

01 | JSX 代码是如何“摇身一变”成为 DOM 的?

2020/10/12 修言



12.88M

00:00/14:00



看视频

时下虽然接入 JSX 语法的框架越来越多，但与之缘分最深的毫无疑问仍然是 React。2013 年，当 React 带着 JSX 横空出世时，社区曾对 JSX 有过不少的争议，但如今，越来越多的人面对 JSX 都要说上一句“真香”！本课时我们就来一起认识下这个“真香”的 JSX，聊一聊“JSX 代码是如何‘摇身一变’成为 DOM 的”。

关于 JSX 的 3 个“大问题”

在日常的 React 开发工作中，我们已经习惯了使用 JSX 来描述 React 的组件内容。关于 JSX 语法本身，相信每位 React 开发者都不陌生。这里我用一个简单的 React 组件，来帮你迅速地唤醒自己脑海中与 JSX 相关的记忆。下面这个组件中的 render 方法返回值，就是用 JSX 代码来填充的：

```
1. import React from "react";
2. import ReactDOM from "react-dom";
3.
4. class App extends React.Component {
5.   render() {
6.     return (
7.       <div className="App">
8.         <h1 className="title">I am the title</h1>
9.         <p className="content">I am the content</p>
10.      </div>
11.    );
12.  }
13. }
14.
15. const rootElement = document.getElementById("root");
16. ReactDOM.render(<App />, rootElement);
```

■ 复制代码

由于本专栏的整体目标是帮助你在 React 这个领域完成从“小工”到“行家”的进阶，此处我无意再去带你反复咀嚼 JSX 的基础语法，而是希望能够引导你去探寻 JSX 背后的故事。针对这“背后的故事”，我总结了 3 个最具代表性和区分度的问题。

在开始正式讲解之前，我希望你能在自己心中尝试回答这 3 个问题：

- JSX 的本质是什么，它和 JS 之间到底是什么关系？

- 为什么要用 JSX? 不用会有什么后果?
- JSX 背后的功能模块是什么, 这个功能模块都做了哪些事情?

面对以上问题, 如果你无法形成清晰且系统的思路, 那么很可能是你把 JSX 想得过于简单了。大多数人只是简单地把它理解为模板语法的一种, 但事实上, JSX 作为 React 框架的一大特色, 它与 React 本身的运作机制之间存在着千丝万缕的联系。

上述 3 个问题的答案, 就恰恰隐藏在这层“联系”中, 在面试场景下, 候选人对这层“联系”吃得透不透, 是我们评价其在 React 方面是否“资深”的一个重要依据。

接下来, 我就将带你由表及里地起底 JSX 相关的底层原理, 帮助你吃透这层“联系”, 建立起强大的理论自信。你可以将“能够用自己的话回答上面 3 个问题”来作为本课时的学习目标, 待课时结束后, 记得回来检验自己的学习成果^_^。

JSX 的本质: JavaScript 的语法扩展

JSX 到底是什么, 我们先来看看 [React 官网](#)给出的一段定义:

JSX 是 JavaScript 的一种语法扩展, 它和模板语言很接近, 但是它充分具备 JavaScript 的能力。

“语法扩展”这一点在理解上几乎不会产生歧义, 不过“它充分具备 JavaScript 的能力”这句, 却总让人摸不着头脑, JSX 和 JS 怎么看也不像是一路人啊? 这就引出了“**JSX 语法是如何在 JavaScript 中生效的**”这个问题。

JSX 语法是如何在 JavaScript 中生效的: 认识 Babel

Facebook 公司给 JSX 的定位是 JavaScript 的“扩展”, 而非 JavaScript 的“某个版本”, 这就直接决定了浏览器并不会像天然支持 JavaScript 一样地支持 JSX。那么, JSX 的语法是如何在 JavaScript 中生效的呢? [React 官网](#)其实早已给过我们线索:

JSX 会被编译为 `React.createElement()`, `React.createElement()` 将返回一个叫作“React Element”的 JS 对象。

这里提到, JSX 在被编译后, 会变成一个针对 `React.createElement` 的调用, 此时你大可不必急于关注 `React.createElement` 这个 API 到底做了什么 (下文会单独讲解)。咱们先来说说这个“编译”是怎么回事: “编译”这个动作, 是由 Babel 来完成的。

什么是 Babel 呢?

