```jsx
import React, { Component } from 'react'

export default class App extends Component {
    constructor(props) {
        super(props);

        this.state = {
            message: "Hello World"
        }
    }

    render() {
        return (
            <div>
                <h2>{this.state.message}</h2>
                <button onClick={e => this.changeText()}>面试官系列</button>
            </div>
        )
    }

    changeText() {
        this.setState({
            message: "JS每日一题"
        })
    }
}
```

通过点击按钮触发 `onclick` 事件，执行 `this.setState` 方法更新 `state` 状态，然后重新执行 `render` 函数，从而导致页面的视图更新

如果直接修改 `state` 的状态，如下：

```jsx
changeText() {
    this.state.message = "你好啊,李银河";
}
```

我们会发现页面并不会有任何反应，但是 `state` 的状态是已经发生了改变

这是因为 `React` 并不像 `vue2` 中调用 `Object.defineProperty` 数据响应式或者 `Vue3` 调用 `Proxy` 监听数据的变化

必须通过 `setState` 方法来告知 `react` 组件 `state` 已经发生了改变

关于 `state` 方法的定义是从 `React.Component` 中继承，定义的源码如下：

```JavaScript
Component.prototype.setState = function(partialState, callback) {
  invariant(
    typeof partialState === 'object' ||
      typeof partialState === 'function' ||
      partialState == null,
    'setState(...): takes an object of state variables to update or a ' +
      'function which returns an object of state variables.',
  );
  this.updater.enqueueSetState(this, partialState, callback, 'setState');
};
```

从上面可以看到 `setState` 第一个参数可以是一个对象，或者是一个函数，而第二个参数是一个回调函数，用于可以实时的获取到更新之后的数据

16.2. 更新类型

在使用 `setState` 更新数据的时候， `setState` 的更新类型分成：

- 异步更新
- 同步更新

16.2.1. 异步更新

先举出一个例子：

```JSX
changeText() {
  this.setState({
    message: "你好啊"
  })
  console.log(this.state.message); // Hello World
}
```

从上面可以看到，最终打印结果为 `Hello world` ，并不能在执行完 `setState` 之后立马拿到最新的 `state` 的结果

如果想要立刻获取更新后的值，在第二个参数的回调中更新后会执行

```jsx
changeText() {
  this.setState({
    message: "你好啊"
  }, () => {
    console.log(this.state.message); // 你好啊
  });
}
```

16.2.2. 同步更新

同样先给出一个在 `setTimeout` 中更新的例子：

```jsx
changeText() {
  setTimeout(() => {
    this.setState({
      message: "你好啊
    });
    console.log(this.state.message); // 你好啊
  }, 0);
}
```

上面的例子中，可以看到更新是同步

再来举一个原生 `DOM` 事件的例子：

```jsx
componentDidMount() {
  const btnEl = document.getElementById("btn");
  btnEl.addEventListener('click', () => {
    this.setState({
      message: "你好啊,李银河"
    });
    console.log(this.state.message); // 你好啊,李银河
  })
}
```

16.2.3. 小结

- 在组件生命周期或React合成事件中，setState是异步

- 在setTimeout或者原生dom事件中，setState是同步

16.2.4. 批量更新

同样先给出一个例子：

JSX | 复制代码

```jsx
handleClick = () => {
    this.setState({
        count: this.state.count + 1,
    })
    console.log(this.state.count) // 1

    this.setState({
        count: this.state.count + 1,
    })
    console.log(this.state.count) // 1

    this.setState({
        count: this.state.count + 1,
    })
    console.log(this.state.count) // 1
}
```

点击按钮触发事件，打印的都是 1，页面显示 `count` 的值为 2

对同一个值进行多次 `setState`， `setState` 的批量更新策略会对其进行覆盖，取最后一次的执行结果

上述的例子，实际等价于如下：

JavaScript | 复制代码

```javascript
Object.assign(
  previousState,
  {index: state.count+ 1},
  {index: state.count+ 1},
  ...
)
```

