

## 07 Webpack 中的模块化开发

更新时间：2019-06-24 09:25:17



“要成就一件大事业，必须从小事做起。

——列宁”

模块是指为了完成某功能所需的程序或者子程序，模块是系统中「职责单一」且「可替换」的部分。所谓的模块化就是指把系统代码分为一系列职责单一且可替换的模块。模块化开发是指如何开发新的模块和复用已有的模块来实现应用的功能。Webpack 作为 JavaScript 模块化打包工具，自然对 JavaScript 的模块化做了不少工作，本文将从模块规范说起，逐渐介绍 Webpack 中对模块化的一些增强处理。

我们先来认识下三大 JavaScript 主流模块规范：CommonJS、AMD 和 ES6 Module。

### CommonJS

CommonJS 规范是 2009 年一名来自 Mozilla 团队的工程师 Kevin Dangoor 开始设计一个叫 ServerJS 的项目提出来的，随着 Node.js 的广泛应用，被广泛接受。通过 **ServerJS** 这个名字就可以知道，CommonJS 主要是服务端用的模块规范。

```
// sayhi.js
var hi = 'hello world';
function sayHi() {
  return hi;
}
module.exports = sayHi;
// index.js
var sayHi = require('./sayhi.js');
console.log(sayHi());
```

上面的代码就是 CommonJS 语法，使用了 `require` 来导入一个模块，`module.exports` 导出模块。在 Node.js 中实际代码会被处理成下面代码而被应用：

```
(function(exports, require, module, __filename, __dirname) {  
    // ...  
    // 模块的代码在这里  
    // ...  
});
```

## CommonJS 的问题

CommonJS 规范是 JavaScript 中最常见的模块格式规范，这一标准的设计初衷是为了让 JavaScript 在多个环境下都实现模块化。起先主要应用在 Node.js 服务端中，但是 Node.js 中的实现依赖了 Node.js 本身功能的实现，包括了 Node.js 的文件系统等，这个规范在浏览器环境是没法使用的。后来随着 Browserify 和 Webpack 等打包工具的崛起，通过处理的 CommonJS 前端代码也可以在浏览器中使用。

## AMD

AMD 规范是在 CommonJS 规范之后推出的一个解决 web 页面动态异步加载 JavaScript 的规范，相对于 CommonJS 规范，它最大特点是浏览器内支持、实现简单、并且支持异步加载，AMD 规范最早是随着 RequireJS 发展而提出来的，它最核心的是两个方法：

- `require()`：引入其他模块；
- `define()`：定义新的模块。

基本语法如下：

```
// sayhi.js  
define(function() {  
    var hi = 'hello world';  
    return function sayHi() {  
        return hi;  
    };  
});  
// index.js  
require(['./sayhi.js'], function(sayHi) {  
    console.log(sayHi());  
});
```

AMD 提出来之后，也有很多变种的规范提出来，比如国内 Sea.js 的 CMD，还有兼容 CommonJS 和 AMD 的 UMD 规范（Universal Module Definition）。虽然 AMD 的模式很适合浏览器端的开发，但是随着 npm 包管理的机制越来越流行，这种方式可能会逐步的被淘汰掉。

### AMD 规范的问题

在 AMD 规范中，我们要声明一个模块，那么需要指定该模块用到的所有依赖项，这些依赖项会被当做形参传到 `factory`（`define` 方法传入的函数叫做 `factory`）中，对于依赖的模块会提前执行，这种做法叫做 **依赖前置**。依赖前置加大了开发的难度，无论我们在阅读代码还是编写代码的时，都会导致引入的模块内容是条件性执行的。

而且不管 AMD 还是 CommonJS 都没有统一浏览器和客户端的模块化规范。

## ES6 Module

ES6 Module，又称为 ES2015 modules，是 ES2015 标准提出的一种模块加载方式，也是 ECMAScript 官方提出的方案，作为 ES 标准，不仅仅在 Web 现代浏览器（例如 Chrome）上面得到实现，而且在 Node.js 9+ 版本也得到原生支持（需要加上 `--experimental-modules` 使用）。

```
// sayhi.js
const hi = 'hello world';
export default function sayHi() {
  return hi;
}
// index.js
import sayHi from './sayhi';
console.log(sayHi());
```

