[Webpack3.X版 成神之路 全网首发 (共24集)](http://jspang.com/2017/09/16/webpack3-2/)

2017-09-16 分类：[其它教程](http://jspang.com/category/learn/other/) / [视频教程](http://jspang.com/category/learn/) 阅读(41823) 评论(195)

**目录**[[隐藏](http://jspang.com/2017/09/16/webpack3-2/)]

* [第00节：文章和问答说明](http://jspang.com/2017/09/16/webpack3-2/#00)
* [第01节：认识WebPack的作用](http://jspang.com/2017/09/16/webpack3-2/#01webpack)
* [第02节：让你快速上手一个Demo](http://jspang.com/2017/09/16/webpack3-2/#02demo)
* [第03节：配置文件：入口和出口](http://jspang.com/2017/09/16/webpack3-2/#03)
* [第04节：配置文件： 服务和热更新](http://jspang.com/2017/09/16/webpack3-2/#04)
* [第05节：模块：CSS文件打包](http://jspang.com/2017/09/16/webpack3-2/#05css)
* [第06节：插件配置：配置JS压缩](http://jspang.com/2017/09/16/webpack3-2/#06js)
* [第07节：插件配置：HTML文件的发布](http://jspang.com/2017/09/16/webpack3-2/#07html)
* [第08节：图片迈坑：CSS中的图片处理](http://jspang.com/2017/09/16/webpack3-2/#08css)
* [第09节：图片迈坑：CSS分离与图片路径处理](http://jspang.com/2017/09/16/webpack3-2/#09css)
* [第10节：图片迈坑：处理HTML中的图片](http://jspang.com/2017/09/16/webpack3-2/#10html)
* [第11节：CSS进阶：Less文件的打包和分离](http://jspang.com/2017/09/16/webpack3-2/#11cssless)
* [第12节：CSS进阶：SASS文件的打包和分离](http://jspang.com/2017/09/16/webpack3-2/#12csssass)
* [第13节：CSS进阶：自动处理CSS3属性前缀](http://jspang.com/2017/09/16/webpack3-2/#13csscss3)
* [第14节：CSS进阶：消除未使用的CSS](http://jspang.com/2017/09/16/webpack3-2/#14csscss)
* [第15节：给webpack增加babel支持](http://jspang.com/2017/09/16/webpack3-2/#15webpackbabel)
* [第16节：打包后如何调试](http://jspang.com/2017/09/16/webpack3-2/#16)
* [第17节：实战技巧：开发和生产并行设置](http://jspang.com/2017/09/16/webpack3-2/#17)
* [第18节：实战技巧：webpack模块化配置](http://jspang.com/2017/09/16/webpack3-2/#18webpack)
* [第19节：实战技巧：优雅打包第三方类库](http://jspang.com/2017/09/16/webpack3-2/#19)
* [第20节：实战技巧：watch的正确使用方法](http://jspang.com/2017/09/16/webpack3-2/#20watch)
* [第21节：实战技巧：webpack优化黑技能](http://jspang.com/2017/09/16/webpack3-2/#21webpack)
* [第22节：实战技巧：静态资源集中输出](http://jspang.com/2017/09/16/webpack3-2/#22)
* [第23节：实战技巧：Json配置文件使用](http://jspang.com/2017/09/16/webpack3-2/#23json)
* [第24节：剧终-webpack自学技巧传授](http://jspang.com/2017/09/16/webpack3-2/#24-webpack)

**第00节：文章和问答说明**

**本文首发：jspang.com，其他网站禁止转载。**

本文讲解的是Webpack3.0+的知识，努力做到全网最好的webpack3.0教程。文章通过**一个半月的时间**创作完成，共**二十四小节**，将近**四万字**，并根据文字教程录制了对应的视频教程（48元24节，350分钟视频讲解，保证你可以学会），当然你不需要看视频教程也能很快的学会webpack最新版的知识。

如果你已经购买了视频教程，可以在视频的**最后一节**看到如何加入讨论微信群，技术胖会每**周上线解**答你学习中的困难。

如果你没有购买视频教程，但学习中遇到问题，希望有一帮webpack高手进行交流，你可以打赏19.8元钱进webpack微信交流群（暂时没有开通打赏功能），然后就会看到加入微信群的方法。

群里已经有450人左右，而且我也会经常再群里交流，为大家解决问题。马上打赏进群吧。

[19.8元,马上进群](javascript:void(0);)

付款后，在这里显示进群方法。

**第01节：认识WebPack的作用**

如果您已经在前端行业中，WebPack在业界的流行程度自然必备多说，成为了**前端小白**升级为前端**工程师**的必备技能。如果你对webpack还不够熟悉，那你在前端前进的脚步会受到阻碍，但是你幸运的搜索到了这篇文章。（但是文章可能不会讲解如何从1.0、2.0版本升级3.0版本的知识，而是直接讲解3.0的知识，所以你可能需要有一个空杯心态来学习）

在学习过程中，我希望你能每节看完文章后，都可以自己亲手做一做，如果你不作就不会出现错误，不出现错误，你就没办法增长经验，那你就什么都学不会。相信我，**慢慢来，比较快**。

**前端为什么需要WebPack？**

现在的前端网页功能丰富，特别是SPA（single page web application 单页应用）技术流行后，JavaScript的复杂度增加和需要一大堆依赖包，还需要解决SCSS，Less……新增样式的扩展写法的编译工作。所以现代化的前端已经完全依赖于WebPack的辅助了。

现在最流行的三个前端框架，可以说和webpack已经紧密相连，框架官方都推出了和自身框架依赖的webpack构建工具。

* React.js+WebPack
* Vue.js+WebPack
* AngluarJS+WebPack

从此可以看出，无论你前端走那条线，你都要有很强的Webpack知识，才能祝你成为这个框架领域的大牛。

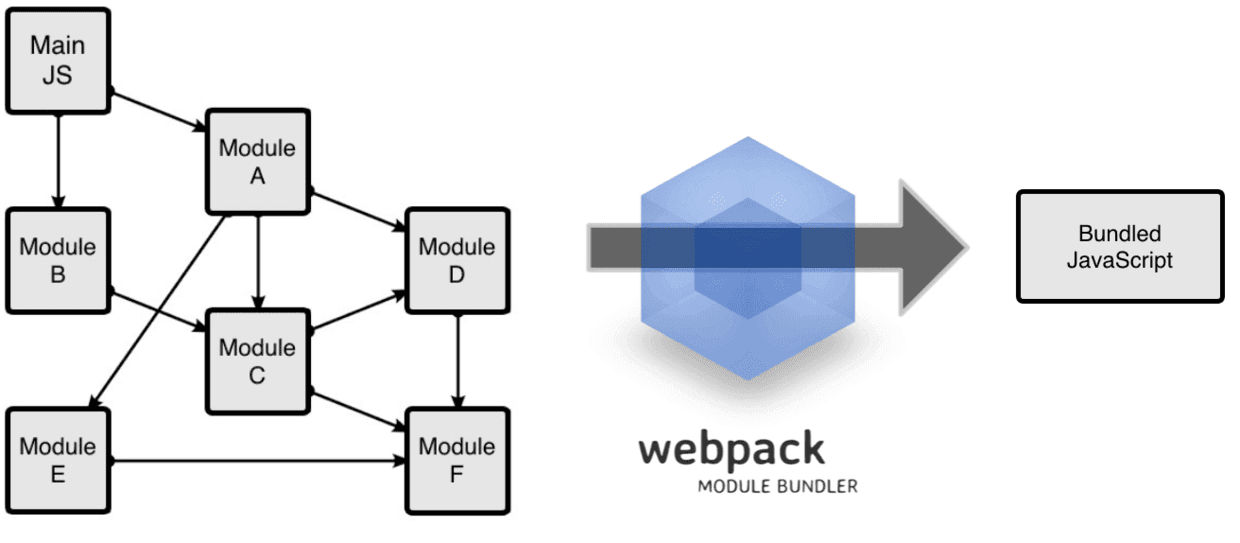
**什么是WebPack？**

WebPack可以看做是**模块打包机**：它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Sass，TypeScript等），并将其转换和打包为合适的格式供浏览器使用。在3.0出现后，Webpack还肩负起了优化项目的责任。

这段话有三个重点：

* 打包：可以把多个Javascript文件打包成一个文件，减少服务器压力和下载带宽。
* 转换：把拓展语言转换成为普通的JavaScript，让浏览器顺利运行。
* 优化：前端变的越来越复杂后，性能也会遇到问题，而WebPack也开始肩负起了优化和提升性能的责任。

我们可以从下图再次了解一下WebPack的作用:



**安装WebPack**

看了这么久，一定着急动手作一作了。要使用WebPack就要先进行安装，就和你要使用微信，必须在手机上下载微信的APP一样，但是不同的是WebPack的安装，采用的是命令行npm形式的安装。

这里我以windows系统为例，给大家做截图示范。苹果安装稍有不同，不过大同小异（如果有问题，可以直接进群交流，进群方法看文章开头）。

**具体安装方法：**

用win+R打开运行对话框，输入cmd进入命令行模式。然后找到你想开始项目的地方，输入下方代码：



|  |  |
| --- | --- |
| 1  2 | mkdir webpack\_demo  cd webpack\_demo |

第一句是建立一个文件夹，第二句是进入这个文件夹。这个文件夹就是我们的项目文件目录了，文件夹建立好后，可以通过下面命令安装WebPack。

需要注意的是,你在执行下一步时必须安装node，可以通过 node -v来查看node安装情况和版本，如果没有安装，要先安装node才可以继续进行。



|  |  |
| --- | --- |
| 1  2 | //全局安装  npm install -g webpack |

如果你这时**安装失败**了（出现了报错信息），一般有三种可能：

* 检查你node的版本号，如果版本号过低，升级为最新版本。
* 网络问题，可以考虑使用cnpm来安装（这个是淘宝实时更新的镜像）,具体可以登录cnpm的官方网站学习<http://npm.taobao.org/>。
* 权限问题，在Liux、Mac安装是需要权限，如果你是Windows系统，主要要使用以管理员方式安装。

