React 进阶实践指南 - 我不是外星人 - 掘金课程

juejin.cn/book/6945998773818490884/section/6948337148202319908

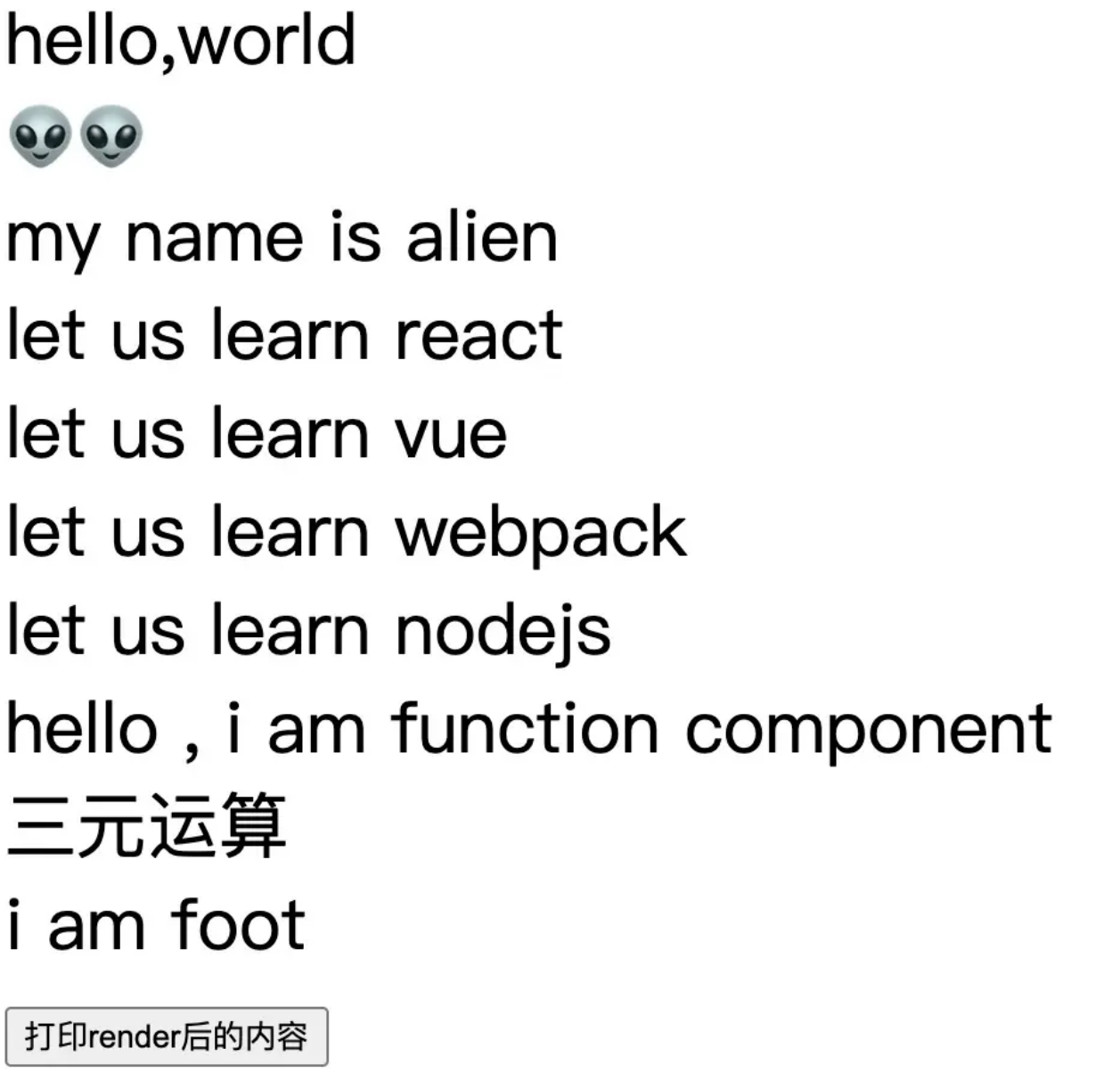
一 我们写的 JSX 终将变成什么

万物始于 `jsx` ，想要深入学习 react ，就应该从 jsx 入手。弄清楚 jsx ，方便学习掌握以下内容：

- 了解常用的元素会被 React 处理成什么，有利于后续理解 react fiber 类型；
- 理解 jsx 的编译过程，方便操纵 children、控制 React 渲染，有利于便捷使用 React 插槽组件。

