# 01_模块和包

## 什么是模块

模块是 Node.js 应用程序的基本组成部分,文件和模块是一一对应的。换言之,一个 Node.js 文件就是一个模块,这个文件可能是 JavaScript 代码、JSON 或者编译过的 C/C++ 扩展。
在前面章节的例子中,我们曾经用到了 `var http = require('http')`,其中 `http` 是 Node.js 的一个核心模块,其内部是用 C++ 实现的,外部用 JavaScript 封装。我们通过 `require` 函数获取了这个模块,然后才能使用其中的对象。

## 创建及加载模块

### 创建模块

在 Node.js 中,创建一个模块非常简单,因为一个文件就是一个模块,我们要关注的问 题仅仅在于如何在其他文件中获取这个模块。Node.js 提供了 `exports` 和 `require` 两个对 象,其中 `exports` 是模块公开的接口,`require` 用于从外部获取一个模块的接口,即所获 取模块的 `exports` 对象。

让我们以一个例子来了解模块。创建一个 module.js 的文件,内容是:

```
//module.js
var name;
exports.setName = function(thyName) { name = thyName;
};
exports.sayHello = function() { console.log('Hello ' +
name);
};
```

在同一目录下创建 getmodule.js,内容是:

```
//getmodule.js
var myModule = require('./module');
myModule.setName('BYVoid');
myModule.sayHello();
```

运行node getmodule.js,结果是:

```
Hello BYVoid
```

在以上示例中,module.js 通过 `exports` 对象把 `setName` 和 `sayHello` 作为模块的访 问接口,在 getmodule.js 中通过 `require('./module')` 加载这个模块,然后就可以直接访 问 module.js 中 `exports` 对象的成员函数了。
这种接口封装方式比许多语言要简洁得多,同时也不失优雅,未引入违反语义的特性,符合传统的编程逻辑。在这个基础上,我们可以构建大型的应用程序,npm 提供的上万个模 块都是通过这种简单的方式搭建起来的。

### 单次加载

上面这个例子有点类似于创建一个对象,但实际上和对象又有本质的区别,因为 `require` 不会重复加载模块,也就是说无论调用多少次 `require`,获得的模块都是同一个。 我们在 getmodule.js 的基础上稍作修改:

```
//loadmodule.js
var hello1 = require('./module'); hello1.setName('BYVoid');
var hello2 = require('./module'); hello2.setName('BYVoid
2');
    hello1.sayHello();
```

运行后发现输出结果是 `Hello BYVoid 2`,这是因为变量 `hello1` 和 `hello2` 指向的是 同一个实例,因此 `hello1.setName` 的结果被 `hello2.setName` 覆盖,最终输出结果是 由后者决定的。

### 覆盖 `exports`

有时候我们只是想把一个对象封装到模块中,例如:

```
//singleobject.js function Hello() {
var name;
this.setName = function (thyName) { name = thyName;
};
    this.sayHello = function () { console.log('Hello ' +
name);
}; };
 exports.Hello = Hello;
```

此时我们在其他文件中需要通过 `require('./singleobject').Hello` 来获取Hello 对象,这略显冗余,可以用下面方法稍微简化:

```
//hello.js
function Hello() { var name;
    this.setName = function(thyName) {
name = thyName;
};
this.sayHello = function() { console.log('Hello ' + name);
}; };
    module.exports = Hello;
```

这样就可以直接获得这个对象了:

```
//gethello.js
var Hello = require('./hello');
hello = new Hello(); hello.setName('BYVoid');
hello.sayHello();
```

注意,模块接口的唯一变化是使用 `module.exports = Hello` 代替了 `exports.Hello= Hello`。在外部引用该模块时,其接口对象就是要输出的 `Hello` 对象本身,而不是原先的 `exports`。

事实上,`exports` 本身仅仅是一个普通的空对象,即 `{}`,它专门用来声明接口,本 质上是通过它为模块闭包`的内部建立了一个有限的访问接口。因为它没有任何特殊的地方, 所以可以用其他东西来代替,譬如我们上面例子中的 `Hello` 对象。

