20180423

❁ Node.js 简介

① 浏览器大战

众所周知，在 Netscape 设计出 JavaScript 后短短几个月，JavaScript 事实上已经是前端开发的唯一标准。后来，微软通过 IE 击败了 Netscape 后一统桌面；结果几年时间，浏览器毫无进步(2001 年推出的古老 IE 6 至今仍然有人在使用!)。微软认为 IE6 已经非常完善而解散了 IE6 开发团队！而 Google 却认为支持现代 Web 应用的新一代浏览器才刚刚起步，尤其是浏览器负责运行 JavaScript 的引擎性能还可提升 10 倍。

先是 Mozilla 借助已壮烈牺牲的 Netscape 遗产在 2002 年推出了 Firefox 浏览器；接着 Apple 于 2003 年在开源的 KHTML 浏览器基础上推出了 WebKit 内核的 Safari 浏览器，不过仅限于 Mac 平台。随后 Google 也看中了 WebKit 内核，基于 WebKit 内核推出了 Chrome 浏览器。Chrome 浏览器是跨 Windows 和 Mac 平台的，并且 Google 认为要运行现代 Web 应用，浏览器必须有一个性能非常强劲的 JavaScript 引擎，于是 Google 开发了一个名叫 V8 的高性能 JavaScript 引擎，以 BSD 许可证开源。

现代浏览器大战让微软的 IE 浏览器远远落后了，因为解散了最有经验、战斗力最强的浏览器团队，回过头再追赶却发现支持 HTML5 的 WebKit 已经成为手机端的标准了，IE 浏览器从此与主流移动端设备绝缘。

② Node.js

有个叫 Ryan Dahl 的歪果仁，工作是用 C/C++写高性能 Web 服务。对于高性能，异步 IO、事件驱动是基本原则，但是用 C/C++写太痛苦了。于是这位仁兄开始设想用高级语言开发 Web 服务。他发现很多语言虽然同时提供了同步 IO 和异步 IO，但是开发人员一旦用了同步 IO，就再也懒得写异步 IO 了。最终，Ryan 选择了 JavaScript。因为 JavaScript 是单线程执行，根本不能进行同步 IO 操作。所以 JavaScript 的这一缺陷导致了只能使用异步 IO。
这位仁兄曾考虑自己写一个引擎，不过明智地放弃了，因为 V8 就是开源的 JavaScript 引擎。让 Google 去优化 V8，自己改造一下拿来用，还不用付钱，这买卖很划算。
2009 年 Ryan 正式推出了基于 JavaScript 语言和 V8 引擎的开源 Web 服务器项目，命名为 Node.js。Node 第一次把 JavaScript 带入到后端服务器开发，加上世界上已经有无数的 JavaScript 开发人员，所以 Node 一下子就火了。

在 Node 上运行的 JavaScript 相比其他后端开发语言，最大的优势是借助 JavaScript 天生的事件驱动机制和 V8 高性能引擎，容易编写高性能 Web 服务。其次，在 Node 环境下，通过模块化的 JavaScript 代码，加上函数式编程，无需考虑浏览器兼容性问题，直接使用最新的 ECMAScript 标准，可以完全满足工程上的需求。

③ io.js

Node.js 是开源项目，虽然由社区推动，但幕后一直由 Joyent 公司资助。由于一些开发者对 Joyent 公司策略不满，于 2014 年从 Node.js 项目 fork 出了 io.js 项目，决定单独发展，但两者实际上是兼容的。然而分家后没多久，Joyent 公司表示要和解，io.js 项目又回归 Node.js。结果是要添加新特性先加到 io.js，如果大家都很满意，就把新特性加入 Node.js；于是 io.js 变成尝鲜版，而 Node.js 相当于线上稳定版。

❁ 安装 Node.js

Node.js 是在后端运行 JavaScript 代码，必须先安装 Node 环境。
目前 Node.js 的最新版本是 9.11.1，官网推荐安装 8.11.1 LTS。
在 Windows 安装必须选择全部组件，包括勾选 Add to Path。

安装完成后，命令行输入 node -v，如果安装正常，可以看到版本号。
命令行输入 node，将进入 Node.js 交互环境。在交互环境可以输入任意 JavaScript 语句，回车后将得到输出结果。
连按两次 Ctrl+C 退出 Node.js 环境。

