第01课：前端模块化指引与实践



对于 JavaScript 开发者来说，现在每天都在和无数的模块打交道。也许你还没有注意到，从 npm 上刚安装几个包，项目中就已经有成千上万个模块了。了解模块是什么，以及它们是如何在项目中工作的，对于现在的 JavaScript 开发者变得从未有过的重要。

这里会介绍 JavaScript 模块的标准以及相关的工具。如果你对 AMD、CommonJS、ES6 Module 这些概念还有点模糊，这篇文章会帮助你理清。后半部分会介绍 Webpack 的打包机制，希望能把 Webpack 是如何处理依赖以及合并模块这件事情讲清楚，这是我们后面讲打包优化的基础。

### 什么是模块？

我们可以把工程比作我们居住的城市，而模块就是这个城市内部的各个职能系统——学校、医院、消防局等等。它们各司其职，负责着不同的功能，共同保证着城市的正常运转。

在软件工程设计中，我们通常将具有特定功能的代码封装为模块。比如在一个 Web 应用里，可能会有专门负责处理网络请求的模块、专门负责日期处理的模块、专门负责渲染的模块等等，每个模块各自发挥着不同的作用，都是系统不可缺少的一部分。

具体到代码层面，模块则是一个较为笼统和宽泛的概念，实现的形式并不一样。在一些情况下，一个文件可以包含多个模块，也可以与模块一一对应。比如在工程中也许会有一个 error.js 专门用于错误处理，或者通过 util.js 来放各种工具函数，由每个文件负责一个单一的功能。

### 为什么要使用模块？

模块化的设计可以为系统带来很多好处，最重要的有以下几个：

**1.作用域封装**：在 JavaScript 中代码执行的顶层作用域即是全局作用域，这意味着变量和函数定义很容易冲突。而使用模块将代码封装起来，可保证内部实现不会暴露在全局作用域中，我们只需将模块的功能通过接口的方式暴露出去给其它模块调用即可，避免了污染全局命名空间的问题。

**2.重用性**：在工程中经常会出现重复的部分，比如一个 Web 应用中各个页面共同的 header、footer，最原始的开发方式是将同样的代码复制粘贴到各个地方。这种做法的缺点是当这些共同的部分发生改变时，我们需要逐一改动每个地方的代码。

而如果把实现一类功能的代码封装为模块之后，就可以提供给各个调用者。比如说应用中有一个 Dialog 组件，可以把它的结构和样式封装起来，在不同的页面中调用它。假如 Dialog 的样式需要调整，那么只要调整该模块那么相当于此修改对所有页面都生效。

**3.解除耦合**：试想一下如果有一个几千行代码的文件在你的工程里，内部实现了各种各样的功能并且互相调用，这样的代码调试起来有多痛苦。将系统分解为模块的一个很重要意义就是解除各部分之间的耦合。当系统的某个部分需要发生改变的时候，通过模块我们可以快速定位问题。由于模块把功能的具体实现封装在了内部，只要模块间的接口不变，模块内部的变化对于外面的其它部分并没有感知。因此通过模块化可以提升系统的可维护性。

**4.按需加载**：如果没有模块，所有的代码将被放在一个大文件里面统统塞给用户。当页面不断地增加功能，不断地添加代码，最终的文件只会越来越大，而页面也打开地越来越慢，对于用户来说非常不友好。使用模块化来拆分逻辑可以使页面需要的资源最先被加载，而后续的模块在恰当的时机再进行异步加载，从而让页面加载速度更快，用户也得到更好的体验。

### JavaScript 模块发展简史

让我们先来回顾一下 JavaScript 模块的历史是怎样演化的。

#### script 标签

最原始的模块实现方式是通过在页面中插入多个 script 标签。

<**script** src="//mycdn.com/moduleA.js"></**script**>

<**script** src="//mycdn.com/moduleB.js"></**script**>

<**script** src="//mycdn.com/main.js"></**script**>

这种方法的缺陷是所有这些脚本都是共用全局的作用域。在任何一个 JavaScript 文件中进行顶层作用域的变量或函数声明，都会暴露在全局中，使得其它脚本也相应获取。当应用的规模和复杂度上升，这些脚本之间很容易发生命名冲突，从而导致不可预知的问题。

