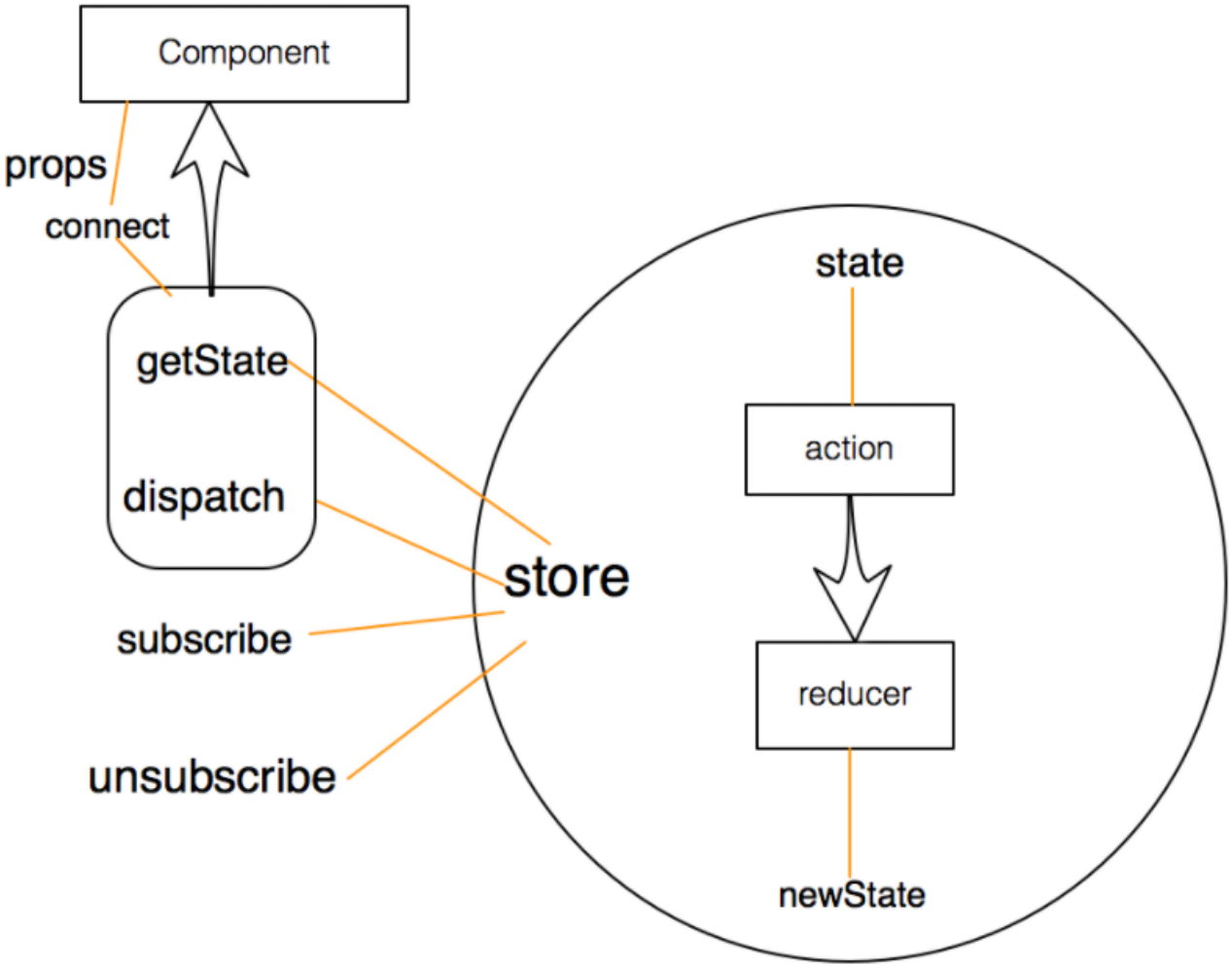
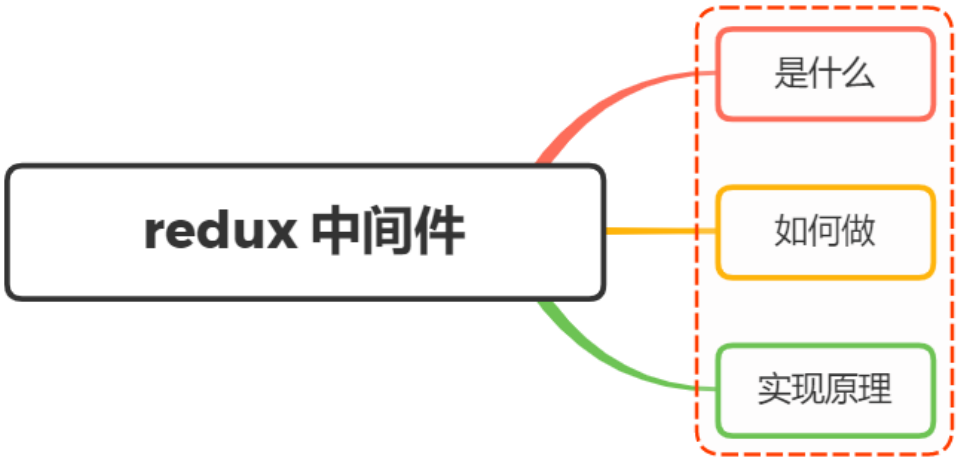


