React 打造命令式通用组件

zhiyizhang 2021年03月23日 12:56

> | 导语 React 的设计哲学里面有一条：一切皆组件。Web App 的组件全部用声明式的方式编写，这是一种很简洁也很优雅的开发方式。但是某些情况下（或者说某些特殊的组件），用声明式的方式反而会让代码复杂难看。这时，我们就需要用命令式组件解决这一问题。

声明式和命令式

各位前端同学对声明式组件调用和命令式组件调用应该是非常熟悉的。就算对这两个名词见的不多，日常工作中也用得不少。当然，虽然大家多半都用过，这里我还是多嘴解释一下。

简介

我们以 TDesign 为例，看看典型的声明式组件和命令式组件。

```
<Message theme="info">这是一个声明式组件</Message>
```

上面就是典型的声明式组件，一个简单的 React Node，用于在网页上弹窗。

```
Message.info({
  content: '这是一个命令式组件',
});
```

上面是一个命令式组件，以函数调用的形式在网页弹窗。

区别

在呈现给用户的视觉效果上，声明式与命令式组件区别不大（也可以说没区别）；但是从开发的角度看，两者有着显著的区别。

声明式组件必须一开始就写进 React Node 中，它的内容和显隐需要通过 state 中的状态控制；而命令式组件可以随用随调，不需要一开始就写好，与 React Node 上下文无关。

按照 React 的设计哲学，声明式组件确实优雅简洁。但是对于弹窗、提醒等特殊场景的组件，命令式组件有着显著的优势。

弹窗和提醒与用户操作、网络 IO 等行为高度相关，而这些行为往往是脱离网页内容上下文的，这些组件也不需要嵌入到 React Node 中。命令式组件随用随调的特性非常符合这种场景，而声明式组件写进 React Node 的方式显得有些格格不入。

此外，在一个页面中有若干个弹窗是很常见的实际业务场景。但是如果我们在 JSX 中写若干个声明式的弹窗，代码显然会变得极不优雅；而使用一个声明式弹窗并通过 state 动态改变内容虽然可以解决代码的优雅性问题，却又会提高维护成本。命令式组件就不会存在这种问题，需要的时候调用就好了。

实现命令式弹窗组件

下面我们来实现一个命令式的弹窗组件。当然，用相同的思路可以实现任何命令式组件，这里我们用实际应用上最有意义的弹窗作为范例。

背景和动机

项目背景是我自己的个人项目，一个博客（翻翻我的 KM 文章可以看到前文）。项目的技术选型是 React，UI 方面 使用了 material UI。动机非常简单，material UI 没有提供命令式的弹窗组件，而命令式弹窗组件的优点确实太香了。你问我为什么不用提供了命令式弹窗的 Ant Desgin？其一是因为 Ant Design 虽然有命令式弹窗，但是功能很单一、可配置项也不够完善，只能说是聊胜于无；其二是我觉得 Ant Design 太丑了……

需求

开发的命令式弹窗组件，我希望能满足这些需求：

- 能以命令式的语法控制其弹出（创建）
- 能支持多实例（也就是同时出现若干弹窗）
- 支持关闭弹窗，并且是移除 DOM 节点而不是通过显隐控制（避免过多废弃节点）
- 支持修改弹窗内容
- 弹窗内容支持字符串或是 JSX Element

思路和步骤

React 本身的设计逻辑要求，每个组件都是声明式的。我们要实现一个命令式组件，应该是用命令式的方法去创建一个声明式组件，并且创建和维护这个声明式组件的过程应该是无感知的。

能用命令的方式创建、并且支持多实例，第一个想到的，就是应该把它做成一个类。可以通过 new 的方式创建实例，每个实例（弹窗）互不干扰。

弹窗创建后，应该把弹窗挂载到网页上。因为弹窗在内容上与所处的页面上下文位置无关，所以可以直接挂载在 body 的最后。

综合以上考量，我们首先应该有一个弹窗组件，这个组件是普通的声明式组件；我们还需要一个包装类，这个包装类负责创建组件并把组件挂载到网页上，以及把实例和方法暴露给用户（此处的用户指的是开发者，也就是日后使用这个命令式组件的我自己）。