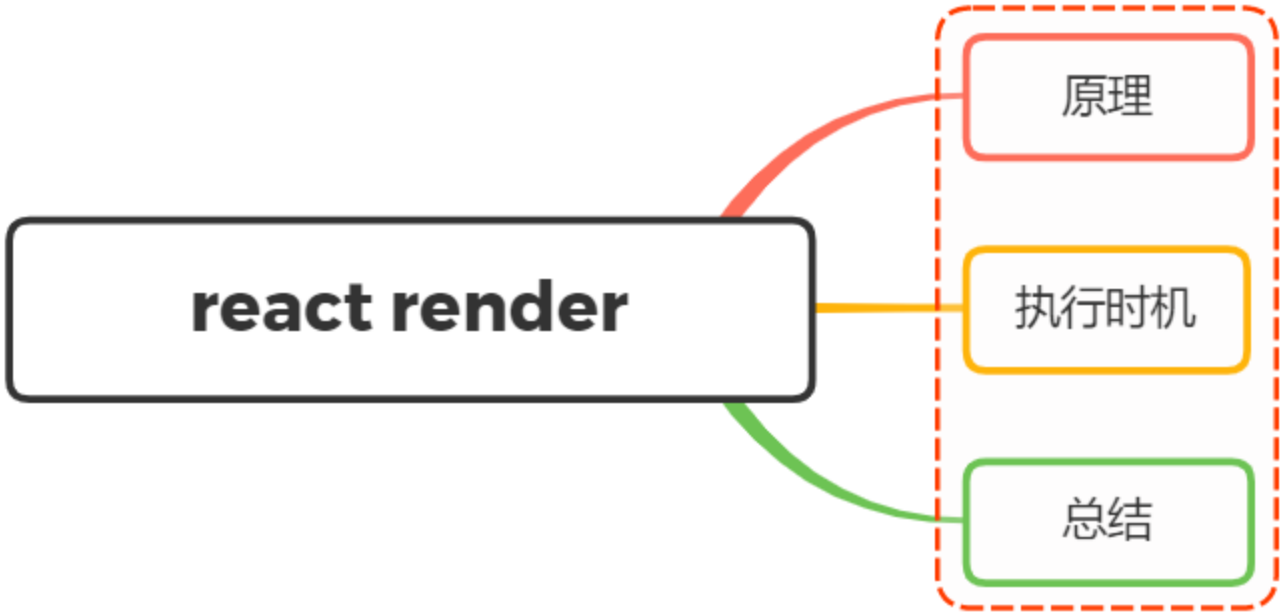
由于后面的数据会覆盖前面的更改，所以最终只加了一次

如果是下一个 `state` 依赖前一个 `state` 的话，推荐给 `setState` 一个参数传入一个 `function`，如下：

```jsx
onClick = () => {
    this.setState((prevState, props) => {
      return {count: prevState.count + 1};
    });
    this.setState((prevState, props) => {
      return {count: prevState.count + 1};
    });
}
```

而在 `setTimeout` 或者原生 `dom` 事件中，由于是同步的操作，所以并不会进行覆盖现象

17. 说说React render方法的原理？在什么时候会被触发？



17.1. 原理

首先， `render` 函数在 `react` 中有两种形式：

在类组件中，指的是 `render` 方法：

```jsx
class Foo extends React.Component {
    render() {
        return <h1> Foo </h1>;
    }
}
```

在函数组件中，指的是函数组件本身：

```javascript
function Foo() {
    return <h1> Foo </h1>;
}
```

在 `render` 中，我们会编写 `jsx`， `jsx` 通过 `babel` 编译后就会转化成我们熟悉的 `js` 格式，如下：

```jsx
return (
  <div className='cn'>
    <Header> hello </Header>
    <div> start </div>
    Right Reserve
  </div>
)
```

`babel` 编译后：

```javascript
return (
  React.createElement(
    'div',
    {
      className : 'cn'
    },
    React.createElement(
      Header,
      null,
      'hello'
    ),
    React.createElement(
      'div',
      null,
      'start'
    ),
    'Right Reserve'
  )
)
```

从名字上来看， `createElement` 方法用来元素的

在 `react` 中，这个元素就是虚拟 `DOM` 树的节点，接收三个参数：

- type：标签
- attributes：标签属性，若无则为null
- children：标签的子节点

这些虚拟 `DOM` 树最终会渲染成真实 `DOM`

在 `render` 过程中， `React` 将新调用的 `render` 函数返回的树与旧版本的树进行比较，这一步是决定如何更新 `DOM` 的必要步骤，然后进行 `diff` 比较，更新 `DOM` 树

17.2. 触发时机

`render` 的执行时机主要分成了两部分：

- 类组件调用 setState 修改状态

JSX | 复制代码

```jsx
class Foo extends React.Component {
  state = { count: 0 };

  increment = () => {
    const { count } = this.state;

    const newCount = count < 10 ? count + 1 : count;

    this.setState({ count: newCount });
  };

  render() {
    const { count } = this.state;
    console.log("Foo render");

    return (
      <div>
        <h1> {count} </h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

点击按钮则调用 `setState` 方法，无论 `count` 发生变化辩护，控制台都会输出 `Foo render`，证明 `render` 执行了

- 函数组件通过 `useState hook` 修改状态

JSX | 复制代码

