## **1. 使用Fragment**

* Fragment可以让你聚合一个子元素列表,并且不在DOM中增加额外节点
* Fragment 看起来像空的 JSX 标签

### **1.1 index.js**

**import** React **from** 'react';**import** ReactDOM **from** 'react-dom';**import** Table **from** './components/Table';**let** data = [

{id:1,name:'zhufeng',age:10},

{id:2,name:'jiagou',age:10}

]

ReactDOM.render(<Table data={data} />, document.getElementById('root'));

### **1.2 Table.js**

src\components\Table.js

**import** React **from** "react";**class** **Columns** **extends** **React**.**Component** {

render() {

**let** data = **this**.props.data;

//Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>

**return** (

<><td>{data.id}</td><td>{data.name}</td><td>{data.age}</td></>

)

}

}**export** **default** **class** **Table** **extends** **React**.**Component** {

render() {

**return** (

<table>

<thead>

<tr>

<td>ID</td>

<td>Name</td>

<td>Age</td>

</tr>

</thead>

<tbody>

{

this.props.data.map((item, index) => (

<tr key={index}>

<Columns data={item} />

</tr>

))

}

</tbody>

</table>

);

}

}

## **2. PureComponent**

* 当一个组件的props或state变更，React会将最新返回的元素与之前渲染的元素进行对比，以此决定是否有必要更新真实的 DOM，当它们不相同时 React 会更新该 DOM
* 如果渲染的组件非常多时可以通过覆盖生命周期方法 shouldComponentUpdate 来进行优化
* shouldComponentUpdate 方法会在重新渲染前被触发。其默认实现是返回 true,如果组件不需要更新，可以在shouldComponentUpdate中返回 false 来跳过整个渲染过程。其包括该组件的 render 调用以及之后的操作

### **2.1 重复渲染**

**import** React, { Component } **from** 'react';**import** ReactDOM **from** 'react-dom';

**class** **App** **extends** **Component**{

state = {counter:{number:0}}

add = ()=>{

**let** oldState = **this**.state;

**let** amount = parseInt(**this**.amount.value);

**let** newState = {...oldState,counter:amount==0?oldState.counter:{number:oldState.counter.number+amount}};

**this**.setState(newState);

}

render(){

console.log('App render');

**return** (

<div>

<Counter counter={this.state.counter}/>

<input ref={inst=>this.amount = inst}/>

<button onClick={this.add}>+</button>

</div>

)

}

}**class** **Counter** **extends** **React**.**Component**{

render(){

console.log('Counter render');

**return** (

<p>{this.props.counter.number}</p>

)

}

}

ReactDOM.render(

<App />,

document.getElementById('root')

)

### **2.2 PureComponent**

* React15.3 中新加了一个类PureComponent,它会在render之前帮组件自动执行一次shallowEqual（浅比较），来决定是否更新组件
* PureComponent通过prop和state的浅比较来实现shouldComponentUpdate

import React, { Component } from "react";

import ReactDOM from "react-dom";+class PureComponent extends Component {+ shouldComponentUpdate(newProps) {+ return !shallowEqual(this.props, newProps);+ }+}+function shallowEqual(obj1, obj2) {+ if (obj1 === obj2) {+ return true;+ }+ if (typeof obj1 != "object" ||obj1 === null ||typeof obj2 != "object" ||obj2 === null) {+ return false;+ }+ let keys1 = Object.keys(obj1);+ let keys2 = Object.keys(obj2);+ if (keys1.length != keys2.length) {+ return false;+ }+ for (let key of keys1) {+ if (!obj2.hasOwnProperty(key) || obj1[key] !== obj2[key]) {+ return false;+ }+ }+ return true;+}

class App extends Component {

state = { counter: { number: 0 } };

add = () => {

let oldState = this.state;

let amount = parseInt(this.amount.value);

let newState = {

...oldState,

counter:

amount == 0

? oldState.counter

: { number: oldState.counter.number + amount }

};

this.setState(newState);

};

render() {

console.log("App render");

return (

<div>

<Counter counter={this.state.counter} />

<input ref={inst => (this.amount = inst)} />

<button onClick={this.add}>+</button>

</div>

);

}

}+class Counter extends PureComponent {

render() {

console.log("Counter render");

return <p>{this.props.counter.number}</p>;