#### 立即执行函数表达式

后来随着 Web 的发展，人们发现通过立即执行函数表达式（ IIFE ）可以解决命名空间的问题。IIFE 的实现原理是把代码包裹在一个函数中，并且在声明这个函数之后立即执行它，这样相当于为代码单独创建了一个作用域。比如在下面的代码中我们创建了一个立即执行函数表达式，变量的声明处于它自己的函数作用域内，与其它的模块作用域隔离开。

(**function**() {

// foobar 并不会暴露在全局作用域

**var** foobar = 'Hello IIFE!';

**console**.log(foobar);

})()

如果通过这种方法封装的模块需要与别的模块发生交互，则可以将特定的对象绑定在全局来允许其它模块通过全局对象获取，比如下面的形式：

// calculator.js

(**function**() {

// 将 calculator 绑定在全局对象上，使其它模块调用

**window**.calculator = {

add: **function**() {

//...

},

sub: **function**() {

//...

},

//...

}

})()

// app.js

calculator.add(1, 2);

立即执行函数表达式的缺点在于，如果模块之间有着依赖关系，必须将它们按照特定的顺序引入到页面中。比如上面的 calculator，必须使定义 calculator 的 JavaScript 先执行，再执行调用的模块。由于这种依赖关系是隐式的，当所有这些模块都互相依赖时，文件的引入顺序将变得难以维护。

#### AMD

AMD 是 Asynchronous Module Definition（异步模块定义）的缩写。下面的代码使用 AMD 规范定义了一个模块：

// 定义一个求和的模块

define('getSum', ['math'], **function**(math) {

**return** **function**(a, b) {

**console**.log('sum: ' + math.sum(a, b));

}

});

在 AMD 中定义模块是使用 define 函数，它可以接受三个参数。第一个参数是当前模块的 ID，相当于给这个模块起一个名字；第二个参数是当前模块的依赖，比如上面我们定义的 getSum 模块需要 math 模块的依赖；第三个参数可以是函数或者对象。如果是函数，可以利用函数的返回值将定义的模块接口导出；如果是对象，则代表它为当前模块的导出值。

通过这种形式定义模块的好处在于，它 **显式** 地表达出了每个模块所依赖的其它模块。并且模块定义也不再绑定到全局对象上，不必担心其在别的地方被篡改。

#### CommonJS 与 Node.js 模块系统

CommonJS 是于 2009 年提出的 JavaScript 规范，它最开始是为了定义服务端标准，而非用于浏览器环境。之后 Node.js 采用并实现了它的部分规范，在模块系统上进行了一些调整。一般来说，我们不会严格区分 CommonJS 与 Node.js 的模块标准，详述两种标准的区别超出了这里的范围，在下面会直接使用 CommonJS 来进行表述。

Browserify 的出现带来了浏览器环境模块的变革。它是一个运行在 Node 环境下的模块打包工具，可以把模块按照 Node.js 的模块规则合并为浏览器支持的形式，这使得浏览器端的框架类库也可以按照 CommonJS 的形式编写。随着 Node.js 以及 npm 流行，近两年来对于开发者来说遵循 CommonJS 标准来编写和使用模块已经成为了一个基本通识。