据此，我们可以写出这个类的构造函数：

```
constructor(props: DialogProps) {
  // 创建容器并把声明式的弹窗组件渲染上去
  const container = document.createElement('div');
  document.body.appendChild(container);
  ReactDOM.render(
    <DialogComponent {...props} />
    this.container,
  );
  return this;
}
```

这里的 props 是弹窗的参数（属性），其接口定义为：

```
interface DialogProps {
  title?: string; // 弹窗标题
  content: string | JSX.Element; //弹窗内容
  showConfirm?: boolean; // 是否显示确认按钮
  showCancel?: boolean; // 是否显示取消按钮
  onConfirm?: () => unknown; // 点击确认后执行的操作
  onCancel?: () => unknown; // 点击取消后执行的操作
  /* 可以根据需要自定义更多属性 */
```

```
  /* 可以根据需要自定义更多属性 */
}
```

构造函数中的 DialogComponent 是另写的声明式弹窗组件，这是真正渲染到页面上的弹窗，可以用类组件也可以用函数组件（本例中我使用函数组件）。构造函数最后返回弹窗类的实例，是需要让开发者能在 new 之后拿到这个弹窗对象。

这个 DialogComponent 的写法也与一般的声明式弹窗组件有所不同。因为我们需要让它既能从外部关闭（通过包装类以命令的方式关闭）也能从内部关闭（点击按钮关闭），所以我们可以用一个 state 来控制它的开闭，并用 forwarRef 和 useImperativeHandle 的 React Hook 来从外部控制这个 state。

所以，我们的 DialogComponent 大致长这样：

```
const DialogComponent = forwardRef<
  DialogMethods,
  DialogProps
>((props: DialogProps, ref) => {
  const [open, setOpen] = useState(true);

  // 可以从外部调用的命令式方法
  useImperativeHandle(ref, () => ({
    close: () => {
      setOpen(false);
    },
  }));

  const onClose = () => {
    setOpen(false);
  };

  return (
    // UI 库中的弹窗组件，通过 visible 控制显隐
    <DialogFromUILibrary {...props} visible={open} onClose={onClose} />
  );
});
```

DialogMethods 是 DialogComponent 可以被外部调用的方法的接口，这里我们需要给一个关闭方法的接口定义：

```
interface DialogMethods {
```

```
  close(): void;
}
```

现在我们回过头看看需求，我们要求弹窗能从外部被修改和关闭，所以除了被包装的组件需要有一个关闭接口之外，它的包装类的构造函数应该做出一点修改，包装类也需要加上对外暴露的关闭方法和修改方法。整个包装类应该是这样的：

```
class Dialog {
  // 弹窗组件挂载的容器
  container: HTMLDivElement;
  // 弹窗组件的 ref
  component: RefObject<DialogMethods>;

  constructor(props: DialogProps) {
    this.container = document.createElement('div');
    document.body.appendChild(this.container);
    this.component = React.createRef();
    ReactDOM.render(
      <DialogComponent ref={this.component} {...props} />,
      this.container,
    );
    return this;
  }

  change(props: DialogProps): void {
    ReactDOM.render(
       <DialogComponent {...props} />,
       this.container,
    );
  }

  close(): void {
    this.component?.current?.close();
  }
}
```

好了，到这里我们已经能够实现命令式的弹窗了。在需要弹窗的时候，只需要这样：

```
const dialog = new Dialog({
  title: '标题',
  content: '内容字符串',
});
```

如果需要修改它或是关闭它，只需要这样：

```
dialog.change({
  title: '修改后的标题',
  // 内容可以是 JSX Element
  content: <a href="www.example.com">链接</a>
});

dialog.close();
```

现在，这个命令式弹窗完成得差不多了；不过，我们还有一个小问题没有解决 —— 弹窗的关闭时并没有移除 DOM 元素，某些极端情况下（比如用户反复开启和关闭弹窗）会留下很多废弃的节点。所以我们还需要做最后的修改，让弹窗关闭同时移除所在的节点。

在包装类里，这个很好解决：

```
class Dialog {
  container: HTMLDivElement;

  component: RefObject<DialogMethods>;

  constructor(props: DialogProps) {
    this.container = document.createElement('div');
    document.body.appendChild(this.container);
    this.component = React.createRef();
    ReactDOM.render(
      <DialogComponent ref={this.component} {...props} />,
      this.container,
    );
    return this;
  }

  change(props: DialogProps): void {
    ReactDOM.render(
      <DialogComponent {...props} />,
      this.container,
    );
  }

  close(): void {
    this.component?.current?.close();
    // 移除节点
```

```
    ReactDOM.unmountComponentAtNode(this.container);
    document.body.removeChild(this.container);
  }
}
```