对于前端项目，可以通过 Babel 或者 Typescript 进行提前体验。

Tips: 随着主流浏览器逐步开始支持 ES Modules (ESM) 标准，越来越多的目光投注于 Node.js 对于 ESM 的支持实现上；目前 Node.js 使用 CommonJS 作为官方的模块解决方案，虽然内置的模块方案促进了 Node.js 的流行，但是也为引入新的 ES Modules 造成了一定的阻碍。不过 Node.js 9.0+ 已经支持 ESM 语法，需要添加 flag `-experimental-modules` 来启动 ESM 语法支持，文件则必须使用 `.mjs` 后缀：`node --experimental-modules some-esm-file.mjs`。

## Webpack 中一切皆模块

在 Web 前端，我们不仅仅只有 JavaScript，还有 CSS、HTML、图片、字体、富媒体等众多资源，还有一些资源是以类似「方言」的方式存在着，比如 less、sass、各种 js 模板库等，这些资源并不能被直接用在 JavaScript 中，如果在 JavaScript 中像使用模块一样使用，那么可以极大的提高我们的开发体验：

```
var img = require('./img/webpack.png');
var style = require('./css/style.css');
var template = require('./template.ejs');
```

这时候，我们就需要 Webpack 了！在 Webpack 中，一切皆模块！

在 Webpack 编译的过程中，Webpack 会要对整个代码进行静态分析，分析出各个模块的类型和它们依赖关系，然后将不同类型的模块提交给对应的加载器 (loader) 来处理。比如一个用 Less 写的样式，可以先用 less-loader 将它转成一个 CSS 模块，然后再通过 css-loader 把他插入到页面的 `<style>` 标签中执行，甚至还可以通过插件将这部分 CSS 导出为 CSS 文件，使用 `link` 标签引入到页面中。

## Webpack 对 Module 的增强

在 Webpack 中，我们不仅可以为所欲为的使用 CommonJS、AMD 和 ES6 Module 三大规范（比如一个文件中混合使用三种规范），还可以使用 Webpack 对 Module 的增强方法和属性。下面介绍下 Webpack 中特有的一些属性和方法。

### `import()` 和神奇注释

在 Webpack 中，`import` 不仅仅是 ES6 Module 的模块导入方式，还是一个类似 `require` 的函数（其实这是 ES2015 loader 规范的实现），我们可以通过 `import('path/to/module')` 的方式引入一个模块，`import()` 返回的是一个 `Promise` 对象：

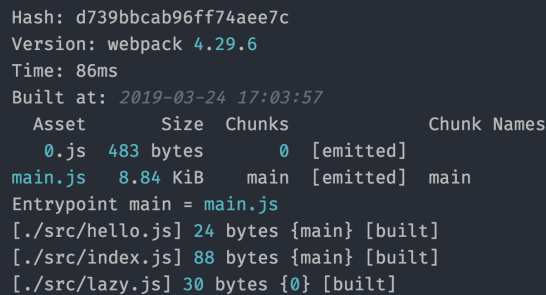
```
import('path/to/module').then(mod => {
  console.log(mod);
});
```

下面看看使用 `import()` 和 `import` 的打包有什么区别：

```
// hello.js
export default 'hello';
// lazy.js
export default 'lazy module';
// index.js
import hello from './hello';
import('./lazy').then(lazy => {
  console.log(lazy);
});
```

执行下命令：

```
npx webpack --mode development:
```



```
Hash: d739bbcab96ff74aee7c
Version: webpack 4.29.6
Time: 86ms
Built at: 2019-03-24 17:03:57
    Asset      Size  Chunks             Chunk Names
    0.js    483 bytes          0 [emitted]
  main.js   8.84 KiB        main [emitted]  main
Entrypoint main = main.js
./src/hello.js 24 bytes {main} [built]
./src/index.js 88 bytes {main} [built]
./src/lazy.js  30 bytes {0} [built]
```