Babel 是一个工具链，主要用于将 ECMAScript 2015+ 版本的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中。

—— Babel 官网

比如说，ES2015+ 版本推出了一种名为“模板字符串”的新语法，这种语法在一些低版本的浏览器里并不兼容。下面是一段模板字符串的示例代码：

```
1. var name = "Guy Fieri";
2. var place = "Flavortown";
3. `Hello ${name}, ready for ${place}?`;
```

■ 复制代码

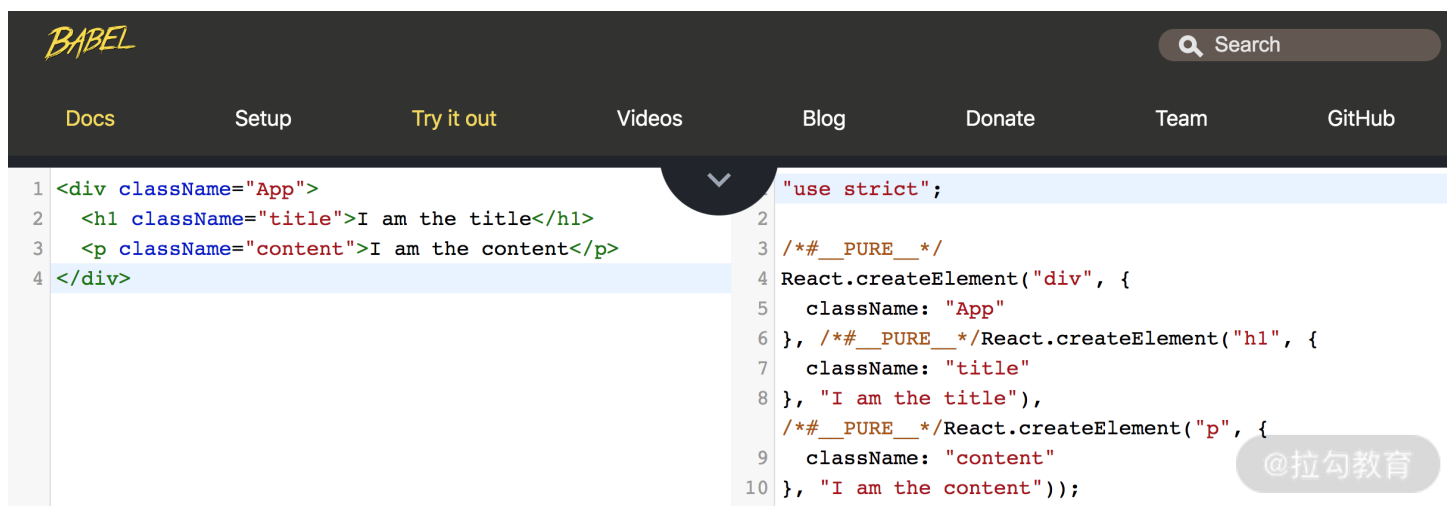
Babel 就可以帮我们把这段代码转换为大部分低版本浏览器也能够识别的 ES5 代码：

```
1. var name = "Guy Fieri";
2. var place = "Flavortown";
3. "Hello ".concat(name, ", ready for ").concat(place, "?");
```

■ 复制代码

类似的，**Babel 也具备将 JSX 语法转换为 JavaScript 代码的能力。**

那么 Babel 具体会将 JSX 处理成什么样子呢？我们不如直接打开 Babel 的 playground 来看一看。这里我仍然键入文章开头示例代码中的 JSX 部分：



