前端模块化原理15道面试题（https://github.com/minsion/interview-special）

面试题 1. 简述前端模块化开发的认识理解？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

1：webpack中是这样定义的：
在模块化编程中，开发者将程序分解成离散功能块(discrete chunks of functionality)，并称之为模块。每个模块具有比完整程序更小的接触面，使得校验、调试、测试轻而易举。精心编写的模块提供了可靠的抽象和封装界限，使得应用程序中每个模块都具有条理清楚的设计和明确的目的。
模块应该是职责单一、相互独立、低耦合的、高度内聚且可替换的离散功能块。

2：模块化的概念
模块化是一种处理复杂系统分解成为更好的可管理模块的方式，它可以把系统代码划分为一系列职责单一，高度解耦且可替换的模块，系统中某一部分的变化将如何影响其它部分就会变得显而易见，系统的可维护性更加简单易得。
模块化是一种分治的思想，通过分解复杂系统为独立的模块实现细粒度的精细控制，对于复杂系统的维护和管理十分有益。模块化也是组件化的基石，是构成现在色彩斑斓的前端世界的前提条件。

3：为什么需要模块化
前端开发和其他开发工作的主要区别，首先是前端是基于多语言、多层次的编码和组织工作，其次前端产品的交付是基于浏览器，这些资源是通过增量加载的方式运行到浏览器端，如何在开发环境组织好这些碎片化的代码和资源，并且保证他们在浏览器端快速、优雅的加载和更新，就需要一个模块化系统。

4：模块化的好处
可维护性。 因为模块是独立的，一个设计良好的模块会让外面的代码对自己的依赖越少越好，这样自己就可以独立去更新和改进。
命名空间。 在 JavaScript 里面，如果一个变量在最顶级的函数之外声明，它就直接变成全局可用。因此，常常不小心出现命名冲突的情况。使用模块化开发来封装变量，可以避免污染全局环境。
重用代码。 我们有时候会喜欢从之前写过的项目中拷贝代码到新的项目，这没有问题，但是更好的方法是，通过模块引用的方式，来避免重复的代码库。我们可以在更新了模块之后，让引用了该模块的所有项目都同步更新，还能指定版本号，避免 API 变更带来的麻烦。

面试题 2. 简述CommonJS和AMD的理解？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

（1）CommonJS
NodeJS是CommonJS规范服务器端的实现，webpack也是CommonJS的形式书写。同步

加载，服务器端从磁盘中读取速度快，运行在服务器端没有问题。

（2）AMD
AMD是Asynchronous Module Definition异步模块定义。
基于CommonJS规范的node.JS是服务端模块化的实现。实现浏览器端的模块化就是AMD，且能与服务器端兼容最好。同一个模块在服务器端和浏览器端都可以维护运行，简单方便很多，效率也提高了不少。
浏览器不能兼容CommonJS，于是AMD就出现了。浏览器不能兼容CommonJS的根本原因在于缺少node.JS的四个环境变量：module、exports、requrie、global。
为什么服务器端可以同步加载，浏览器端不能同步加载，需要异步加载?

```
var math = require('math');
math.add(2,3)
```

解析：第二行代码在第一行之后运行，必须等到math.js加载完成后运行，如果加载时间特别特别长，整个程序就会卡顿。这里的这里的requrie是同步加载的。浏览器的模块都在服务器端，等待时间取决于网络速度的快慢，等待的时间越长，浏览器响应的时间越长，甚至造成“假死”的状态。服务器端的模块放在本地硬盘中，同步加载读取的时间很快，几乎不会产生什么影响。

（3）CMD
AMD推崇依赖前置，CMD推崇依赖就近，延迟加载。

（4）ES6模块化
ES6模块化采用静态编译，在编译的时候就能确定依赖关系，以及输入和输出的变量。
CommonJS和AMD模块只能运行时确定。

二、AMD规范与CMD规范的区别
①CMD推崇依赖就近，AMD推崇依赖前置
②CMD延迟执行，AMD是提前执行
③CMD性能好，按需加载，当用户有需要在执行。AMD用户体验好，不延迟执行，依赖模块提前加载完毕

```
//AMD默认推荐的是
define(['./a', './b'], function (a, b) { //依赖前置，必须一开始就写好
a.doSomething()
b.doSomething()
})
//CMD默认推荐的是
define(function (require, exports, module) {
var a = require('./a')
a.doSomething()
var b = require('./b') //依赖就近，按需加载，需要哪个写哪个
b.doSomething()
})
```

