# 一 Node全局对象

## 1 全局对象global

js中定义全局变量，可以通过window顶层对象访问（即全局变量是顶层对象的属性）：

|  |
| --- |
| **var *a*** = 100; ***console***.log(**window**.***a***); |

Node没有window这个说法，也就是其没有这个顶层对象，Node自身的顶层对象是 global。

顶层对象：

JS---------window

node-----global

定义全局变量方式：

方式一： a = 3;

方式二： global.a = 3;

如果使用var a = 3; global是无法访问到的。

## 2 模块属性\_\_filename \_\_dirname

\_\_filename：

返回当前模块的文件的解析后的绝对路径，该属性并非全局的（不能global.\_\_filename），而每个模块都有该属性，所以直接可以使用。

\_\_dirname：

返回当前模块文件所在目录解析后的绝对路径，返回的是文件夹。同\_\_filename一样，其也非全局对象，而是每个模块都有的属性。

## 3 process对象

### 3.1 process常见属性

全局对象process在任何地方都能被访问到，提供对当前运行的程序的进程进行访问和控制。

访问：

process 或者 global.process

常用参数：

argv： 一组包含命令行参数的数组，可以让进程中带入一些参数

execPath： 开启当前进程的绝对路径

env： 返回用户的环境信息

pid： 返回当前进程的pid

title： 当前进程的显示名称（Getter/Setter）

exit(): 退出进程

### 3.2 IO操作

stdin和stdout提供操作输入数据和输出数据的方法。

stdout 标准输出流

|  |
| --- |
| **function** *log*(data) {  process.**stdout**.write(data); } *log*(**'你好'**); |

stdin 标准输入流

|  |
| --- |
| *//默认情况下，输入流失关闭的，要监听处理的话需要先开启输入流*  process.**stdin**.resume(); process.**stdin**.on(**'data'**,**function** (chunk) { *//on是监听函数* ***console***.log(**'用户输入了：'** + chunk); }); |

# 三 Node模块

## 1 模块初识 http模块

let http = require('http');  
let server = http.createServer(function (req,res) {  
 res.writeHead(200,{  
 'Content-Type':'text/plain;charset=UTF8'  
 });  
 res.end('hello world');  
}).listen(8000);

## 2 Node的模块定义与导出

● **狭义**的说，每一个JavaScript文件都是一个模块；而多个JavaScript文件之间可以相互require，他们共同实现了一个功能，他们整体对外，又称为一个**广义**上的模块。

● **Node.js中，一个JavaScript文件中定义的变量、函数，都只在这个文件内部有效**。当需要从此JS文件外部引用这些变量、函数时，必须使用exports对象进行暴露。使用者要用require()命令引用这个JS文件。

示例：foo.js中的msg是foo内部才有作用域的变量。**如果别人想用这个变量，需要exports暴露**。

|  |
| --- |
| var msg = "你好";  exports.text= msg; |

使用者：使用者用foo来接收exports对象，即上述文件中的exports变量

|  |
| --- |
| var **foo** = require("./foo.js");  console.log(foo.text); |

● 一个JavaScript文件，可以向外exports无数个变量、函数。但是require的时候，仅仅需要require这个JS文件一次。使用的它的变量、函数的时候，用点语法即可。所以，无形之中，增加了一个顶层命名空间。

|  |
| --- |
| var msg = "你好";  var info = "呵呵";  function showInfo(){  console.log(info);  }  exports.msg = msg;  exports.info = info;  exports.showInfo = showInfo; |

|  |
| --- |
| var foo = require("./test/foo.js"); |

|  |
| --- |
| console.log(foo.msg);  console.log(foo.info);  foo.showInfo(); |

● 可以将一个JavaScript文件中，描述一个类：

也就是说，js文件和js文件之间有两种合作的模式：

1） 某一个js文件中，提供了函数：exports.msg=msg;

2） 某一个js文件，描述了一个类：module.exports = Person;

## 3 模块引入使用

● 模块分为两类：

原生模块：Node启动时加载。

文件模块：动态加载，由原生模块的module实现，使用require来调用。

不同类型的模块引入方式也不相同：

引入核心模块： require(‘fs’);

引入自定义模块： require(‘./mytest/test1.js’);

