Node.js模块系统

- 为了让Node.js的文件可以相互调用，Node.js提供了一个简单的模块系统。
- 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。
- Node.js的模块分为两类，一类为原生（核心）模块，一类为文件模块。
- 在文件模块中，又分为3类模块。这三类文件模块以后缀来区分，Node.js会根据后缀名来决定加载方法。
    - .js。通过fs模块同步读取js文件并编译执行。
    - .node。通过C/C++进行编写的Addon。通过dlopen方法进行加载。
    - .json。读取json文件，调用JSON.parse解析加载。
- Node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接口,即所获取模块的exports对象

require

- require函数用于在当前模块中加载和使用别的模块，传入一个模块名，返回一个模块导出对象。
- require方法接受以下几种参数的传递：
    - 如：http、fs、path等。原生模块。
    - ./mod或../mod。相对路径的文件模块。
    - /a/mod，绝对路径的文件模块。
    - mod，非原生模块的文件模块。

exports和module.exports

- exports对象是当前模块的导出对象，用于导出模块公有方法和属性。
- 别的模块通过require函数使用当前模块时得到的就是当前模块的exports对象。
- exports 变量是在模块的文件级作用域内可用的，且在模块执行之前赋值给 module.exports。
- exports是一种快捷方式，因此 module.exports.f = ... 可以更简洁地写成 exports.f = ...。

① 使用module.exports导出一个方法：

- 首先定义一个模块文件 hello.js 代码如下：

```
function hello() {
    console.log('hello');
}

/* ES6中箭头函数写法
const hello = () => {
    console.log('hello');
}
*/

module.exports = hello;
```

- 在同目录下定义一个主文件 main.js 代码如下，并运行：node main.js

```
const hello = require('./hello');
hello();

//输出：hello
```

- 上面代码定义一个hello模块，模块里定义了一个hello方法，通过替换当前模块exports对象的方式将hello方法导出。
- 在main.js中加载这个模块，得到的是一个函数，调用执行该函数，控制台打印 hello。
- 其实上面hello.js文件代码也可以写成如下格式，效果一样：

```
//等价于上面hello.js文件中的代码
module.exports = function () {
   console.log('world');
}
/*
//ES6中的箭头函数
module.exports = () => {
    console.log('hello2');
}
*/
```

② exports导出多个变量

- 当要导出多个变量怎么办呢？这个时候替换当前模块对象的方法就不实用了，我们需要用到exports对象。
- 定义个文件 myExports.js 代码如下：

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

exports.b = function () {
    console.log('b exports');
```

```
}

/*
//同上效果也可如此定义：
module.exports.a = () => {
    console.log('a exports');
}

module.exports.b = () => {
    console.log('b exports');
}
*/
```

- 同级目录下 main.js 代码如下：

```
const myExports = require('./myExports');

myExports.a();
myExports.b();

//a exports
//b exports
```

③ 导出类

- 文件：circle.js 定义一个‘圆’类

```
//定义匿名类
module.exports = class{
    //构造方法
    constructor(r){
        this.r = r;
    }

    //周长
    perimeter(){
        return 2*Math.PI*this.r;
    }

    //面积
    area(){
        return Math.PI*this.r*this.r;
    }
}
```

- 同级目录下 main.js 代码如下：

```
const Circle = require("./mod/circle");
//实例化导入的类
const c = new Circle(10);
console.log("周长：",c.perimeter());
console.log("面积：",c.area());

//输出：
//周长：  62.83185307179586
//面积：  314.1592653589793
```

④ 导出字符串

- 不仅可以替换为方法，也可以替换为字符串等。
- 文件：mystr.js

```
module.exports = 'Hello Node.js!';
```

- 同级目录下：main.js

```
const string = require('./mystr');
console.log(string);

//执行此文件输出结果：Hello Node.js!
```

⑤ 理解模块的初始化

- 一个模块中的JS代码仅在模块第一次被使用时执行一次，并在执行过程中初始化模块的导出对象。
- 之后，缓存起来的导出对象被重复利用。
- 例如：定义一个文件：count.js:

```
var i = 0;

