## 课程简介

## 学习准备

### 模块化(1)

### 模块化(2)

### 环境准备(mac)

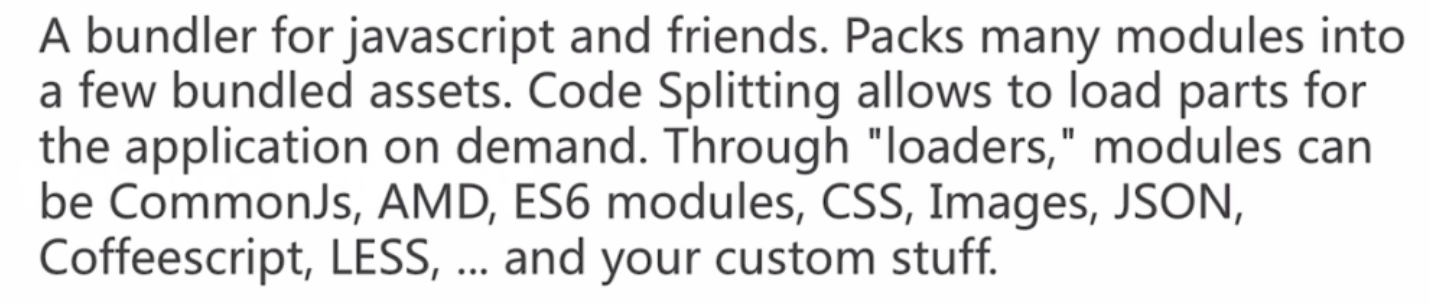
主要就是安装 webpack

**版本是3.10.0**

***npm install webpack@3.10.0 -g***

### webpack 简介

》》下面是官网对 webpack 的一个介绍



》》新官网地址

*https://webpack.js.org*

目前的版本是3.8.1

github 地址：

*https://github.com/webpack/webpack*

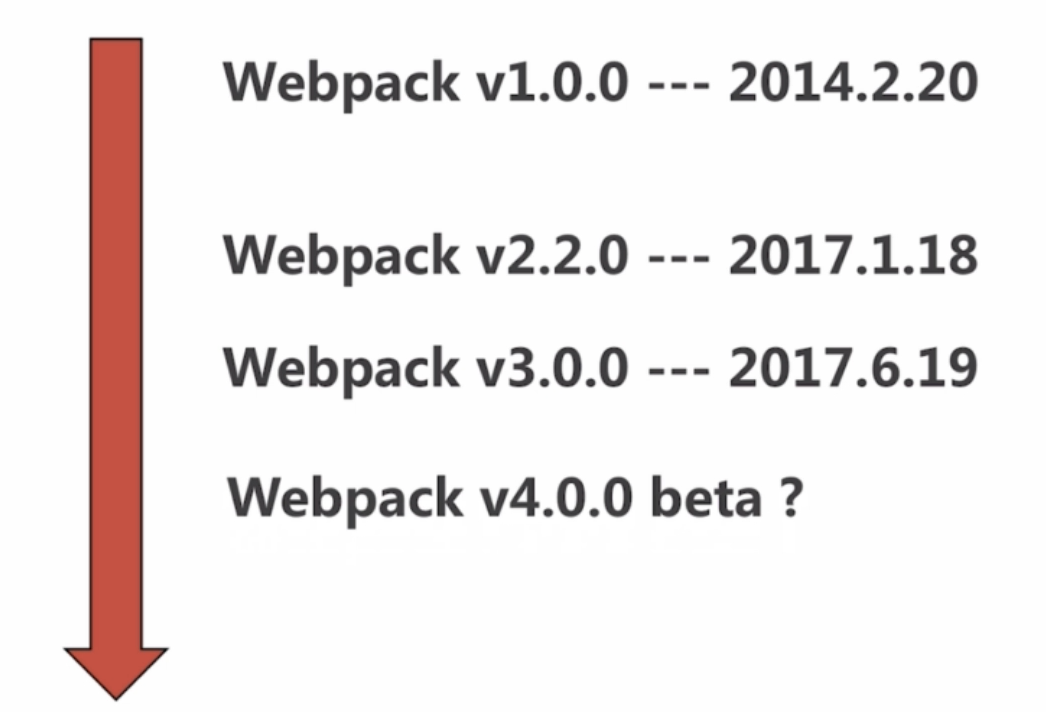
中文的官网：

*https://doc.webpack-china.org*

》》webpack 版本更迭

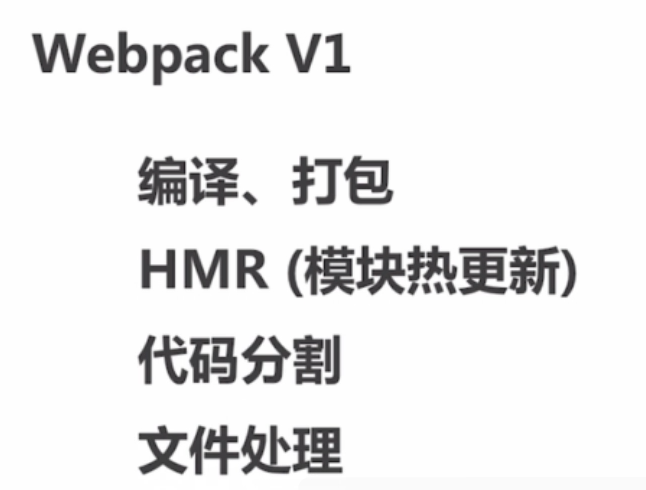
*https://github.com/webpack/webpack/release*

