```
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

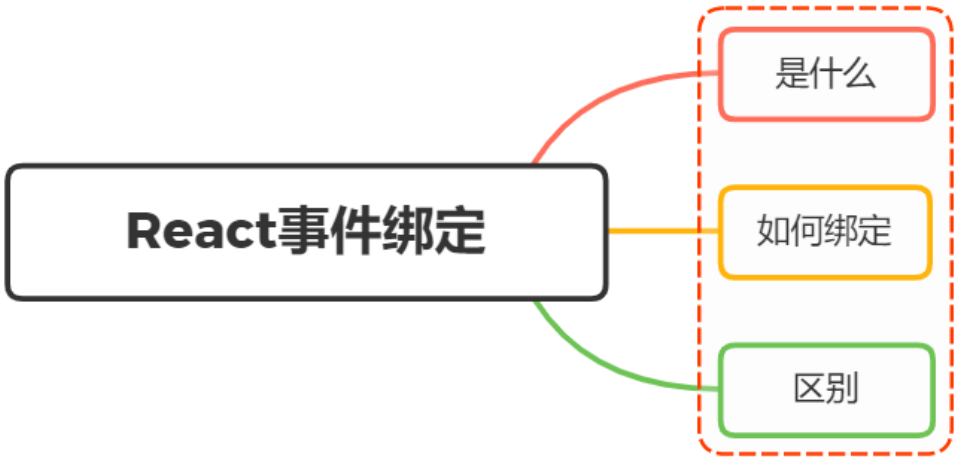
```javascript
document.body.addEventListener('click', e => {
    if (e.target && e.target.matches('div.code')) {
        return;
    }
    this.setState({   active: false,    });   });
}
```

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：

JSX | 复制代码

```jsx
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

JSX | 复制代码

```jsx
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

```jsx
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

```jsx
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：

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

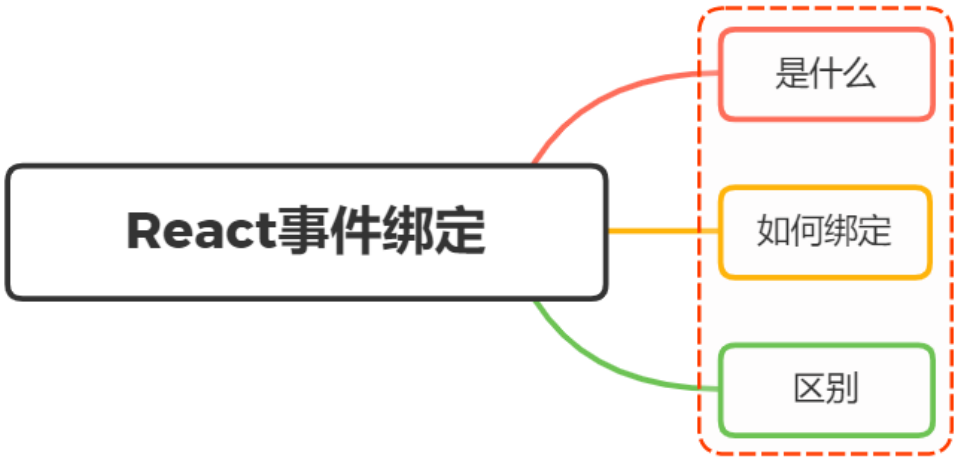
7.3. 区别

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：

JSX | 复制代码

```jsx
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：

JSX | 复制代码

```jsx
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：

```jsx
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

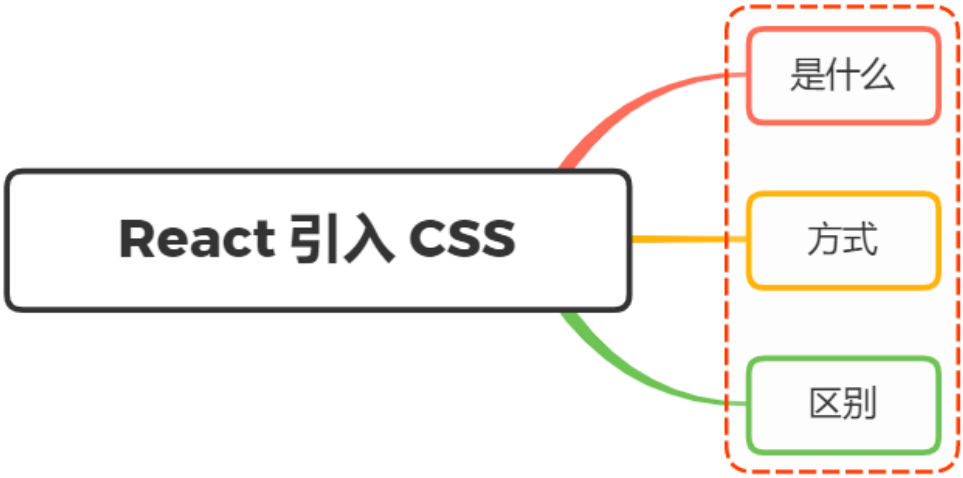
由于 `React.createClass` 创建的方式过于冗杂，并不建议使用

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：

JavaScript | 复制代码

```javascript
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：

CSS | 复制代码

