

04 | JSX：如何理解这种声明式语法糖？

2022-08-30 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 23:51 大小 21.78M



你好，我是宋一玮。上节课我们利用 Create React App（CRA）脚手架工具创建了一个 React 项目，并在项目中部分实现了一个简单的看板应用。在接下来的课程里，我们会把看板应用抽丝剥茧，逐一认识学习项目里涉及到的 React 概念和 API。很自然地，我们这节课会讲到 JSX 语法和 React 组件。

有不少初学者对 React 的第一印象就是 JSX 语法，以至于会有这样的误解：

- JSX 就是 React？
- JSX 就是 React 组件？
- JSX 就是另一种 HTML？
- JSX 既能声明视图，又能混入 JS 表达式，那是不是可以把所有逻辑都写在 JSX 里？

这些误解常会导致开发时遇到各种问题：

- 写出连续超百行、甚至近千行的 **JSX** 代码，既冗长又难维护；
- 在 **JSX** 的标签上添加了 **HTML** 属性却不生效；
- **JSX** 混入 **JS** 表达式后，页面一直报错。

其实只要**理清了 JSX 和 React 组件的关系**，这些问题自然不在话下。

总的来说，**React** 是一套声明式的、组件化的前端框架。顾名思义，**声明（动词）组件**是 **React** 前端开发工作最重要的组成部分。在声明组件的代码中使用了 **JSX** 语法，**JSX** 不是 **HTML**，也不是组件的全部。

接下来，我们就详细展开介绍 **JSX** 和 **React** 组件。

JSX 是语法糖

Web 应用日益复杂，其视图中往往包含很多的控制逻辑，比如条件、循环等。以声明式开发视图，就需要把控制逻辑代码也加入到声明语句中去。而这样的代码，就对可读性、可维护性提出了挑战。

在 **JSX** 之前，前端领域有各种视图模版技术，**JSP**、**Struts**、**Handlebars**、**Pug** 等，都在用各自的方法满足这些需求。那么 **JSX** 语法与其他声明式模版语法有什么异同？不用 **JSX** 可以写 **React** 吗？

我们在这节课开始时提到了 **React** 组件，组件是 **React** 开发的基本单位。在组件中，需要被渲染的内容是用 `React.createElement(component, props, ...children)` 声明的，而 **JSX** 正是 `createElement` 函数的语法糖。浏览器本身不支持 **JSX**，所以在应用发布上线前，**JSX 源码需要工具编译成由若干createElement函数组成的 JS 代码，然后才能在浏览器中正常执行**。至于编译工具，我们在后面的课程会有所涉及。

例如，上节课看板组件的部分 **JSX**：

 复制代码

```
1 <header className="App-header">
2   <h1>我的看板</h1>
3 </header>
```

编译成 JS 就会变成：

 复制代码

```
1 React.createElement("header", {className: "App-header"},
2   React.createElement("h1", null, "我的看板")
3 );
```

当然你也可以选择不用 JSX，而是自己手写这些 JS 代码。这样做最显著的好处就是，这部分代码不需要针对 JSX 做编译，直接可以作用于浏览器。但当元素或者元素的嵌套层级比较多时，JS 代码的右括号会越来越多。当你看到成篇的)))))))；时，你的代码和内心会有一个先崩溃。就算 IDE 帮忙自动格式化，对应层级缩进，也没法减少括号嵌套的数量。

也许是因为先入为主，在 Web 领域，类 HTML 语法天生就更受欢迎。**JSX 提供的类 HTML/XML 的语法会让声明代码更加直观**，在 IDE 的支持下，语法高亮更醒目，**比起纯 JS 也更容易维护**。相比 JSX 带来的开发效率的提升，编译 JSX 的成本基本可以忽略不计。

如果光看 JSX 中“X”的部分，还不足以让它和其他 HTML/XML 模版技术区别开来，这里还要强调一下 JSX 中“JS”的部分。请你回忆一下我们在上节课写的 JSX 代码，以里面的条件渲染为例：