**注意：**全局安装是可以的，但是webpack官方是不推荐的。这会将您项目中的 webpack 锁定到指定版本，并且在使用不同的 webpack 版本的项目中，可能会导致构建失败。

**对项目目录进行安装**

全局安装完成后，我们还要进行一个项目目录的安装。在用npm安装前，我们先要进行一下初始化，初始化的主要目的是生成package.json文件（这是一个标准的npm说明文件，里面蕴含了丰富的信息，包括当前项目的依赖模块，自定义的脚本任务等等，如果你对此文件还不了解，可以看看node 的相关知识）。

在命令行输入：



|  |  |
| --- | --- |
| 1 | npm init |

输入完成后，npm终端会问你关于项目的名称，描述……一堆内容，如果你不考虑发布到npm上，这些内容都不重要，而且我们后期还可以用文本的形式修改这些内容。现在我们只要一路回车就完成了初始化。这时用dir命令已经可以看到生成的package.json文件了。

输入下面命令进行项目目录的安装：



|  |  |
| --- | --- |
| 1 | npm install --save-dev webpack |

这里的参数–save是要保存到package.json中，dev是在开发时使用这个包，而生产环境中不使用。

**开发环境and生产环境：**

* 开发环境：在开发时需要的环境，这里指在开发时需要依赖的包。
* 生产环境：程序开发完成，开始运行后的环境，这里指要使项目运行，所需要的依赖包。

**查看webpack版本**

你安装好后，会想知道你现在webpack版本，在工作中交流时，也会经常问到你，你的打包版本是什么？这时候我们可以用下面的命令进行查看。



|  |  |
| --- | --- |
| 1 | webpack -v |

可以看到我现在的版本的3.6.0版本，这是目前（2017/9/16）最新的版本了。出现了版本号，也说明你的webpack安装成功了。

看到这里，我们第一节的内容就完成了，我建议你停一下，把webpack安装到你的电脑上，再进行向下观看，如果在安装时遇到什么问题，可以在下方的留言区给我留言。

**第02节：让你快速上手一个Demo**

上节课已经安装好了Webpack到电脑上，这节课就开始一个简单的Demo，让你快速上手和熟悉Webpack的基本用法，学习并作完这节课内容你就可以和朋友小吹一下说：我也会Webpack。

**建立基本项目结构**

首先进入上节课我们建立的webpack\_demo目录（每个人建立的位置不同，可能建立在了D盘或者E盘）。进入后在根目录建立两个文件夹，分别是**src文件夹**和**dist文件夹**：

* src文件夹：用来存放我们编写的javascript代码，可以简单的理解为用JavaScript编写的模块。
* dist文件夹：用来存放供浏览器读取的文件，这个是webpack打包成的文件。

你可以理解成src是源码文件，dist是我们编译打包好的文件；一个用于**开发环境**，一个用于**生产环境**。

**编写程序文件：**

文件夹建立好后，我们在dist文件下手动建立一个**index.html**文件，并写入下面的代码。

**/dist/index.html**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | *<!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>jspang webpack</title>  </head>  <body>      <div id="title"></div>      <script src="./bundle.js"></script>  </body>  </html> |

这里引入了一个JavaScript的bundle.js文件。这个文件现在还没有，这是用webpack执行打包命令后生产的文件。我们的index.html写好后，接下来在src文件夹下建立entery.js的文件，用于编写我们的JavaScript代码，也是我们的入口文件。

**src/entery.js**



|  |  |
| --- | --- |
| 1 | document.getElementById('title').innerHTML='Hello Webpack'; |

这个文件的代码很简单，就是在**<div id=”title”></div>**标签里写入Hello Webpack这句话。

**第一次Webpack打包**

Webpack其实是可以在终端（命令行）中使用的，基本使用方法如下：



|  |  |
| --- | --- |
| 1 | webpack {entry file} {destination for bundled file} |

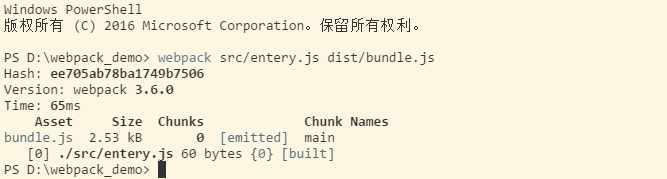
* **{entry file}:入**口文件的路径，本文中就是src/entery.js的路径；
* **{destination for bundled file}:**填写打包后存放的路径。
* **注意：**在命令行中是不需要写{ }的。

**在我写的例子中，终端执行命令如下：**



|  |  |
| --- | --- |
| 1 | webpack src/entry.js dist/bundle.js |

执行的结果如下图：



命令执行成功后，会在dist目录下出现**bundle.js文件**，这时我们就可以在浏览器中预览结果了，网页中显示出了Hello Webpack的信息。

**总结：**

从这个Demo中你会了解到webpack的基本用法和使用过程，并会了命令行打包的方法。在这节文章的最后，我还是要强调，你一定要把本节内容在自己的电脑上敲一遍，这样你才能深入了解。

**第03节：配置文件：入口和出口**

首先要说明的是：学习这节课前，务必作完上节课的代码，否则你会学的一脸懵逼。

上节课通过一个小Demo我们对Webpack有了初步了解，但是上节课的终端打包方案，在实际开发中并不使用，而是使用Webpack的配置文件的方式进行设置。这节课我们就学一下配置文件的大体结构和入口出口文件的配置。

**配置文件webpack.config.js**

webpack.config.js就是Webpack的配置文件，这个文件需要自己在项目根目录下手动建立。建立好后我们对其进行配置，先看下面的代码（webpack.config.js的基本结构），这是一个没有内容的标准webpack配置模版。

**webpack.config.js**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | module.exports={      //入口文件的配置项      entry:{},      //出口文件的配置项      output:{},      //模块：例如解读CSS,图片如何转换，压缩      module:{},      //插件，用于生产模版和各项功能      plugins:[],      //配置webpack开发服务功能      devServer:{}  } |

* entry：配置入口文件的地址，可以是单一入口，也可以是多入口。
* output：配置出口文件的地址，在webpack2.X版本后，支持多出口配置。
* module：配置模块，主要是解析CSS和图片转换压缩等功能。
* plugins：配置插件，根据你的需要配置不同功能的插件。
* devServer：配置开发服务功能，后期我们会详细讲解。

**entry选项（入口配置）**

这个选项就是配置我们要压缩的文件一般是JavaScript文件（当然也可以是CSS…..）。按照上节课的代码(如果你上节课的代码还没作，那你可以返回去重新作一下)，这里要填写的是src目录下的entery.js文件。

**wepback.config.js中的entry选项**



|  |  |
| --- | --- |
| 1  2  3  4  5 | //入口文件的配置项  entry:{      //里面的entery是可以随便写的      entry:'./src/entry.js'  }, |

**output选项（出口配置）**

出口配置是用来告诉webpack最后打包文件的地址和文件名称的。按照上节课的操作，应该打包到dist目录下。在编写出口文件时，我们需要用到一点Node的知识，如果你还不会Node也没有 关系，就简单的两句代码，你记住就可以了（在视频中我还有详细的讲解）。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //出口文件的配置项  output:{      //打包的路径文职      path:path.resolve(\_\_dirname,'dist'),      //打包的文件名称      filename:'bundle.js'    }, |

如果你只这样写，是会报错的：找不到path这个东西。所以我们要在webpack.config.js的头部引入path，代码如下：



|  |  |
| --- | --- |
| 1 | const path = require('path'); |

这里我们使用了**const**，这是ES6的语法，如果你对ES6还不熟悉，也可以看技术胖ES6的课程哦（<http://jspang.com/2017/06/03/es6/>）。

其实**path.resolve(\_\_dirname,’dist’)**就是获取了项目的**绝对路径**。如果你还是不理解，我会在视频中给你进行演示。

**filename:**是打包后的文件名称，这里我们起名为bundle.js。

现在webpack.config.js的代码：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | const path = require('path');  module.exports={      //入口文件的配置项      entry:{          entry:'./src/entry.js'      },      //出口文件的配置项      output:{          //输出的路径，用了Node语法          path:path.resolve(\_\_dirname,'dist'),          //输出的文件名称          filename:'bundle.js'      },      //模块：例如解读CSS,图片如何转换，压缩      module:{},      //插件，用于生产模版和各项功能      plugins:[],      //配置webpack开发服务功能      devServer:{}  } |

这个代码写完后，可以在终端中直接输入**webpack**就会进行打包。

在实际开发中我们都是通过配置文件进行打包的，所以必须要掌握好。

**多入口、多出口配置**

Webpack在版本1的时候很难设置多出口文件，但是在2版本开始就变的很方便了。直接看多入口和多出口的文件配置，然后可以和单一出口对比一下，你会发现这种设置非常简单（只需改动两点配置就可以）。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | const path = require('path');  module.exports={      //入口文件的配置项      entry:{          entry:'./src/entry.js',          //这里我们又引入了一个入口文件          entry2:'./src/entry2.js'      },      //出口文件的配置项      output:{          //输出的路径，用了Node语法          path:path.resolve(\_\_dirname,'dist'),          //输出的文件名称          filename:'[name].js'      },      //模块：例如解读CSS,图片如何转换，压缩      module:{},      //插件，用于生产模版和各项功能      plugins:[],      //配置webpack开发服务功能      devServer:{}  } |