AMD和CMD最大的区别是对依赖模块的执行时机处理不同，注意不是加载的时机或者方式不同

①AMD和CMD都是异步加载模块。AMD依赖前置，js可以提前知道所有的依赖模块，立即加载。CMD就近依赖，模块解析为字符串后才能知道依赖哪些模块。CMD性能好，按需加载，用户需要才加载。AMD用户体验好，模块全部提前加载好。
②AMD加载完模块之后就会立即执行它，所有模块加载完之后进入require回调函数，执行主逻辑。依赖模块的执行顺序和开发人员写的不一样，哪一个模块网速好先下载哪个就先执行，但是主逻辑一定是所有模块加载完成后才执行。
③CMD加载完某个模块后并不执行，只是下载，在所有模块加载完成后进入主逻辑，遇到require语句的时候才执行对应的模块。依赖模块的执行顺序和开发人员写的一样。

三、ES6模块和CommonJS规范区别
①CommonJS支持动态导入，ES6不支持，是静态编译。
②CommonJS同步加载，用于服务端，文件放在本地磁盘，读取速度快。同步导入卡住主线程也并无影响。ES6是异步加载，用于浏览器端，不能同步加载，会导致页面渲染，用户体验差。
③CommonJS模块输出的是值拷贝，内部的变化影响不到值的变化。ES6模块输出的是值引用，原始值变化，加载的值也会跟着变化，ES6模块是动态引用，并且不会缓存值。
④CommonJS模块是运行时加载，ES6模块是编译时输出接口。CommonJS模块就是对象，输入时先加载整个模块，生成一个对象，然后从对象读取方法。ES6模块不是对象，export输出指定代码，import导入加载某个值，而不是整个模块。
⑤关于模块顶层的this指向问题，在CommonJS顶层，this指向当前模块；而在ES6模块中，this指向undefined。
⑥ES6模块当中，是支持加载CommonJS模块的。但是反过来，CommonJS并不能requireES6模块，在NodeJS中，两种模块方案是分开处理的。

四、加载模式的总结
（1）CommonJS加载模式—同步/运行时加载
CommonJS加载模块是同步的。输入的时候加载整个模块，生成一个对象，从这个对象上读取方法。子模块完成加载，才能执行后面的操作。输入的值是被输出的值的拷贝，父模块引入子模块，引入的是子模块的值拷贝，模块的内部变化无法影响这个值。
（2）AMD加载模式—异步加载、依赖前置
AMD在浏览器端异步加载，AMD推崇依赖前置，加载完模块之后就会立即执行它。
（3）CMD加载模式—异步加载、依赖就近
CMD在浏览器端异步加载，CMD推崇依赖就近，加载完模块不会立即执行，只是加载，等到需要的时候才会执行。
（4）ES6加载模式—静态编译
ES6静态编译，在编译的时候就能确定依赖，编译的时候输出接口。export输出指定代码，import某个值不是整个模块。

五、ES6、CommonJS循环引用问题
什么是循环引用？循环加载指的是a脚本的执行依赖b脚本，b脚本的执行依赖a脚本。
①CommonJS模块是加载时执行。一旦出现某个模块被“循环加载”，就只输出已经执行的部分，没有执行的部分不会输出。
②ES6模块对导出模块，变量，对象是动态引用，遇到模块加载命令import时不会去执行模块，只是生成一个指向被加载模块的引用

面试题 3. 简述前端模块化开发的好处是什么?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

模块化开发可以带来以下好处：

1：提高代码的复用性：模块化可以将代码划分成可重用的部分，降低代码的冗余和重复，提高代码的复用性。
2：简化代码的维护和调试：当一个软件系统变得越来越复杂时，进行模块化开发可以使得每个模块都相对独立，这样就可以方便地维护和调试每个模块，而不必考虑整个系统的复杂性。
3：提高代码的可读性：模块化可以使得代码更加结构化，清晰明了，从而提高代码的可读性和可维护性。
4：提高开发效率：模块化开发可以使得团队成员在不同模块上并行开发，从而提高开发效率。
5：降低项目的风险：模块化开发可以使得开发人员更加关注模块之间的接口和依赖关系，从而降低项目的风险。
6：总之，模块化开发是一种有效的软件开发模式，可以提高软件开发的质量、效率和可维护性，特别是在大型软件系统的开发中，模块化更是必不可少的

面试题 4. 简述require.js解决了什么问题？

推荐指数： ★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

解决了以下问题。
（1）实现了 JavaScript文件的异步加载。
（2）有助于管理模块之间的依赖性。
（3）便于代码的编写和维护

面试题 5. 如何实现前端模块化开发?