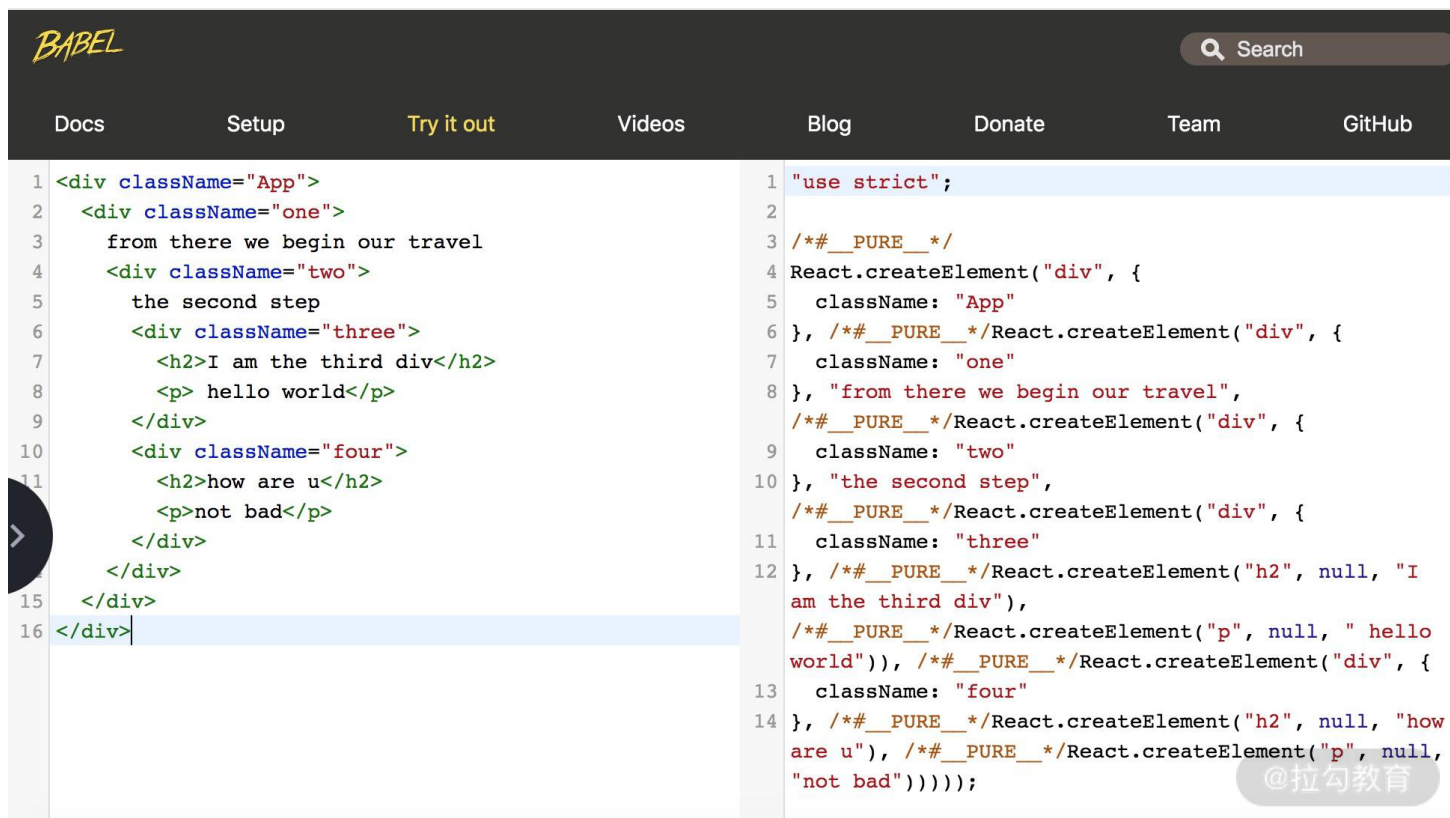
The screenshot shows the Babel Playground interface. On the left, the input JSX code is: `<div className="App">
 <h1 className="title">I am the title</h1>
 <p className="content">I am the content</p>
</div>`. On the right, the output JavaScript code is: `"use strict";
/**__PURE__*/
React.createElement("div", {
 className: "App"
}, /**__PURE__*/React.createElement("h1", {
 className: "title"
}, "I am the title"),
/**__PURE__*/React.createElement("p", {
 className: "content"
}, "I am the content"));`. The Babel logo is in the top left, and a search bar is in the top right. Navigation links (Docs, Setup, Try it out, Videos, Blog, Donate, Team, GitHub) are in the middle. A watermark "@拉勾教育" is in the bottom right.

可以看到，所有的 JSX 标签都被转化成了 `React.createElement` 调用，这也就意味着，我们写的 JSX 其实写的就是 `React.createElement`，虽然它看起来有点像 HTML，但也只是“看起来像”而已。**JSX 的本质是 `React.createElement` 这个 JavaScript 调用的语法糖**，这也就完美地呼应上了 React 官方给出的“**JSX 充分具备 JavaScript 的能力**”这句话。

React 选用 JSX 语法的动机

换个角度想想，既然 JSX 等价于一次 `React.createElement` 调用，那么 React 官方为什么不直接引导我们用 `React.createElement` 来创建元素呢？

