**前端工程引入Typescript规范化编码**

目录

[一、 背景 2](#_Toc75511828)

[二、 Webpack解决的问题 2](#_Toc75511829)

[三、 使用webpack模块化打包 2](#_Toc75511830)

[四、 Webpack使用loader实现特殊资源加载 2](#_Toc75511831)

[五、 Webpack使用plugin扩展功能 2](#_Toc75511832)

[六、 Webpack的整体架构 3](#_Toc75511833)

[七、 Webpack的工作流程 3](#_Toc75511834)

[八、 Webpack的核心概念 3](#_Toc75511835)

摘要

关键词

正文

# 背景

只有在了解Webpack的构建流程和原理后，我们才能去对其全链路进行性能优化。

# Webpack解决的问题

# 使用webpack模块化打包

# Webpack使用loader实现特殊资源加载

## Loader的概念

loader 用于对模块的源代码进行转换。loader 可以使你在 import 或"加载"模块时预处理文件。因此，loader 类似于其他构建工具中“任务(task)”，并提供了处理前端构建步骤的强大方法。loader 可以将文件从不同的语言（如 TypeScript）转换为 JavaScript，或将内联图像转换为 data URL。loader 甚至允许你直接在 JavaScript 模块中 import CSS文件！

## Loader特性

* loader 支持链式传递。能够对资源使用流水线(pipeline)。一组链式的 loader 将按照相反的顺序执行，从右向左或从下向上执行。loader 链中的第一个 loader 返回值给下一个 loader。在最后一个 loader，返回 webpack 所预期的 JavaScript。
* loader 可以是同步的，也可以是异步的。
* loader 运行在 Node.js 中，并且能够执行任何可能的操作。
* loader 接收查询参数。用于对 loader 传递配置。
* loader 也能够使用 options 对象进行配置。
* 除了使用 package.json 常见的 main 属性，还可以将普通的 npm 模块导出为 loader，做法是在 package.json 里定义一个 loader 字段。
* 插件(plugin)可以为 loader 带来更多特性。
* loader 能够产生额外的任意文件。

loader 通过（loader）预处理函数，为 JavaScript 生态系统提供了更多能力。 用户现在可以更加灵活地引入细粒度逻辑，例如压缩、打包、语言翻译。

## 使用loader

module: {

rules: [

{

test: /\.css$/,

use: [

{ loader: ['style-loader'](https://www.webpackjs.com/loaders/style-loader) },

{

loader: ['css-loader'](https://www.webpackjs.com/loaders/css-loader),

options: {

modules: true

}

}

]

}

]

}

# Webpack使用plugin扩展功能

## Plugin的概念

插件目的在于解决 [loader](https://www.webpackjs.com/concepts/loaders) 无法实现的**其他事**。

webpack **插件**是一个具有 [apply](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply) 属性的 JavaScript 对象。apply 属性会被 webpack compiler 调用，并且 compiler 对象可在**整个**编译生命周期访问。

## 使用plugin

由于**插件**可以携带参数/选项，你必须在 webpack 配置中，向 plugins 属性传入 new 实例。

const HtmlWebpackPlugin = require('html-webpack-plugin'); //通过 npm 安装

const webpack = require('webpack'); //访问内置的插件

const path = require('path');

const config = {

entry: './path/to/my/entry/file.js',

output: {

filename: 'my-first-webpack.bundle.js',

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

},

module: {

rules: [

{

test: /\.(js|jsx)$/,

use: 'babel-loader'

}

]

},

plugins: [

new webpack.optimize.UglifyJsPlugin(),

new HtmlWebpackPlugin({template: './src/index.html'})

]

};

module.exports = config;

## 常用的第三方plugin

除了压缩 JS 代码的 [uglifyjs-webpack-plugin](https://webpack.js.org/plugins/uglifyjs-webpack-plugin/" \t "_blank)，

常用的还有定义环境变量的 [DefinePlugin](https://webpack.js.org/plugins/define-plugin/" \t "_blank)，

生成 CSS 文件的 [ExtractTextWebpackPlugin](https://webpack.js.org/plugins/extract-text-webpack-plugin/" \t "_blank)

默认html模板的 html-webpack-plugin

# Webpack的整体架构

# Webpack的工作流程

Webpack的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

1. 初始化参数：从配置文件和shell语句中读取与合并参数，得出最终的参数。
2. 开始编译：用上一步得到的参数初始化compiler对象，加载所有配置的插件，通过执行对象的run方法开始执行编译。
3. 确定入口：根据配置中的entry找到所有入口文件。
4. 编译模块：从入口文件出发，调用所有配置的loader对模块进行编译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。
5. 完成模块编译：在经过第4步使用loader编译完所有模块后，得到了每个模块被编译后的最终内容以及它们之间的依赖关系。
6. 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的chunk，再将每一个chunk转换成一个单独的文件加入输出列表中，这是可以修改输出内容的最后机会。
7. 输出完成：再确定好输出内容后，根据配置确定输出的路径和文件名，将文件的内容写入文件系统中。

# Webpack的核心概念

1. Entry：入口，webpack打包之前从这个配置中获得所有文件
2. Module：模块，一个模块对应一个文件
3. Plugins：扩展插件，提供webpack更多功能
4. Output：输出，提供webpack输出出口，webpack会将多个chunk组合成的文件列表输出到对应的系统文件
5. Chunk：代码块，是由一个个代码模块组成，用于代码的合并和分割
6. Loader：代码编译工具，将原内容通过函数转化为新内容

总结