ES6、ES7、Webpack和Babel

第2天课堂笔记

讲师：邵山欢

日期：2017年11月15日

# AMD规范和CMD规范

## 概述

这是我们经常写的HTML中的语句，引用了很多js文件，但是看不清他们彼此之间的依赖关系：

<script type="text/javascript" src="1.js"></script>

<script type="text/javascript" src="2.js"></script>

<script type="text/javascript" src="3.js"></script>

<script type="text/javascript" src="4.js"></script>

<script type="text/javascript" src="5.js"></script>

删除1.js、2.js之后，其他的js文件还能不能正常工作。

还会出现一个问题，就是js文件进行合并的时候，只能进行无脑合并。为了防止它们之间有变量名冲突，所以每个JS文件都加上IIFE的外壳：

(function(){

})();

(function(){

})();

(function(){

})();

(function(){

})();

(function(){

})();

这种无脑合并的隐患很大，比如一个sum函数其他的js中要用，此时是没有定义域的。

(function(){

//这个函数写在IIFE中，

function sum(){

}

})();

(function(){

//这里面没有sum的定义

alert(sum(3,4));

})();

(function(){

//也定义了一个sum，很乱

function sum(){

}

})();

所以这种合并，需要的不是IIFE，而是一个命名空间。

## 1.2 AMD规范 - require.js

Asynchronous Module Defination 异步模块定义。**代表库就是require.js**。

可以参考阮一峰的博客：http://www.ruanyifeng.com/blog/2012/11/require\_js.html

百度的静态资源库：http://cdn.code.baidu.com/ 上面可以下载jQuery、underscore等一些库。

### 1.2.1 最简单的DEMO

|  |
| --- |
| ┣ js  ┃ ┣ lib  ┃ ┃ ┣ require.js  ┃ ┣ main.js  ┣ index.html |

在main.js中写一条语句：

alert("你好");

index.html中：

<script type="text/javascript" **src="js/lib/require.min.js"** **data-main="js/main"**></script>

引入了库，在这个库的文件的script上，写data-main，路径是主入口的main.js文件，省略了扩展名.js。

### 1.2.2 暴露和引用

我们在js文件夹中创建yuan.js、fang.js的文件：

|  |
| --- |
| ┣ js  ┃ ┣ lib  ┃ ┃ ┣ require.js  ┃ ┣ main.js  ┃ ┣ yuan.js  ┃ ┣ fang.js  ┣ index.html |

yuan.js:

我们使用define()来包裹一个函数，define不是ECMAScript的语法，而是require.js的语法。

函数里面可以写任何语句，需要暴露的东西，写return来暴露。

define(function(){

function mianji(r){

return 3.14 \* r \* r;

}

return {

mianji //k、v相同，省略v

}

});

fang.js:

define(function(){

function mianji(a,b){

return a \* b;

}

return {

mianji

}

});

main.js：

主文件需要使用**“依赖注入”（dependencies injection）的方式引入这个模块。**

require(['yuan','fang'],function(y,f){

alert(y.mianji(10));

alert(f.mianji(10,20));

});

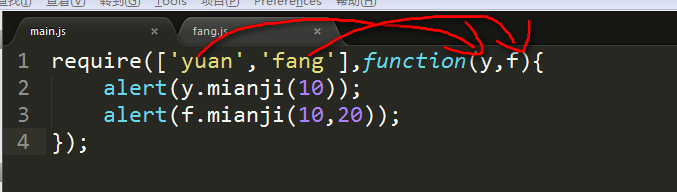
我们发现，依赖注入的语法是：

require(**[**"模块1","模块2","模块3"**]**,function(模块1,模块2,模块3){

});

注意：

1. 依赖的文件名必须有引号，注入的时候没有引号的；
2. 注入的时候的名字可以任取，所以require.js是通过依赖注入的顺序来产生对应关系的：



1. 依赖的时候，没有拓展名，模块的默认名字就是文件名。

### 1.2.3 别名

模块的名字默认是文件名。

require(['yuan','fang'],function(y,f){

alert(y.mianji(10));

alert(f.mianji(10,20));