 复制代码

```
1 { showAdd && <KanbanNewCard onSubmit={handleSubmit} /> }
```

我们来对比一下 Java SSH（Spring+Struts2+Hibernate）技术栈里 Struts2 模版的写法：

 复制代码

```
1 <s:if test="showAdd">
2   <div>KanbanNewCard ...</div>
3 </s:if>
4
```


可以发现两者判断条件的语义是相同的，区别是 Struts2 用 XML 定义了一套名为标签库的 DSL（Domain-Specific Language，领域特定语言），由标签库提供的 `<s:if></s:if>` 做条件渲染；而 JSX 则直接利用了 JS 语句。很明显，JS 表达式能做的，JSX 都能做，不需要开发者再去学习一套新的 DSL。

也正是因为 **JSX** 作为语法糖足够“甜”，我们才能得到这样的结论：**JSX** 是前端视图领域“最**JS**”的声明式语法，它为 **React** 的推广和流行起了至关重要的作用。

前端开发中的声明式与命令式

既然刚才提到了声明式（**Declarative**），就一定要提一下命令式（**Imperative**）。这两种编程范式的 **PK** 存在于软件开发的各个领域。下面的表格呢，从（非）现实世界用例、各领域代表性技术、具体 **JS** 语句三个方面，将声明式和命令式做了一个对比。

	声明式	命令式
现实世界例子	我希望大象在冰箱里。	打开冰箱门； 把大象装到冰箱里； 关上冰箱门。
代表性编程范式	函数式编程	面向对象编程
代表性后端技术	SQL	Java
代表性前端技术	HTML、CSS	DOM API
代表性前端框架	React	jQuery
JS条件语句	a ? b : c	if (a) { console.log(b) } else { console.log(c) }
JS循环语句	array.map(item => (item.prop))	for (let i; i < array.length; i++) { console.log(array[i]) }

 极客时间 | 现代React Web开发实战@宋一玮

React 是声明式的前端技术，这一点首先就体现在创建组件的视图上，无论是使用 **JSX** 语法还是直接利用 **React.createElement()** 函数，都是在**描述开发者期待的视图状态**。开发者只需关心渲染结果，而 **React** 框架内部会实现具体的渲染过程，最终调用浏览器 **DOM API**。

你可能会感兴趣：“除了 **jQuery**，还有其他的前端框架是命令式的吗？”肯定是有的，但很明显，声明式才是主流。目前的三大主流前端框架，**React**、**Vue**、**Angular** 都是声明式的。包括 **Flutter** 这样的新兴跨端框架也类似，都采用了典型的声明式 **API**，以下是 **Flutter** 的官方例子：

 复制代码

```
1 Widget titleSection = Container(  
2   padding: const EdgeInsets.all(32),  
3   child: Row(  
4     child: Text(  
5       titleSectionTitle,  
6       style: TextStyle(  
7         color: Colors.black87,  
8         fontSize: 16,  
9       ),  
10    ),  
11  ),  
12 );
```

```

4     children: [
5         Expanded(
6             child: Column(
7                 crossAxisAlignment: CrossAxisAlignment.start,
8                 children: [
9                     Container(
10                        padding: const EdgeInsets.only(bottom: 8),
11                        child: const Text('Oeschinen Lake Campground'),
12                    ),
13                    Text('Kandersteg, Switzerland'),
14                ],
15            ),
16        ),
17        Icon(
18            Icons.star,
19            color: Colors.red[500],
20        ),
21        const Text('41'),
22    ],
23 ),
24 );

```

多少有点眼熟吧。很有意思的是，从 2017 年开始，每年都有 Flutter 用户在社区中呼吁引入 JSX 语法（[🔗#11609](#)、[🔗#15922](#)、[🔗#70928](#)），但这一愿望都没有实现。这又一次彰显了 JSX 这种语法糖的吸引力。

理解了 JSX 是语法糖，其真实身份是声明式的 `React.createElement()` 函数，接下来我们来看看它的具体写法。

JSX 的写法和常见坑

先回到一个简单的问题上，JSX 是哪几个单词的缩写？是的，**JavaScript XML**，即在 JS 语言里加入类 XML 的语法扩展。这样我们就可以把 JSX 一分为二：先介绍 **X** 的部分，即标签的命名规则，支持的元素类型、子元素类型；然后是 **JS** 的部分，即 JSX 中都有哪里可以加入 JS 表达式、规则是什么，进一步回顾上节课的条件渲染和循环渲染表达式。

