# 前端工程化

## A. 模块化相关规范

### 模块化概述

#### 传统开发模式的主要问题

- 命名冲突
  指的是多个 JS 文件之间如果存在命名相同的变量，那么就会面临变量覆盖的问题。
- 文件依赖
  指的是 JS 文件之间无法实现相互的引用。

#### 通过模块化解决上述问题

- **模块化**就是把单独的一个功能封装到一个模块（文件）中，模块之间相互隔离，但是可以通过特定的接口公开内部成员，也可以依赖别的模块。
- 模块化开发的好处：方便代码的重用，从而提升开发效率，并且方便后期的维护。

### 浏览器端模块化规范

#### AMD (outdate)

- Require.js (http://www.requirejs.cn/)
- Sea.js (https://seajs.github.io/seajs/docs)

### 服务器端的模块化规范

#### CommonJS

1. 模块分为单文件模块和包
2. 模块成员导出：module.exports 和 exports
3. 模块成员导入：require('模块标识符')

#### 大一统的模块化规范 - ES6 模块化

在 ES6 模块化规范诞生之前，Javascript 社区已经尝试并提出了 AMD/CMD/CommonJS 等模块化规范。
但是，这些社区提出的模块化标准，还是存在一定的**差异性**与**局限性**，并不是浏览器与服务器**通用的模块化标准**，例如：

- AMD 和 CMD 适用于浏览器端的 Javascript 模块化

- CommonJS 适用于服务器端的 Javascript 模块化

因此，ES6 语法规范中，在语言层面上定义了 ES6 模块化规范，是浏览器端与服务器端通用的模块化开发规范。

- 每一个 js 文件都是一个独立的模块
- ***导入模块成员*使用import**关键字
- ***暴露模块成员*使用export**关键字

1. Node.js 中通过**babel**体验 ES6 模块化

- babel：将由兼容性的高级的 JS 代码转换为没有兼容性的低级的 JS 代码

  > npm install --save-dev @babel/core @babel/cli @babel/preset-env @babel/node
  > npm install --save @babel/polyfill
  > 项目根目录创建文件 babel.config.js
  > babel.config.js 文件内容如下：

```
const presets = [
    ["@babel/env",{
            targets:{
                edge:"17",
                firefox: "60",
                chrome: "67",
                safari: "11.1"
        }
    }]
];
module.exports = {presets};
```

- > 通过 npx babel-node index.js 执行代码

## ES6 模块化的基本语法

### 默认导出与默认导入

- 默认导出语法 **export default** 默认导出成员

```
// 当前文件模块为 modoule.js
// 定义私有成员 a 和 c
let a = 10;
let c = 20;
// 外界访问不到变量 d， 因为没有将它暴露出去
let d = 30;
function show() {
    return a + c + d;
}
// 将本模块中的私有成员暴露出去，供其它模块使用
export default {
        a,
        c,
        show,
};
```

- 默认导入语法 **import** 接收名称 **from** ‘模块标识符’

```
import m1 from './module.js';
console.log(m1)
// 打印输出结果为：
// {a:10,c:20,show:[Function:show]}
```

注意：每个模块中，只允许使用唯一的一次 export default，否则会报错

### 按需导出与按需导入

- 按需导出语法 **export** let s1 = 10

```
//当前文件模块为module.js
//向外按需导出变量 s1
export let s1 = 'aaa'
//向外按需导出变量 s2
export let s2 = 'ccc'
//向外按需导出变量 s3
export function say = function(){}
```

- 按需导入语法 **import** {s1} **from** ‘模块标识符’

```
//导入模块成员
import {s1, s2 as ss2, say} from './module.js'
console.log(s1) // 输出 aaa
console.log(ss2) // 输出 ccc
console.log(say) // 输出 [Function:say]
```

注意，在每个模块中可以使用多次按需导出

### 直接导入并执行模块代码

有时候，我们只想单纯的执行某个模块中的代码，并不需要得到模块中向外暴露的成员，此时，可以直接导入并执行模块代码。

```
//在 m2.js 文件中 只有一个循环
for(let i = 0; i < 3; i++){
console.log(i)
}

// 在index.js中 直接执行
import './m2.js'
```

# B. Webpack

## 当前 Web 开发中面临的困境

- 文件依赖关系错综复杂
- 静态资源请求效率低
- 模块化支持不友好
- 浏览器对高级 Javascript 特性兼容程度低
- etc...