通过打包后的 `log` 和 `dist` 文件夹内容发现，我们的代码被分割成了两个文件，一个是 `main.js` 一个是 `0.js`，这是因为相对于 `import from` 的静态分析打包语法，`import()` 是动态打包语法，即我们的内容不是第一时间打进 `main.js` 的，而是通过异步的方式加载进来的。代码分割是 `webpack` 进行代码结构组织，实现动态优化的一个重要功能

**Tips:** 与 `import()` 用法一样的是 `require.ensure` 的方法，这个方法已经被 `import()` 方式替换掉；针对 `import t()` 打包产物跟普通的静态分析打包的实现不同之处，后面原理篇讲解打包产出物的时候会详细介绍。

下面我们再来看下 `import()` 的神奇注释特性，上面 `index.js` 的代码修改成下面这样，增加注释 `webpackChunkName: 'lazy-name'`

```
import hello from './hello';
import(
  /*
    webpackChunkName: 'lazy-name'
  */
  './lazy'
).then(lazy => {
  console.log(lazy);
});
```

则打包后的结果，`0.js` 变成了 `lazy-name.js` 了，这个文件的名称就是在 `import()` 注释里面指定的 `webpackChunkName`，这就是神奇注释（Magic Comments）。

```

Hash: fa4f8df27a0b816fb792
Version: webpack 4.29.6
Time: 81ms
Built at: 2019-03-24 17:05:27
    Asset      Size      Chunks             Chunk Names
  lazy-name.js  493 bytes    lazy-name  [emitted]  lazy-name
    main.js  8.89 KiB         main  [emitted]  main
Entrypoint main = main.js
[./src/hello.js] 24 bytes {main} [built]
[./src/index.js] 143 bytes {main} [built]
[./src/lazy.js]  30 bytes {lazy-name} [built]

```

目前支持的注释有：

- `webpackInclude`：如果是 `import` 的一个目录，则可以指定需要引入的文件特性，例如只加载 json 文件：`/\.json$/`；
- `webpackExclude`：如果是 `import` 的一个目录，则可以指定需要过滤的文件，例如 `/\.noimport\.json$/`；
- `webpackChunkName`：这是 chunk 文件的名称，例如 `lazy-name`；
- `webpackPrefetch`：是否预取模块，及其优先级，可选值 `true`、或者整数优先级别，0 相当于 `true`，webpack 4.6+ 支持；
- `webpackPreload`：是否预加载模块，及其优先级，可选值 `true`、或者整数优先级别，0 相当于 `true`，webpack 4.6+ 支持；
- `webpackMode`：可选值 `lazy` / `lazy-once` / `eager` / `weak`。

这里最复杂的是 `webpackMode`：

- `lazy`：是默认的模式，为每个 `import()` 导入的模块，生成一个可延迟加载 chunk；
- `lazy-once`：生成一个可以满足所有 `import()` 调用的单个可延迟加载 chunk，此 chunk 将在第一次 `import()` 调用时获取，随后的 `import()` 调用将使用相同的网络响应；注意，这种模式仅在部分动态语句中有意义，例如 `import('./locales/${language}.json')`，其中可能含有多个被请求的模块路径；
- `eager`：不会生成额外的 chunk，所有模块都被当前 chunk 引入，并且没有额外的网络请求。仍然会返回 Promise，但是是 `resolved` 状态。和静态导入相对比，在调用 `import()` 完成之前，该模块不会被执行。
- `weak`：尝试加载模块，如果该模块函数已经以其他方式加载（即，另一个 chunk 导入过此模块，或包含模块的脚本被加载）。仍然会返回 Promise，但是只有在客户端上已经有该 chunk 时才成功解析。如果该模块不可用，Promise 将会是 `rejected` 状态，并且网络请求永远不会执行。当需要的 chunks 始终在（嵌入在页面中的）初始请求中手动提供，而不是在应用程序导航在最初没有提供的模块导入的情况触发，这对于 Server 端渲染（SSR，Server-Side Render）是非常有用的。

通过上面的神奇注释，`import()` 不再是简单的 JavaScript 异步加载器，还是任意模块资源的加载器，举例说明：如果我们页面用到的图片都放在 `src/assets/img` 文件夹下，你们可以通过下面方式将用到的图片打包到一起：

```
import(/* webpackChunkName: "image", webpackInclude: /\.png|jpg|gif/ */ './assets/img');
```