JSX 的基本写法

请你回顾一下上节课 `src/App.js` 的内容，我们将以 App 组件为例，串讲一下 JSX 的常规写法和写 JSX 时常踩的坑。为了方便参考，我会在这里贴一部分 App 组件的源码。

```

1 function App() {
2   const [showAdd, setShowAdd] = useState(false);
3   const [todoList, setTodoList] = useState([]);
4   const handleAdd = (evt) => {
5     setShowAdd(true);
6   };
7   const handleSubmit = (title) => {
8     setTodoList(currentTodoList => [
9       { title, status: new Date().toLocaleDateString() },
10      ...currentTodoList
11    ]);
12    setShowAdd(false);
13  };
14
15  return (
16    <div className="App">
17      <header className="App-header">
18        <h1>我的看板</h1>
19        <img src={logo} className="App-logo" alt="logo" />
20      </header>
21      <main className="kanban-board">
22        <section className="kanban-column column-todo">
23          <h2>待处理<button onClick={handleAdd}
24            disabled={showAdd}>&#8853; 添加新卡片</button></h2>
25          <ul>
26            { showAdd && <KanbanNewCard onSubmit={handleSubmit} /> }
27            { todoList.map(props => <KanbanCard {...props} />) }
28          </ul>
29        </section>
30        { /* ...省略 */ }
31      </main>
32    </div>
33  );
34 }

```

虽然在写 JSX 时并不需要时时惦记着编译出来的 `React.createElement()` 语句，但在学习时还是很有帮助的。我们来看一下 JSX 各个组成部分与 `React.createElement()` 函数各参数的对应关系，代码如下：

 复制代码

```

1 React.createElement(type)
2 React.createElement(type, props)
3 React.createElement(type, props, ...children)

```

其中 `type` 参数是必须的，`props` 可选，当参数数量大于等于 3 时，可以有一个或多个 `children`。

以下是一个具体例子：

 复制代码

```
1   <li className="kanban-card">
2   <!-- ^^  ^^^^^^^^^^^  ^^^^^^^^^^^^^^^^^^^
3   type  props-key  props-value                                -->
4       <div className="card-title">{title}</div>  <!-- children -->
5       <div className="card-status">{status}</div> <!-- ____|      -->
6   </li>
```

把 children 中的一个成员单独来看，也是对应一条createElement() 语句的：

 复制代码

```
1   <div className="card-title">{title}</div>
2   <!-- ^^^  ^^^^^^^^^^^  ^^^^^^^^^^^^^^^^^^^  ^^^^^^^^^
3   type  props-key  props-value  children -->
```


你可以在这个 [在线 Babel 编译器](#) 中做各种实验。

这里额外说一个大坑。当App 代码 return 语句返回 JSX 时，将 JSX 包在了一对括号 () 里，这是为了避免踏入 JS 自动加分号的陷阱。例如：

 复制代码

```
1  function Component() {
2    return
3      <div>{/*假设这行JSX语句很长，为了提升一些代码可读性才特地换行*/}</div>;
4  }
```

放到编译器里会生成：

 复制代码

```
1  function Component() {
2    return;
3    React.createElement("div", null);
4  }
```


整个函数短路了！根本不会执行到`React.createElement()` 语句。为了修正这个问题，我们需要为 **JSX** 加上括号：

 复制代码

```
1 function Component() {
2   return (
3     <div>{/*假设这行JSX语句很长，为了提升一些代码可读性，特地换行*/}</div>
4   );
5 }
```

再次编译：

 复制代码

```
1 function Component() {
2   return React.createElement("div", null);
3 }
```

