## **1. 什么是HMR**

* Hot Module Replacement是指当你对代码修改并保存后，webpack将会对代码进行得新打包，并将新的模块发送到浏览器端，浏览器用新的模块替换掉旧的模块，以实现在不刷新浏览器的前提下更新页面。
* 相对于live reload刷新页面的方案，HMR的优点在于可以保存应用的状态,提高了开发效率

## **2. 搭建HMR项目**

### **2.1 安装依赖的模块**

cnpm i webpack@4.39.1 webpack-cli@3.3.6 webpack-dev-server@3.7.2 mime html-webpack-plugin express socket.io -S

### **2.2 package.json**

package.json

{

"name": "zhufeng\_hmr",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"build": "webpack",

"dev": "webpack-dev-server"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

"webpack": "4.39.1",

"webpack-cli": "3.3.6",

"webpack-dev-server": "3.7.2"

}

}

### **2.2 webpack.config.js**

webpack.config.js

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

mode:'development',

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

output: {

filename: 'main.js',

path: path.join(\_\_dirname, 'dist')

},

devServer: {

contentBase:path.join(\_\_dirname, 'dist')

},

plugins:[

**new** HtmlWebpackPlugin({

template:'./src/index.html',

filename:'index.html'

})

]

}

### **2.3 src\index.js**

src\index.js

**let** root = document.getElementById('root');**function** **render**(){

**let** title = require('./title').default;

root.innerHTML= title;

}

render();

### **2.4 src\title.js**

src\title.js

**export** **default** 'hello';

### **2.5 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>webpack热更新</title>

</head>

<body>

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

</body>

</html>

### **2.6 dist\bundle.js**

dist\main.js

(**function**(modules) {

**var** installedModules = {};

//封装的客户端的require方法

**function** **\_\_webpack\_require\_\_**(moduleId) {

//判断此模块是否在缓存中

**if**(installedModules[moduleId]) {

//如果在缓存中的话返回缓存模块的导出对象

**return** installedModules[moduleId].exports;

}

//创建一个模块并且把它放在缓存中

**var** module = installedModules[moduleId] = {

i: moduleId,//模块ID

l: false,//是否已经加载false

exports: {}//导出对象默认为空对象

};

//执行模块函数

modules[moduleId].call(module.exports, module, module.exports, \_\_webpack\_require\_\_);

//把模块标识为已经加载

module.l = true;

//返回模块的导出对象

**return** module.exports;

}

//暴露模块对象并挂载到\_\_webpack\_require\_\_.m属性上

\_\_webpack\_require\_\_.m = modules;

//暴露已经安装的模块到模板的缓存上

\_\_webpack\_require\_\_.c = installedModules;

// define getter function for harmony exports

//在exports对象上定义name属性的getter方法

\_\_webpack\_require\_\_.d = **function**(exports, name, getter) {

//判断exports对象上是否有name属性

**if**(!\_\_webpack\_require\_\_.o(exports, name)) {

//在exports对象上添加name属性，可枚举为true,get为getter,当访问该属性时，该方法会被执行，

Object.defineProperty(exports, name, { enumerable: true, get: getter });

}

};

//在导出对象上定义\_\_esModule属性

\_\_webpack\_require\_\_.r = **function**(exports) {

//如果有这样的Symbol的话

**if**(**typeof** Symbol !== 'undefined' && Symbol.toStringTag) {

Object.defineProperty(exports, Symbol.toStringTag, { value: 'Module' });

}//否则定义一个\_\_esModule属性

Object.defineProperty(exports, '\_\_esModule', { value: true });

};

// create a fake namespace object 创建一个命名空间对象

// mode & 1: value is a module id, require it 值是模块ID，加载它

// mode & 2: merge all properties of value into the ns 把所有的属性合并到命名空间上

// mode & 4: return value when already ns object 当已经是命名空间对象的话直接返回值

// mode & 8|1: behave like require 就像require一样

\_\_webpack\_require\_\_.t = **function**(value, mode) {

**if**(mode & 1) value = \_\_webpack\_require\_\_(value);//直接加载

**if**(mode & 8) **return** value; //不用加载，直接返回

//如果value已经是ns对象并且\_\_esModule属性为true的话就直接返回value

**if**((mode & 4) && **typeof** value === 'object' && value && value.\_\_esModule) **return** value;

//否则就创建一个空对象,加载这个对象,

**var** ns = Object.create(null);

\_\_webpack\_require\_\_.r(ns);//在对象上设置\_\_esModule属性为true

Object.defineProperty(ns, 'default', { enumerable: true, value: value });