```jsx
function Foo() {
  const [count, setCount] = useState(0);

  function increment() {
    const newCount = count < 10 ? count + 1 : count;
    setCount(newCount);
  }

  console.log("Foo render");

  return (
    <div>
      <h1> {count} </h1>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

函数组件通过 `useState` 这种形式更新数据，当数组的值不发生改变了，就不会触发 `render`

- 类组件重新渲染

JavaScript | 复制代码

```javascript
class App extends React.Component {
  state = { name: "App" };
  render() {
    return (
      <div className="App">
        <Foo />
        <button onClick={() => this.setState({ name: "App" })}>
          Change name
        </button>
      </div>
    );
  }
}

function Foo() {
  console.log("Foo render");

  return (
    <div>
      <h1> Foo </h1>
    </div>
  );
}
```

只要点击了 `App` 组件内的 `Change name` 按钮，不管 `Foo` 具体实现是什么，都会被重新 `render` 渲染

- 函数组件重新渲染

JSX | 复制代码

```jsx
function App(){
    const [name,setName] = useState('App')

    return (
        <div className="App">
            <Foo />
            <button onClick={() => setName("aaa")}>
                { name }
            </button>
      </div>
    )
}

function Foo() {
  console.log("Foo render");

  return (
    <div>
      <h1> Foo </h1>
    </div>
  );
}
```

可以发现，使用 `useState` 来更新状态的时候，只有首次会触发 `Foo render` ，后面并不会导致 `Foo render`

17.3. 总结

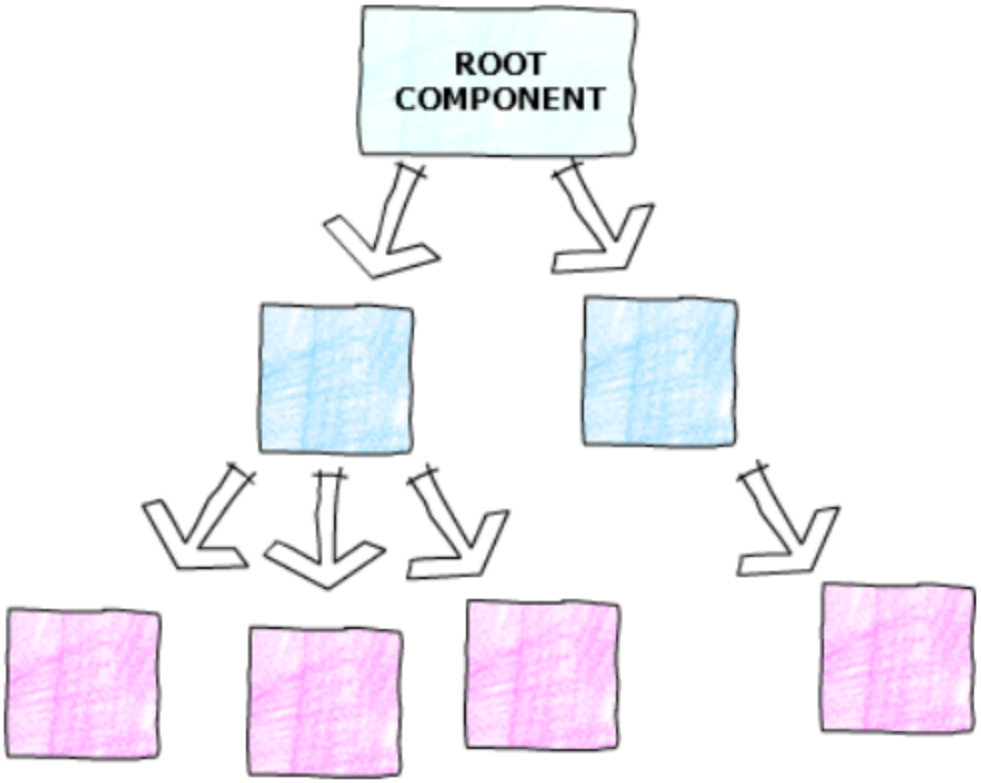
`render` 函数里面可以编写 `JSX` ，转化成 `createElement` 这种形式，用于生成虚拟 `DOM` ，最终转化成真实 `DOM`