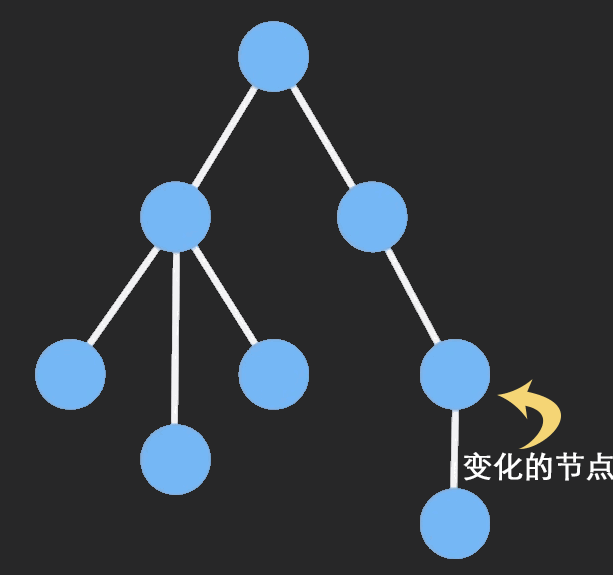
}

}

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

### **2.3 PureComponent+Immutable.js**

* [Immutable.js](https://immutable-js.github.io/immutable-js/)是 Facebook 在 2014 年出的持久性数据结构的库
* Immutable Data 就是一旦创建，就不能再被更改的数据。对 Immutable 对象的任何修改或添加删除操作都会返回一个新的 Immutable 对象一个新的 Immutable 对象
* Immutable 实现的原理是 Persistent Data Structure（持久化数据结构），也就是使用旧数据创建新数据时，要保证旧数据同时可用且不变,同时为了避免 deepCopy 把所有节点都复制一遍带来的性能损耗
* Immutable 使用了 Structural Sharing（结构共享），即如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其它节点则进行共享



#### **2.4.1 immutable**

* [immutable-js](https://immutable-js.github.io/immutable-js/)内部实现了一套完整的 Persistent Data Structure,还有很多易用的数据类型。像 Collection、List、Map、Set、Record、Seq

##### **2.4.1.1 安装**

cnpm install immutable -S

##### **2.4.1.2 使用**

**let** { Map } = require("immutable");**const** map1 = Map({ a: { aa: 1 }, b: 2, c: 3 });**const** map2 = map1.set('b', 50);console.log(map1 !== map2); // trueconsole.log(map1.get('b')); // 2console.log(map2.get('b')); // 50console.log(map1.get('a') === map2.get('a')); // true

##### **2.4.1.3 重构**

import React, { Component } from "react";

import ReactDOM from "react-dom";+ import { Map,is } from "immutable";

class PureComponent extends Component {

shouldComponentUpdate(newProps) {

return !shallowEqual(this.props, newProps);

}

}

function shallowEqual(obj1, obj2) {

if (obj1 === obj2) {

return true;

}

if (typeof obj1 != "object" ||obj1 === null ||typeof obj2 != "object" ||obj2 === null) {

return false;

}

let keys1 = Object.keys(obj1);

let keys2 = Object.keys(obj2);

if (keys1.length != keys2.length) {

return false;

}

for (let key of keys1) {+ if (!obj2.hasOwnProperty(key) || !is(obj1[key],obj2[key])) {

return false;

}

}

return true;

}

class App extends Component {+ state = { counter: Map({ number: 0 }) };

add = () => {

/\*\*

let oldState = this.state;

let amount = parseInt(this.amount.value);

this.setState({counter:{ number: oldState.counter.number + amount }});

\*/+ this.state.counter = this.state.counter.set('number',this.state.counter.get('number') + parseInt(this.amount.value));+ this.setState(this.state);

};

render() {

console.log("App render");

return (

<div>

<Counter counter={this.state.counter} />

<input ref={inst => (this.amount = inst)} />

<button onClick={this.add}>+</button>

</div>

);

}

}

class Counter extends PureComponent {

render() {

console.log("Counter render");

return <p>{this.props.counter.number}</p>;

}

}

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

## **3. memo**

* React.memo()是一个高阶函数，它与 React.PureComponent类似，但是一个函数组件而非一个类

### **3.1 memoization(memorization)方案**

* memoization(memorization)方案是一种将函数执行结果用变量缓存起来的方法
* 当函数进行计算之前，先看缓存对象中是否有次计算结果，如果有，就直接从缓存对象中获取结果；如果没有，就进行计算，并将结果保存到缓存对象中

### **3.2 优化**

import React, { Component } from "react";

import ReactDOM from "react-dom";

import { Map,is } from "immutable";

class PureComponent extends Component {

isPureReactComponent = true;

shouldComponentUpdate(newProps, newState) {

return (

!shallowEqual(this.props, newProps)

);

}

}

class App extends Component {

state = { title:'计数器',counter: Map({ number: 0 }) };

add = () => {

this.state.counter = this.state.counter.set('number',this.state.counter.get('number') + parseInt(this.amount.value));

this.setState(this.state);

};

render() {

console.log("App render");

return (

<div>+ <Title title={this.props.title}/>

<Counter counter={this.state.counter} />

<input ref={inst => (this.amount = inst)} />

<button onClick={this.add}>+</button>

</div>

);

}

}+function memo(Func){+ class Proxy extends PureComponent{+ render(){+ return <Func {...this.props}/>+ }+ }+ return Proxy;+}+const Title = memo(props=>{+ console.log('Title render');+ return <p>{props.title}</p>;+});

class Counter extends PureComponent {

render() {

console.log("Counter render");

return <p>{this.props.counter.get('number')}</p>;

}

}

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

function shallowEqual(obj1, obj2) {

if (obj1 === obj2) {

return true;

}

if (

typeof obj1 != "object" ||

obj1 === null ||

typeof obj2 != "object" ||

obj2 === null

) {

return false;

}

let keys1 = Object.keys(obj1);

let keys2 = Object.keys(obj2);

if (keys1.length != keys2.length) {

return false;