我写了一段 react JSX 代码，接下来，我们一步步看看它最后会变成什么样子。

```
const toLearn = [ 'react' , 'vue' , 'webpack' , 'nodejs'  ]

const TextComponent = ()=> <div> hello , i am function component </div>

class Index extends React.Component{
    status = false /* 状态 */
    renderFoot=()=> <div> i am foot</div>
    render(){
        /* 以下都是常用的jsx元素节 */
        return <div style={{ marginTop:'100px' }}   >
            { /* element 元素类型 */ }
            <div>hello,world</div>
            { /* fragment 类型 */ }
            <React.Fragment>
                <div> 👽👽 </div>
            </React.Fragment>
            { /* text 文本类型 */ }
            my name is alien
            { /* 数组节点类型 */ }
            { toLearn.map(item=> <div key={item} >let us learn { item } </div> ) }
            { /* 组件类型 */ }
            <TextComponent/>
            { /* 三元运算 */  }
            { this.status ? <TextComponent /> : <div>三元运算</div> }
            { /* 函数执行 */ }
            { this.renderFoot() }
            <button onClick={ ()=> console.log( this.render() ) } >打印render后的内容
</button>
        </div>
    }
}
```



1 babel 处理后的样子

首先，看一下上述例子中的 **jsx** 模版会被**babel**编译成什么？

和如上看到的一样，我写的 JSX 元素节点会被编译成 React Element 形式。那么，我们首先来看一下 React.createElement 的用法。

```
React.createElement(
  type,
  [props],
  [...children]
)
```

`createElement` 参数：

- 第一个参数：如果是组件类型，会传入组件对应的类或函数；如果是 dom 元素类型，传入 div 或者 span 之类的字符串。
- 第二个参数：一个对象，在 dom 类型中为标签属性，在组件类型中为 props 。
- 其他参数：依次为 children，根据顺序排列。

举个例子：

```
<div>
   <TextComponent />
   <div>hello,world</div>
   let us learn React!
</div>
```

上面的代码会被 babel 先编译成：

```
 React.createElement("div", null,
        React.createElement(TextComponent, null),
        React.createElement("div", null, "hello,world"),
        "let us learn React!"
    )
```

| --------问与答-------- |

问：老版本的 React 中，为什么写 jsx 的文件要默认引入 React?
如下

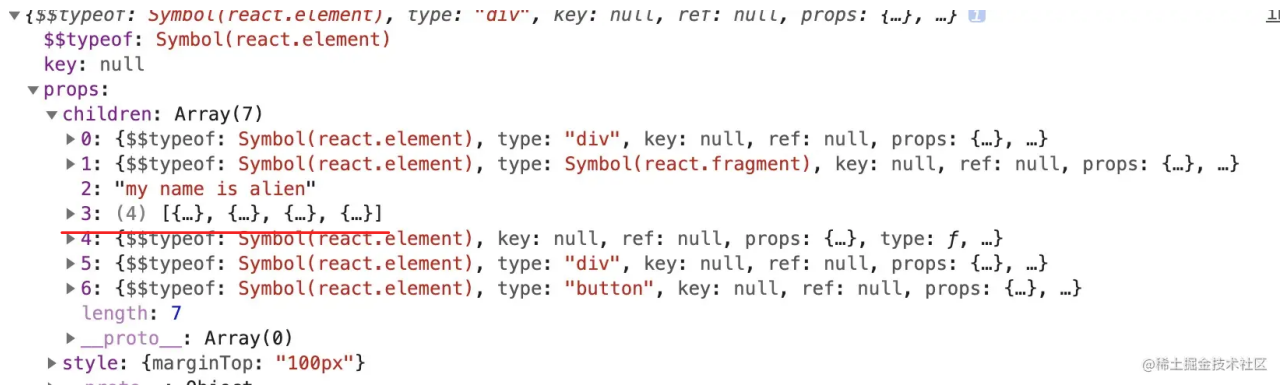
```
import React from 'react'
function Index(){
    return <div>hello,world</div>
}
```

