七天学会 NodeJS #3

让前端觉得如获神器的不是 NodeJS 能做网络编程，而是 NodeJS 能够操作文件。小至文件 查找，大至代码编译，几乎没有一个前端工具不操作文件。换个角度讲，几乎也只需要一些数据处理逻辑，再加上一些文件操作，就能够编写出大多数前端工具。本 章将介绍与之相关的 NodeJS 内置模块。

开门红

NodeJS 提供了基本的文件操作 API，但是像文件拷贝这种高级功能就没有提供，因此我们先拿文件拷贝程序练手。与 copy 命令类似，我们的程序需要能接受源文件路径与目标文件路径两个参数。

小文件拷贝

我们使用 NodeJS 内置的 fs 模块简单实现这个程序如下。

```
var fs = require('fs');



function copy(src, dst) {

    fs.writeFileSync(dst, fs.readFileSync(src));

}



function main(argv) {

    copy(argv[0], argv[1]);

}



main(process.argv.slice(2));
```

以上程序使用 fs.readFileSync 从源路径读取文件内容，并使用 fs.writeFileSync 将文件内容写入目标路径。

豆知识： process 是一个全局变量，可通过 process.argv 获得命令行参数。由于 argv[0]固定等于 NodeJS 执行程序的绝对路径，argv[1]固定等于主模块的绝对路径，因此第一个命令行参数从 argv[2]这个位置开始。

大文件拷贝

上边的程序拷贝一些小文件没啥问题，但这种一次性把所有文件内容都读取到内存中后再一次性写入磁盘的方式不适合拷贝大文件，内存会爆仓。对于大文件，我们只能读一点写一点，直到完成拷贝。因此上边的程序需要改造如下。

```
var fs = require('fs');



function copy(src, dst) {

    fs.createReadStream(src).pipe(fs.createWriteStream(dst));

}



function main(argv) {

    copy(argv[0], argv[1]);

}



main(process.argv.slice(2));
```

以上程序使用 `fs.createReadStream` 创建了一个源文件的只读数据流，并使用 `fs.createWriteStream` 创建了一个目标文件的只写数据流，并且用 `pipe` 方法把两个数据流连接了起来。连接起来后发生的事情，说得抽象点的话，水顺着水管从一个桶流到了另一个桶。

API 走马观花

我们先大致看看 **NodeJS** 提供了哪些和文件操作有关的 **API**。这里并不逐一介绍每个 **API** 的使用方法，官方文档已经做得很好了。

Buffer（数据块）

官方文档： http://nodejs.org/api/buffer.html

JS 语言自身只有字符串数据类型，没有二进制数据类型，因此 **NodeJS** 提供了一个与 `String` 对等的全局构造函数 `Buffer` 来提供对二进制数据的操作。除了可以读取文件得到 `Buffer` 的实例外，还能够直接构造，例如：

```
var bin = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6c ]);
```

Buffer 与字符串类似，除了可以用.length 属性得到字节长度外，还可以用[index]方式读取指定位置的字节，例如：

```
bin[0]; // => 0x48;
```

Buffer 与字符串能够互相转化，例如可以使用指定编码将二进制数据转化为字符串：

```
var str = bin.toString('utf-8'); // => "hello"
```

或者反过来，将字符串转换为指定编码下的二进制数据：

```
var bin = new Buffer('hello', 'utf-8'); // => <Buffer 68 65 6c 6c 6f>
```

Buffer 与字符串有一个重要区别。字符串是只读的，并且对字符串的任何修改得到的都是一个新字符串，原字符串保持不变。至于 Buffer，更像是可以做指针操作的 C 语言数组。例如，可以用[index]方式直接修改某个位置的字节。

```
bin[0] = 0x48;
```

而.slice 方法也不是返回一个新的 Buffer，而更像是返回了指向原 Buffer 中间的某个位置的指针，如下所示。

```
[ 0x48, 0x65, 0x6c, 0x6c, 0x6c ]

  ^           ^

  |           |

 bin      bin.slice(2)
```

因此对.slice 方法返回的 Buffer 的修改会作用于原 Buffer，例如：

```
var bin = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6c ]);

var sub = bin.slice(2);


sub[0] = 0x65;

console.log(bin); // => <Buffer 48 65 65 6c 6f>
```

也因此，如果想要拷贝一份 Buffer，得首先创建一个新的 Buffer，并通过.copy 方法把原 Buffer 中的数据复制过去。这个类似于申请一块新的内存，并把已有内存中的数据复制过去。以下是一个例子。

```
var bin = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6c ]);

var dup = new Buffer(bin.length);
```

```
bin.copy(dup);

dup[0] = 0x68;

console.log(bin); // => <Buffer 48 65 6c 6c 6f>

console.log(dup); // => <Buffer 68 65 65 6c 6f>
```

总之，Buffer 将 JS 的数据处理能力从字符串扩展到了任意二进制数据。

Stream（数据流）

官方文档： http://nodejs.org/api/buffer.html

当内存中无法一次装下需要处理的数据时，或者一边读取一边处理更加高效时，我们就需要用到数据流。NodeJS 中通过各种 Stream 来提供对数据流的操作。

以上边的大文件拷贝程序为例，我们可以为数据来源创建一个只读数据流，示例如下：

```
var rs = fs.createReadStream(pathname);


rs.on('data', function (chunk) {

    doSomething(chunk);

});


rs.on('end', function () {

    cleanUp();

});
```

