

05 | 前端组件化：将完整应用拆分成React组件

2022-09-01 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 20:55 大小 19.11M



你好，我是宋一玮。

上节课我们从相当于 React 门面的 JSX 语法入手，了解了 JSX 是 React 核心 API 之一 `React.createElement()` 的语法糖，是一种声明式的前端模版技术，然后深入学习了 JSX 的写法，也捎带提了一下 JSX 与 React 组件的关系。

那么这节课我们就来进一步讲讲 React 组件。

组件化开发已经成为前端开发的主流趋势，市面上大部分前端框架都包含组件概念，有些框架里叫 Component，有些叫 Widget。**React 更是把组件作为前端应用的核心。**

不过无论是哪种框架，几乎每一位学习前端组件的开发者都会遇到下面这些问题：

- 开发应用时是不是一定要拆分组件？一个应用我只用一个组件开发行不行？

- 如果一定要拆分组件，面对需求文档我该怎么下手？
- 组件拆分的粒度是应该大些还是小些？有没有可以参照的标准？

其实**组件拆分并无唯一标准**。拆分时需要你理解业务和交互，设计组件层次结构（Hierarchy），以关注点分离（Separation Of Concern）原则检验每次拆分。另外也要避免一个误区：组件确实是代码复用的手段之一，但并不是每个组件都需要复用。

这节课我们就从实践入手，学习如何拆分 React 组件，同时也介绍一些最佳实践。相信这节课结束时，你对上面的问题已经有自己的答案了。

为什么要组件化？

在前端领域，**组件是对视图以及与视图相关的逻辑、数据、交互等的封装**。如果没有组件这层封装，这些代码将有可能四散在各个地方，低内聚，也不一定能低耦合，这种代码往往难写、难读、难维护、难扩展。

类似下面这样的 HTML 表单代码，常见于前几年所见即所得的网站制作工具：

 复制代码

```
1 <div id="_panel1">
2   <form id="_form1"><input name="__text1" value="" /></form>
3 </div>
4 <div id="_panel2">
5   <img id="_img1" src="" />
6 </div>
7 <div id="_panel3">
8   <form id="_form2"><input name="__check1" type="checkbox" value="checked" /></form>
9 </div>
10 <div id="_panel4">
11   <script type="text/javascript">
12     function _form3_submit(event) {
13       var data = {};
14       data.text1 = document.getElementById('_form1').elements['__text1'].value;
15       data.check1 = document.getElementById('_check1').elements['__check1'].value;
16       ajaxPost(_handler_url, data);
17     }
18   </script>
19   <form id="_form3" onsubmit="_form3_submit">
20     <input name="__submit1" type="submit" />
21     <input name="__reset1" type="reset" />
22   </form>
23 </div>
```

上面的代码貌似工整，但实则杂乱无章。工具生成代码时一定有它自己的模型，但很明显，这个模型不是面向开发者的。你若是接手了这样的代码，一定会欲哭无泪，还不如重写一遍。

也许在不远的将来，AI 会代替我们开发前端应用，但现阶段，既然是由前端开发者编写代码，那么**前端技术就有必要辅助开发者写出更好的代码**。**低耦合高内聚**的封装已经被证明是更加有效的软件工程实践，那么组件化，就让前端开发走在了正确的道路上。

