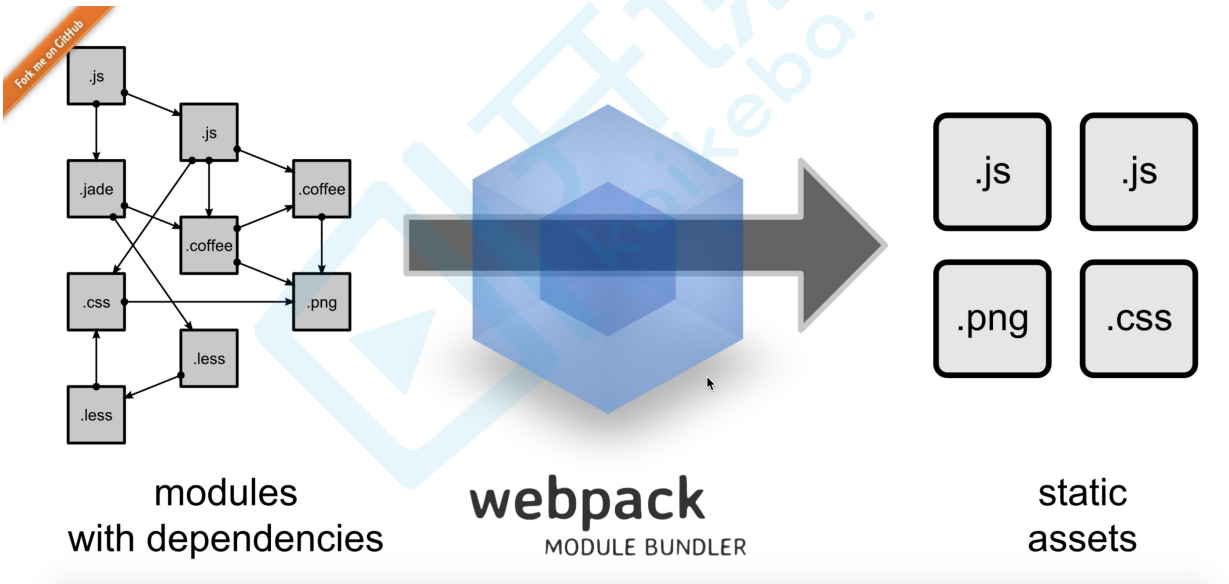
Webpack-Day1

课前准备

- nodeJS
- webpack

什么是webpack



webpack is a module bundler.(模块打包工具)

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（Scss，TypeScript等），并将其打包为合适的格式以供浏览器使用。

- 官方网站：https://webpack.js.org/

安装

- 环境：nodeJs https://nodejs.org/en/

  版本参考官网发布的最新版本，可以提升webpack的打包速度

- 全局 不推荐

```
npm install webpack webpack-cli -g//webpack-cli 可以帮助我们在命令行里使用npx
,webpack等相关指令

webpack -v

npm uninstall webpack webpack-cli -g
```

- 局部安装 项目内安装

```
npm install webpack webpack-cli --save-dev //-D

webpack -v //command not found 默认在全局环境中查找

npx webpack -v// npx帮助我们在项目中的node_modules里查找webpack
```

- 安装指定版本

```
npm info webpack//查看webpack的历史发布信息

npm install webpack@x.xx webpack-cli -D
```

