

加餐01 | 留言区心愿单：真·子组件以及jsx-runtime

2022-09-20 宋一玮 来自北京



天下无鱼

<https://shikey.com/>

《现代React Web开发实战》

[课程介绍 >](#)



讲述：宋一玮

时长 08:27 大小 7.73M



你好，我是宋一玮，欢迎回到 React 应用开发的学习。

这是专栏的第一次加餐。我和专栏编辑从已上线课程的留言区中，选择了一些具有代表性的问题。这次加餐我们先来讲讲“真·子组件”，以及 JSX 这一语法糖在 React 17 版本以后发生的变化。在心愿单里呼声同样比较高的，还有 Fiber 协调引擎，会放到下节加餐中。

好的，接下来开始我们的加餐内容。

真·子组件

在第 5 节课中，我曾提到过：

React 还流行过一波真·子组件（Sub-components）的设计模式，代表性的组件库有

🔗 [Semantic UI React](#)、🔗 [Recharts](#).....如果你感兴趣的话，在靠后面的课程中我会讲解一

下这种模式的具体实现。

在 React 领域，一般提到中文“子组件”，指的是 Child Component，用于描述在 React 运行时（Runtime）构建的组件树（元素树）中，组件与组件之间的父子关系。

而这里提到的 Sub-components，主要还是在**描述设计时**（Design-time）**组件与组件间的强包含关系**（Containment），而在运行时这些组件之间却不一定是父子关系。所以，把 Sub-components 直译成“子组件”就不太合适，我就改用了“真·子组件”这种中二的翻译，意在与 Child Component 区别开。事实上，“附属组件”、“次级组件”、“副组件”也都是可行的名字。

如果用真·子组件模式设计 KanbanColumn 组件，那么它的 title 属性可能是这样的：

 复制代码

```
1 <KanbanColumn className="column-todo">
2   <KanbanColumn.Title>
3     待处理<button onClick={handleAdd}
4       disabled={showAdd}>&#8853; 添加新卡片</button>
5   </KanbanColumn.Title>
6   {/* ...省略 */}
7 </KanbanColumn>
```

你也许会吐槽，这跟在 title={} 里直接写 JSX 区别不大啊。那我们再来看一个 props 比较复杂的组件：

 复制代码

```
1 <Dialog
2   modal
3   onClose={() => {}}
4   title="这是标题"
5   titleClass="dialog-title"
6   titleStyle={{ color: 'blue' }}
7   content="这是正文。"
8   contentClass="dialog-content"
9   contentStyle={{ color: 'red' }}
10  showConfirmButton={true}
11  confirmButtonText="确认"
12  onConfirmButtonClick={() => {}}
13  showCancelButton={false}
14  cancelButtonText=""
15  onCancelButtonClick={() => {}}
16  {/* ...还有很多props */}
17 />
```

也许这个组件的设计者对加入这么多 **props** 不以为然，但这个组件的使用者们，看着茫茫 **props** 会觉得无从下手。这种情况下，双方就组件接口设计会提出如下需求：

1. 组件的 **props** 需要更加结构化、语义化；
2. 降低组件 **props** 结构与组件内部实现的耦合。

这就轮到真·子组件上场了，通过简单的梳理，我们为 **Dialog** 设计了如下几个真·子组件：

复制代码

```
1 const Dialog = (props) => { /* 待实现 */ };
2 Dialog.Title = () => null;
3 Dialog.Content = () => null;
4 Dialog.Action = () => null;
```

期待的使用方式如下：

复制代码

```
1 <Dialog modal onClose={() => {}}>
2   <Dialog.Title className="dialog-title" style={{ color: 'blue' }}>
3     这是标题
4   </Dialog.Title>
5   <Dialog.Content>
6     <p>这是正文。</p>
7     <p>这是正文第二段。</p>
8   </Dialog.Content>
9   <Dialog.Action type="confirm" onClick={() => {}}>确认</Dialog.Action>
10  <Dialog.Action type="cancel" onClick={() => {}}>取消</Dialog.Action>
11 </Dialog>
```