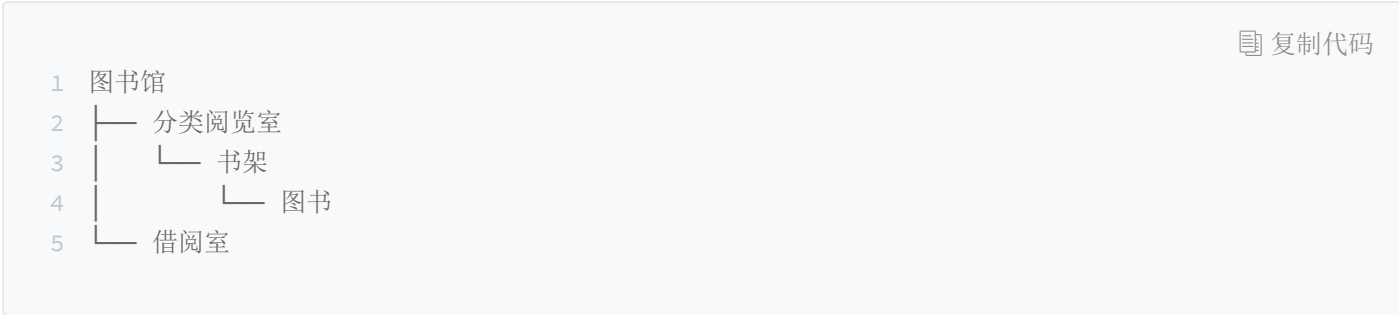
为什么要有组件层次结构？

比组件化更进一步的概念是**组件层次结构**（Hierarchy）。在面向对象编程里也有 Hierarchy 这个概念，一般是指父类子类之间的继承关系。

React 并没有用类继承的方式扩展现有组件（类组件继承 `React.Component` 类，但类组件之间没有继承关系），所以在 React 中提到 Hierarchy，一般都是指组件与组件间的层次结构。

组件层次结构可以帮助我们在设计开发组件过程中，**将前端应用需要承担的业务和技术复杂度分摊到多个组件中去，并把这些组件拼装在一起**。

React 组件层次结构从一个根部组件开始，一层层加入子组件，最终形成一棵组件树。假设我们有一个图书馆组件，那它对应的组件树可能是这样：



（你有多久没去过图书馆了？）

在这个例子中，图书馆组件是分类阅览室和借阅室的组合，而分类阅览室里面的陈列基本单元是书架，书架里才是图书（组件）。在开发图书馆组件时，不需要考虑图书组件；在开发书架组件时，也不需要考虑分类阅览室，更不需要考虑借阅室。这正符合我们常提到的关注点分离（Separation Of Concern）原则。

