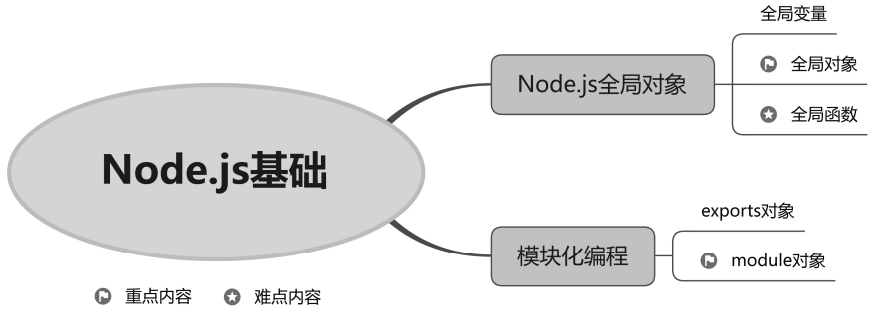
第 4 章

Node.js 基础

本章将对 Node.js 中的基础知识进行讲解，包括全局变量、全局对象、全局函数以及用于实现模块化编程的 exports 和 module 对象等内容，这些知识是学习 Node.js 应用开发的基础。

本章知识架构及重难点如下。



4.1 Node.js 全局对象

全局，即程序中任何地方都可以使用，Node.js 内置了多个全局变量、全局对象和全局函数，在开发 Node.js 程序时都可以使用，下面分别对它们进行讲解。

4.1.1 全局变量

Node.js 中的全局变量有两个，分别是__filename 和__dirname，它们的说明如下。

☑ __filename 全局变量：__filename 表示当前正在执行的脚本的文件名，包括文件所在位置的绝对路径，但该路径和命令行参数所指定的文件名不一定相同。如果在模块中，则返回的值是模块文件的路径。

☑ __dirname 全局变量：__dirname 表示当前执行的脚本所在的目录。

例如，下面代码用来分别输出 Node.js 中两个全局变量的值：

```
console.log('当前文件名：',__filename);
console.log('当前目录：',__dirname);
```

程序运行效果如图 4.1 所示。

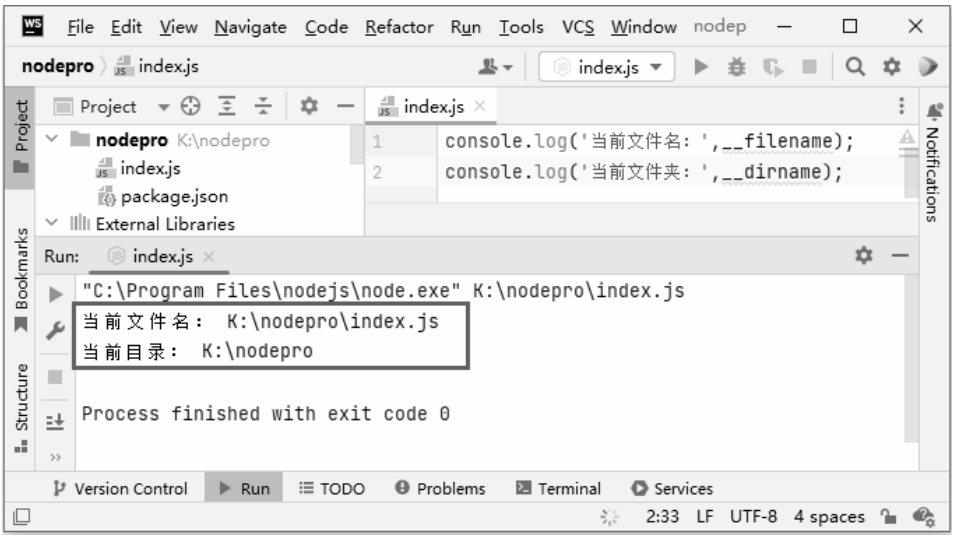


图 4.1　输出 Node.js 中全局变量的值

4.1.2　全局对象

全局对象可以在 Node.js 程序的任何地方进行访问，它可以为程序提供经常使用的特定功能。Node.js 中的全局对象如表 4.1 所示。

表 4.1　Node.js 中的全局对象

对　象	说　明
console	console 对象用于提供控制台标准输出
process	process 对象用于描述当前程序状态
exports	exports 对象是 Node.js 模块系统中公开的接口

接下来分别讲解 console 对象和 process 对象的使用，exports 对象将在 4.2 节中讲解。

1．console 对象

console 对象提供了 Node.js 控制台的标准输出，其常用方法及说明如表 4.2 所示。

表 4.2　console 对象的常见方法

方　法	说　明
log()	向标准输出流打印字符并以换行符结束，该方法可以接收若干个参数。如果只有一个参数，则输出这个参数的字符串形式；如果有多个参数，则以类似于 C 语言中 printf()命令的格式输出
time()	开始计时
timeEnd()	结束计时，并输出完成时间（从 console.time()到 console.timeEnd()之间所花费的时间）

