JS 模块化

孔德建（憬浩）

为什么要模块化？

- 模块化在项目中十分的重要，一个复杂的项目肯定有很多相似的功能模块，如果每次都需要重新编写模块肯定既费时又耗力。
- 模块化的开发方式可以提高代码复用率，方便进行代码的管理。
- 增加团队协作能力。

script标签

- 这是最原始的 JavaScript 文件加载方式，如果把每一个文件看做是一个模块，那么他们的接口通常是暴露在全局作用域下，也就是定义在 window 对象中，不同模块的接口调用都是一个作用域中，一些复杂的框架，会使用命名空间的概念来组织这些模块的接口。
- 缺点:
  1、污染全局作用域
  2、开发人员必须主观解决模块和代码库的依赖关系
  3、文件只能按照script标签的书写顺序进行加载
  4、在大型项目中各种资源难以管理，长期积累的问题导致代码库混乱不堪

CommonJS规范

- 该规范的核心思想是允许模块通过require方法来同步加载所要依赖的其他模块，然后通过exports或module.exports来导出需要暴露的接口。

```
require("module");
require("../file.js");
exports.doStuff = function() {};
module.exports = someValue;
```

- 优点：
  1、简单并容易使用
  2、服务器端模块便于重用
- 缺点:
  1、同步的模块加载方式不适合在浏览器环境中，同步意味着阻塞加载，浏览器资源是异步加载的
  2、不能非阻塞的并行加载多个模块

AMD规范

- 由于浏览器端的模块不能采用同步的方式加载，会影响后续模块的加载执行，因此AMD(Asynchronous Module Definition异步模块定义)规范诞生了。
- AMD标准中定义了以下两个API
  1、require([module], callback);
  2、define(id, [depends], callback);

  require接口用来加载一系列模块，define接口用来定义并暴露一个模块。

```
define("module", ["dep1", "dep2"], function(d1, d2) { return someExportedValue; });
require(["module", "../file"], function(module, file) { /* ... */ });
```

AMD规范

- 优点：
  1、适合在浏览器环境中异步加载模块
  2、可以并行加载多个模块

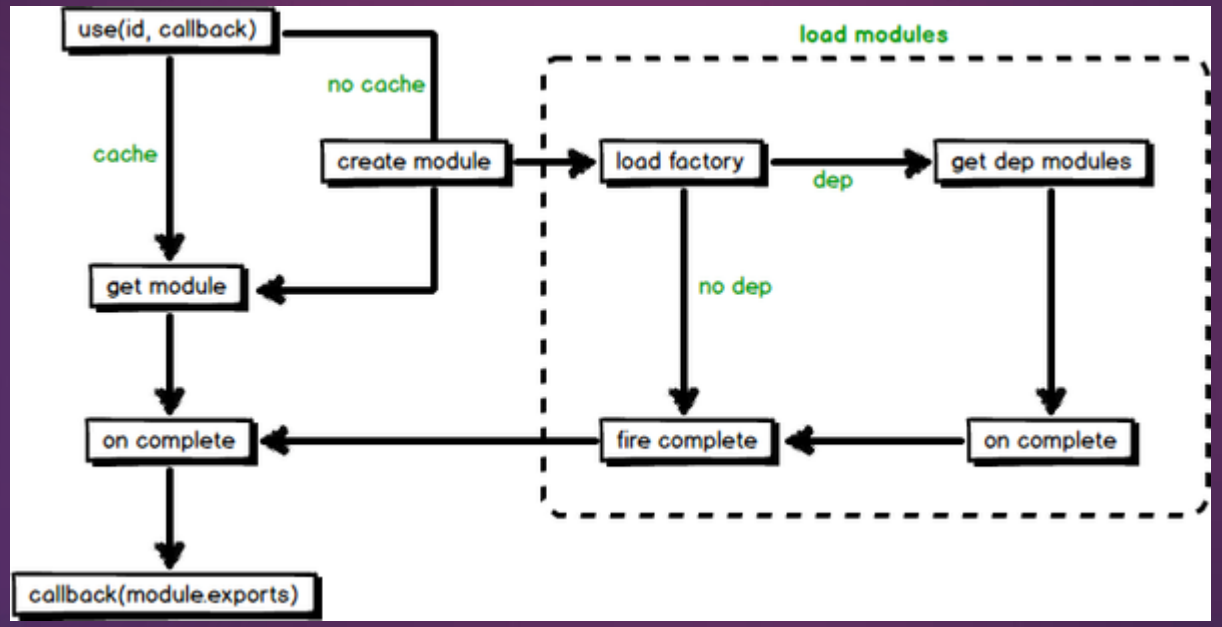
CMD规范

- CMD(Common Module Definition)规范和AMD很相似，尽量保持简单，并与CommonJS和Node.js的 Modules 规范保持了很大的兼容性。
- 在CMD规范中，一个模块就是一个文件。

```
define(function(require, exports, module) {
        var $ = require('jquery');
        var Spinning = require('./spinning');
        exports.doSomething = ...
        module.exports = ...
})
```

- 优点：
  1、依赖就近，延迟执行
  2、可以很容易在 Node.js 中运行

CMD规范


use(id, callback)
no cache
cache
create module
load modules
load factory
dep
get dep modules
get module
no dep
on complete
fire complete
on complete
callback(module.exports)

AMD和CMD的区别

- AMD和CMD起来很相似，但是还是有一些细微的差别，让我们来看一下他们的区别在哪里：
  1、对于依赖的模块，AMD是提前执行，CMD是延迟执行。
  2、AMD推崇依赖前置；CMD推崇依赖就近，只有在用到某个模块的时候再去require。

```
// AMD
define(['./a', './b'], function(a, b) {
        // 依赖必须一开始就写好
        a.doSomething()
        // 此处略去 100 行
        b.doSomething()
        ...
});
```

```
// CMD
define(function(require, exports, module) {
        var a = require('./a')
        a.doSomething()
        // 此处略去 100 行
        var b = require('./b')
        // 依赖可以就近书写
        b.doSomething()
        // ...
});
```

ES6模块化

- EcmaScript6标准增加了JavaScript语言层面的模块体系定义。ES6 模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS和AMD模块，都只能在运行时确定这些东西。
- 在 ES6 中，我们使用export关键字来导出模块，使用import关键字引用模块。需要说明的是，ES6的这套标准和目前的标准没有直接关系，目前也很少有JS引擎能直接支持。因此Babel的做法实际上是将不被支持的import翻译成目前已被支持的require。
- 尽管目前使用import和require的区别不大(本质上是一回事)，但依然强烈推荐使用import关键字，因为一旦JS引擎能够解析ES6的import关键字，整个实现方式就会和目前发生比较大的变化。如果目前就开始使用import关键字，将来代码的改动会非常小。

ES6模块化

```
import "jquery";
export function doStuff() {}
```

- 优点：
  1、容易进行静态分析
  2、面向未来的 EcmaScript 标准

webpack模块化机制

- webpack并不强制你使用某种模块化方案，而是通过兼容所有模块化方案让你无痛接入项目，当然这也是webpack牛逼的地方。
  有了webpack，你可以随意选择你喜欢的模块化方案，至于怎么处理模块之间的依赖关系及如何按需打包，放轻松，webpack会帮你处理好的。