❁ npm

npm 是 Node.js 的包管理工具(package manager)。
// 相当于 Python 的 pip? 直接下载其他人开源的模块。
npm 还可以根据依赖关系，把所有依赖的包都下载并管理。
npm 在 Node.js 安装时顺便安装了；命令行输入 npm -v，输出 npm 的版本号，说明 npm 已经正确安装了。

❁ 第一个 Node 程序

hello.js：

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

第一行总是写'use strict';严格模式运行 JavaScript 代码，避免各种潜在陷阱。
命令行切到 hello.js 目录：

```
PS C:\Users\hikari\Desktop> node hello.js
hello world !
```

❁ Node 交互模式和命令行模式

Node 交互式环境会把每一行 JavaScript 代码的结果自动打印；但是命令行运行 JavaScript 文件却不会。
例如：在 Node 交互式环境下：

```
> 1+2+3;
6
```

但是写一个 calc.js 的文件，命令行模式下执行：

```
PS C:\Users\hikari\Desktop> node calc.js
```

什么也没有输出，必须用 console.log()才能输出结果。

输入 node 进入交互模式，相当于启动了 Node 解释器，每输入一行就执行一行；
运行 node calc.js 相当于启动了 Node 解释器，一次性把 calc.js 的代码执行了。
js 文件就是用于写大段代码；交互模式一般用于验证部分代码。

❁ 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。可以给 Nodejs 传递--use_strict 参数开启严格模式：

```
node --use_strict calc.js
```

❁ Node 集成开发环境

使用文本编辑器开发 Node 程序，最大的缺点是效率太低，运行 Node 程序还需在命令行单独敲命令。如果还需要调试程序，就更加麻烦了。

所以需要一个 IDE 集成开发环境，能在一个环境里编码、运行、调试，这样就可以大大提升开发效率。

Java 集成开发环境有 Eclipse，Intellij idea 等；C#集成开发环境有 Visual Studio，那么问题来了：Node.js 的集成开发环境到底哪家强？

此处推荐 Visual Studio Code，微软出品，是 Visual Studio 发精简版，并且 Visual Studio Code 可以跨平台。

安装时推荐勾上：桌面快捷方式、资源管理器目录上下文菜单、添加到 PATH

为 VS Code 创建工作目录，新建文件 hello.js

```
let name = 'world';
let s = `hello, ${name} !`;
console.log(s);
```

点击调试，发现没有配置，点击齿轮自动生成配置文件 launch.json：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "启动程序",
            "program": "${workspaceFolder}/hello.js" // 运行的 js 文件
        }
    ]
}
```

点击开始调试。调试控制台输出结果：

```
问题    输出    调试控制台    终端
D:\software\nodejs\node.exe --inspect-brk=18841 hello.js
Debugger listening on ws://127.0.0.1:18841/d42d423e-ea75-483b-a0c4-e4d97d079956
hello, world !
```

20180424

❁ 模块

随着程序代码越写越多，在一个文件里代码越来越长，越来越不容易维护。

为了编写可维护的代码，把函数分组放到不同的 js 文件，在 Node 环境一个 js 文件就是一个模块(module)，很多语言都采用这种代码组织方式。

除了提高代码可维护性，模块还可以避免重复造轮子，直接引用 Node 内置模块或第三方模块就可以使用；可以避免函数名和变量名冲突，相同名字的函数和变量可以存在不同模块中，因此在编写模块时，不必考虑名字与其他模块冲突。

示例：
hello.js：

```
function greet(name) {
    console.log(`hello, ${name} !`);
}
// 把 greet()函数作为模块输出暴露出去，其他模块就可以使用 greet()
module.exports = greet;
```

main.js：

```
let greet = require('./hello'); // 相对路径
let s = 'hikari';
greet(s); // hello, hikari !
```

引入的模块作为变量保存在 greet 变量中，就是在 hello.js 输出的 greet()函数。所以 main.js 成功地引用了 hello.js 模块的 greet()函数，然后可以直接使用了。
如果写成 require('hello')，Node 会依次在内置模块、全局模块和当前模块下查找 hello.js，很可能会得到一个错误：`Error: Cannot find module 'hello'`

VSCode 推荐使用 ES6 标准：