》》大版本变化



讲课的时候 4.0还没有发布正式版

》》功能进化





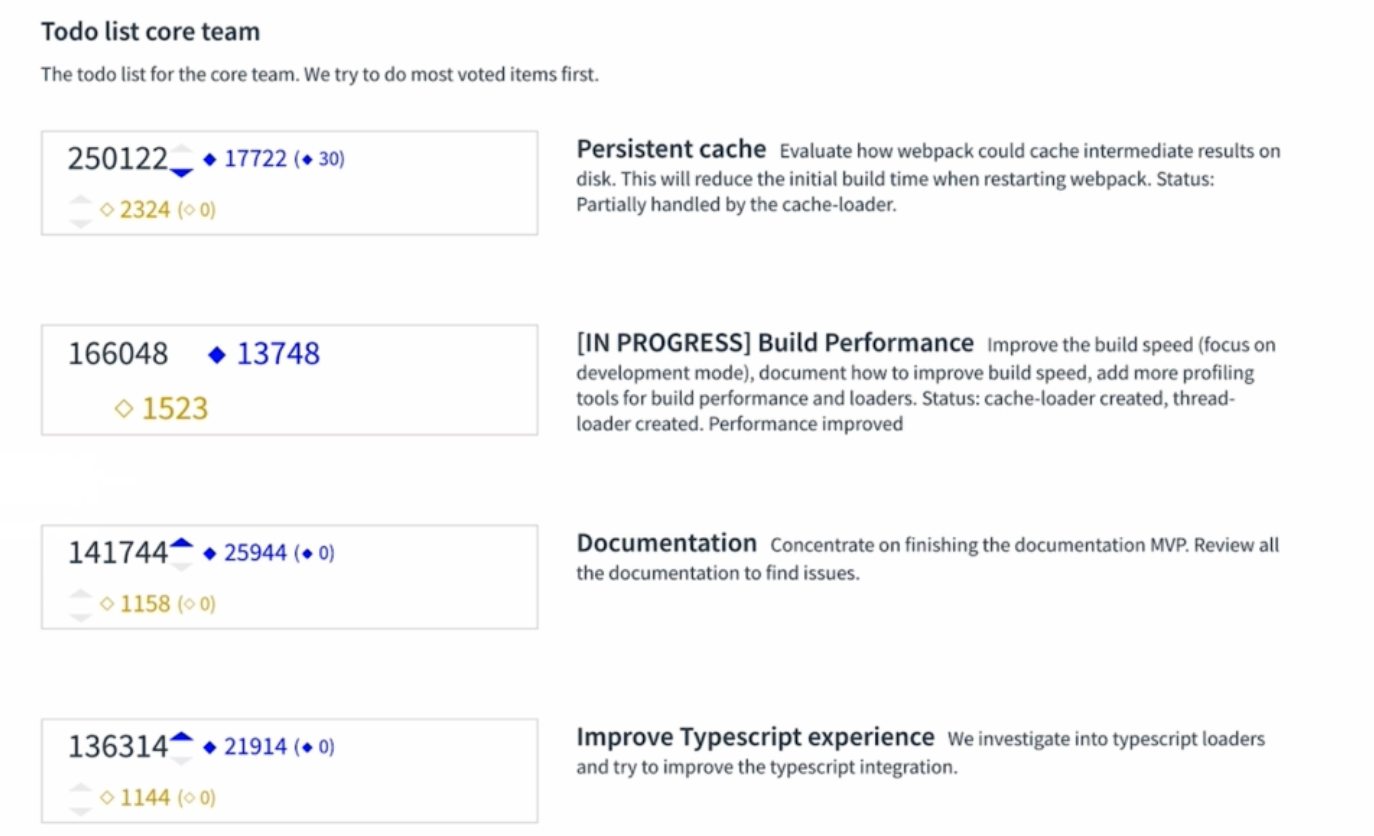


》》版本迁移



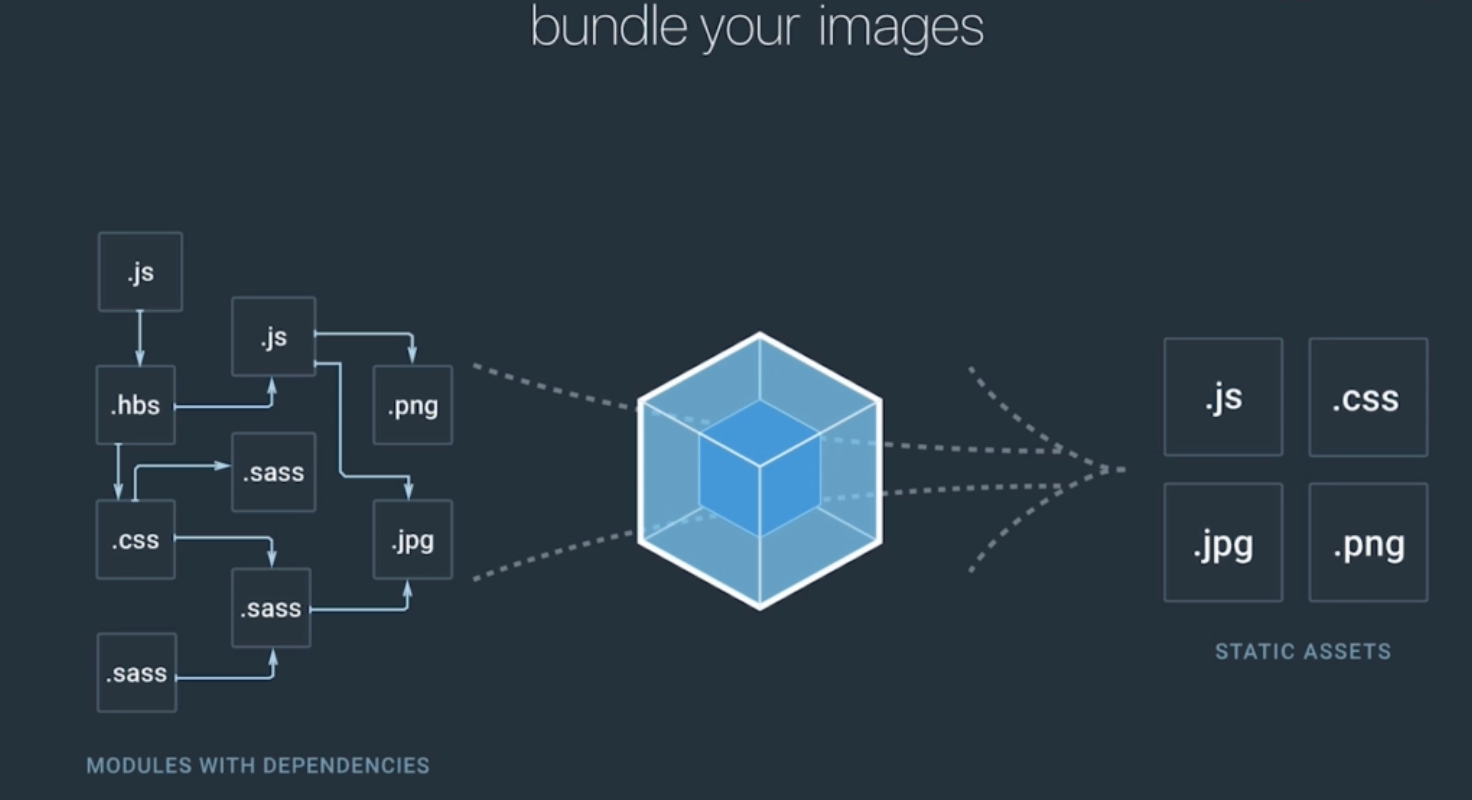
》》参与社区投票

可以看见核心团队的 TodoList



vote: https://webpack.js.org/vote

》》新官网



### webpack 核心概念

从 1.0 版本开始的核心概念就没有变过

四个：**Entry**、**Output**、**Loaders**、**Plugins**

1、Entry

主入口，理解某一段**代码的入口**，需要告诉 webpack，import 了什么，有什么初始化的代码，可以间接的找到依赖的一些模块

其实也是**打包的入口**，依赖找依赖。找全所有的

Entry 可以是**一个**入口，也可以是**多个入口**的。多入口：多页面的应用 ，单页面的应用读到的代码是一个 Entry，框架的是一个 Entry

module.exorts 是一个对象，就是 webpack 的对象，里面有一个 entry 的属性，这个属性的值可以是路径，也可以是单独的 js 文件。这个值也是可以是数组的。

|  |
| --- |
