link: null
title: 珠峰架构师成长计划
description: 指定extension之后可以不用在require或是import的时候加文件扩展名,会依次尝试添加扩展名进行匹配
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=68 sentences=70, words=484

1. 缩小范围

指定extension之后可以不用在 `require`或是 `import`的时候加文件扩展名,会依次尝试添加扩展名进行匹配

```
resolve: {
  extensions: [".js",".jsx",".json",".css"]
},
```

配置别名可以加快webpack查找模块的速度

- 每当引入bootstrap模块的时候，它会直接引入 `bootstrap`,而不需要从 `node_modules`文件夹中按模块的查找规则查找

```
const bootstrap = path.resolve(__dirname,'node_modules/bootstrap/dist/css/bootstrap.css')
resolve: {
+    alias:{
+        bootstrap
+    }
},
```

- 对于直接声明依赖名的模块（如 react ），webpack 会类似 Node.js 一样进行路径搜索，搜索 `node_modules`目录
- 这个目录就是使用 `resolve.modules`字段进行配置的 默认配置

```
resolve: {
modules: ['node_modules'],
}
```

如果可以确定项目内所有的第三方依赖模块都是在项目根目录下的 node_modules 中的话

```
resolve: {
modules: [path.resolve(__dirname, 'node_modules')],
}
```

默认情况下package.json 文件则按照文件中 main 字段的文件名来查找文件

```
resolve: {

  mainFields: ['browser', 'module', 'main'],

  mainFields: ["module", "main"],
}
```

当目录下没有 package.json 文件时，我们说会默认使用目录下的 index.js 这个文件，其实这个也是可以配置的

```
resolve: {
  mainFiles: ['index'],
},
```

`resolve.resolveLoader`用于配置解析 loader 时的 resolve 配置,默认的配置：

```
module.exports = {
  resolveLoader: {
    modules: [ 'node_modules' ],
    extensions: [ '.js', '.json' ],
    mainFields: [ 'loader', 'main' ]
  }
};
```

2. noParse

- `module.noParse` 字段，可以用于配置哪些模块文件的内容不需要进行解析
- 不需要解析依赖（即无依赖） 的第三方大型类库等，可以通过这个字段来配置，以提高整体的构建速度

```
module.exports = {

module: {
  noParse: /jquery|lodash/,

  noParse(content) {
    return /jquery|lodash/.test(content)
  },
}
}...
```

> 使用 noParse 进行忽略的模块文件中不能使用 import、require、define 等导入机制

3. IgnorePlugin

IgnorePlugin用于忽略某些特定的模块，让 webpack 不把这些指定的模块打包进去

```
import moment from  'moment';
import 'moment/locale/zh-cn'
console.log(moment().format('MMMM Do YYYY, h:mm:ss a'));
```

```
import moment from  'moment';
console.log(moment);
```

```
    new webpack.IgnorePlugin({

        contextRegExp: /moment$/,

        resourceRegExp: /^\.\/locale/
    new MiniCSSExtractPlugin({
        filename:'[name].css'
    })
```

- 第一个是匹配引入模块路径的正则表达式
- 第二个是匹配模块的对应上下文，即所在目录名

4.费时分析

```
const SpeedMeasureWebpackPlugin = require('speed-measure-webpack-plugin');
const smw = new SpeedMeasureWebpackPlugin();
module.exports =smw.wrap({
});
```

5.webpack-bundle-analyzer

- 是一个webpack的插件，需要配合webpack和webpack-cli一起使用。这个插件的功能是生成代码分析报告，帮助提升代码质量和网站性能

```
cnpm i webpack-bundle-analyzer -D
```

```
const {BundleAnalyzerPlugin} = require('webpack-bundle-analyzer');
module.exports={
  plugins: [
    new BundleAnalyzerPlugin()
  ]
}
```

6. libraryTarget 和 library

- outputlibrarytarget (https://webpack.js.org/configuration/output/#outputlibrarytarget)
- 当用 Webpack 去构建一个可以被其他模块导入使用的库时需要用到它们
- output.library 配置导出库的名称
- output.libraryExport 配置要导出的模块中哪些子模块需要被导出。 它只有在 output.libraryTarget 被设置成 commonjs 或者 commonjs2 时使用才有意义
- output.libraryTarget 配置以何种方式导出库,是字符串的枚举类型，支持以下配置

libraryTarget 使用者的引入方式 使用者提供给被使用者的模块的方式 var 只能以script标签的形式引入我们的库 只能以全局变量的形式提供这些被依赖的模块 commonjs 只能按照commonjs的规范引入我们的库 被依赖模块需要按照commonjs规范引入 commonjs2 只能按照commonjs2的规范引入我们的库 被依赖模块需要按照commonjs2规范引入 amd 只能按amd规范引入 被依赖的模块需要按照amd规范引入 this window global umd 可以用script、commonjs、amd引入 按对应的方式引入

编写的库将通过 var被赋值给通过 library指定名称的变量。

```
{
  output: {
        path: path.resolve("build"),
        filename: "[name].js",
+       library:'calculator',
+       libraryTarget:'var'
  }
}
```

```
module.exports =  {
    add(a,b) {
        return a+b;
    }
}
```

```
var calculator=(function (modules) {}({})
```

```
        let ret = calculator.add(1,2);
        console.log(ret);
```

- 编写的库将通过 CommonJS 规范导出。

```
exports["calculator"] = (function (modules) {}({})
```

```
let main = require('./main');
console.log(main.calculator.add(1,2));
```

```
require('npm-name')['calculator'].add(1,2);
```

> npm-name是指模块发布到 Npm 代码仓库时的名称

- 编写的库将通过 CommonJS 规范导出。**6.3.1 导出方式**

```
module.exports = (function (modules) {}({})
```

```
require('npm-name').add();
```

> 在 output.libraryTarget 为 commonjs2 时，配置 output.library 将没有意义。

- 编写的库将通过 this 被赋值给通过 library 指定的名称，输出和使用的代码如下：**6.4.1 导出方式**

```
this["calculator"]= (function (modules) {}({})
```

```
this.calculator.add();
```

- 编写的库将通过 window 被赋值给通过 library 指定的名称，即把库挂载到 window 上，输出和使用的代码如下：**6.5.1 导出方式**

```
window["calculator"]= (function (modules) {}({})
```

```
window.calculator.add();
```

- 编写的库将通过 global 被赋值给通过 library 指定的名称，即把库挂载到 global 上，输出和使用的代码如下：**6.6.1 导出方式**

```
global["calculator"]= (function (modules) {}({})
```

```
global.calculator.add();
```

```
(function webpackUniversalModuleDefinition(root, factory) {
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory();
  else if(typeof define === 'function' && define.amd)
    define([], factory);
  else if(typeof exports === 'object')
    exports['MyLibrary'] = factory();
  else
    root['MyLibrary'] = factory();
})(typeof self !== 'undefined' ? self : this, function() {
  return _entry_return_;
});
```