原因非常简单，我们来看一个相对复杂一些的组件的 JSX 代码和 `React.createElement` 调用之间的对比。它们各自的形态如下图所示，图中左侧是 JSX 代码，右侧是 `React.createElement` 调用：



The image shows a side-by-side comparison of JSX code and its equivalent `React.createElement` calls. The left side displays JSX code, which is more readable and structured, while the right side shows the corresponding JavaScript code, which is more verbose and nested. The JSX code is highlighted in blue, and the JavaScript code is highlighted in yellow. The JSX code is as follows:

```
1 <div className="App">
2   <div className="one">
3     from there we begin our travel
4     <div className="two">
5       the second step
6       <div className="three">
7         <h2>I am the third div</h2>
8         <p>hello world</p>
9       </div>
10      <div className="four">
11        <h2>how are u</h2>
12        <p>not bad</p>
13      </div>
14    </div>
15  </div>
16 </div>
```

The JavaScript code is as follows:

```
1 "use strict";
2
3 /**__PURE__*/
4 React.createElement("div", {
5   className: "App"
6 }, /**__PURE__*/React.createElement("div", {
7   className: "one"
8 }, "from there we begin our travel",
9 /**__PURE__*/React.createElement("div", {
10   className: "two"
11 }, "the second step",
12 /**__PURE__*/React.createElement("div", {
13   className: "three"
14 }, /**__PURE__*/React.createElement("h2", null, "I
15 am the third div"),
16 /**__PURE__*/React.createElement("p", null, "hello
17 world")), /**__PURE__*/React.createElement("div", {
18   className: "four"
19 }, /**__PURE__*/React.createElement("h2", null, "how
20 are u"), /**__PURE__*/React.createElement("p", null,
21 "not bad"))));
```

你会发现，在实际功能效果一致的前提下，JSX 代码层次分明、嵌套关系清晰；而 `React.createElement` 代码则给人一种非常混乱的“杂糅感”，这样的代码不仅读起来不友好，写起来也费劲。

JSX 语法糖允许前端开发者使用我们最为熟悉的类 HTML 标签语法来创建虚拟 DOM，在降低学习成本的同时，也提升了研发效率与研发体验。

读到这里，相信你已经充分理解了“**JSX 是 JavaScript 的一种语法扩展，它和模板语言很接近，但是它充分具备 JavaScript 的能力。**”这一定义背后的深意。那么我们文中反复提及的 `React.createElement` 又是何方神圣呢？下面我们就深入相关源码来一窥究竟。

JSX 是如何映射为 DOM 的：起底 `createElement` 源码

在分析开始之前，你可以先尝试阅读我追加进源码中的逐行代码解析，大致理解 `createElement` 中每一行代码的作用：

```
1. /**
2.  101. React的创建元素方法
3.  */
4. export function createElement(type, config, children) {
5.  // propName 变量用于储存后面需要用到的元素属性
6.  let propName;
7.  // props 变量用于储存元素属性的键值对集合
8.  const props = {};
9.  // key、ref、self、source 均为 React 元素的属性，此处不必深究
10. let key = null;
11. let ref = null;
12. let self = null;
13. let source = null;
14.
15. // config 对象中存储的是元素的属性
16. if (config !== null) {
17.   // 进来之后做的第一件事，是依次对 ref、key、self 和 source 属性赋值
18.   if (hasValidRef(config)) {
19.     ref = config.ref;
20.   }
21.   // 此处将 key 值字符串化
22.   if (hasValidKey(config)) {
23.     key = '' + config.key;
24.   }
25.   self = config.__self === undefined ? null : config.__self;
26.   source = config.__source === undefined ? null : config.__source;
27.   // 接着就是要把 config 里面的属性都一个一个挪到 props 这个之前声明好的对象里面
28.   for (propName in config) {
29.     if (
30.       // 筛选出可以提进 props 对象里的属性
31.       hasOwnProperty.call(config, propName) &&
32.       !RESERVED_PROPS.hasOwnProperty(propName)
33.     ) {
34.       props[propName] = config[propName];
35.     }
36.   }
37. }
38. // childrenLength 指的是当前元素的子元素的个数，减去的 2 是 type 和 config 两个参数占用
39. const childrenLength = arguments.length - 2;
40. // 如果抛去type和config，就只剩下一个参数，一般意味着文本节点出现了
41. if (childrenLength === 1) {
42.   // 直接把这个参数的值赋给props.children
43.   props.children = children;
44.   // 处理嵌套多个子元素的情况
45. } else if (childrenLength > 1) {
46.   // 声明一个子元素数组
47.   const childArray = Array(childrenLength);
48.   // 把子元素推进数组里
49.   for (let i = 0; i < childrenLength; i++) {
50.     childArray[i] = arguments[i + 2];
51.   }
52.   // 最后把这个数组赋值给props.children
53.   props.children = childArray;
54. }
55.
56. // 处理 defaultProps
57. if (type && type.defaultProps) {
```

```
58.     const defaultProps = type.defaultProps;
59.     for (propName in defaultProps) {
60.         if (props[propName] === undefined) {
61.             props[propName] = defaultProps[propName];
62.         }
63.     }
64. }
65.
66. // 最后返回一个调用ReactDOM执行方法，并传入刚才处理过的参数
67. return ReactDOM(
68.     type,
69.     key,
70.     ref,
71.     self,
72.     source,
73.     ReactCurrentOwner.current,
74.     props,
75. );
76. }
```