可以想象得到，这些组件最终能显示到浏览器里，肯定要在 `render()` 方法中加入不少 HTML 元素。

拆分 React 组件

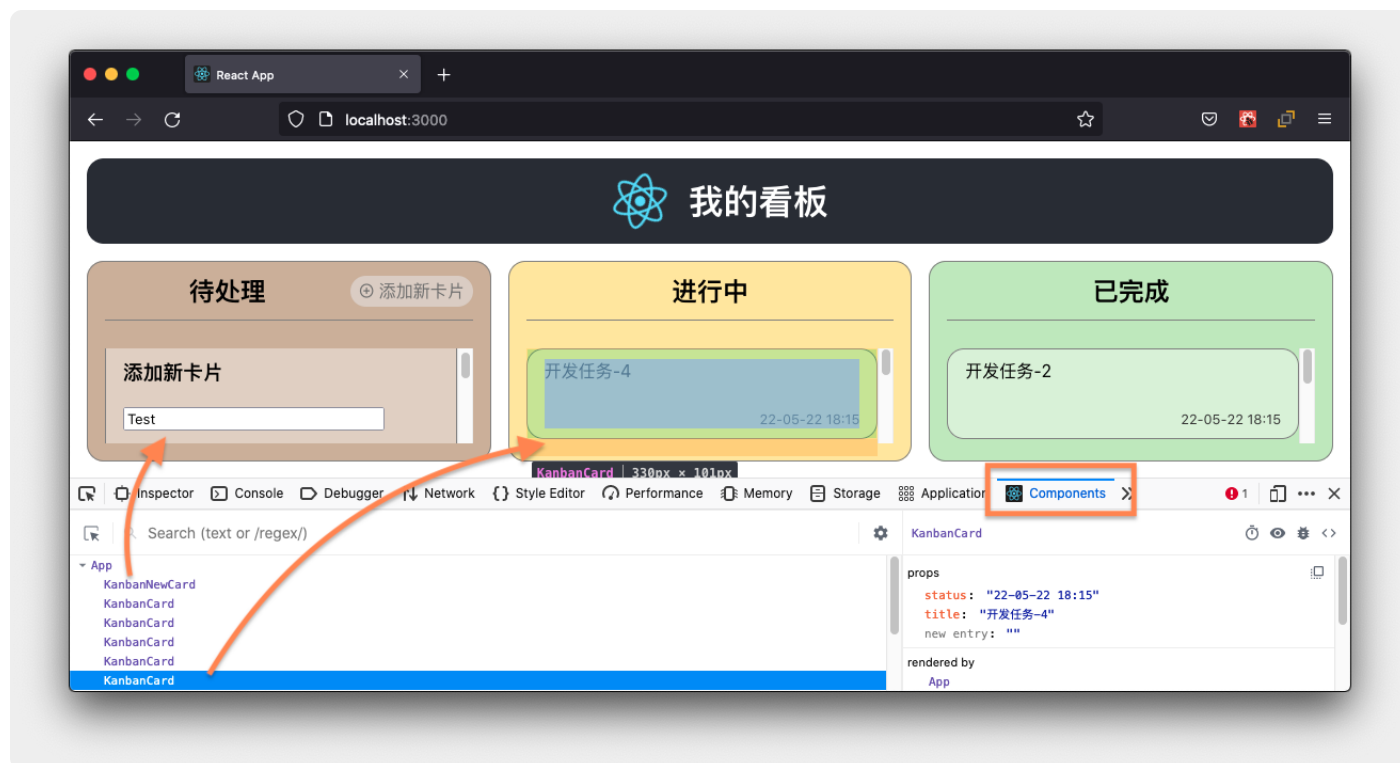
前面说的组件化和组件层次结构，基本也适用于其他前端框架。接下来具体到 **React**，该如何拆分组件呢？

用 JSX 协助拆分 React 组件

如何拆分组件，首先影响的就是 **JSX** 的写法。反过来说，你可以用 **JSX** 来快速验证拆分出来的组件层次结构。

现在请你用手头的 `oh-my-kanban` 项目（啥？你已经把第三节课写的代码删了？）做几个实验。第三节课的思考题是，请你安装 **FB** 官方的 **React Developer Tools** 扩展，并用扩展观察你的 `oh-my-kanban` 项目。现在它不止是思考题了，而是这些实验的必要准备工作。

安装好了，先 `npm start` 在浏览器中打开 <http://localhost:3000/>，打开开发者工具，切换到 **React** 的 **Component** 页签。如下图所示，你能看到一棵以 `App` 组件为根的组件树：



里面既没有 `kanban-board`，也没有 `kanban-column`，直接就到 `KanbanCard` 了。

```

1 App
2   |— KanbanNewCard
3   |— KanbanCard
4   |— KanbanCard
5   |— ...
6   |— KanbanCard

```

显然，React 扩展并没有把 `<main>`、`section` 当作组件，多个 `KanbanCard` 组件也没有分成“待处理”、“进行中”和“已完成”三个组，而是并列在了一起。不知你怎么样，我是犯了强迫症，我打算强迫 React 扩展认出这些组件。

目前的代码是这样的：

```

1 <main className="kanban-board">
2   <section className="kanban-column column-todo">
3     <h2>待处理</h2>
4     <ul>
5       { todoList.map(props => <KanbanCard {...props} />) }
6     </ul>
7   </section>
8   <section className="kanban-column column-ongoing">
9     <h2>进行中</h2>
10    <ul>
11      { ongoingList.map(props => <KanbanCard {...props} />) }
12    </ul>
13  </section>
14  { /* ...省略 */ }
15 </main>

```

你会怎么做呢？答对了，就是把 `<main>`、`section` 改写成 React 组件：

```

1 const KanbanBoard = ({ children }) => (
2   <main className="kanban-board">{children}</main>
3 );
4
5 const KanbanColumn = ({ children, className }) => {
6   const combinedClassName = `kanban-column ${className}`;
7   return (
8     <section className={combinedClassName}>
9       {children}