警告: 不可以通过对 `exports` 直接赋值代替对 `module.exports` 赋值。`exports` 实际上只是一个和 `module.exports` 指向同一个对象的变量, 它本身会在模块执行结束后释放,但 `module` 不会,因此只能通过指定 `module.exports` 来改变访问接口。

## 创建包

包是在模块基础上更深一步的抽象,Node.js 的包类似于 C/C++ 的函数库或者 Java/.Net 的类库。它将某个独立的功能封装起来,用于发布、更新、依赖管理和版本控制。Node.js 根 据 CommonJS 规范实现了包机制,并发了 npm来解决包的发布和获取需求。

Node.js 的包是一个目录,其中包含一个 JSON 格式的包说明文件 package.json。严格符 合 CommonJS 规范的包应该具备以下特征:

- ? package.json 必须在包的顶层目录下;
- ? 二进制文件应该在 `bin` 目录下;
- ? JavaScript 代码应该在 `lib` 目录下;
- ? 文档应该在 `doc` 目录下;
- ? 单元测试应该在 `test` 目录下。

Node.js 对包的要求并没有这么严格,只要顶层目录下有 package.json,并符合一些规范 即可。当然为了提高兼容性,我们还是建议你在制作包的时候,严格遵守 CommonJS 规范。

### 作为文件夹的模块

模块与文件是一一对应的。文件不仅可以是 JavaScript 代码或二进制代码,还可以是一 个文件夹。最简单的包,就是一个作为文件夹的模块。下面我们来看一个例子,建立一个叫 做 somepackage 的文件夹,在其中创建 `index.js`,内容如下:

```
//somepackage/index.js
exports.hello = function()
{ console.log('Hello.');
};
```

然后在 somepackage 之外建立 `getpackage.js`,内容如下:

```
//getpackage.js
var somePackage = require('./somepackage');
somePackage.hello();
```

运行 `node getpackage.js`,控制台将输出结果 `Hello.`。

我们使用这种方法可以把文件夹封装为一个模块,即所谓的包。包通常是一些模块的集 合,在模块的基础上提供了更高层的抽象,相当于提供了一些固定接口的函数库。通过定制 `package.json`,我们可以创建更复杂、更完善、更符合规范的包用于发布。

### package.json

在前面例子中的 somepackage 文件夹下,我们创建一个叫做 `package.json` 的文件,内容如 下所示:

```
{
  "main" : "./lib/interface.js"
}
```

然后将 `index.js` 重命名为 `interface.js` 并放入 `lib` 子文件夹下。以同样的方式再次调用这个 包,依然可以正常使用。

Node.js 在调用某个包时,会首先检查包中 `package.json` 文件的 `main` 字段,将其作为包的接口模块,如果 `package.json` 或 `main` 字段不存在,会尝试寻找 `index.js` 或 `index.node` 作 为包的接口。

`package.json` 是 CommonJS 规定的用来描述包的文件,完全符合规范的 `package.json` 文 件应该含有以下字段:

- ? `name`:包的名称,必须是唯一的,由小写英文字母、数字和下划线组成,不能包含 空格。
- ? `description`:包的简要说明。
- ? `version`:符合语义化版本识别[1]规范的版本字符串。
- ? `keywords`:关键字数组,通常用于搜索。
- ? `maintainers`:维护者数组,每个元素要包含 `name`、`email`(可选)、`web`(可选)字段。
- ? `contributors`:贡献者数组,格式与`maintainers`相同。包的作者应该是贡献者数组的第一个元素。
- ? `bugs`:提交bug的地址,可以是网址或者电子邮件地址。
- ? `licenses`:许可证数组,每个元素要包含 `type`(许可证的名称)和 `url`(链接到许可证文本的地址)字段。
- ? `repositories`:仓库托管地址数组,每个元素要包含 `type`(仓库的类型,如 `git`)、
- ? `url`(仓库的地址)和 `path`(相对于仓库的路径,可选)字段。
- ? `dependencies`:包的依赖,一个关联数组,由包名称和版本号组成。

