```
'use strict';
console.log("hello world");
```

第一行总是写上'use strict';是因为我们总是以严格模式运行JavaScript代码，避免各种潜在陷阱。

### 运行js文件

node 文件名.js

### 使用严格模式

如果在JavaScript文件开头写上'use strict';，那么Node在执行该JavaScript时将使用严格模式。但是，在服务器环境下，如果有很多JavaScript文件，每个文件都写上'use strict';很麻烦。我们可以给Nodejs传递一个参数.

node --use_strict 文件名.js　　让Node直接为所有js文件开启严格模式：

### CommonJS规范

模块加载机制被称为CommonJS规范，在这个规范下，每个.js文件都是一个模块，它们内部各自使用的变量名和函数名都互不冲突，例如，hello.js和main.js都申明了全局变量var s = 'xxx'，但互不影响。

实现原理：Node.js也并不会增加任何JavaScript语法。实现“模块”功能的奥妙就在于JavaScript是一种函数式编程语言，它支持闭包。

如果我们把一段JavaScript代码用一个函数包装起来，这段代码的所有“全局”变量就变成了函数内部的局部变量。

```
// 准备module对象:
var module = {
    id: 'hello',
    exports: {}
};
var load = function (module) {
    // 读取的hello.js代码:
    function greet(name) {
        console.log('Hello, ' + name + '!');
    }

    module.exports = greet;
```

```
    // hello.js代码结束
    return module.exports;
};
var exported = load(module);
// 保存module:
save(module, exported);
```

通过把参数module传递给load()函数，hello.js就顺利地把一个变量传递给了Node执行环境，Node会把module变量保存到某个地方。

由于Node保存了所有导入的module，当我们用require()获取module时，Node找到对应的module，把这个module的exports变量返回，这样，另一个模块就顺利拿到了模块的输出：

## module.exports vs exports

很多时候，你会看到，在Node环境中，有两种方法可以在一个模块中输出变量：

```
exports.hello = hello;
exports.greet = greet;
var load = function (exports, module) {
    // hello.js的文件内容
    ...
    // load函数返回:
    return module.exports;
};
var exported = load(module.exports, module);
```

也就是说，默认情况下，Node准备的exports变量和module.exports变量实际上是同一个变量，并且初始化为空对象{}，于是，我们可以写：

**最终，强烈建议使用module.exports = xxx的方式来输出模块变量，这样，你只需要记忆一种方法。**