在 `React` 中，类组件只要执行了 `setState` 方法，就一定会触发 `render` 函数执行，函数组件使用 `useState` 更改状态不一定导致重新 `render`

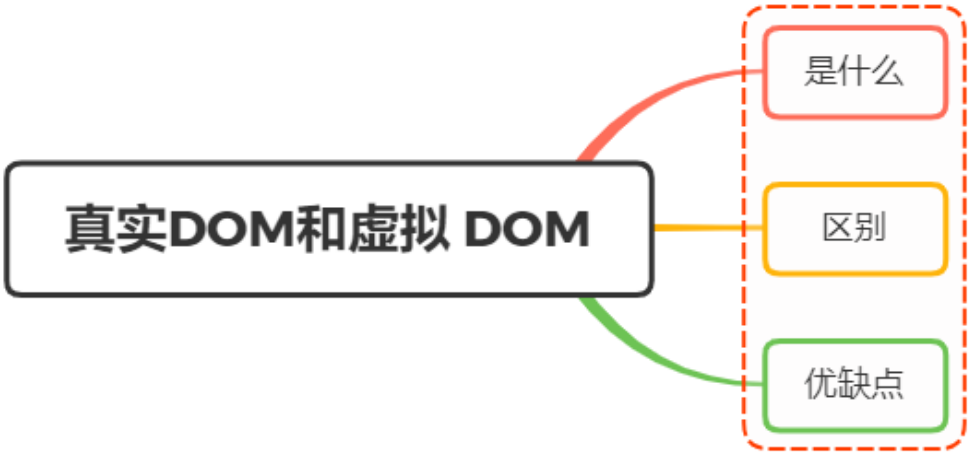
组件的 `props` 改变了，不一定触发 `render` 函数的执行，但是如果 `props` 的值来自于父组件或者祖先组件的 `state`

在这种情况下，父组件或者祖先组件的 `state` 发生了改变，就会导致子组件的重新渲染

所以，一旦执行了 `setState` 就会执行 `render` 方法， `useState` 会判断当前值有无发生改变确定是否执行 `render` 方法，一旦父组件发生渲染，子组件也会渲染

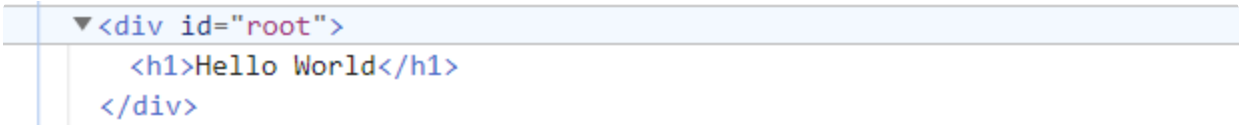


18. 说说 Real DOM 和 Virtual DOM 的区别？优缺点？



18.1. 是什么

Real DOM，真实 `DOM`，意思为文档对象模型，是一个结构化文本的抽象，在页面渲染出的每一个结点都是一个真实 `DOM` 结构，如下：



`Virtual Dom`，本质上是以 `JavaScript` 对象形式存在的对 `DOM` 的描述

创建虚拟 `DOM` 目的就是为了更好将虚拟的节点渲染到页面视图中，虚拟 `DOM` 对象的节点与真实 `DOM` 的属性一一照应

在 `React` 中，`JSX` 是其一大特性，可以让你在 `JS` 中通过使用 `XML` 的方式去直接声明界面的 `DOM` 结构

JSX | 复制代码

```jsx
// 创建 h1 标签，右边千万不能加引号
const vDom = <h1>Hello World</h1>;
// 找到 <div id="root"></div> 节点
const root = document.getElementById("root");
// 把创建的 h1 标签渲染到 root 节点上
ReactDOM.render(vDom, root);
```

上述中，`ReactDOM.render()` 用于将你创建好的虚拟 `DOM` 节点插入到某个真实节点上，并渲染到页面上

`JSX` 实际是一种语法糖，在使用过程中会被 `babel` 进行编译转化成 `JS` 代码，上述 `VDOM` 转化为如下：

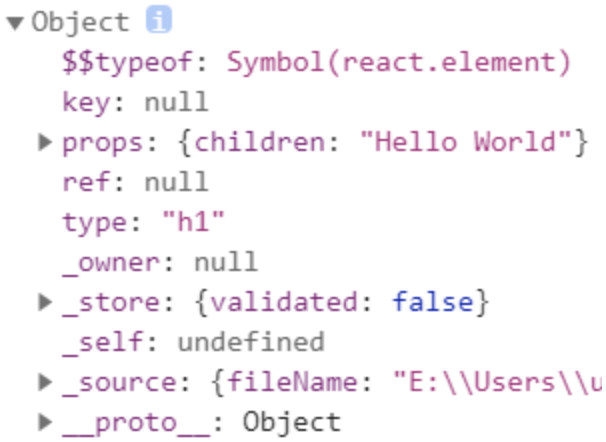
JSX | 复制代码

```jsx
const vDom = React.createElement(
  'h1',
  { className: 'hClass', id: 'hId' },
  'hello world'
)
```