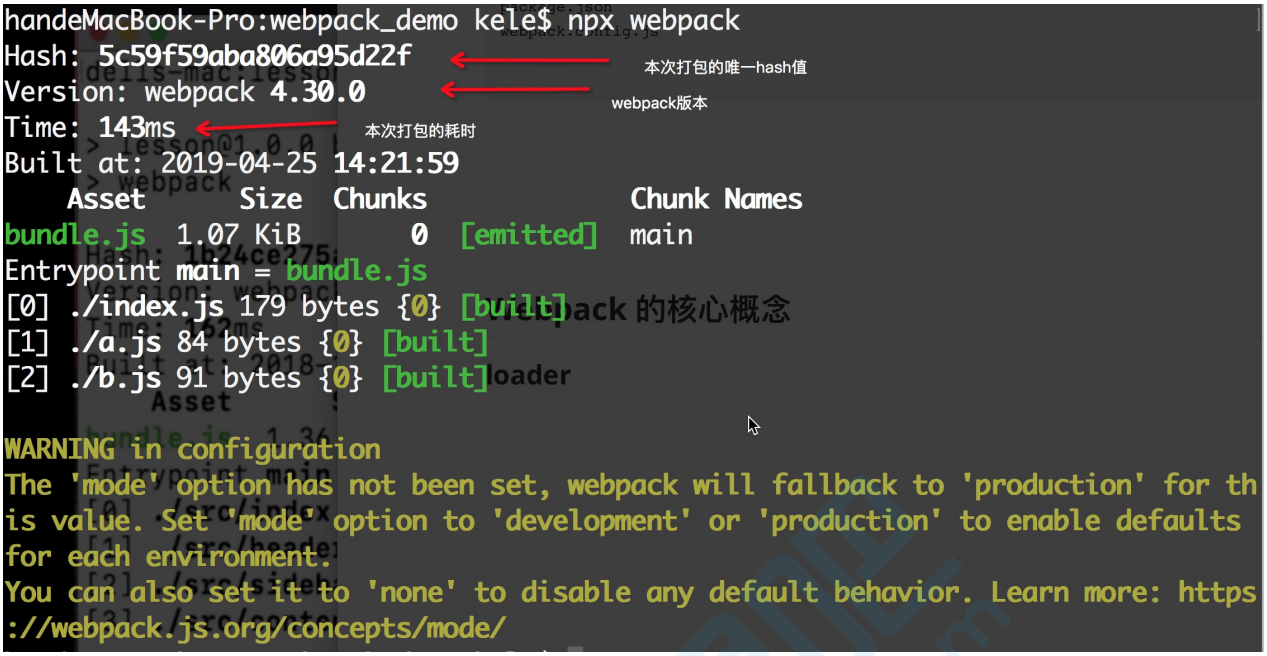
测试：启动webpack打包

```
// es moudule 模块引入
// commonJs 模块引入

import add from "./a"; //需要使用es moudule导出
import minux from "./b";////需要使用es moudule导出


npx webpack index.js //打包命令 使用webpack处理index.js这个文件
```

总结：webpack 是一个模块打包工具，可以识别出引入模块的语法 ，早起的webpack只是个js模块的打包工具，现在可以是css，png，vue的模块打包工具



webpack 配置文件

零配置是很弱的，特定的需求，总是需要自己进行配置

当我们使用npx webpack index.js时，表示的是使用webpack处理打包，名为index.js的入口模块。默认放在当前目录下的dist目录，打包后的模块名称是main.js，当然我们也可以修改

webpack有默认的配置文件，叫webpack.config.js，我们可以对这个文件进行修改，进行个性化配置

- 默认的配置文件：webpack.config.js

```
npx webpack //执行命令后，webpack会找到默认的配置文件，并使用执行
```

- 不使用默认的配置文件： webpackconfig.js

```
npx webpack --config webpackconfig.js //指定webpack使用webpackconfig.js文件来作为配置文件并执行
```

- 修改package.json scripts字段：有过vue react开发经验的同学 习惯使用npm run来启动，我们也

可以修改下

```
"scripts":{
  "bundle":"webpack"//这个地方不要添加npx ,因为npm run执行的命令，会优先使用项目工程里的包，效果和npx非常类似
}

npm run bundle
```

项目结构优化

```
dist
  //打包后的资源目录
node_modules
  //第三方模块
src
  //源代码
  css
  images
  index.js

package.json
webpack.config.js
```

Webpack 的核心概念

entry:

指定打包入口文件:Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入

```
entry:{
  main: './src/index.js'
}
==相当于简写===
entry:"./src/index.js"
```

output:

打包后的文件位置:输出结果，在 Webpack 经过一系列处理并得出最终想要的代码后输出结果。

```
output: {
    publicPath:"xxx",
    filename: "bundle.js",
    //   必须是绝对路径
    path: path.resolve(__dirname, "dist")
  },
```

loader

模块转换器，用于把模块原内容按照需求转换成新内容。

webpack是模块打包工具，而模块不仅仅是js，还可以是css，图片或者其他格式

但是webpack默认只知道如何处理js模块，那么其他格式的模块处理，和处理方式就需要loader了

moudle

模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。

```
module:{
  rules:[
  {
    test:/\.xxx$/,
    use:{
      loader: 'xxx-load'
    }
  }
  ]
}
```

当webpack处理到不认识的模块时，需要在webpack中的module处进行配置，当检测到是什么格式的模块，使用什么loader来处理。

- loader: file-loader：处理静态资源模块

  loader: file-loader

  原理是把打包入口中识别出的资源模块，移动到输出目录，并且返回一个地址名称

  所以我们什么时候用file-loader呢?