在 CommonJS 中每个文件是一个模块，并且拥有属于自己的作用域和上下文。模块的依赖通过 require 函数来引入。

**const** math = **require**('./math');

如果想把模块的接口暴露给外部，则要通过 exports 将其导出，如：

exports.getSum = **function**(a, b) {

**return** a + b;

}

AMD 和 CommonJS 具有同样的特性——模块的依赖必须显式引入，这样就解决了之前维护复杂模块引入时的顺序问题。

#### ES6 Module

ES6 Module 是目前比较推荐开发者使用的模块标准。之所以在过去我们有各种不同的模块化标准是因为 JavaScript 这门语言本身不具备模块化的特性，而现在 ES6 中已经具备了。ES6 Module 的模块语法和 CommonJS 很像，它通过 import 和 export 来进行模块的导入和导出。

**import** math **from** './math';

**export** **function** **sum**(a, b) {

**return** a + b;

}

在 ES6 Module 中也是每个文件作为一个模块。和 CommonJS 不同的是，ES6 Module 的模块的依赖是静态的，或者说是在编译时确定的，而不是运行时确定的。

举个例子，我们可以在 CommonJS 中的 if 语句中 require 模块，根据代码运行时 if 的判断条件决定是否要引入该模块。

// 根据运行时条件确定是否引入

**if**(**Date**.now() > **new** **Date**('2019-01-01')) {

**require**('./my\_module');

}

而在 ES6 Module 中则不允许这样做，import 必须在代码的顶层作用域，这意味着你不能把它放在 if 等代码块中。ES6 Module 这样规定的原因在于可以使编译器在编译阶段就可以获取到整个依赖树，从而进行代码静态分析层面的优化，比如检测出哪些模块是从来没有被使用过的，然后从打包结果中优化掉等等。

### 动态加载模块

有些场景下我们希望能够动态地去加载一些模块，在 CommonJS 中可以直接使用 require 实现。

**if**(condition) {

**require**('moduleA');

} **else** {

**require**('moduleB');

}

但是在 ES6 Module 中，由于上面我们提到的 import 是在编译时被处理而非运行时，因此无法实现动态加载的特性。

// 报错

**if**(condition) {

**require**('moduleA');

}

// 报错

**var** foo = 'foo';

**var** bar = 'bar';

**import** foobar **from** (foo + bar);

为了解决这个问题，tc39 提出了一个 import() 函数提案。它可以接受一个参数，指定所加载的模块，并且返回一个 Promise 对象。

**var** foo = 'foo';

**var** bar = 'bar';

**import**(foo + bar).then(module => {

**console**.log('foobar loaded:', **module**);

}).catch(err => {

**console**.log(err);

});

目前 Webpack 从 2.0.0 版本开始已经支持该动态加载形式，后面会更加详细地进行介绍。

### 模块打包原理简述

现在假设我们依照 ES6 Module 的规范写了一些模块，问题是如何将它们作用于我们的应用呢？现在最主流的解决方案是使用 Webpack 来处理它们。Webpack 以及其它的一些打包工具最基本的功能就是按照我们定义好的依赖树将模块合并成单一的文件，让浏览器能够按照预想的依赖顺序去执行。这个过程我们通常将它叫做模块打包。

下面我们以 Webpack 打包 ES6 Module 为例看看具体过程是如何实现的。

首先让我们创建两个文件，app.js 和 module.js：

// app.js

**import** moduleLog **from** './module.js';

**document**.write('app.js loaded.');

moduleLog();

// module.js

**export** **default** **function**() {

**document**.write('module.js loaded.');

}

然后使用 Webpack 进行打包：

# Webpack 版本需要大于等于 2，这里使用的版本是 3.5.5

webpack app.js dist/bundle.js

app.js 是我们的打包入口文件，dist/bundle.js 是最终的打包合并结果文件。Webpack 会在打包的过程中从入口 app.js 开始查找所有依赖的模块，并最终包装和合并这些模块放在 bundle.js 中。接下来让我们分析一下打包结果 dist/bundle.js 的大体结构：

(**function**(modules) {

**var** installedModules = {};

**function** **\_\_webpack\_require\_\_**(moduleId) {

/\* code \*/

}

// ...