可以看到代码的第7和14行进行了增加和修改，在入口文件配置中，增加了一个entry2.js的入口文件（这个文件你需要自己手动建立），这时候要打包的就有了两个入口文件。在代码14行我们把原来的bundle.js修改成了[name].js。

**[name]**的意思是根据入口文件的名称，打包成相同的名称，有几个入口文件，就可以打包出几个文件。

**总结：**

这节课的内容是需要仔细消化的，不求你记住，但是要练习，因为你无论配置任何项目的Webpack都要作这些操作。你可以把本文当作一个字典，在需要的时候进行查询。

**第04节：配置文件： 服务和热更新**

作为一个前端工程师，最大的编程需求之一就是所见即所得的工具，也就是常说的热更新。现在一般有点规模的公司都为前端工程师准备了双屏显示器，其目的就是一个屏幕编写代码，一个屏幕实时显示页面效果。这节课就学习用webpack3.6版本实现热更新效果。

从这节课开始视频需要购买才可以观看。教程一共30集左右，学完后完全可以达到独立构建中大型项目水平，从此webpack不在是你职业发展的瓶颈了。

**购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**设置webpack-dev-server**

要执行webpack-dev-server是要先用npm install webpack-dev-server –save-dev 来进行下载的。下载好后，需要配置一下devServer。最简单的devServer**配置项只有四个**。先看一下代码，然后我再作解释。

**/webpack.config.js**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | devServer:{          //设置基本目录结构          contentBase:path.resolve(\_\_dirname,'dist'),          //服务器的IP地址，可以使用IP也可以使用localhost          host:'localhost',          //服务端压缩是否开启          compress:true,          //配置服务端口号          port:1717      } |

* contentBase:配置服务器基本运行路径，用于找到程序打包地址。
* host：服务运行地址，建议使用本机IP，这里为了讲解方便，所以用localhost。
* compress：服务器端压缩选型，一般设置为开启，如果你对服务器压缩感兴趣，可以自行学习。
* port：服务运行端口，建议不使用80，很容易被占用，这里使用了1717.

**注意：**这里需要使用npm 来进行安装webpack-dev-server了， 命令如下：



|  |  |
| --- | --- |
| 1 | npm install webpack-dev-server --save-dev |

这是本地安装，所以使用了–save-dev。

配置好后，你可以试着在终端中输入webpack-dev-server,如果可以执行成功，但是往往提示下面的错误（或者是无法找到内部或外部命令）。



出现下面的错误不用慌张，我们只要在package.json里配置一下scripts选项就可以执行了。

**/package.json**



|  |  |
| --- | --- |
| 1  2  3 | "scripts": {      "server":"webpack-dev-server"  }, |

配置好保存后，在终端里输入**npm  run  server**  打开服务器。然后在浏览器地址栏输入http://localhost:1717就可以看到结果了。

**支持热更新**

在npm run server  启动后，它是有一种监控机制的（也叫watch）。它可以监控到我们修改源码，并立即在浏览器里给我们更新。

**注意：**这里只是我们的webpack3.6版本支持，在3.5版本时要支持热更新还需要一些其他的操作。因为已经有了成熟的3.6版本，我就不再介绍低版本的操作方法。还有一种情况。如果你都设置好了，但是不进行热更新，可能是你系统的问题，在Linux和Ma上支持良好，在Windows上有时会出现问题。

如果你在操作时，在Windows上出现问题了，请在文章下方给我留言。

**第05节：模块：CSS文件打包**

Webpack在生产环境中有一个重要的作用就是减少http的请求数，就是把多个文件打包到一个js里，这样请求数就可以减少好多。这节课我们就学习一个重要的知识，把我们的CSS文件打包。在学习CSS打包之前，需要先对webpack.config.js里的Loaders配置项进行了解。

**购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**Loaders**

Loaders是Webpack最重要的功能之一，他也是Webpack如此盛行的原因。通过使用不同的Loader，Webpack可以的脚本和工具，从而对不同的文件格式进行特定处理。

简单的举几个Loaders使用例子：

* 可以把SASS文件的写法转换成CSS，而不在使用其他转换工具。
* 可以把ES6或者ES7的代码，转换成大多浏览器兼容的JS代码。
* 可以把React中的JSX转换成JavaScript代码。

注意：所有的Loaders都需要在npm中单独进行安装，并在webpack.config.js里进行配置。下面我们对Loaders的配置型简单梳理一下。

* test：用于匹配处理文件的扩展名的表达式，这个选项是必须进行配置的；
* use：loader名称，就是你要使用模块的名称，这个选项也必须进行配置，否则报错；
* include/exclude:手动添加必须处理的文件（文件夹）或屏蔽不需要处理的文件（文件夹）（可选）；
* query：为loaders提供额外的设置选项（可选）。

明白了Loader是什么后，就开始这节课的正题，如何打包CSS文件。

**打包CSS文件：**

**建立index.css文件**

要打包CSS你必须先要有个CSS文件，在/src目录下，我们建立一个css文件夹，在文件夹里建立index.css文件。代码内容如下。

**./src/css/index.css**



|  |  |
| --- | --- |
| 1  2  3  4 | body{      background-color: red;      color: white;  } |

CSS文件建立好后，需要引入到入口文件中，才可以打包到，这里我们引入到entry.js中。

/src/entery.js中在首行加入代码：



|  |  |
| --- | --- |
| 1 | import css from './css/index.css'; |

CSS和引入做好后，我们就需要使用loader来解析CSS文件了，这里我们需要两个解析用的loader，分别是style-loader和css-loader。

**style-loader:**

它是用来处理css文件中的url()等，npm中的网址：<https://www.npmjs.com/package/style-loader>

用npm install 进行项目安装：



|  |  |
| --- | --- |
| 1 | npm install style-loader --save-dev |

**css-loader：**

它是用来将css插入到页面的style标签。npm中的网址：<https://www.npmjs.com/package/css-loader>

用npm install 进行项目安装：



|  |  |
| --- | --- |
| 1 | npm install --save-dev css-loader |

两个loader都下载安装好后，我们就可以配置我们loaders了。

**loaders配置：**

修改webpack.config.js中module属性中的配置代码如下：

**webpack.config.js**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | module:{          rules: [              {                test: /\.css$/,                use: [ 'style-loader', 'css-loader' ]              }            ]      }, |

这个文件的详细讲解，我们在上面已经提及，如果你还是无法理解可以观看视频。

**总结：**

敲黑板，这节课的内容非常重要，上面的配置过程最好作两遍以上，完全了解后，再进行下节课的学习。loader的使用也决定着你webpack水平的高低。所以一定要重视和练习。

**第06节：插件配置：配置JS压缩**

通过五节课的学习，我相信小伙伴已经对Webpack有所入门。这节课让我们初步了解插件（plugins[ ]）的用法。在学习新知识之前，我先回答一个小伙伴提的问题，他的问题就是：“我看到别人写的CSS打包配置文件和你写的不一样，是不是有其他的写法？”

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**loader的三种写法：**

上节课学习了如何把CSS文件进行打包到JS当中去，有小伙伴就提问，我看到别人的CSS打包的写法和你的写法不太一样，是不是他们写错了，loader还有几种写法，这里我们就看两种另外的写法。

**第一种写法：直接用use。**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | module:{          rules:[              {                  test:/\.css$/,                  use:['style-loader','css-loader']              }          ]      }, |

**第二种写法：把use换成loader。**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | module:{          rules:[              {                  test:/\.css$/,                  loader:['style-loader','css-loader']              }          ]      }, |

**第三种写法：用use+loader的写法：**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | module:{          rules:[              {                  test:/\.css$/,                  use: [                      {                          loader: "style-loader"                      }, {                          loader: "css-loader"                      }                  ]              }          ]      }, |

由此看出，webpack的扩展和灵活性是非常强的，你习惯于那种写法都可以。最重要的是，你看见别人项目的其他写法也不要慌张，自己去试一试，有可能就可以Get到新知识。

**压缩JS代码：**

现在你写的JS代码，在上线之前，都是需要进行压缩的，在没有webpack和gulp这些工具前，你可能需要找一个压缩软件或者在线进行压缩，在Webpack中可以很轻松的实现JS代码的压缩，它是通过插件的方式实现的，这里我们就先来引入一个**uglifyjs-webpack-plugin**(JS压缩插件，简称uglify)。

**注意：**虽然uglifyjs是插件，但是webpack版本里默认已经集成，不需要再次安装。

**引入：**

我们需要在webpack.config.js中引入uglifyjs-webpack-glugin插件



|  |  |
| --- | --- |
| 1 | const uglify = require('uglifyjs-webpack-plugin'); |

引入后在plugins配置里new一个 uglify对象就可以了，代码如下。



|  |  |
| --- | --- |
| 1  2  3 | plugins:[          new uglify()      ], |

这时候在终端中使用webpack进行打包，你会发现JS代码已经被压缩了。如果你用的VSCode的话，可以按Alt+Z让他文件自动换行，查看效果。

贴出通过6节课学习，现在**webpack.config.js文件中**的所有代码，这样大家可以对照学习。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | const path=require('path');  const uglify = require('uglifyjs-webpack-plugin');  module.exports={      entry:{          entry:'./src/entry.js',          entry2:'./src/entry2.js'      },      output:{          path:path.resolve(\_\_dirname,'dist'),          filename:'[name].js'      },      module:{          rules:[              {                  test:/\.css$/,                  use: ["style-loader", "css-loader"]              }          ]      },      plugins:[          new uglify()      ],      devServer:{         contentBase:path.resolve(\_\_dirname,'dist'),         host:'192.168.0.104',         compress:true,         port:1717      }  } |