可以看到，`JSX` 就是为了简化直接调用 `React.createElement()` 方法：

- 第一个参数是标签名，例如 h1、span、table...
- 第二个参数是个对象，里面存着标签的一些属性，例如 id、class 等
- 第三个参数是节点中的文本

通过 `console.log(VDOM)`，则能够得到虚拟 `VDOM` 消息



所以可以得到，`JSX` 通过 `babel` 的方式转化成 `React.createElement` 执行，返回值是一个对象，也就是虚拟 `DOM`

18.2. 区别

两者的区别如下：

- 虚拟 DOM 不会进行排版与重绘操作，而真实 DOM 会频繁重排与重绘
- 虚拟 DOM 的总损耗是“虚拟 DOM 增删改+真实 DOM 差异增删改+排版与重绘”，真实 DOM 的总损耗是“真实 DOM 完全增删改+排版与重绘”

传统的原生 `api` 或 `jQuery` 去操作 `DOM` 时，浏览器会从构建 `DOM` 树开始从头到尾执行一遍流程

当你在一次操作时，需要更新 10 个 `DOM` 节点，浏览器没这么智能，收到第一个更新 `DOM` 请求后，并不知道后续还有 9 次更新操作，因此会马上执行流程，最终执行 10 次流程

而通过 `VNode`，同样更新 10 个 `DOM` 节点，虚拟 `DOM` 不会立即操作 `DOM`，而是将这 10 次更新的 `diff` 内容保存到本地的一个 `js` 对象中，最终将这个 `js` 对象一次性 `attach` 到 `DOM` 树上，避免大量的无谓计算

18.3. 优缺点

真实 `DOM` 的优势：

- 易用

缺点：

- 效率低，解析速度慢，内存占用量过高
- 性能差：频繁操作真实 DOM，易于导致重绘与回流

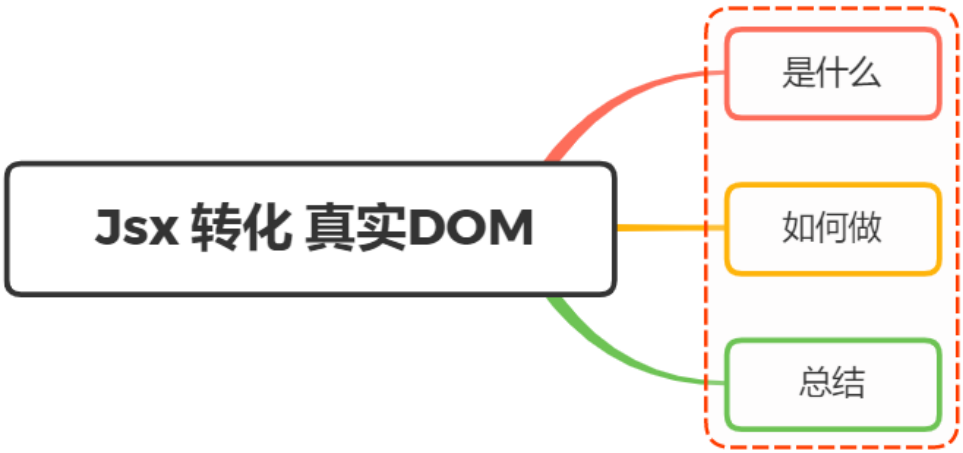
使用虚拟 `DOM` 的优势如下：

- 简单方便：如果使用手动操作真实 `DOM` 来完成页面，繁琐又容易出错，在大规模应用下维护起来也很困难
- 性能方面：使用 Virtual DOM，能够有效避免真实 DOM 数频繁更新，减少多次引起重绘与回流，提高性能
- 跨平台：React 借助虚拟 DOM，带来了跨平台的能力，一套代码多端运行

缺点：

- 在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化
- 首次渲染大量 DOM 时，由于多了一层虚拟 DOM 的计算，速度比正常稍慢

19. 说说React Jsx转换成真实DOM过程?