  场景：就是当我们需要模块，仅仅是从源代码挪移到打包目录，就可以使用file-loader来处理，txt，svg，csv，excel，图片资源啦等等

  ```
  npm install file-loader -D
  ```

  案例：

```
module: {
    rules: [
      {
        test: /\.(png|jpe?g|gif)$/,
        //use使用一个loader可以用对象，字符串，两个loader需要用数组
        use: {
          loader: "file-loader",
          //   options额外的配置，比如资源名称
          options: {
            //  placeholder 占位符  [name]老资源模块的名称
            //   [ext]老资源模块的后缀
            // https://webpack.js.org/loaders/file-loader#placeholders
            name: "[name]_[hash].[ext]",
            //打包后的存放位置
            outputPath: "images/"
          }
        }
      }
    ]
  },
```

- url-loader

可以处理file-loader所有的事情，但是遇到jpg格式的模块，会把该图片转换成base64格式字符串，并打包到js里。对小体积的图片比较合适，大图片不合适。

```
npm install url-loader -D
```

案例；

```
module: {
    rules: [
      {
        test: /\.(png|jpe?g|gif)$/,
        use: {
          loader: "url-loader",
          options: {
            name: "[name]_[hash].[ext]",
            outputPath: "images/",
            //小于2048，才转换成base64
            limit: 2048
          }
        }
      }
    ]
  },
```

样式处理：

Css-loader 分析css模块之间的关系，并合成一个css

Style-loader 会把css-loader生成的内容，以style挂载到页面的heade部分

```
npm install style-loader css-loader -D
```

```
{
  test: /\.css$/,
  use: ["style-loader", "css-loader"]
}
```

sass样式处理

sass-load 把sass语法转换成css ，依赖node-sass模块

```
npm install sass-loader node-sass -D
```

案例：

loader有顺序，从右到左，从下到上

```
{
  test: /\.scss$/,
  use: ["style-loader", "css-loader", "sass-loader"]
}
```

样式自动添加前缀：

Postcss-loader

```
npm i -D postcss-loader
```

webpack.config.js

```
{
        test: /\.css$/,
        use: ["style-loader", "css-loader", "postcss-loader"]
      },
```

新建postcss.config.js

安装autoprefixer

```
//npm i autoprefixer -D

module.exports = {
  plugins: [require("autoprefixer")]
};
```

Plugins

plugin 可以在webpack运行到某个阶段的时候，帮你做一些事情，类似于生命周期的概念

扩展插件，在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情。

HtmlWebpackPlugin

htmlwebpackplugin会在打包结束后，自动生成一个html文件，并把打包生成的js模块引入到该html中。

```
npm install --save-dev html-webpack-plugin
```

配置：

```
title: 用来生成页面的 title 元素
filename: 输出的 HTML 文件名，默认是 index.html, 也可以直接配置带有子目录。
template: 模板文件路径，支持加载器，比如 html!./index.html
inject: true | 'head' | 'body' | false  ,注入所有的资源到特定的 template 或者 templateContent 中，如果设置为 true 或者 body，所有的 javascript 资源将被放置到 body 元素的底部，'head' 将放置到 head 元素中。
favicon: 添加特定的 favicon 路径到输出的 HTML 文件中。
minify: {} | false , 传递 html-minifier 选项给 minify 输出
hash: true | false, 如果为 true, 将添加一个唯一的 webpack 编译 hash 到所有包含的脚本和 CSS 文件，对于解除 cache 很有用。
cache: true | false，如果为 true, 这是默认值，仅仅在文件修改之后才会发布文件。
showErrors: true | false, 如果为 true, 这是默认值，错误信息会写入到 HTML 页面中
chunks: 允许只添加某些块 (比如，仅仅 unit test 块)
chunksSortMode: 允许控制块在添加到页面之前的排序方式，支持的值：'none' | 'default' | {function}-default:'auto'
excludeChunks: 允许跳过某些块，(比如，跳过单元测试的块)
```

案例：

```
const path = require("path");
const htmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
 ...
  plugins: [
```

```
    new htmlWebpackPlugin({
      title: "My App",
      filename: "app.html",
      template: "./src/index.html"
    })
  ]
};


//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><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

clean-webpack-plugin

```
npm install --save-dev clean-webpack-plugin
```

```
const { CleanWebpackPlugin } = require("clean-webpack-plugin");