}

for (let key of keys1) {

if (!obj2.hasOwnProperty(key) || !is(obj1[key],obj2[key])) {

return false;

}

}

return true;

}

## **4. Lazy+Error Boundaries**

### **4.1 React.Lazy**

* React.Lazy帮助我们按需加载组件，从而减少我们应用程序的加载时间，因为只加载我们所需的组件。
* React.lazy 接受一个函数，这个函数内部调用 import() 动态导入。它必须返回一个 Promise，该 Promise 需要 resolve 一个 default export 的 React 组件
* React.Suspense 用于包装延迟组件以在加载组件时显示后备内容

**import** React, { Component, Suspense } **from** 'react'**import** ReactDOM **from** 'react-dom';**import** Loading **from** './components/Loading';**function** **lazy**(loadFunction){

**return** **class** **LazyComponent** **extends** **React**.**Component**{

state = {Comp:null}

componentDidMount(){

loadFunction().then(result=>{

**this**.setState({Comp:result.default});

});

}

render(){

**let** Comp = **this**.state.Comp;

**return** Comp?<Comp {...this.props}/>:null;

}

}

}**const** AppTitle = React.lazy(()=>**import**(/\* webpackChunkName: "title" \*/'./components/Title'))

**class** **App** **extends** **Component**{

state = {visible:false}

show = ()=>{

**this**.setState({visible:true});

}

render() {

**return** (

<>

{this.state.visible&&(

<Suspense fallback={<Loading/>}>

<AppTitle/>

</Suspense>

)}

<button onClick={this.show}>加载</button>

</>

)

}

}

ReactDOM.render(<App />, document.querySelector('#root'));

### **4.2 错误边界(Error Boundaries)**

* 如果当一个组件异步加载下载js文件时，网络错误，无法下载 js 文件 Suspense 无法处理这种错误情况， 在 react 中有一个 错误边界 （Error Boundaries）的概念，用来解决这种问题，它是利用了 react 生命周期的 componentDidCatch 方法来处理
* 有两种方式，一种是 生命周期 componentDidCatch 来处理错误，还有一种 是 静态方法 static getDerivedStateFromError 来处理错误，
* 请使用static getDerivedStateFromError()渲染备用 UI ，使用 componentDidCatch() 打印错误信息。

import React, { Component, Suspense } from 'react'

import ReactDOM from 'react-dom';

import Loading from './components/Loading';+ const AppTitle = React.lazy(()=>import(/\* webpackChunkName: "title" \*/'./components/Title'))

class App extends Component{+ state = {visible:false,isError: false}

show = ()=>{

this.setState({visible:true});

}

+ static getDerivedStateFromError(error) {+ return { isError: true };+ }+ componentDidCatch (err, info) {+ console.log(err, info)+ }

render() {

if (this.state.isError) {

return (<div>error</div>)

}

return (

<>

{this.state.visible&&(

<Suspense fallback={<Loading/>}>

<AppTitle/>

</Suspense>

)}

<button onClick={this.show}>加载</button>

</>

)

}

}

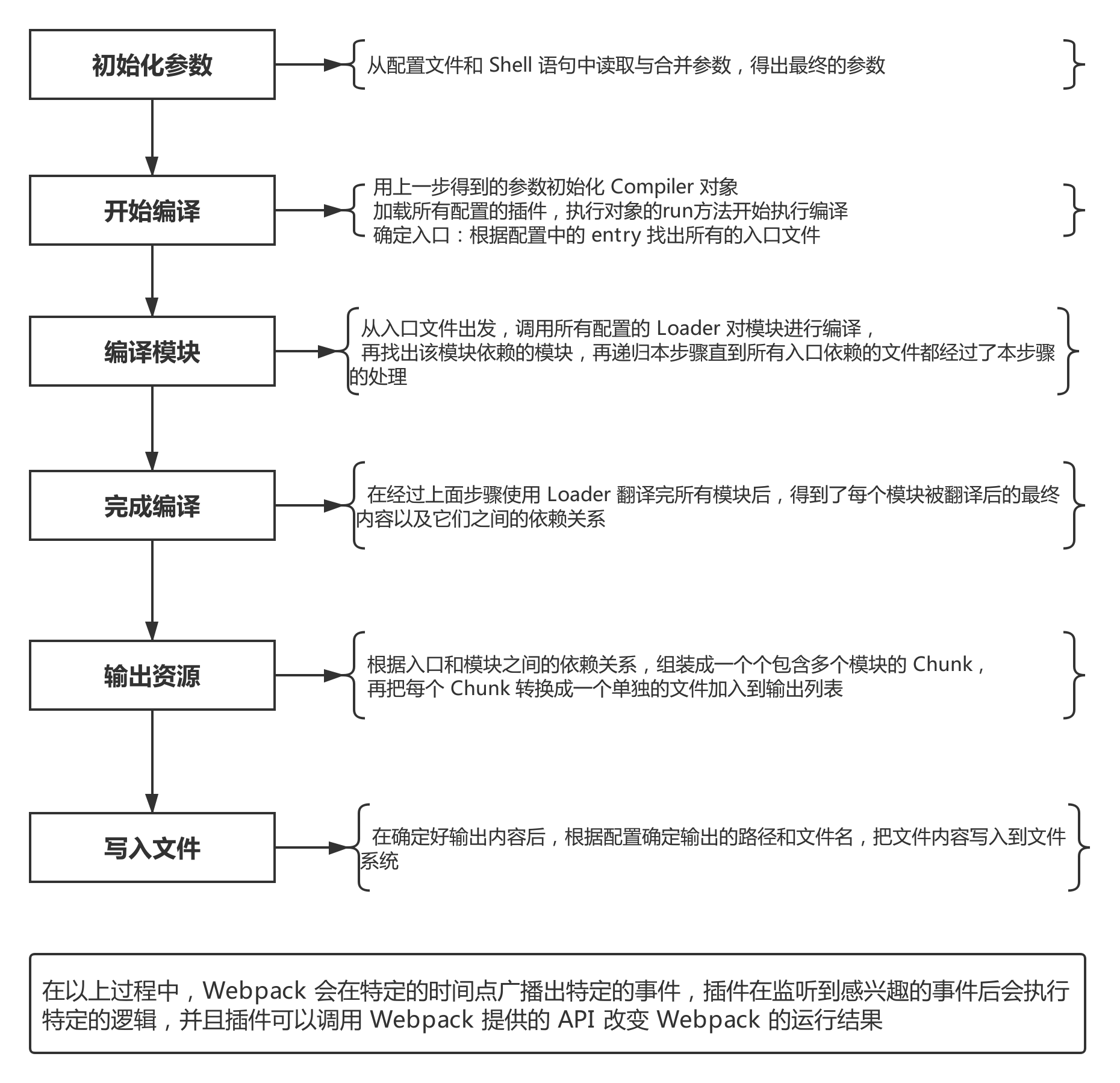
ReactDOM.render(<App />, document.querySelector('#root'));