## Webpack 概述

Webpack是一个流行的前端项目构建工具（打包工具），可以解决当前 web 开发中所面临的困境。Webpack提供了友好的模块化支持，以及代码压缩混淆，处理 JS 兼容性问题，性能优化等强大功能，从而让程序员把工作的重心放到具体的功能实现上，提高了开发效率和项目的可维护性。

## Webpack 的基本使用

### 1. 创建列表隔行变色的项目 (直接使用会存在浏览器对 ES6 代码兼容性的问题)

① 新建项目空白目录，并运行 npm init -y 命令，初始化包管理配置文件 package.json
② 新建 src 源代码目录
③ 新建 src -> index.html 首页和 src -> index.js 脚本文件
④ 初始化首页基本的结构
⑤ 运行 npm install jquery -S 命令，安装 jQuery
⑥ 通过 ES6 模块化的方式导入 jQuery，实现列表隔行变色效果

快捷创建多个标签：ul>li{This is $ li}*9

### 2. 在项目中安装和配置 Webpack，以解决兼容性问题

① 运行 npm install webpack webpack-cli -D 命令，安装 webpack 相关的包。// npm install webpack@4.29
-S：是--save的简写
-D：是--save-dev的简写
② 在项目根目录中，创建名为 webpack.config.js 的 webpack 配置文件
③ 在 webpack 的配置文件中，初始化如下基本配置：

```
module.exports = {
    mode: 'development' //mode 用来指定构建模式
}
```

④ 在package.json 配置文件中的scripts 节点下，新增dev 脚本如下：

```
"script":{
    "dev": "webpack" // script 节点下的脚本，可以通过npm run执行
}
e.g., npm run dev
```

### 3. 配置打包的入口与出口

在 webpack 4.x 和 5.x 的版本中，有如下的默认约定：

① 默认的打包入口文件为 src -> index.js
② 默认的输出文件路径为 dist -> main.js

注意：可以在 webpack.config.js 中修改打包的默认约定

```
const path = require('path') // 导入node.js中专门操作路径的模块
module.exports = {
    entry:path.join(__dirname,'./src/index.js'), //打包入口文件的路径
    output:{
        path:path.join(__dirname,'./dist'), //输出文件存放的目录路径
        filename:'bundle.js' //输出文件的名称
    }
}
```

### 4. 配置 Webpack 的自动打包功能

① 运行 npm install webpack-dev-server -D 命令，安装支持项目自动打包的工具
② 修改 package.json -> scripts 中的 dev 命令如下：

```
"script":{
    "dev": "webpack-dev-server" // script 节点下的脚本，可以通过npm run执行
}
```

③ 将 src -> index.html 中，script 脚本的引用路径，修改为 "/buldle.js"
④ 运行 npm run dev 命令，重新打包
⑤ 在浏览器中访问 http://localhost:8080 地址，查看自动打包效果

**Notice:**

- webpack-dev-server 会启动一个实时打包的 http 服务器， 当改变代码并保存后，它会自动编译代码
- webpack-dev-server 打包生成的输出文件是默认放到了项目的根目录中，而且是虚拟的，看不见

### 5. 配置 *html-webpack-plugin* 生成预览页面

```
① 运行 npm install html-webpack-plugin -D 命令，安装生成预览页面的插件
② 修改 webpack.config.js 文件头部区域，添加如下配置信息：
    // 导入生成预览页面的插件，得到一个构造函数
    const HtmlWebpackPlugin = require('html-webpack-plugin')
    const htmlPlugin = new HtmlWebpackPlugin({ // 创建插件的实例对象
        template: './src/index.html',          // 指定要用到的模版文件
        filename: 'index.html'                // 指定生成的文件的名称，该文件存在于内存中，在目录中不显示
    })
③ 修改 webpack.config.js 文件中向外暴露的配置对象，新增如下配置节点：
    module.exports = {
        plugins: [ htmlPlugin ] // plugins 数组是 webpack 打包期间会用到的一些插件列表
    }
```

### 6. 配置自动打包相关的参数

```
// package.json 中的配置
// --open 打包完成后自动打开浏览器页面
// --host 配置IP地址
// --port 配置端口
"script":{
        "dev": "webpack-dev-server --open --host 127.0.0.1 --port 3000" // script 节点下的脚本，可
    }
```

## Webpack 中的加载器

### 1. 通过 loader 打包非 js 模块

在实际开发过程中，webpack 默认只能打包处理以 .js 后缀名结尾的模块，其他非 .js 后缀名结尾的模块， webpack 默认处理不了，需要调用 loader 加载器才可以正常打包，否则会报错！

loader 加载器可以协助 webpack 打包处理特定的文件模块，比如：

- less-loader 可以打包处理 .less 相关的文件
- sass-loader 可以打包处理 .sass 相关的文件
- url-loader 可以打包处理 css 中与 url 路径相关的文件