终于对了。

你能想象当年我和同事找 **Bug** 找了一整天，最后发现只是 `()` 两个字符的问题吗？“一朝被蛇咬，十年怕井绳。”自此，我养成了为 **JSX** 最外层加括号的习惯，甚至连单行 `return` 都会加上括号。毕竟在改老代码时，单行 `return` 有可能会改成多行，留下忘加括号的隐患。

命名规则

俗话说“无规矩不以成方圆”，学习 **JSX**，就让我们从命名规则开始。

自定义 **React** 组件时，组件本身采用的变量名或者函数名，需要以大写字母开头。

 复制代码

```
1 function MyApp() {
2   //_____^
3   return (<div></div>);
4 }
5 const KanbanCard = () => (
6   //____^
7   <div></div>
8 );
```


在 JSX 中编写标签时，HTML 元素名称均为小写字母，自定义组件首字母务必大写。

复制代码

```
1      <h1>我的看板</h1>
2  <!-- ^_____全小写 -->
3      <img src={logo} className="App-logo" alt="logo" />
4  <!-- ^^^_____全小写 -->
5      <button onClick={handleAdd} disabled={showAdd}>添加新卡片</button>
6  <!-- ^^^^^^^____全小写 -->
7
8
9      <KanbanCard />
10 <!-- ^_____首字母大写 -->
```

如果你很坚持自定义组件也要全小写，那我鼓励你亲手试一下，比如

`<camelCaseComponent />`。在浏览器开发者工具中定位到这个元素，你会发现 React 把它当成了一个不规范的 HTML 标签直接丢给了浏览器，而浏览器也不认识它，直接解析成 `<camelcasecomponent></camelcasecomponent>`。这也算是 React 的一种约定大于配置（Convention Over Configuration）了。

至于 props 属性名称，在 React 中使用驼峰命名（camelCase），且区分大小写，比如在 `<FileCard filename="文件名" fileName="另一个文件名" />` 中，你可以同时传两个字母相同但大小写不同的属性，这与传统的 HTML 属性不同。

JSX 元素类型

从前面的源码来看，我们在代表组件的函数里，返回了一整段 JSX。JSX 产生的每个节点都称作 React 元素，它是 React 应用的最小单元。React 元素有三种基本类型：

1. React 封装的 DOM 元素，如 `<div></div>`、``，这部分元素会最终被渲染为真实的 DOM；
2. React 组件渲染的元素，如 `<KanbanCard />`，这部分元素会调用对应组件的渲染方法；
3. React Fragment 元素，`<React.Fragment></React.Fragment>` 或者简写成 `<></>`，这一元素没有业务意义，也不会产生额外的 DOM，主要用来将多个子元素分组。

其他还有 Portal、Suspense 等类型，这节课我们先不展开。

我们会为 JSX 元素加入 props，不同类型元素的 props 有所区别。

React 封装的 DOM 元素将浏览器 DOM 整体做了一次面向 React 的标准化，比如在 HTML 中很容易引起混淆的 `readonly="true"`，它的 W3C 标准应该是 `readonly="readonly"`，而常被误用的 `readonly="false"` 其实是无用的（谐音梗），在 React JSX 中就统一为 `readOnly={true}` 或 `readOnly={false}`，更贴近 JS 的开发习惯。

至于前面反复出现的 `className="kanban-card"`，更多是因为 HTML 标签里的 `class` 是 JS 里的保留字，需要避开。

React 组件渲染的元素，JSX 中的 props 应该与自定义组件定义中的 props 对应起来；如果没有特别处理，没有对应的 props 会被忽略掉。这也是开发 JSX 时偶尔会犯的错误，在组件定义中改了 props 的属性名，但忘了改对应的 JSX 元素中的 props，导致子组件拿不到属性值。

至于 Fragment 元素，没有 props。

JSX 子元素类型

JSX 元素可以指定子元素。在之后的课程里你会看到很多子组件的概念，这里先留一个印象：**子元素不一定是子组件，子组件一定是子元素。**

子元素的类型包括：

1. 字符串，最终会被渲染成 HTML 标签里的字符串；
2. 另一段 JSX，会嵌套渲染；
3. JS 表达式，会在渲染过程中执行，并让返回值参与到渲染过程中；
4. 布尔值、null 值、undefined 值，不会被渲染出来；
5. 以上各种类型组成的数组。