## **5. 骨架屏**

* Skeleton Screen(骨架屏)就是在页面数据尚未加载前先给用户展示出页面的大致结构，直到请求数据返回后再渲染页面，补充进需要显示的数据内容。常用于文章列表、动态列表页。
* [react-content-loader](https://www.npmjs.com/package/react-content-loader) SVG-Powered component to easily create placeholder loadings
* [create-content-loader](http://danilowoz.com/create-content-loader/)
* [react-skeleton-webpack-plugin](https://github.com/lavas-project/react-skeleton-webpack-plugin) is a Webpack plugin based on React which generates Skeleton Screen for SPA

cnpm i @babel/core @babel/plugin-proposal-**class**-**properties** @**babel**/**plugin**-**proposal**-**decorators** @**babel**/**preset**-**env** @**babel**/**preset**-**react** **babel**-**loader** **html**-**webpack**-**plugin** **webpack** **webpack**-**cli** **webpack**-**dev**-**server** **webpack**-**merge** **webpack**-**node**-**externals** **memory**-**fs** **require**-**from**-**string** **react**-**content**-**loader** **react**-**router**-**dom** **prerender**-**spa**-**plugin** **react**-**lazyload** **react**-**window** **immutable** -**D**

**npx** **webpack** --**config** **webpack**.**skeleton**.**jsnpx** **webpack**



### **5.1 skeleton.js**

src\skeleton.js

**import** React **from** 'react';**import** ReactDOM **from** 'react-dom';**import** ReactDOMServer **from** 'react-dom/server';**import** ContentLoader **from** 'react-content-loader';**export** **default** ReactDOMServer.renderToStaticMarkup(<ContentLoader />);

### **5.2 index.js**

src\index.js

**import** React **from** "react";**import** ReactDOM **from** "react-dom";**let** style = { width: "100%", height: "300px", backgroundColor: "orange" };

setTimeout(() => {

ReactDOM.render(<div style={style}></div>, document.getElementById("root"));

}, 2000);

### **5.3 index.html**

src\index.html

<!DOCTYPE html><html lang="en"><head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title></head><body>

<div id="root"></div></body></html>

### **5.4 webpack.base.js**

webpack.base.js

**const** path = require('path');module.exports = {

mode:'development',

devtool:"none",

context: process.cwd(),

output: {

path: path.resolve(\_\_dirname, "dist")

},

module: {

rules: [

{

test: /\.jsx?$/,

use: {

loader: "babel-loader",

options: {

presets: ["@babel/preset-env","@babel/preset-react"],

plugins: [

["@babel/plugin-proposal-decorators", { legacy: true }],

["@babel/plugin-proposal-class-properties", { loose: true }]

]

}

},

include: path.join(\_\_dirname, "src"),

exclude: /node\_modules/

}

]

}

};

### **5.5 webpack.skeleton.js**

webpack.skeleton.js

* [targets](https://webpack.docschina.org/concepts/targets/)

**const** path = require("path");**const** HtmlWebpackPlugin = require("html-webpack-plugin");**const** { smart } = require("webpack-merge");**const** base = require("./webpack.base");**const** nodeExternals = require('webpack-node-externals');module.exports = smart(base, {

target: 'node',

mode: "development",

context: process.cwd(),

entry: "./src/skeleton.js",

output:{

filename:'skeleton.js',

libraryTarget: 'commonjs2'