## Webpack 中加载器的基本使用

### 1. 打包处理 css 文件

① 运行 npm i style-loader css-loader -D 命令，安装处理 css 文件的 loader
② 在webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
// 所有第三方文件模块的匹配规则
module: {
    rules: [
        {test:/\.css$/, use:['style-loader','css-loader']}
    ]
}
```

其中，$代表以css结尾的文件，test 表示匹配的文件类型，use 表示对应要调用的 loader

**Notice:**

- use 数组中指定的 loader 顺序是固定的
- 多个 loader 的调用顺序是：从后往前调用

### 2. 打包处理 less 文件

① 运行 npm i less-loader less -D 命令
② 在webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
// 所有第三方文件模块的匹配规则
module: {
    rules: [
        {test:/\.less$/, use:['style-loader','css-loader','less-loader']}
    ]
}
```

其中，$代表以less结尾的文件，test 表示匹配的文件类型，use 表示对应要调用的 loader

### 3. 打包处理 sass 文件

① 运行 npm i sass-loader node-sass -D 命令
② 在webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
// 所有第三方文件模块的匹配规则
module: {
    rules: [
        {test:/\.scss$/, use:['style-loader','css-loader','sass-loader']}
    ]
}
```

其中，$代表以scss结尾的文件，test 表示匹配的文件类型，use 表示对应要调用的 loader

### 4. 配置 postCSS 自动添加 css 的兼容前缀（对 css 中的伪元素进行浏览器兼容性的配置）

① 运行 npm i postcss-loader autoprefixer -D 命令
② 在项目根目录中创建 postcss 的配置文件 postcss.config.js，并初始化如下配置：

```
const autoprefixer = require('autoprefixer') //导入自动添加前缀的插件
module.exports = {
    plugins: [autoprefixer] // 挂载插件
}
```

③ 在webpack.config.js 的module -> rules 数组中，修改 css 的loader规则如下：

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

### 5. 打包样式表中的图片和字体文件

- webpack 4.x, 貌似在 webpack 5.x 内能够自动处理 css 文件中的 url 问题了, 但不能处理 html 中的问题

① 运行 npm i url-loader file-loader -D 命令
② 在webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
module: {
    rules: [
        {
            test:/\.jpg|png|gif|bmp|ttf|eot|svg|woff|woff2$/,
            use: 'url-loader?limit = 16940'
        }
    ]
}
```

其中 ？ 之后的是 loader 的参数项。
limit 用来指定图片的大小，单位是字节（byte），只有小于 limit 大小的图片，才会被转为 base64 图片

这种 url-loader 的配置只能用于处理 webpack 4.x 中 css 文件下 background url()中的图片路径
在 webpack 5.x 中，这种配置就完全不适用了， 因为在 webpack 5.x 中，css-loader 完全能够处理 background url()图片格式的问题了，如果再加上这个配置，那么 css 中的图像不仅会被 css-loader 处理，同时还会被 url-loader 处理，因此导致图片无法正常显示。 url-loader 主要是处理图片或者其他文件信息通过 url 的形式被加载到js 文件中的一种处理方式，与 file-loader 有异曲同工之妙，它的优势在于能够对文件加 limit，从而减少请求文件的次数。

- file-loader 是 url-loader 的内置项
- 因此在 webpack 5.x 中，解决 css-loader 与 url-loader 对 css 文件下 处理 background url() 冲突的问题，我们做如下配置：

```
module: {
            rules: [
                {
                    test: /\.css$/i,
                    use: ["style-loader", "css-loader"],
                },
                {
                    test: /\.(png|jpe?g|gif|svg|eot|ttf|woff|woff2)$/i,
                    // More information here https://webpack.js.org/guides/asset-modules/
                    type: "asset",
                },
            ]
        }
```

See: https://github.com/webpack-contrib/css-loader#recommend

该配置下，能够通过 css-loader 同时来处理，css 文件和 js 文件中的 url 图片资源

#### 6. 打包处理 js 文件中的高级语法 （webpack 4.x）

① 安装 babel 转换器相关的包： npm i babel-loader @babel/core @babel/runtime -D

② 安装 babel 语法插件相关的包：npm i @babel/preset-env @babel/plugin-transform-runtime @babel/plugin-p

③ 在项目根目录中，创建babel配置文件 babel.config.js 并初始化基本配置如下：

```
module.exports = {
    presets: ['@babel/preset-env'],
    plugins: ['@babel/plugin-transform-runtime', '@babel/plugin-proposal-class-properties']
}
```