});

如果我们想把依赖的名字改掉，此时可以使用require对象的config方法来配置paths项：

**require.config({**

**paths: {**

**"circle" : "yuan" //没有.js扩展名**

**}**

**});**

require([**'circle'**,'fang'],function(y,f){

alert(y.mianji(10));

alert(f.mianji(10,20));

});

### 1.2.4 设置暴露口

**AMD规范最强大的事情，就是在于一个普通的JS文件如果不符合AMD规范，可以设置暴露口**。

什么叫做符合AMD规范：如果一个库在创建的时候有这么一条语句（伪代码）：

if(define){

define(function(){

return {

$

}

})

}

此时就说明对require.js兼容了，此时叫做符合AMD规范。

比如我们引入jquery库：

require.config({

　　paths: {

　　　　"circle" : "yuan",

"jq" : "lib/jquery.min"

　　}

});

require(['circle','fang','jq'],function(y,f,$){

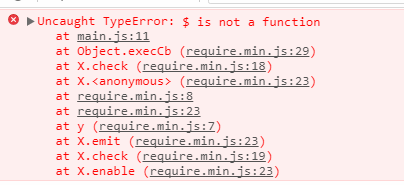
alert(y.mianji(10));

alert(f.mianji(10,20));

$("#box").css("background-color" , "red");

});

结果报错：



为什么会报错，就是因为jQuery是不符合AMD规范的，天生是没有对define做适配。

此时解决办法就是加shim：

require.config({

　　paths: {

　　　　"circle" : "yuan",

"jq" : "lib/jquery.min"

　　},

**shim: {**

**'jq': {**

**exports : '$'**

**}**

**}**

});

require(['circle','fang','jq'],function(y,f,$){

alert(y.mianji(10));

alert(f.mianji(10,20));

$("#box").animate({"font-size":400},1000);

});

### 1.2.5 引入有依赖的依赖的库

比如我们引入jquery-ui，很明显jquery-ui是jquery的插件。此时我们说jquery-ui依赖jquery。

此时我们要在shim中定义这层关系：

require.config({

　　paths: {

　　　　"circle" : "yuan",

"jq" : "lib/jquery.min",

"jqui" : "lib/jquery-ui.min"

　　},

shim: {

'jq': {

exports : '$'

　　　　},

**'jqui': {**

**deps: ['jq']**

**}**

}

});

require(['circle','fang','jq','jqui'],function(y,f,$,jqui){

alert(y.mianji(10));

alert(f.mianji(10,20));

$("#box").animate({"font-size":400},1000);

$("#box").draggable();

});

### 1.2.6 为什么叫AMD规范？

首先我们要知道依赖注入的时候，依赖的模块们（比如下面的4个依赖）彼此之间没有加载顺序之分的。只有一个亘古不变的真理：**当他们都加载完毕之后，执行回调函数**。

require(['circle','fang','jq','jqui'],function(y,f,$,jqui){

alert(y.mianji(10));

alert(f.mianji(10,20));

$("#box").animate({"font-size":400},1000);

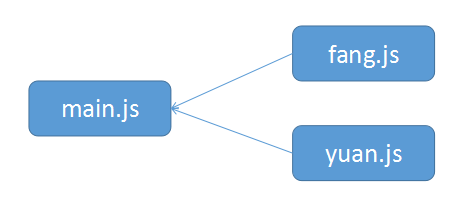
$("#box").draggable();

});

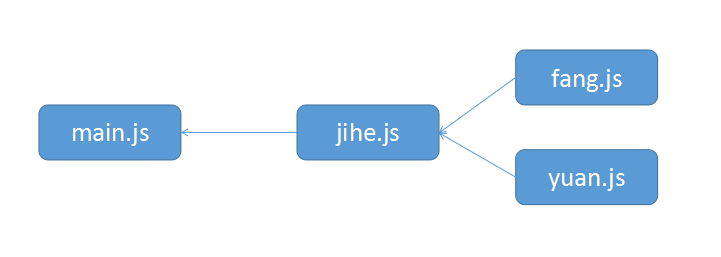
**alert("我最先执行");**

在回调函数外面的语句是最先执行的。

我们现在的demo是这样的：



变为：



|  |
| --- |
| ┣ js  ┃ ┣ lib  ┃ ┃ ┣ require.js  ┃ ┣ main.js  ┃ ┣ yuan.js  ┃ ┣ fang.js  ┃ ┣ jihe.js  ┣ index.html |

jihe.js:

define(**["fang","circle"]**,function(**fang,circle**){

**return {**

**fang,**

**circle**

**}**

});