},

externals: nodeExternals()

});

### **5.6 webpack.config.js**

webpack.config.js

**const** path = require("path");**const** HtmlWebpackPlugin = require("html-webpack-plugin");**const** { smart } = require("webpack-merge");**const** base = require("./webpack.base");**const** SkeletonWebpackPlugin = require('./SkeletonWebpackPlugin');module.exports = smart(base, {

mode: "development",

context: process.cwd(),

entry: {main:"./src/index.js"},

output:{

filename:'main.js'

},

plugins: [

**new** HtmlWebpackPlugin({

template: "./src/index.html", //指定模板文件

filename: "index.html" //产出后的文件名

}),

**new** SkeletonWebpackPlugin({

webpackConfig: require('./webpack.skeleton')

})

]

});

### **5.7 SkeletonWebpackPlugin.js**

SkeletonWebpackPlugin.js

* [memory-fs](https://www.npmjs.com/package/memory-fs) is a simple in-memory filesystem
* [require-from-string](https://www.npmjs.com/package/require-from-string) Load module from string in Node.
* [html-webpack-plugin](https://www.npmjs.com/package/html-webpack-plugin)

**let** requireFromString = require('require-from-string');**let** result = requireFromString('module.exports = "hello"');console.log(result);// hello

**let** webpack = require("webpack");**let** path = require('path');**let** MFS = require("memory-fs");**var** requireFromString = require("require-from-string");**let** mfs = **new** MFS();**class** **SkeletonPlugin** {

**constructor**(options) {

**this**.options = options;

}

apply(compiler) {

**let** { webpackConfig } = **this**.options;

compiler.hooks.compilation.tap("SkeletonPlugin", compilation => {

compilation.hooks.htmlWebpackPluginBeforeHtmlProcessing.tapAsync(

"SkeletonPlugin",

(htmlPluginData, callback) => {

**let** outputPath = path.join(webpackConfig.output.path,webpackConfig.output.filename);

**let** childCompiler = webpack(webpackConfig);

childCompiler.outputFileSystem = mfs;

childCompiler.run((err, stats) => {

**let** skeleton= mfs.readFileSync(outputPath, "utf8");

**let** skeletonHtml = requireFromString(skeleton);

**if** (skeletonHtml.default) {

skeletonHtml = skeletonHtml.default;

}

htmlPluginData.html=htmlPluginData.html.replace(`<div id="root"></div>`,`<div id="root">${skeletonHtml}</div>`);

callback(null, htmlPluginData);

});

}

);

});

}

}module.exports = SkeletonPlugin;

## **6. 预渲染**

* 由于SPA项目普通的爬虫无法爬取项目的静态文本的内容，通过预渲染插件[prerender-spa-plugin([https://github.com/chrisvfritz/prerender-spa-plugin)解决SPA项目的SEO问题](https://github.com/chrisvfritz/prerender-spa-plugin)%E8%A7%A3%E5%86%B3SPA%E9%A1%B9%E7%9B%AE%E7%9A%84SEO%E9%97%AE%E9%A2%98)
* prerender-spa-plugin 利用了 Puppeteer 的爬取页面的功能。
* Puppeteer 是一个 Chrome官方出品的 headless Chrome node 库。它提供了一系列的 API, 可以在无 UI 的情况下调用 Chrome 的功能, 适用于爬虫、自动化处理等各种场景
* 原理是在 Webpack 构建阶段的最后，在本地启动一个 Puppeteer 的服务，访问配置了预渲染的路由，然后将 Puppeteer 中渲染的页面输出到 HTML 文件中，并建立路由对应的目录

### **6.1 src\index.js**

src\index.js

import React from 'react';

import ReactDOM from 'react-dom';

import {BrowserRouter as Router,Route,Link} from 'react-router-dom';

let Home = props=><div>Home</div>

let User = props=><div>User</div>

let Profile = props=><div>Profile</div>

ReactDOM.render(

<Router>

<>

<Link to="/">home</Link>

<Link to="/user">user</Link>

<Link to="/profile">profile</Link>

<Route path="/" exact={true} component={Home} />

<Route path="/user" component={User} />

<Route path="/profile" component={Profile}/>

</>

</Router>

,document.getElementById('root'));

### **6.2 src\index.html**

src\index.html

<!DOCTYPE html><html lang="en"><head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title></head><body>

<div id="root"></div></body></html>

### **6.3 webpack.config.js**

webpack.config.js

**const** path = require("path");**const** HtmlWebpackPlugin = require("html-webpack-plugin");**const** PrerenderSPAPlugin = require("./prerender-spa-plugin");

module.exports = {

mode: "development",

context: process.cwd(),

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "bundle.js"

},

module: {

rules: [

{

test: /\.jsx?$/,

use: {

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

plugins: [

["@babel/plugin-proposal-decorators", { legacy: true }],

["@babel/plugin-proposal-class-properties", { loose: true }]

]

}

},

include: path.join(\_\_dirname, "src"),

exclude: /node\_modules/

}

]