```

```
10     </section>
11   );
12 };
```

这两个组件的 props 中都有一个叫 children 的属性，这个属性一般不需要显式地传值，只要在 JSX 中写这个组件的标签时，在闭合标签内部加入子元素即可，子元素会自动作为 children 传给标签对应的组件。

因此我们在下面代码中，会用新建的两个 React 组件替代原来的 HTML 元素：

```
1  -<main className="kanban-board">
2  +<KanbanBoard>
3  -   <section className="kanban-column column-todo">
4  +   <KanbanColumn className="column-todo">
5       <h2>待处理</h2>
6       <ul>
7         { todoList.map(props => <KanbanCard {...props} />) }
8       </ul>
9  -   </section>
10  +   </KanbanColumn>
11  -   <section className="kanban-column column-ongoing">
12  +   <KanbanColumn className="column-ongoing">
13       <h2>进行中</h2>
14       <ul>
15         { ongoingList.map(props => <KanbanCard {...props} />) }
16       </ul>
17  -   </section>
18  +   </KanbanColumn>
19   { /* ...省略 */ }
20 -</main>
21 +</KanbanBoard>
```

保存文件，页面刷新了。如下图所示：



好，可以看到组件树包含了新写的两个组件，层次变深了，谢谢你满足了我的强迫症。

复制代码

```
1 App
2   └─ KanbanBoard
3       └─ KanbanColumn
4           └─ KanbanColumn
5               └─ KanbanColumn
6                   └─ KanbanCard
7                       ...
8                   └─ KanbanCard
```

可以看出，无论是这两个组件之间，还是它们与之前就有的 `KanbanCard` 之间，并没有明显关联。也就是说，它们三个组件只需要各扫门前雪，不需要关心其他组件的实现。此外，虽然它们之间最终形成了树形层次结构，但这层联系是在最外层的 `App` 定义的。

这时我估计你也上头了，你提出 `<h2> 待处理 </h2>`、`` 应该是 `KanbanColumn` 的实现细节，不应该由 `App` 来提供。我非常认同你的看法。

我们再进行一步调整 `KanbanColumn` 组件：

```

1 -const KanbanColumn = ({ children, className }) => {
2 +const KanbanColumn = ({ children, className, title }) => {
3   const combinedClassName = `kanban-column ${className}`;
4   return (
5     <section className={combinedClassName}>
6 -     {children}
7 +     <h2>{title}</h2>
8 +     <ul>{children}</ul>
9     </section>
10  );
11 };

```

这样就可以把 App 的代码改成：

```

1 <KanbanBoard>
2 - <KanbanColumn className="column-todo">
3 + <KanbanColumn className="column-todo" title="待处理">
4 -   <h2>待处理</h2>
5 -   <ul>
6     { todoList.map(props => <KanbanCard {...props} />) }
7 -   </ul>
8   </KanbanColumn>
9 - <KanbanColumn className="column-ongoing">
10 + <KanbanColumn className="column-ongoing" title="进行中">
11 -   <h2>进行中</h2>
12 -   <ul>
13     { ongoingList.map(props => <KanbanCard {...props} />) }
14 -   </ul>
15   </KanbanColumn>
16 - { /* ...省略 */ }
17 + { /* 代码减少，不用省略了 */ }
18 + <KanbanColumn className="column-done" title="已完成">
19 +   { doneList.map(props => <KanbanCard {...props} />) }
20 + </KanbanColumn>
21 </KanbanBoard>

```

这下 App 的负担就更小了。然而敏锐的你发现，页面少了些东西——是的，“添加新卡片”的按钮被我省略掉了。那要怎样加回来呢？

可以用 `props` 传进去，具体来讲就是用 `title` 属性传进去。在为 `KanbanColumn` 加入 `title` 属性时，虽然在前面 `App` 的 `JSX` 代码中传了字符串值，但我们还没有约定它必须是什么类型。

