1 总结Node.js中的模块有哪些类型。

参考答案,

Node.js中没有全局命名空间的概念，每个被require加载的JS文件或者目录都是一个模块，JS文件中定义的变量和函数都是模块内部的对象，只有通过module.exports导出的成员才能被其它模块使用；所以不会出现一个模块中定义的成员与其它模块中的成员命名冲突的问题。

Node.js中的模块分为三种类型：

(1)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，如果想加载和使用http核心模块，可以这样做：

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

(2)本地模块：

可以使用绝对路径从文件系统里加载模块：

```
01.    var myModule1 = require('d:/my_modules/my_module1');
```

或者用一个基于当前文件的相对路径：

```
01.    var myModule2 = require('../my_modules/my_module2');
02.    var myModule3 = require('./lib/my_module_3');
```

还可以使用目录的路径来加载模块：

```
01.    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。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。

(3)NPM安装的第三方模块：

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:

```
01.    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。

2 简述require('X') 加载路径Y下的一个模块的搜索顺序。

参考答案,

顺序如下：

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"

对于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

对于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

对于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)

对于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

3 分别使用同步和异步的方法把当前目录下的1.jpg复制为2.jpg。

参考答案

使用同步方法实现文件的复制，是传统的多线程服务器通常采用的方法，此方法会阻塞当前执行线程，直到文件系统返回了足够的数据，才能执行后续的操作。而并发线程过多的情况下，会增加程序设计的复杂度和CPU执行上下文切换的开销。Node.js的服务器端应用，一般只在初始化启动阶段才会使用异步文件读取（如加载配置文件），提供客户端响应过程中一般都使用异步文件操作。同步文件复制示例代码如下：

```
01.     const fs = require('fs');
02.
03.     /****同步拷贝文件****/
04.     //由于被拷贝的文件可能比较大，不推荐一次性全部读入内存
```

```
05.
06.     var buf = new Buffer(1024); //暂存文件内容的缓冲区
07.     var inputFile = fs.openSync('./1.jpg', 'r');
08.     var outputFile = fs.openSync('./2.jpg', 'w');
09.
10.     console.log('开始文件复制');
11.     var count = 0;
12.     while( (count=fs.readSync(inputFile, buf, 0, 1024, null))>0 ){
13.       console.log('COUNT:'+count)
14.       fs.writeSync(outputFile, buf, 0, count);
15.     }
16.
17.     fs.closeSync(inputFile);
18.     fs.closeSync(outputFile);
19.     console.log('文件复制完成');
```

异步文件读写可以在Node.js主线程无需等待的情况下，实现文件系统的异步IO；待耗时的IO操作完成后，会交由主线程执行指定的回调函数，处理操作的结果。异步代码如下所示：

```
01.     const fs = require('fs');
02.
03.     /****异步拷贝文件****/
04.     //由于被拷贝的文件可能比较大，不推荐一次性全部读入内存
05.
06.     var buf = new Buffer(1024); //暂存文件内容的缓冲区
07.
08.     console.log('开始复制文件')
09.     var readable = fs.createReadStream('./1.jpg');
10.     var writable = fs.createWriteStream('./2.jpg');
11.
12.     readable.on('data', function(data){
13.       writable.write(data);
14.     })
15.
16.     readable.on('end', function(){
17.       console.log('文件复制完成');
18.     })
```