},

plugins: [

**new** HtmlWebpackPlugin({

template: "./src/index.html", //指定模板文件

filename: "index.html" //产出后的文件名

}),

**new** PrerenderSPAPlugin({

staticDir: path.join(\_\_dirname, "dist"),

routes: ["/","/user","/profile"]

})

]

};

### **6.4 prerender-spa-plugin.js**

prerender-spa-plugin.js

**const** path = require("path");**const** Prerenderer = require("@prerenderer/prerenderer");**const** PuppeteerRenderer = require("@prerenderer/renderer-puppeteer");**class** **PrerenderSPAPlugin** {

**constructor**(options) {

**this**.\_options = options;

**this**.\_options.renderer = **new** PuppeteerRenderer({ headless: true });

}

apply(compiler) {

**let** \_this = **this**;

**const** compilerFS = compiler.outputFileSystem;

**const** afterEmit = (compilation, done) => {

**const** PrerendererInstance = **new** Prerenderer(\_this.\_options);

PrerendererInstance.initialize()

.then(() => {

**return** PrerendererInstance.renderRoutes(\_this.\_options.routes || []);

})

.then(renderedRoutes => {

**let** promises = renderedRoutes.map(rendered => {

**return** **new** Promise(**function**(resolve) {

rendered.outputPath = path.join(

\_this.\_options.staticDir,

rendered.route,

"index.html"

);

**let** dir = path.dirname(rendered.outputPath);

compilerFS.mkdirp(dir, (err, made) => {

compilerFS.writeFile(

rendered.outputPath,

rendered.html,

err => {

resolve();

}

);

});

});

});

**return** Promise.all(promises);

})

.then(() => {

PrerendererInstance.destroy();

done();

});

};

compiler.hooks.afterEmit.tapAsync("PrerenderSPAPlugin", afterEmit);

}

}module.exports = PrerenderSPAPlugin;

不适合不同的用户看都会不同的页面，这种类型的页面不适用预渲染 对于一些经常发生变化的页面，如体育比赛等，会导致编译后的数据不是实时更新的

## **7. 图片懒加载**

* [react-lazyload](https://github.com/twobin/react-lazyload)
* [lazyimages.zip](http://img.zhufengpeixun.cn/lazyimages.zip)

### **7.1 webpack.config.js**

webpack.config.js

**const** path = require('path');**const** HtmlWebpackPlugin=require('html-webpack-plugin');module.exports = {

mode:'development',

context: process.cwd(),

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "bundle.js"

},

module: {

rules: [

{

test: /\.jsx?$/,

use: {

loader: "babel-loader",

options: {

presets: ["@babel/preset-env","@babel/preset-react"],

plugins: [

["@babel/plugin-proposal-decorators", { legacy: true }],

["@babel/plugin-proposal-class-properties", { loose: true }]

]

}

},

include: path.join(\_\_dirname, "src"),

exclude: /node\_modules/

},

{

test:/\.(jpg|png|gif)$/,

use:{loader:'url-loader',options:{limit:0}}

},

{

test:/\.css$/,

use:["style-loader",'css-loader']

}

]

},

plugins: [

**new** HtmlWebpackPlugin({

template:'./src/index.html',//指定模板文件

filename:'index.html',//产出后的文件名

})

]

};

### **7.2 index.js**

src\index.js

**import** React **from** "react";**import** ReactDOM **from** "react-dom";**import** './index.css';**import** LazyLoad **from** "./react-lazyload";**const** App = (props) => {

**return** (

<ul className="list" style={{overflow:'auto'}}>

{

props.images.map((image,index)=>(

<LazyLoad key={index} height={200} >

<li> <img src={image} /></li>

</LazyLoad>

))

}

</ul>

);

};**let** images = [

require('./images/1.jpg'),

require('./images/2.jpg'),

require('./images/3.jpg'),

require('./images/4.jpg'),

require('./images/5.jpg'),

require('./images/6.jpg'),

require('./images/7.jpg'),

require('./images/8.jpg'),

]

ReactDOM.render(<App images={images}/>, document.getElementById("root"));

### **7.3 index.css**

src\index.css

\*{

margin: 0;

padding: 0;

}ul,li{

list-style: none;

}li img{

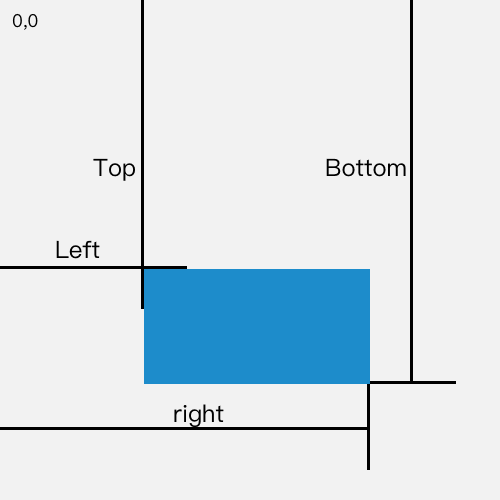
width:100%;

height:100%;