...

plugins: [
    new CleanWebpackPlugin()
]
```

mini-css-extract-plugin

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");


{
  test: /\.css$/,
  use: [MiniCssExtractPlugin.loader, "css-loader"]
}



 new MiniCssExtractPlugin({
   filename: "[name].css"
 })
```

sourceMap

源代码与打包后的代码的映射关系

在dev模式中，默认开启，关闭的话 可以在配置文件里

```
devtool:"none"
```

devtool的介绍：https://webpack.js.org/configuration/devtool#devtool

eval:速度最快,使用eval包裹模块代码,

source-map： 产生`.map`文件

cheap:较快，不用管列的信息,也不包含loader的sourcemap

Module：第三方模块，包含loader的sourcemap（比如jsx to js ，babel的sourcemap）

inline： 将`.map`作为DataURI嵌入，不单独生成`.map`文件

配置推荐：

```
devtool:"cheap-module-eval-source-map",// 开发环境配置
devtool:"cheap-module-source-map",   // 线上生成配置
```

WebpackDevServer

提升开发效率的利器

每次改完代码都需要重新打包一次，打开浏览器，刷新一次，很麻烦

我们可以安装使用webpackdevserver来改善这块的体验

启动服务后，会发现dist目录没有了，这是因为devServer把打包后的模块不会放在dist目录下，而是放到内存中，从而提升速度

```
npm install webpack-dev-server -D
```

修改下package.json

```
"scripts": {
    "server": "webpack-dev-server"
  },
```

在webpack.config.js配置：

```
devServer: {
    contentBase: "./dist",
    open: true,
    port: 8081
  },
```

跨域：

联调期间，前后端分离，直接获取数据会跨域，上线后我们使用nginx转发，开发期间，webpack就可以搞定这件事

启动一个服务器，mock一个接口：

```
// npm i express -D
// 创建一个server.js 修改scripts "server":"node server.js"

//server.js
const express = require('express')

const app = express()

app.get('/api/info', (req,res)=>{
    res.json({
        name:'开课吧',
```

```
        age:5,
        msg:'欢迎来到开课吧学习前端高级课程'
    })
})

app.listen('9092')



//node server.js

http://localhost:9092/api/info
```

项目中安装axios工具

```
//npm i axios -D

//index.js
import axios from 'axios'
axios.get('http://localhost:9092/api/info').then(res=>{
    console.log(res)
})

会有跨域问题
```

修改webpack.config.js 设置服务器代理

```
proxy: {
      "/api": {
        target: "http://localhost:9092"
      }
    }
```

修改index.js

```
axios.get("/api/info").then(res => {
  console.log(res);
});
```

搞定!

Hot Module Replacement (HMR:热模块替换)

启动hmr

```
devServer: {
    contentBase: "./dist",
    open: true,
    hot:true,
    //即便HMR不生效，浏览器也不自动刷新，就开启hotOnly
    hotOnly:true
  },
```

配置文件头部引入webpack

```
//const path = require("path");
//const HtmlWebpackPlugin = require("html-webpack-plugin");
//const CleanWebpackPlugin = require("clean-webpack-plugin");

const webpack = require("webpack");
```

在插件配置处添加：

```
plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      template: "src/index.html"
    }),
    new webpack.HotModuleReplacementPlugin()
  ],
```

案例：

```
//index.js
import "./css/index.css";

var btn = document.createElement("button");
btn.innerHTML = "新增";
document.body.appendChild(btn);
```

```
btn.onclick = function() {
  var div = document.createElement("div");
  console.log("1");
  div.innerHTML = "item";
  document.body.appendChild(div);
};


//index.css
div:nth-of-type(odd) {
  background: yellow;
}
```

处理js模块HMR

需要使用module.hot.accept来观察模块更新 从而更新

案例：

```
//counter.js
function counter() {
  var div = document.createElement("div");
  div.setAttribute("id", "counter");
  div.innerHTML = 1;
  div.onclick = function() {
    div.innerHTML = parseInt(div.innerHTML, 10) + 1;
  };
  document.body.appendChild(div);
}
export default counter;


//number.js
function number() {
  var div = document.createElement("div");
  div.setAttribute("id", "number");
  div.innerHTML = 13000;
  document.body.appendChild(div);
}
export default number;
```

```
//index.js

import counter from "./counter";
import number from "./number";

counter();
number();

if (module.hot) {
  module.hot.accept("./b", function() {
    document.body.removeChild(document.getElementById("number"));
    number();
  });
}
```

