组件通信&事件处理学习总结

Created and last modified by 王文磊 on Mar 08, 2022

一、组件通信

- 父组件 => 子组件：
  1. **Props**
  2. **Instance Methods**
- 子组件 => 父组件：
  1. **Callback Functions**
  2. **Event Bubbling**
- 兄弟组件之间：
  1. **Parent Component**
- 不太相关的组件之间：
  1. **Context**
  2. **Portals**
  3. **Global Variables**
  4. **Observer Pattern**
  5. **Redux等**

1. Props

最常见的react组件通信方式，父组件通过props把数据传给子组件，子组件通过 this.props (类组件)去使用相应的数据

```
const Child = ({ name }) => {
    <div>{name}</div>
}
class Parent extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            name: 'zach'
        }
    }
    render() {
        return (
            <Child name={this.state.name} />
        )
    }
}
```

2. Instance Method

父组件借助 ref 调用子组件实例方法（类组件），常见的使用场景是：子组件是一个modal弹窗组件，子组件有显示/隐藏这个modal弹窗的各种方法，我们就可以在父组件上通过ref绑定子组件实例来调用其方法控制弹窗显示/隐藏

```
class Modal extends React.Component {
  show = () => {// do something to show the modal}
  hide = () => {// do something to hide the modal}
  render() {
    return <div>I'm a modal</div>
  }
}
class Parent extends React.Component {
  componentDidMount() {
    if(// some condition) {
        this.modal.show()
    }
  }
  render() {
    return (
      <Modal
        ref={el => {
          this.modal = el
        }}
      />
    )
  }
}
```

3. Callback Functions

回调函数也是React中比较常见的一种通信方式，子组件通过props拿到父组件定义的callback，通过调用参数传递数据给父组件

```
const Child = ({ onClick }) => {
    <div onClick={() => onClick('zach')}>Click Me</div>
}
class Parent extends React.Component {
    handleClick = (data) => {
        console.log("Parent received value from child: " + data)
    }
    render() {
        return (
            <Child onClick={this.handleClick} />
        )
    }
}
```

4. Event Bubbling

与React无关，利用的是原生DOM的事件冒泡机制

```
class Parent extends React.Component {
  render() {
    return (
      <div onClick={this.handleClick}>
         <Child />
      </div>
    );
  }
  handleClick = () => {
    console.log('clicked')
  }
}
function Child {
  return (
    <button>Click</button>
  );
}
```

5. Parent Component

对于同级组件，若为兄弟组件，则通过父组件进行通信。若不是，考虑增加一层父组件包裹二者是否合适

```
class Parent extends React.Component {
  render() {
    return (
      <div onClick={this.handleClick}>
         <Child />
      </div>
    );
  }
  handleClick = () => {
    console.log('clicked')
  }
}
function Child {
  return (
    <button>Click</button>
  );
}
```

6. Context

React提供的全局Api，适用于组件层级深，且该数据需要被多个组件同时使用的情况

需要注意的是虽然Context使用方便，但不应该被滥用造成混乱，如果只是为了解决层级很深的 props 传递问题，可以直接用 comonent composition

类组件：

```
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

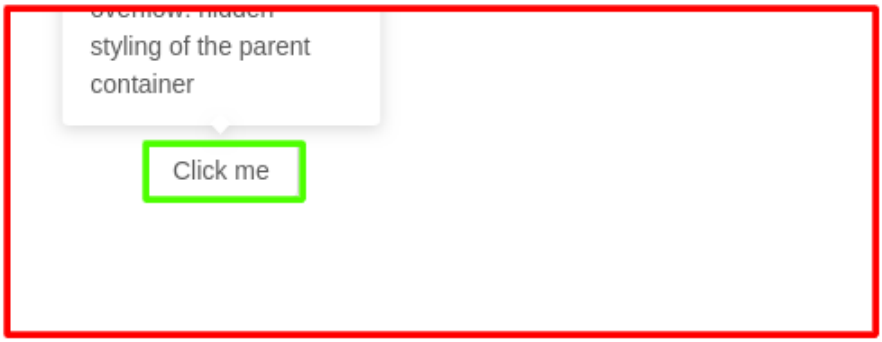
函数组件：