}

### **7.4 react-lazyload.js**

* [getBoundingClientRect](https://developer.mozilla.org/zh-CN/docs/Web/API/Element/getBoundingClientRect)返回值是一个 DOMRect 对象，这个对象是由该元素的 getClientRects() 方法返回的一组矩形的集合, 即：是与该元素相关的CSS 边框集合
* DOMRect 对象包含了一组用于描述边框的只读属性——left、top、right和bottom，单位为像素。除了 width 和 height 外的属性都是相对于视口的左上角位置而言的



src\react-lazyload.js

**import** React **from** "react";**import** ReactDOM **from** "react-dom";**let** listeners = [];**let** lazyLoadHandler = () => {

**for** (**var** i = 0; i < listeners.length; ++i) {

**var** listener = listeners[i];

checkVisible(listener);

}

};**let** checkVisible = component => {

**let** node = ReactDOM.findDOMNode(component);

**let** { top } = node.getBoundingClientRect();

**let** visible = top <= (window.innerHeight || document.documentElement.clientHeight);

**if** (visible) {

listeners = listeners.filter(item => item != component);

component.setState({visible});

}

};**class** **LazyLoad** **extends** **React**.**Component** {

state = {visible:false}

**constructor**(props) {

**super**(props);

**this**.divRef = React.createRef();

}

componentDidMount() {

**if** (listeners.length == 0) {

window.addEventListener("scroll", lazyLoadHandler);

}

listeners.push(**this**);

checkVisible(**this**);

}

render() {

**return** **this**.state.visible ? (

**this**.props.children

) : (

<div

style={{ height: this.props.height }}

className="lazyload-placeholder"

ref={this.divRef}

/>

);

}

}**export** **default** LazyLoad;

## **8. 长列表优化**

* 用数组保存所有列表元素的位置，只渲染可视区内的列表元素，当可视区滚动时，根据滚动的offset大小以及所有列表元素的位置，计算在可视区应该渲染哪些元素
* [react-window](https://www.npmjs.com/package/react-window)
* [fixed-size](https://react-window.now.sh/" \l "/examples/list/fixed-size)
* [react-virtualized](https://www.npmjs.com/package/react-virtualized)

### **8.1 index.js**

index.js

**import** React, { Component, lazy, Suspense } **from** "react";**import** ReactDOM **from** "react-dom";**import** { FixedSizeList **as** List } **from** './react-window';**import** './index.css'**const** Row = ({ index, style }) => {

**return** <div key={index} style={{...style,backgroundColor:getRandomColor(),lineHeight:'30px',textAlign:'center'}}>Row {index+1}</div>

};

**const** Container = () => (

<List

height={150}

itemCount={100}

itemSize={30}

width={'100%'}

>

{Row}

</List>

);

ReactDOM.render(<Container/>, document.querySelector("#root"));**function** **getRandomColor**( ) {

**var** rand = Math.floor(Math.random( ) \* 0xFFFFFF).toString(16).toUpperCase();

**if**(rand.length == 6){

**return** '#'+rand;

}**else**{

**return** getRandomColor();

}

}

### **8.2 index.css**

index.css

\*{

margin: 0;

padding: 0;

}

ul,li{

list-style: none;

}

### **8.3 react-window.js**

react-window.js

**import** React, { Component} **from** "react";**export** **class** **FixedSizeList** **extends** **React**.**Component**{

state = {start:0}

**constructor**(){

**super**();

**this**.containerRef = React.createRef();

}

componentDidMount(){

**this**.containerRef.current.addEventListener('scroll',()=>{

**let** scrollTop = **this**.containerRef.current.scrollTop;

**let** start = Math.floor(scrollTop/**this**.props.itemSize);//起始的索引

**this**.setState({start});

});

}

render(){

**let** {width,height,itemCount,itemSize} = **this**.props;

**let** children = [];

**let** size = Math.floor(height/itemSize)+1;//每页的条数

**let** itemStyle = {height:itemSize,width:'100%',position:'absolute',left:0,top:0};

**for**(**let** index=**this**.state.start;index<**this**.state.start+size;index++){

**let** style = {...itemStyle,top:(index)\*itemSize};

children.push(**this**.props.children({index,style}));

}

**let** containerStyle = {height,width:width||'100%',position:'relative',overflow:'auto'};

**return** (

<div style={containerStyle} ref={this.containerRef}>

<div style={{width:'100%',height:itemSize\*itemCount}}>

{children}

</div>

</div>

)