Babel处理ES6

官方网站：https://babeljs.io/

中文网站：https://www.babeljs.cn/

```
npm i babel-loader @babel/core @babel/preset-env -D



//babel-loader是webpack 与 babel的通信桥梁，不会做把es6转成es5的工作，这部分工作需要用到@babel/preset-env来做

//@babel/preset-env里包含了es6转es5的转换规则
```

```
//index.js
const arr = [new Promise(() => {}), new Promise(() => {})];

arr.map(item => {
  console.log(item);
});
```

通过上面的几步 还不够，Promise等一些还有转换过来，这时候需要借助@babel/polyfill，把es的新特性都装进来，来弥补低版本浏览器中缺失的特性

@babel/polyfill

以全局变量的方式注入进来的。windows.Promise，它会造成全局对象的污染

```
npm install --save @babel/polyfill
```

Webpack.config.js

```
{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader",
  options: {
    presets: ["@babel/preset-env"]
  }
}
```

```
//index.js 顶部
import "@babel/polyfill";
```

会发现打包的体积大了很多，这是因为polyfill默认会把所有特性注入进来，假如我想我用到的es6+，才会注入，没用到的不注入，从而减少打包的体积，可不可以呢

当然可以

修改Webpack.config.js

```
options: {
          presets: [
            [
              "@babel/preset-env",
              {
                targets: {
                  edge: "17",
                  firefox: "60",
                  chrome: "67",
                  safari: "11.1"
                },
                useBuiltIns: "usage"//按需注入
```

```
            }
          ]
        ]
      }
```

当我们开发的是组件库，工具库这些场景的时候，polyfill就不适合了，因为polyfill是注入到全局变量，window下的，会污染全局环境，所以推荐闭包方式：@babel/plugin-transform-runtime

@babel/plugin-transform-runtime

它不会造成全局污染

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

npm install --save @babel/runtime
```

怎么使用?

先注释掉index.js里的polyfill

```
//import "@babel/polyfill";

const arr = [new Promise(() => {}), new Promise(() => {})];

arr.map(item => {
  console.log(item);
});
```

修改配置文件：注释掉之前的presets，添加plugins

```
options: {
  presets: [
    [
      "@babel/preset-env",
      {
        targets: {
```

```
          edge: "17",
          firefox: "60",
          chrome: "67",
          safari: "11.1"
        },
        useBuiltIns: "usage",
        corejs: 2
      }
    ]
  ],
"plugins": [
    [
      "@babel/plugin-transform-runtime",
      {
        "absoluteRuntime": false,
        "corejs": 2,
        "helpers": true,
        "regenerator": true,
        "useESModules": false
      }
    ]
  ]
}
```

`useBuiltIns`选项是`babel 7`的新功能，这个选项告诉`babel`如何配置`@babel/polyfill`。 它有三个参数可以使用： ①entry: 需要在`webpack`的入口文件里`import "@babel/polyfill"`一次。`babel`会根据你的使用情况导入垫片，没有使用的功能不会被导入相应的垫片。 ②usage: 不需要`import`，全自动检测，但是要安装`@babel/polyfill`。（试验阶段） ③false: 如果你`import "@babel/polyfill"`，它不会排除掉没有使用的垫片，程序体积会庞大。(不推荐)

请注意： usage 的行为类似 babel-transform-runtime，不会造成全局污染，因此也会不会对类似 Array.prototype.includes() 进行 polyfill。

扩展：

babelrc文件：

新建.babelrc文件，把options部分移入到该文件中，就可以了

```
//.babelrc
{
  "plugins": [
    [
```

```
      "@babel/plugin-transform-runtime",
      {
        "absoluteRuntime": false,
        "corejs": 2,
        "helpers": true,
        "regenerator": true,
        "useESModules": false
      }
    ]
  ]
}


//webpack.config.js

{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader"
}
```

配置React打包环境

安装

```
npm install react react-dom --save
```

编写react代码：

```
//index.js
import "@babel/polyfill";

import React, { Component } from "react";
import ReactDom from "react-dom";

class App extends Component {
  render() {
    return <div>hello world</div>;
  }
}

ReactDom.render(<App />, document.getElementById("app"));
```

安装babel与react转换的插件：