**return** \_\_webpack\_require\_\_(0); // entry file

})([ /\* modules array \*/ ]);

这段代码总体上来看是一个立即执行函数表达式，它只有一个参数 modules，即 Webpack 从入口文件开始检索到的所有依赖模块。每一个依赖模块会被 Webpack 进行一次包装，放到 modules array 的数组中等待代码运行时调用。

上面立即执行的匿名函数体内分为几个部分。首先定义了一个 installedModules 对象用来放置已经加载过的模块。接着定义了 \_\_webpack\_require\_\_，这个函数是 Webpack 模块加载的核心，可以认为它是浏览器环境下的 require。在函数体最后使用 \_\_webpack\_require\_\_ 加载了工程的入口模块，在浏览器中执行时即会从入口开始去逐步执行后面的模块。

让我们看一下 \_\_webpack\_require\_\_ 的具体实现：

**function** **\_\_webpack\_require\_\_**(moduleId) {

// Check if module is in cache

**if** (installedModules[moduleId]) {

**return** installedModules[moduleId].exports;

}

// Create a new module (and put it into the cache)

**var** **module** = installedModules[moduleId] = {

i: moduleId,

l: false,

exports: {}

};

// Execute the module function

modules[moduleId].call(

**module**.exports,

**module**, **module**.exports,

\_\_webpack\_require\_\_

);

// Flag the module as loaded

**module**.l = true;

// Return the exports of the module

**return** **module**.exports;

}

通过上面代码可以看出 \_\_webpack\_require\_\_ 主要做了几个事情：

1. 检查该模块是否已经加载过，如果是则直接返回已加载过的
2. 添加 moduleId 等属性，并把该模块放进 installedModules
3. 将模块 this 指到它的 module.exports，并执行已经包装好的模块逻辑
4. 返回 modules.exports

在 Webpack 包装模块代码的时候，会把 \_\_webpack\_require\_\_ 作为参数传进去。在实际的模块代码中，导入其它模块的语句都会被替换为这种浏览器可以执行的形式，通过这种方式使模块间有了互相调用的能力。

纵观整个 bundle.js，Webpack 主要在打包中处理了下面这些问题：

1. 从入口文件开始分析整个应用的依赖树
2. 将每个依赖模块包装起来，并放到一个数组中等待调用
3. 实现模块加载的方法，并提供到模块执行的环境中，使得模块间可以互相调用
4. 将执行入口文件的逻辑放在一个立即执行函数表达式中

当浏览器执行这个 bundle.js 时，首先会执行入口文件的逻辑，接着会利用 Webpack 提供好的模块以及模块的加载方法来根据依赖关系一步步执行整个应用。以上就是一个简单的 Webpack 处理打包的过程。

### ES6 Module 与浏览器

现代的浏览器已经开始逐渐支持 JavaScript 模块语法，包括 Chrome、Safari、Edge 以及 Firefox（后两者需要开启实验特性标识），这意味着我们也可以不使用 Webpack 等打包工具，而直接采用 ES6 Module 的语法编写我们的模块。下面的示例可以直接运行在支持 ES6 Module 的浏览器上：

<**script** type="module" src="app.js"></**script**>

// main.js

**import** {log} **from** "./module.js";

log();

// module.js

**export** **const** log = **function**() {

**console**.log('module.js loaded.');

}

不过这种方式目前还存在两个问题：

1. 对于不支持 ES6 Module 的浏览器如何向下兼容。对于同一个页面来说要兼容两种浏览器就需要插入两种脚本，打包过的和未打包过的，目前来看这个问题不可避免。
2. 仍然需要由工具来完成一些其它的构建任务，比如代码预编译、tree-shaking、代码压缩等等，这些还没有办法直接在浏览器层面解决。

随着浏览器对于 JavaScript 模块化支持的成熟，这些问题也会有相应地解决方案，相信在未来几年 JavaScript 模块化的实现会给开发者带来更多的改变。