上面是对源码细节的初步展示，接下来我会带你逐步提取源码中的关键知识点和核心思想。

入参解读：创建一个元素需要知道哪些信息

我们先来看看方法的入参：

```
1. export function createElement(type, config, children)
```

■ 复制代码

createElement 有 3 个入参，这 3 个入参囊括了 React 创建一个元素所需要知道的全部信息。

- type：用于标识节点的类型。它可以是类似“h1”“div”这样的标准 HTML 标签字符串，也可以是 React 组件类型或 React fragment 类型。
- config：以对象形式传入，组件所有的属性都会以键值对的形式存储在 config 对象中。
- children：以对象形式传入，它记录的是组件标签之间嵌套的内容，也就是所谓的“子节点”“子元素”。

如果文字描述使你觉得抽象，下面这个调用示例可以帮你增进对概念的理解：

```
1. React.createElement("ul", {
2.   // 传入属性键值对
3.   className: "list"
4.   // 从第三个入参开始往后，传入的参数都是 children
5. }, React.createElement("li", {
6.   key: "1"
7. }, "1"), React.createElement("li", {
8.   key: "2"
```

■ 复制代码


```
9. }, "2"));
```

这个调用对应的 DOM 结构如下：

```
1. <ul className="list">
2.   <li key="1">1</li>
3.   <li key="2">2</li>
4. </ul>
```

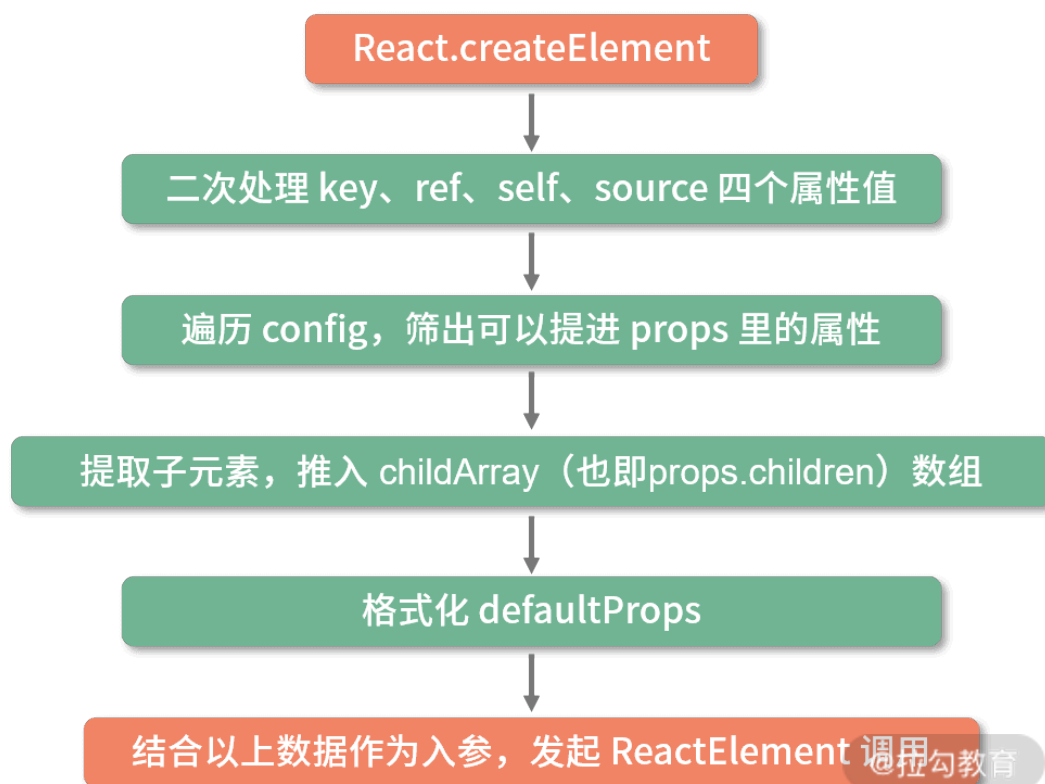
[复制代码](#)