| module.exports{  entry: 'index.js'  }  module.exports{  entry: ['index.js', 'vendor.js']  }  // 这里的每个 entry 是有名字的，就会知道是哪个 entry，推荐这种写法的  // 想要继续添加一个新的入口，可以灵活的添加  module.exports{  entry: {  index: 'index.js',  vendor: 'vendor.js' // 添加新的入口  }  } |

2、Output

表示输出

**打包生成的文件的 bundle** 的一个描述

也有是**一个或多个**

当多个的时候**自定义规则**，这个 output 对应的规则名

代码演示：

|  |
| --- |
| module.exports={  entry: 'index.js'  output: {  filename: 'index.min.js' // 指定输出的文件名  }  }  module.exports={  entry: {  index: 'index.js',  vendor: 'vendor.js' // 添加新的入口  }  output: {  filename: '[name].min.[hash:5].js' // name 就是 entry 的 name，hash 是 md5 过后的一个字符串，这就是自定义规则的  }  }  // 还可以配合 cdn，后面会讲到 |

3、Loaders

可以处理 js 之外的东西

使用来**处理文件**的

把我们所说的**文件转化成模块**，如下面的代码就是，可以当成 js 的一个模块，引入进来

|  |
| --- |
| module.exports={  **module**: {  rules: {  test: /\.css$/,  use: 'css-loader'  }  }  } |

哪些常用的 loader

》》编译相关的：babel-loader, ts-loader

》》样式相关的：style-loader, css-loader, less-loader, postcss-loader

》》文件相关的：file-loader, url-loader

4、Plugins

强大的插件系统

是参与整个打包过程的

打包优化和压缩

配置编译时的变量

极其灵活

|  |
| --- |
| const webpack = require('webpack')  module.exports={  plugins: {  new webpack.optimize.UglifyJsPlugin()  }  } |