答：因为 jsx 在被 babel 编译后，写的 jsx 会变成上述 React.createElement 形式，所以需要引入 React，防止找不到 React 引起报错。

| ---------end--------- |

2 createElement 处理后的样子

然后点击按钮，看一下写的 demo 会被 React.createElement 变成什么:



从上面写的 jsx 结构来看，外层的 div 被 react.createElement 转换成 react element 对象，div 里面的 8 个元素分别转换成 children 子元素列表。下面就是 jsx 的转换规则，请一定要记住，以便后续能更流畅地使用 jsx 语法。

jsx 元素类型	react.createElement 转换后	type 属性
element 元素类型	react element 类型	标签字符串，例如 div
fragment 类型	react element 类型	symbol react.fragment 类型
文本类型	直接字符串	无
数组类型	返回数组结构，里面元素被 react.createElement 转换	无
组件类型	react element 类型	组件类或者组件函数本身
三元运算 / 表达式	先执行三元运算，然后按照上述规则处理	看三元运算返回结果
函数执行	先执行函数，然后按照上述规则处理	看函数执行返回结果

3 React 底层调和处理后，终将变成什么?

最终，在调和阶段，上述 React element 对象的每一个子节点都会形成一个与之对应的 fiber 对象，然后通过 sibling、return、child 将每一个 fiber 对象联系起来。