对入参的形式和内容有了大致的把握之后，下面我们继续来讲解 createElement 的函数逻辑。

createElement 函数体拆解

前面你已经阅读过 createElement 源码细化到每一行的解读，这里我想和你探讨的是 createElement 在逻辑层面的任务流转。针对这个过程，我为你总结了下面这张流程图：

入口

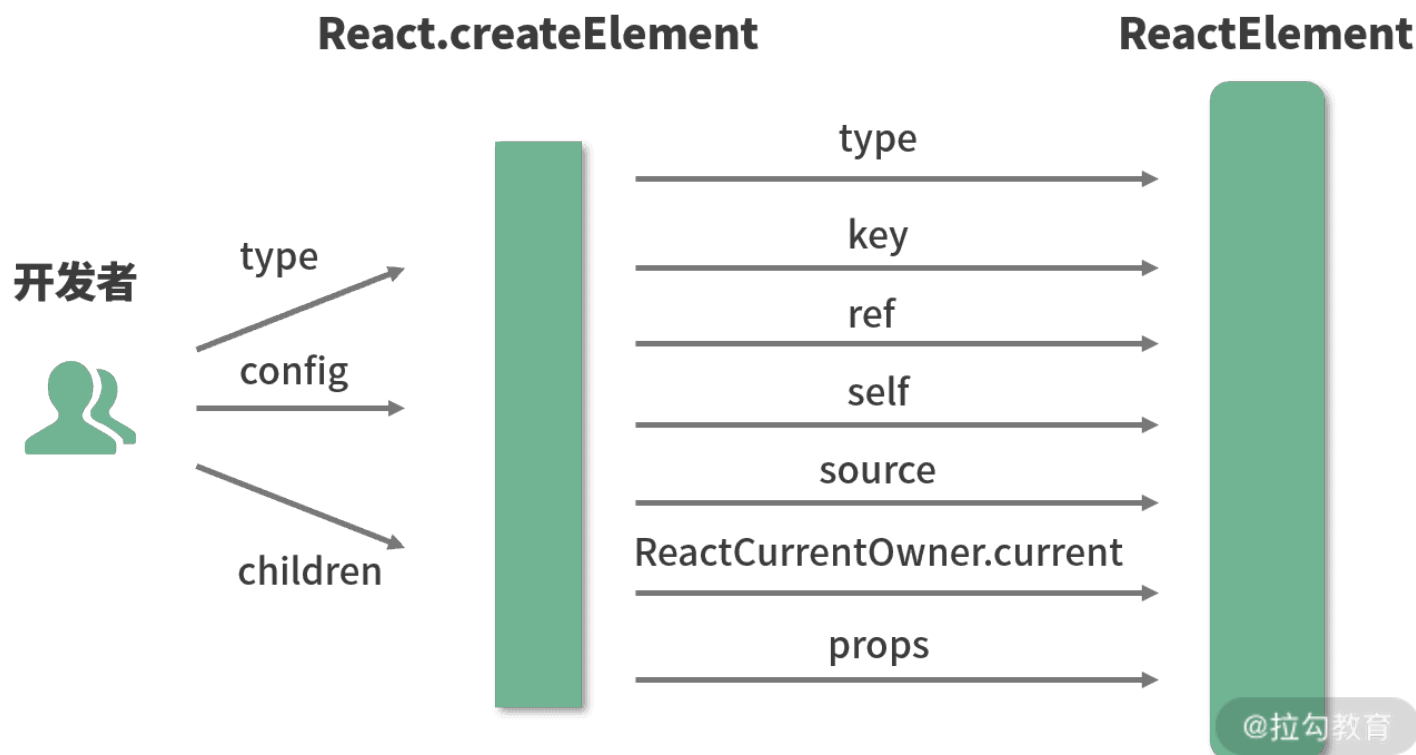


落脚点

这个流程图，或许会打破不少同学对 createElement 的幻想。在实际的面试场景下，许多候选人由于缺乏对源码的了解，谈及 createElement 时总会倾向于去夸大它的“工作量”。但其实，相信你也已经发现了，createElement 中并没有十分复杂的涉及算法或真实 DOM 的逻辑，它的每一个步骤几乎都是在格式化数据。