注意：引入自定义模块可以使用绝对路径或者相对路径，但是相对路径必须以 ./ 开始，因为在Node中，没有 ./ 会被识别为Node的核心模块。

**理解：其实require()别的js文件的时候，就是执行那个js文件。**

注意：

**require()中的路径，是从当前这个js文件出发，找到别人。而fs是从命令提示符找到别人。**

示例：

**桌面上有一个a.js， test文件夹中有b.js、c.js、1.txt，**a要引用b：

|  |
| --- |
| var b = require(“./test/b.js”); |

b要引用c：

|  |
| --- |
| var b = require(“./c.js”); |

但是，fs等其他的模块用到路径的时候，都是相对于cmd命令光标所在位置。所以，在b.js中想读1.txt文件，推荐用绝对路径：（fs.readFile的fs是从光标所在的位置查找，由于b.js并不是自己引入直接执行的，而是b自己内部执行了read函数，那么如果不加\_\_dirname就会出现错误！）

|  |
| --- |
| fs.readFile(**\_\_dirname** + "/1.txt",function(err,data){  if(err) { throw err; }  console.log(data.toString());  }); |

总结：当创建的js文件要被另外一个文件引入使用，而这个js文件要使用到类似readFile这种带路径的方法时，需要\_\_dirname。

## 4 模块加载机制

1. 首先按照加载的模块的文件名进行查找；
2. 没找到，在模块文件名后加上 .js 后缀进行查找；
3. 没找到，在文件名后加上 .json 后缀查找；
4. 没找到，在文件名后加上 .node 后缀查找；
5. 抛出错误。

● node\_modules文件夹并不一定在同级目录里面，在任何直接祖先级目录中，都可以。甚至可以放到NODE\_PATH环境变量的文件夹中。这样做的好处稍后你将知道：分享项目的时候，不需要带着modules一起给别人。

● 我们可以使用文件夹来管理模块，比如

|  |
| --- |
| 1. var bar = require("bar"); 我们称bar为入口文件 |

那么Node.js将会去寻找node\_modules目录下的bar文件夹中的index.js去执行。

NodeJS会对原生、文件模块都进行缓存，在第二次require的时候，不会重复开销去加载模块，只需要从缓存中读取响应模块的数据。

如果没有index文件就会报错，找不到入口文件。不过我们可以在bar文件夹中新建一个package.json文件，也可以解决这个问题。

每一个模块文件夹中，都可以写一个package.json文件，这个文件的名字不能改。node将自动读取里面的配置。有一个main项，就是默认入口文件：

|  |
| --- |
| 1. { 2. "name": "kaoladebar", 3. "version": "1.0.1", 4. **"main" : "app.js"** 5. } |

## 5 module.exports 和 exports

在一个模块中，通过var定义了变量，其作用域只是当前模块，外部不能直接访问，如果想让其能被外部访问，可以：

1. 把变量作为global对象的一个属性；无论这个变量在哪个文件都可以访问。

2-使用模块对象 module

不推荐第一种做法。

module对象保存提供了当前模块的一些信息，module对象下有一个export对象，可以通过这个对象，把一个模块中的局部变量对外暴露，让外部可以访问。

案例：自定义一个tool工具模块，向外暴露属性方法

tool.js文件：

|  |
| --- |
| **var** a = 100; module.exports.b = a; *//在模块作用域中，还有一个内置对象 exports ，他其实就是module.exports* exports.c = a; |

test.js文件：

|  |
| --- |
| **var** tool = *require*(**'./tool'**); *//其实加载的是模块tool中的 module.exports* ***console***.log(tool); ***console***.log(tool.b); ***console***.log(tool.c); ***console***.log(module.exports === exports); |

注意： 不要轻易改变 module.exports 和 exports 的指向，如：exprots = [1,2,3];

一旦改变，再向外暴露其他成员，就不能被访问了。

exports不能代替module.exports，因为exports只是module对象的下的一个属性而已。其实所有的exports都是通过module.exports传递的。

module.exports 可以单独返回一个数据类型如：数组、字符串、数字，而exports不行。

如果创建了一个既有exports又有module.exports的模块，那么返回module.exports，expot会被忽略。因为exports只是对module.exports的一个全局引用，最初被定义为一个可以添加属性的孔对象。

## 6 依赖管理与npm

我们可以用package.json来管理依赖。

在cmd中，使用npm init可以初始化一个package.json文件，用回答问题的方式生成一个新的package.json文件。安装的依赖信息（版本等）都来自于每个node\_module中的包额package文件。

npm是Node自带的一个包管理器，用来安装Node的各种第三方模块。

npm install \*\*\*

npm uninstall \*\*\*

npm install -g \*\*\* 将包安装在全局