概述

Node 应用由模块组成，采用 CommonJS 模块规范。

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。

```
// example.js var x = 5; var addX = function
(value) { return value + x; };
```

上面代码中，变量`x`和函数`addX`，是当前文件`example.js`私有的，其他文件不可见。

如果想在多个文件分享变量，必须定义为`global`对象的属性。

```
global.warning = true;
```

上面代码的`warning`变量，可以被所有文件读取。当然，这样写法是不推荐的。

CommonJS规范规定，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。加载某个模块，其实是加载该模块的`module.exports`属性。

```
var x = 5; var addX = function (value) { return
value + x; }; module.exports.x = x;
module.exports.addX = addX;
```

上面代码通过`module.exports`输出变量`x`和函数`addX`。

`require`方法用于加载模块。

```
var example = require('./example.js');
console.log(example.x); // 5
console.log(example.addX(1)); // 6
```

`require`方法的详细解释参见《Require命令》一节。

CommonJS模块的特点如下。

- 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。

module对象

Node内部提供一个`Module`构建函数。所有模块都是`Module`的实例。

```
function Module(id, parent) { this.id = id;
this.exports = {}; this.parent = parent; // ...
```

每个模块内部，都有一个`module`对象，代表当前模块。它有以下属性。

- `module.id` 模块的识别符，通常是带有绝对路径的模块文件名。
- `module.filename` 模块的文件名，带有绝对路径。
- `module.loaded` 返回一个布尔值，表示模块是否已经完成加载。
- `module.parent` 返回一个对象，表示调用该模块的模块。
- `module.children` 返回一个数组，表示该模块要用到的其他模块。
- `module.exports` 表示模块对外输出的值。

下面是一个示例文件，最后一行输出module变量。

```
// example.js var jquery = require('jquery');
exports.$ = jquery; console.log(module);
```

执行这个文件，命令行会输出如下信息。

```
{ id: '.', exports: { '$': [Function] }, parent:
null, filename: '/path/to/example.js', loaded:
false, children: [ { id:
'/path/to/node_modules/jquery/dist/jquery.js',
exports: [Function], parent: [Circular],
filename:
'/path/to/node_modules/jquery/dist/jquery.js',
loaded: true, children: [], paths: [Object] } ],
paths: [ '/home/user/deleted/node_modules',
'/home/user/node_modules', '/home/node_modules',
'/node_modules' ] }
```

如果在命令行下调用某个模块，比如`node something.js`，那么`module.parent`就是`null`。如果是在脚本之中调用，比如`require('./something.js')`，那么`module.parent`就是调用它的模块。利用这一点，可以判断当前模块是否为入口脚本。

```
if (!module.parent) { // ran with `node something.js` app.listen(8088, function() { console.log('app listening on port 8088'); }) } else { // used with `require('/.something.js')` module.exports = app; }
```

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。

```
var EventEmitter = require('events').EventEmitter; module.exports = new EventEmitter(); setTimeout(function() { module.exports.emit('ready'); }, 1000);
```

上面模块会在加载后1秒后，发出ready事件。其他文件监听该事件，可以写成下面这样。

```
var a = require('./a'); a.on('ready', function() { console.log('module a is ready'); });
```

exports变量

为了方便，Node为每个模块提供一个exports变量，指向module.exports。这等同在每个模块头部，有一行这样的命令。

```
var exports = module.exports;
```

造成的结果是，在对外输出模块接口时，可以向exports对象添加方法。

```
exports.area = function (r) { return Math.PI * r * r; }; exports.circumference = function (r) { return 2 * Math.PI * r; };
```

注意，不能直接将exports变量指向一个值，因为这样等于切断了`exports`与`module.exports`的联系。

```
exports = function(x) {console.log(x)};
```

上面这样的写法是无效的，因为`exports`不再指向`module.exports`了。

下面的写法也是无效的。

```
exports.hello = function() { return 'hello'; };

module.exports = 'Hello world';
```

上面代码中，`hello`函数是无法对外输出的，因为`module.exports`被重新赋值了。这意味着，如果一个模块的对外接口，就是一个单一的值，不能使用`exports`输出，只能使用`module.exports`输出。

```
module.exports = function (x){ console.log(x);};
```

如果你觉得，`exports`与`module.exports`之间的区别很难分清，一个简单的处理方法，就是放弃使用`exports`，只使用`module.exports`。

AMD规范与CommonJS规范的兼容性

CommonJS规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。AMD规范则是非同步加载模块，允许指定回调函数。由于Node.js主要用于服务器编程，模块文件一般都已经存在于本地硬盘，所以加载起来比较快，不用考虑非同步加载的方式，所以CommonJS规范比较适用。但是，如果是浏览器环境，要从服务器端加载模块，这时就必须采用非同步模式，因此浏览器端一般采用AMD规范。

AMD规范使用define方法定义模块，下面就是一个例子：

```
define(['package/lib'], function(lib){ function

foo(){ lib.log('hello world!'); } return { foo:

foo }; });
```