29. 说说对Redux中间件的理解？常用的中间件有哪些？实现原理？

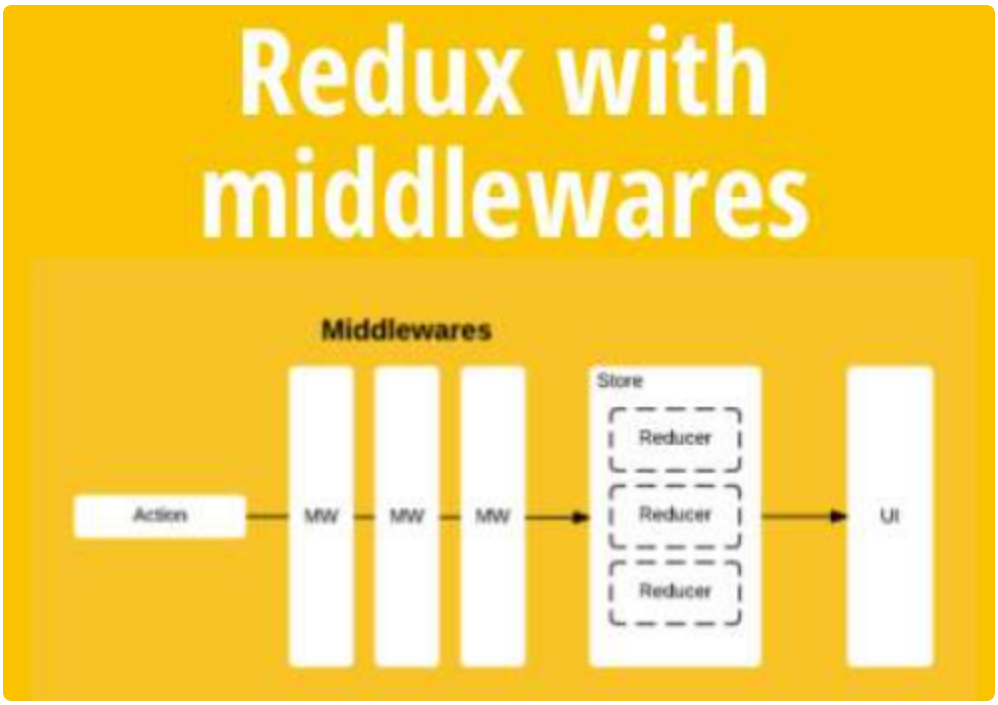


29.1. 是什么

中间件（Middleware）是介于应用系统和系统软件之间的一类软件，它使用系统软件所提供的基础服务（功能），衔接网络上应用系统的各个部分或不同的应用，能够达到资源共享、功能共享的目的

那么如果需要支持异步操作，或者支持错误处理、日志监控，这个过程就可以用上中间件

`Redux` 中，中间件就是放在就是在 `dispatch` 过程，在分发 `action` 进行拦截处理，如下图：



其本质上一个函数，对 `store.dispatch` 方法进行了改造，在发出 `Action` 和执行 `Reducer` 这两步之间，添加了其他功能

29.2. 常用的中间件

有很多优秀的 `redux` 中间件，如：

- redux-thunk：用于异步操作
- redux-logger：用于日志记录

上述的中间件都需要通过 `applyMiddlewares` 进行注册，作用是将所有的中间件组成一个数组，依次执行

然后作为第二个参数传入到 `createStore` 中

```JavaScript
const store = createStore(
  reducer,
  applyMiddleware(thunk, logger)
);
```

29.2.1. redux-thunk

`redux-thunk` 是官网推荐的异步处理中间件

默认情况下的 `dispatch(action)` ， `action` 需要是一个 `JavaScript` 的对象

`redux-thunk` 中间件会判断你当前传进来的数据类型，如果是一个函数，将会给函数传入参数值（dispatch，getState）

- dispatch函数用于我们之后再次派发action
- getState函数考虑到我们之后的一些操作需要依赖原来的状态，用于让我们可以获取之前的一些状态

所以 `dispatch` 可以写成下述函数的形式：

```JavaScript
const getHomeMultidataAction = () => {
  return (dispatch) => {
    axios.get("http://xxx.xx.xx.xx/test").then(res => {
      const data = res.data.data;
      dispatch(changeBannersAction(data.banner.list));
      dispatch(changeRecommendsAction(data.recommend.list));
    })
  }
}
```

29.2.2. redux-logger

如果想要实现一个日志功能，则可以使用现成的 `redux-logger`

```JavaScript
import { applyMiddleware, createStore } from 'redux';
import createLogger from 'redux-logger';
const logger = createLogger();

const store = createStore(
  reducer,
  applyMiddleware(logger)
);
```

这样我们就能简单通过中间件函数实现日志记录的信息

29.3. 实现原理

首先看看 `applyMiddlewares` 的源码

```JavaScript
export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, preloadedState, enhancer) => {
    var store = createStore(reducer, preloadedState, enhancer);
    var dispatch = store.dispatch;
    var chain = [];

    var middlewareAPI = {
      getState: store.getState,
      dispatch: (action) => dispatch(action)
    };
    chain = middlewares.map(middleware => middleware(middlewareAPI));
    dispatch = compose(...chain)(store.dispatch);

    return {...store, dispatch}
  }
}
```

所有中间件被放进了一个数组 `chain` ，然后嵌套执行，最后执行 `store.dispatch` 。可以看到，中间件内部（ `middlewareAPI` ）可以拿到 `getState` 和 `dispatch` 这两个方法