JSX 中的 JS 表达式

在 JSX 中可以插入 JS 表达式，特征是用大括号 { } 包起来，主要有两个地方：

1. 作为 props 值，如 `<button disabled={showAdd}>添加新卡片</button>`；
2. 作为 JSX 元素的子元素，如 `<div className="card-title">{title}</div>`。

这些表达式可以简单到原始数据类型 `{true}`、`{123}`，也可以复杂到一大串 Lambda 组成的函数表达式 `{ todoList.filter(card => card.title.startsWith('TODO:')).map(props => <KanbanCard {...props} />) }`，只要确保最终的返回值符合 props 值或者 JSX 子元素的要求，就是有效的表达式。

前面也讲到，**JSX 是声明式的，所以它的内部不应该出现命令式的语句**，如 `if ... else ...`。当你拿不准自己写到 JSX `{ }` 里的代码到底是不是表达式，可以试着把这部分代码直接赋值给一个 JS 变量。如果这个赋值能成功，说明它确实是表达式；如果赋值不成功，可以从如下四个方面进行检查：

- 是否有语法错误；
- 是否使用了 `for...of` 的声明式变体 `array.forEach`，这个中招几率比较高；
- 是否没有返回值；
- 是否有返回值，但不符合 props 或者子元素的要求。

另外有个 props 表达式的特殊用法：属性展开，`<KanbanCard {...props} />` 利用 JS `...` 语法把 props 这个对象中的所有属性都传给 KanbanCard 组件。

对了，如果你想在 JSX 里加注释，会发现 HTML 注释 根本没法通过编译，这时需要改用 `{/* */}` 来加注释，编译时它会被识别成 JS 注释然后抛弃掉。

回顾条件渲染和循环渲染

有了上面的知识，我请你再回顾一下上节课中的条件渲染和循环渲染：

 复制代码

```
1 { showAdd && <KanbanNewCard onSubmit={handleSubmit} /> }  
2
```

上面是一个典型的条件表达式，如果 `showAdd` 为 `true` 时，会返回后面的 `JSX`，渲染《新建看板卡片》组件；否则会返回 `showAdd` 的值，即 `false`。根据子元素类型中描述的，`false` 值并不会被渲染出来，《新建看板卡片》组件就不会被渲染了。

 复制代码

```
1 { todoList.map(props => <KanbanCard {...props} />) }
```

上面是一个典型的数组转换表达式。当 `todoList` 为空数组时，表达式返回一个新的空数组，不会渲染出来；而当 `todoList` 包含 1 个或更多个项目时，会返回一个 `JSX` 的数组，相当于：

 复制代码

```
1 {[
2   <KanbanCard title="开发任务-1" status="22-05-22 18:15" />,
3   <KanbanCard title="开发任务-2" status="22-05-22 18:15" />
4 ]}
```

JSX 与 React 组件的关系

你终于忍不住问出这个问题：“前面课里反复提到 `React` 组件，为啥一个普普通通的 `function App() {}` 函数就成组件了？”

这是个好问题！

鲁迅笔下的名人孔乙己曾说过“回字有四样写法”，巧了，`React` 组件也是。`React` 组件最初不是这么精简的。目前 `React` 的版本是 `v18`，7 年前的 2015 年 `React` 发布了两个大版本 `v0.13` 和 `v0.14`（你可以理解成 `v13` 和 `v14`），当时 `React` 组件的主流写法是：

 复制代码

```
1 const KanbanCard = React.createClass({
2   render: function() {
3     return (<div>KanbanCard ...</div>);
4   }
5 });
```

FB 官方在 `v0.13` 中开始推广 [ES6 class](#) 的写法：

```
1 class KanbanCard extends React.Component {  
2   render() {  
3     return (<div>KanbanCard {this.props.title}</div>);  
4   }  
5 }
```

用这两种方式定义组件时，最核心的就是实现 `render()` 方法。`render()` 方法的返回值可以是一段 **JSX**（或对应的 **React 元素**）、**原始数据类型**（注：该方法在 React v18 以前的版本不可以返回 `undefined`，否则会报错）、**其他 React 数据类型**或者是这几种类型的数组。