```css
.title {
  color: red;
  font-size: 20px;
}

.desc {
  color: green;
  text-decoration: underline;
}
```

组件中引入：

JavaScript | 复制代码

```javascript
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

JSX | 复制代码

```jsx
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：

JavaScript 复制代码

```
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：

```jsx
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={fi
st}>
          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

JSX | 复制代码

```jsx
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

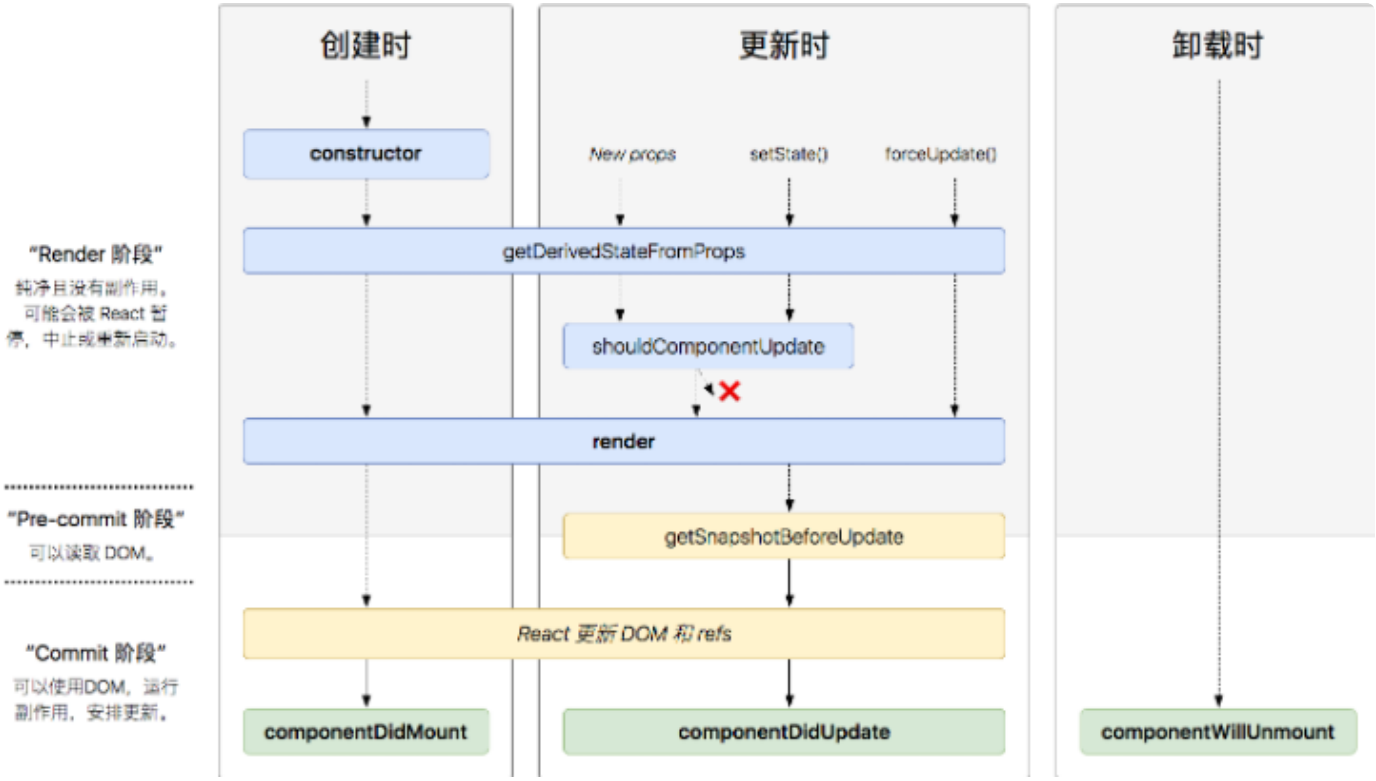
10.4. componentWillUnmount

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

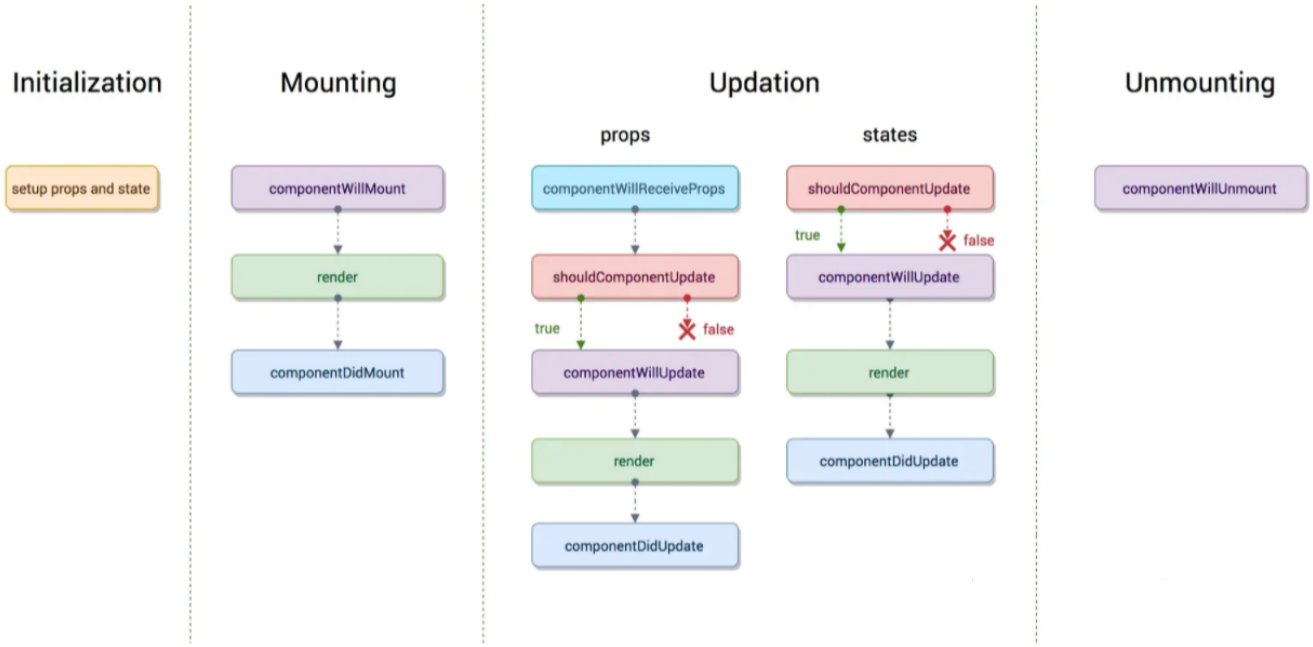
一旦一个组件实例被卸载，其不会被再次挂载，而只可能是被重新创建

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：

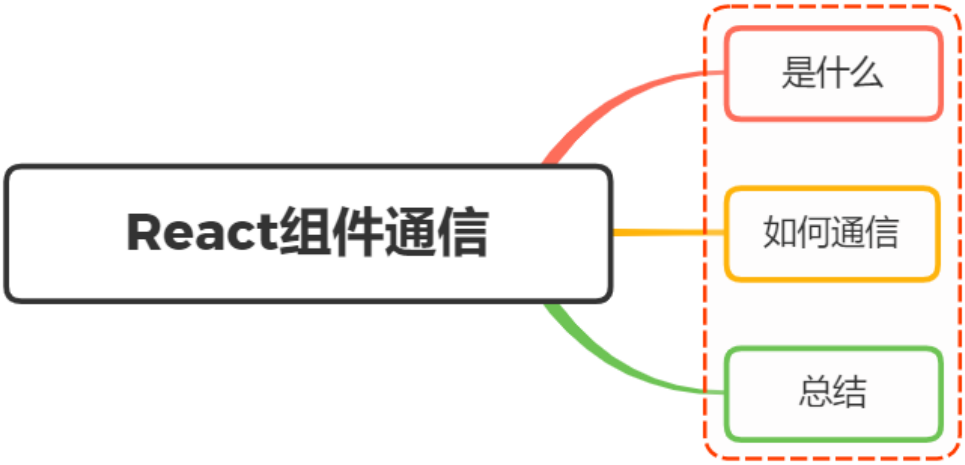
- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

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