**第07节：插件配置：HTML文件的发布**

有经验的小伙伴其实一眼就可以看出，现在我们的项目结构是有问题的，我们把index.html直接放到了dist文件夹下，这肯定是不正确的，应该放到我们src目录下。但是前期我们为了循序渐进的学习，所以把index.html放到了dist目录下。这节课我们就学习如何把html文件打包到我们的生产路径下。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**devServer和JS压缩的冲突**

上节课学习了JS压缩，在视频中我使用了webpack进行打包，而没有使用npm run server 进行预览，也就是说没有启用devServer里的配置。那有些小伙伴在学习完视频后，在终端中输入了npm run server进行了预览，发现终端中报错了。

要弄明白这个问题，我们先要弄清楚什么是**开发环境**，什么是**生产环境**。开发环境中是基本不会对js进行压缩的，在开发预览时我们需要明确的报错行数和错误信息，所以完全没有必要压缩JavasScript代码。而生产环境中才会压缩JS代码，用于加快程序的工作效率。devServer用于开发环境，而压缩JS用于生产环境，**在开发环境中作生产环境的事情所以Webpack设置了冲突报错**。

在实际开发中，webpack配置文件是分开的，开发环境一个文件，生产环境一个文件。所以在讲课之前我并没有发现这个问题，感谢小伙伴提出的问题。如果你在学习过程中有任何疑问，欢迎在**文章下方留言**。

**打包HTML文件**

我们先把dist中的html文件剪切到src目录中，并**去掉我们的JS引入代码**（webpack会自动为我们引入JS），因为这才是我们真实工作的目录文件结构。

然后我们配置webpack.config.js文件，先引入我们的html-webpack-plugin插件。



|  |  |
| --- | --- |
| 1 | const htmlPlugin= require('html-webpack-plugin'); |

引入后使用npm进行安装包。



|  |  |
| --- | --- |
| 1 | npm install --save-dev html-webpack-plugin |

最后在webpack.config.js里的plugins里进行插件配置，配置代码如下。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | new htmlPlugin({              minify:{                  removeAttributeQuotes:true              },              hash:true,              template:'./src/index.html'            }) |

* minify：是对html文件进行压缩，removeAttrubuteQuotes是却掉属性的双引号。
* hash：为了开发中js有缓存效果，所以加入hash，这样可以有效避免缓存JS。
* template：是要打包的html模版路径和文件名称。

上边的都配置完成后，我们就可以在终端中使用webpack，进行打包。你会看到index.html文件已经被打包到我们的dist目录下了，并且自动为我们引入了路口的JS文件。

**总结：**

html文件的打包可以有效的区分开发目录和生产目录，在webpack的配置中也要搞清楚哪些配置用于生产环境，哪些配置用于开发环境，避免两种环境的配置冲突。

**第08节：图片迈坑：CSS中的图片处理**

在学习Webapck过程中你可能遇到的第一个坑就是CSS中的图片处理。很多webpack新手都在图片的坑中无法自拔（有的小伙伴在开发环境中是可以找到图片的，但是一打包后就找不到图片了，有的小伙伴是不知道如何正确引入html或者css中的图片，导致程序出错），我们将用三节课时间搞彻底走出webpack图片的坑。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**图片写入CSS**

你可以先在网上找一个图片，我这里就自恋的使用了我的头像，如果你需要下载，也可以下载（当然你可以完全自己找一个自己喜欢的）。

找到图片后在src目录下新建一个images文件夹，把图片放入images文件夹。

在**index.html**文件中增加一个放置div的标签（需要注意的是这里修改的是src下的index.html文件，不是dist下的，这点新手很容易弄混，要格外注意），代码如下。



|  |  |
| --- | --- |
| 1 | <div id="tupian"></div> |

编写**css**文件，把你用的图片作为背景显示。



|  |  |
| --- | --- |
| 1  2  3  4  5 | #tupian{     background-image: url(../images/manhua.png);     width:466px;     height:453px;  } |

编写完成后，我们可以试着用webpack去打包一下。你会发现终端中是报错的，具体错误可以看下图。



**file-loader、url-loader**

上面的错误是由于缺少loader的解析，对loader其实我们并不陌生，因为前边已经学习了CSS打包的loader。我们先安装两个解析图片用的loader。

安装file-loader和url-loader



|  |  |
| --- | --- |
| 1 | npm install --save-dev file-loader url-loader |

安装好后我们需要对两个loader进行基本的了解，学习尽量做到知其然知其所以然。

**file-loader：**解决引用路径的问题，拿background样式用url引入背景图来说，我们都知道，webpack最终会将各个模块打包成一个文件，因此我们样式中的url路径是相对入口html页面的，而不是相对于原始css文件所在的路径的。这就会导致图片引入失败。这个问题是用file-loader解决的，file-loader可以解析项目中的url引入（不仅限于css），根据我们的配置，将图片拷贝到相应的路径，再根据我们的配置，修改打包后文件引用路径，使之指向正确的文件。

**url-loader：**如果图片较多，会发很多http请求，会降低页面性能。这个问题可以通过url-loader解决。url-loader会将引入的图片编码，生成dataURl。相当于把图片数据翻译成一串字符。再把这串字符打包到文件中，最终只需要引入这个文件就能访问图片了。当然，如果图片较大，编码会消耗性能。因此url-loader提供了一个limit参数，小于limit字节的文件会被转为DataURl，大于limit的还会使用file-loader进行copy。

配置url-loader

我们安装好后，就可以使用这个loader了，记得在loader使用时不需要用require引入，在plugins才需要使用require引入。

**webpack.config.js文件**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | //模块：例如解读CSS,图片如何转换，压缩      module:{          rules: [              {                test: /\.css$/,                use: [ 'style-loader', 'css-loader' ]              },{                 test:/\.(png|jpg|gif)/ ,                 use:[{                     loader:'url-loader',                     options:{                         limit:500000                     }                 }]              }            ]      }, |

* test:/\.(png|jpg|gif)/是匹配图片文件后缀名称。
* use：是指定使用的loader和loader的配置参数。
* limit：是把小于500000B的文件打成Base64的格式，写入JS。

写好后就可以使用webpack进行打包了，这回你会发现打包很顺利的完成了。具体的Base64的格式，你可以查看视频中的样子。

**为什么只使用了url-loader**

有的小伙伴会发现我们并没有在webpack.config.js中使用file-loader，但是依然打包成功了。我们需要了解file-loader和url-loader的关系。url-loader和file-loader是什么关系呢？简答地说，url-loader封装了file-loader。url-loader不依赖于file-loader，即使用url-loader时，只需要安装url-loader即可，不需要安装file-loader，因为url-loader内置了file-loader。通过上面的介绍，我们可以看到，url-loader工作分两种情况：

1.文件大小**小于**limit参数，url-loader将会把文件转为DataURL（Base64格式）；

2.文件大小**大于**limit，url-loader会调用file-loader进行处理，参数也会直接传给file-loader。

也就是说，其实我们只安装一个url-loader就可以了。但是为了以后的操作方便，我们这里就顺便安装上file-loader。

**第09节：图片迈坑：CSS分离与图片路径处理**

通过上节课的学习已经能把小图片打包成Base64格式，也对webpack对图片的打包有个基本了解。这节课主要学习两个知识：第一个是把CSS从JavasScript代码中分离出来，第二个是如何处理分离出来后CSS中的图片路径不对问题。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**CSS分离:extract-text-webpack-plugin**

有些简单的交互页面中，你的JavasScript页面代码会非常少，而大部分代码都在CSS中，这时候项目组长会要求把CSS单独提取出来，方便以后更改。遇到这个需求你不要惊慌，已经有大神为我们准备好了对象的插件（plugin）。

**extract-text-webpack-plugin**

这个插件就可以完美的解决我们提取CSS的需求，但是webpack官方其实并不建议这样作，他们认为CSS就应该打包到JavasScript当中以减少http的请求数。但现实中的需求往往不是我们前端能控制的，有些需求是我们不能控制的，分离CSS就是这样一个既合理由不合理的需求。

**安装：**录制课程时的版本是3.0.0版本，直接使用npm install 就可以安装。



|  |  |
| --- | --- |
| 1 | npm install --save-dev extract-text-webpack-plugin |

**引入：**安装完成后，需要先用require引入。



|  |  |
| --- | --- |
| 1 | const extractTextPlugin = require("extract-text-webpack-plugin"); |

**设置Plugins：**引入成功后需要在plugins属性中进行配置。这里只要new一下这个对象就可以了。



|  |  |
| --- | --- |
| 1 | new extractTextPlugin("/css/index.css") |

这里的/css/index.css是分离后的路径位置。这部配置完成后，**包装代码：**还要修改原来我们的style-loader和css-loader。

修改代码如下。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | module:{          rules: [              {                test: /\.css$/,                use: extractTextPlugin.extract({                  fallback: "style-loader",                  use: "css-loader"                })              },{                 test:/\.(png|jpg|gif)/ ,                 use:[{                     loader:'url-loader',                     options:{                         limit:500000                     }                 }]              }            ]      }, |