在上面的学习中，我们了解到了 `redux-thunk` 的基本使用

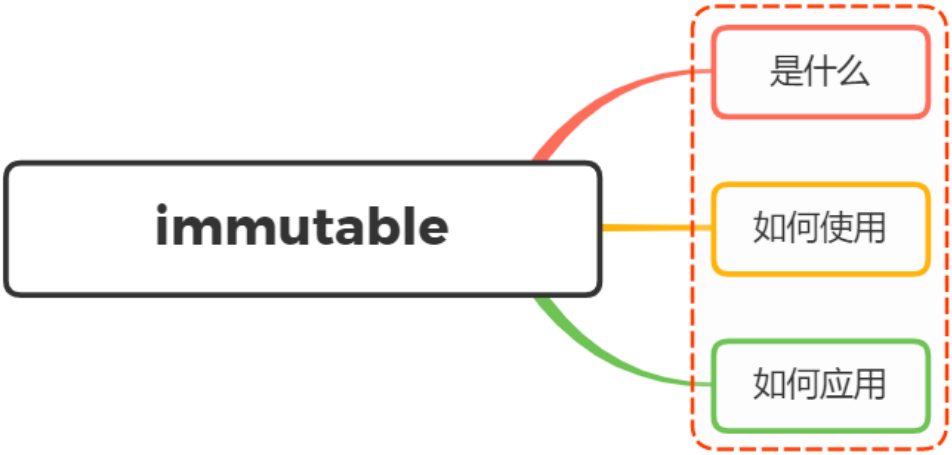
内部会将 `dispatch` 进行一个判断，然后执行对应操作，原理如下：

```javascript
function patchThunk(store) {
    let next = store.dispatch;

    function dispatchAndThunk(action) {
        if (typeof action === "function") {
            action(store.dispatch, store.getState);
        } else {
            next(action);
        }
    }

    store.dispatch = dispatchAndThunk;
}
```

实现一个日志输出的原理也非常简单，如下：

```javascript
let next = store.dispatch;

function dispatchAndLog(action) {
  console.log("dispatching:", addAction(10));
  next(addAction(5));
  console.log("新的state:", store.getState());
}

store.dispatch = dispatchAndLog;
```

30. 说说你对immutable的理解？如何应用在react项目中？



30.1. 是什么

Immutable，不可改变的，在计算机中，即指一旦创建，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

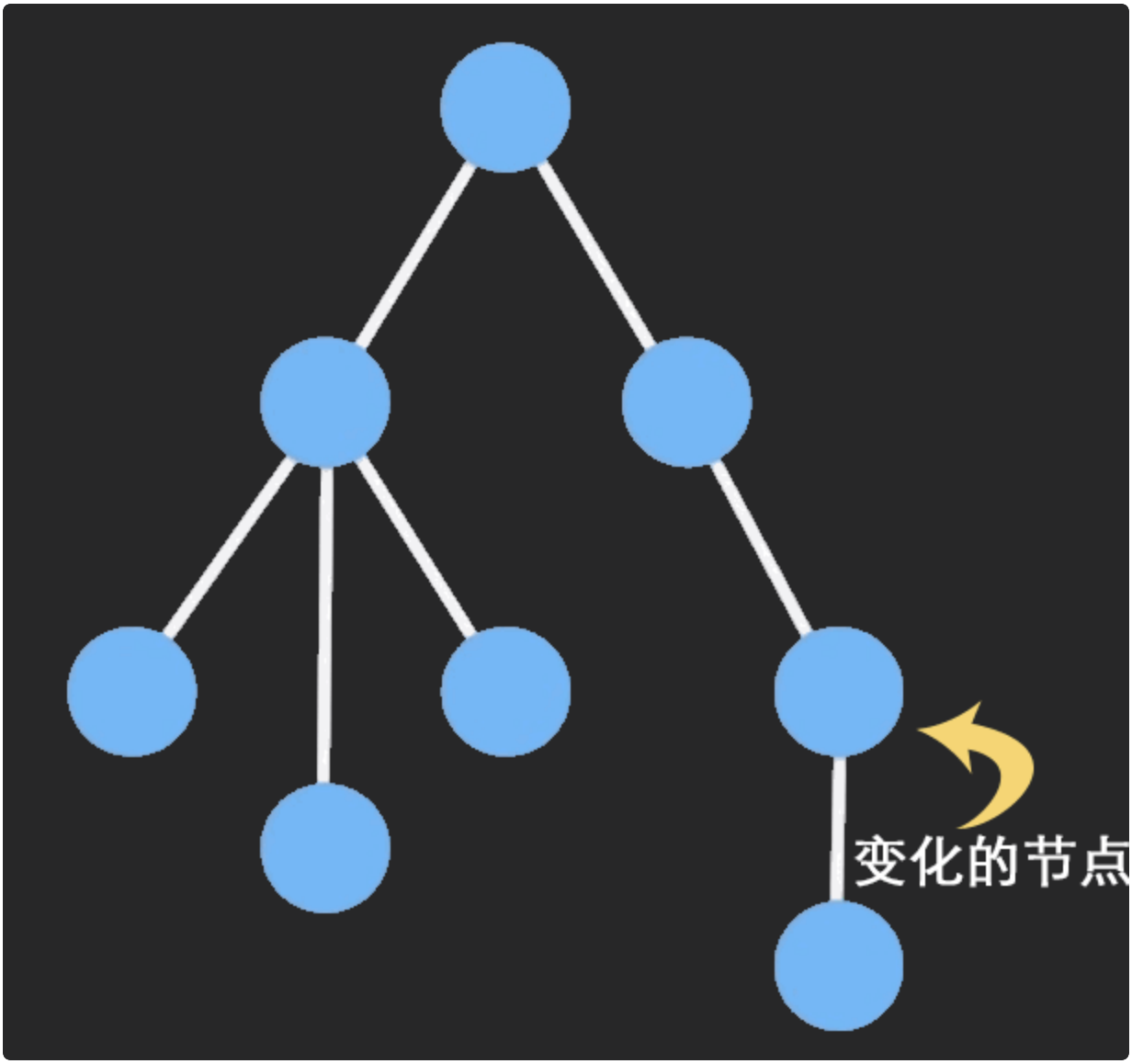
`Immutable` 实现的原理是 `Persistent Data Structure` （持久化数据结构）：

- 用一种数据结构来保存数据
- 当数据被修改时，会返回一个对象，但是新的对象会尽可能的利用之前的数据结构而不会对内存造成浪费

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）

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

如下图所示：



30.2. 如何使用

使用 `Immutable` 对象最主要的库是 `immutable.js`

immutable.js 是一个完全独立的库，无论基于什么框架都可以用它

其出现场景在于弥补 Javascript 没有不可变数据结构的问题，通过 structural sharing来解决的性能问题

内部提供了一套完整的 Persistent Data Structure，还有很多易用的数据类型，如 `Collection` 、 `List` 、 `Map` 、 `Set` 、 `Record` 、 `Seq` ，其中：