1）console.log()方法

在 console.log()方法中，可以使用占位符输出变量（如数字变量、字符串变量和 JSON 变量等），常用的占位符如表 4.3 所示。

表 4.3　console.log()方法中的常用占位符

占 位 符	说　明
%d	输出数字变量
%s	输出字符串变量
%j	输出 JSON 变量

例如，下面代码使用%d 占位符输出一个整数值：

```
console.log('变量的值是：%d',57);
```

上面代码中，在 console.log()方法里添加了两个参数。第一个参数是字符串'变量的值是：%d'，第二个参数是数字 57。其中，%d 是占位符，会寻找后面位置的数字，因为第二个参数 57，紧紧跟在后面，所以输出结果如下：

```
变量的值是：57
```

使用 console.log()方法输出内容时，还可以有多个占位符，示例代码如下：

```
console.log('%d+%d=%d',273,52,273+52);
console.log('%d+%d=%d',273,52,273+52,52273);
console.log('%d+%d=%d & %d',273,52,273+52);
```

上面代码运行效果如下：

```
273+52=325
273+52=325 52273
273+52=325 & %d
```

观察代码可以发现，第 1 行代码中，占位符的个数是 3 个，后面的数字变量的个数也是 3 个，所以输出结果是“273+52=325”；第 2 行代码中，占位符的个数是 3 个，但是后面数字变量的个数是 4 个，输出结果“273+52=325 52273”，说明多出的数字变量 52273 原样输出；第 3 行代码中，占位符的个数是 4 个，但是后面数字变量的个数是 3 个，输出结果“273+52=325　& %d”，说明多余的占位符没有找到匹配的数字变量，只能原样输出。

使用其他占位符的示例代码如下：

```
console.log('字符串 %s','hello world','和顺序无关');
console.log('JSON %j',{name:'Node.js'});
```

上面代码运行效果如下：

```
字符串 hello world 和顺序无关
JSON {"name":"Node.js"}
```

2）console.time()方法和 console.timeEnd()方法

console.time()方法和 console.timeEnd()方法用来记录程序的执行时间段。console.time()方法用来开始计时，其参数只是起到标识的作用；console.timeEnd()方法用来结束计时，并输出程序运行所需的时间，它在显示结果时，会在标识参数后面自动添加以毫秒为单位的时间。例如，下面代码用来输出执

行 10 的阶乘运算所需要的时间：

```
//开始计时
console.time('时间');
var output = 1;
for (var i = 1; i <= 10; i++) {
        output *= i;
}
console.log('Result:', output);
//结束计时，并输出程序执行时间
console.timeEnd('时间');
```

运行效果如下：

```
Result: 3628800
时间: 11.027ms
```

说明

程序的执行时间不是固定时间，它与计算机配置有关。

2．process 对象

process 对象用于描述当前程序的状态，与 console 对象不同的是，process 对象只在 Node.js 中存在，在 JavaScript 中并不存在该对象。process 对象的常用属性及说明如表 4.4 所示。

表 4.4　process 对象的常用属性

属　性	说　明
argv	返回一个数组，由命令行执行脚本时的各个参数组成
env	返回当前系统的环境变量
version	返回当前 Node.js 的版本
versions	返回当前 Node.js 的版本号以及依赖包
arch	返回当前 CPU 的架构，如 arm 或 x64 等
platform	返回当前运行程序所在的平台系统，如 win32、linux 等
ExecPath	返回执行当前脚本的 Node 二进制文件的绝对路径
execArgv	返回一个数组，成员是命令行下执行脚本时在 Node 可执行文件与脚本文件之间的命令行参数
exitCode	进程退出时的代码，如果进程通过 process.exit()退出，不需要指定退出码
config	一个包含用来编译当前 Node 执行文件的 JavaScript 配置选项的对象。它与运行./configure 脚本生成的 config.gypi 文件相同
pid	当前进程的进程号
ppid	当前进程的父进程的进程号
title	进程名
arch	当前 CPU 的架构：arm、ia32 或者 x64
platform	运行程序所在的平台系统 darwin、freebsd、linux、sunos 或 win32
mainModule	require.main 的备选方法。不同点是，如果主模块在运行时改变，require.main 可能会继续返回老的模块。可以认为，这两者引用了同一个模块

process 对象的常用方法及说明如表 4.5 所示。

表 4.5　process 对象的常用方法

方　法	说　明
exit([code])	使用指定的 code 结束进程。如果忽略，将会使用 code0
memoryUsage()	返回一个对象，描述了 Node 进程所用的内存状况，单位为字节
uptime()	返回 Node 已经运行的秒数