作完上边这四部后，就可以使用webpack进行打包了。

**图片路径问题：**

利用extract-text-webpack-plugin插件很轻松的就把CSS文件分离了出来，但是CSS路径并不正确，很多小伙伴就在这里搞个几天还是没有头绪，网上也给出了很多的解决方案，我觉的最好的解决方案是使用publicPath解决，我也一直在用。

**publicPath：**是在webpack.config.js文件的output选项中，主要作用就是处理静态文件路径的。

在处理前，我们在webpack.config.js 上方声明一个对象，叫website。



|  |  |
| --- | --- |
| 1  2  3 | var website ={      publicPath:"http://192.168.1.108:1717/"  } |

注意，这里的IP和端口，是你本机的ip或者是你devServer配置的IP和端口。

然后在output选项中引用这个对象的publicPath属性。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //出口文件的配置项      output:{          //输出的路径，用了Node语法          path:path.resolve(\_\_dirname,'dist'),          //输出的文件名称          filename:'[name].js',          publicPath:website.publicPath      }, |

配置完成后，你再使用webpack命令进行打包，你会发现原来的相对路径改为了绝对路径，这样来讲速度更快。

**总结：**这节课我们实现了CSS的分离，并在分离后处理了图片路径不对的问题。处理路径的方法一定要充分理解，因为这在工作中经常用到。

**第10节：图片迈坑：处理HTML中的图片**

在webpack中是不喜欢你使用标签<img>来引入图片的，但是我们作前端的人特别热衷于这种写法，国人也为此开发了一个：**html-withimg-loader**。他可以很好的处理我们在html 中引入图片的问题。因为是国人开发的，文档都是中文，所以学习起来还是比较简单的。所以在学习新课之前我们先解决两个小伙伴的问题。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**只有项目安装webpack，如何打包？**

有的小伙伴在学习视频时，并没有全局安装webpack，而是使用了项目安装。首先我要说的是，这种做法是webpack推崇的，webpack并不鼓励全局安装webpack。但是小伙伴看我视频中直接在终端用webpack进行打包项目，他使用时会出现不是内部命令或者外部命令。

这时候需要配置package.json里的scripts选项，我们以前的课程已经学习了配置 webpack-dev-server命令，在这个命令下面我们再加一个**build**命令进行打包项目使用。



|  |  |
| --- | --- |
| 1  2  3  4 | "scripts": {      "server": "webpack-dev-server --open",      "build":"webpack"    }, |

配置完成后，可以在控制台输入**npm run build** 进行打包。

**如何把图片放到指定的文件夹下**

前边两节课程，打包后的图片并没有放到images文件夹下，要放到images文件夹下，其实只需要配置我们的url-loader选项就可以了。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | module:{          rules: [              {                test: /\.css$/,                use: extractTextPlugin.extract({                  fallback: "style-loader",                  use: "css-loader"                })              },{                 test:/\.(png|jpg|gif)/ ,                 use:[{                     loader:'url-loader',                     options:{                         limit:5000,                         outputPath:'images/',                     }                 }]              }            ]      }, |

这回你再执行打包就可以把图片打包到images文件夹里了。

**html-withimg-loader**

html-withimg-loader就是我们今天的重点了，这个插件并不是很火，也是我个人喜欢的一个小loader。解决的问题就是在hmtl文件中引入<img>标签的问题。

安装：



|  |  |
| --- | --- |
| 1 | npm install html-withimg-loader --save |

**配置loader**

webpack.config.js



|  |  |
| --- | --- |
| 1  2  3  4 | {      test: /\.(htm|html)$/i,       use:[ 'html-withimg-loader']  } |

然后在终端中可以进行打包了。你会发现images被很好的打包了。并且路径也完全正确。

**总结：**我们通过三节课的时间讲了webpack图片中的坑，这些坑在我初学webpack初期给我带来了不少的麻烦，我也算是倾囊相教了，希望小伙伴们有所收获。在你工作中遇到图片的问题，也可以返回文章里进行对比查找问题。

**第11节：CSS进阶：Less文件的打包和分离**

第05节中已经讲过CSS文件的打包，后来又讲了CSS分离。这节我们讲解一下Less文件如何打包和分离。Less 是一门 CSS 预处理语言，它扩展了 CSS 语言，增加了变量、Mixin、函数等特性，使 CSS 更易维护和扩展。也就是说Less给我们枯燥单一的样式文件加入了编程机制，这让我们这些前端程序员很受用，所以在工作中大部分程序员都使用了Leess开发。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**打包Less文件**

**安装:**

要使用Less，我们要首先安装Less的服务，当然也是用npm来进行安装。



|  |  |
| --- | --- |
| 1 | npm install --save-dev less |

还需要安装Less-loader用来打包使用。



|  |  |
| --- | --- |
| 1 | npm install --save-dev less-loader |

**写loader配置：**

安装好后，需要在webpack.config.js里编写loader配置，当然要想正确解析成CSS，还是需要style-loader和css-loader的帮助，但是这两个loader前边已经讲过了，所以在这里就不重复了，如果你还对这两个loader不熟悉，那自行回去补前边的第五节吧。

webpack.config.js



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | {      test: /\.less$/,      use: [{             loader: "style-loader" // creates style nodes from JS strings          }, {              loader: "css-loader" // translates CSS into CommonJS          , {              loader: "less-loader" // compiles Less to CSS          }]  } |

**编写一个less文件**

现在webpack的配置好了，我们还需要编写一个less文件，这里明文为black.less.里边只做一件是就是把一个层的背景设置成黑色。

black.less



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @base :#000;  #gogo{      width:300px;      height:300px;      background-color:@base;  } |

这里#gogo是层的ID名称。@base是我们设置的变量名称。

**引入到我们entery.js文件中**



|  |  |
| --- | --- |
| 1 | import less from './css/black.less'; |

这样我们就可以把less文件进行打包了。我们可以使用webpack命令打包试一试。

**把Lees文件分离。**

我们之前讲了extract-text-webpack-plugin这个插件，想把Less文件分离出来的方法跟这个几乎一样，之前的我们在第09节中讲过，这里我们就只讲less的loader配置方法。（此处建议收看视频）



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | {              test: /\.less$/,              use: extractTextPlugin.extract({                  use: [{                      loader: "css-loader"                  }, {                      loader: "less-loader"                  }],                  // use style-loader in development                  fallback: "style-loader"              })  } |

配置好后，你会发现less被分离到了index.css文件里。

总结：Less是非常好的CSS扩展，但是Less得转换稍显麻烦，好的是webpack为我们提供了简单轻松的转换方法。希望小伙伴可以学好这一课，在你们的工作中都开始使用Less编写你们css代码。

**第12节：CSS进阶：SASS文件的打包和分离**

上节课学习了Less的打包和分离，群里使用SASS的小伙伴马上就不干了，要求讲一下SASS的配置，其实你会了Less得配置，SASS的配置可以很轻松的学会，为了公平公正，那我们就用一节课的时间学一下SASS的配置的。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**安装SASS打包的loader**

这里需要 在项目目录下用npm安装两个包。**node-sass和sass-loader**

node-sass：因为sass-loader依赖于node-sass，所以需要先安装node-sass



|  |  |
| --- | --- |
| 1 | npm install --save-dev node-sass |

sass-loader:



|  |  |
| --- | --- |
| 1 | npm install --save-dev sass-loader |

**注意：**在用npm安装时，这个loader很容易安装失败，最好使用cnpm来进行安装。如果你安装一直报错，最好是把node\_modules文件夹删除后，再重新安装。

**编写loader配置**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | {                  test: /\.scss$/,                  use: [{                      loader: "style-loader" // creates style nodes from JS strings                  }, {                      loader: "css-loader" // translates CSS into CommonJS                  }, {                      loader: "sass-loader" // compiles Sass to CSS                  }]  } |

需要注意的是loader的加载要有先后顺序。

**Sass文件的编写**

写好loader配置后，就可以愉快的编写sass文件拉，但是不要忘记把sass文件引入到entery.js中。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $nav-color: #FFF;  #nav {    $width: 100%;    width: $width;    height:30px;    background-color: $nav-color;  } |

都完成后，你就可以启动我们**npm run server** 来查看效果了。

**把SASS文件分离。**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | {              test: /\.scss$/,              use: extractTextPlugin.extract({                  use: [{                      loader: "css-loader"                  }, {                      loader: "sass-loader"                  }],                  // use style-loader in development                  fallback: "style-loader"              })  } |

这节课算是专门为sass使用者录制的吧，其实整体过程和less的使用差不多，希望你能在工作中开始使用sass，并写出漂亮的css代码。

**第13节：CSS进阶：自动处理CSS3属性前缀**

CSS3已经成了前端的**必会技能**，但是你一定为那些属性需要加前缀，那些属性不需要加前缀而头疼。以前我在课程中讲过一个can i use的网站，可以查询这些，但是每次都查实在是编码效率太低了。这节课我们就学习一下如何通过**postcss-loader**给css3属性自动添加前缀。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**什么是属性前缀**

我们先来看一下代码：



|  |  |
| --- | --- |
| 1  2 | -webkit-transform: rotate(45deg);          transform: rotate(45deg); |

为了浏览器的兼容性，有时候我们必须加入-webkit,-ms,-o,-moz这些前缀。目的就是让我们写的页面在每个浏览器中都可以顺利运行。

**PostCSS**

PostCSS是一个CSS的处理平台，它可以帮助你的CSS实现更多的功能，但是今天我们就通过其中的一个加前缀的功能，初步了解一下PostCSS。

**安装**

需要安装两个包postcss-loader 和autoprefixer（自动添加前缀的插件）



|  |  |
| --- | --- |
| 1 | npm install --save-dev postcss-loader autoprefixer |

**postcss.config.js**

postCSS推荐在项目根目录（和webpack.config.js同级），建立一个postcss.config.js文件。

**postcss.config.js**



|  |  |
| --- | --- |
| 1  2  3  4  5 | module.exports = {      plugins: [          require('autoprefixer')      ]  } |