这样设计对于 **Dialog** 组件的使用者来说，还是很好用的，但对于 **Dialog** 组件的开发者就有一定挑战了。

具体来说，在渲染时，这些真·子组件与其他自定义组件一样，会创建对应的 **React** 元素出来，但它们会导致元素树变得冗长。我们并不希望这样，而只想把它们当作是 **Dialog** 组件的一种扩展属性。这就需要在 **Dialog** 的 **children** 属性上做文章。

首先基于 `React.Children` API，定义两个工具函数 `findByType` 和 `findAllByType`，用于选取 `children` 中特定类型的 `React` 元素：

 复制代码

```
1 function findByType(children, type) {
2   return React.Children.toArray(children).find(c => c.type === type);
3 }
4
5 function findAllByType(children, type) {
6   return React.Children.toArray(children).filter(c => c.type === type);
7 }
```

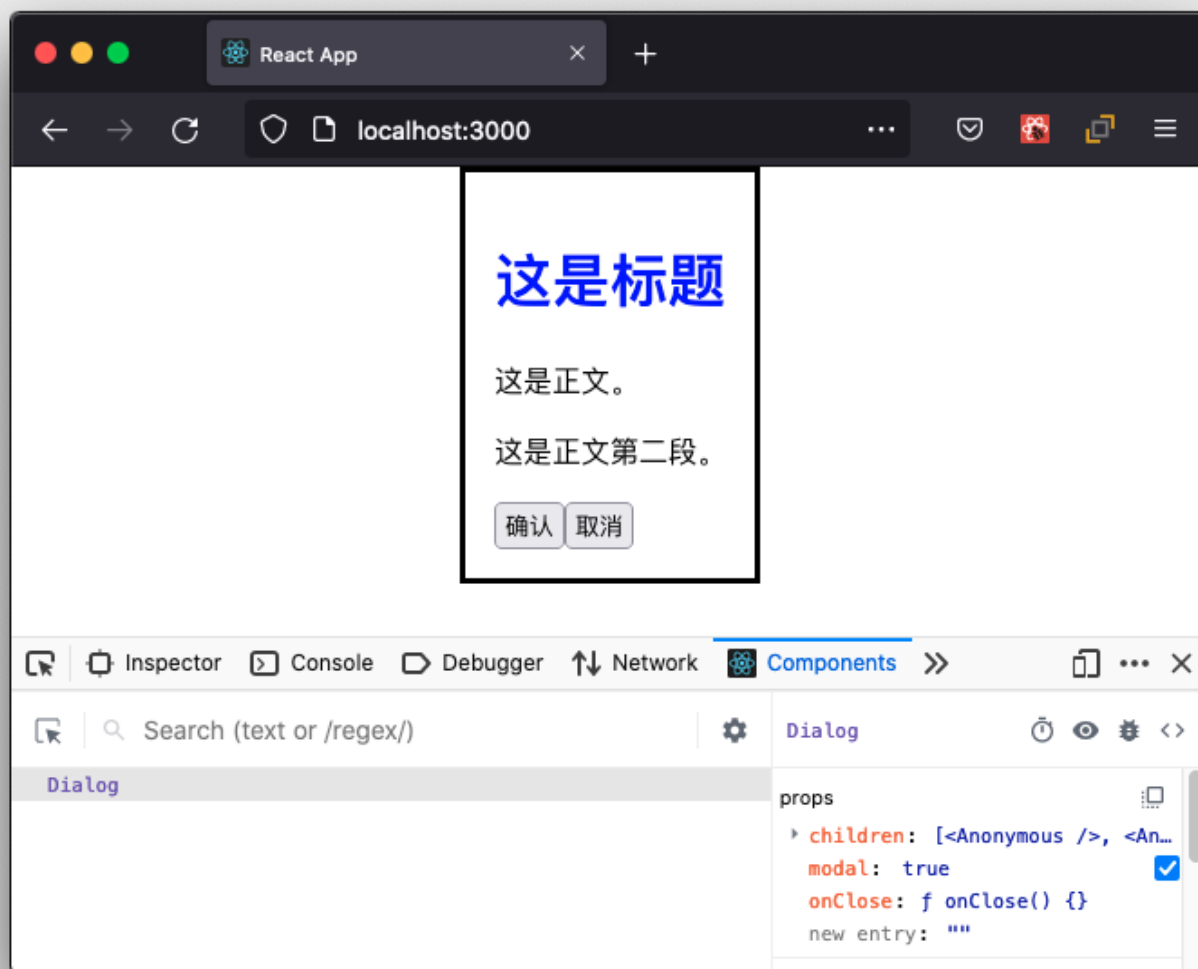
然后在 `Dialog` 组件函数体中，定义渲染标题、正文和动作按钮的函数，并在返回的 `JSX` 中调用它们：