所以，我们有必要先来看一下 React 常用的 fiber 类型，以及 element 对象和 fiber 类型的对应关系。

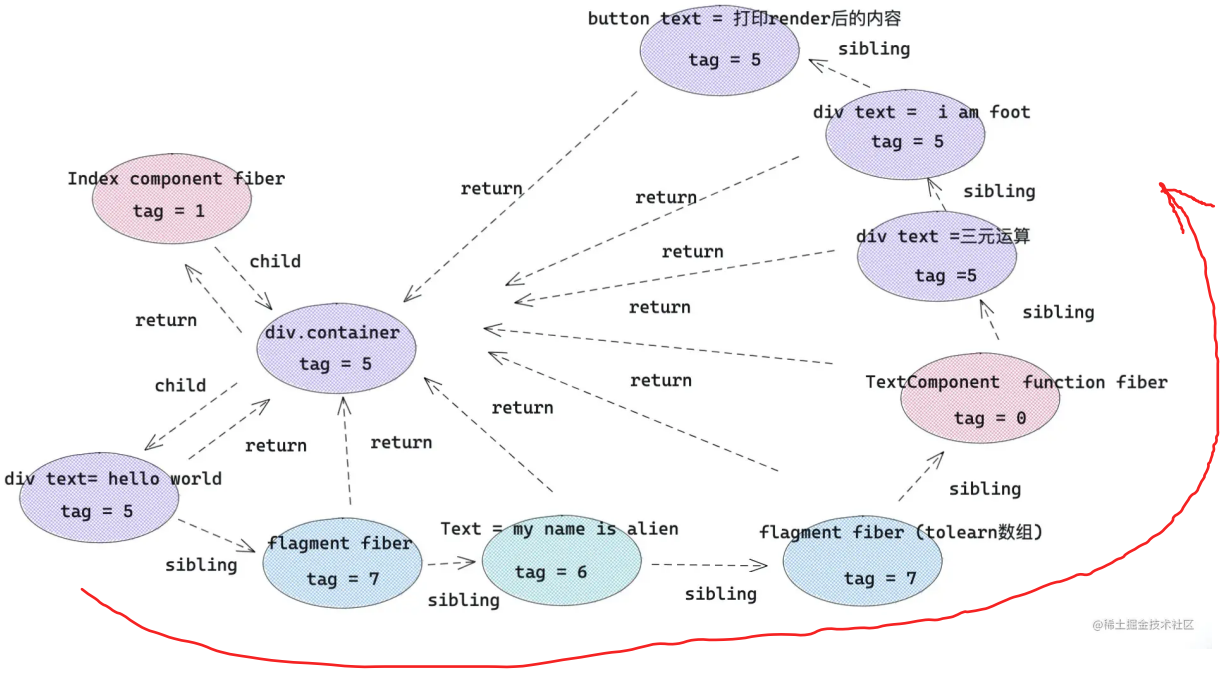
不同种类的 fiber Tag

React 针对不同 React element 对象会产生不同 tag (种类) 的fiber 对象。首先，来看一下 tag 与 element 的对应关系：

```
export const FunctionComponent = 0;        // 函数组件
export const ClassComponent = 1;           // 类组件
export const IndeterminateComponent = 2;   // 初始化的时候不知道是函数组件还是类组件
export const HostRoot = 3;                 // Root Fiber 可以理解为根元素 ， 通过
reactDom.render()产生的根元素
export const HostPortal = 4;               // 对应  ReactDOM.createPortal 产生的 Portal
export const HostComponent = 5;            // dom 元素 比如 <div>
export const HostText = 6;                 // 文本节点
export const Fragment = 7;                 // 对应 <React.Fragment>
export const Mode = 8;                     // 对应 <React.StrictMode>
export const ContextConsumer = 9;          // 对应 <Context.Consumer>
export const ContextProvider = 10;         // 对应 <Context.Provider>
export const ForwardRef = 11;              // 对应 React.ForwardRef
export const Profiler = 12;                // 对应 <Profiler/ >
export const SuspenseComponent = 13;       // 对应 <Suspense>
export const MemoComponent = 14;           // 对应 React.memo 返回的组件
```

jsx 最终形成的 fiber 结构图

最终写的 jsx 会变成如下格式：



fiber 对应关系

- child： 一个由父级 fiber 指向子级 fiber 的指针。
- return：一个子级 fiber 指向父级 fiber 的指针。
- sibling: 一个 fiber 指向下一个兄弟 fiber 的指针。

温馨提示：

- 对于上述在 jsx 中写的 map 数组结构的子节点，外层会被加上 fragment ；
- map 返回数组结构，作为 fragment 的子节点。

二 进阶实践-可控性 render

上面的 demo 暴露出了如下问题：

1. 返回的 `children` 虽然是一个数组，但是数组里面的数据类型却是不确定的，有对象类型(如 `ReactElement`)，有数组类型(如 `map` 遍历返回的子节点)，还有字符串类型(如文本)；
2. 无法对 render 后的 React element 元素进行可控性操作。

针对上述问题，我们需要对demo项目进行改造处理，具体过程可以分为4步：

1. 将上述children扁平化处理，将数组类型的子节点打开；
2. 干掉children中文本类型节点；
3. 向children最后插入
   say goodbye

   元素；
4. 克隆新的元素节点并渲染。

希望通过这个实践 demo，大家可以加深对 **jsx** 编译后结构的认识，学会对 **jsx** 编译后的 **React.element** 进行一系列操作，达到理想化的目的，以及熟悉 **React API** 的使用。

由于，我们想要把 render 过程变成可控的，因此需要把上述代码进行改造。

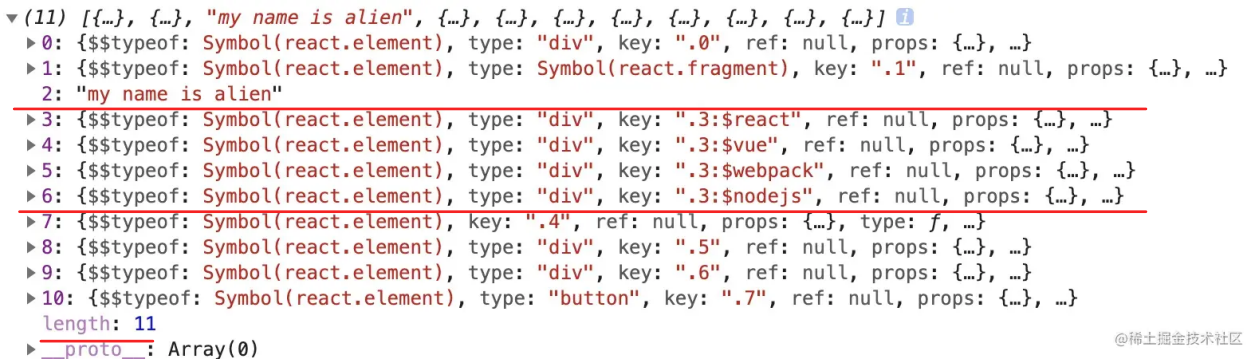
```
class Index extends React.Component{
    status = false /* 状态 */
    renderFoot=()=> <div> i am foot</div>
    /* 控制渲染 */
    controlRender=()=>{
        const reactElement = (
            <div style={{ marginTop:'100px' }} className="container"  >
                 { /* element 元素类型 */ }
                <div>hello,world</div>
                { /* fragment 类型 */ }
                <React.Fragment>
                    <div> 👽👽 </div>
                </React.Fragment>
                { /* text 文本类型 */ }
                my name is alien
                { /* 数组节点类型 */ }
                { toLearn.map(item=> <div key={item} >let us learn { item } </div> ) }
                { /* 组件类型 */ }
                <TextComponent/>
                { /* 三元运算 */  }
                { this.status ? <TextComponent /> :  <div>三元运算</div> }
                { /* 函数执行 */ }
                { this.renderFoot() }
                <button onClick={ ()=> console.log( this.render() ) } >打印render后的内容
</button>
            </div>
        )
        console.log(reactElement)
        const { children } = reactElement.props
        /* 第1步 ： 扁平化 children  */
        const flatChildren = React.Children.toArray(children)
        console.log(flatChildren)
        /* 第2步 ： 除去文本节点 */
        const newChildren :any= []
        React.Children.forEach(flatChildren,(item)=>{
            if(React.isValidElement(item)) newChildren.push(item)
        })
        /* 第3步，插入新的节点 */
        const lastChildren = React.createElement(`div`,{ className :'last' } ,`say
goodbye`)
        newChildren.push(lastChildren)

        /* 第4步：修改容器节点 */
        const newReactElement =  React.cloneElement(reactElement,{} ,...newChildren )
        return newReactElement
    }
    render(){
        return this.controlRender()
    }
}
```