这就是对postCSS一个简单的配置，引入了autoprefixer插件。让postCSS拥有添加**前缀**的能力，它会根据 can i use 来增加相应的css3属性前缀。

**编写loader**

对postcss.config.js配置完成后，我们还需要编写我们的loader配置。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | {        test: /\.css$/,        use: [              {                loader: "style-loader"              }, {                loader: "css-loader",                options: {                   modules: true                }              }, {                loader: "postcss-loader"              }        ]  } |

**提取CSS**

配置提取CSS的loader配置.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {      test: /\.css$/,      use: extractTextPlugin.extract({          fallback: 'style-loader',          use: [              { loader: 'css-loader', options: { importLoaders: 1 } },              'postcss-loader'          ]      })    } |

总结:postcss还有很多功能，我希望小伙伴学会自学。这里给出**postcss-loader**的github地址：<https://github.com/postcss/postcss-loader>

**第14节：CSS进阶：消除未使用的CSS**

像Bootstrap这样的框架往往会带有很多CSS。在项目中通常我们只使用它的一小部分。就算我们自己写CSS，随着项目的进展，CSS也会越来越多，有时候需求更改，带来了DOM结构的更改，这时候我们可能无暇关注CSS样式，造成很多CSS的冗余。这节课就学习用webpack消除未使用的CSS。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**PurifyCSS**

使用PurifyCSS可以大大减少CSS冗余，比如我们经常使用的BootStrap(140KB)就可以减少到只有35KB大小。这在实际开发当中是非常有用的。

**安装PurifyCSS-webpack**

从名字你就可以看出这是一个插件，而不是loader。所以这个需要安装还需要引入。 PurifyCSS-webpack要以来于purify-css这个包，所以这两个都需要安装。



|  |  |
| --- | --- |
| 1 | npm i -D purifycss-webpack purify-css |

这里的-D代表的是–save-dev ,只是一个简写。

**引入glob**

因为我们需要同步检查html模板，所以我们需要引入node的glob对象使用。在webpack.config.js文件头部引入glob。



|  |  |
| --- | --- |
| 1 | const glob = require('glob'); |

引入purifycss-webpack

同样在webpack.config.js文件头部引入purifycss-webpack



|  |  |
| --- | --- |
| 1 | const PurifyCSSPlugin = require("purifycss-webpack"); |

配置plugins

引入完成后我们需要在webpack.config.js里配置plugins。代码如下，重点看标黄部分。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | plugins:[      //new uglify()      new htmlPlugin({          minify:{              removeAttrubuteQuotes:true          },          hash:true,          template:'./src/index.html'        }),      new extractTextPlugin("css/index.css"),      new PurifyCSSPlugin({          // Give paths to parse for rules. These should be absolute!          paths: glob.sync(path.join(\_\_dirname, 'src/\*.html')),          })    ] |

这里配置了一个paths，主要是需找html模板，purifycss根据这个配置会遍历你的文件，查找哪些css被使用了。

**注意**：使用这个插件必须配合extract-text-webpack-plugin这个插件，这个插件在前边的课程已经讲解过了。如果你还不会请自学一下。

配置好上边的代码，我们可以故意在css文件里写一些用不到的属性，然后用webpack打包，你会发现没用的CSS已经自动给你删除掉了。在工作中记得一定要配置这个plugins，因为这决定你代码的质量，非常有用。

**第15节：给webpack增加babel支持**

在前端开发中都开始使用ES6的语法了，虽然说webpack3增加了一些ES6的转换支持，但是实际效果不是很好，也可能是本人技术有限，没发挥出真正的功能。所以我在开发中还是喜欢添加Babel-loader的，我也查看了一些别人的webpack配置也都增加了babel-loader，所以这节课我们学习一下如何增加Babel支持。（此节文章部分内容引用了zhangwang大神的文章内容）

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**Babel是什么？**

Babel其实是一个编译JavaScript的平台，它的强大之处表现在可以通过便宜帮你达到以下目的：

* 使用下一代的javaScript代码(ES6,ES7….)，即使这些标准目前并未被当前的浏览器完全支持。
* 使用基于JavaScript进行了扩展的语言，比如React的JSX。

**Babel的安装与配置**

Babel其实是几个模块化的包，其核心功能位于称为babel-core的npm包中，webpack可以把其不同的包整合在一起使用，对于每一个你需要的功能或拓展，你都需要安装单独的包（用得最多的是解析ES6的babel-preset-es2015包和解析JSX的babel-preset-react包）。

我们先一次性安装这些依赖包



|  |  |
| --- | --- |
| 1 | cnpm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react |

在webpack中配置Babel的方法如下：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | {      test:/\.(jsx|js)$/,      use:{          loader:'babel-loader',          options:{              presets:[                  "es2015","react"              ]          }      },      exclude:/node\_modules/  } |

现在你已经可以用webapck转换ES6的语法兼容各个浏览器了，我们可以修改一下entry.js的代码如下：



|  |  |
| --- | --- |
| 1  2  3  4  5 | import css from './css/index.css';  {      let jspangString = 'Hello Webpack'      document.getElementById('title').innerHTML=jspangString;  } |

上面的代码使用了ES6的let声明方法。如果你不使用Babel来进行转换，你会发现打包出来的js代码没有作兼容处理，使用了Babel转换的代码是进行处理过的。

**.babelrc配置**

虽然Babel可以直接在webpack.config.js中进行配置，但是考虑到babel具有非常多的配置选项，如果卸载webapck.config.js中会非常的雍长不可阅读，所以我们经常把配置卸载.babelrc文件里。

在项目根目录新建.babelrc文件，并把配置写到文件里。

**.babelrc**



|  |  |
| --- | --- |
| 1  2  3 | {      "presets":["react","es2015"]  } |

**.webpack.config.js里的loader配置**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | {      test:/\.(jsx|js)$/,      use:{          loader:'babel-loader',      },      exclude:/node\_modules/  } |

**ENV：**

现在网络上已经不流行babel-preset-es2015，现在官方推荐使用的是babel-preset-env,那我们为了紧跟潮流，我们在讲一下env的配置方法。

首先需要下载：



|  |  |
| --- | --- |
| 1 | npm install --save-devv babel-preset-env |

然后修改.babelrc里的配置文件。其实只要把之前的**es2015**换成**env**就可以了。



|  |  |
| --- | --- |
| 1  2  3 | {      "presets":["react","env"]  } |

总结：对于在React中Babel的使用，如何解析JSX，我会在后边的课程作详细了解，大家不要着急。在实际工作中还是要安装Babel的，这样能更好的兼容每种浏览器，而把Babel的配置文件分解出来是最好的选择。

**第16节：打包后如何调试**

作为一个程序员每天的大部分工作就是调试自己写的程序，那我们使用了webpack后，所以代码都打包到了一起，给调试带来了麻烦，但是webpack已经为我们充分考虑好了这点，它支持生产Source Maps来方便我们的调试。（敲黑板，这节可能偏理论一点。）

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

在使用webpack时只要通过简单的devtool配置，webapck就会自动给我们生产source maps 文件，map文件是一种对应编译文件和源文件的方法，让我们调试起来更简单。

四种选项

在配置devtool时，webpack给我们提供了四种选项。

* source-map:在一个单独文件中产生一个完整且功能完全的文件。这个文件具有最好的source map,但是它会减慢打包速度；
* cheap-module-source-map:在一个单独的文件中产生一个不带列映射的map，不带列映射提高了打包速度，但是也使得浏览器开发者工具只能对应到具体的行，不能对应到具体的列（符号）,会对调试造成不便。
* eval-source-map:使用eval打包源文件模块，在同一个文件中生产干净的完整版的sourcemap，但是对打包后输出的JS文件的执行具有性能和安全的隐患。在开发阶段这是一个非常好的选项，在生产阶段则一定要不开启这个选项。
* cheap-module-eval-source-map:这是在打包文件时最快的生产source map的方法，生产的 Source map 会和打包后的JavaScript文件同行显示，没有影射列，和eval-source-map选项具有相似的缺点。

四种打包模式，有上到下打包速度越来越快，不过同时也具有越来越多的负面作用，较快的打包速度的后果就是对执行和调试有一定的影响。

个人意见是，如果大型项目可以使用source-map，如果是中小型项目使用eval-source-map就完全可以应对，需要强调说明的是，source map只适用于开发阶段，上线前记得修改这些调试设置。

简单的配置：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | module.exports = {    devtool: 'eval-source-map',    entry:  \_\_dirname + "/app/main.js",    output: {      path: \_\_dirname + "/public",      filename: "bundle.js"    }  }    //作者：zhangwang |

总结：调试在开发中也是必不可少的，但是一定要记得在上线前一定要修改webpack配置，在打出上线包。

**第17节：实战技巧：开发和生产并行设置**

一周没有写博客了，这一周都在出差（10月9-10月12日），肯定有小伙伴等着看教程了，在这里跟小伙伴说对不起了。这节详细讲讲用webapck开发和生产（或者说开发和上线）的那些事。把小伙伴容易迷茫的几个点讲清楚。（从这篇开始强烈建议看视频学习，文章很难表述我的意思）