```
function ThemedButton {
    return (
        <ThemeContext.Consumer>
            {value => <Button theme={value} />}
        </ThemeContext.Consumer>
    )
}
```

7. Portals

React提供的新特性，其产生并非是为了解决通信问题，但也涉及到了组件通信，主要应用场景是：当两个组件在react项目中是父子组件的关系，但在HTML DOM里并不是父子元素的关系。

举个例子，有一个父组件Parent，它里面包含了一个子组件Tooltip，虽然在react层级上它们是父子关系，但我们希望子组件Tooltip渲染的元素在DOM中直接挂载在body节点里，而不是挂载在父组件的元素里。这样就可以避免父组件的一些样式（如`overflow:hidden`、`z-index`、`position`等）导致子组件无法渲染成我们想要的样式。



用portals解决上述问题

首先，修改html模版文件，给portals增加一个节点：

```
<html>
    <body>
        <div id="react-root"></div>
        <div id="portal-root"></div>
    </body>
</html>
```

然后我们创建一个可复用的portal容器：

```
import { useEffect } from "react";
import { createPortal } from "react-dom";
const Portal = ({children}) => {
  const mount = document.getElementById("portal-root");
  const el = document.createElement("div");
  useEffect(() => {
    mount.appendChild(el);
    return () => mount.removeChild(el);
  }, [el, mount]);
  return createPortal(children, el)
};
export default Portal;
```

最后在父组件中使用我们的portal容器组件，并将Tooltip作为children传给portal容器组件：

```
const Parent = () => {
  const [coords, setCoords] = useState({});
  return <div style={{overflow: "hidden"}}>
      <Button>
        Hover me
      </Button>
      <Portal>
        <Tooltip coords={coords}>
          Awesome content that is never cut off by its parent container!
         </Tooltip>
      </Portal>
  </div>
}
```

这样tooltip就不会受Parent组件的样式影响被截断了，同样适用的还有：Modal、Popup、Dropdown等等

8. Global Variables

使用window挂载全局变量，尽量避免使用：

```
class ComponentA extends React.Component {
    handleClick = () => window.a = 'test'
    ...
}
class ComponentB extends React.Component {
    render() {
        return <div>{window.a}</div>
    }
}
```

9. Observer Pattern（Event Bus）

观察者模式是一种常见的设计模式，核心是提供了一个订阅模型，subscriber（订阅者）与 publisher（发布者）直接关联，当 发布者 有 订阅者 subscribe 的事件发生时，订阅者会直接收到发布者的通知，典型的 addEventListener 就是一个观察者模式的实现。

当两个完全不相关（跨级）的组件间想要通信时，就可以利用这种模式，其中一个组件订阅某个消息，而另一个组件将其绑定到某个事件（用户行为、网络请求.....）上，由该组件负责发送。JS提供了现成的api来发送自定义事件：CustomEvent

组件A中，我们负责接收这个自定义事件：

```
 class ComponentA extends React.Component {
    componentDidMount() {
        document.addEventListener('myEvent', this.handleEvent)
    }
    componentWillUnmount() {
        document.removeEventListener('myEvent', this.handleEvent)
    }

    handleEvent = (e) => {
        console.log(e.detail.log)  //i'm zach
    }
}
```

组件B中，在某个事件发生时发送该自定义事件：

```
class ComponentB extends React.Component {
    sendEvent = () => {
        document.dispatchEvent(new CustomEvent('myEvent', {
          detail: {
             log: "i'm zach"
          }
        }))
    }

    render() {
        return <button onClick={this.sendEvent}>Send</button>
    }
}
```

上述事件都绑定在document上，容易导致冲突，用独立一个小模块EventBus的方式改造一下：

```
class EventBus {
    constructor() {
        this.bus = document.createElement('fakeelement');
    }
    addEventListener(event, callback) {
        this.bus.addEventListener(event, callback);
    }
    removeEventListener(event, callback) {
        this.bus.removeEventListener(event, callback);
    }
    dispatchEvent(event, detail = {}){
        this.bus.dispatchEvent(new CustomEvent(event, { detail }));
    }
}
export default new EventBus
```

使用：