常用 Plugins

》》优化相关的：CommonsChunkPlugin，UglifyjsWebpackPlugin

》》功能相关的：ExtractTextWebpackPlugin，HtmlWebpackPlugin，HotModuleReplacementPlugin

CopyWebpackPlugin

》》名词

Chunk：这个是什么了？英文解释就是块，在 webpack 里面就是代码块。

Bundle：英文解释就是绑定，webpack 就是打包的意思了

Module：图片，通过 loader 处理完以后就一个模块

## 由浅入深 webpack(1)

### 使用 webpack

》》使用 webpack 的方式

1、webpack 命令的方式

2、使用 webpack 配置的，平时工作中，多使用这个方式，配合 node，npm 一起使用

3、第三方脚手架，很常见的，如 vue-cli。等等， vue 等这些框架都会有

下面就是详细的讲解：

第一种： webpack 命令

webpack -h ：帮助

webpack -v ：版本

webpack <entry>[<entry>] <output> : 打包命令

下面介绍一个很厉害的，很惊讶的东西，**webpack-cli**

***npm install webpack-cli@3.3.10 -g***

这是额外插入的介绍的： webpack-cli 不过最后也没怎么用这个东东

1、可以创建一个交互式的初始化项目，后面填写的是模板的名称

webpack-cli init wepack-addons-demo

2、迁移项目：v1 -> v2

webpack-migrate 迁移文章，就进行迁移了

刚开始还是很惊艳的，很激动的，但是细细一想，或者脚手架的确实，也没有官方的脚手架，所以还是慢慢丰富，所以 cli 的功能还是不太好用。

目前来看，也很久没更新了，v4 版本都是alpha。

第二种： webpack 配置

webpack

webpack --config webpack.conf.dev.js

可以不指定配置文件，就会使用默认的配置文件，也可以手动指定配置文件

第三种： 使用第三方的脚手架

vue-cli,angular-cli,react-starter

### 打包 JS

两种方式

1、命令行工具：webpack entry<entry> output

entry可以是多个，output 可以是自定义规则

2、配置文件方式：webpack --config webpack.config.js

更加灵活的方式

**》》命令行模式：**

mkdir demo3-2

demo3-2->app.js

|  |
| --- |
| // es module  import sum from './sum'  // commonjs  var minus = require('./minus')  // amd  require(['./multi'], function(multi){  console.log('multi(2, 3) =', multi(2, 3))  })  console.log('sum(23, 24) = ', sum(23, 24))  console.log('minus(17, 3) = ', sum(17, 3)) |

demo3-2->sum.js

|  |
| --- |
| export default function(a, b) {  return a + b  } |

执行命令*webpack app.js bundle.js*

demo3-2->index.html

|  |
| --- |
| <body>  <script src="./bundle.js">  </body> |

直接用浏览器打开这个 html 文件

这个时候再添加 minus.js 文件，然后继续在 app.js引入这个文件，并执行 minus 方法

demo3-2->minus.js

|  |
| --- |
| moduel.exports = function(a, b){ // commonjs的规范  return a - b  } |

继续执行打包命令*webpack app.js bundle.js*

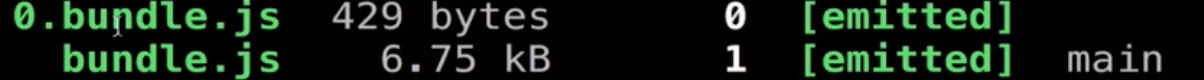
然后在浏览器访问html 文件，可以看见 minus 方法也执行了

这个时候再添加 multi.js文件，使用 AMD 规范

|  |
| --- |
| define(function(require, factory){  'use strict';  return function(a, b) {  return a \* b  }  }) |

继续执行打包命令*webpack app.js bundle.js*

这个时候会产生两个 bundle 的 js 文件，AMD 是异步的加载，所以形成了一个 js 文件



**》》配置文件模式**

webpackconfigjs的是 commonjs 的规范

demo3-2 -> webpack.config.js

|  |
| --- |
| module.exports = {  entry: {  app: './pp.js'  },  output: {  filename: '[name].[hash:5].js'  }  } |