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**依赖不同**

一个项目中是有开发环境和生产环境的，这两个环境的依赖也是不同的。

* 开发依赖：只在开发中用来帮助你进行开发，简化代码或者生成兼容设置的以来包。你可以打开package.json来查看，devDependencies的下面的这些包为开发使用的包。这些包在生产环境中并没有用处。
* 生产依赖：就是比如我们的js使用了jquery，jquery的程序要在浏览器端起作用，也就是说我们最终的程序也需要这个包，这就是生产依赖。这些包在dependencies中。

**npm安装**

假如我们要在项目中使用jquery库，这时候我们一般有三种安装方法，每种我都详细讲解一下。

**第一种：**



|  |  |
| --- | --- |
| 1 | npm install jquery |

安装完成后，你会发现在package.json中并不存在这个包的依赖。如果你项目拷贝给别人继续开发，或者别人和你git合作，再次下载项目npm install时就会缺少这个jquery包。项目就会无法正常运行，所以这也是我们最不赞成的安装方法。

**第二种：**



|  |  |
| --- | --- |
| 1 | npm install jquery --save |

安装完成后，它存在于package.json的dependencies中，也就是说它是生产环境需要依赖的包（上线时需要的以来包）。

**第三种：**



|  |  |
| --- | --- |
| 1 | npm install jquery --save-dev |

安装完成后，它存在于package.json的devDependencies中，也就是说它是开发环境中需要的，上线并不需要这个包的依赖。

**安装全部项目依赖包：**



|  |  |
| --- | --- |
| 1 | npm install |

**安装生产环境依赖包：**



|  |  |
| --- | --- |
| 1 | npm install --production |

**配置生产和开发并行**

我们在以前的配置中设置了一个变量**website**，用于静态资源正确找到路径。那如果生产环境和开发环境不一样，而且我们需要来回切换，这时候我们需要更好的设置方法。



|  |  |
| --- | --- |
| 1  2  3 | var website={      publicPath:"http://192.168.0.104:1717/"  } |

**修改package.json命令**

其实就是添加一个dev设置，并通过环境变量来进行区分，下面是package.json里的值。



|  |  |
| --- | --- |
| 1  2  3  4  5 | "scripts": {      "server": "webpack-dev-server --open",      "dev":"set type=dev&webapck",      "build": "set type=build&webpack"    }, |

**修改webpack.config.js文件**

可以利用node的语法来读取type的值，然后根据type的值用if–else判断。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | if(process.env.type== "build"){      var website={          publicPath:"http://192.168.0.104:1717/"      }  }else{      var website={          publicPath:"http://cdn.jspang.com/"      }  } |

如果你说我想看一下传过来的值到底是什么？可以用下面的输出语句。



|  |  |
| --- | --- |
| 1 | console.log( encodeURIComponent(process.env.type) ); |

**Mac下的package.json设置**

MAC电脑下需要把set换成export，并且要多加一个&符，具体代码如下。



|  |  |
| --- | --- |
| 1  2  3  4  5 | "scripts": {      "server": "webpack-dev-server --open",      "dev":"export type=dev&&webpack",      "build": "export type=build&&webpack"    }, |

**第18节：实战技巧：webpack模块化配置**

现在的前端开发随着ES6的普及已经大面积使用模块化进行开发了，那在webpack.config.js配置文件中，如何进行模块化开发那？例如把开发环境的写到一个模块中，把生产环境的写到一个模块中。这节课我们就用一节课的时间学习一下webpack模块化配置的技巧。

**视频教程购买地址**：[https://chungold.com](https://www.chungold.com/course/32)

**JS中的模块化实现**

先来看一下JavaScript如何实现模块话开发。其实很多小伙伴都会这种操作，那我们就当复习了，再预习一遍知识。看下面ES6中的模块化代码。



|  |  |
| --- | --- |
| 1  2  3  4 | function jspang(){      alert('jspang.com:'+'webpack');  }  module.exports=jspang; |

上面的代码是一个最简单的es6模块化写法，我们声明了一个jspang方法，并且把这个方法用module.exports进行暴露出去。然后我们在入口文件中用import进行引入，并进行使用。



|  |  |
| --- | --- |
| 1  2 | import jspang from './jspang.js';  jspang(); |

我们了解如何作Javascript的模块化后，其实webpack的模块化和上边的过程很类似。

**webpack模块**

为了让大家容易看懂，我把webpack.config.js中的entry入口文件进行模块化设置，单独拿出来制作成一个模块。

首先在根目录，新建一个webpack\_config文件夹，然后新建entry\_webpack.js文件，代码如下：

**entry\_webpack.js**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //声明entry变量  const entry ={};  //声明路径属性  entry.path={      entry:'./src/entry.js'  }  //进行模块化  module.exports =entry; |

配置的模块化代码编写好以后，需要在webpack.config.js中引入，注意这里的引入只能使用require的方法。



|  |  |
| --- | --- |
| 1 | const entry = require("./webpack\_config/entry\_webpack.js") |

然后在入口文件部分，修改成如下代码：



|  |  |
| --- | --- |
| 1 | entry:entry.path, |

这时候你可以再次使用npm  run dev 进行测试，你会发现模块化成功了。

**总结：**模块化在实际工作中是必不可少的操作，但是现在的webpack教程还很少讲到，大家一定要重视这节，因为如果你搞不清这节的内容，可能你看别人的配置也会看不明白。记得一定要动手练习操作，否则你下面的课程也没办法学习。

**第19节：实战技巧：优雅打包第三方类库**

在工作中引用第三方的框架是必不可少的，比如引入JQuery或者Vue，但是很多小伙伴一遇到引入第三方的类库时就不知道如何操作了。这节课就学习一下如何优雅并正确的用webpack引入第三方库。（**本节建议收看视频学习**）我们讲两种打包的方法，小伙伴们可以根据自己的喜好选择，个人认为两种都是比较不错的。

**引入JQuery**

其实引用第三方库的方法有很多种，但是有些并不是很优雅，还有些方法会出现打包问题，技术胖在这里介绍一下自己工作中引入第三方模块的方法，我们就拿JQuery为例。小伙伴们要举一反三，学会后试着自己引入Vue试试。

**安装JQuery**



|  |  |
| --- | --- |
| 1 | npm install --save jquery |

安装时需要注意的时Jquery最终要在生产环境中使用，所以我们这里要使用**–save**进行安装。

**修改entry.js文件**

安装好后，还需要引入到我们的entry.js中，这里直接使用import进行引入就可以。



|  |  |
| --- | --- |
| 1 | import $ from 'jquery'; |

这里引入是不需要我们写相对路径的，因为jquery的包是在node\_modules里的，只要写一个包名jquery，系统会自动为我们查找的。

引入好后我们就可以在entry.js里使用jquery，我们可以加入下面的代码，然后进行测试。



|  |  |
| --- | --- |
| 1 | $('#title').html('Hello JSpang'); |

可以看到上面是标准的jquery代码，你可以使用npm  run server 进行测试，现在代码顺利运行了，这说明我们引用的JQuery库成功了。需要说的是你不仅可以在入口中进行引入，你还可以在任何你需要的js中引入，webpack并不会重复打包，它只给我们打包一次。

**用plugin引入**

如果你觉的上面的方法和webpack没什么关系，只是普通的引入，webpack只是负责了一下打包，这样并没有全局感。那再学习一种在webapck.config.js中配置的方法，这种不需要你在入口文件中引入，而是webpack给你作了全局引入。这个插件就是**ProvidePlugin**。

ProvidePlugin是一个webpack自带的插件，Provide的意思就是装备、提供。因为ProvidePlugin是webpack自带的插件，所以要先再webpack.config.js中引入webpack。



|  |  |
| --- | --- |
| 1 | const webpack = require('webpack'); |

在webpack.config.js里引入必须使用require，否则会报错的，这点小伙伴们一定要注意。

引入成功后配置我们的plugins模块，代码如下。



|  |  |
| --- | --- |
| 1  2  3  4  5 | plugins:[      new webpack.ProvidePlugin({          $:"jquery"      })  ], |

配置好后，就可以在你的入口文件中使用了，而不用再次引入了。这是一种全局的引入，在实际工作中也可以很好的规范项目所使用的第三方库。

**总结：**每一个项目都可能引入第三方类库，而像Vue和Angular这样的成熟框架都推出了自己的webpack框架，比如**vue-cli**。但是很多情况还是需要我们手动更改这些配置好的webpack来适用于我们的公司项目，所以这节课的知识也是在工作中经常使用的，希望小伙伴们一定要重视并进行练习。

**第20节：实战技巧：watch的正确使用方法**

在初级开发阶段，使用webpack-dev-server就可以充当服务器和完成打包任务，但时随着你项目的进一步完成，可能需要前后台联调或者两个前端合并代码时，就需要一个公共的服务器了。这时候我们每次保存后手动打包显然效率太低，我们希望的场景是代码发生变化后，只要保存，webpack自动为我们进行打包。这个工具就是watch，这节课我们把wacht完全学会，你会发现在开发中更加的得心应手。

**watch的配置**

很多小伙伴认为–warch直接使用就可以，并没有什么需要讲的。其实这只是初级的用法，但是在学习一种技术时，我们必须要做到了解全部，也就是常说的知其然知其所以然。我们看下面的配置代码，我在代码中已经做出了解释。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | watchOptions:{      //检测修改的时间，以毫秒为单位      poll:1000,      //防止重复保存而发生重复编译错误。这里设置的500是半秒内重复保存，不进行打包操作      aggregateTimeout:500,      //不监听的目录      ignored:/node\_modules/,  } |