```
export default greet;
import greet from "./hello";
```

然后报错了…也就是不支持 export 和 import?

CommonJS 规范

这种模块加载机制称为 CommonJS 规范。此规范下，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。

要在模块中对外输出变量用：`module.exports = variable;`
输出的变量可以是任意对象、函数、数组等等。
要引入其他模块输出的对象用：`let foo = require('other_module');`
引入的对象具体是什么，取决于引入模块输出的对象。

深入了解模块原理

在浏览器中大量使用全局变量不好。如果在 a.js 中使用了全局变量 s，那么在 b.js 中也使用全局变量 s，将造成冲突，b.js 中对 s 赋值会改变 a.js 的运行逻辑。
JavaScript 本身并没有一种模块机制来保证不同模块可以使用相同的变量名。

但由于 JavaScript 支持闭包，如果把一段 JavaScript 代码用函数包装起来，其所有全局变量就变成了函数内部的局部变量，从而支持不同模块同名变量。

Node.js 加载了 js 后，把代码外面套一层匿名函数，包装一下变成：

```
(function () {
    // js 代码
})();
```

原来的全局变量变成了匿名函数内部的局部变量。如果 Node.js 继续加载其他模块，这些模块中定义的同名变量也互不干扰。

模块的输出 module.exports 实现：

```
// node 在加载 js 文件前准备的 module 对象
let module = {
    id: 'hello',
    exports: {}
};
let load = function (module) {
    // 读取的 hello.js 代码
    function greet(name) {
        console.log(`hello, ${name} !`);
    }
    module.exports = greet;
    return module.exports;
};
let exported = load(module);
// 保存 module
save(module, exported);
```

变量 module 是 Node 在加载 js 文件前准备的一个变量，并将其传入加载函数，在 hello.js 中可以直接使用变量 module 原因在于它实际上是函数的一个参数。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。

练习：编写 hello.js，输出多个函数；编写 main.js，引入 hello 模块，调用其函数。

hello.js：

```
let s = 'hello';
function greet(name) {
    console.log(`${s}, ${name} !`);
}
function hi(name) {
    console.log(`hi, ${name} !`);
}
function goodbye(name) {
    console.log(`goodbye, ${name} !`);
}
module.exports = {
    greet: greet,
    hi: hi,
    goodbye: goodbye
};
```

main.js：

```
const hello = require('./hello');
var s = 'hikari';
hello.greet(s); // hello, hikari !
hello.hi(s); // hi, hikari !
hello.goodbye(s); // goodbye, hikari !
```

20180425

❁ 基本模块

Node.js 是运行在服务区端的 JavaScript 环境。服务器程序和浏览器程序相比最大的特点是没有浏览器的安全限制。服务器程序必须能接收网络请求、读写文件、处理二进制内容，所以 Node.js 内置的常用模块就是为了实现基本的服务器功能。这些模块在浏览器环境中无法执行，因为它们的底层代码是用 C/C++在 Node.js 运行环境中实现的。

❁ Node.js 常用对象

① global

JavaScript 有且仅有一个全局对象，浏览器中是 window 对象。而在 Node.js 环境中也有唯一的全局对象 global，其属性和方法也和浏览器的 window 不同。

② process

process 也是 Node.js 提供的一个对象，代表当前 Node.js 进程。

```
// 获取 js 文件名和目录
console.log('current js file: ' + __filename); // d:\hello\01 global.js
console.log('current js dir: ' + __dirname); // d:\hello
// ver:v8.11.1; platform: win32; arch: x64
console.log(`ver:${process.version}; platform: ${process.platform}; arch:
${process.arch}`)
// process.argv 存储了命令行参数
console.log('arguments: ' + JSON.stringify(process.argv)); // arguments:
["D:\\software\\nodejs\\node.exe","d:\\hello\\01 global.js"]

// process.cwd() 返回当前工作目录
console.log('cwd: ' + process.cwd()); // cwd: d:\hello

// 切换当前工作目录
let d = '/private/tmp';
if (process.platform === 'win32') {
    // 如果是 Windows，切换到 C:\Windows\System32
    d = 'C:\\Windows\\System32';
}
process.chdir(d);
console.log('cwd: ' + process.cwd()); // cwd: C:\Windows\System32
```

JavaScript 是由事件驱动执行的单线程模型，Node.js 也不例外。Node.js 不断执行响应事件函数，直到没有任何响应事件函数可以执行时，Node.js 退出。

如果想在下一次事件响应中执行代码，可以调用 process.nextTick()：

