javascript模块化开发

> JS 模块化就是指 JS 代码分成不同的模块，模块内部定义变量作用域只属于模块内部，模块之间变量命名不会相互冲突。各个模块相互独立，而且又可以通过某种方式相互引用协作。

模块系统概述

模块化是将系统分离成独立功能的方法，这样我们需要什么功能，就加载什么功能。

当一个项目开发得越来越复杂时，会遇到一些问题，例如：命名冲突、文件依赖等。

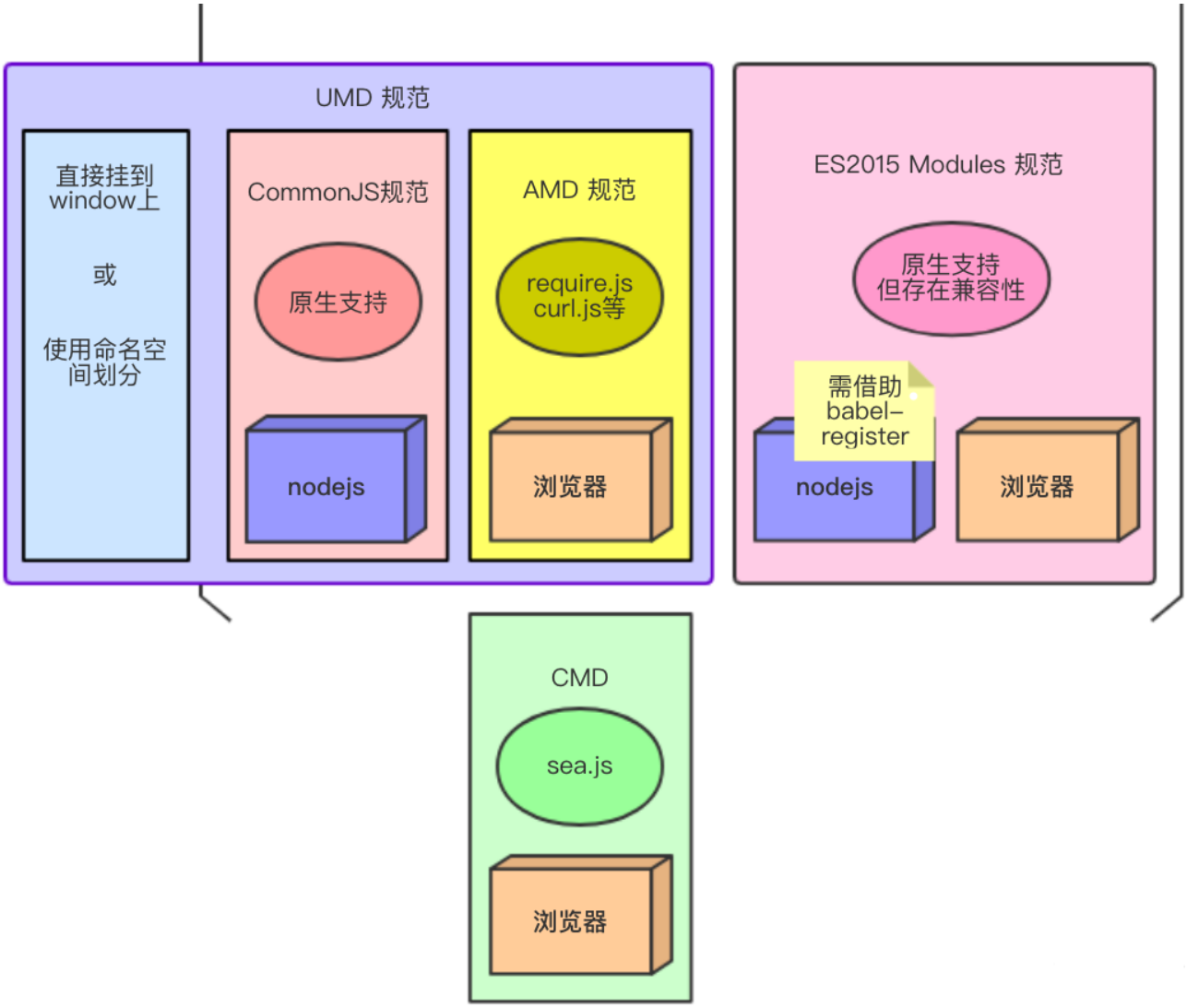
文件依赖就是，A负责`a.js`组件的开发，B负责`b.js`组件的开发，在`a.js`文件里面可能引用了`b.js`里面的某一个变量，而在项目运行时`a.js`在`b.js`之前加载，这个时候就会出现一些错误信息，但是`a.js`和`b.js`本身都没有错误，这就造成了我们排错的一些困难，这种问题我们称之为文件依赖的问题。

