：



参考文献

- <https://zh-hans.reactjs.org/docs/lists-and-keys.html#gatsby-focus-wrapper>
- <https://segmentfault.com/a/1190000017511836>