```
// process.nextTick()将在下一轮事件循环中调用
process.nextTick(function () {
    console.log('nextTick callback!');
});
console.log('nextTick was set!');
// nextTick was set! nextTick callback!
```

说明传入 process.nextTick()的函数不是立刻执行，而要等到下一次事件循环。

Node.js 进程本身的事件就由 process 对象来处理。如果响应 exit 事件，就可以在程序即将退出时执行某个回调函数：

```
// 程序即将退出时的回调函数
process.on('exit', function (code) {
    console.log('about to exit with code: ' + code);
}); // about to exit with code: 0
```

❁ 判断 JavaScript 执行环境

很多 JavaScript 代码既能在浏览器中执行，也能在 Node 环境执行。

有时程序本身需要判断自己到底在什么环境下执行，常用方式是根据浏览器和 Node 环境提供的全局变量名称来判断：

```
let env = typeof window === 'undefined' ? 'node.js' : 'browser';
console.log(`environment: ${env}`); // environment: node.js
```

20180427

❁ fs 模块

Node.js 内置的 fs 模块是文件系统模块，负责读写文件。

和所有其它 JavaScript 模块不同的是 fs 模块同时提供了异步和同步的方法。

① 异步读文件

```
const fs = require('fs');
console.log('>>> BEGIN >>>')
fs.readFile('sample.in', 'utf-8', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data);
    }
});
console.log('>>> END >>>');
```

```
>>> BEGIN >>>
>>> END >>>
高坂穂乃果，絢瀬絵里，南ことり，
園田海未，星空凛，西木野真姫，
東條希，小泉花陽，矢澤にこ
```

异步读取时，回调函数接收 err 和 data 两个参数。正常读取时，err 为 null，data 为读取到的字符串；读取发生错误时，err 代表一个错误对象，data 为 undefined。这也是 Node.js 标准的回调函数：第一个参数为错误信息，第二个参数为结果。

当读取二进制文件时，不传入文件编码：

```
fs.readFile('sample.png', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(Array.isArray(data)); // false
        console.log(`${data.length} bytes`); // 73770 bytes
    }
});
```

回调函数的 data 是返回一个 Buffer 对象。在 Node.js 中，Buffer 对象是一个包含任意个字节的数组(注意和 Array 不同)，length 是字节数。

Buffer 对象可以和 String 转换：

```
// Buffer --> String
let text = data.toString('utf-8');
console.log(text); // 乱码
// String --> Buffer
let buf = Buffer.from(text, 'utf-8');
console.log(`buf.length=${buf.length}; data.length=${data.length}`);
// buf.length=133830; data.length=73770, 两者大小都不一样?
```

② 同步读文件

readFileSync()函数为同步读取文件函数，不接收回调函数，函数直接返回结果。

```
const fs = require('fs');
console.log('>>> BEGIN >>>');
let data = fs.readFileSync('sample.in', 'utf-8');
console.log(data);
console.log('>>> END >>>');
```

```
>>> BEGIN >>>
高坂穂乃果，絢瀬絵里，南ことり，
園田海未，星空凛，西木野真姫，
東條希，小泉花陽，矢澤にこ
>>> END >>>
```

如果同步读取文件发生错误，可以使用 try...catch...捕获

③ 写入文件

写入同样有异步的 writeFile()和同步的 writeFileSync()

```
const fs = require('fs');
//异步写入文件
```

```
function write(f, data) {
    console.log('>>> write begin >>>');
    fs.writeFile(f, data, function (err) {
        if (err) {
            console.log(err);
        } else {
            console.log('ok.');
        }
    });
    console.log('>>> write end >>>');
}
// 异步读取文件，异步将 data 写入另一文件，相当于复制
function read_write(f) {
    console.log('>>> read begin <<<')
    fs.readFile(f, function (err, data) {
        if (err) {
            console.log(err);
        } else {
            console.log(`${data.length} bytes`); // 73770 bytes
            write('output.png', data);
        }
    });
    console.log('>>> read end <<<');
}
read_write('sample.png');
```

```
>>> read begin <<<
>>> read end <<<
73770 bytes
>>> write begin >>>
>>> write end >>>
ok.
```

writeFile()的参数依次为文件名、数据和回调函数。如果传入的 data 是 String，默认按 UTF-8 编码写入文本文件；如果 data 是 Buffer，写入的是二进制文件。回调函数只关心成功与否，只需要一个 err 参数。

同步写入，不需要回调函数，直接写入。

```
// 同步写入文件
let s = 'hello hikari\nhello world';
fs.writeFileSync('hello.out', s);
```

④ stat

fs.stat()返回一个 Stat 对象，能获取文件或目录的详细信息：