 复制代码

```
1 const Dialog = ({ modal, onClose, children }) => {
2   const renderTitle = () => {
3     const subElement = findByType(children, Dialog.Title);
4     if (subElement) {
5       const { className, style, children } = subElement.props;
6       return (<h1 {...{ className, style }}>{children}</h1>);
7     }
8     return null;
9   };
10  const renderContent = () => {
11    const subElement = findByType(children, Dialog.Content);
12    return subElement?.props?.children;
13  };
14  const renderButtons = () => {
15    const subElements = findAllByType(children, Dialog.Action);
16    return subElements.map(({ props: { onClick, children } }) => (
17      <button onClick={onClick} key={children}>{children}</button>
18    ));
19  };
20  return (
21    <dialog open>
22      <header>{renderTitle()}</header>
23      <main>{renderContent()}</main>
24      <footer>{renderButtons()}</footer>
25    </dialog>
26  );
27 };
28 Dialog.Title = () => null;
29 Dialog.Content = () => null;
30 Dialog.Action = () => null;
```

可以看到，三个渲染函数行为都稍有不同，`renderTitle` 是从 `<Dialog.Title>` 中获取 `className`、`children` 等 props，然后用在 `<h1>` 上；`renderContent` 是直接返回 `<Dialog.Content>` 的 `children` 子元素；而 `renderButtons` 则是从多个 `<Dialog.Action>` 中获取多组 `onClick`、`children` 属性，然后分别渲染成 `<button>`。

在浏览器中可以观察到渲染结果：



还有一种情况，如果是用真·子组件定义类似模版的元素，在组件中有可能需要调用 `React.cloneElement` API 来克隆这个模版元素。

更详细的例子请参考 [Github 上 Semantic-UI-React 的 v3 版本](#)。之所以推荐 v3 版本，是因为这个版本大量使用了组件函数 + Hooks，而目前主干版本的 v2.x，主要还是基于类组件实现的。

除了真·子组件，你仍然有其他选择可以实现上述目标：

- 使用类似 JSON 这样的 DSL（Domain Specific Language）作为 props，让组件内部逻辑解析 DSL 来决定如何渲染；
- 组件的组合（Composition），这方面的知识和最佳实践，我们在后面第 18 节课代码复用会讲到。

React 17/18 中的 react/jsx-runtime

在第 4 节课我们提到过 JSX 是 `React.createElement` 的语法糖。如果你对 React 底层实现感兴趣，那你也需要了解这个语法糖在 React 新版中的变化：React 从 17 版本开始已经启用全新的 JSX 运行时来替代 `React.createElement`。这要感谢留言区“[Geek_fcdf7b](#)”同学的提醒。

在启用新 JSX 运行时的状态下，用代码编译器编译 JSX：

- 在生产模式下被编译成了 `react/jsx-runtime` 下的 `jsx` 或 `jsxjs`（目前同 `jsx`）；
- 在开发模式下 JSX 被编译成了 `react/jsx-dev-runtime` 下的 `jsxDEV`。

作为编译输入，JSX 的语法没有改变，编译输出无论是 `jsx-runtime` 还是 `React.createElement` 函数，它们的返回值也同样都是 React 元素。可见，代码编译器为开发者隐藏了新旧 API 的差异。这个变化并不影响已有的对 JSX 的理解。