19.1. 是什么

`react` 通过将组件编写的 `JSX` 映射到屏幕，以及组件中的状态发生了变化之后 `React` 会将这些「变化」更新到屏幕上

在前面文章了解中， `JSX` 通过 `babel` 最终转化成 `React.createElement` 这种形式，例如：

JSX | 复制代码

```jsx
<div>
  < img src="avatar.png" className="profile" />
  <Hello />
</div>
```

会被 `bebel` 转化成如下：

```jsx
React.createElement(
  "div",
  null,
  React.createElement("img", {
    src: "avatar.png",
    className: "profile"
  }),
  React.createElement(Hello, null)
);
```

在转化过程中， `babel` 在编译时会判断 JSX 中组件的首字母：

- 当首字母为小写时，其被认定为原生 `DOM` 标签， `createElement` 的第一个变量被编译为字符串
- 当首字母为大写时，其被认定为自定义组件，createElement 的第一个变量被编译为对象

最终都会通过 `RenderDOM.render(...)` 方法进行挂载，如下：

```jsx
ReactDOM.render(<App />,  document.getElementById("root"));
```

19.2. 过程

在 `react` 中，节点大致可以分成四个类别：

- 原生标签节点
- 文本节点
- 函数组件
- 类组件

如下所示：

```jsx
class ClassComponent extends Component {
  static defaultProps = {
    color: "pink"
  };
  render() {
    return (
      <div className="border">
        <h3>ClassComponent</h3>
        <p className={this.props.color}>{this.props.name}</p >
      </div>
    );
  }
}

function FunctionComponent(props) {
  return (
    <div className="border">
      FunctionComponent
      <p>{props.name}</p >
    </div>
  );
}

const jsx = (
  <div className="border">
    <p>xx</p >
    < a href=" ">xxx</ a>
    <FunctionComponent name="函数组件" />
    <ClassComponent name="类组件" color="red" />
  </div>
);
```

这些类别最终都会被转化成 `React.createElement` 这种形式

`React.createElement` 其被调用时会传入标签类型 `type`，标签属性 `props` 及若干子元素 `children`，作用是生成一个虚拟 `Dom` 对象，如下所示：

JavaScript | 复制代码

```javascript
function createElement(type, config, ...children) {
    if (config) {
        delete config.__self;
        delete config.__source;
    }
    // ! 源码中做了详细处理，比如过滤掉key、ref等
    const props = {
        ...config,
        children: children.map(child =>
    typeof child === "object" ? child : createTextNode(child)
   )
    };
    return {
        type,
        props
    };
}
function createTextNode(text) {
    return {
        type: TEXT,
        props: {
            children: [],
            nodeValue: text
        }
    };
}
export default {
    createElement
};
```

`createElement` 会根据传入的节点信息进行一个判断：

- 如果是原生标签节点， type 是字符串，如div、span
- 如果是文本节点， type就没有，这里是 TEXT
- 如果是函数组件，type 是函数名
- 如果是类组件，type 是类名

虚拟 `DOM` 会通过 `ReactDOM.render` 进行渲染成真实 `DOM` ，使用方法如下：

JSX | 复制代码

```jsx
ReactDOM.render(element, container[, callback])
```

当首次调用时，容器节点里的所有 `DOM` 元素都会被替换，后续的调用则会使用 `React` 的 `diff` 算法进行高效的更新

如果提供了可选的回调函数 `callback`，该回调将在组件被渲染或更新之后被执行

`render` 大致实现方法如下：

JavaScript | 复制代码

```javascript
function render(vnode, container) {
    console.log("vnode", vnode); // 虚拟DOM对象
    // vnode _> node
    const node = createNode(vnode, container);
    container.appendChild(node);
}

// 创建真实DOM节点
function createNode(vnode, parentNode) {
    let node = null;
    const {type, props} = vnode;
    if (type === TEXT) {
        node = document.createTextNode("");
    } else if (typeof type === "string") {
        node = document.createElement(type);
    } else if (typeof type === "function") {
        node = type.isReactComponent
            ? updateClassComponent(vnode, parentNode)
        : updateFunctionComponent(vnode, parentNode);
    } else {
        node = document.createDocumentFragment();
    }
    reconcileChildren(props.children, node);
    updateNode(node, props);
    return node;
}

// 遍历下子vnode，然后把子vnode->真实DOM节点，再插入父node中
function reconcileChildren(children, node) {
    for (let i = 0; i < children.length; i++) {
        let child = children[i];
        if (Array.isArray(child)) {
            for (let j = 0; j < child.length; j++) {
                render(child[j], node);
            }
        } else {
            render(child, node);
        }
    }
}
function updateNode(node, nextVal) {
    Object.keys(nextVal)
        .filter(k => k !== "children")
        .forEach(k => {
        if (k.slice(0, 2) === "on") {
```