所以使用模块化开发时，可以避免以上问题，并且提升开发效率。

怎么理解呢？在模块化开发里面，会把一个文件当做一个单独的作用域存在，它们定义的变量、函数都不会相互影响。那如果在`a.js`里面依赖了`b.js`，我们就会进行声明，通过这种方式也可以减少文件依赖带来的BUG。

而提升开发效率主要是从代码可复用性和可维护性来提升的。

总结：在生产的角度，模块化开发是一种生产方式，这种方式生产效率高，维护成本低。



模块化开发演变

全局函数

在早期的开发过程中，就是将重复的代码封装搭到函数中，再将一系列的函数放到一个文件中。这种方式存在一些问题：存在污染全局变量、看不出相互的直接关系。

这种方式并不能解决根本的问题：命名冲突和文件依赖，于是演变为对象命名空间。

对象命名空间

通过对象命名空间的形式，从某种程度上解决了变量命名冲突的问题，但是并不能从根本上解决命名冲突。存在问题：内部状态可被外部改写、命名空间越来越长。

私有公有成员分离

利用此种方式将函数包装成一个独立的作用域，私有空间的变量和函数不会影响到全局作用域。这种方式相当于现在写插件的形式，解决了变量命名冲突的问题，但是没有解决降低开发复杂度的问题。

CommonJS

`CommonJS`规范加载模块是同步的，也就是说，加载完成才执行后面的操作。`Node.js`主要用于服务器编程，模块都是存在本地硬盘中，加载比较快，所以`Node.js`采用`CommonJS`规范。

`CommonJS`规范分为三部分：`module`(模块标识)、`require`(模块引用)、`exports`(模块定义)。`module`变量在每个模块内部，就代表当前模块；`exports`属性是对外的接口，用于导出当前模块的方法或变量；`require()`用来加载外部模块，读取并执行js文件，返回该模块的`exports`对象。

module.exports属性

module.exports属性表示当前模块对外输出的接口，其他文件加载该模块，实际上就是读取module.exports变量。

exports变量

node为每一个模块提供了一个exports变量(可以说是一个对象)，指向 module.exports。这相当于每个模块中都有一句这样的命令 var exports = module.exports;

这样，在对外输出时，可以在这个变量上添加方法。例如 exports.add = function (r){return Math.PI * r *r};注意：不能把exports直接指向一个值，这样就相当于切断了 exports 和module.exports 的关系。例如 exports=function(x){console.log(x)};

一个模块的对外接口，就是一个单一的值，不能使用exports输出，必须使用 module.exports输出。module.exports=function(x){console.log(x);};

AMD(require.js)

> https://requirejs.org/docs/download.html
>
> https://requirejs.org/

AMD 即Asynchronous Module Definition，中文名是“异步模块定义”的意思，它采用异步方式加载模块，模块的加载不影响它后面语句的运行，所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行

AMD也就是异步模块定义，它采用异步方式加载模块，主要针对的是浏览器端的模块规范。它通过`define`方法去定义模块，`require`方法去加载模块。

AMD定义：如果这个模块还需要依赖其他模块，那么`define`函数的第一个参数，必须是一个数组，指明该模块的依赖。

```
define([module-name?], [array-of-dependencies?], [module-factory-or-object]);

//module-name: 模块标识，可以省略。

//array-of-dependencies: 所依赖的模块，可以省略。

//module-factory-or-object: 模块的实现，或者一个JavaScript对象。

define([tools], function(){});
```

AMD模块的加载：

```
require(['modules'], callback);
```

第一个参数`['modules']`，是一个数组，里面的成员就是需要加载的模块；第二个参数`callback`，则是加载成功之后的回调函数，例如加载`math.js`：

```
require(['math'], function(){});
```

`require()`异步加载`math`，浏览器不会失去响应；它指定的回调函数，只有前面的模块加载完成后才会运行，解决了依赖性的问题。

```
define("alpha", ["require", "exports", "beta"], function (require, exports, beta) {
    exports.verb = function() {
        return beta.verb();
        //Or:
        return require("beta").verb();
    }
});
```

上面的代码定义了一个alpha的模块，这个模块依赖require，exports，beta，因此需要先加载它们，再执行后面的factory

require.js中采用require()语句加载模块，在定义好了模块后，我们可以使用require进行模块的加载

```
require([module], callback);
```

require要传入两个参数，第一个参数[module]，是一个数组，里面的成员就是要加载的模块，第二个参数callback，则是加载成功之后的回调函数

下面我们来看一个例子

```
require(['increment'], function (increment) {
    increment.add(1);
});
```