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

require. js、 SeaJS都是适用于web浏览器端的模块加载器，使用它们可以更好地组织Javascript代码

面试题 6. 简述模块化的 JavaScript开发的优势？

推荐指数： ★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

优势如下。
（1）将功能分离出来
（2）具有更好的代码组织方式
（3）可以按需加载。
（4）避免了命名冲突。
（5）解决了依赖管理问题

面试题 7. 简述 CommonJS规范？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

每个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。
在模块中使用global 定义全局变量，不需要导出，在别的文件中可以访问到。
每个模块内部，module变量代表当前模块。这个变量是一个对象，它的exports属性（即module.exports）是对外的接口。
通过 require加载模块，读取并执行一个js文件，然后返回该模块的exports对象。
所有代码都运行在模块作用域，不会污染全局作用域。
模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
模块加载的顺序，按照其在代码中出现的顺序

面试题 8. 简述CMD（Common module Definition，通用模块定义）规范的理解？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

CMD 叫做通用模块定义规范（Common Module Definiton），它是类似于 CommonJs 模块化规范，但是运行于浏览器之上的，关于模块化的好处我们在 CommonJs 篇文章中我们了解过。它是随着前端业务和架构的复杂度越来越高运用而生的，来自淘宝玉伯的 SeaJS 就是它的实现。
CMD 规范尽量保持简单，并与 CommonJS 的 Modules 规范保持了很大的兼容性。通过 CMD 规范书写的模块，可以很容易在 Node.js 中运行。在 CMD 规范中，一个模块就是一个文件。格式如下：

```
define(factory);
```

具体用法如下：

```
// moudle-a.js
define(function(require, exports, module) {
module.exports = {
a: 1
}
;
}
);
```

```
// moudle-b.js
define(function(require, exports, module) {
var ma = require('./moudle-a');
var b = ma.a + 2;
module.exports = {
b: b
}
;
}
);
```

CMD 规范拥有简单、异步加载脚本、友好的调试并且兼容 Nodejs，它的确在开发过程中给我们提供了较好的模块管理方式。

面试题 9. 简述EMAScript 6模块规范？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

在任何一个大型应用中模块化是很常见的。ES6的模块为JavaScript提供了这个特性，并且为这些模块提供了许多选择方法来导出和引入对象。Ravi Kiran 在《 Modules in ECMAScript 6 (ES6) 》一文中主要讨论了ES6模块系统。以下为该文章的简译内容：

无论使用何种编程语言开发大型应用，最关键的特性就是代码模块化。这个概念在不同的编程语言里有着不同的命名，在C里为头部文件，C++和C#里为命名空间，Java中为包，名称不一样但解决的是同一问题。正如《 ECMAScript 6 — New language improvements in JavaScript 》系列文章中第一篇所提到的那样，最初JavaScript并不是用来编写大量代码的，比如创建大型框架、App应用等。就在我们因为JavaScript缺少对模块的支持而编写大量代码时，开源开发者提出了一些标准，如CommoneJs模块模型、异步模块定义（AMD）以及一些库，来实现模块化。在过去几年里,这些库获得了广泛关注，并成功应用到多个企业规模级的应用程序中。

ES6为JavaScript带来了模块特性。浏览器实现这一特性还需要一段时间，因为它们必须定义一个方法来动态下载文件。在浏览器支持该特性以前，我们可以使用编译器，如Traceur、6to5、ES6 Module Loader以及其它可以让ES6模块转换成ES5的转码器。

JavaScript模块系统的现状

CommonJS模块系统

CommonJs是一个由开源开发者组成的团队，主要围绕JavaScript实现一些API及开展研发实践。该团队提出了一个JavaScript模块规范。每个文件都可当作一个模块，并且每个文件可以访问两个对象：require和export。require用来接收字符串（模块名），并返回该模块

输出的对象。export对象用来导出该模块的方法和变量。require方法返回的就是export对象。模块同步加载。服务器端JavaScript引擎Node.js就是用的这个模块系统。

异步模块定义（AMD）

AMD是一个采用异步方式加载依赖模块的模块系统。如果模块在不同文件中，它们将采用XHR进行加载。某一模块将等其所依赖的模块一一加载后才会被执行。AMD模块必须是一个函数，并作为参数传入define函数中。函数的返回值将传输给所有依赖的模块，所获得返回值又将作为参数传给模块方法。Require.js库中实现了AMD。

