- 什么是模块；
- 如何创建并加载模块；
- 如何创建一个包；
- 如何使用包管理器 ；

3.3.1 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。

3.3.2 创建及加载模块

介绍了什么是模块之后，下面我们来看看如何创建并加载它们。

1. 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。

2. 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。

3. 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[①]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。

① 闭包是函数式编程语言的常见特性，具体说明见本书附录A。

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。

3.3.3　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，开发了 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。

1. 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。

2. 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。

① 语义化版本识别（Semantic Versioning）是由Gravatars和GitHub创始人Tom Preston-Werner提出的一套版本命名规范，最初目的是解决各式各样版本号大小比较的问题，目前被许多包管理系统所采用。

- dependencies：包的依赖，一个关联数组，由包名称和版本号组成。

下面是一个完全符合 CommonJS 规范的 package.json 示例：

```
{
  "name": "mypackage",
  "description": "Sample package for CommonJS. This package demonstrates the required
     elements of a CommonJS package.",
  "version": "0.7.0",
  "keywords": [
     "package",
     "example"
  ],
  "maintainers": [
     {
        "name": "Bill Smith",
        "email": "bills@example.com",
     }
  ],
  "contributors": [
     {
        "name": "BYVoid",
        "web": "http://www.byvoid.com/"
     }
  ],
  "bugs": {
     "mail": "dev@example.com",
     "web": "http://www.example.com/bugs"
  },
  "licenses": [
     {
        "type": "GPLv2",
        "url": "http://www.example.org/licenses/gpl.html"
     }
  ],
  "repositories": [
     {
        "type": "git",
        "url": "http://github.com/BYVoid/mypackage.git"
     }
  ],
  "dependencies": {
     "webkit": "1.2",
     "ssl": {
        "gnutls": ["1.0", "2.0"],
        "openssl": "0.9.8"
     }
  }
}
```