【例 4.1】输出 process 对象常用属性的值。（**实例位置：资源包\源码\04\01**）

代码如下：

```
console.log('- process.env:', process.env);
console.log('- process.version:', process.version);
console.log('- process.versions:', process.versions);
console.log('- process.arch:', process.arch);
console.log('- process.platform:', process.platform);
console.log('- process.connected:', process.connected);
console.log('- process.execArgv:', process.execArgv);
console.log('- process.exitCode:', process.exitCode);
console.log('- process.mainModule:', process.mainModule);
console.log('- process.release:', process.release);
console.log('- process.memoryUsage():', process.memoryUsage());
console.log('- process.uptime():', process.uptime());
```

运行效果如下：

```
- process.env: {
        USERDOMAIN_ROAMINGPROFILE: 'DESKTOP-05BA7LG',
        LOCALAPPDATA: 'C:\\Users\\XIAOKE\\AppData\\Local',
        PROCESSOR_LEVEL: '6',
        ……
}
- process.version: v19.0.1
- process.versions: {
        node: '19.0.1',
        v8: '10.7.193.13-node.16',
        uv: '1.43.0',
        zlib: '1.2.11',
        brotli: '1.0.9',
        ares: '1.18.1',
        modules: '111',
        nghttp2: '1.47.0',
        napi: '8',
        llhttp: '8.1.0',
        openssl: '3.0.7+quic',
        cldr: '41.0',
        icu: '71.1',
        tz: '2022b',
```

```
        unicode: '14.0',
        ngtcp2: '0.8.1',
        nghttp3: '0.7.0'
}
- process.arch: x64
- process.platform: win32
- process.connected: undefined
- process.execArgv: []
- process.exitCode: undefined
- process.mainModule: Module {
        id: '.',
        path: 'K:\\nodepro',
        exports: {},
        filename: 'K:\\nodepro\\index.js',
        loaded: false,
        children: [],
        paths: [ 'K:\\nodepro\\node_modules', 'K:\\node_modules' ]
}
- process.release: {
        name: 'node',
        sourceUrl: 'https://nodejs.org/download/release/v19.0.1/node-v19.0.1.tar.gz',
        headersUrl: 'https://nodejs.org/download/release/v19.0.1/node-v19.0.1-headers.tar.gz',
        libUrl: 'https://nodejs.org/download/release/v19.0.1/win-x64/node.lib'
}
- process.memoryUsage(): {
        rss: 28078080,
        heapTotal: 6639616,
        heapUsed: 6031272,
        external: 455122,
        arrayBuffers: 17378
}
- process.uptime(): 0.095098
```

4.1.3　全局函数

全局函数，即可以在程序的任何地方调用的函数，Node.js 主要提供了 6 个全局函数，其说明如表 4.6 所示。

表 4.6　Node.js 中的全局函数

函　　数	说　　明
setTimeout(cb,ms)	添加一个定时器，在指定的毫秒（ms）数后执行指定函数（cb）
clearTimeout(t)	取消定时器，停止一个之前调用 setTimeout()创建的定时器
setInterval(cb,ms)	添加一个定时器，每隔一定的时间（ms）就执行一次函数（cb）
clearInterval(t)	取消定时器，停止之前调用 setInterval()创建的定时器
setImmediate(callback[,...args])	安排在 I/O 事件的回调之后立即执行的 callback
clearImmediate(immediate)	取消由 setImmediate()创建的 Immediate 对象

下面对 Node.js 中全局函数的使用进行讲解。

1．setTimeout(cb,ms)和 clearTimeout(t)

这两个全局函数分别用来设置和取消一个定时器，此处需要说明的是，setTimeout(cb,ms)设置的定时器仅调用一次指定的方法。示例代码如下：

```
var timer=setTimeout(function(){
    console.log("您将在 2 秒后看到这句话")
},2000)
//clearTimeout(timer)
```

运行上面代码，2 秒后将会显示如下内容：

```
您将在 2 秒后看到这句话
```

如果将上面代码中的最后一行取消注释，则运行程序时不会输出任何内容，因为虽然前 3 行代码添加了一个定时器，但是第 4 行又取消了该定时器，所以在控制台不会输出内容。

2．setInterval(cb,ms)和 clearInterval(t)

这两个全局函数分别用来添加和取消一个定时器。其中参数 cb 为要执行的函数；ms 为调用 cb 函数前等待的时间；t 表示要取消的 setInterval()方法设置的定时器。使用 setInterval()方法设置定时器与使用 setTimeout()方法设置定时器的区别是，使用 setInterval()方法设置的定时器可以多次调用指定的方法，而使用 setTimeout()方法设置的定时器只能调用一次指定的方法。示例代码如下：

```
var i = 0                                          //记录执行程序的次数
var timer
timer = setInterval(function () {
    i += 1
    console.log("已执行" + i + "次")
    if (i >= 5) {
        clearInterval(timer)                       //执行 5 次后，取消定时器
        console.log("执行完毕")
    }
}, 2000)
```