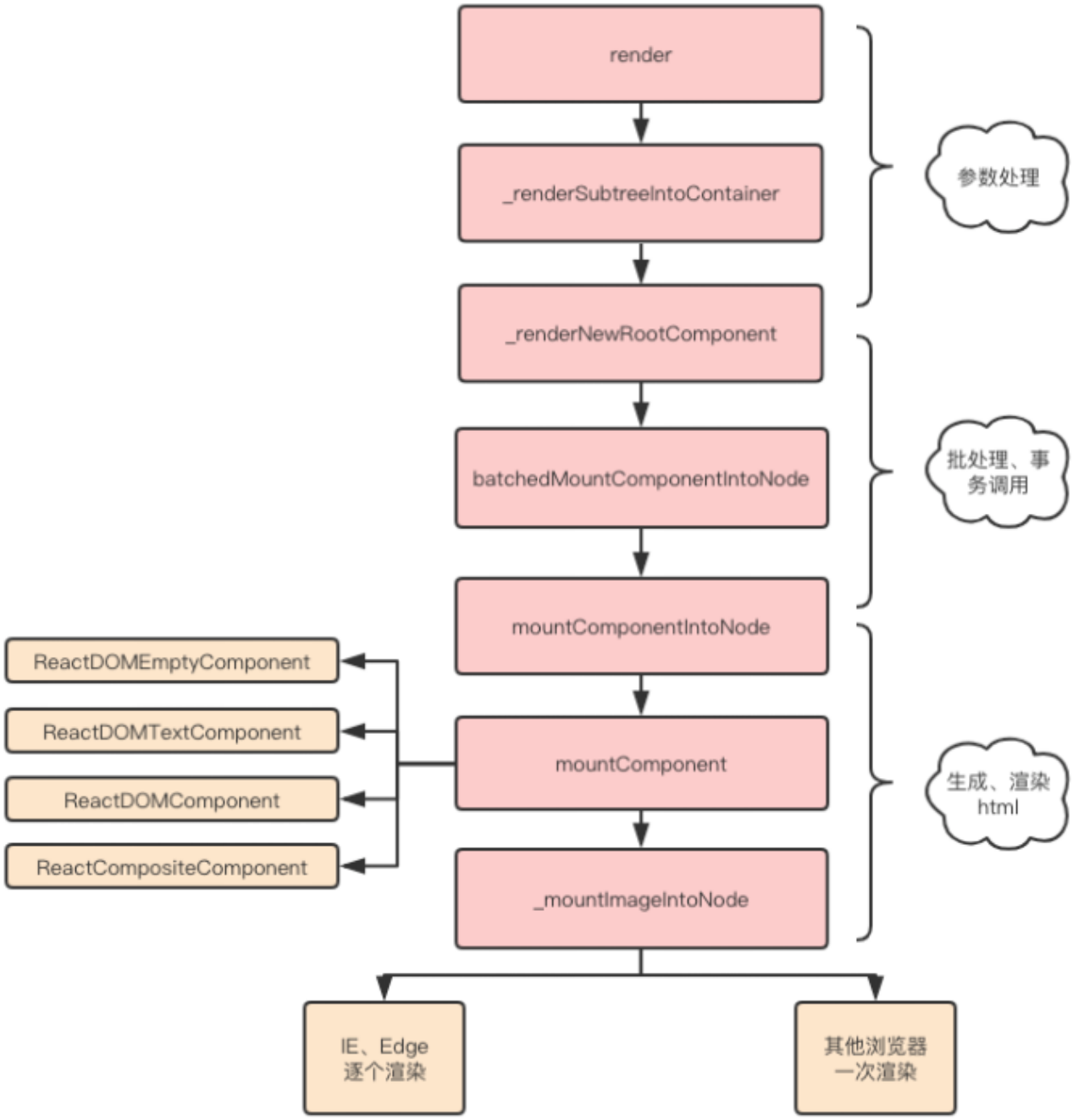
```
            let eventName = k.slice(2).toLocaleLowerCase();
            node.addEventListener(eventName, nextVal[k]);
        } else {
            node[k] = nextVal[k];
        }
    });
}

// 返回真实dom节点
// 执行函数
function updateFunctionComponent(vnode, parentNode) {
    const {type, props} = vnode;
    let vvnode = type(props);
    const node = createNode(vvnode, parentNode);
    return node;
}

// 返回真实dom节点
// 先实例化，再执行render函数
function updateClassComponent(vnode, parentNode) {
    const {type, props} = vnode;
    let cmp = new type(props);
    const vvnode = cmp.render();
    const node = createNode(vvnode, parentNode);
    return node;
}
export default {
    render
};
```