- List: 有序索引集，类似 JavaScript 中的 Array
- Map: 无序索引集，类似 JavaScript 中的 Object
- Set: 没有重复值的集合

主要的方法如下：

- fromJS()：将一个js数据转换为Immutable类型的数据

```
const obj = Immutable.fromJS({a:'123',b:'234'})
```

- toJS()：将一个Immutable数据转换为JS类型的数据
- is()：对两个对象进行比较

```
import { Map, is } from 'immutable'
const map1 = Map({ a: 1, b: 1, c: 1 })
const map2 = Map({ a: 1, b: 1, c: 1 })
map1 === map2   //false
Object.is(map1, map2) // false
is(map1, map2) // true
```

- get(key)：对数据或对象取值
- getIn([])：对嵌套对象或数组取值，传参为数组，表示位置

```
let abs = Immutable.fromJS({a: {b:2}});
abs.getIn(['a', 'b']) // 2
abs.getIn(['a', 'c']) // 子级没有值

let arr = Immutable.fromJS([1 ,2, 3, {a: 5}]);
arr.getIn([3, 'a']); // 5
arr.getIn([3, 'c']); // 子级没有值
```

如下例子：使用方法如下：

```
import Immutable from 'immutable';
foo = Immutable.fromJS({a: {b: 1}});
bar = foo.setIn(['a', 'b'], 2);   // 使用 setIn 赋值
console.log(foo.getIn(['a', 'b']));  // 使用 getIn 取值，打印 1
console.log(foo === bar);  //  打印 false
```

如果换到原生的 `js`，则对应如下：

```javascript
let foo = {a: {b: 1}};
let bar = foo;
bar.a.b = 2;
console.log(foo.a.b);  // 打印 2
console.log(foo === bar);  //  打印 true
```

30.3. 在React中应用

使用 `Immutable` 可以给 `React` 应用带来性能的优化，主要体现在减少渲染的次数

在做 `react` 性能优化的时候，为了避免重复渲染，我们会在 `shouldComponentUpdate()` 中做对比，当返回 `true` 执行 `render` 方法

`Immutable` 通过 `is` 方法则可以完成对比，而无需像一样通过深度比较的方式比较

在使用 `redux` 过程中也可以结合 `Immutable` ，不使用 `Immutable` 前修改一个数据需要做一个深拷贝

```jsx
import '_' from 'lodash';

const Component = React.createClass({
  getInitialState() {
    return {
      data: { times: 0 }
    }
  },
  handleAdd() {
    let data = _.cloneDeep(this.state.data);
    data.times = data.times + 1;
    this.setState({ data: data });
  }
}
```

使用 Immutable 后：

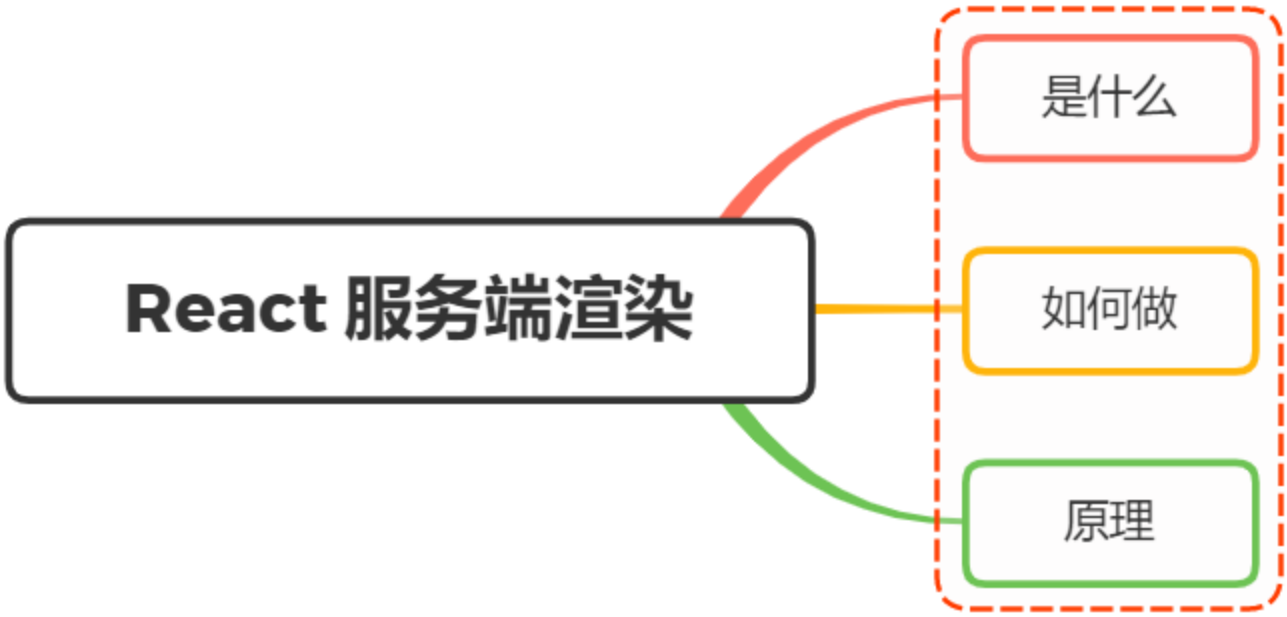
```jsx
getInitialState() {
  return {
    data: Map({ times: 0 })
  }
},
  handleAdd() {
    this.setState({ data: this.state.data.update('times', v => v + 1) });
    // 这时的 times 并不会改变
    console.log(this.state.data.get('times'));
  }
```