任何普通模块也可以有依赖注入。

主文件的局部：

……

require([**'jihe'**,'jq','jqui'],function(**jihe**,$,jqui){

alert(jihe.circle.mianji(10));

alert(jihe.fang.mianji(10,20));

$("#box").animate({"font-size":400},1000);

$("#box").draggable();

});

我们的主文件只依赖了jihe（jquery先不考虑），但是由于jihe依赖了fang和yuan，所以fang和yuan都加载了。

总结，到底为什么叫做AMD规范：

1. AMD规范使用依赖注入的形式，所有的依赖项是同时加载的，没有回来的先后之分，都回来了才执行回调函数。也就是说，回调函数是AMD规范的特征。
2. require()里面的语句是异步语句，把依赖项都加载完毕之后才执行回调函数，此时就是“异步”的由来。
3. AMD规范中，如果两个模块之间有“插件”的关系，彼此依赖，可以用shim中定义这个关系。

AMD的现状：很惨。

* 只有require.js对AMD进行了实现。
* Angular1使用了AMD规范，而Angular2放弃了AMD规范，转入了CMD阵营，Angular4、5都是CMD的。

看一下Angular1中的AMD特性：

<script type="text/javascript">

var myapp = angular.module("myapp" , []);

myapp.controller("MainCtrl",[**"shuxue"**,function(**shuxue**){

this.a = 100;

var self = this;

this.add = function(){

self.a++;

}

this.minus = function(){

self.a--;

}

this.pf = function(){

self.a = shuxue.pingfang(self.a);

}

}]);

myapp.factory("**shuxue**",[function(){

function pingfang(a){

return a \* a;

}

**return {**

**pingfang**

**}**

}]);

</script>

## 1.3 CMD规范

Common Module Definition，通用模块定义。

它的实现：common.js、sea.js（淘宝玉伯）、node.js。

我们看sea.js的实现。

|  |
| --- |
| ┣ js  ┃ ┣ lib  ┃ ┃ ┣ sea.js  ┃ ┣ main.js  ┃ ┣ yuan.js  ┃ ┣ fang.js  ┃ ┣ People.js  ┣ index.html |

main.js:

**define(function(require,exports,module){**

var yuan = **require**("./yuan.js");

var fang = **require**("./fang.js");

alert(yuan.mianji(10))

alert(fang.mianji(10,20))

**});**

yuan.js:

**define(function(require,exports,module){**

function mianji(r){

return r \* r \* 3.14;

}

**exports**.mianji = mianji;

**});**

fang.js:

**define(function(require,exports,module){**

function mianji(a,b){

return a \* b;

}

**exports**.mianji = mianji;

**});**

People.js:

**define(function(require,exports,module){**

function People(){

}

**module**.exports = People;

**});**

index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

**<script type="text/javascript" src="js/lib/sea.min.js"></script>**

<script type="text/javascript">

**seajs.use("./js/main.js");**

</script>

</body>

</html>

CMD总结：

1. nodejs是遵循CMD规范的，可以裸奔CMD规范。所以大家已经会了CMD规范了；
2. 所有的模块都要用define(function(require,exports,module){})包裹，称为“标准壳”；
3. 暴露有两种途径exports.\*\* = \*\* ; module.exports = \*\* ;区别见《nodejs第1天笔记》；
4. 引用的时候用require()引用，require谁就执行谁，会死等这个文件加载完毕，没有回调函数。
5. CMD规范中没有node\_modules这个神奇的文件夹的概念，是nodejs自己添加的特性。