④ 在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
// exclude 为排除项，表示 babel-loader不需要处理 node_modules 中的js文件
{test:/\.js$/, use:'babel-loader',exclude:/node_modules/}
```

# C. Vue 单文件组件

## 传统组件的问题和解决方案

- 问题
  - 1. 全局定义的组件必须保证组件的名称不重复
  - 2. 字符串模版缺乏语法高亮，在 HTML 有多行的时候，需要用到丑陋的\
  - 3. 不支持 CSS 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏
  - 4. 没有构建步骤限制，只能使用 HTML 和 ES5 JavaScript， 而不能使用预处理器（如：babel）
- 解决方案

  针对传统组件的问题，Vue 提供了一个解决方案 ---- 使用 Vue 单文件组件。

## Vue 单文件组件的基本用法

## 单文件组件的组成结构

- template 组件的模版区域
- script 业务逻辑区域
- style085372

```
<template>
    <!-- 这里用于定义Vue组件的模版内容 -->
</template>
<script>
    // 这里用于定义Vue组件的业务逻辑
    export default {
        data(){return{}}, // 私有数据
        methods:{} //处理函数
        // ...其他业务逻辑
    }
</script>
<style scoped>
    /* 这里用于定义组件的样式 */
</style>
```

## Webpack 中配置 vue 组件加载器

① 运行 npm i vue-loader vue-template-compiler -D 命令， 后面的这个包是vue-loader的内置依赖项
② 在 webpack.config.js 配置文件中，添加 vue-loader 的配置项如下：

```
const VueLoaderPlugin = require('vue-loader')
module.exports = {
    module:{
        rules: [
            // ... 其他规则
            {test:/\.vue$/, loader: 'vue-loader'}
        ]
    },
    plugins:[
        // ... 其他插件
        new VueLoaderPlugin() // 请确保引入这个插件
    ]
}
```

有问题！！！！！！

## 在 webpack 项目中使用 vue

① 运行 npm i vue -S 安装 vue
② 在 src -> index.js 入口文件中，通过 import Vue from 'vue' 来导入vue构造函数
③ 创建 vue 的实例对象，并指定要控制的 el 区域
④ 通过 render 函数渲染 App 根组件

```
// 1. 导入Vue构造函数
import Vue from 'vue'
// 2. 导入 App 根组件
import App from './components/App.vue'

const vm = new Vue({
    // 3. 指定vm实例要控制的页面区域
    el: '#app',
    // 4. 通过 render 函数，把指定的组件渲染到 el 区域中
    render: h => h(App)
})
```

### webpack 打包发布

上线之前需要通过 webpack 将应用进行整体打包，可以通过 package.json 文件配置打包命令：

```
// 在package.json文件中配置webpack打包命令
// 该命令默认加载项目根目录中的 webpack.config.js配置文件
"scripts":{
    //用于打包的命令
    "build": "webpack",
    //用于开发调试的命令
    "dev": "webpack-dev-server --open --host 127.0.0.1 --port 3000",
}
```

# 发布到 NPM

## 在 webpack 中的配置

- 配置 terserwebpackplugin:

① 运行 npm install terser-webpack-plugin --save-dev
② 在 webpack.config.js 配置文件中，更改如下配置：

```
const TerserPlugin = require("terser-webpack-plugin"); // 引入压缩插件
 entry: {
    "abi": "./src/index.js",
    "abi.min": "./src/index.js",
},
output: {
    filename: "[name].js",
    library: "abi",
    libraryExport: "default", // 不添加的话引用的时候需要 abi.default
    libraryTarget: "umd", // var this window ...
},
optimization: {
    minimize: true,
    minimizer: [
    new TerserPlugin({
        // 使用压缩插件
        include: /\.min\.js$/,
    }),
    ],
},
```

③ 在根目录文件中，创建一个index.js 文件， 用来选择暴露模块库 (可选)：

```
if (process.env.NODE_ENV === "production") {
    // 通过环境变量来决定入口文件
    module.exports = require("./dist/abi.min.js");
} else {
    module.exports = require("./dist/abi.js");
}
```

④ 在package.json中，做如下配置（非常重要）：

```
{
    "name": "webpack4-demo",
    "version": "1.0.0",
    "description": "webpack 模块化相关规范",
    "main": "/dist/abi.js", // 无敌巨重要，必须要指明文件的入口
}
```

- 使用 terserwebpackplugin 的优势：
  与默认的 webpack 打包器（uglifyjs-webpack-plugin）相比，我们可以通过使用 terserwebpackplugin 在打包的过程中，过滤到 JS 代码中的 console.log 语句和 debugger 等。
  设置一些过滤项配置：

```
const TerserPlugin = require('terser-webpack-plugin');
module.exports = {
    configureWebpack:{
        optimization:{
            minimizer:[
                new TerserPlugin({
                    terserOptions:{
                        compress:{
                            warnings:false,
                            drop_console:true,
                            drop_debugger:true,
                            pure_funcs:["console.log"]
                        }
                    }
                })
            ]
        }
    }
}
```