Tips: `prefetch` 优先级低于 `preload`，`preload` 会并行或者加载完主文件之后立即加载；`prefetch` 则会在主文件之后、空闲时在加载。`prefetch` 和 `preload` 可以用于提前加载图片、样式等资源的功能。

`require.resolve()` 和



`require.resolveWeak()`

`require.resolve()` 和 `require.resolveWeak()` 都可以获取模块的唯一 ID (`moduleId`)，区别在于 `require.resolve()` 会把模块真实引入进 bundle，而 `require.resolveWeak()` 则不会，配合 `require.cache` 和 `__webpack_module_s__` 可以用于判断模块是否加载成功或者是否可用。

`require.context()`

`require.context(directory, includeSubdirs, filter)` 可以批量将 `directory` 内的文件全部引入进文件，并且返回一个具有 `resolve` 的 `context` 对象，使用 `context.resolve(moduleId)` 则返回对应的模块。

- `directory`: 目录 string;
- `includeSubdirs`: 是否包含子目录，可选，默认值是 `true`;
- `filter`: 过滤正则规则，可选项。

Tips: 注意 `require.context()` 会将所有的文件都引入进 bundle!

`require.include()`

`require.include(dependency)` 顾名思义为引入某个依赖，但是并不执行它，可以用于优化 chunk，例如下面示例代码：

```
require.include('./hello.js');
require.ensure(['./hello.js', './weak.js'], function(require) {
  /* ... */
});
require.ensure(['./hello.js', './lazy.js'], function(require) {
  /* ... */
});
```

上面代码打包之后结果：

```
Hash: 3078370c7b0b43440d91
Version: webpack 4.29.6
Time: 83ms
Built at: 2019-03-23 11:04:52
   Asset      Size  Chunks             Chunk Names
   0.js    483 bytes          0  [emitted]
   1.js    501 bytes          1  [emitted]
  main.js   8.84 KiB         main  [emitted]  main
Entrypoint main = main.js
./src/hello.js 24 bytes {main} [built]
./src/index.js 187 bytes {main} [built]
./src/lazy.js  30 bytes {0} [built]
./src/weak.js  47 bytes {1} [built]
```

- `main` 包含了 `hello` 和 `index`;
- `weak` 和 `lazy` 分别被打包到 `1`, `0` 两个文件。

这实际上使用了 `require.include()` 直接优化了代码分割，如果不用 `require.include('./hello.js');` 则 `hello.js` 会分别和 `weak`、`lazy` 打包，注意下面打包 log 的 `./src/hello.js 24 bytes {0} {1} [built]` 说明 `hello.js` 被打包进了 `0`, `1` 两个文件。

```
Hash: 8839bee89fe987fc3614
Version: webpack 4.29.6
Time: 101ms
Built at: 2019-03-23 11:10:04
    Asset      Size  Chunks             Chunk Names
    0.js    893 bytes          0  [emitted]
    1.js    911 bytes          1  [emitted]
  main.js   8.4 KiB        main  [emitted]  main
Entrypoint main = main.js
[./src/hello.js] 24 bytes {0} {1} [built]
[./src/index.js] 156 bytes {main} [built]
[./src/lazy.js]  30 bytes {0} [built]
[./src/weak.js]  47 bytes {1} [built]
```

## \_\_resourceQuery

当前模块的资源查询（resource query），即当前模块引入是传入的 query 信息，例如：

```
// main.js
const component = require('./component-loader?component=demo');
// component-loader.js
const querystring = require('querystring');
const query = querystring.parse(__resourceQuery.slice(1)); // 去掉?
console.log(query); // {component: demo}
```