最后说一嘴：AMD、CMD规范和业务一点关系没有，就是纯粹的文件组织的形式。网页DOM开发是不会用AMD、CMD规范的，现在AMD、CMD学习的意义就是服务于Angular、React、Vue的。

# Webpack

## 2.1 安装

**webpack 是一个模块打包器。它的主要目标是将 JavaScript 文件打包在一起，打包后的文件用于在浏览器中使用，但它也能够胜任转换(transform)、打包(bundle)或包裹(package)任何资源(resource or asset)。**

webpack是nodejs的工作流工具，首先进行安装：

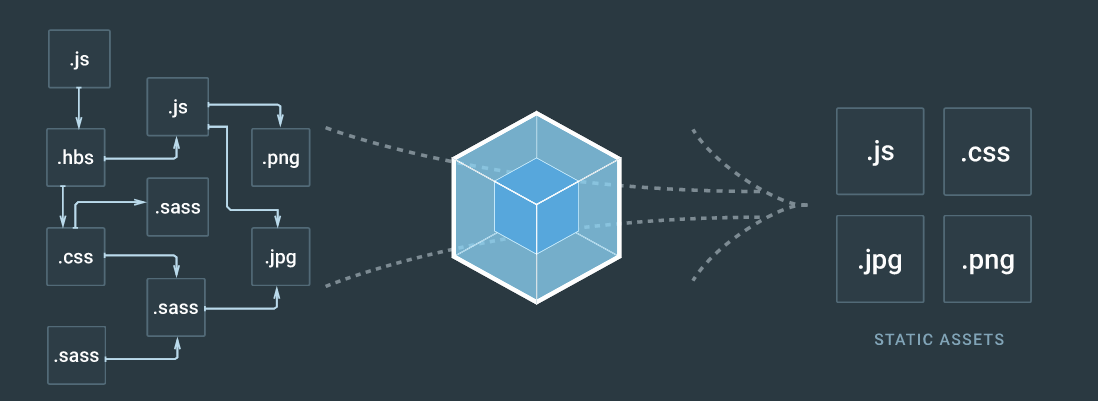
cnpm install -g webpack

安装完毕之后，查看版本号：





官网：https://doc.webpack-china.org/



## 2.2 基本使用

|  |
| --- |
| ┣ js  ┃ ┣ main.js  ┃ ┣ yuan.js  ┣ index.html |

我们的项目很简单，没有任何的第三方库，没有sea.js、没有require.js。

程序也很简单，所有的js文件都在裸奔CMD规范，没有任何的标准壳：

main.js：

var yuan = require("./yuan.js");

alert(yuan.mianji(20));

这里面出现了require这个词，浏览器不认识，但是打包之后的all.js中会定义require，会自动给他加上标准壳。

yuan.js:

function mianji(r){

return 3.14 \* r \* r;

}

exports.mianji = mianji;