//如果mode为2，并且value不是字符串，把值的所有属性都定义到ns对象上

**if**(mode & 2 && **typeof** value != 'string') **for**(**var** key **in** value) \_\_webpack\_require\_\_.d(ns, key, **function**(key) { **return** value[key]; }.bind(null, key));

//返回ns对象

**return** ns;

};

// getDefaultExport function for compatibility with non-harmony modules

\_\_webpack\_require\_\_.n = **function**(module) {

//如果是\_\_esModule,getDefault方法返回模块的default属性,getModuleExports返回模块本身

**var** getter = module && module.\_\_esModule ?

**function** **getDefault**() { **return** module['default']; } :

**function** **getModuleExports**() { **return** module; };

//给getter添加一个a的属性，就是gett方法本身

\_\_webpack\_require\_\_.d(getter, 'a', getter);

**return** getter;

};

// Object.prototype.hasOwnProperty.call 判断对象上是否有属性 o=own

\_\_webpack\_require\_\_.o = **function**(object, property) { **return** Object.prototype.hasOwnProperty.call(object, property); };

// \_\_webpack\_public\_path\_\_ 公开访问路径

\_\_webpack\_require\_\_.p = "";

// Load entry module and return exports 加载入口模块并且返回导出对象

**return** \_\_webpack\_require\_\_(\_\_webpack\_require\_\_.s = "./src/index.js");

})

({

"./src/index.js":

(**function**(module, \_\_webpack\_exports\_\_, \_\_webpack\_require\_\_) {eval(`

\_\_webpack\_require\_\_.r(\_\_webpack\_exports\_\_);//因为是es模块，所以要添加\_\_esModule属性

var \_title\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_ = \_\_webpack\_require\_\_(\"./src/title.js\");

function render(){

let root = document.getElementById('root');

root.innerHTML= \_title\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_[\"default\"];

}

render();`);

}),

"./src/title.js":

(**function**(module, \_\_webpack\_exports\_\_, \_\_webpack\_require\_\_) {eval(`

\_\_webpack\_require\_\_.r(\_\_webpack\_exports\_\_);//因为是es模块，所以要添加\_\_esModule属性

\_\_webpack\_exports\_\_[\"default\"] = ('hello');

`);

})

});

## **3. webpack的编译流程**

* 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数； 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；
* 确定入口：根据配置中的 entry 找出所有的入口文件；
* 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；
* 完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；
* 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
* 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

在以上过程中，Webpack 会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变 Webpack 的运行结果。 chunk 就是若干 module 打成的包，一个 chunk 应该包括多个 module，一般来说最终会形成一个 file。而 js 以外的资源，webpack 会通过各种 loader 转化成一个 module，这个模块会被打包到某个 chunk 中，并不会形成一个单独的 chunk

## **3. 实现热更新**

### **3.1 webpack.config.js**

webpack.config.js

module.exports = {

devServer:{+ hot:true,

contentBase:path.join(\_\_dirname,'dist')

},

plugins:[+ new webpack.HotModuleReplacementPlugin()

]

}

### **3.2 index.js**

src\index.js

import './client';

let root = document.getElementById('root');

function render(){

let title = require('./title').default;