AMD规范允许输出的模块兼容CommonJS规范，这时`define`方法需要写成下面这样：

```
define(function (require, exports, module){ var

someModule = require("someModule"); var

anotherModule = require("anotherModule");

someModule.doTehAwesome();

anotherModule.doMoarAwesome(); exports.asplode =

function (){ someModule.doTehAwesome();

anotherModule.doMoarAwesome(); }; });
```

require命令

基本用法

Node使用CommonJS模块规范，内置的`require`命令用于加载模块文件。

`require`命令的基本功能是，读入并执行一个JavaScript文件，然后返回该模块的exports对象。如果没有发现指定模块，会报错。

```
// example.js var invisible = function () {
console.log("invisible"); } exports.message =
"hi"; exports.say = function () {
console.log(message); }
```

运行下面的命令，可以输出exports对象。

```
var example = require('./example.js'); example //
{ // message: "hi", // say: [Function] // }
```

如果模块输出的是一个函数，那就不能定义在exports对象上面，而要定义在`module.exports`变量上面。

```
module.exports = function () { console.log("hello
world") } require('./example2.js')()
```

上面代码中，require命令调用自身，等于是执行`module.exports`，因此会输出hello world。

加载规则

`require`命令用于加载文件，后缀名默认为`.js`。

```
var foo = require('foo'); // 等同于 var foo =
require('foo.js');
```

根据参数的不同格式，`require`命令去不同路径寻找模块文件。

（1）如果参数字符串以“/”开头，则表示加载的是一个位于绝对路径的模块文件。比如，`require('/home/marco/foo.js')`将加载`/home/marco/foo.js`。

（2）如果参数字符串以“./”开头，则表示加载的是一个位于相对路径（跟当前执行脚本的位置相比）的模块文件。比如，`require('./circle')`将加载当前脚本同一目录的`circle.js`。

（3）如果参数字符串不以“./“或”/“开头，则表示加载的是一个默认提供的核心模块（位于Node的系统安装目录中），或者一个位于各级node_modules目录的已安装模块（全局安装或局部安装）。

举例来说，脚本`/home/user/projects/foo.js`执行了`require('bar.js')`命令，Node会依次搜索以下文件。

- /usr/local/lib/node/bar.js
- /home/user/projects/node_modules/bar.js
- /home/user/node_modules/bar.js
- /home/node_modules/bar.js
- /node_modules/bar.js

这样设计的目的是，使得不同的模块可以将所依赖的模块本地化。

（4）如果参数字符串不以“./“或”/“开头，而且是一个路径，比如`require('example-module/path/to/file')`，则将先找到`example-module`的位置，然后再以它为参数，找到后续路径。

（5）如果指定的模块文件没有发现，Node会尝试为文件名添加`.js`、`.json`、`.node`后，再去搜索。`.js`件会以文本格式的JavaScript脚本文件解析，`.json`文件会以JSON格式的文本文件解析，`.node`文件会以编译后的二进制文件解析。

（6）如果想得到`require`命令加载的确切文件名，使用`require.resolve()`方法。

目录的加载规则

通常，我们会把相关的文件会放在一个目录里面，便于组织。这时，最好为该目录设置一个入口文件，让`require`方法可以通过这个入口文件，加载整个目录。

在目录中放置一个`package.json`文件，并且将入口文件写入`main`字段。下面是一个例子。

```
// package.json { "name" : "some-library", "main"
: "./lib/some-library.js" }
```

`require`发现参数字符串指向一个目录以后，会自动查看该目录的`package.json`文件，然后加载`main`字段指定的入口文件。如果`package.json`文件没有`main`字段，或者根本就没有`package.json`文件，则会加载该目录下的`index.js`文件或`index.node`文件。

模块的缓存

第一次加载某个模块时，Node会缓存该模块。以后再加载该模块，就直接从缓存取出该模块的`module.exports`属性。

```
require('./example.js');
```

```
require('./example.js').message = "hello";
```

```
require('./example.js').message // "hello"
```

上面代码中，连续三次使用`require`命令，加载同一个模块。第二次加载的时候，为输出的对象添加了一个`message`属性。但是第三次加载的时候，这个message属性依然存在，这就证明`require`命令并没有重新加载模块文件，而是输出了缓存。

如果想要多次执行某个模块，可以让该模块输出一个函数，然后每次`require`这个模块的时候，重新执行一下输出的函数。
所有缓存的模块保存在`require.cache`之中，如果想删除模块的缓存，可以像下面这样写。

```
// 删除指定模块的缓存 delete
require.cache[moduleName]; // 删除所有模块的缓存
Object.keys(require.cache).forEach(function(key)
{ delete require.cache[key]; })
```