继续执行打包命令*webpack --config webpack.conf.js*

如果配置文件是 webpack.config.js的话，就可以直接执行 webpack 命令。

A picture containing object

Description automatically generated

看浏览器的 network 页面：

A screenshot of a cell phone

Description automatically generated

### 编译 ES6/7

如果需要对 ES6/7的语法的编译支持，需要学习 babel，以及 babel-presets，还有 babel-plugin

babel-loader：编译性质的 loader，并且速度不是太快，需要做很多的词法分析的工作

官网地址是：babel.io

Babel的安装

*npm install babel-loader@8.0.0-beta.0 @babel/core*

*npm install babel-loader babel-core --save-dev*

进入demo3-3的目录：

npm init 等一系列交互后，就生成了 package.json的文件

使用最新版的 babel 来安装

**》》新建几个文件，app.js,index.html,webpack.config.js，目录如下：**

A screen shot of a social media post

Description automatically generated

webpack.config.js

|  |
| --- |
| module.exports = {  entry: {  app: './app.js' // 指定相对路径  },  output: {  fielname: '[name].[hash:5].js'  },  module: {  // rules是一个数组  rules: [  test: '/\.js$/',  // use: 'babel-loader',  use: {  loader: 'babel-loader',  options: {  presets: ['@babel/preset-env'， { }]  }  }  exclude: '/node\_modules/'  ]  }  } |

targets: { browsers: ['> %1','last 2 versions '] } 对这个版本打包，发现箭头函数变了

targets: { chrome: '52' } 对这个版本打包，发现箭头函数没有，支持挺好

rules: 是一个数组，里面有很多的规则，

test: 是一个正则，符合这个正则会使用下面的use所复制的具体 loader

use: 具体使用的 loader 的名称，use 也可以写成对象的模式，可以指定 presets 等参数

exclude：可以进行具体的排除，如这里对 nodule\_modules 进行排除，不进行当前 loader 的编译

target 参数：执行最终的环境

app.js

|  |
| --- |
| let func = () => {}  const NUM = 45  let arr = [1, 2, 3]  let arrB = arr.map(item => item \* 2)  arr.includes(8)  console.log('new Set(arrB)', new Set(arrB)) |

命令行执行：webpack 进行打包。就可以看见打包的结果：

看见箭头函数已经变了

A screenshot of a cell phone

Description automatically generated

**》》Babel Presets**

就是一些规则的总结，预设的一些规则的模板

babel的一些列表如下面所示：

es2015

es2016

es2017

env

babel-preset-react

babel-preset-stage 0 -3

preset也是需要安装的，如下面的安装命令：

*npm install @babel/preset-env --save-dev 最新版本的*

*npm install babel-preset-env --save-dev 正常版本的*

target 参数：根据指定的目标，哪些语法编译，哪些语法不编译

targets.browsers 指定浏览器环境，如 'last 2 versions' 或者 '> %1'

这些数据是从 browserlist 来的，而这些数据又是从 Can I Use 来的

**》》Babel Polyfill 和Babel Runtime Transform 两个插件**

上面的includes和 set 可能没有在浏览器里面实现，并没有对他进行编译，上面的 presets 是针对的语法，这两个插件就是针对的函数和方法的。

有哪些方法和函数了？

Generator，Set，Array.from，Array.prototype.includes

这些方法都没有在 presets 里面进行处理

Babel Polyfill：填充器，垫片。这个**垫片是全局**的，在浏览器里面的方法里面进行调用，为**开发应用准备**的，

*npm install babel-polyfill --save*

*import 'babel-polyfill'*

Babel Runtime Transform：**局部的垫片**，**开发框架准备**的，不会污染全局，局部里面新增方法。另外的好处：代码里面使用的时候，如果 ES6，ES7语法，不会打包相同的代码