CMD(sea.js)

> https://github.com/seajs/seajs/issues/242
>
> https://www.zhangxinxu.com/sp/seajs/docs/zh-cn/cheatsheet.html

CMD即通用模块定义，CMD规范是国内发展出来的；正如AMD有`require.js`，CMD有个浏览器的实现`sea.js`。`sea.js`要解决的问题和`require.js`一样，只不过在模块定义方式和模块加载方式上有所不同。

在CMD规范中，一个模块就是一个文件。代码的书写格式如下：

```
define(function (require, exports, module) {
  // 模块代码
})
```

`require`是可以把其他模块导入进来的一个参数；`exports`可以把模块内的一些属性和方法导出；`module`是一个对象，上面存储了与当前模块相关联的一些属性和方法。CMD是按需加载，推崇依赖就近，延迟执行。文件是提前加载好的，只有在`require`的时候才去执行文件；

```
define(function(require, exports, module){
  var math = require('./math');
  math.add();
})
```

AMD和CMD的核心原理原理是通过动态加载 script 和事件监听的方式来异步加载模块；

require.js与sea.js写法对比

- `reuiqre.js`写法：

```
// 加载完jquery.js后，得到的执行结果$作为参数传入了回调函数
define(['jquery'], function($) {
  $('#header').hide();
});
```

- `sea.js`写法：

```
// 预加载了jquery.js
define(function(require, exports, module) {
  // 执行jquery.js模块，并得到结果赋值给$
  var $ = require('jquery');
  // 调用jquery.js模块提供的方法
  $('#header').hide();
});
```

ES6 Module

在ES6之前没有模块化，为了解决问题，提出了CommonJS，AMD，CMD；ES6模块化汲取CommonJS和AMD的优点，语法简洁，支持异步加载，未来可以成为浏览器和服务器通用的模块化解决方案。

ES6中模块的定义：ES6新增了两个关键字：`export`和`import`。`export`用于把模块里的内容暴露出来，`import`用于引入模块提供的功能。

ES6中模块的加载，`import`加载模块：

```
import{bar,foo,test,obj} from './lib'
foo();
```

注意：可以使用`export default`命令，为模块指定默认输出，一个模块只能有一个默认输出，所以`export default`只能使用一次。

ES6模块运行机制：ES6模块是动态引用，如果使用`import`从一个模块加载变量(即`import foo from 'foo'`)，变量不会被缓存，而是成为一个指向被加载模块的引用。等脚本执行时，根据只读引用，到被加载的那个模块中去取值。

Node 的模块化

Node.js 有一个简单的模块加载系统，遵循的是 CommonJS 的规范。 在 Node.js 中，文件和模块是一一对应的（每个文件被视为一个独立的模块）。

Node 在加载 JS 文件的时候，自动给 JS 文件包装上定义模块的头部和尾部。

Node会自动给js文件模块传递的5个参数，每个模块内的代码都可以直接用。而且您也看到了，我们的代码都会被包装到一个函数中，所以我们的代码的作用域都是在这个包装的函数内，这点跟浏览器的window全局作用域是不同的。

模块内的参数说明：

- __dirname： 当前模块的文件夹名称
- __filename： 当前模块的文件名称---解析后的绝对路径。
- module: 当前模块的引用，通过此对象可以控制当前模块对外的行为和属性等。
- require：是一个函数，帮助引入其他模块.
- exports：这是一个对于 module.exports 的更简短的引用形式，也就是当前模块对外输出的引用。

加载策略

Node.js的模块分为两类，一类为原生（核心）模块，一类为文件模块。

1. 模块在第一次加载后会被缓存。 这也意味着如果每次调用 require('foo') 都解析到同一文件，则返回相同的对象。
2. Node.js提供了一些底层的核心模块，它们定义在 Node.js 源代码的 lib/ 目录下。这些原生模块在Node.js源代码编译的时候编译进了二进制执行文件，加载的速度最快。开发人员自定义的js文件是动态加载的，加载速度比原生模块慢，这个只是在第一次加载有区别，模块加载完后都会被缓存，后续使用就不会被再次加载。
3. require() 总是会优先加载核心模块。 例如，require('http') 始终返回内置的 HTTP 模块，即使有同名文件。

文件模块中，又分为3类模块。这三类文件模块以后缀来区分，Node.js会根据后缀名来决定加载方法。

- .js。通过fs模块同步读取js文件并编译执行。
- .node。通过C/C++进行编写的Addon。通过dlopen方法进行加载。
- .json。读取文件，调用JSON.parse解析加载。