说得更直白点，createElement 就像是开发者和 ReactElement 调用之间的一个“转换器”、一个数据处理层。它可以从开发者处接受相对简单的参数，然后将这些参数按照 ReactElement 的预期做一层格式

化，最终通过调用 `ReactElement` 来实现元素的创建。整个过程如下图所示：



现在看来，`createElement` 原来只是个“参数中介”。此时我们的注意力自然而然地就聚焦在了 `ReactElement` 上，接下来我们就乘胜追击，一起去挖一挖 `ReactElement` 的源码吧！

出参解读：初识虚拟 DOM

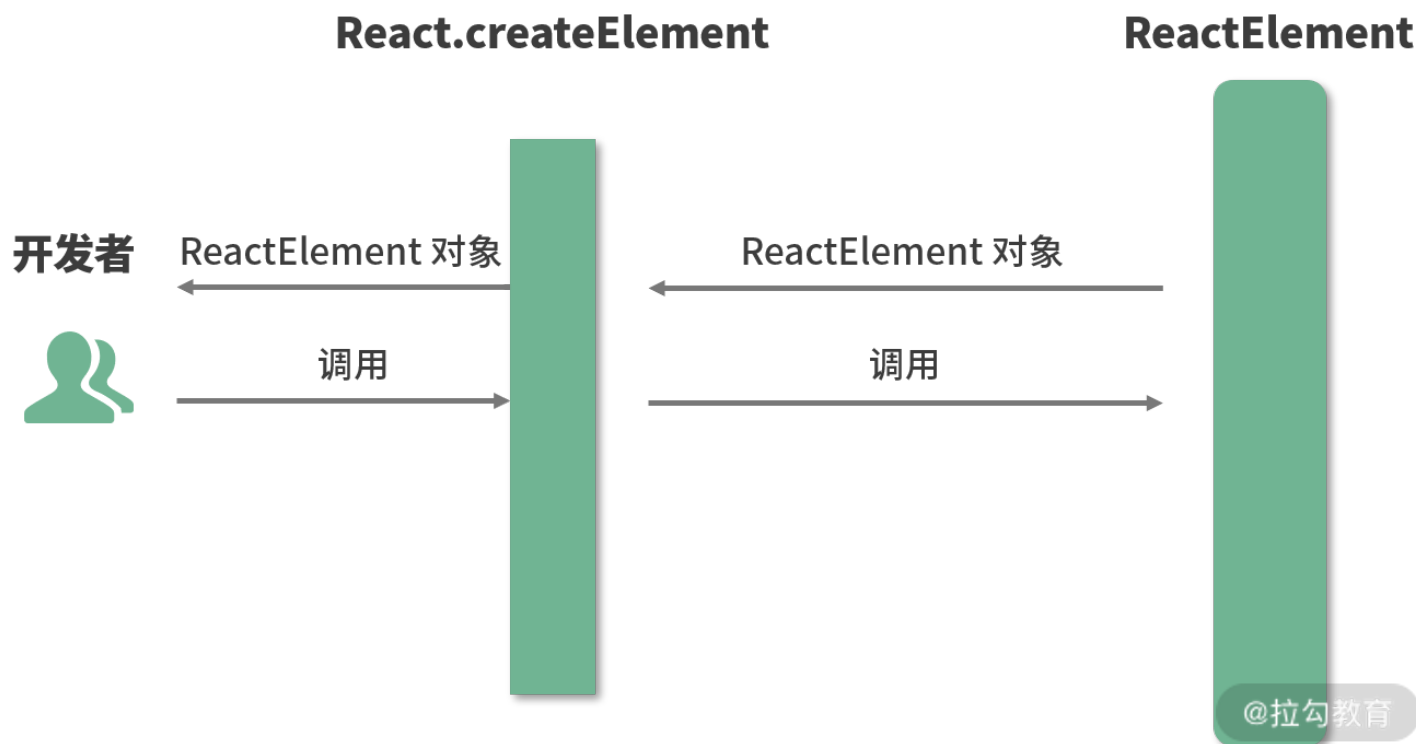
上面已经分析过，`createElement` 执行到最后会 `return` 一个针对 `ReactElement` 的调用。这里关于 `ReactElement`，我依然先给出源码 + 注释形式的解析：

```
1. const ReactElement = function(type, key, ref, self, source, owner, props) {  
2.   const element = {  
3.     // REACT_ELEMENT_TYPE是一个常量，用来标识该对象是一个ReactElement  
4.     $$typeof: REACT_ELEMENT_TYPE,  
5.  
6.     // 内置属性赋值  
7.     type: type,  
8.     key: key,  
9.     ref: ref,  
10.    props: props,  
11.  
12.    // 记录创造该元素的组件  
13.    _owner: owner,  
14.  };  
15.  
16.  //  
17.  if (___DEV___) {  
18.    // 这里是一些针对 ___DEV___ 环境下的处理，对于大家理解主要逻辑意义不大，此处我直接省略掉，  
19.  }
```