```
npm install --save-dev @babel/preset-react
```

在babelrc文件里添加：

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1",
          "Android":"6.0"
        },
        "useBuiltIns": "usage", //按需注入

      }
    ],
    "@babel/preset-react"
  ]
}
```

tree Shaking

webpack2.x开始支持 tree shaking概念，顾名思义，"摇树"，只支持ES module的引入方式！！！！，

```
//webpack.config.js


optimization: {
    usedExports: true
  }


//package.json
"sideEffects":false  正常对所有模块进行tree shaking  或者 "sideEffects":
['*.css','@babel/polyfill']
```

开发模式设置后，不会帮助我们把没有引用的代码去掉

案例：

```
//expo.js
export const add = (a, b) => {
  console.log(a + b);
};

export const minus = (a, b) => {
  console.log(a - b);
};


//index.js
import { add } from "./expo";
add(1, 2);
```

development vs Production模式区分打包

```
npm install webpack-merge -D
```

案例

```
const merge = require("webpack-merge")
const commonConfig =  require("./webpack.common.js")
const devConfig = {
  ...
```

```
}

module.exports = merge(commonConfig,devConfig)



//package.js
"scripts":{
  "dev":"webpack-dev-server --config ./build/webpack.dev.js",
  "build":"webpack --config ./build/webpack.prod.js"
}
```

案例2

基于环境变量

```
//外部传入的全局变量
module.exports = (env)=>{
  if(env && env.production){
    return merge(commonConfig,prodConfig)
  }else{
    return merge(commonConfig,devConfig)
  }
}


//外部传入变量
scripts:" --env.production"
```

代码分割 code Splitting

```
import _ from "lodash";

console.log(_.join(['a','b','c','****']))
```

假如我们引入一个第三方的工具库，体积为1mb，而我们的业务逻辑代码也有1mb，那么打包出来的体积大小会在2mb

导致问题：
体积大，加载时间长
业务逻辑会变化，第三方工具库不会，所以业务逻辑一变更，第三方工具库也要跟着变。

引入代码分割的概念：

```
//lodash.js

import _ from "lodash";

window._ = _;


//index.js 注释掉lodash引用
//import _ from "lodash";

console.log(_.join(['a','b','c','****']))


//webpack.config.js
entry: {
    lodash: "./lodash.js",
    index: "./index.js"
  },
  //指定打包后的资源位置
  output: {
    path: path.resolve(__dirname, "./build"),
    filename: "[name].js"
  }
```

其实code Splitting概念 与 webpack并没有直接的关系，只不过webpack中提供了一种更加方便的方法供我们实现代码分割

基于https://webpack.js.org/plugins/split-chunks-plugin/

```
optimization: {
    splitChunks: {
      chunks: 'async',//对同步，异步，所有的模块有效
      minSize: 30000,//当模块大于30kb
      maxSize: 0,//对模块进行二次分割时使用，不推荐使用
      minChunks: 1,//打包生成的chunk文件最少有几个chunk引用了这个模块
      maxAsyncRequests: 5,//模块请求5次
      maxInitialRequests: 3,//入口文件同步请求3次
      automaticNameDelimiter: '~',
      name: true,
      cacheGroups: {
        vendors: {
          test: /[\\/]node_modules[\\/]/,
          priority: -10//优先级 数字越大，优先级越高
        },
        default: {
          minChunks: 2,
          priority: -20,
          reuseExistingChunk: true
        }
      }
    }
  }
```

使用下面配置即可：

```
optimization:{
  //帮我们自动做代码分割
  splitChunks:{
    chunks:"all",//默认是支持异步，我们使用all
  }
}
```

打包分析

https://github.com/webpack/analyse

官方推荐工具

https://webpack.js.org/guides/code-splitting/#bundle-analysis

webpack 官方推荐的编码方式

```
optimization:{
  //帮我们自动做代码分割
  splitChunks:{
    chunks:"async",//默认是支持异步
  }
}
```

代码利用率的问题

```
//index.js

document.addEventListener("click", () => {
  const element = document.createElement("div");
  element.innerHTML = "welcome to webpack4.x";
  document.body.appendChild(element);
});



通过控制台看看代码利用率
```

把里面异步代码抽离出来

```
//index.js

document.addEventListener("click", () => {
  import("./click.js").then(({ default: func }) => {
  //需要用到 npm install --save-dev @babel/plugin-syntax-dynamic-import

    func();
  });
});
```

```
//click.js
function handleClick() {
  const element = document.createElement("div");
  element.innerHTML = "welcome to webpack4.x";
  document.body.appendChild(element);
}

export default handleClick;
```