上边的每一行配置我都作了说明，有时候你在没配置的情况下，直接用webpack –watch是不起作用的，这时候你需要进行配置这些选项。

配置好后，我们就可以痛快的使用watch了，在大型项目中，这大大加快了我们的开发效率，不用反复的手动打包了。

**BannerPlugin插件**

由于这节课的内容太少了，我们再讲一个工作中的小技巧，再工作中每个人写的代码都要写上备注，为的就是在发生问题时可以找到当时写代码的人。有时候也用于版权声明。

这个插件就是BannerPlugin，我们使用后会在JS中加上我们的版权或开发者声明。



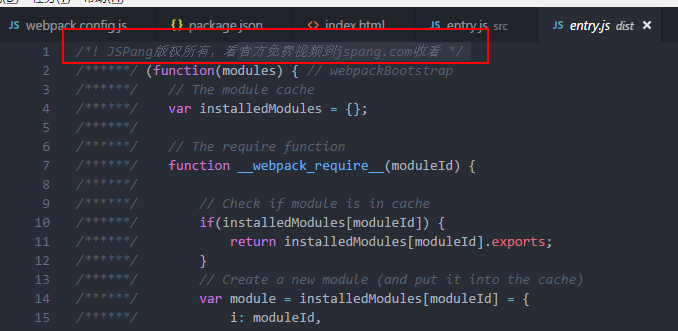
|  |  |
| --- | --- |
| 1 | new webpack.BannerPlugin('JSPang版权所有，看官方免费视频到jspang.com收看') |

需要注意的是在使用这个插件之前必须引入webpack。



|  |  |
| --- | --- |
| 1 | const webpack = require('webpack'); |

这时在dist目录下的entery.js已经加上了版权声明。



**第21节：实战技巧：webpack优化黑技能**

作为一个程序员，无论是写什么程序都i要有一颗不断优化的心。webpack在优化这条路上，也为我们作了很多配置，这节课我们就看看工作中常用的webpack优化黑技能。

**ProvidePlugin和import**

在第19节中学习了如何引入第三方类库，并引入了jquery，在引用JQuery时我们用了两种方法，第一种时import，第二种时使用ProvidePlugin插件。那两种引入方法有什么区别那?

* import引入方法：引用后不管你在代码中使用不适用该类库，都会把该类库打包起来，这样有时就会让代码产生冗余。
* ProvidePlugin引入方法：引用后只有在类库使用时，才按需进行打包，所以建议在工作使用插件的方式进行引入。

具体的对比操作，会在视频中演示，你会看出两种引入方法的对比打包结果。差距还是蛮大的。

**抽离JQuery**

上边的方法只是优化的第一步，工作中你会发现，不适用的类库几乎我们也不会引入，所以上边只是一个必要操作的第一步。那往往把第三方类库抽离出来，才是最好的解决方法。

**第一步：修改入口文件**

抽离的第一步是修改入口文件，把我们的JQuery也加入到入口文件中，看下面的代码。

**webpack.config.js**



|  |  |
| --- | --- |
| 1  2  3  4 | entry:{          entry:'./src/entry.js',          jquery:'jquery'  }, |

**第二步：引入插件**

我们需要引入optimize优化插件，插件里边是需要配置的，具体配置项看下面的代码。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | new webpack.optimize.CommonsChunkPlugin({      //name对应入口文件中的名字，我们起的是jQuery      name:'jquery',      //把文件打包到哪里，是一个路径      filename:"assets/js/jquery.min.js",      //最小打包的文件模块数，这里直接写2就好      minChunks:2  }), |

**minChunks**一般都是固定配置，但是不写是不行的，你会打包失败。

f**ilename**是可以省略的，这是直接打包到了打包根目录下，我们这里直接打包到了dist文件夹下边。

配置完成后，我们可以先删掉以前打包的dist目录，然后用webpack再次打包，你会发现jquery被抽离了出来，并且我们的entry.js文件变的很小。

**多个第三方类库抽离**

会了如何抽离Jquery，但是在实际开发中，我们会引用不止一个第三方类库，这时也需要抽离。我们拿引入Vue为例，看看如何抽离出来。

**第一步:我们先用npm 进行安装。**



|  |  |
| --- | --- |
| 1 | npm instawll vue --save |

注意这里是–save，而不是–save-dev。因为这个类库在生产环境中也是要使用的。

**第二步：在入口配置中引入vue和jquery**



|  |  |
| --- | --- |
| 1  2  3  4  5 | entry:{      entry:'./src/entry.js',      jquery:'jquery',      vue:'vue'  }, |

只是多比上边多加了一个vue选项。

**第三步：修改CommonsChunkPlugin配置**

需要修改两个位置：

* 第一个是在name属性里把原来的字符串改为数组，因为我们要引入多个模块，所以是数组；
* 第二个是在filename属性中把我们输出的文件名改为匹配付[name],这项操作就是打包出来的名字跟随我们打包前的模块。

下面是我们修改的代码，你可以跟jquery抽离时对比一下。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | new webpack.optimize.CommonsChunkPlugin({      //name对应入口文件中的名字，我们起的是jQuery      name:['jquery','vue'],      //把文件打包到哪里，是一个路径      filename:"assets/js/[name].js",      //最小打包的文件模块数，这里直接写2就好      minChunks:2  }), |

配置好后，我们就可以在控制台输入webpack进行打包了。你会看到我们预想的结果，jquery和vue都被我们抽离出来了。

**总结：**在项目开发中，我们很使用很多第三方类库，那好的做法就是把第三方这些类库全部抽离处理，这样在项目维护和性能上都是不错的选择。希望学会这个技巧后，你也能在工作中使用上。

**第22节：实战技巧：静态资源集中输出**

工作中会有一些已经存在但在项目中没有引用的图片资源或者其他静态资源（比如设计图、开发文档），这些静态资源有可能是文档，也有可能是一些额外的图片。项目组长会要求你打包时保留这些静态资源，直接打包到制定文件夹。其实打包这些资源只需要用到copy-webpack-plugin。

**使用copy-webpack-plugin**

copy-webpack-plugin就是专门为我们作静态资源转移的插件，不过它不同上两节使用的插件，它是需要安装的。

**插件安装**

插件的安装只要使用npm就可以了。



|  |  |
| --- | --- |
| 1 | cnpm install --save-dev copy-webpack-plugin |

如果在安装过程中出错，你可以使用npm来进行安装。

**引入插件**

安装好后，需要在webpack.config.js文件的头部引入这个插件才可以使用。



|  |  |
| --- | --- |
| 1 | const copyWebpackPlugin= require("copy-webpack-plugin"); |

**配置插件**

引入之后我们就可以在plugins里边进行配置插件了，我们先看下面的插件配置代码，然后再进行详细讲解。



|  |  |
| --- | --- |
| 1  2  3  4 | new copyWebpackPlugin([{          from:\_\_dirname+'/src/public',          to:'./public'      }]) |

* from:要打包的静态资源目录地址，这里的\_\_dirname是指项目目录下，是node的一种语法，可以直接定位到本机的项目目录中。
* to:要打包到的文件夹路径，跟随output配置中的目录。所以不需要再自己加\_\_dirname。

配置好后，我们就可以使用webpack 进行打包了，你会发现图片按照我们的配置打包了过去。

**总结：**你是不是觉的学习起来越来越容易了，因为你经过20多节课的学习，已经基本掌握了webpack的知识，剩下的就是不断练习和在实际项目中发现新的需求，然后找到新的loader或者pulgin来解决你的问题。

**第23节：实战技巧：Json配置文件使用**

在实际工作中，我们的项目都会配置一个Json的文件或者说API文件，作为项目的配置文件。有时候你也会从后台读取到一个json的文件，这节课就学习如何在webpack环境中使用Json。如果你会webpack1或者webpack2版本中，你是需要加载一个json-loader的loader进来的，但是在webpack3.x版本中，你不再需要另外引入了。

**读出Json内容**

**第一步**：现在我们的index.html模板中加入一个层，并给层一个Id，为了是在javascript代码中可以方便引用。



|  |  |
| --- | --- |
| 1 | <div id="json"></div> |

**第二步**：到src文件夹下，找到入口文件，我这里是entry.js文件。修改里边的代码，如下：



|  |  |
| --- | --- |
| 1  2 | var json =require('../config.json');  document.getElementById("json").innerHTML= json.name; |

这两行代码非常简单，第一行是引入我们的json文件，第二行驶写入到到DOM中。

**第三部**：启用我们的npm run server 命令就可以在浏览器中看到结果了。

**说说热更新**

其实在webpack3中启用热加载相当的容易，只要加入HotModuleReplacementPlugin这个插件就可以了。



|  |  |
| --- | --- |
| 1 | new webpack.HotModuleReplacementPlugin() |

现在只要你启动 npm run server 后，修改index.html中的内容，浏览器可以自动给我们更新出最新的页面。

但这里说的热加更新和我们平时写程序的热加载不是一回事，比如说我们Vue或者React中的热更新，并不是刷新整个页面，而是一个局部更新，而这里的更新是重新刷新了页面。

这点小伙伴们要搞清楚。

**总结：**这节主要回答了两个同学的问题，这两个问题在实际开发中也经常用到，所以我就总结成课，给小伙伴们讲解一下，希望对你们有所帮助。

**第24节：剧终-webpack自学技巧传授**

本来打算在最后加一个多页webpack实战给大家的，但是这个要录制10多节，所以决定再独立出一套视频教程给大家（当然这是免费的）。请关注我的博客（http://jspang.com）。

这节课我会教大家一些自学webpack的技巧，让大家可以再本教程结束后继续自行精进，也算倾囊相授。（精彩内容视频呈现，请关注视频内容）