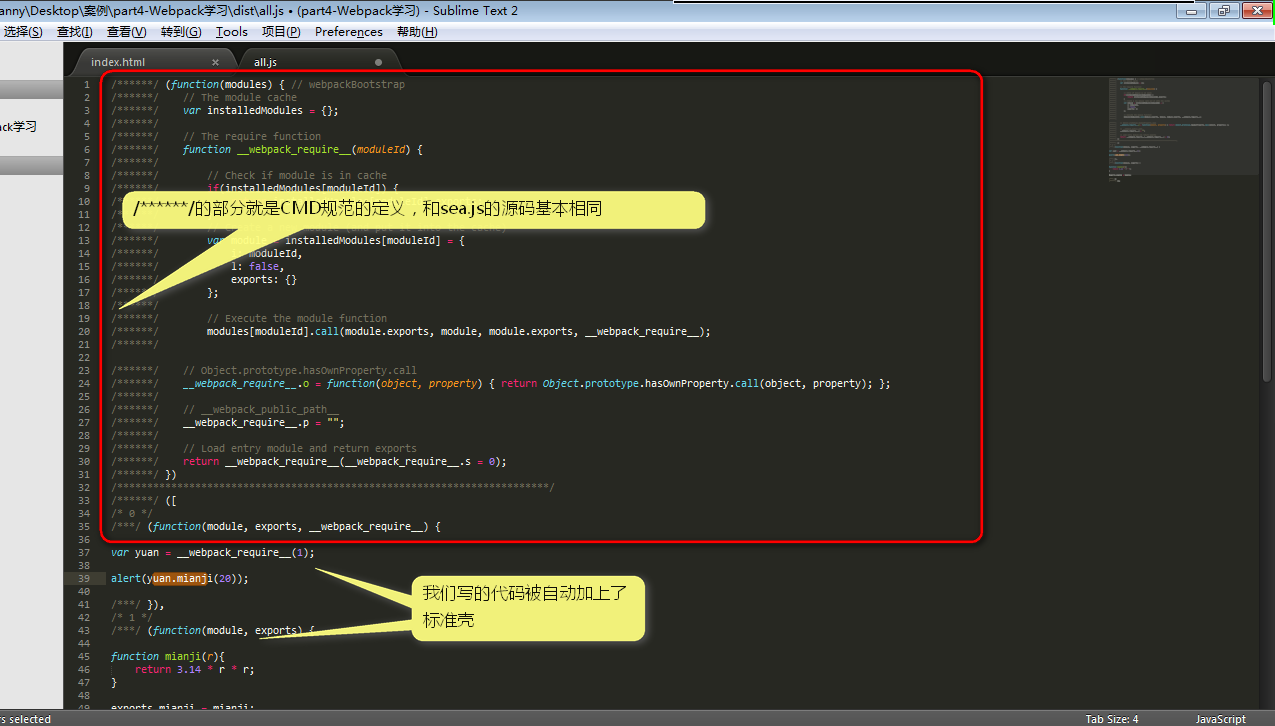
然后我们在CMD窗口中执行：

webpack js/main.js dist/all.js

此时神奇的事情发生了，两个js文件：main.js、yuan.js被打包成为了一个js文件。



我们看一下生成的dist/all.js文件：



为什么说webapck是智能打包、智能合并：

* **webpack不是无脑合并js文件！和我们今后要学习的Grunt、Gulp不同；**
* **webpack是按图索骥的，只有require链中出现的js文件，才会被打包到all.js文件中！**
* **如果多个文件都require了某文件，这个文件只会在all.js中出现一次。**

## 2.3 webpack.config.js文件

在项目的根目录，**创建一个webpack.config.js文件，这个文件可以指导webpack的工作**。

这个文件要求背诵：

var path = require("path");

**//进行配置的东西，实际上是本文件的暴露项，要写module.exports**

module.exports = {

**//配置入口文件**

entry : "./js/main.js" ,

**//配置产出文件**

output : {

**//产出文件文件夹**

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

**//产出文件的文件名**

filename : "all.js"

},

**//实时监测文件更新，一旦文件更新了，就重新合并打包一份**

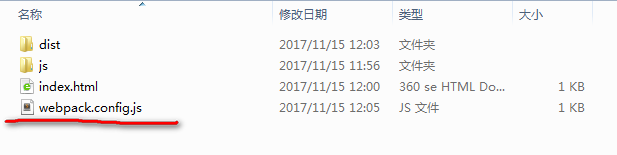
watch : true

}

写完这个文件之后，我们就能在项目的根目录打开CMD窗口，输入简单的命令：

webpack

webpack就知道自己找入口文件了，自己合并到产出文件了，自己监听文件的变化了。



## 2.4 loader

我们写了一些ES6语句，**希望webpack在打包的同时，捎带脚用babel进行一下翻译**。

webpack和各个厂商的关系非常好，各个厂商给webpack提供了自己的loader，这些loader都是用于在webpack合并文件的时候，“捎带脚”做的事情。

安装babel-loader和babel-core（babel核心）：

cnpm install --save-dev babel-loader babel-core

还要安装babel的一个preset：

cnpm install --save-dev babel-preset-es2015

改变webpack.config.js文件：

var path = require("path");