按照目前的需要，我们希望传 `React` 元素进去：

 复制代码

```
1 <KanbanColumn className="column-todo" title={
2   待处理<button onClick={handleAdd}
3     disabled={showAdd}>&#8853; 添加新卡片</button>
4 }>
5   { showAdd && <KanbanNewCard onSubmit={handleSubmit} /> }
6   { todoList.map(props => <KanbanCard {...props} />) }
7 </KanbanColumn>
```

这么写貌似是我们想要的，但这次修改导致编译失败了。在命令行终端中或者是浏览器页面上都能看到报错信息：

 复制代码

```
1 Failed to compile.
2
3 SyntaxError: /Users/evisong/dev/projects/oh-my-kanban/src/App.js: Unexpected tc
4 85 |         <KanbanBoard>
5 86 |             <KanbanColumn className="column-todo" title={
6 > 87 |                 待处理<button onClick={handleAdd}
7       |                     ^
8 88 |                 disabled={showAdd}>&#8853; 添加新卡片</button>
9 89 |             }>
10 90 |             { showAdd && <KanbanNewCard onSubmit={handleSubmit} /> }
11 ERROR in ./src/App.js
12 Module build failed (from ./node_modules/babel-loader/lib/index.js):
13 SyntaxError: /Users/evisong/dev/projects/oh-my-kanban/src/App.js: Unexpected tc
```



你可能会感到费解，一段文字加一个 HTML 元素，之前直接写在 `<h2></h2>` 里明明好好的，但抽取成 `props` 就不对了。从报错的位置可以看出是赋值给 `title` 属性的表达式有问题，但报错信息里提到的语法错误 `SyntaxError` 并没有告诉我们什么是正确写法，毕竟它不知道我们的本意是什么。

如果你还有印象，上节课提过一个小技巧，把这个有问题的表达式赋值给一个 JS 变量，看看会发生什么：

复制代码

```
1 const todoTitle = (待处理<button onClick={handleAdd}  
2   disabled={showAdd}>&#8853; 添加新卡片</button>);
```

虽然报错信息还是很模糊的语法错误，但你可以相信自己的判断，这并不是一段合法的 JSX。我们来试着修改它，把“待处理”字符串包在一对 HTML 标签里：

复制代码

```
1 const todoTitle = (<span>待处理</span><button onClick={handleAdd}  
2   disabled={showAdd}>&#8853; 添加新卡片</button>);
```

这次编译有了更友好的错误信息：

```

1 SyntaxError: /Users/evisong/dev/projects/oh-my-kanban/src/App.js: Adjacent JSX
2   77 |   };
3   78 |   const todoTitle = (
4 > 79 |     <span>待处理</span><button onClick={handleAdd}
5     |                                     ^
6   80 |         disabled={showAdd}>添加新卡片</button>
7   81 |   );
8   82 |   return (

```

这段 JSX 最外层需要包一对 Fragment，即 `<> </>`：

```

1 const todoTitle = (
2   <>
3     <span>待处理</span><button onClick={handleAdd}
4       disabled={showAdd}>添加新卡片</button>
5   </>
6 );

```

终于改对了！让我们把这段表达式放回 `title` 属性里。你在意的话，可以把额外加的 `` 删掉，它并不是之前语法错误的原因。

```

1 <KanbanColumn className="column-todo" title={
2   <>
3     待处理<button onClick={handleAdd}
4       disabled={showAdd}>添加新卡片</button>
5   </>
6 }>

```

大功告成！恭喜你既完成了组件拆分，又保证了实现的功能与拆分 `KanbanBoard` 和 `KanbanColumn` 组件之前一致。

顺便提一下，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](#)。下面代码来自 Semantic UI 的官方例子：

```

1 <Message icon>

```

```
2   <Icon name='circle notched' loading />
3   <Message.Content>
4     <Message.Header>
5       Just one second
6     </Message.Header>
7     We're fetching that content for you.
8   </Message.Content>
9 </Message>
```