TypeScript模块

TypeScript，作为JavaScript的超集，也提供了一个模块系统。当它被编译时，便开始使用JavaScript模块模式。TypeScript模块使用module关键字定义，任何被输出的对象必须使用export关键字定义。import关键字用来将其它模块加载入模块中，并捕捉该模块导出的对象。TypeScript模块是同步加载的。

ES6模块系统

ES6模块系统启发于上述现有模块系统，它具有以下特性：

使用export关键词导出对象。这个关键字可以无限次使用；
使用import关键字将其它模块导入某一模块中。它可用来导入任意数量的模块；
支持模块的异步加载；
为加载模块提供编程支持。
接下来让我们通过具体编程方法看看每一个特性。

导出对象

在现有的模块系统中，每个JavaScript代码文件在ES6中都是一个模块。只有模块中的对象需要被外部调用时，模块才会输出对象，其余则都是模块的私有对象。该处理方式将细节进行封装，仅导出必要的功能。

从模块里导出对象，ES6为我们提供了不同方法，见下面的讨论。

内联导出

ES6模块里的对象可在创建它们的声明中导出。一个模块中可无数次使用export，所有的对象将被一起导出。请看下面的例子：

```
export class Employee{
constructor(id, name, dob){
this.id = id;
this.name=name;
this.dob= dob;
}
getAge(){
```

```
return (new Date()).getYear() – this.dob.getYear();
}
}
export function getEmployee(id, name, dob){
return new Employee(id, name, dob);
}
var emp = new Employee(1, "Rina", new Date(1987, 1, 22));
```

案例中的模块导出了两个对象： Employee类，getEmployee函数。因对象emp未被导出，所以其仍为模块私有。

导出一组对象

尽管内联导出很有效，但在大规模模块中，它就很难发挥作用了，因为我们可能无法追踪到模块导出来的对象。在这种情况下，更好的办法是，在模块的末尾单独进行导出声明，以导出该模块中的全部对象。

使用单独导出声明重写上一案例中的模块，结果如下：

```
class Employee{
constructor(id, name, dob){
this.id = id;
this.name=name;
this.dob= dob;
}
getAge(){
return (new Date()).getYear() – this.dob.getYear();
}
}
function getEmployee(id, name, dob){
return new Employee(id, name, dob);
}
var x = new Employee(1, "Rina", new Date(1987, 1, 22));
export {Employee, getEmployee};
```

在导出时，重命名对象也是可以的。如下例所示，Employee在导出时名字改为了Associate，函数GetEmployee改名为getAssociate。

```
export {
Associate as Employee,
getAssociate as getEmployee
};
```

Default导出

使用关键字default，可将对象标注为default对象导出。default关键字在每一个模块中只能使用一次。它既可以用于内联导出，也可以用于一组对象导出声明中。

下面案例展示了在组导出语句中使用default：

```
export default {
Employee,
getEmployee
};
```

导入模块

现有模块可以使用关键字import导入到其它模块。一个模块可以被导入任意数量的模块中。下文展示了导入模块的不同方式。

无对象导入

如果模块包含一些逻辑要执行，且不会导出任何对象，此类对象也可以被导入到另一模块中。如下面案例所示：

```
import './module1.js';
```

导入默认对象

采用Default导出方式导出对象，该对象在import声明中将直接被分配给某个引用，如下例中的“d”。

```
import d from './module1.js';
```

导入命名的对象

正如以上讨论的，一个模块可以导出许多命名对象。如果另一模块想导入这些命名对象，需要在导入声明中一一列出这些对象。举个例子：

```
import {Employee, getEmployee} from './module1.js';
```

当然也可在同一个声明中导入默认对象和命名对象。这种情况下，默认对象必须定义一个别名，如下例。

```
import {default as d, Employee} from './module1.js';
```

导入所有对象

以上几种情况，只有import声明中列举的对象才会被导入并被使用，而其它对象则无法在导入模块中使用。当然，这就要求用户了解哪些对象可以导出并加以利用。如果模块导出大量对象，另一模块想引入所有导出的对象，就必须使用如下声明：

```
import * as allFromModule1 from './module1.js';
```

allFromModule1这一别名将指向所有从module1导出的对象。在导入模块中，它们作为属性可被访问。

可编程式的按需导入

如果想基于某些条件或等某个事件发生后再加载需要的模块，可通过使用加载模块的可编程API（programmatic API）来实现。使用System.import方法，可按程序设定加载模块。这是一个异步的方法，并返回Promise。