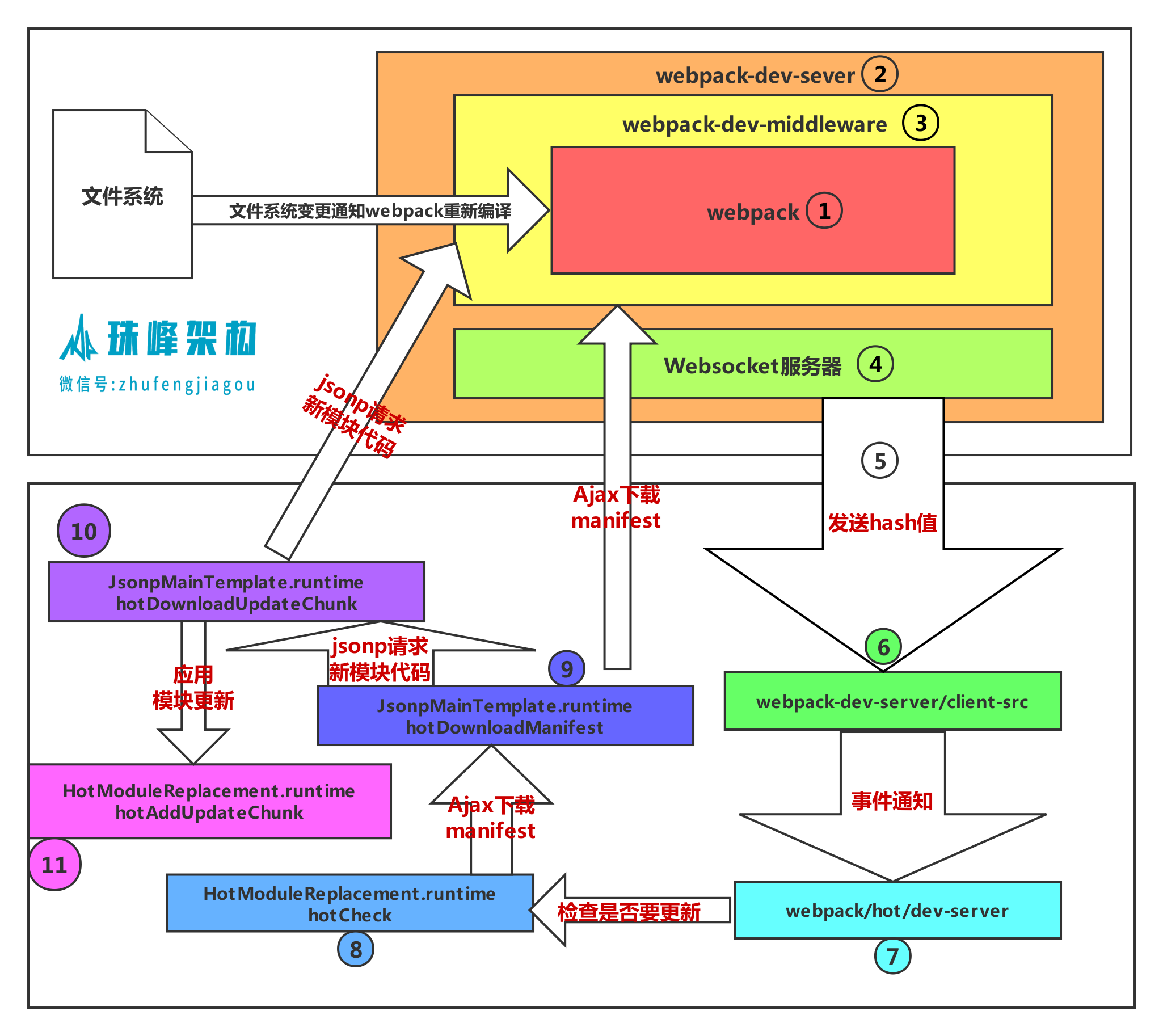
root.innerHTML= title;

}

render();

+if(module.hot){+ module.hot.accept(['./title'],()=>{+ render();+ });+}

## **4. debug**



debugger.js

**debugger**require('./node\_modules/webpack-dev-server/bin/webpack-dev-server.js');

## **5. 源代码位置**

### **5.1. 服务器部分**

1. 启动webpack-dev-server服务器
2. 创建webpack实例
3. 创建Server服务器
4. 添加webpack的done事件回调，在编译完成后会向浏览器发送消息
5. 创建express应用app
6. 使用监控模式开始启动webpack编译,在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中
7. 设置文件系统为内存文件系统
8. 添加webpack-dev-middleware中间件
9. 创建http服务器并启动服务
10. 使用sockjs在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端,浏览器端根据这些socket消息进行不同的操作。当然服务端传递的最主要信息还是新模块的hash值，后面的步骤根据这一hash值来进行模块热替换