//进行配置的东西，实际上是本文件的暴露项，要写module.exports

module.exports = {

//配置入口文件

entry : "./js/main.js" ,

//配置产出文件

output : {

//产出文件文件夹

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

//产出文件的文件名

filename : "all.js"

},

**module : {**

**rules : [**

**{**

**"test" : /\.js$/ ,**

**"use" : [**

**{**

**"loader" : "babel-loader" ,**

**"options" : {**

**"presets" : ["es2015"]**

**}**

**}**

**]**

**}**

**]**

**},**

//实时监测文件更新，一旦文件更新了，就重新合并打包一份

watch : true

}

配置方法是看的API文档：

https://doc.webpack-china.org/concepts/loaders/#-loader

所以昨天学习的.babelrc文件没有用了！

当我们还想使用更高级语法，比如强制解构，此时要装babel插件。

安装依赖：

cnpm install --save-dev babel-plugin-transform-object-rest-spread

改变webpack.config.js文件：

var path = require("path");

//进行配置的东西，实际上是本文件的暴露项，要写module.exports

module.exports = {

//配置入口文件

entry : "./js/main.js" ,

//配置产出文件

output : {

//产出文件文件夹

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

//产出文件的文件名

filename : "all.js"

},

module : {

rules : [

{

"test" : /\.js$/ ,

"use" : [

{

"loader" : "babel-loader" ,

"options" : {

"presets" : ["es2015"] ,

**"plugins" : ["transform-object-rest-spread"]**

}

}

]

}

]

},

//实时监测文件更新，一旦文件更新了，就重新合并打包一份

watch : true

}

# ES6中的CMD语法

上午学习的CMD语法是commonjs在2005年左右提出的语法，被nodejs全盘采纳。

但是，当官的，不可能允许民办的人提出标准，一定要改改。

简单的对应关系：

require变为了import

exports变为了export

module变为了default

浏览器厂商对此很抗议，到今天2017年11月15日，没有任何浏览器支持上述语法。

但是webpack支持，webpack在没有babel-loader的存在下，天生能够识别import的导入语法，从而进行智能的文件合并。

● 基本的暴露和引入

yuan.js:

**export const** mianji = function(r){

return 3.14 \* r \* r;

}

**export const** zhouchang = function(r){

return 6.28 \* r;

}

main.js:

**import {zhouchang , mianji} from "./yuan.js";**

alert(zhouchang(10));

alert(mianji(10));

暴露的名字和接收的名字必须一样。暴露的是mianji接收必须是mianji。

● 命名空间

如果两个包中都有mianji、zhouchang函数，此时用import {zhouchang,mianji}就冲突了。

所以用import \* as的语法制定命名空间。

**import \* as yuan from "./yuan.js";**

**import \* as fang from "./fang.js";**

alert(**yuan.**zhouchang(10));

alert(**yuan.**mianji(10));

alert(**fang.**zhouchang(10,20));

alert(**fang.**mianji(10,20));

● 默认暴露

我们知道，当一个js文件是一个类的时候，此时我们只需要暴露这一个类，nodejs用module.exports=\*\*暴露，这里改了，变为**export default 类名**。

class People{

constructor(name,age,sex){

this.name = name;

this.age = age;

this.sex = sex;

}

haha(){

console.log(`我是${this.name}，哈哈`);

}

}

**export default People;**

神奇的是，ES6中的默认暴露可以和普通暴露共存：

class People{

constructor(name,age,sex){

this.name = name;

this.age = age;

this.sex = sex;

}

haha(){

console.log(`我是${this.name}，哈哈`);

}

}

//随便一个函数

**export const lajihanshu** = function(){

console.log("哈哈");

}

**export default People;**

导入的时候，普通暴露加大括号，默认暴露不加大括号，中间用逗号隔开。

import \* as yuan from "./yuan.js";

import \* as fang from "./fang.js";

**import People , {lajihanshu} from "./People.js";**

console.log(yuan.zhouchang(10));

console.log(yuan.mianji(10));

console.log(fang.zhouchang(10,20));

console.log(fang.mianji(10,20));

var xiaoming = new People("小明",12,"男");

xiaoming.haha();

lajihanshu();