function count() {
    return ++i;
}

exports.count = count;
```

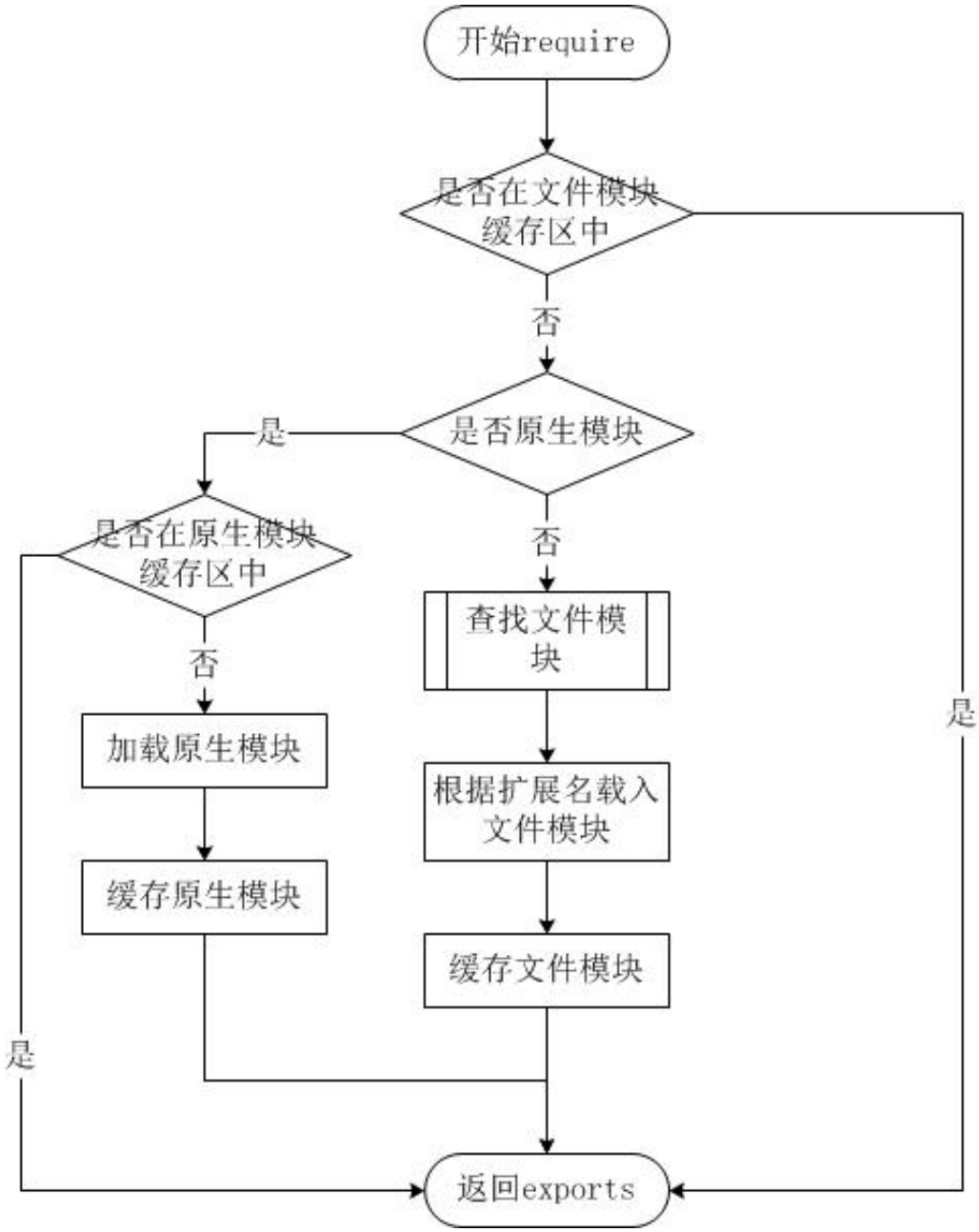
- 测试文件 main.js 代码如下：

```
const c1 = require('./count');
const c2 = require('./count');

console.log(c1.count());
console.log(c2.count());
console.log(co2.count());

//执行此文件后的输出结果
//1
//2
//3
```

- 可以看到，count.js并没有因为被require了两次而初始化两次。

⑥ 模块的加载优先级

- 由于Node.js中存在4类模块（原生模块和3种文件模块），尽管require方法极其简单，但是内部的加载却是十分复杂的，其加载优先级也各自不同，下面是require加载的逻辑图：



- 原生模块在Node.js源代码编译的时候编译进了二进制执行文件，加载的速度最快。
- 另一类文件模块是动态加载的，加载速度比原生模块慢。
- 但是Node.js对原生模块和文件模块都进行了缓存，于是在第二次require时，是不会有重复开销的。

⑦ exports与module.exports区别：

- 首先先看一个例子：文件mymod.js

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

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

- 测试文件 main.js

```
const one = require('./mymod');

//one.hello(); //执行这句话会报错one.hello is not a function

one() //打印world
```

- 其实，exports 是module.exports的一个引用，exports 的地址指向module.exports。
- 而我们的modOne.js中通过module.exports = function的方式将module.exports给替换掉了。
- 而require方法所返回的是module.exports这个实实在在的对象，但是它已经被替换成了function，这就导致了exports指向了空，所以，你所定义的exports.hello是无效的。
- 举个例子：module.exportes好比电脑硬盘目录中的一个文件，那么exportes就是此文件在电脑桌面上的一个快捷方式。
- 总之,当我们想让模块导出的是一个对象时， 使用exports 和 module.exports 都可以（但 exports 也不能重新覆盖为一个新的对象）
- 而当我们想导出非对象接口时，就必须也只能覆盖 module.exports。