第 **1** 步： `React.Children.toArray` 扁平化，规范化 **children** 数组。

```
const flatChildren = React.Children.toArray(children)
console.log(flatChildren)
```

React.Children.toArray 可以扁平化、规范化 React.element 的 children 组成的数组，只要 children 中的数组元素被打开，对遍历 children 很有帮助，而且 React.Children.toArray 还可以深层次 flat 。

打印结果：



第 **2** 步：遍历 **children** ，验证 **React.element** 元素节点，除去文本节点。

```
const newChildren :any= []
React.Children.forEach(flatChildren,(item)=>{
    if(React.isValidElement(item)) newChildren.push(item)
})
```

用 React.Children.forEach 去遍历子节点，如果是 react Element 元素，就添加到新的 children 数组中，通过这种方式过滤掉非 React element 节点。React.isValidElement 这个方法可以用来检测是否为 React element 元素，接收一个参数——待验证对象，如果是返回 true ， 否则返回 false 。

这里可能会有一个疑问就是如下：

难道用数组本身方法 filter 过滤不行么 ？ 为什么要用 React.Children.forEach 遍历？

这种情况下，是完全可以用数组方法过滤的，因为 React.Children.toArray 已经处理了 children ，使它变成了正常的数组结构 也就是说 `React.Children.forEach = React.Children.toArray + Array.prototype.forEach` 。

React.Children.forEach 本身就可以把 children 扁平化了，也就是上述第一步操作多此一举了。为什么要有第一步，主要是更多的学习一下 React api。

第 **3** 步：用 **React.createElement** ，插入到 **children** 最后

```
 /* 第三步，插入新的节点 */
const lastChildren = React.createElement(`div`,{ className :'last' } ,`say goodbye`)
newChildren.push(lastChildren)
```

上述代码实际等于用 `JSX` 这么写：

```
newChildren.push(<div className="last" >say goodbye</div>)
```