除了 `render()` 方法，这两种写法还能加入其他属性和方法，完整实现 React 组件具有的状态管理、生命周期、事件处理等功能，这些功能我们放在后续的课程里，在这里暂时不展开。所以说 **JSX 只是 React 组件的一部分**，这就澄清了“**JSX 就是 React 组件**”这个误解。

除了前面两种写法，在 v0.14，React 新加入了一种更为简化的**无状态函数组件**（[🔗 Stateless Function Component](#)）：

```
1 // ES6箭头函数  
2 const KanbanCard = (props) => {  
3   var title = props.title;  
4   return (<div>KanbanCard {title}</div>);  
5 };  
6  
7 // 更简单的箭头函数+参数解构  
8 const KanbanCard = ({title}) => (  
9   <div>KanbanCard {title}</div>  
10 );
```

函数的参数就是 `props`，函数的返回值与前面两种写法中 `render()` 方法的返回值相同。这种函数组件在 **React Hooks** 尚未发布时，还不能自己处理 `state` 状态，需要在它的父组件提供状态，并通过 `props` 传递给它。虽然函数组件功能受限，但它贵在简单，受到了开发者的广泛欢迎。以至于开源社区开发了各种支持库，用诸如高阶组件的方式补足函数组件缺失的功能。

当时最出名的库莫过于 [🔗 recompose](#)，举个简单的例子：

```
1 import { useState } from 'recompose';
2
3 const enhance = useState('showAdd', 'setShowAdd', false);
4 const KanbanColumn = enhance(({ showAdd, setShowAdd }) => (
5   <section className="kanban-column column-todo">
6     <h2>
7       待处理
8       <button onClick={() => setShowAdd(true)}>添加新卡片</button>
9     </h2>
10    <ul>
11      { showAdd && <KanbanNewCard /> }
12    </ul>
13  </section>
14  ));
```

其中可以看到 KanbanColumn 组件的主体是 enhance 参数的箭头函数组件。前面 recompose 的 withState(stateName, stateUpdaterName, initialState) 函数会创建一个单一功能的高阶组件（高阶组件后面课程会讲到），它会创建名为 showAdd 的 state，并通过 props 传递给作为子组件的函数组件，父子组件结合在一起，形成一个功能完整的 React 组件。顺便一提，后来 recompose 的作者还加入了 React 官方开发组。

到了 React v16.8，Hooks 正式发布，函数组件取代类组件成为了 React 组件的 C 位。题外话，对于 React 函数组件的流行，我在当年是有点意外的。我本人是 ES6 class 的死忠粉，但后来先后上手了 recompose 和官方的 Hooks，真香。

当然，介绍这段历史并不是为了吃瓜，最重要的还是回答你刚才的问题“为啥一个普普通通的函数就成组件了”。

简单总结一下，函数组件上位的原因包括：

- React 的哲学 $UI=f(state)$ ；
- 更彻底的关注点分离（Separation Of Concerns）；
- 函数式编程的影响；
- React 内部实现的不断优化；
- 开源社区的反哺。

小结

这节课我们学习了 **JSX** 的概念和写法，同时也引出了 **React** 声明式的特性，也初步聊了一下 **React** 组件。

这时我相信你已经不会再有这节课开头的误解了：

- **JSX 就是 React?**
 - 不是。**JSX** 只是 **React** 其中一个 **API**，**createElement** 函数的语法糖。
- **JSX 就是 React 组件?**
 - 不是。**JSX** 是 **React** 组件渲染方法返回值的一部分，**React** 组件还有其他的功能。
- **JSX 就是另一种 HTML?**
 - 不是。**JSX** 本质还是 **JS**，只是在最终渲染时才创建修改 **DOM**。
- **JSX 既能声明视图，又能混入 JS 表达式，那是不是可以把所有逻辑都写在 JSX 里?**
 - 可以是可以，但毕竟不能在 **JSX** 里使用命令式语句，能做的事情很有限。

