第三章：加载模块

本章内容：

- 加载模块
- 创建模块
- 使用 node_modules 目录

JavaScript 是世界上使用频率最高的编程语言之一，它是 Web 世界的通用语言，被所有浏览器所使用。JavaScript 的诞生要追溯到 Netscape 那个时代，它的核心内容被仓促的开发出来，用以对抗 Microsoft，参与当时白热化的浏览器大战。由于过早的发布，无可避免的造成了它的一些不太好的特性。

尽管它的开发时间很短，但是 JavaScript 依然具备了很多强大的特性，不过，每个脚本共享一个全局命名空间这个特性除外。

一旦 Web 页面加载了 JavaScript 代码，它就会被注入到全局命名空间，会和其他所有已加载的脚本公用同一个地址空间，这会导致很多安全问题，冲突，以及一些常见问题，让 bug 即难以跟踪又很难解决。

不过谢天谢地，Node 为服务器端 JavaScript 定了一些规范，还实现了 CommonJS 的模块标准，在这个标准里，每个模块有自己的上下文，和其他模块相区分。这意味着，模块不会污染全局作用域，因为根本就没有所谓的全局作用域，模块之间也不会相互干扰。

本章，我们将了解几种不同的模块以及如何加载它们。

把代码拆分成一系列定义良好的模块可以帮你掌控你的应用程序，下面我们将学习如何创建和使用你自己的模块。

了解 Node 如何加载模块

Node 里,可以通过文件路径来引用模块，也可以通过模块名引用，如果用名称引用非核心模块，Node 最终会把模块名影射到对应的模块文件路径。而那些包含了核心函数的核心模块，会在 Node 启动时被预先加载。

非核心模块包括使用 NPM（Node Package Manager）安装的第三方模块，以及你或你的同事创建的本地模块。

每个被当前脚本导入的模块都会向程序员暴露一组公开 API，使用模块前，需要用 require 函数来导入它，像这样：

```
var module = require('module_name')
```

上面的代码会导入一个名为 module_name 的模块，它可能是个核心模块，也可以是用 NPM 安装的模块，require 函数返回一个包含模块所有公共 API 的对象。随模块的不同，返回的对象可能是任何 JavaScript 值，可以是一个函数，也可以是个包含了一系列属性（函数，数组或者任何 JavaScript 对象）的对象。

导出模块

CommonJS 模块系统是 Node 下文件间共享对象和函数的唯一方式。对于一个很复杂的程序，你应该把一些类，对象或者函数重构成一系列良好定义的可重用模块。对于模块使用者来说，模块仅对外暴露出那些你指定的代码。

在下面的例子里你将会了解到，在 Node 里文件和模块是一一对应的，我们创建了一个叫 circle.js 的文件，它仅对外导出了 Circle 构造函数。

```
function Circle(x, y, r) {
    function r_squared() {
        return Math.pow(r, 2);
    }
    function area() {
        return Math.PI * r_squared();
    }
    return {
        area: area
    };
}
module.exports = Circle;
```

代码里最重要的是最后一行，它定义了模块对外导出了什么内容。module 是个特殊的变量，它代表当前模块自身，而 module.exports 是模块对外导出的对象，它可以是任何对象，在这个例子里，我们把 Circle 的构造函数导出了，这样模块使用者就可以用这个模块来创建 Circle 实例。

你也可以导出一些复杂的对象，module.exports 被初始化成一个空对象，你把任何你想暴露给外界的内容，作为 module.exports 对象的属性来导出。比如，你设计了一个模块，它对外暴露了一组函数：

```
function printA() {
    console.log('A');
}
function printB() {
    console.log('B');
}
function printC() {
    console.log('C');
}
module.exports.printA = printA;
module.exports.printB = printB;
module.exports.pi = Math.PI;
```

这个模块导出了两个函数（printA 和 printB）和一个数字(pi)，调用代码看起来像这样：

```
var myModule2 = require('./myModule2');
myModule2.printA(); // -> A
myModule2.printB(); // -> B
console.log(myModule2.pi); // -> 3.141592653589793
```

加载模块

前面提到过，你可以使用 require 函数来加载模块，不用担心在代码里调用 require 会影响全局命名空间，因为 Node 里就没有全局命名空间这个概念。如果模块存在且没有任何语法或初始化错误，require 函数就会返回这个模块对象，你还可以这个对象赋值给任何一个局部变量。

模块有几种不同的类型，大概可以分为核心模块，本地模块和通过 NPM 安装的第三方模块，根据模块的类型，有几种引用模块的方式，下面我们就来了解下这些知识。

加载核心模块

Node 有一些被编译到二进制文件里的模块，被称为核心模块，它们不能通过路径来引用，只能用模块名。核心模块拥有最高的加载优先级，即使已经有了一个同名的第三方模块，核心模块也会被优先加载。

比如，如果你想加载和使用 http 核心模块，可以这样做：

```
var http = require('http');
```

这将返回一个包含了 http 模块对象，它包含了 Node API 文档里定义的那些 htpp 模块的 API。

加载文件模块

你也可以使用绝对路径从文件系统里加载模块：

```
var myModule = require('/home/pedro/my_modules/my_module');
```

也可以用一个基于当前文件的相对路径：

```
var myModule = require('../my_modules/my_module');
var myModule2 = require('./lib/my_module_2');
```

注意上面的代码，你可以省略文件名的扩展名，如果 Node 找不到这个文件，会尝试在文件名后加上 js 后缀再次查找（译者注：其实除了 js，还会查找 json 和 node，具体可以看官网文档），因此，如果在当前目录下存在一个叫 my_module.js 的文件，会有下面两种加载方式：

```
var myModule = require('./my_module');
var myModule = require('./my_module.js');
```