}

}

## **9. key的优化**

### **9.1 diff策略**

* DOM节点跨节点层级移动可以忽略
* 相同类型的组件生成相似的结构，不同类型的组件生成不同的结构
* 对于同一层次的子节点可以通过唯一的key进行区分

### **9.2 tree diff**

* 对树进行分层比较，两棵树只会对同一层次节点进行比较
* 当出现跨层级移动时，并不会出现移动操作，而是直接删除重建

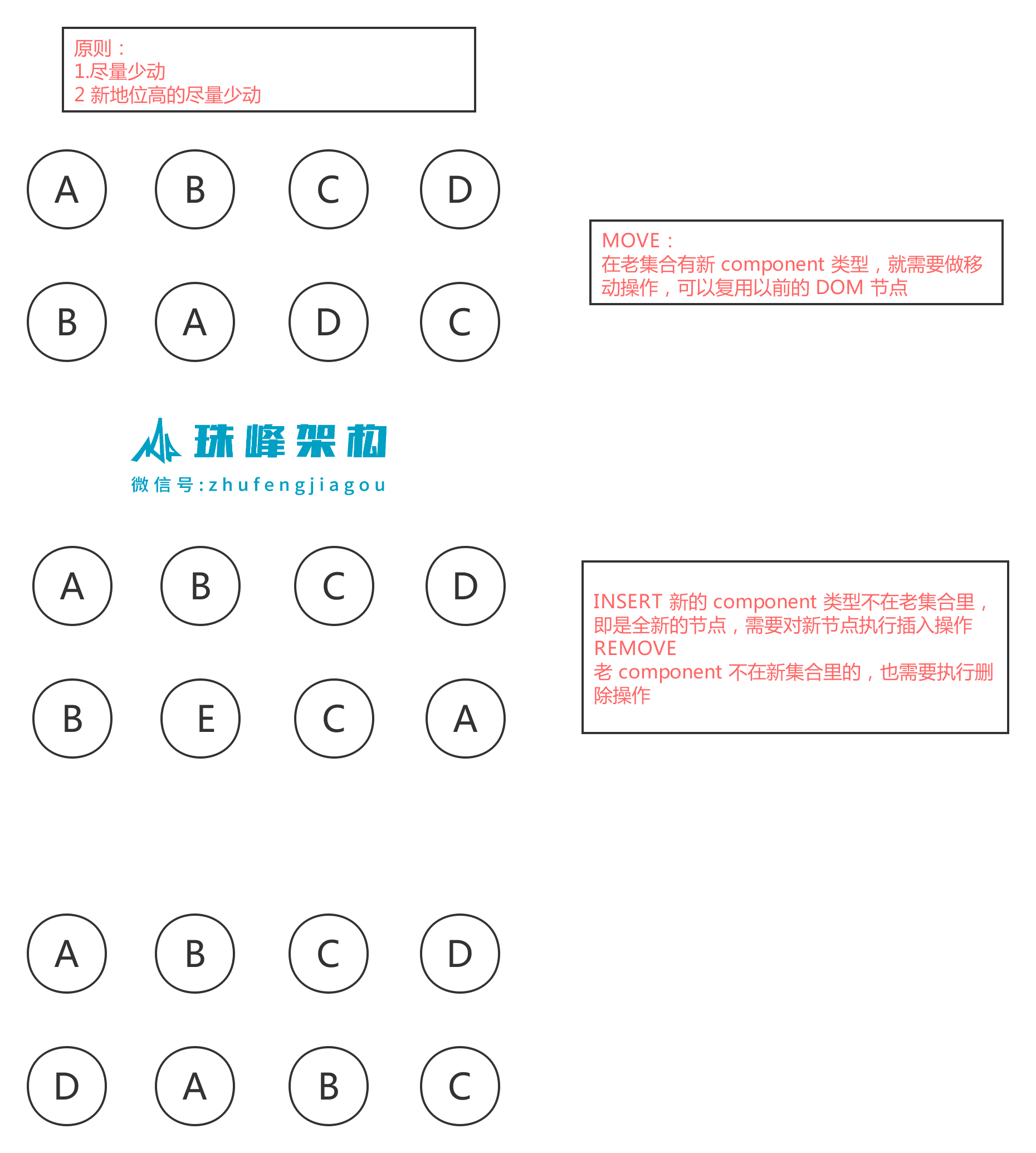
### **9.3 组件diff**

* 如果是同一个类型的组件，会向下继续比较子节点
* 如果类型不同，则替换组件下的所有子节点

### **9.4 element diff**

当节点处于同一层级时，React diff 提供了三种节点操作,分别为：INSERT(插入)、MOVE(移动)和 REMOVE(删除)

* INSERT: 新的 component 类型不在老集合里， 即是全新的节点，需要对新节点执行插入操作
* MOVE: 在老集合有新 component 类型，就需要做移动操作，可以复用以前的 DOM 节点
* REMOVE: 老 component 不在新集合里的，也需要执行删除操作



## **10. React 性能分析器**

* [introducing-the-react-profiler](http://react.html.cn/blog/2018/09/10/introducing-the-react-profiler.html)
* React 16.5 增加了对新的开发者工具 DevTools 性能分析插件的支持
* 此插件使用 React 实验性的 Profiler API 来收集有关每个组件渲染的用时信息，以便识别 React 应用程序中的性能瓶颈

## **10.1 分析解析**

* 分析一个应用程序的性能（Profiling an application）
* 查看性能数据(render(渲染)阶段和commit(提交)阶段)
* 过滤 commits（Filtering commits）
* 火焰图表（Flame chart）
* 排序图表（Ranked chart）
* 组件图表（Component chart）
* 交互（Interactions）

## **10.2 react-flame-graph**

* [react-flame-graph](https://github.com/bvaughn/react-flame-graph)是用来可视化性能数据的React组件