其中 `Message.Content` 组件就是 `Message` 组件的 Sub-component。用这种方式改写 `KanbanColumn` 组件，大概会是这样的 JSX：

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

 复制代码

从 JSX 的角度看，这种模式下用 `KanbanColumn.Title` 子组件声明 `title`，比前面的表达式赋值显得更加规整一些。如果你感兴趣的话，在后面的课程中我会讲解一下这种模式的具体实现。

拆分组件的基本原则

通过刚才的实验，我们相当于用 `oh-my-kanban` 项目做了一道填空题：

现需用 **React** 技术开发一个看板应用项目，按照组件层次结构，可以将整个应用拆分成根组件 `App`、（1）组件、（2）组件和 `KanbanCard` 组件。

答案：（1）`KanbanBoard`；（2）`KanbanColumn`

这个答案肯定是可以得分的，但这却不是唯一的答案。只要你认为合理，尽管可以拆分出 `KanbanTodoColumn`、`KanbanOngoingColumn`、`KanbanDoneColumn` 等，也可以把两个组件合并成 `KanbanBoardWithColumns`。

有一点要专门提一下，在上面的实验中，我们先实现了 DOM 树，再返回来做组件拆分。而现实情况下，一般而言都是先做组件拆分，把具体实现留到拆分之后。类比一下，有点像是 Java 语言中先定义接口（Interface）再实现（Implement）接口，当然也可以先写一个实现再抽象一个接口出来。

就拆分方向而言，一般面对中小型应用，更倾向于从上到下拆分，先定义最大粒度的组件，然后逐渐缩小粒度；面对大型应用，则更倾向于从下往上拆分，先从较小粒度的组件开始。

无论从哪个方向拆分组件，都尽量遵守以下基本原则：

1. **单一职责（Single Responsibility）原则。**
2. **关注点分离（Separation of Concern）原则。**
3. **一次且仅一次（DRY, Don't Repeat Yourself）原则。**
4. **简约（KISS, Keep It Simple & Stupid）原则。**

你也许会怀疑我只是把最著名的几个编程原则罗列在这里，但请你放心，这些原则在后续的课程中都会一一露脸，届时会有实例来印证它们。

对拆分组件的建议

最后我想分享一个对拆分组件的建议。

决策疲劳（[Decision Fatigue](#)）是个心理学概念，大致意思就是说**当你连续做决定时，你的决定的效率和效果都会逐渐下降，甚至会做出错误的决定。**

在开发 React 应用时，为了实现一个完整的设计稿，你需要将其拆分成若干组件。而组件可大可小，可复杂可简单，你往往在组件拆分阶段需要连续做出决策，在颗粒度、复杂度、可维护性、可测试性间达到平衡。

这就容易导致决策疲劳，可能造成的后果就是，越靠后拆分的组件越拿不准，越怀疑前面我是不是拆错了，搞得自己很累。

为了减轻或避免拆分组件时的决策疲劳，我的建议是：

1. 没必要追求一次性拆分彻底，在具体实现过程中依然可以继续拆分组件；

2. 没必要追求绝对正确，在后续开发中可以根据需要，随时调整拆分过的组件；
3. 在拆分组件时尽量专注，暂时不要分神去考虑其他方面（如后端），少做些决策；
4. 在平时开发工作中有意积累组件拆分的经验，这会让你在后续的项目中游刃有余。

另外预告一下，组件什么时候该用 `props`、什么时候该用 `state`、还有 `context`，这也是 `React` 中容易导致决策疲劳的地方，后面课程里会讲到的单向数据流，可以帮你尽可能避免这种情况。

对 `React` 子组件概念的澄清

从第三节课开始，你可能已经对 `React` 的组件树形成了一个印象。如果你还有其他前端框架的开发经验，也许会很自然地把 `React` 组件树跟其他框架的组件树做类比，来帮助自己理解和学习。我个人也很推荐这种学习方式，类比老技术，在新技术学习初期是非常有帮助的。