同理，在 `redux` 中也可以将数据进行 `fromJS` 处理

JavaScript | 复制代码

```
import * as constants from './constants'
import {fromJS} from 'immutable'
const defaultState = fromJS({ //将数据转化成immutable数据
    home:true,
    focused:false,
    mouseIn:false,
    list:[],
    page:1,
    totalPage:1
})
export default(state=defaultState,action)=>{
    switch(action.type){
        case constants.SEARCH_FOCUS:
            return state.set('focused',true) //更改immutable数据
        case constants.CHANGE_HOME_ACTIVE:
            return state.set('home',action.value)
        case constants.SEARCH_BLUR:
            return state.set('focused',false)
        case constants.CHANGE_LIST:
            // return state.set('list',action.data).set('totalPage',actio
n.totalPage)
            //merge效率更高，执行一次改变多个数据
            return state.merge({
                list:action.data,
                totalPage:action.totalPage
            })
        case constants.MOUSE_ENTER:
            return state.set('mouseIn',true)
        case constants.MOUSE_LEAVE:
            return state.set('mouseIn',false)
        case constants.CHANGE_PAGE:
            return state.set('page',action.page)
        default:
            return state
    }
}
```

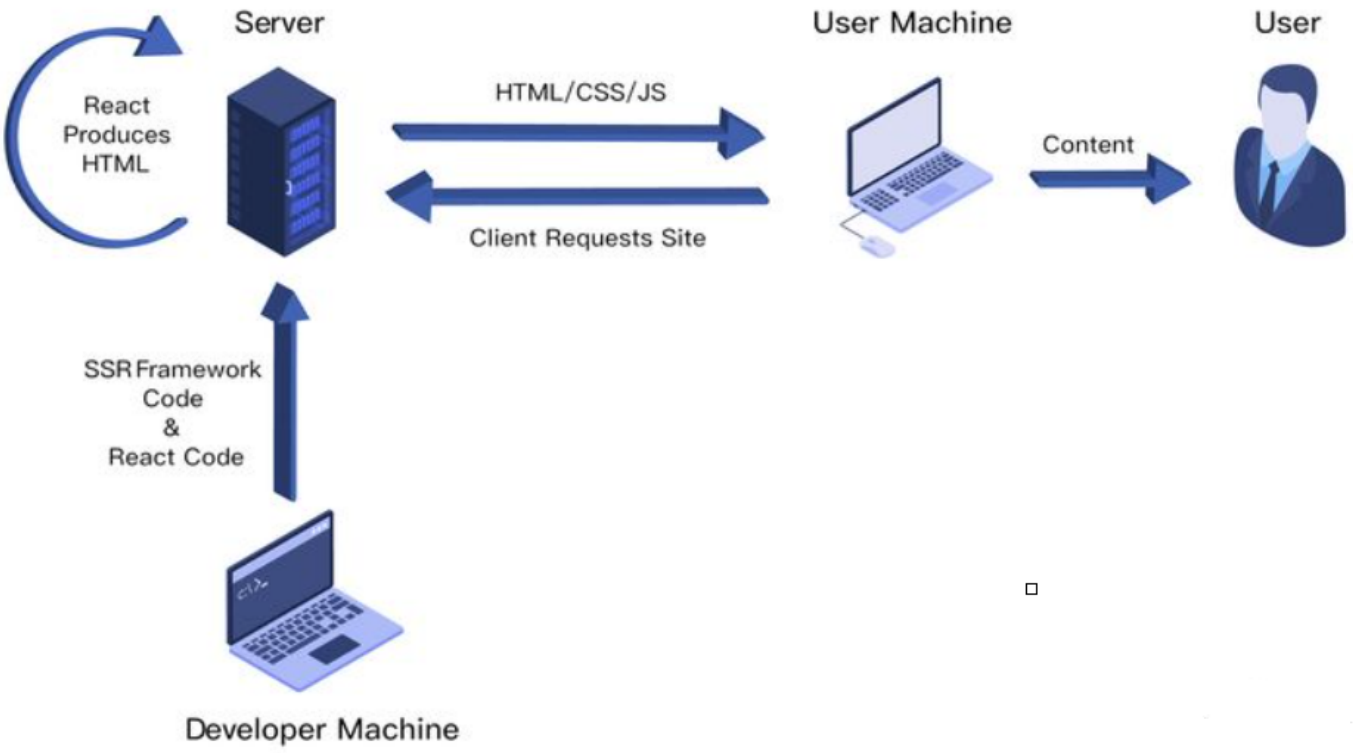
31. 说说React服务端渲染怎么做？原理是什么？



31.1. 是什么

`Server-Side Rendering` ，简称 `SSR` ，意为服务端渲染

指由服务侧完成页面的 `HTML` 结构拼接的页面处理技术，发送到浏览器，然后为其绑定状态与事件，成为完全可交互页面的过程



其解决的问题主要有两个：

- SEO，由于搜索引擎爬虫抓取工具可以直接查看完全渲染的页面
- 加速首屏加载，解决首屏白屏问题

31.2. 如何做

在 `react` 中，实现 `SSR` 主要有两种形式：

- 手动搭建一个 SSR 框架
- 使用成熟的SSR 框架，如 Next.JS

这里主要以手动搭建一个 `SSR` 框架进行实现

首先通过 `express` 启动一个 `app.js` 文件，用于监听3000端口的请求，当请求根目录时，返回 `HTML` ，如下：

JavaScript | 复制代码

```javascript
const express = require('express')
const app = express()
app.get('/', (req,res) => res.send(`
<html>
   <head>
       <title>ssr demo</title>
   </head>
   <body>
       Hello world
   </body>
</html>
`))

app.listen(3000, () => console.log('Exampleapp listening on port 3000!'))
```

然后再服务器中编写 `react` 代码，在 `app.js` 中进行应引用

JSX | 复制代码

```jsx
import React from 'react'

const Home = () =>{

    return <div>home</div>

}

export default Home
```

为了让服务器能够识别 `JSX`，这里需要使用 `webpakc` 对项目进行打包转换，创建一个配置文件 `webpack.server.js` 并进行相关配置，如下：