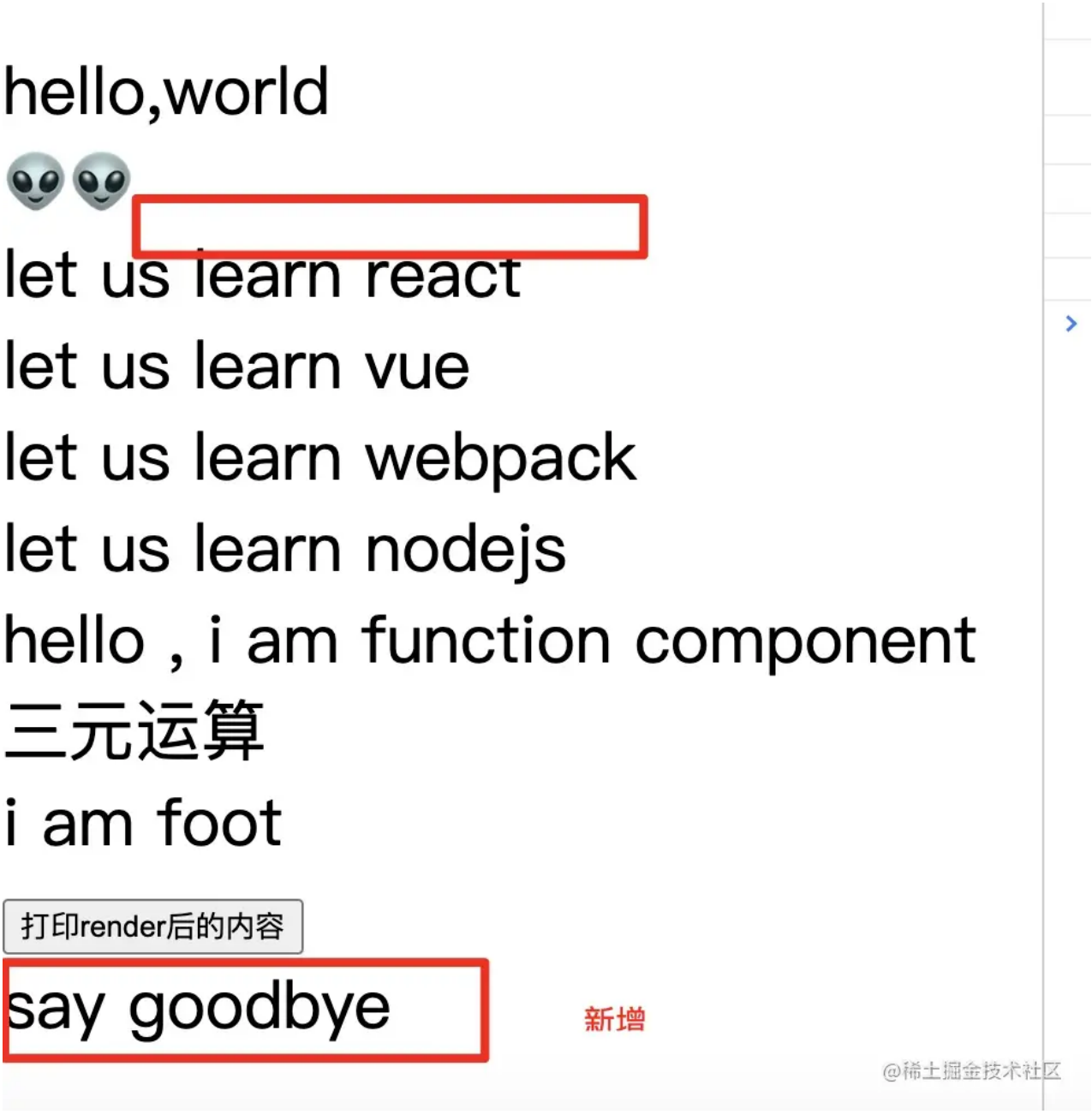
第 4 步：已经修改了 **children**，现在做的是，通过 **cloneElement** 创建新的容器元素。

为什么要用 React.cloneElement ， createElement 把上面写的 jsx，变成 element 对象; 而 cloneElement 的作用是以 element 元素为样板克隆并返回新的 React element 元素。返回元素的 props 是将新的 props 与原始元素的 props 浅层合并后的结果。

这里 React.cloneElement 做的事情就是，把 reactElement 复制一份，再用新的 children 属性，从而达到改变 render 结果的目的。

```
/* 第 4 步：修改容器节点 */
const newReactElement =  React.cloneElement(reactElement,{} ,...newChildren )
```

效果



验证：

- ① children 已经被扁平化。
- ② 文本节点 `my name is alien` 已经被删除。
- ③ `<div className="last" > say goodbye</div>` 元素成功插入。

达到了预期效果。

| --------问与答-------- |

问: React.createElement 和 React.cloneElement 到底有什么区别呢?