运行上面的代码，每隔 2 秒会显示一次执行函数的次数，直到执行 5 次以后，取消定时器，最终输出结果如下：

```
已执行 1 次
已执行 2 次
已执行 3 次
已执行 4 次
已执行 5 次
执行完毕
```

3．setImmediate(callback[,...args])和 clearImmediate(immediate)

这两个全局函数用来安排在 I/O 事件的回调之后立即执行的函数，以及取消 setImmediate()创建的

Immediate 对象。其中，callback 参数指的是要执行的函数，immediate 参数表示使用 setImmediate()创建的 Immediate 对象。

说明

I/O（input/output）即输入/输出，通常指数据在内部存储器与外部存储器或其他周边设备之间的输入和输出。

示例代码如下：

```
console.log("正常执行 1");
var a = setImmediate(function () {
        console.log("我被延迟执行了");
});
console.log("正常执行 2")
//clearImmediate(a)
```

上面代码的运行结果如下：

```
正常执行 1
正常执行 2
我被延迟执行了
```

如果将最后一行代码取消注释，则运行结果如下：

```
正常执行 1
正常执行 2
```

4.2　模块化编程

Node.js 主要使用模块系统进行编程，所谓模块，是指为了方便调用功能，预先将相关方法和属性封装在一起的集合体。模块和文件是一一对应的，即一个 Node.js 文件就是一个模块，这个文件可以是 JavaScript 代码、JSON 或者编译过的 C/C++扩展等。下面对 Node.js 模板化编程中用到的两个对象 exports 和 module 进行讲解。

4.2.1　exports 对象

在 Node.js 中创建模块需要使用 exports 对象，该对象可以共享方法、变量、构造和类等，下面通过一个实例讲解如何使用 exports 创建一个模块。

【例 4.2】使用 exports 对象实现模块化编程。**（实例位置：资源包\源码\04\02）**

步骤如下。

（1）在 WebStorm 中创建一个 module.js 文件，其中通过 exports 对象共享求绝对值和计算圆面积的方法，代码如下：

```
//求绝对值的方法 abs
exports.abs = function (number) {
        if (0 < number) {
                return number;
        } else {
                return -number;
        }
};
//求圆面积的方法 circleArea
exports.circleArea = function (radius) {
        return radius * radius * Math.PI;
};
```

（2）在 WebStorm 中创建一个 main.js 文件，用来调用前面创建的模块来计算指定值的绝对值及指定半径的圆面积，代码如下：

```
//加载 module.js 模块文件
var module = require('./module.js');
//使用模块方法
console.log('abs(-273) = %d', module.abs(-273));
console.log('circleArea(3) = %d', module.circleArea(3));
```

说明

上面代码中，通过使用 require()导入了创建的 module.js 模块文件。

运行 main.js 文件，结果如下：

```
abs(-273) = 273
circleArea(3) = 28.274333882308138
```

4.2.2 module 对象

在 Node.js 中，除了使用 exports 对象进行模块化编程，还可以使用 module 对象进行模块化编程。module 对象的常用属性如表 4.7 所示。

表 4.7 module 对象的常用属性

属　性	说　明
id	模块的标识符，通常是完全解析后的文件名，默认输出
path	Node.js 运行 js 模块所在的文件路径
exports	公开的内容，也就是导出的对象，引入该模块会得到这个对象
filename	当前模块文件名，包含路径
loaded	模块是否加载完毕
parent	当前模块的父模块对象
children	当前模块的所有子模块对象

【例 4.3】使用 module 对象实现模块化编程。（**实例位置：资源包\源码\04\03**）

步骤如下。

（1）在 WebStorm 中创建一个 module.js 文件，其中定义一个输出方法，然后通过 module 对象的 exports 属性指定对外接口，代码如下：

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

（2）在 WebStorm 中创建一个 main.js 文件，用来调用创建的模块以输出内容，代码如下：

```
var Hello = require('./module.js');
hello = new Hello();
hello.setName('2023');
hello.sayHello();
```

运行 main.js 文件，结果如下：

```
2023，你好
```

注意

与使用 exports 对象相比，唯一的变化是使用 module.exports = Hello 代替了 exports。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。

4.3　要 点 回 顾

本章主要对 Node.js 编程的通用基础知识进行了介绍，以便为开发 Node.js 应用储备必要的基础。其中，分别讲解了 Node.js 中内置的全局变量、全局对象和全局函数以及实现模块化编程的两个对象 exports 和 module。学习本章时，重点掌握 console 对象和 module 对象的使用。