JavaScript | 复制代码

```
const path = require('path')    //node的path模块
const nodeExternals = require('webpack-node-externals')

module.exports = {
    target:'node',
    mode:'development',           //开发模式
    entry:'./app.js',             //入口
    output: {                     //打包出口
        filename:'bundle.js',     //打包后的文件名
        path:path.resolve(__dirname,'build')    //存放到根目录的build文件夹
    },
    externals: [nodeExternals()],  //保持node中require的引用方式
    module: {
        rules: [{                  //打包规则
           test:   /\.js?$/,       //对所有js文件进行打包
           loader:'babel-loader',  //使用babel-loader进行打包
           exclude: /node_modules/,//不打包node_modules中的js文件
           options: {
               presets: ['react','stage-0',['env', {
                                   //loader时额外的打包规则,对react,JSX，ES6进行转换
                    targets: {
                        browsers: ['last 2versions']   //对主流浏览器最近两个版本进行兼容
                    }
               }]]
           }
       }]
    }
}
```

接着借助 `react-dom` 提供了服务端渲染的 `renderToString` 方法，负责把 `React` 组件解析成 `html`

JavaScript | 复制代码

```
import express from 'express'
import React from 'react'//引入React以支持JSX的语法
import { renderToString } from 'react-dom/server'//引入renderToString方法
import Home from'./src/containers/Home'

const app= express()
const content = renderToString(<Home/>)
app.get('/',(req,res) => res.send(`
<html>
   <head>
       <title>ssr demo</title>
   </head>
   <body>
        ${content}
   </body>
</html>
`))

app.listen(3001, () => console.log('Exampleapp listening on port 3001!'))
```

上面的过程中，已经能够成功将组件渲染到了页面上

但是像一些事件处理的方法，是无法在服务端完成，因此需要将组件代码在浏览器中再执行一遍，这种服务器端和客户端共用一套代码的方式就称之为**同构**

重构通俗讲就是一套React代码在服务器上运行一遍，到达浏览器又运行一遍：

- 服务端渲染完成页面结构
- 浏览器端渲染完成事件绑定

浏览器实现事件绑定的方式为让浏览器去拉取 `JS` 文件执行，让 `JS` 代码来控制，因此需要引入 `script` 标签

通过 `script` 标签为页面引入客户端执行的 `react` 代码，并通过 `express` 的 `static` 中间件为 `js` 文件配置路由，修改如下：

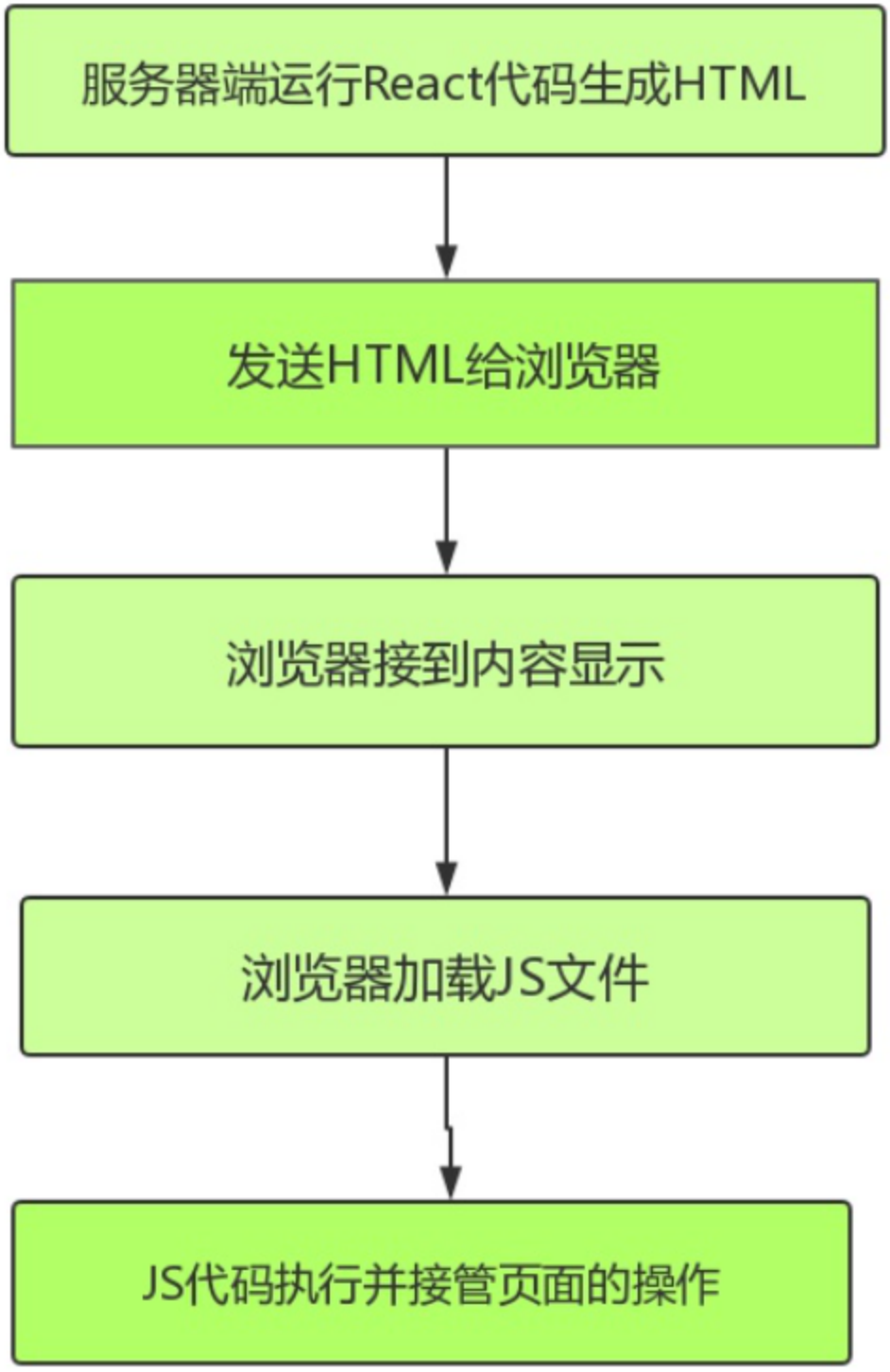
JavaScript | 复制代码

```
import express from 'express'
import React from 'react'//引入React以支持JSX的语法
import { renderToString } from'react-dom/server'//引入renderToString方法
import Home from './src/containers/Home'

const app = express()
app.use(express.static('public'));
//使用express提供的static中间件,中间件会将所有静态文件的路由指向public文件夹
 const content = renderToString(<Home/>)

app.get('/',(req,res)=>res.send(`
<html>
   <head>
       <title>ssr demo</title>
   </head>
   <body>
        ${content}
   <script src="/index.js"></script>
   </body>
</html>
`))

 app.listen(3001, () =>console.log('Example app listening on port 3001!'))
```