但如果一直这样类比其他框架的组件树，你可能会对 `React` 独特的组件渲染机制有所误解，不利于后续课程的展开。所以在这节课末尾，正好是个合适的时机，澄清一下 `React` 的组件树跟其他框架有什么不一样。

严格来说，**`React` 没有组件树（Component Tree），只有元素树（Element Tree）**，即从根元素开始，父元素子元素之间形成的树。上节课学到，`React` 元素的子元素可以是 `React` 组件渲染的元素、`HTML` 元素，也可以是字符串，那么一定可以有下面的元素树：

 复制代码

```
1  图书馆
2  |— main
3  |   |— div
4  |       |— 分类阅览室
5  |— footer
6  |   |— "地址：XX路YY号"
```

其中图书馆、分类阅览室是 `React` 组件，`main`、`div` 是图书馆组件 `render()` 方法返回值的一部分，分类阅览室的渲染结果则作为前者 `div` 的子元素。

在 `React` 内部，尤其是引入新的 [Fiber 协调引擎](#) 之后，已经逐步不再依赖以类（`Class`）为中心的实现。元素（`Element`）只是节点的 `POJO`（`Plain Old JavaScript Object`）描述，非常

轻量，元素本身并不负责实例化类组件或是调用 `render` 方法。在类组件的实例上，也没有 `addChild()`，`getParent()` 这样描述组件间父子关系的方法或属性。函数组件更是如此。

我们经常提到的组件树和父子组件，其实可以从**组件声明**和**组件实例**两个层面来理解。

从组件声明层面：根据静态代码，**在一个组件返回的 JSX 中，加入另一个组件作为子元素，那么可以说前者是父组件，后者是子组件**。父子组件形成的树即为组件树。

但这种定义方法有可能会有误判，比如我们把 `oh-my-kanban` 的 `App` 组件稍作修改（`MyCustomTitle` 仅为示意，不用实现）：

 复制代码

```
1  const App = () => (  
2    <>  
3      <MyCustomTitle><h1>我的看板</h1></MyCustomTitle>  
4      <div>  
5        <KanbanBoard>  
6          <KanbanColumn title={<MyCustomTitle><h2>待处理</h2></MyCustomTitle>}>  
7            <KanbanCard />  
8            { /* ...省略 */ }  
9          </KanbanColumn>  
10         </KanbanBoard>  
11       </div>  
12     </>  
13   );
```

通过阅读静态代码，我们可以观察到组件间的关系如下：

- 很容易看出 `MyCustomTitle` 和 `KanbanBoard` 都是 `App` 组件的子组件；
- 但要注意 `KanbanColumn` 不是 `App` 的子组件，而是 `KanbanBoard` 的子组件；
- `KanbanCard` 明显是 `KanbanColumn` 的子组件，但通过 `title` 属性传递的 `MyCustomTitle` 是不是 `KanbanColumn` 的子组件呢？因为我们提前知道了内部实现，所以会把它认作是子组件，但如果 `KanbanColumn` 对你而言是个黑箱的话，并不能 100% 肯定 `MyCustomTitle` 是前者的子组件。

从组件实例层面：**组件树是来自运行时的 React 元素树、从逻辑上排除掉 HTML、Fragment 等元素，仅保留对应 React 组件的元素节点而形成的精简树**。在这棵组件树中，对应元素呈

父子关系的一对组件可以称作父子组件。

仍然以上面包含 `MyCustomTitle` 的代码为例，在运行时会产生如下 `React` 元素树：

复制代码

```
1 App
2   └─ Fragment
3       └─ MyCustomTitle
4           └─ h1
5       └─ div
6           └─ KanbanBoard
7               └─ KanbanColumn
8                   └─ MyCustomTitle
9                       └─ h2
10                  └─ KanbanCard
11                  └─ KanbanCard
```

