当我们编写JavaScript代码时，我们可以申明全局变量：

**var** s = 'global';

在浏览器中，大量使用全局变量可不好。如果你在a.js中使用了全局变量s，那么，在b.js中也使用全局变量s，将造成冲突，b.js中对s赋值会改变a.js的运行逻辑。

也就是说，JavaScript语言本身并没有一种模块机制来保证不同模块可以使用相同的变量名。

那Node.js是如何实现这一点的？

其实要实现“模块”这个功能，并不需要语法层面的支持。Node.js也并不会增加任何JavaScript语法。实现“模块”功能的奥妙就在于JavaScript是一种函数式编程语言，它支持闭包。如果我们把一段JavaScript代码用一个函数包装起来，这段代码的所有“全局”变量就变成了函数内部的局部变量。

请注意我们编写的hello.js代码是这样的：

**var** s = 'Hello';**var** name = 'world';

console.log(s + ' ' + name + '!');

Node.js加载了hello.js后，它可以把代码包装一下，变成这样执行：

(**function** () {

*// 读取的hello.js代码:*

**var** s = 'Hello';

**var** name = 'world';

console.log(s + ' ' + name + '!');

*// hello.js代码结束*

})();

这样一来，原来的全局变量s现在变成了匿名函数内部的局部变量。如果Node.js继续加载其他模块，这些模块中定义的“全局”变量s也互不干扰。

所以，Node利用JavaScript的函数式编程的特性，轻而易举地实现了模块的隔离。

但是，模块的输出module.exports怎么实现？

这个也很容易实现，Node可以先准备一个对象module：

*// 准备module对象:***var** module = {

id: 'hello',

exports: {}

};**var** load = **function** (module) {

*// 读取的hello.js代码:*

**function** greet(name) {

console.log('Hello, ' + name + '!');

}

module.exports = greet;

*// hello.js代码结束*

**return** module.exports;

};**var** exported = load(module);*// 保存module:*

save(module, exported);

可见，变量module是Node在加载js文件前准备的一个变量，并将其传入加载函数，我们在hello.js中可以直接使用变量module原因就在于它实际上是函数的一个参数：

**module**.exports = greet;

通过把参数module传递给load()函数，hello.js就顺利地把一个变量传递给了Node执行环境，Node会把module变量保存到某个地方。

由于Node保存了所有导入的module，当我们用require()获取module时，Node找到对应的module，把这个module的exports变量返回，这样，另一个模块就顺利拿到了模块的输出：

**var** greet = **require**('./hello');

以上是Node实现JavaScript模块的一个简单的原理介绍。

### module.exports vs exports

很多时候，你会看到，在Node环境中，有两种方法可以在一个模块中输出变量：

方法一：对module.exports赋值：

*// hello.js*

**function** hello() {

console.log('Hello, world!');

}

**function** greet(name) {

console.log('Hello, ' + name + '!');

}

module.exports = {

hello: hello,

greet: greet

};

方法二：直接使用exports：

*// hello.js*

**function** hello() {

console.log('Hello, world!');

}

**function** greet(name) {

console.log('Hello, ' + name + '!');

}

**function** hello() {

console.log('Hello, world!');

}

exports.hello = hello;

exports.greet = greet;

但是你不可以直接对exports赋值：

*// 代码可以执行，但是模块并没有输出任何变量:*

exports = {

hello: hello,

greet: greet

};

如果你对上面的写法感到十分困惑，不要着急，我们来分析Node的加载机制：

首先，Node会把整个待加载的hello.js文件放入一个包装函数load中执行。在执行这个load()函数前，Node准备好了module变量：

var **module = {**

id: 'hello',

exports: {}

};

load()函数最终返回module.exports：

var **load** = function (exports, **module**) {

// hello.js的文件内容

...

// **load**函数返回:

return **module**.exports;

};

var exported = **load**(**module**.exports, **module**);

也就是说，默认情况下，Node准备的exports变量和module.exports变量实际上是同一个变量，并且初始化为空对象{}，于是，我们可以写：

exports.foo = **function** () { **return** 'foo'; };

exports.bar = **function** () { **return** 'bar'; };

也可以写：

module.exports.foo = **function** () { **return** 'foo'; };

module.exports.bar = **function** () { **return** 'bar'; };

换句话说，Node默认给你准备了一个空对象{}，这样你可以直接往里面加东西。

但是，如果我们要输出的是一个函数或数组，那么，只能给module.exports赋值：

module.exports = **function** () { **return** 'foo'; };

给exports赋值是无效的，因为赋值后，module.exports仍然是空对象{}。

### 结论

如果要输出一个键值对象{}，可以利用exports这个已存在的空对象{}，并继续在上面添加新的键值；

如果要输出一个函数或数组，必须直接对module.exports对象赋值。

所以我们可以得出结论：直接对module.exports赋值，可以应对任何情况：

module.exports = {

foo: **function** () { **return** 'foo'; }

};

或者：

module.exports = **function** () { **return** 'foo'; };

最终，我们强烈建议使用module.exports = xxx的方式来输出模块变量，这样，你只需要记忆一种方法。