然后再客户端执行以下 `react` 代码，新建 `webpack.client.js` 作为客户端React代码的 `webpack` 配置文件如下：

```
const path = require('path')                        //node的path模块

module.exports = {
    mode:'development',                             //开发模式
    entry:'./src/client/index.js',                  //入口
    output: {                                       //打包出口
        filename:'index.js',                        //打包后的文件名
        path:path.resolve(__dirname,'public')       //存放到根目录的build文件夹
    },
    module: {
        rules: [{                                   //打包规则
           test:   /\.js?$/,                        //对所有js文件进行打包
           loader:'babel-loader',                   //使用babel-loader进行打包
           exclude: /node_modules/,                 //不打包node_modules中的js文件
           options: {
               presets: ['react','stage-0',['env', {
                    //loader时额外的打包规则,这里对react,JSX进行转换
                    targets: {
                        browsers: ['last 2versions']   //对主流浏览器最近两个版本进行兼容
                    }
               }]]
           }
       }]
    }
}
```

这种方法就能够简单实现首页的 `react` 服务端渲染，过程对应如下图：



在做完初始渲染的时候，一个应用会存在路由的情况，配置信息如下：

```JavaScript
import React from 'react'                    //引入React以支持JSX
import { Route } from 'react-router-dom'     //引入路由
import Home from './containers/Home'         //引入Home组件

export default (
    <div>
        <Route path="/" exact component={Home}></Route>
    </div>
)
```

然后可以通过 `index.js` 引用路由信息，如下：

```JavaScript
import React from 'react'
import ReactDom from 'react-dom'
import { BrowserRouter } from'react-router-dom'
import Router from'../Routers'

const App= () => {
    return (
        <BrowserRouter>
           {Router}
        </BrowserRouter>
    )
}

ReactDom.hydrate(<App/>, document.getElementById('root'))
```

这时候控制台会存在报错信息，原因在于每个 `Route` 组件外面包裹着一层 `div` ，但服务端返回的代码中并没有这个 `div`

解决方法只需要将路由信息在服务端执行一遍，使用使用 `StaticRouter` 来替代 `BrowserRouter` ，通过 `context` 进行参数传递

JavaScript | 复制代码

```
import express from 'express'
import React from 'react'//引入React以支持JSX的语法
import { renderToString } from 'react-dom/server'//引入renderToString方法
import { StaticRouter } from 'react-router-dom'
import Router from '../Routers'

const app = express()
app.use(express.static('public'));
//使用express提供的static中间件,中间件会将所有静态文件的路由指向public文件夹

app.get('/',(req,res)=>{
    const content  = renderToString((
        //传入当前path
        //context为必填参数,用于服务端渲染参数传递
        <StaticRouter location={req.path} context={{}}>
           {Router}
        </StaticRouter>
    ))
    res.send(`
   <html>
       <head>
           <title>ssr demo</title>
       </head>
       <body>
       <div id="root">${content}</div>
       <script src="/index.js"></script>
       </body>
   </html>
    `)
})


app.listen(3001, () => console.log('Exampleapp listening on port 3001!'))
```

这样也就完成了路由的服务端渲染

31.3. 原理

整体 `react` 服务端渲染原理并不复杂，具体如下：

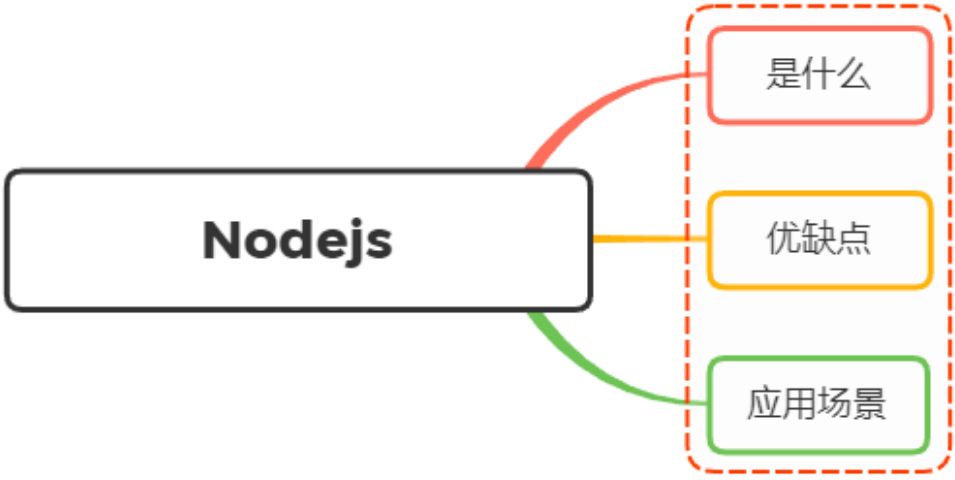
`node server` 接收客户端请求，得到当前的请求 `url` 路径，然后在已有的路由表内查找到对应的组件，拿到需要请求的数据，将数据作为 `props` 、 `context` 或者 `store` 形式传入组件