```
const fs = require('fs');
fs.stat('sample.in', function (err, stat) {
    if (err) {
        console.log(err);
```

```
    } else {
        // 是不是文件，是不是目录?
        console.log('isFile: ' + stat.isFile());
        console.log('isDirectory: ' + stat.isDirectory());
        if (stat.isFile()) { // 是文件，打印文件大小，创建日期，修改日期
            console.log('size: ' + stat.size);
            console.log('birth time: ' + stat.birthtime); // date 对象
            console.log('modified time: ' + stat.mtime); // date 对象
        }
    }
});
```

结果：

```
isFile: true
isDirectory: false
size: 128
birth time: Fri Apr 27 2018 11:11:19 GMT+0800 (中国标准时间)
modified time: Fri Apr 27 2018 11:20:55 GMT+0800 (中国标准时间)
```

stat()对应同步函数 statSync()：

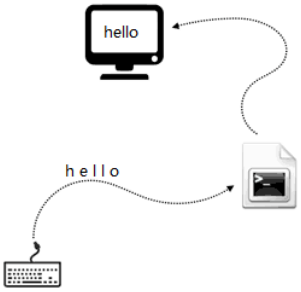
```
try {
    let info = fs.statSync('sample.in');
    console.log('birth time: ' + info.birthtime);
} catch (err) {
    console.log(err);
}
```

- 异步还是同步

在 fs 模块中，提供同步方法是为了方便使用。
绝大部分在服务器端反复执行业务逻辑的 JavaScript 代码，必须使用异步；否则同步代码在执行时，服务器将停止响应，因为 JavaScript 是单线程。
服务器启动时读取配置文件或结束时写入状态文件时，可以使用同步代码。因为这些代码只在启动和结束时执行一次，不影响服务器正常运行时的异步执行。

- stream 模块

stream 是仅在服务区端可用的模块，目的是支持流这种数据结构。



流是一种抽象的数据结构，可以把数据看成是数据流。比如敲键盘时每个字符连起来看成字符流，此流是从键盘输入到应用程序，称为标准输入流(stdin)。

反之应用程序把字符一个一个输出到显示器上，称为标准输出流(stdout)。
流的特点是数据有序，必须依次读取或依次写入。

Node.js 中流也是一个对象，只要响应流的事件就可以：data 事件表示流的数据可以读取；end 事件表示流结束，没有数据可读取了；error 事件表示出错。

① 文件流读取文本

```
const fs = require('fs');
// 打开一个输入流
let rs = fs.createReadStream('sample.in', 'utf-8');
// data 事件可能有多次，每次传递的 chunk 是流的一部分数据
rs.on('data', function (chunk) {
    console.log('data: ');
    console.log(chunk);
});
rs.on('end', function () {
    console.log('read stream end...');
});
rs.on('error', function (err) {
    console.log(err);
});
```

② 以流的形式写入文件，只要不断调用 write()，最后以 end()结束：

```
let ws1 = fs.createWriteStream('output1.txt', 'utf-8');
ws1.write('使用 Stream 写入文本数据...\n');
ws1.write('END.');
ws1.end();

let ws2 = fs.createWriteStream('output2.txt');
ws2.write(Buffer.from('使用 Stream 写入二进制数据...\r\n', 'utf-8'));
ws2.write(Buffer.from('END.', 'utf-8'));
ws2.end();
```

用 Windows 自带的记事本'\n'没有换行；而'\r\n'换行。然而 sublime、vscode 都是有换行的...怪不得记事本总是被黑...
stream.Readable 和 stream.Writable 是读取流和写入流的父类。

③ pipe
一个 Readable 流和一个 Writable 流串起来后，所有数据自动从 Readable 流进入 Writable 流，这种操作叫 pipe。
Node.js 中 Readable 流的 pipe()方法，可以将一个文件流和另一个文件流串起来，源文件的数据自动写入到目标文件中，所以实际上是一个复制文件的过程：

```
function copy_file(f1, f2) {
    let rs = fs.createReadStream(f1);
    let ws = fs.createWriteStream(f2);
```

```
    rs.pipe(ws);
}
copy_file('sample.png', 'copy.png');
```

默认当 Readable 流的数据读取完毕，end 事件触发将自动关闭 Writable 流。

如果不希望自动关闭 Writable 流，需要传入参数：

```
readable.pipe(writable, { end: false });
```

20180428