## Node.js 包管理器

Node.js包管理器,即npm是 Node.js 官方提供的包管理工具',它已经成了 Node.js 包的 标准发布平台,用于 Node.js 包的发布、传播、依赖控制。npm 提供了命令行工具,使你可 以方便地下载、安装、升级、删除包,也可以让你作为开发者发布并维护包。

### 获取一个包

使用 npm 安装包的命令格式为:

```
npm [install/i] [package_name]
```

例如你要安装 `express`,可以在命令行运行:

```
$ npm install express
或者:
    $ npm i express
```

此时 `express` 就安装成功了,并且放置在当前目录的 `node_modules` 子目录下。npm 在 获取 `express` 的时候还将自动解析其依赖,并获取 `express` 依赖的 `mime`、`mkdirp`、`qs` 和 `connect`。

### 本地模式和全局模式

npm在默认情况下会从http://npmjs.org搜索或下载包,将包安装到当前目录的node_modules 子目录下。

在使用 npm 安装包的时候,有两种模式:本地模式和全局模式。默认情况下我们使用 `npm install`命令就是采用本地模式,即把包安装到当前目录的 `node_modules` 子目录下。Node.js 的`require`在加载模块时会尝试搜寻 `node_modules`子目录,因此使用npm本地模式安装 的包可以直接被引用。

npm 还有另一种不同的安装模式被成为全局模式,使用方法为:

```
npm [install/i] -g [package_name]
```

为什么要使用全局模式呢?多数时候并不是因为许多程序都有可能用到它,为了减少多 重副本而使用全局模式,而是因为本地模式不会注册 `PATH` 环境变量。举例说明,我们安装 supervisor 是为了在命令行中运行它,譬如直接运行 supervisor script.js,这时就需要在 `PATH` 环境变量中注册 supervisor。npm 本地模式仅仅是把包安装到 `node_modules` 子目录下,其中 的 `bin` 目录没有包含在 `PATH` 环境变量中,不能直接在命令行中调用。而当我们使用全局模 式安装时,npm 会将包安装到系统目录,譬如 `/usr/local/lib/node_modules/`,同时 `package.json` 文 件中 `bin` 字段包含的文件会被链接到 `/usr/local/bin/`。`/usr/local/bin/` 是在`PATH` 环境变量中默认 定义的,因此就可以直接在命令行中运行 `supervisor script.js`命令了。

本地模式和全局模式的特点

| 模　　式 | 可通过 `require` 使用 | 注册`PATH` |
|---|---|---|
| 本地模式 | 是 | 否 |
| 全局模式 | 否 | 是 |

总而言之,当我们要把某个包作为工程运行时的一部分时,通过本地模式获取,如果要 在命令行下使用,则使用全局模式安装。

### 包的发布

npm 可以非常方便地发布一个包,比 pip、gem、pear 要简单得多。在发布之前,首先 需要让我们的包符合npm的规范,npm有一套以CommonJS为基础包规范,但与CommonJS 并不完全一致,其主要差别在于必填字段的不同。通过使用 `npm init` 可以根据交互式问答 产生一个符合标准的 package.json,例如创建一个名为 byvoidmodule 的目录,然后在这个 目录中运行`npm init`:

这样就在 byvoidmodule 目录中生成一个符合 npm 规范的 package.json 文件。创建一个 index.js 作为包的接口,一个简单的包就制作完成了。

在发布前,我们还需要获得一个账号用于今后维护自己的包,使用 `npm adduser` 根据 提示输入用户名、密码、邮箱,等待账号创建完成。完成后可以使用 `npm whoami` 测验是 否已经取得了账号。

接下来,在 package.json 所在目录下运行 `npm publish`,稍等片刻就可以完成发布了。 打开浏览器,访问 http://search.npmjs.org/ 就可以找到自己刚刚发布的包了。现在我们可以在 世界的任意一台计算机上使用 `npm install byvoidmodule` 命令来安装它。图 1-6 是npmjs. org上包的描述页面。

如果你的包将来有更新,只需要在 package.json 文件中修改 `version` 字段,然后重新 使用 `npm publish` 命令就行了。如果你对已发布的包不满意(比如我们发布的这个毫无意 义的包),可以使用 `npm unpublish` 命令来取消发布。