运用好 **JSX**，可以很大程度提高你的 **React** 开发效率和效果。

下一讲，我们将趁热打铁，继续探讨 **React** 组件，从比 **React** 元素颗粒度更大的层面，认识 **React** 渲染的机制。同时也学习如何从业务和技术两方面入手，将一份原始的需求拆解为若干 **React** 组件。

思考题

这一讲中间举过一个 **Flutter** 的例子，提到用户希望将 **JSX** 语法引入 **Flutter**。想请你按这个思路思考如下两个问题：

1. **JSX** 一定得是 **React** 吗？**React** 以外的技术能不能使用 **JSX**？
2. **JSX** 一定得生成 **HTML** 吗？可以用 **JSX** 生成其他模版吗？

欢迎把你的想法分享在留言区，我会和你交流。相信经过你的深度思考，学习效果会更好！我们下节课再见！

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 18 元



生成海报并分享



赞 5



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 03 | 看板应用：从零开始快速搭建一个React项目

下一篇 05 | 前端组件化：将完整应用拆分成React组件

精选留言 (13)

写留言



万千观众之一

2022-08-30 来自北京

这一节课下来干货好多哇

作者回复: 你好，万千观众之一，多谢你的评论，希望你有所收获。当然，如果发现后面课程内容过于干以至于不好消化的话，请务必告诉我，我会尽量调整。



2



coder

2022-08-31 来自北京

1. JSX 是函数的语法糖，那 JS 相关框架函数都能实现 JSX
2. 函数的原理就是输入什么会得到一个确定的结果返回，理论上就可以输出成其他需要的结果

作者回复: 你好，coder，很棒的答案！正如你提到的，诸如Vue和Solid.js这些JS框架也都加入了JSX特性。



1



东方奇骥

2022-08-31 来自北京

1. JSX 一定得是 React 吗？React 以外的技术能不能使用 JSX？

答：不一定。JSX 并不是一个新的模板语言，可以认为是一个语法糖。比如Vue也有JSX。

2. JSX 一定得生成 HTML 吗？可以用 JSX 生成其他模版吗？

答：课程中讲到，本质上来说，JSX可以认为是一个语法糖，最终还是调用React.createElement. 所以理解重写一个createElement也可能生成别的，不一定是HTML。

作者回复: 你好，东方奇骥，很棒的答案！你说得对，createElement也可以生成别的，比如React Native中的移动端原生组件。



心叶

2022-08-31 来自北京

讲到jsx，为什么不直接拿出官方文档呢？

<https://facebook.github.io/jsx/>

从文档你可以知道：

jsx不是react的api，虽然是react团队搞出来的

他的灵感是es4里面的e4x，但原本的e4x因为涉及到语法和语义的定义，实现过于复杂所以被弃用。

jsx的目标是供预处理器使用，将其转换成es

作者回复: 你好，心叶，你提到的JSX的官方文档 <https://facebook.github.io/jsx/> 确实是最权威的也是最精炼的，不过它是以web规范的格式写作的，初学时读起来可能会有些费力。其他同学在基本掌握JSX后，感兴趣的话也可以回来读一下这篇文档，一定会发现它的妙处。

另外你提到了ES4，真是闻者伤心听者流泪，业界最贴近ES4规范的实现是Adobe Flash中的ActionScript 3语言，随着Flash技术的覆灭，ECMA毅然抛弃了ES4，转而发布了ES5，之后也成为了各浏览器JS标准化的基石。



Hello, Tomrrow

2022-08-30 来自北京

JSX 不是在 React 中发明的，二者的关系更像是相互成就。

作者回复: 你好，Hello, Tomrrow，很高兴看到你坚持打卡。你说得对，二者互相成就。



1



01

2022-09-15 来自福建

jsx 并不直接生成html。



Geek_fcdf7b

2022-09-03 来自北京

首先，感谢老师对于评论区的每个问题几乎都在回复，从评论中也学到了很多。然后，请教一下，V17之后，JSX好像不一定是编译成React.createElement了吧，好像有个react/jsx-runtime