| **步骤** | **代码位置** |
| --- | --- |
| 1.启动webpack-dev-server服务器 | [webpack-dev-server.js#L159](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/bin/webpack-dev-server.js" \l "L73) |
| 2.创建webpack实例 | [webpack-dev-server.js#L89](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/bin/webpack-dev-server.js" \l "L89) |
| 3.创建Server服务器 | [webpack-dev-server.js#L100](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/bin/webpack-dev-server.js" \l "L107) |
| 4. 添加webpack的done事件回调 | [Server.js#L120](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/Server.js" \l "L122) |
| 编译完成向客户端发送消息 | [Server.js#L183](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/Server.js" \l "L184) |
| 5.创建express应用app | [Server.js#L121](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/Server.js" \l "L123) |
| 6. 添加webpack-dev-middleware中间件 | [Server.js#L121](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/Server.js" \l "L121) |
| 中间件负责返回生成的文件 | [middleware.js#L20](https://github.com/webpack/webpack-dev-middleware/blob/v3.7.0/lib/middleware.js" \l "L20) |
| 启动webpack编译 | [index.js#L51](https://github.com/webpack/webpack-dev-middleware/blob/v3.7.0/index.js" \l "L51) |
| 7. 设置文件系统为内存文件系统 | [fs.js#L115](https://github.com/webpack/webpack-dev-middleware/blob/v3.7.0/lib/fs.js" \l "L115) |
| 8. 创建http服务器并启动服务 | [Server.js#L135](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/Server.js" \l "L135) |
| 9. 使用sockjs在浏览器端和服务端之间建立一个 websocket 长连接 | [Server.js#L745](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/Server.js" \l "L745) |
| 创建socket服务器 | [SockJSServer.js#L34](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/servers/SockJSServer.js" \l "L34) |

### **5.2. 客户端部分**

1. webpack-dev-server/client-src/default/index.js端会监听到此hash消息,会保存此hash值
2. 客户端收到ok的消息后会执行reloadApp方法进行更新
3. 在reloadApp中会进行判断，是否支持热更新，如果支持的话发射webpackHotUpdate事件,如果不支持则直接刷新浏览器
4. 在webpack/hot/dev-server.js会监听webpackHotUpdate事件,然后执行check()方法进行检查
5. 在check方法里会调用module.hot.check方法
6. 它通过调用 JsonpMainTemplate.runtime的hotDownloadManifest方法，向 server 端发送 Ajax 请求，服务端返回一个 Manifest文件，该 Manifest 包含了所有要更新的模块的 hash 值和chunk名
7. 调用JsonpMainTemplate.runtime的hotDownloadUpdateChunk方法通过JSONP请求获取到最新的模块代码
8. 补丁JS取回来后会调用JsonpMainTemplate.runtime.js的webpackHotUpdate方法，里面会调用hotAddUpdateChunk方法,用新的模块替换掉旧的模块
9. 然后会调用HotModuleReplacement.runtime.js的hotAddUpdateChunk方法动态更新模块代 码
10. 然后调用hotApply方法进行热更新

| **步骤** | **代码** |
| --- | --- |
| 1. webpack-dev-server/client端会监听到此hash消息 | [index.js#L54](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/client-src/default/index.js" \l "L54) |
| 2. 客户端收到ok的消息后会执行reloadApp方法进行更新 | [index.js#L101](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/client-src/default/index.js" \l "L101) |
| 3. 在reloadApp中会进行判断，是否支持热更新，如果支持的话发射webpackHotUpdate事件,如果不支持则直接刷新浏览器 | [reloadApp.js#L7](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/client-src/default/utils/reloadApp.js" \l "L7) |
| 4. 在webpack/hot/dev-server.js会监听webpackHotUpdate事件 | [dev-server.js#L55](https://github.com/webpack/webpack/blob/v4.39.1/hot/dev-server.js" \l "L55) |
| 5. 在check方法里会调用module.hot.check方法 | [dev-server.js#L13](https://github.com/webpack/webpack/blob/v4.39.1/hot/dev-server.js" \l "L13) |
| 6. HotModuleReplacement.runtime请求Manifest | [HotModuleReplacement.runtime.js#L180](https://github.com/webpack/webpack/blob/v4.39.1/lib/HotModuleReplacement.runtime.js" \l "L180) |
| 7. 它通过调用 JsonpMainTemplate.runtime的hotDownloadManifest方法 | [JsonpMainTemplate.runtime.js#L23](https://github.com/webpack/webpack/blob/v4.39.1/lib/web/JsonpMainTemplate.runtime.js" \l "L23) |
| 8. 调用JsonpMainTemplate.runtime的hotDownloadUpdateChunk方法通过JSONP请求获取到最新的模块代码 | [JsonpMainTemplate.runtime.js#L14](https://github.com/webpack/webpack/blob/v4.39.1/lib/web/JsonpMainTemplate.runtime.js" \l "L14) |
| 9. 补丁JS取回来后会调用JsonpMainTemplate.runtime.js的webpackHotUpdate方法 | [JsonpMainTemplate.runtime.js#L8](https://github.com/webpack/webpack/blob/v4.39.1/lib/web/JsonpMainTemplate.runtime.js" \l "L8) |
| 10. 然后会调用HotModuleReplacement.runtime.js的hotAddUpdateChunk方法动态更新模块代码 | [HotModuleReplacement.runtime.js#L222](https://github.com/webpack/webpack/blob/v4.39.1/lib/HotModuleReplacement.runtime.js" \l "L222) |
| 11.然后调用hotApply方法进行热更新 | [HotModuleReplacement.runtime.js#L257](https://github.com/webpack/webpack/blob/v4.39.1/lib/HotModuleReplacement.runtime.js" \l "L257) [HotModuleReplacement.runtime.js#L278](https://github.com/webpack/webpack/blob/v4.39.1/lib/HotModuleReplacement.runtime.js" \l "L278) |