# React的配置

今天是一个前导课程，至于什么是MVVM、单页面应用（SPA）等等下节课介绍。

今天只配置。

**React需要webpack + babel-loader的环境**。

为什么要用babel-loader呢？因为React中采用了一个新的语法糖叫做jsx，需要babel-loader的babel-preset-react进行翻译。我们编程的时候全面采用ES6，所以也需要babel-preset-es2015进行翻译，现在叫做babel-preset-env。

所以React的配置有三个层次：

1. 进行webpack的配置
2. 进行babel-loader的配置
3. 写React的HelloWorld

## 4.1 第一步，配置webpack

创建新的项目文件夹，创建package.json身份证：

npm init

创建webpack.config.js文件，书写最简单的入口、出口、监听变化：

var path = require("path");

module.exports = {

entry : "./app/main.js" ,

output : {

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

filename : "bundle.js"

},

watch : true

}

下面要写一些程序，验证webpack配置正确。

创建app文件夹，创建main.js和aaaa.js（瞎写一个）：

main.js:

import {afda} from "./aaaa.js";

afda();

aaaa.js:

export const afda = function(){

alert("哈哈");

}

创建一个html文件，引入bundle.js文件：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<script type="text/javascript" src="dist/bundle.js"></script>

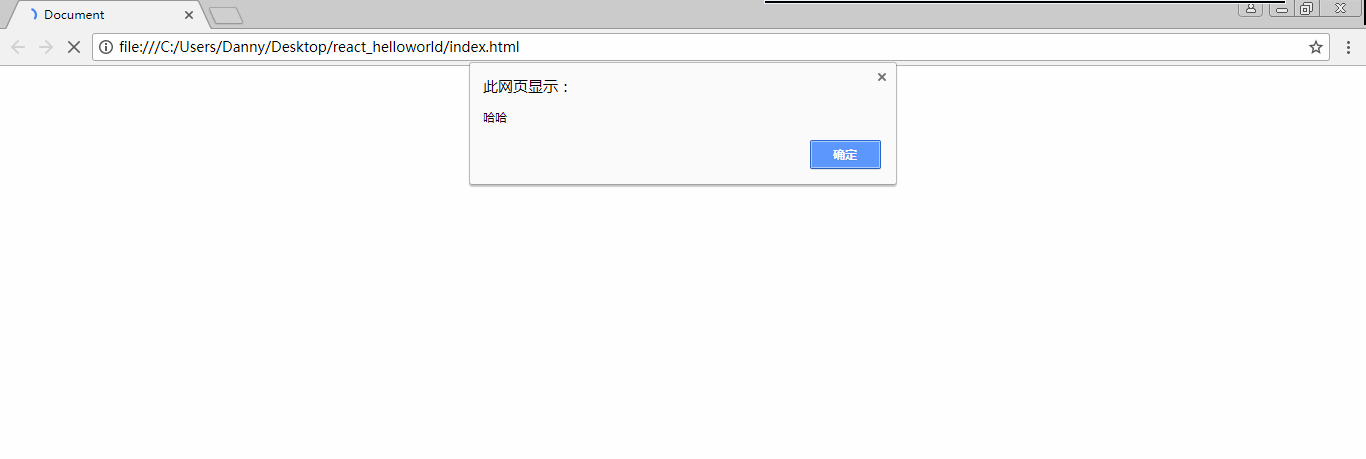
</body>

</html>

运行

webpack

自己看页面：



至此，第一步完成，文件夹结构：

|  |
| --- |
| ┣ js  ┃ ┣ main.js  ┃ ┣ aaaa.js  ┣ index.html  ┣ package.json  ┣ webpack.config.js |

## 4.2 第二步，配置babel-loader

需要注意的是，babel-loader需要安装两个preset：env、react。

安装babel-loader和babel-core：

cnpm intsall --save-dev babel-loader babel-core

安装两个preset：

cnpm intsall --save-dev babel-preset-env babel-preset-react

然后改变webpack.config.js文件，注意黄色部分很关键：

var path = require("path");

module.exports = {

entry : "./app/main.js" ,

output : {

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

filename : "bundle.js"