答: 可以完全理解为，一个是用来创建 element 。另一个是用来修改 element，并返回一个新的 React.element 对象。

| ---------end--------- |

三、Babel 解析 JSX 流程

1 @babel/plugin-syntax-jsx 和 @babel/plugin-transform-react-jsx

JSX 语法实现来源于这两个 babel 插件：

- @babel/plugin-syntax-jsx： 使用这个插件，能够让 Babel 有效的解析 JSX 语法。
- @babel/plugin-transform-react-jsx： 这个插件内部调用了 @babel/plugin-syntax-jsx，可以把 React JSX 转化成 JS 能够识别的 createElement 格式。

Automatic Runtime

新版本 React 已经不需要引入 createElement， 这种模式来源于 `Automatic Runtime` ，看一下是如何编译的。

业务代码中写的 JSX 文件：

```
function Index(){
    return <div>
        <h1>hello,world</h1>
        <span>let us learn React</span>
    </div>
}
```

被编译后的文件：

```
import { jsx as _jsx } from "react/jsx-runtime";
import { jsxs as _jsxs } from "react/jsx-runtime";
function Index() {
  return  _jsxs("div", {
            children: [
                _jsx("h1", {
                   children: "hello,world"
                }),
                _jsx("span", {
                    children:"let us learn React" ,
                }),
            ],
        });
}
```

plugin-syntax-jsx 已经向文件中提前注入了 _jsxRuntime api。不过这种模式下需要我们在 .babelrc 设置 runtime: automatic 。

```
"presets": [
    ["@babel/preset-react",{
    "runtime": "automatic"
    }]
],
```

Classic Runtime

还有一个就是经典模式，在经典模式下，使用 JSX 的文件需要引入 React ，不然就会报错。

业务代码中写的 JSX 文件：

```
import React from 'react'
function Index(){
    return <div>
        <h1>hello,world</h1>
        <span>let us learn React</span>
    </div>
}
```

被编译后的文件：

```
import React from 'react'
function Index(){
    return  React.createElement(
        "div",
        null,
        React.createElement("h1", null,"hello,world"),
        React.createElement("span", null, "let us learn React")
    );
}
```

2 api层面模拟实现

接下来我们通过 api 的方式来模拟一下 Babel 处理 JSX 的流程。

第一步：创建 element.js，写下将测试的 JSX 代码。

```
import React from 'react'

function TestComponent(){
    return <p> hello,React </p>
}
function Index(){
    return <div>
        <span>模拟 babel 处理 jsx 流程。</span>
        <TestComponent />
    </div>
}
export default Index
```

第二步：因为 babel 运行在 node 环境，所以同级目录下创建 jsx.js 文件。来模拟一下编译的效果。

```
const fs = require('fs')
const babel = require("@babel/core")

/* 第一步：模拟读取文件内容。 */
fs.readFile('./element.js',(e,data)=>{
    const code = data.toString('utf-8')
    /* 第二步：转换 jsx 文件 */
    const result = babel.transformSync(code, {
        plugins: ["@babel/plugin-transform-react-jsx"],
    });
    /* 第三步：模拟重新写入内容。 */
    fs.writeFile('./element.js',result.code,function(){})
})
```

如上经过三步处理之后，再来看一下 element.js 变成了什么样子。

```
import React from 'react';

function TestComponent() {
  return /*#__PURE__*/React.createElement("p", null, " hello,React ");
}

function Index() {
  return /*#__PURE__*/React.createElement("div", null,
/*#__PURE__*/React.createElement("span", null, "\u6A21\u62DF babel \u5904\u7406 jsx
\u6D41\u7A0B\u3002"), /*#__PURE__*/React.createElement(TestComponent, null));
}
export default Index;
```

如上可以看到已经成功转成 React.createElement 形式，从根本上弄清楚了 Babel 解析 JSX 的大致流程。

四、总结

本章节主要讲到了两方面的知识。

一方面，我们写的 JSX 会先转换成 React.element，再转化成 React.fiber 的过程。这里要牢牢记住 jsx 转化成 element 的处理逻辑，还有就是 element 类型与转化成 fiber 的 tag 类型的对应关系。这对后续的学习会很有帮助。

另一方面，通过学习第一个实践 demo，我们掌握了如何控制经过 render 之后的 React element 对象。

同时也搞清楚了 Babel 解析 JSX 的大致流程。

下一章节，我们将从React组件角度出发，全方面认识React组件。

案例代码的**github**地址**(点击即可跳转)**

留言

全部评论（109）