*npm install babel-plugin-transform-runtime --save-dev*

*npm install babel-runtime --save*

*npm install @babel/plugin-transform-runtime --save-dev*

*npm install @babel/core --save*

新建 .babelrc 文件，在这个里面配置

下面演示 plyfill 和 runtime 两个插件

》》polyfill 的使用

|  |
| --- |
| import 'babel-polyfill' // 这一句就可以使用了 polyfill 了  function\* func(){  } |

所有的实现，都会给你实现，然后加载这个打包的 js 文件里面

A close up of a screen

Description automatically generated

》》runtime 的使用

在工程根目录新建 .babelrc 文件，其实就是一个 json 文件,如下面的截图所示

A screenshot of a cell phone

Description automatically generated

打开打包后的文件，看看里面的内容

A screenshot of a cell phone screen with text

Description automatically generated

### 编译 typescript

先简单介绍下TS，他是 JS 的超集

typescritplang.org

tslang.cn

TS 其实是来自于微软

如果要在 webpack 中使用 TS，就需要使用typescript-loader，那么这里有两个选择

安装:

n*pm install typescript ts-loader --save-dev*

*npm install typescript awesome-typescript-loader --save-dev*

配置：tsconfig.json 的配置文件，webpack.config.js是关于 webpack 的配置

配置选项：官网/docs/handbook/compiler-options.html

常用选项：

compilerOptions

include

exclude

项目实战来

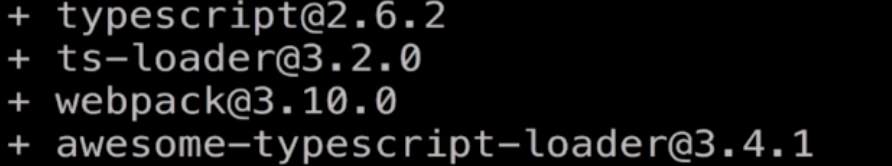
按照重新的项目初始化目录来演示

npm init

新建tsconfig.json文件，然后安装

npm install webpack typescript ts-loader awesome-typescript-loader --save-dev

可以看见版本是：



webpack.config.js

|  |
| --- |
| module.exports = {  entry: {  app: './app.js'  },  output: {  filename: '[name].[hash:5].js'  },  module: {  rules: [  {  test: '/\.tsx?$/',  use: {  loader: 'ts-loader'  }  }  ]  }  } |

tsconfig.js

|  |
| --- |
| {  "compilerOptions":{  "module": "commonjs",  "target": "es5",  "allowJs": true,  },  "include": ["./src/\*"],  "exclude": ["./node\_modules"]  } |

module：

target：编译后的文件在什么环境之下执行的

allowJs: 是否允许 js 文件

src/app.ts

|  |
| --- |
| const NUM = 45  interface Cat {  name: String,  sex: String  }  function touchCat(cat: Cat) {  console.log('miao~', cat.name)  }  touchCat({name: 'tom', sex: 'gender'}) |

然后我们进行 webpack 进行编译。

看编译后的代码如下：

A screenshot of a cell phone

Description automatically generated

》》实际开发中，我们可能还是用 loadsh 这种包

npm install lodash --save

src/app.ts 新增的代码是

|  |
| --- |
| import \* as \_ from 'lodash'  console.log(\_.chunk([1,2,3,4,5], 2)) |

然后打包

》》声明文件

*npm install @types/lodash*

*npm install @types/vue*

如果对代码里的进行错误的写法，编译器不会有提示，需要安装这些声明文件，然后编译打包的时候就可以进行错误提示了。

》》typings

*npm install typings*

*typings install lodash*

如果上面的声明文件很多的时候，就可以使用这个 typings 方法，装完以后，就用 typins 来安装

A screenshot of a cell phone

Description automatically generated

### 打包公共代码(1)

模块化开发，就会遇见公共模块的问题，为什么需要提取出来了？

1、减少代码的冗余，避免公共代码在每个页面上都有，带宽的浪费，也会提高加载速度

### 打包公共代码(2)

## 由浅入深 webpack(2)