加载目录模块

你还可以使用目录的路径来加载模块：

```
var myModule = require('./myModuleDir');
```

Node 会假定这个目录是个模块包，并尝试在这个目录下搜索包定义文件 package.json。

如果没找到，Node 会假设包的入口点是 index.js 文件（译者注：除了 index.js 还会查找 index.node，.node 文件是 Node 的二进制扩展包，具体见官方文档），以上面代码为例，Node 会尝试查找./myModuleDir/index.js 文件。

反之，如果找到了 package.json 文件，Node 会尝试解析它，并查找包定义里的 main 属性，然后把 main 属性的值当作入口点的相对路径。以本例来说，如果 package.json 定义如下：

```
{
```

```
    "name" : "myModule",
    "main" : "./lib/myModule.js"
}
```

Node 就会尝试加载./myModuleDir/lib/myModule.js 文件

从 node_modules 目录加载

如果 require 函数的参数不是相对路径，也不是核心模块名，Node 会在当前目录的 node_modules 子目录下查找，比如下面的代码，Node 会尝试查找文件./node_modules/myModule.js:

```
var myModule = require('myModule.js');
```

如果没找到，Node 会继续在上级目录的 node_modules 文件夹下查找，如果还没找到就继续向上层目录查找，直到找到对应的模块或者到达根目录。

你可以使用这个特性来管理 node_modules 目录的内容或模块，不过最好还是把模块的管理任务交给 NPM（见第一章），本地 node_modules 目录是 NPM 安装模块的默认位置，这个设计把 Node 和 NPM 关联在了一起。通常，作为开发人员不必太关心这个特性，你可以简单的使用 NPM 安装，更新和删除包，它会帮你维护 node_modules 目录

缓存模块

模块在第一次成功加载后会被缓存起来，就是说，如果模块名被解析到同一个文件路径，那么每次调用 require('myModule')都确切地会返回同一个模块。

比如，有一个叫 my_module.js 的模块，包含下面的内容：

```
console.log('module my_module initializing...');
module.exports = function() {
    console.log('Hi!');
};
console.log('my_module initialized.');
```

然后用下面的代码加载这个模块：

```
var myModuleInstance1 = require('./my_module');
```

它会产生下面的输出：

```
module my_module initializing...
my_module initialized
```

如果我们两次导入它：

```
var myModuleInstance1 = require('./my_module');
var myModuleInstance2 = require('./my_module');
```

输出依然是：

```
module my_module initializing...
my_module initialized
```

也就是说，模块的初始化代码仅执行了一次。当你构建自己的模块时，如果模块的初始化代码里含有可能产生副作用的代码，一定要特别注意这个特性。

小结

Node 取消了 JavaScript 的默认全局作用域，转而采用 CommonJS 模块系统，这样你可以更好的组织你的代码，也因此避免了很多安全问题和 bug。可以使用 require 函数来加载核心模块，第三方模块，或从文件及目录加载你自己的模块

还可以用相对路径或者绝对路径来加载非核心模块，如果把模块放到了 node_modules 目录下或者对于用 NPM 安装的模块，你还可以直接使用模块名来加载。

译者注：

建议读者把官方文档的模块章节阅读一遍，个人感觉比作者讲得更清晰明了，而且还附加了一个非常具有代表性的例子，对理解 Node 模块加载会很有很大帮助。下面把那个例子也引用过来：

用 require(X) 加载路径 Y 下的模块

1. *如果 X 是核心模块,*
   a. *加载并返回核心模块*
   b. *结束*
2. *如果 X 以 './' or '/' or '../ 开始'*
   a. *LOAD_AS_FILE(Y + X)*
   b. *LOAD_AS_DIRECTORY(Y + X)*
3. *LOAD_NODE_MODULES(X, dirname(Y))*
4. *抛出异常："not found"*

LOAD_AS_FILE(X)

1. *如果 X 是个文件，把 X 作为 JavaScript 脚本加载，加载完毕后结束*
2. *如果 X.js 是个文件，把 X.js 作为 JavaScript 脚本加载，加载完毕后结束*
3. *如果 X.node 是个文件，把 X.node 作为 Node 二进制插件加载，加载完毕后结束*

LOAD_AS_DIRECTORY(X)

1. *如果 X/package.json 文件存在,*
   a. *解析 X/package.json, 并查找 "main" 字段.*
   b. *另 M = X + (main 字段的值)*
   c. *LOAD_AS_FILE(M)*
2. *如果 X/index.js 文件存在，把 X/index.js 作为 JavaScript 脚本加载，加载完毕后结束*
3. *如果 X/index.node 文件存在，把 load X/index.node 作为 Node 二进制插件加载，加载完毕后结束*

LOAD_NODE_MODULES(X, START)

1. *另 DIRS=NODE_MODULES_PATHS(START)*
2. *对 DIRS 下的每个目录 DIR 做如下操作:*
   a. *LOAD_AS_FILE(DIR/X)*
   b. *LOAD_AS_DIRECTORY(DIR/X)*

NODE_MODULES_PATHS(START)

1. *另 PARTS = path split(START)*
2. *另 ROOT = index of first instance of "node_modules" in PARTS, or 0*
3. *另 I = count of PARTS - 1*
4. *另 DIRS = []*
5. *while I > ROOT,*
    a. *如果 PARTS[I] = "node_modules" 则继续后续操作，否则下次循环*
    c. *DIR = path join(PARTS[0 .. I] + "node_modules")*
    b. *DIRS = DIRS + DIR*
    c. *另 I = I - 1*
6. *返回 DIRS*

译者注：

本文对应原文第二部分第三章：Node Core API Basics：Loading Modules

本系列文章列表和翻译进度，请移步：Node.js 高级编程：用 Javascript 构建可伸缩应用（〇）