从弹窗组件内部关闭时移除所在节点稍麻烦一些，这意味着我们需要把它所在的节点作为参数传给它。所以我们需要修改传给它的参数，它的接口就不能直接用 DialogProps 了，但是可以从 DialogProps 继承：

```
interface DialogComponentProps extends DialogProps {
  container?: HTMLDivElement;
}
```

当然，包装类传给它的参数也就需要稍微调整一下：

```
class Dialog {
  container: HTMLDivElement;

  component: RefObject<DialogMethods>;

  constructor(props: DialogProps) {
    this.container = document.createElement('div');
    document.body.appendChild(this.container);
    this.component = React.createRef();
    // 把所在的节点加入参数中，传给组件
    const componentProps = {
      ...props,
      container: this.container,
    };
    ReactDOM.render(
      <DialogComponent ref={this.component} {...componentProps} />,
      this.container,
    );
    return this;
  }

  change(props: DialogProps): void {
  // 把所在的节点加入参数中，传给组件
    const componentProps = {
      ...props,
      container: this.container,
    };
```

```
    ReactDOM.render(
      <DialogComponent {...componentProps} />,
      this.container,
    );
  }

  close(): void {
    this.component?.current?.close();
    ReactDOM.unmountComponentAtNode(this.container);
    document.body.removeChild(this.container);
  }
}
```

弹窗组件的内部从参数获取到自己所在的节点，关闭时移除即可：

```
const DialogComponent = forwardRef<
  DialogMethods,
  DialogComponentProps
>((props: DialogComponentProps, ref) => {
  const [open, setOpen] = useState(true);

  useImperativeHandle(ref, () => ({
    // 这里不需要移除节点，因为包装类中调用之后移除了
    close: () => {
      setOpen(false);
    },
  }));

  const onClose = () => {
    setOpen(false);
    // 从 props 中获取所在节点并移除
    const { container } = props;
    ReactDOM.unmountComponentAtNode(container);
    document.body.removeChild(container);
  };

  return (
    <DialogFromUILibrary {...props} visible={open} onClose={onClose} />
  );
});
```

到这里，我们就完整实现了一个命令式的弹窗组件。

完整代码

最后附上完整代码：

```
interface DialogProps {
  title?: string;
  content: string | JSX.Element;
  showConfirm?: boolean;
  showCancel?: boolean;
  onConfirm?: () => unknown;作
  onCancel?: () => unknown;
  /* 可以根据需要自定义更多属性 */
}

interface DialogComponentProps extends DialogProps {
  container?: HTMLDivElement;
}

interface DialogMethods {
  close(): void;
}

const DialogComponent = forwardRef<
  DialogMethods,
  DialogComponentProps
>((props: DialogComponentProps, ref) => {
  const [open, setOpen] = useState(true);

  useImperativeHandle(ref, () => ({
    close: () => {
      setOpen(false);
    },
  }));

  const onClose = () => {
    setOpen(false);
    const { container } = props;
    ReactDOM.unmountComponentAtNode(container);
    document.body.removeChild(container);
  };

  return (
    <DialogFromUILibrary {...props} visible={open} onClose={onClose} />
  );
```

```
});

class Dialog {
  container: HTMLDivElement;

  component: RefObject<DialogMethods>;

  constructor(props: DialogProps) {
    this.container = document.createElement('div');
    document.body.appendChild(this.container);
    this.component = React.createRef();
    const componentProps = {
      ...props,
      container: this.container,
    };
    ReactDOM.render(
      <DialogComponent ref={this.component} {...componentProps} />,
      this.container,
    );
    return this;
  }

  change(props: DialogProps): void {
    const componentProps = {
      ...props,
      container: this.container,
    };
    ReactDOM.render(
      <DialogComponent {...componentProps} />,
      this.container,
    );
  }

  close(): void {
    this.component?.current?.close();
    ReactDOM.unmountComponentAtNode(this.container);
    document.body.removeChild(this.container);
  }
}
```