### **5.3 相关代码**

* [webpack-dev-server.js](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/bin/webpack-dev-server.js)
* [Server.js](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/Server.js)
* [webpack-dev-middleware/index.js](https://github.com/webpack/webpack-dev-middleware/blob/v3.7.0/index.js)
* [SockJSServer.js](https://github.com/webpack/webpack-dev-server/blob/v3.7.2/lib/servers/SockJSServer.js)

## **6. 实现热更新**

### **6.1 webpack-dev-server.js**

**const** path = require('path');**const** express = require('express');**const** mime = require('mime');**const** webpack = require('webpack');//1. 启动webpack-dev-server服务器//2. 创建webpack实例**let** config = require('./webpack.config');**let** compiler = webpack(config);**class** **Server**{

**constructor**(compiler){

**this**.compiler = compiler;

//4. 添加webpack的`done`事件回调，在编译完成后会向浏览器发送消息

**let** lastHash;

**let** sockets=[];

compiler.hooks.done.tap('webpack-dev-server', (stats) => {

lastHash = stats.hash;

sockets.forEach(socket=>{

socket.emit('hash',stats.hash);

socket.emit('ok');

});

});

//5. 创建express应用app

**let** app = **new** express();

//6. 使用监控模式开始启动webpack编译,在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中

compiler.watch(config.watchOptions||{}, (err)=>{

console.log('编译成功');

});

//7. 设置文件系统为内存文件系统

**const** MemoryFileSystem = require('memory-fs');

**const** fs = **new** MemoryFileSystem();;

compiler.outputFileSystem = fs;

//8. 添加webpack-dev-middleware中间件

**const** devMiddleware = (req,res,next)=>{

**if**(req.url === '/favicon.ico'){

**return** res.sendStatus(404);

}

**let** filename = path.join(config.output.path,req.url.slice(1));

console.error(filename);

**if**(fs.statSync(filename).isFile()){

**let** content = fs.readFileSync(filename);

res.header('Content-Type',mime.getType(filename));

res.send(content);

}**else**{

next();

}

}

app.use(devMiddleware);

//8. 创建http服务器并启动服务

**this**.server = require('http').createServer(app);

// 10. 使用sockjs在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端,浏览器端根据这些`socket`消息进行不同的操作。当然服务端传递的最主要信息还是新模块的`hash`值，后面的步骤根据这一`hash`值来进行模块热替换

**let** io = require('socket.io')(**this**.server);

io.on('connection',(socket)=>{

sockets.push(socket);

**if**(lastHash){

socket.emit('hash',lastHash);

socket.emit('ok');

}

});

}

//9. 创建http服务器并启动服务

listen(port){

**this**.server.listen(port,()=>{

console.log(port+'服务启动成功!')

