webpack 原理与实战

JS 中的模块化

要明白我们的打包工具究竟做了什么，首先必须明白的一点就是 JS 中的模块化，在 ES6 规范之前，我们有 CommonJS 、AMD 等主流的模块化规范。

CommonJS

Node.js 就是一个基于 V8 引擎，事件驱动 I/O 的服务端 JS 运行环境，在 2009 年刚推出时，它就实现了一套名为 **CommonJS** 的模块化规范。

在 CommonJS 规范里，每个 JS 文件就是一个 **模块 (module)** ，每个模块内部可以使用 `require` 函数和 `module.exports` 对象来对模块进行导入和导出。

```
// index.js
require("./moduleA");
var m = require("./moduleB");
console.log(m);

// moduleA.js
var m = require("./moduleB");
setTimeout(() => console.log(m), 1000);

// moduleB.js
var m = new Date().getTime();
module.exports = m;
```

- **index.js** 代表的模块通过执行 `require` 函数，分别加载了相对路径为 `./moduleA` 和 `./moduleB` 的两个模块，同时输出 **moduleB** 模块的结果。
- **moduleA.js** 文件内也通过 `require` 函数加载了 **moduleB.js** 模块，在 1s 后也输出了加载进来的结果。
- **moduleB.js** 文件内部相对来说就简单的多，仅仅定义了一个时间戳，然后直接通过 `module.exports` 导出。

AMD

另一个为 WEB 开发者所熟知的 JS 运行环境就是浏览器了。浏览器并没有提供像 Node.js 里一样的 `require` 方法。不过，受到 CommonJS 模块化规范的启发，WEB 端还是逐渐发展起来了 AMD，SystemJS 规范等适合浏览器端运行的 JS 模块化开发规范。

AMD 全称 **Asynchronous module definition**，意为`异步的模块定义`，不同于 CommonJS 规范的同步加载，AMD 正如其名所有模块默认都是异步加载，这也是早期为了满足 web 开发的需要，因为如果在 web 端也使用同步加载，那么页面在解析脚本文件的过程中可能使页面暂停响应。

```
// index.js
require(['moduleA', 'moduleB'],
function(moduleA, moduleB) {
    console.log(moduleB);
});

// moduleA.js
define(function(require) {
    var m = require('moduleB');
    setTimeout(() => console.log(m), 1000);
});

// moduleB.js
```

```
define(function(require) {
    var m = new Date().getTime();
    return m;
});
```

如果想要使用 AMD 规范，我们还需要添加一个符合 AMD 规范的加载器脚本在页面中，符合 AMD 规范实现的库很多，比较有名的就是 **require.js**。

ESModule

前面我们说到的 CommonJS 规范和 AMD 规范有这么几个特点：

1. 语言上层的运行环境中实现的模块化规范，模块化规范由环境自己定义。
2. 相互之间不能共用模块。例如不能在 Node.js 运行 AMD 模块，不能直接在浏览器运行 CommonJS 模块。

在 EcmaScript 2015 也就是我们常说的 ES6 之后，JS 有了语言层面的模块化导入导出关键词与语法以及与之匹配的 ESModule 规范。使用 ESModule 规范，我们可以通过 `import` 和 `export` 两个关键词来对模块进行导入与导出。