然后基于 `react` 内置的服务端渲染方法 `renderToString()` 把组件渲染为 `html` 字符串在把最终的 `html` 进行输出前需要将数据注入到浏览器端

浏览器开始进行渲染和节点对比，然后执行完成组件内事件绑定和一些交互，浏览器重用了服务端输出的 `html` 节点，整个流程结束

Node.js面试真题（14题）

1. 说说你对Node.js 的理解？优缺点？应用场景？



1.1. 是什么

`Node.js` 是一个开源与跨平台的 `JavaScript` 运行时环境

在浏览器外运行 V8 JavaScript 引擎（Google Chrome 的内核），利用事件驱动、非阻塞和异步输入输出模型等技术提高性能

可以理解为 `Node.js` 就是一个服务器端的、非阻塞式I/O的、事件驱动的 `JavaScript` 运行环境

1.1.1. 非阻塞异步

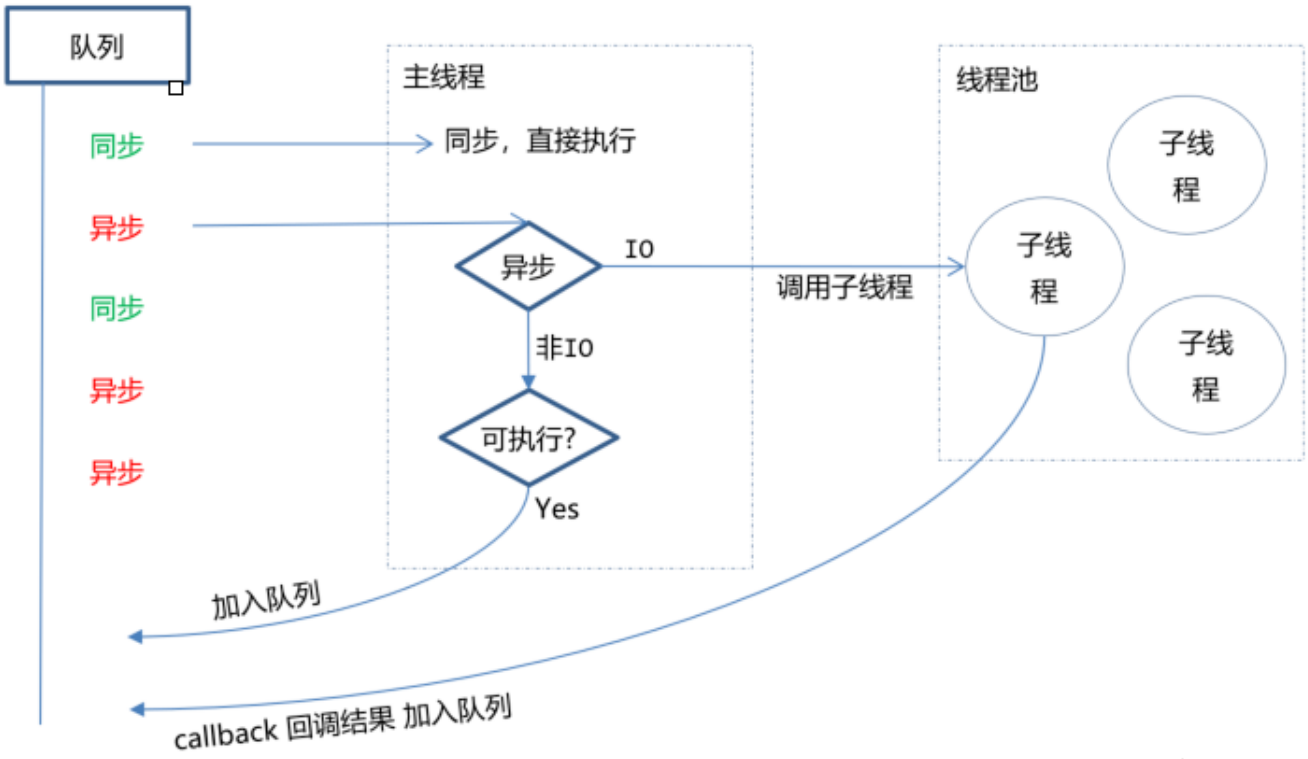
`Nodejs` 采用了非阻塞型 `I/O` 机制，在做 `I/O` 操作的时候不会造成任何的阻塞，当完成之后，以时间的形式通知执行操作

例如在执行了访问数据库的代码之后，将立即转而执行其后面的代码，把数据库返回结果的处理代码放在回调函数中，从而提高了程序的执行效率

1.1.2. 事件驱动

事件驱动就是当进来一个新的请求的时，请求将会被压入一个事件队列中，然后通过一个循环来检测队列中的事件状态变化，如果检测到有状态变化的事件，那么就执行该事件对应的处理代码，一般都是回调函数

比如读取一个文件，文件读取完毕后，就会触发对应的状态，然后通过对应的回调函数来进行处理



1.2. 优缺点

优点：

- 处理高并发场景性能更佳
- 适合I/O密集型应用，值的是应用在运行极限时，CPU占用率仍然比较低，大部分时间是在做 I/O硬盘内存读写操作

因为 `Nodejs` 是单线程，带来的缺点有：

- 不适合CPU密集型应用
- 只支持单核CPU，不能充分利用CPU
- 可靠性低，一旦代码某个环节崩溃，整个系统都崩溃

1.3. 应用场景

借助 `Nodejs` 的特点和弊端，其应用场景分类如下：

- 善于 `I/O` ，不善于计算。因为Nodejs是一个单线程，如果计算（同步）太多，则会阻塞这个线程
- 大量并发的I/O，应用程序内部并不需要进行非常复杂的处理