});

}

}//3. 创建Server服务器**let** server = **new** Server(compiler);

server.listen(8000);

### **6.2 client.js**

src\client.js

**let** socket = io('/');**class** **Emitter**{

**constructor**(){

**this**.listeners = {};

}

emit(type){

**this**.listeners[type]&&**this**.listeners[type]();

}

on(type,listener){

**this**.listeners[type] = listener;

}

}**const** hotEmitter= **new** Emitter();

**let** hotCurrentHash;**let** currentHash;**const** onConnected = ()=>{

console.log('客户端已经连接');

//1. `webpack-dev-server/client`端会监听到此hash消息

socket.on('hash',(hash)=>{

currentHash = hash;

});

//2. 客户端收到`ok`的消息后会执行`reloadApp`方法进行更新

socket.on('ok',()=>{

reloadApp(true);

});

socket.on('disconnect',()=>{

hotCurrentHash=currentHash=null;

});

}// 3. 在reloadApp中会进行判断，是否支持热更新，如果支持的话发射`webpackHotUpdate`事件,如果不支持则直接刷新浏览器**function** **reloadApp**(hot){

**if**(!hot){

**return** window.location.reload();

}

hotEmitter.emit('webpackHotUpdate');

}

//4. 在`webpack/hot/dev-server.js`会监听`webpackHotUpdate`事件,在监听里会调用hotCheck方法

hotEmitter.on("webpackHotUpdate", **function**() {

**if**(!hotCurrentHash || hotCurrentHash === currentHash){

**return** hotCurrentHash = currentHash;

}

//5. 在check方法里会调用`module.hot.check`方法

hotCheck();

});**function** **hotCheck**(){

//6. 它通过调用 `JsonpMainTemplate.runtime`的`hotDownloadManifest`方法，向 server 端发送 Ajax 请求，服务端返回一个 `Manifest`文件，该 `Manifest` 包含了所有要更新的模块的 `hash` 值和chunk名

hotDownloadManifest().then(update=>{

**let** chunkIds = Object.keys(update.c);

chunkIds.forEach((chunkId)=>{

//7. 调用`JsonpMainTemplate.runtime`的`hotDownloadUpdateChunk`方法通过JSONP请求获取到最新的模块代码

hotDownloadUpdateChunk(chunkId);

});

});

}

**function** **hotDownloadUpdateChunk**(chunkId) {

**var** script = document.createElement("script");

script.charset = "utf-8";

script.src = "/" + chunkId + "." + hotCurrentHash + ".hot-update.js";

document.head.appendChild(script);

}**function** **hotDownloadManifest**(){

**return** **new** Promise((resolve,reject)=>{

**var** request = **new** XMLHttpRequest();

**var** requestPath = "/" + hotCurrentHash + ".hot-update.json";

request.open("GET", requestPath, true);

request.onreadystatechange = **function**() {

**if**(request.readyState == 4){

**let** update = JSON.parse(request.responseText);

resolve(update);

}

}

request.send();

});

}//9. 补丁JS取回来后会调用`JsonpMainTemplate.runtime.js`的`webpackHotUpdate`方法，里面会调用`hotAddUpdateChunk`方法,用新的模块替换掉旧的模块//10. 然后会调用`HotModuleReplacement.runtime.js`的`hotAddUpdateChunk`方法动态更新模块代码//11. 然后调用`hotApply`方法进行热更新window.webpackHotUpdate = (chunkId, moreModules)=>{

**for**(**let** moduleId **in** moreModules){

**let** oldModule = \_\_webpack\_require\_\_.c[moduleId];

**let** {parents,children} = oldModule;

**var** module = \_\_webpack\_require\_\_.c[moduleId] = {

i: moduleId,

l: false,exports: {},

parents,children,

hot: window.hotCreateModule(moduleId)

};

moreModules[moduleId].call(module.exports, module, module.exports,\_\_webpack\_require\_\_);

module.l = true;

parents.forEach(parent=>{

**let** parentModule = \_\_webpack\_require\_\_.c[parent];

parentModule.hot&&parentModule.hot.\_acceptedDependencies[moduleId]&&parentModule.hot.\_acceptedDependencies[moduleId]();

});

hotCurrentHash = currentHash;

}

}

socket.on('connect',onConnected);

### **6.3 HotModuleReplacement.runtime.js**

webpack\lib\HotModuleReplacement.runtime.js

**function** **hotCreateModule**(moduleId) {

**var** hot = {

\_acceptedDependencies: {},

accept: **function**(dep, callback) {

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

hot.\_acceptedDependencies[dep[i]] = callback;

}

}

}

**return** hot;

}