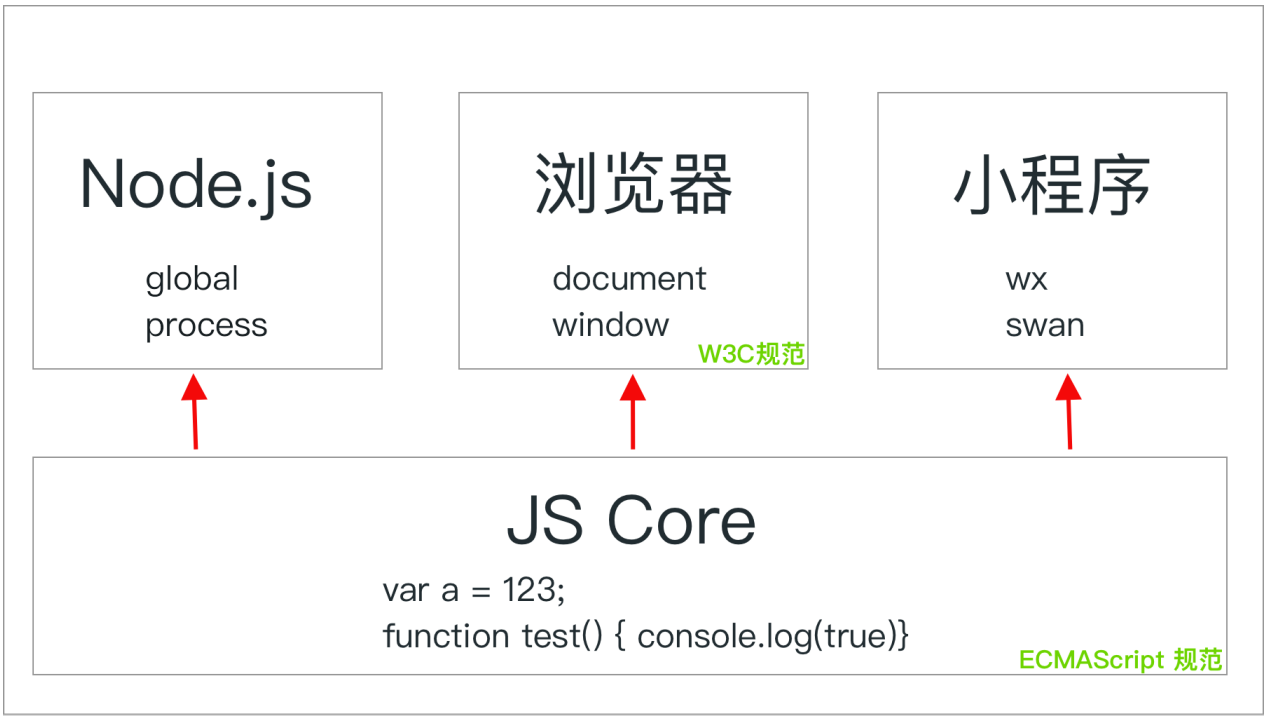
还是之前的例子，使用 ESModule 规范和新的关键词就需要这样定义：

```
// index.js
import './moduleA';
import m from './moduleB';
console.log(m);

// moduleA.js
import m from './moduleB';
setTimeout(()) => console.log(m), 1000);

// moduleB.js
var m = new Date().getTime();
export default m;
```

每个 JS 的运行环境都有一个解析器，否则这个环境也不会认识 JS 语法。它的作用就是用 ECMAScript 的规范去解释 JS 语法，也就是处理和执行语言本身的内容，例如按照逻辑正确执行 `var a = "123";`，`function func() {console.log("hahaha");}` 之类的内容。

在解析器的上层，每个运行环境都会在解释器的基础上封装一些环境相关的 API。例如 Node.js 中的 `global` 对象、`process` 对象，浏览器中的 `window` 对象，`document` 对象等等。这些运行环境的 API 受到各

自规范的影响，例如浏览器端的 W3C 规范，它们规定了 `window` 对象和 `document` 对象上的 API 内容，以使得我们能让 `document.getElementById` 这样的 API 在所有浏览器上运行正常。



ESModule 就属于 JS Core 层面的规范，而 AMD，CommonJS 是运行环境的规范。所以，想要使运行环境支持 ESModule 其实是比较简单的，只需要升级自己环境中的 JS Core 解释引擎到足够的版本，引擎层面就能认识这种语法，从而不认为这是个 **语法错误(syntax error)**，运行环境中只需要做一些兼容工作即可。

Node.js 在 V12 版本之后才可以使用 ESModule 规范的模块，在 V12 没进入 LTS 之前，我们需要加上 `--experimental-modules` 的 flag 才能使用这样的特性，也就是通过 `node --experimental-modules`