19.3. 总结

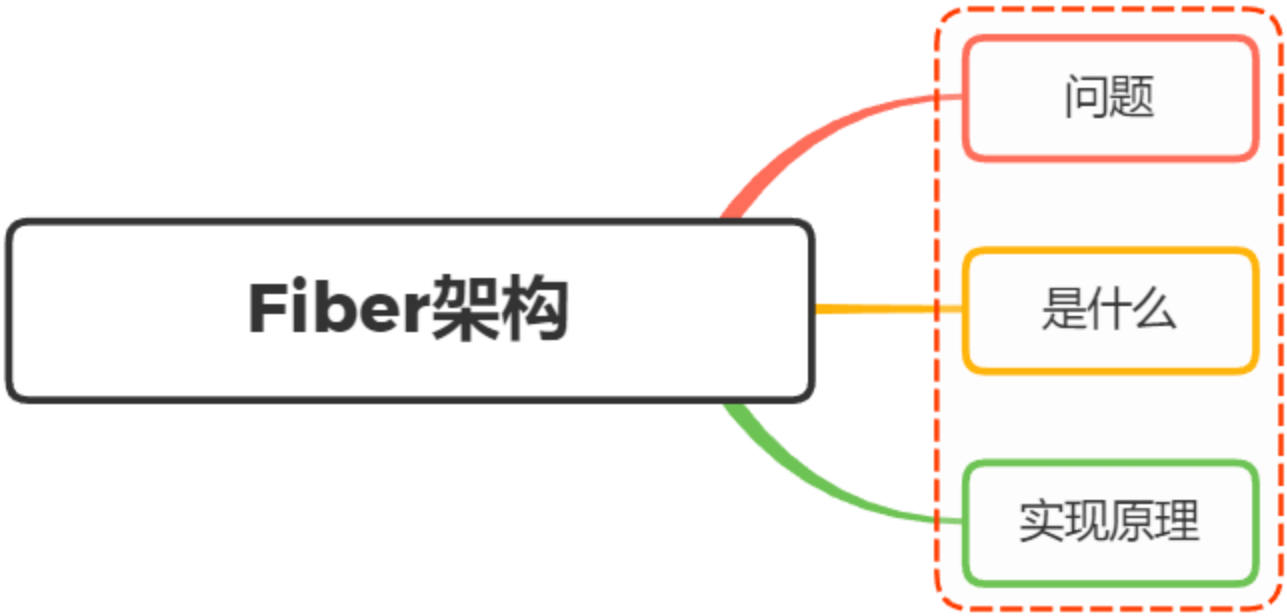
在 `react` 源码中，虚拟 `Dom` 转化成真实 `Dom` 整体流程如下图所示：



其渲染流程如下所示：

- 使用React.createElement或JSX编写React组件，实际上所有的 JSX 代码最后都会转换成 React.createElement(...)，Babel帮助我们完成了这个转换的过程。
- createElement函数对key和ref等特殊的props进行处理，并获取defaultProps对默认props进行赋值，并且对传入的孩子节点进行处理，最终构造成一个虚拟DOM对象
- ReactDOM.render将生成好的虚拟DOM渲染到指定容器上，其中采用了批处理、事务等机制并且对特定浏览器进行了性能优化，最终转换为真实DOM

20. 说说对Fiber架构的理解？解决了什么问题？



20.1. 问题

`JavaScript` 引擎和页面渲染引擎两个线程是互斥的，当其中一个线程执行时，另一个线程只能挂起等待

如果 `JavaScript` 线程长时间地占用了主线程，那么渲染层面的更新就不得不长时间地等待，界面长时间不更新，会导致页面响应度变差，用户可能会感觉到卡顿

而这也正是 `React 15` 的 `Stack Reconciler` 所面临的问题，当 `React` 在渲染组件时，从开始到渲染完成整个过程是一气呵成的，无法中断

如果组件较大，那么 `js` 线程会一直执行，然后等到整棵 `VDOM` 树计算完成后，才会交给渲染的线程

这就会导致一些用户交互、动画等任务无法立即得到处理，导致卡顿的情况