作者回复: 你好，Geek_fcdf7b，感谢你的指正。你说得对，React从17开始已经启用全新的JSX运行时来替代React.createElement。对这个变化，我的印象还停留在当时一个预发布版本的可选功能，就疏忽了。

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>

我在oh-my-kanban项目里验证了一下，确实在开发模式下JSX被编译成了react/jsx-dev-runtime下的jsxDEV，在生产模式下则被编译成了react/jsx-runtime下的jsx或jsxss（目前同jsx）。

```
var KanbanCard = function KanbanCard(_ref) {
  var title = _ref.title,
      status = _ref.status;
  return /*#__PURE__*/(0, jsx_runtime.jsx)("li", {
    className: "kanban-card",
    children: [
      /*#__PURE__*/(0, jsx_runtime.jsx)("div", {
        className: "card-title",
        children: title,
      }),
      /*#__PURE__*/(0, jsx_runtime.jsx)("div", {
        className: "card-status",
        children: status,
      })
    ]
  });
};
```

```
],  
});  
};
```

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

JSX的语法没有改变，返回值也都是`React`元素，代码编译器对开发者隐藏了新旧API的差异。从学习理解JSX的角度，影响不大。大家可以先按照目前的文稿来学习，我后续写完新稿后会回来更新文稿。

再次感谢Geek_fcdf7b！



雨猫

2022-09-02 来自四川

怎么大家收货这么多呀



学习前端-react 

2022-09-01 来自北京

请问：如上我们理解的声明式在编程上便是函数式编程，在`jsx`上便是 三目运算符 和 `Function map`，所有在`vue`的模板里，`v-if v-for` 是不是不太声明式？

作者回复: 你好，Geek_8aba0d，函数式编程是声明式的一种，但声明式还有其他编程范式；JSX是一种声明式，但声明式还有其他模版技术。在前端领域，我们提到声明式时，对应的另一边主要还是命令式。在我看来Vue的`v-if`、`v-for`，与Angular里`*ngIf`、`*ngFor`，都不能算是命令式，所以它们也是声明式的。



学习前端-react 

2022-09-01 来自北京

JSX 一定得是 `React` 吗？`React` 以外的技术能不能使用 `JSX`？

不是 这是一个`dsl`，其他语言只要实现其底层，便可使用其上层的`jsx`

JSX 一定得生成 `HTML` 吗？可以用 `JSX` 生成其他模版吗？

如上。

作者回复: 你好，Geek_8aba0d，很好的答案。



阿阳

2022-09-01 来自北京

最近在vue项目中引入了jsx，在自定义组件的时候，恰好踩到了这节课说的几个坑。帮助很大。

jsx应该不是react独有的，它只是个语法糖，它可以被编译为任意的其他渲染函数。

作者回复: 我印象中Vue是从2版本加入了JSX特性，与它原有的template有一定替代关系，需要开发者做一些思路上的转换。不过在底层，正如你提到的，两者都会编译成Vue的渲染函数。



杨永安

2022-08-31 来自北京

jsx本质是一个返回格式为json的node节点描述信息。可以用在跨端跨平台的用途，比如拿到json作为render蓝本的时候，最后的render会根据宿主环境对应调用相应API。

话说这课没有了吗？

编辑回复: 每周二四六更新哈，还在更新中

共 2 条评论 >



即将暴富的木杉

2022-08-30 来自北京

vue 的 template 的实现就是基于jsx的吧

作者回复: 你好，即将暴富的木杉，Vue的template严格来说不是JSX，它虽然也是被编译成渲染函数，但编译过程，尤其是v-if、v-for这样的指令，与JSX语句是不同的；JSX是Vue的可选特性，它离渲染函数更近，也更“JS”。据我所知，Vue的同一个组件中不能混用<template>和JSX。

以下是摘自Vue的官方文档：

```
<ul>
  <li v-for="{ id, text } in items" :key="id">
    {{ text }}
  </li>
</ul>
```

等价于使用如下渲染函数 / JSX 语法：

```
<ul>
  {this.items.map(({ id, text }) => {
    return <li key={id}>{text}</li>
  })}
</ul>
```