把这棵元素树中非组件的节点过滤掉：

```
1 App
2   └─ Fragment
3       └─ MyCustomTitle
4           └─ h1
5       └─ div
6           └─ KanbanBoard
7               └─ KanbanColumn
8                   └─ MyCustomTitle
9                       └─ h2
10                  └─ KanbanCard
11                  └─ KanbanCard
```

会形成一棵逻辑上包含父子组件关系的 `React` 组件树，这正是你在 `React Developer Tools` 浏览器扩展中看到的：

复制代码

```
1 App
2   └─ MyCustomTitle
3   └─ KanbanBoard
4       └─ KanbanColumn
5           └─ MyCustomTitle
```

在后面的课程中，我们会讲到 **React** 的虚拟 **DOM** 和协调过程，那时你会对组件层级结构有更深入的理解。

小结

我们在这节课讲到了组件化是前端框架普遍采用的封装形式，将一个完整应用拆分成组件层次结构，会把业务和技术复杂度分摊到多个组件中去。

然后用 **oh-my-kanban** 项目的源码，实践了如何利用 **JSX** 协助拆分 **React** 组件。介绍了拆分组件的四个基本原则，也借此机会向你兜售了“决策疲劳”的理论。最后基于 **React** 的内部机制，形而上学地纠正了你对 **React** 子组件的理解。

下节课我们依然会从拆分组件工作入手，更深入地介绍 **React** 组件的渲染过程，也为之后要学习的组件生命周期、单项数据流等概念打基础。

最后附上本节课所涉及的源代码，供你学习与参考。对应的 Pull Request 是：

🔗 <https://gitee.com/evisong/geektime-column-oh-my-kanban/pulls/3>。此外，老师也打了一个 **v0.5.0** 版本标签：<https://gitee.com/evisong/geektime-column-oh-my-kanban/releases/tag/v0.5.0>。

思考题

除了浏览器，你在电脑上最常用的桌面应用是什么？是不是 **macOS** 的 **Finder** 或 **Windows** 的 **资源管理器**？

如果是的话就好办了。请你尝试把 **Finder** 或资源管理器当作要用 **React** 开发的 **Web** 应用，按自己的理解做一遍组件拆分。注意，拆分出来的组件不需要有完整的 **props**、**HTML**、事件处理实现，只要能用 **JSX** 搭建出来即可。

希望通过这一过程，能帮助你巩固对 **React** 组件颗粒度的把握。我们下节课再见！

Ta单独购买本课程，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 04 | JSX：如何理解这种声明式语法糖？

下一篇 06 | 虚拟DOM：聊聊React组件渲染机制

精选留言 (6)

💬 写留言



置顶

2022-09-01 来自北京

大家好，咱们课程的代码地址在这里哦

对应的Pull Request是：<https://gitee.com/evisong/geektime-column-oh-my-kanban/pulls/3>
打了一个v0.5.0版本标签：<https://gitee.com/evisong/geektime-column-oh-my-kanban/releases/tag/v0.5.0>



杨永安

2022-09-01 来自北京

哇，周四的凌晨更新

作者回复: 留言时间凌晨一点半，还请保重身体。



👍 2



杨永安

2022-09-01 来自北京

好奇真子组件模式

作者回复: 你好，杨永安，“真·子组件”学习心愿单+1，我记下来了：)

共 2 条评论 >

👍 1



tron

2022-09-13 来自浙江

对 React 子组件概念的澄清这一小节

对于组件树和元素树的不同之处，有点不是太理解

不知道是不是可以理解为，组件树是代码运行前的结构，代码运行后，组件return出元素，就成了元素树呢



阿阳

2022-09-01 来自江苏

周四，继续追



都市夜归人

2022-09-01 来自北京

```
const KanbanBoard = ({ children }) => ( <main className="kanban-board">{children}</main>);
```

缺少 return

作者回复: 你好，都市夜归人，感谢你捉虫，不过这次这个不是bug哈，ES6的箭头函数语法在函数体仅为单一表达式时，允许简写，类似这样：(params) => expression，其中 expression 会成为返回值。