该方法的语法示例如下：

```
System.import('./module1.js')
.then(function(module1){
//use module1
}, function(e){
//handle error
});
```

如果模块加载成功且将导出的模块成功传递给回调函数，Promise将会通过。如果模块名称有误或由于网络延迟等原因导致模块加载失败，Promise将会失败。

ES6模块使用现状

到目前为止，所有浏览器还不能自然支持ES6模块，所以在浏览器加载之前，我们需要使用转译器（transpiler）将代码转换成ES5。直到现在，我一直使用Traceur作为我的转译器，建议大家使用相同的工具将模块代码转化为浏览器可识别的代码。让我们看看编译ES6模块的几种不同的方法。

使用Traceur动态编译ES6模块

当浏览器加载脚本后，我们可以使用Traceur的客户端库动态编译ES6模块。使用该方法，运行模块无需运行任何命令。我们要做得就是，在页面上加载Traceur库，及添加代码脚本来运行WebPageTranscoder。

现在，我们就可以在script标签内，将类型指定成模块，以此导入任何一个ES6文件。

类型指定为模块的任何脚本标签将被ES6客户端库获取并处理。上面代码块中的导入语句将发送AJAX请求，捕获相应的JavaScript文件，并载入它。如果模块内部引用了另一个模块，单独的AJAX请求将发出，以加载与引用模块相对应的文件。

使用Traceur命令编译ES6模块

使用Traceur命令可以将ES6模块编译成AMD或者CommonJS模块。这个方法有两大优点。

模块完成编译，浏览器不必执行额外动作；
如果应用已经使用ES5及AMD（或CommonJs）模块系统构建了一半，程序的另一半也可以使用ES6，并被编译为这些模块系统中的任何一个，而不是立即把整个应用编译成ES6。
为了使用编译完成的AMD/CommonJs的模块，我们需要包含支持模块系统的库。我个人比较倾向AMD，所以我将在这里介绍一下它。CommonJS模块的步骤和这个差不多。

下面这句命令是用来让包含ES6模块的文件夹编译成AMD，并把它们存储在一个单独的文件夹：

```
traceur --dir modules es5Modules --modules=amd
```

使用CommonJs，你需要在上面命令中使用commonjs代替modules。

在这里，modules指的是包含ES6的文件夹，es5Modules指的是输出目录。如果查看es5Modules文件夹下的任何文件，你将看到该AMD定义块。require.js支持AMD，所以我们可以在HTML页面中，使用script引入require.js，并用data-main属性指明开始文件，就像下面这样：

使用Traceur Grunt Task编译ES6模块

使用命令编译模块很累而且更容易出错。我们可以使用grunt-traceur自动化编译过程。此时，你需要安装NPM包。

npm intall grunt-traceur —save
Grunt任务所需数据与提供给命令的数据一样。下面是任务的一些配置：

```
traceur: {
options: {
modules: "amd"
},
custom: {
files: [{
expand: true,
cwd: 'modules',
src: ['*.js'],
dest: 'es5Modules'
}]
}
}
```

现在你可以在控制台里使用下面的命令来运行上面的Grunt任务：

grunt traceur
正如你所看见的那样，它和我们使用命令所产生的效果是一样的。

结论

任何一个大型应用中，模块化十分必要。ES6模块为JavaScript提供了该特性，这些模块提供了众多选择来导出和引入对象。我很期待该特性被浏览器支持的那一天，到时我们无需加载任何第三方库即可创建、加载JavaScript模块。目前流行的客户端MV*框架Angular.js在其2.0版本（目前还在开发中）中就使用了ES6的模块化。

让我们开始使用模块系统，从而让我们的代码更具组织和可读性。

面试题 10. 简述AMD与CMD的区别？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

1、CMD和AMD都是为了JavaScript模块化开发的规范
2、CMD是sea.js推广过程中对模块定义的规范化产出；AMD是require.js推广过程中对模块定义的规范化产出
3、AMD是异步模块定义的意思，他是一个在浏览器端模块开发规范，由于不是JS原生支持，使用AMD规范进行页面开发时，需要对应的函数库
4、require.js解决的问题，多个JS文件可以有依赖关系，被依赖的文件需要早于依赖它的文件加载到浏览器，JS加载的时候浏览器停止页面渲染，加载文件越多，页面失去响应时间越长
5、CMD通用模块定义，是国内发展的，有浏览器实现Sea.js，Sea.js要解决的问题和require.js一样，只不过模块定义的方式和模块加载时机有所不同
6、CMD 推崇依赖就近，AMD 推崇依赖前置

面试题 11. 解释前端模块化是否等同于 JavaScript模块化?