},

module : {

rules : [

{

test : /\.js$/ ,

use : [

{

loader : "babel-loader" ,

options : {

presets : ["env","react"]

}

}

],

include: [

path.resolve(\_\_dirname, "app")

],

exclude: [

path.resolve(\_\_dirname, "node\_modules")

]

}

]

},

watch : true

}

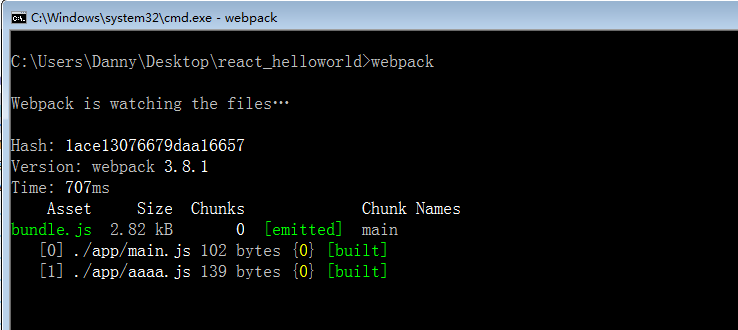
更改我们之前创建的main.js文件，写一些解构的语句，测试babel是否正常工作。

import {afda} from "./aaaa.js";

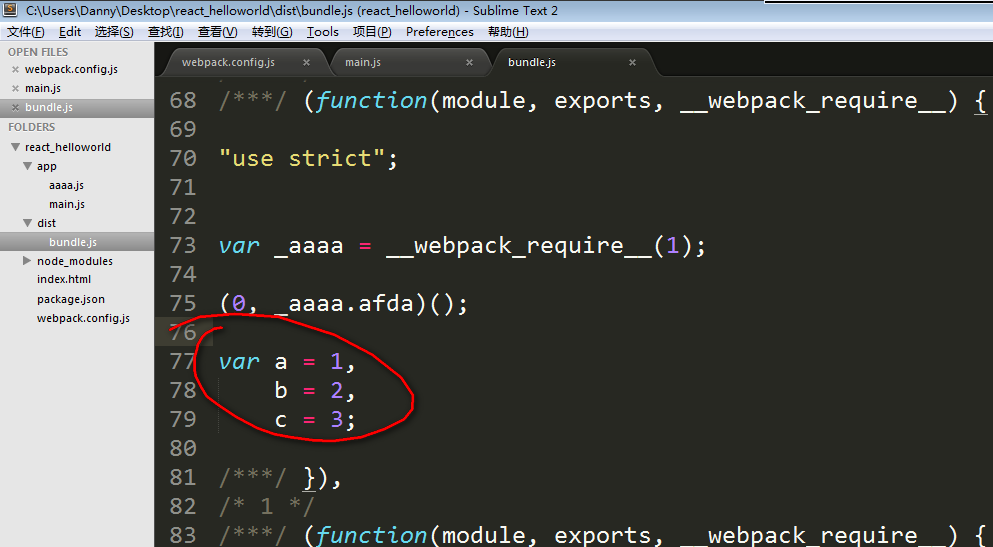
afda();

var [a,b,c] = [1,2,3];

webpack不报错。



查看bundle.js文件，发现自动解构的语句变为了单独写的var，此时说明正确：



## 4.3 第三步，写React程序

安装依赖，两个：react和react-dom

cnpm install --save react react-dom

写程序：

import React from "react";

import ReactDOM from "react-dom";

ReactDOM.render(

<h1>你好，我是React，很高兴遇见你！</h1>

,

document.getElementById("app")

);

红色部分是jsx语法，js和html可以非常简单的融合，不需要双引号了，也不需要连字符什么的了。

改变index.html，增加一个div：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

**<div id="app"></div>**

<script type="text/javascript" src="dist/bundle.js"></script>

</body>

</html>

然后webpack构建出来：

webpack

看页面：