`index.js` 来执行。浏览器端 Chrome 61 之后的版本可以开启支持 ESModule 的选项，只需要通过 ``` 这样的标签加载即可。

这也就是说，如果想在 Node.js 环境中使用 ESModule，就需要升级 Node.js 到高版本，这相对来说比较容易，毕竟服务端 Node.js 版本控制在开发人员自己手中。但浏览器端具有分布式的特点，是否能使用这种高版本特性取决于用户访问时的版本，而且这种解释器语法层面的内容无法像 AMD 那样在运行时进行兼容，所以想要直接使用就会比较麻烦。

后模块化的编译时代

通过前面的分析我们可以看出来，使用 ESModule 的模块明显更符合 JS 开发的历史进程，因为任何一个支持 JS 的环境，随着对应解释器的升级，最终一定会支持 ESModule 的标准。但是，WEB 端受制于用户使用的浏览器版本，我们并不能随心所欲的随时使用 JS 的最新特性。为了能让我们的新代码也运行在用户的老浏览器中，社区涌现出了越来越多的工具，它们能静态将高版本规范的代码编译为低版本规范的代码，最为大家所熟知的就是 `babel`。

它把 JS Core 中高版本规范的语法，也能按照相同语义在静态阶段转化为低版本规范的语法，这样即使是早期的浏览器，它们内置的 JS 解释器也能看懂。

然后，不幸的是，对于模块化相关的 `import` 和 `export` 关键字，`babel` 最终会将它编译为包含 `require` 和 `exports` 的 CommonJS 规范。

这就造成了另一个问题，这样带有模块化关键词的模块，编译之后还是没办法直接运行在浏览器中，因为浏览器端并不能运行 CommonJS 的模块。为了能在 WEB 端直接使用 CommonJS 规范的模块，除了编译之外，我们还需要一个步骤叫做**打包(bundle)**。

打包工具的作用，就是将模块化内部实现的细节抹平，无论是 AMD 还是 CommonJS 模块化规范的模块，经过打包处理之后能变成能直接运行在 WEB 或 Node.js 的内容。

如何处理打包

参考 Node.js 源码来熟悉 CommonJS 的处理方式

我们可以参考 CommonJS 模块的处理方式来处理一个 CommonJS 模块。在 node.js 中，所有的 CommonJS 模块都会被包裹在一个函数中，然后在 node.js 中使用 vm 来运行它，最终达到一个模块化导入和导出的目的。

浏览器中对 CommonJS 处理

我们在浏览器中也可以按照相同的思路来进行处理，我们在打包阶段将每个模块包裹上一层函数字符串，然后放置到浏览器中去执行它。

同时我们实现一个简单版本的 require 函数和 module 对象，来处理运行时加载的问题。

这样一个基本的流程就做好了，接下来我们要处理的就是运行时的依赖关系，我们需要运行时明白模块之间的依赖关系，所以我们需要自己维护一个配置，运行时来进行查找。

有了查找关系之后，我们就可以在运行时注入这部分内容，获取内容的时候通过这部分配置来拿到模块之间的映射关系。

异步组件打包

上面我们说到的都是同步组件的打包，最终所有的组件都同步的打包进同一个文件当中，但有的时候我们需要将组件进行异步加载，异步加载的过程中，我们的组件需要不在主包当中，而在其他的子包文件当中。

这时候，我们就需要使用另外的异步策略来处理，我们这里采用 jsonp 的原理来执行。

HMR 原理

明白了同步打包和异步打包之后，我们的工具基本上就已经覆盖了大部分功能，那我们如何进行 hot module reload 呢？这里简单给大家讲解一下核心的原理。

hot module reload 就是当我们对文件内容有改动的时候，就会重新触发编译，同时也会重新触发 UI 的更新，达到了我们无须重新更新打包就能更新我们的应用。

我们只需要清除我们函数内部的缓存和模块的代码，这样不刷新页面的情况下，只需要重新加载组件就能达到效果。

面向切面的插件设计

我们可以面向整个流程的切面来实现编译的效果，例如引入 babel 之后，在读取了文件之后对其中的内容进行编译，达到引入 ES5 文件的效果。