其他

- **webpack\_public\_path**: 等同于 `output.publicPath` 配置选项；
- **webpack\_require**: 原始 `require` 函数。这个表达式不会被解析器解析为依赖。
- **webpack\_chunk\_load**: 内部 `chunk` 载入函数，用法 `__webpack_chunk_load__(chunkId, callback(require))`；
  - `chunkId`：需要载入的 chunk id
  - `callback(require)`：chunk 载入后调用的回调函数。
- **webpack\_modules**: 所有模块的内部对象，可以通过传入 `moduleId` 来获取对应的模块；`require.resolve()` 和 `require.resolveWeak()` 获取 `moduleId`；
- `module.hot`: 用于判断是否在 `hotModuleReplace` 模式下，一般可以用于 loader 编写中判断在 HMR 模式下增加 reload 逻辑代码等；
- **webpack\_hash**: 这个变量只有在启用 `HotModuleReplacementPlugin` 或者 `ExtendedAPIPlugin` 时才生效。这个变量提供对编译过程中(compilation)的 hash 信息的获取。
- **non\_webpack\_require**: 生成一个不会被 webpack 解析的 `require` 函数

## Webpack 对 Node.js 模块的 polyfill

Webpack 还对一些常用的 Node.js 模块和属性进行了 mock，例如在 web 的 js 文件中可以直接引入 Node.js 的 `querystring` 模块，这个模块实际引入的是 `node-libs-browser` 来对 Node.js 核心库 polyfill，详细在 web 页面中可以用到的 Node.js 模块，可以参考 `node-libs-browser` Readme 文件的表格。

polyfill: 英文原意为一种用于衣物、床具等的聚酯填充材料，例如装修时候的腻子，作用是抹平坑坑洼洼的墙面；在 JavaScript 中表示一些可以抹平浏览器实现差异的代码，比如某浏览器不支持 Promise，可以引入 `es6-promise-polyfill` 等库来解决。

# Webpack 对资源的模块化处理

下面在讲解下 Webpack 对其他资源的模块化处理方案。

样式文件的 `@import` 和

JavaScript 中的 `import`

在 Webpack 中的 css （包括其预处理语言，例如 Less、Sass）等，都可以在内部通过 `@import` 语法直接引入使用：

```
@import 'vars.less';
body {
  background: @bg-color;
}
```

除了样式文件中的 `@import`，在 JavaScript 文件中，还支持直接使用 ES6 Module 的 `import`（包括 `require`）直接引入样式文件，例如：

```
import styles from './style.css';
```

JavaScript 的这种语法其实是 [CSS Modules 语法](#)，目前浏览器支持程度有限，但是在 Webpack 中，我们可以通过配置 loader 优先使用这种方式！

Tips: 后面讲解 CSS 样式配置的时候会更加详细的讲解 CSS Modules。

使用 loader 把资源作为模块引入

在 Webpack 中，除了 CSS 可以直接使用 `import` 语法引入，类似 HTML 和页面模板等，可以直接使用对应的 loader，通过下面的语法来引入：

```
const html = require('html-loader!./loader.html');
console.log(html);
```

上面的代码得到 `html` 变量实际为引入的 `loader.html` 的 string 片段。除了 `html`，类似模板文件，还可以直接转换为对应的 render 函数，例如下面代码使用了 `ejs-loader`：

```
const render = require('ejs!./file.ejs');
// => 得到 ejs 编译后的 render 函数
render(data); // 传入 data，直接返回的是 html 片段
```

Tips: 原理篇将动手手写一个 `markdown-loader`，更深入的了解其功能和原理实现。

小结



本小节从 JavaScript 的模块化发展史讲起，逐渐介绍了应用到服务端的 CommonJS 规范、浏览端的 AMD 规范和 ES6 Modules 规范。在 Webpack 中一切皆模块，任何资源都可以被当做模块引入进来，不仅仅是 JavaScript 模块和 Node.js 的模块，甚至 CSS、Less、JavaScript 模板、图片等任何资源，只需要配合对应的 loader 就可以实现资源的引入。本小节介绍的按需加载和神奇注释，在日常项目优化中经常使用。

本小节 Webpack 相关面试题：

1. 什么是 JavaScript 的模块化开发？有哪些可以遵循的规范？
2. 在 js 文件中怎么调用 loader 来处理一个模块？
3. Webpack 中怎么获取一个模块引用另外一个模块是传入的 query？
4. 怎么实现 Webpack 的按需加载？什么是神奇注释？

专栏代码已经整理好给大家共享出来：

[点此下载源码 源码](#)



06 基础概念和常见配置项介绍  
(二)

08 在 Webpack 中使用 Babel  
转换 JavaScript 代码