注意，缓存是根据绝对路径识别模块的，如果同样的模块名，但是保存在不同的路径，`require`命令还是会重新加载该模块。

环境变量NODE_PATH

Node执行一个脚本时，会先查看环境变量`NODE_PATH`。它是一组以冒号分隔的绝对路径。在其他位置找不到指定模块时，Node会去这些路径查找。
可以将NODE_PATH添加到`.bashrc`。

```
export NODE_PATH="/usr/local/lib/node"
```

所以，如果遇到复杂的相对路径，比如下面这样。

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

有两种解决方法，一是将该文件加入`node_modules`目录，二是修改`NODE_PATH`环境变量，`package.json`文件可以采用下面的写法。

```
{ "name": "node_path", "version": "1.0.0",
"description": "", "main": "index.js", "scripts":
{ "start": "NODE_PATH=lib node index.js" },
"author": "", "license": "ISC" }
```

`NODE_PATH`是历史遗留下来的一个路径解决方案，通常不应该使用，而应该使用`node_modules`目录机制。

模块的循环加载

如果发生模块的循环加载，即A加载B，B又加载A，则B将加载A的不完整版本。

```
// a.js exports.x = 'a1'; console.log('a.js ',
require('./b.js').x); exports.x = 'a2'; // b.js
```

```
exports.x = 'b1'; console.log('b.js ',
require('./a.js').x); exports.x = 'b2'; //
main.js console.log('main.js ',
require('./a.js').x); console.log('main.js ',
require('./b.js').x);
```

上面代码是三个JavaScript文件。其中，a.js加载了b.js，而b.js又加载a.js。这时，Node返回a.js的不完整版本，所以执行结果如下。

```
$ node main.js b.js a1 a.js b2 main.js a2 main.js
b2
```

修改main.js，再次加载a.js和b.js。

```
// main.js console.log('main.js ',
require('./a.js').x); console.log('main.js ',
require('./b.js').x); console.log('main.js ',
require('./a.js').x); console.log('main.js ',
require('./b.js').x);
```

执行上面代码，结果如下。

```
$ node main.js b.js a1 a.js b2 main.js a2 main.js
b2 main.js a2 main.js b2
```

上面代码中，第二次加载a.js和b.js时，会直接从缓存读取exports属性，所以a.js和b.js内部的console.log语句都不会执行了。

require.main

`require`方法有一个`main`属性，可以用来判断模块是直接执行，还是被调用执行。

直接执行的时候（`node module.js`），`require.main`属性指向模块本身。

```
require.main === module // true
```

调用执行的时候（通过`require`加载该脚本执行），上面的表达式返回false。

模块的加载机制

CommonJS模块的加载机制是，输入的是被输出的值的拷贝。也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。请看下面这个例子。

下面是一个模块文件`lib.js`。

```
// lib.js var counter = 3; function incCounter()
{ counter++; } module.exports = { counter:
counter, incCounter: incCounter, };
```

上面代码输出内部变量`counter`和改写这个变量的内部方法`incCounter`。
然后，加载上面的模块。

```
// main.js var counter =
require('./lib').counter; var incCounter =
require('./lib').incCounter;
console.log(counter); // 3 incCounter();
console.log(counter); // 3
```

上面代码说明，`counter`输出以后，`lib.js`模块内部的变化就影响不到`counter`了。

require的内部处理流程

`require`命令是CommonJS规范之中，用来加载其他模块的命令。它其实不是一个全局命令，而是指向当前模块的`module.require`命令，而后者又调用Node的内部命令`Module._load`。

```
Module._load = function(request, parent, isMain)
{ // 1. 检查 Module._cache，是否缓存之中有指定模块 //
2. 如果缓存之中没有，就创建一个新的Module实例 // 3. 将它
保存到缓存 // 4. 使用 module.load() 加载指定的模块文
件， // 读取文件内容之后，使用 module.compile() 执行文
件代码 // 5. 如果加载/解析过程报错，就从缓存删除该模块 //
6. 返回该模块的 module.exports };
```

上面的第4步，采用`module.compile()`执行指定模块的脚本，逻辑如下。

```
Module.prototype._compile = function(content,
filename) { // 1. 生成一个require函数，指向
module.require // 2. 加载其他辅助方法到require // 3.
```

```
将文件内容放到一个函数之中，该函数可调用 require // 4. 执行该函数 };
```

上面的第1步和第2步，`require`函数及其辅助方法主要如下。

- `require()`：加载外部模块
- `require.resolve()`：将模块名解析到一个绝对路径
- `require.main`：指向主模块
- `require.cache`：指向所有缓存的模块
- `require.extensions`：根据文件的后缀名，调用不同的执行函数

一旦`require`函数准备完毕，整个所要加载的脚本内容，就被放到一个新的函数之中，这样可以避免污染全局环境。该函数的参数包括`require`、`module`、`exports`，以及其他一些参数。

```
(function (exports, require, module, __filename, __dirname) { // YOUR CODE INJECTED HERE! });
```

`Module._compile`方法是同步执行的，所以`Module._load`要等它执行完成，才会向用户返回`module.exports`的值。