复制代码


```
20.  
21.   return element;  
22. };
```

ReactElement 的代码出乎意料的简短，从逻辑上我们可以看出，ReactElement 其实只做了一件事情，那就是“创建”，说得更精确一点，是“组装”：ReactElement 把传入的参数按照一定的规范，“组装”进了 element 对象里，并把它返回给了 React.createElement，最终 React.createElement 又把它交回到了开发者手中。整个过程如下图所示：



如果你想要验证这一点，可以尝试输出我们示例中 App 组件的 JSX 部分：

```
1. const AppJSX = (<div className="App">  
2.   <h1 className="title">I am the title</h1>  
3.   <p className="content">I am the content</p>  
4. </div>  
5.  
6. console.log(AppJSX)
```

■ 复制代码

你会发现它确实是一个标准的 ReactElement 对象实例，如下图（生产环境下的输出结果）所示：

```

▼ {$$typeof: Symbol(react.element), type: "div", key: null,
  ref: null, props: {...}, ...} ⓘ
  $$typeof: Symbol(react.element)
  key: null
  ▼ props:
    ▼ children: Array(2)
      ► 0: {$$typeof: Symbol(react.element), type: "h1", key...
      ► 1: {$$typeof: Symbol(react.element), type: "p", key:...
        length: 2
      ► __proto__: Array(0)
    className: "App"
    ► __proto__: Object
    ref: null
    type: "div"
    _owner: null
    ► __proto__: Object

```

@拉勾教育

这个 `ReactElement` 对象实例，本质上是以 **JavaScript 对象形式存在的对 DOM 的描述**，也就是老生常谈的“虚拟 DOM”（准确地说，是虚拟 **DOM** 中的一个节点。关于虚拟 DOM，我们将在专栏的“模块二：核心原理”中花大量的篇幅来研究它，此处你只需要能够结合源码，形成初步认知即可）。

既然是“虚拟 DOM”，那就意味着和渲染到页面上的真实 DOM 之间还有一些距离，这个“距离”，就是由大家喜闻乐见的 **ReactDOM.render** 方法来填补的。

在每一个 React 项目的入口文件中，都少不了对 `React.render` 函数的调用。下面我简单介绍下 `ReactDOM.render` 方法的入参规则：

```

1. ReactDOM.render(
2.   // 需要渲染的元素 (ReactElement)
3.   element,
4.   // 元素挂载的目标容器 (一个真实DOM)
5.   container,
6.   // 回调函数，可选参数，可以用来处理渲染结束后的逻辑
7.   [callback]
8. )

```

■ 复制代码

ReactDOM.render 方法可以接收 3 个参数，其中**第二个参数就是一个真实的 DOM 节点**，这个真实的 **DOM 节点充当“容器”的角色**，React 元素最终会被渲染到这个“容器”里面去。比如，示例中的 App 组件，它对应的 render 调用是这样的：

```
1. const rootElement = document.getElementById("root");  
2. ReactDOM.render(<App />, rootElement);
```

[复制代码](#)

注意，这个真实 DOM 一定是确实存在的。比如，在 App 组件对应的 index.html 中，已经提前预置了 id 为 root 的根节点：

```
1. <body>  
2.   <div id="root"></div>  
3. </body>
```

[复制代码](#)