推荐指数：★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

前端开发相对其他语言来说比较特殊，因为我们实现一个页面功能总是需要JavaScript、CSS和HTML三者相互交织。
如果一个功能只有 JavaScript实现了模块化， CSS和 Template还处于原始状态，那么调用这个功能的时候并不能完全通过模块化的方式，这样的模块化方案并不是完整的。
所以我们真正需要的是一种可以将 JavaScript 、CSS和HTML同时都考虑进去的模块化方案，而非只使用 JavaScript模块化方案。综上所述，前端模块化并不等同于 JavaScript模块化

面试题 12. JavaScript模块化是否等同于异步模块化?

推荐指数：★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

主流的 JavaScript模块化方案都使用“异步模块定义”的方式，这种方式给开发带来了极大的不便，所有的同步代码都需要修改为异步方式。
当在前端开发中使用“ CommonJs”模块化开发规范时，开发者可以使用自然、容易理解的模块定义和调用方式，不需要关注模块是否异步，不需要改变开发者的开发行为。因此JavaScript模块化并不等同于异步模块化

面试题 13. 简述require.JS与 SeaJS的异同是什么?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

相同之处
RequireJS 和 SeaJS 都是模块加载器，倡导的是一种模块化开发理念，核心价值是让 JavaScript 的模块化开发变得更简单自然。

不同之处
两者的区别如下：

定位有差异。RequireJS 想成为浏览器端的模块加载器，同时也想成为 Rhino / Node 等环境的模块加载器。SeaJS 则专注于 Web 浏览器端，同时通过 Node 扩展的方式可以很方便跑在 Node 服务器端。

遵循的规范不同。RequireJS 遵循的是 AMD（异步模块定义）规范，SeaJS 遵循的是 CMD （通用模块定义）规范。规范的不同，导致了两者 API 的不同。SeaJS 更简洁优雅，更贴近 CommonJS Modules/1.1 和 Node Modules 规范。

社区理念有差异。RequireJS 在尝试让第三方类库修改自身来支持 RequireJS，目前只有少数社区采纳。SeaJS 不强推，采用自主封装的方式来“海纳百川”，目前已有较成熟的封装策略。

代码质量有差异。RequireJS 是没有明显的 bug，SeaJS 是明显没有 bug。

对调试等的支持有差异。SeaJS 通过插件，可以实现 Fiddler 中自动映射的功能，还可以实现自动 combo 等功能，非常方便。RequireJS 无这方面的支持。

插件机制不同。RequireJS 采取的是在源码中预留接口的形式，源码中留有为插件而写的代码。SeaJS 采取的插件机制则与 JavaScript 语言以及Node 的方式一致：开放自身，让插件开发者可直接访问或修改，从而非常灵活，可以实现各种类型的插件。

还有不少细节差异就不多说了。

总之，SeaJS 从 API 到实现，都比 RequireJS 更简洁优雅。如果说 RequireJS 是 Prototype 类库的话，则 SeaJS 是 jQuery 类库。

最重要的
最后，向 RequireJS 致敬！RequireJS 和 SeaJS 是好兄弟，一起努力推广模块化开发思想，这才是最重要的

面试题 14. 解释什么是前端模块化规范 ？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

模块化是前端领域发展的趋势之一，他的好处非常的多：

他可以抽离公共的代码，避免重复的复制粘贴

他可以隔离作用域，避免变量的冲突（在es6出现之前，人们会使用IIFE来完成这个操作）
他可以将一个复杂的系统分解为多个子模块，便于开发和维护

面试题 15. 简述为什么要通过模块化方式进行开发?

推荐指数： ★★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

原因如下。
（1）高内聚低耦合，有利于团队开发。当项目很复杂时，将项目划分为子模块并分给不同的人开发，最后再组合在一起，这样可以降低模块与模块之间的依赖关系，实现
低耦合，模块中又有特定功能体现高内聚特点。
（2）可重用，方便维护。模块的特点就是有特定功能，当两个项目都需要某种功能时，定义一个特定的模块来实现该功能，这样只需要在两个项目中都引入这个模块就能够实现该功能，不需要书写重复性的代码。
另外，当需要变更该功能时，直接修改该模块，这样就能够修改所有项目的功能，维护起来很方便