另外，如果是开发者手工创建 React 元素，依旧应该调用 `React.createElement`。这个 API 并不会被移除。而 `jsx-runtime` 代码只应由编译器生成，开发者不应直接调用这个函数。

在 React 17 版本，新 JSX 运行时的具体更新日志可参考：<https://zh-hans.reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html>；

引入新 JSX 运行时的动机主要是因为原有的 `React.createElement` 是为了类组件设计的，而目前函数组件已然成为主流，老接口限制了进一步的优化，具体可以参考官方的征求意见贴：<https://github.com/reactjs/rfcs/pull/107>。里面提及 React v0.12 版本以来 JSX 的实现，在性能优化方面存在一些痛点，包括：

- 每次创建元素时都需要动态检查组件是否用 `.defaultProps` 定义了 `props` 默认值；
- 在 `React.lazy` 懒加载中，更是需要在渲染阶段解析 `props` 默认值；
- 子元素 `children` 需要被动态合并到 `props` 中，导致调用方无法更早获知元素 `props` 的完整结构；
- 从 JSX 编译出来的 `React.createElement` 是 `React` 对象的属性，而不是更容易优化的模块范围常量；
- 无法确定传入的 `props` 是否是一个用户创建的可变对象，所以每次都必须克隆对象；
- 必须从 `props` 中取出 `key` 和 `ref` ；
- 同样是 `key` 和 `ref` ，也有可能以属性展开的方式传进来，如 `<div {...props} />` ，我们需要动态检查其中是否有这两个属性；
- 要想让 JSX 编译出来的 `React.createElement` 生效，需要模块显式导入 `React`。

为了解决上面这些痛点，以及在远期能对 `React` 框架的部分概念做简化，`React` 官方将陆续引入三个步骤：

1. 新的 JSX 编译目标；
2. 对部分功能标注即将弃用；
3. 语义层面的破坏性更新。

`React 17` 版本加入的新 JSX 运行时就是这第一步。与 `React.createElement` 相比的变化包括：

- 自动导入；
- 在 `props` 之外传递 `key` 属性；
- 将 `children` 直接作为 `props` 的一部分；
- 分离生产模式和开发模式的 JSX 运行时。

我在 `oh-my-kanban` 项目里验证了一下，确实。

 复制代码

```
1 var KanbanCard = function KanbanCard(_ref) {
```




```
2   var title = _ref.title,
3     status = _ref.status;
4   return /*#__PURE__*/ (0, jsx_runtime.jsx)("li", {
5     className: "kanban-card",
6     children: [
7       /*#__PURE__*/ (0, jsx_runtime.jsx)("div", {
8         className: "card-title",
9         children: title,
10      }),
11      /*#__PURE__*/ (0, jsx_runtime.jsx)("div", {
12        className: "card-status",
13        children: status,
14      }),
15    ],
16  });
17 };
```

从编译结果看，与 `React.createElement` 在 `children` 的处理上是不同，`jsx_runtime.jsx` 的 `children` 直接就是 `props` 的一部分。

在本专栏选用的 `React 18.2.0` 版本和与它配套的 `CRA` 中，新 `JSX` 运行时也是被默认启用的。

好了，这节加餐就到这里。如果你还有其他想听的话题，或者在课程学习中有什么疑问，欢迎在留言区告诉我。下节课再见！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 13 | 组件表与里（下）：用接口的思路设计开发React组件

[下一篇](#) 加餐02 | 留言区心愿单：Fiber协调引擎



joel

2022-09-23 来自广东

老师你好，我希望以下心愿单：

- 1、react 更新机制原理等比较进阶的东西
- 2、react 自定义hooks 比较经典的案例场景，以及hooks 实现原理
- 3、对比vue 的原理机制，比如 vue 没有fiber, react 的设计的原理貌似跟vue 不一样，虽然都是有xxx等特点



风太太太太

2022-09-20 来自湖北

想听听react的高阶级用法，例如使用高阶组件。

怎么利用react-hooks 进行项目工程化改造，

怎么自己封装合理且好用的自定义hooks