```
import EventBus from './EventBus'
class ComponentA extends React.Component {
    componentDidMount() {
        EventBus.addEventListener('myEvent', this.handleEvent)
    }
    componentWillUnmount() {
        EventBus.removeEventListener('myEvent', this.handleEvent)
    }

    handleEvent = (e) => {
        console.log(e.detail.log)  //i'm zach
    }
}
class ComponentB extends React.Component {
    sendEvent = () => {
        EventBus.dispatchEvent('myEvent', {log: "i'm zach"}))
    }

    render() {
        return <button onClick={this.sendEvent}>Send</button>
    }
}
```

不使用 CustomEvent api，手动实现一个观察者模式：

```
function EventBus() {
  const subscriptions = {};
  this.subscribe = (eventType, callback) => {
    const id = Symbol('id');
    if (!subscriptions[eventType]) subscriptions[eventType] = {};
    subscriptions[eventType][id] = callback;
    return {
      unsubscribe: function unsubscribe() {
        delete subscriptions[eventType][id];
        if (Object.getOwnPropertySymbols(subscriptions[eventType]).length === 0) {
          delete subscriptions[eventType];
        }
      },
    };
  };
  this.publish = (eventType, arg) => {
    if (!subscriptions[eventType]) return;
    Object.getOwnPropertySymbols(subscriptions[eventType])
      .forEach(key => subscriptions[eventType][key](arg));
  };
}
export default EventBus;
```

10. Redux / MobX 等三方状态管理库

当项目体积较大时可以引入这种三方状态管理库，虽然 reducer 可以模块化，但 redux 维护的状态是全局唯一的，可以方便的实现很多通信。

而MobX项目中通常也都会有一个appStore，用来存储全局状态以供通信。

二、事件处理

1. React事件机制

```
<div onClick={this.handleClick.bind(this)}>click me</div>
```

React并不是将click事件绑定到了div的真实DOM上，而是在document处监听了所有的事件，当事件发生并且冒泡到document处的时候，React将事件内容封装并交由真正的处理函数运行。这样的方式不仅仅减少了内存的消耗，还能在组件挂在销毁时统一订阅和移除事件。

除此之外，冒泡到document上的事件也不是原生的浏览器事件，而是由react自己实现的合成事件（SyntheticEvent）。因此如果不想要是事件冒泡的话应该调用event.preventDefault()方法，而不是调用event.stopProppagation()方法。

 77fa6b2a59c92e160bc171f9c80783e7.jpg

实现合成事件的目的如下：

- **合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力；**
- **对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。**

2. React的事件与普通的HTML事件

区别：

- **对于事件名称命名方式，原生事件为全小写，react 事件采用小驼峰；**
- **对于事件函数处理语法，原生事件为字符串，react 事件为函数；**

- **react 事件不能采用 return false 的方式来阻止浏览器的默认行为，而必须要地明确地调用 `preventDefault()`来阻止默认行为。**

合成事件是 react 模拟原生 DOM 事件所有能力的一个事件对象，其优点如下：

- **兼容所有浏览器，更好的跨平台；**
- **将事件统一存放在一个数组，避免频繁的新增与删除（垃圾回收）。**
- **方便 react 统一管理和事务机制。**

事件的执行顺序为原生事件先执行，合成事件后执行，合成事件会冒泡绑定到 document 上，所以尽量避免原生事件与合成事件混用，如果原生事件阻止冒泡，可能会导致合成事件不执行，因为需要冒泡到document 上合成事件才会执行。

3. React 组件事件代理实现

React基于Virtual DOM实现了一个SyntheticEvent层（合成事件层），定义的事件处理器会接收到一个合成事件对象的实例，它符合W3C标准，且与原生的浏览器事件拥有同样的接口，支持冒泡机制，所有的事件都自动绑定在最外层上。

在React底层，主要对合成事件做了两件事：

- **事件委派： React会把所有的事件绑定到结构的最外层，使用统一的事件监听器，这个事件监听器上维持了一个映射来保存所有组件内部事件监听和处理函数。**
- **自动绑定： React组件中，每个方法的上下文都会指向该组件的实例，即自动绑定this为当前组件。**

编写时一些注意事项参考：https://zh-hans.reactjs.org/docs/handling-events.html

---

学习资料：

https://segmentfault.com/a/1190000023585646

https://zh-hans.reactjs.org/docs/handling-events.html

https://juejin.cn/post/6941546135827775525

Like Be the first to like this