豆知识： Stream 基于事件机制工作，所有 Stream 的实例都继承于 NodeJS 提供的 EventEmitter。

上边的代码中 data 事件会源源不断地被触发，不管 doSomething 函数是否处理得过来。代码可以继续做如下改造，以解决这个问题。

```
var rs = fs.createReadStream(src);
```

```
rs.on('data', function (chunk) {
    rs.pause();
    doSomething(chunk, function () {
        rs.resume();
    });
});

rs.on('end', function () {
    cleanUp();
});
```

以上代码给 `doSomething` 函数加上了回调，因此我们可以在处理数据前暂停数据读取，并在处理数据后继续读取数据。

此外，我们也可以为数据目标创建一个只写数据流，示例如下：

```
var rs = fs.createReadStream(src);
var ws = fs.createWriteStream(dst);

rs.on('data', function (chunk) {
    ws.write(chunk);
});

rs.on('end', function () {
    ws.end();
});
```

我们把 `doSomething` 换成了望只写数据流里写入数据后，以上代码看起来就像是一个文件拷贝程序了。但是以上代码存在上边提到的问题，如果写入速度跟不上读取速度的话，只写数

据流内部的缓存会爆仓。我们可以根据.write方法的返回值来判断传入的数据是写入目标了，还是临时放在了缓存了，并根据drain事件来判断什么时候只写数据流已经将缓存中的数据写入目标，可以传入下一个待写数据了。因此代码可以改造如下：

```
var rs = fs.createReadStream(src);

var ws = fs.createWriteStream(dst);



rs.on('data', function (chunk) {

    if (ws.write(chunk) === false) {

        rs.pause();

    }

});



rs.on('end', function () {

    ws.end();

});



ws.on('drain', function () {

    rs.resume();

});
```

以上代码实现了数据从只读数据流到只写数据流的搬运，并包括了防爆仓控制。因为这种使用场景很多，例如上边的大文件拷贝程序，NodeJS直接提供了.pipe方法来做这件事情，其内部实现方式与上边的代码类似。

File System（文件系统）

官方文档： http://nodejs.org/api/buffer.html

NodeJS通过fs内置模块提供对文件的操作。fs模块提供的API基本上可以分为以下三类：

文件属性读写。

其中常用的有 fs.stat、fs.chmod、fs.chown 等等。

文件内容读写。

其中常用的有 fs.readFile、fs.readdir、fs.writeFile、fs.mkdir 等等。

底层文件操作。

其中常用的有 fs.open、fs.read、fs.write、fs.close 等等。

NodeJS 最精华的异步 IO 模型在 fs 模块里有着充分的体现，例如上边提到的这些 API 都通过回调函数传递结果。以 fs.readFile 为例：

```
fs.readFile(pathname, function (err, data) {
    if (err) {
        // Deal with error.
    } else {
        // Deal with data.
    }
});
```

如上边代码所示，基本上所有 fs 模块 API 的回调参数都有两个。第一个参数在有错误发生时等于异常对象，第二个参数始终用于返回 API 方法执行结果。

此外，fs 模块的所有异步 API 都有对应的同步版本，用于无法使用异步操作时，或者同步操作更方便时的情况。同步 API 除了方法名的末尾多了一个 Sync 之外，异常对象与执行结果的传递方式也有相应变化。同样以 fs.readFileSync 为例：

```
try {
    var data = fs.readFileSync(pathname);
    // Deal with data.
} catch (err) {
    // Deal with error.
}
```

fs 模块提供的 API 很多，这里不一一介绍，需要时请自行查阅官方文档。

Path（路径）

官方文档： http://nodejs.org/api/buffer.html

操作文件时难免不与文件路径打交道。NodeJS 提供了 path 内置模块来简化路径相关操作，并提升代码可读性。以下分别介绍几个常用的 API。

path.normalize

将传入的路径转换为标准路径，具体讲的话，除了解析路径中的.与..外，还能去掉多余的斜杠。如果有程序需要使用路径作为某些数据的索引，但又允许用户随意输入路径时，就需要使用该方法保证路径的唯一性。以下是一个例子：

```
var cache = {};


function store(key, value) {
    cache[path.normalize(key)] = value;
}


store('foo/bar', 1);
store('foo//baz//../bar', 2);
console.log(cache);  // => { "foo/bar": 2 }
```

坑出没注意： 标准化之后的路径里的斜杠在 Windows 系统下是\，而在*nix 系统下是/。如果想保证任何系统下都使用/作为路径分隔符的话，需要用.replace(/\\/g, '/')再替换一下标准路径。

path.join

将传入的多个路径拼接为标准路径。该方法可避免手工拼接路径字符串的繁琐，并且能在不同系统下正确使用相应的路径分隔符。以下是一个例子：

```
path.join('foo/', 'baz/', '../bar'); // => "foo/bar"
```

path.extname

当我们需要根据不同文件扩展名做不同操作时，该方法就显得很好用。以下是一个例子：

```
path.extname('foo/bar.js'); // => ".js"
```

path 模块提供的其余方法也不多，稍微看一下官方文档就能全部掌握。

遍历目录

遍历目录是操作文件时的一个常见需求。比如写一个程序，需要找到并处理指定目录下的所有 **JS** 文件时，就需要遍历整个目录。

递归算法

遍历目录时一般使用递归算法，否则就难以编写出简洁的代码。递归算法与数学归纳法类似，通过不断缩小问题的规模来解决问题。以下示例说明了这种方法。

```
function factorial(n) {
    if (n === 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

上边的函数用于计算 **N** 的阶乘（**N!**）。可以看到，当 **N** 大于**1**时，问题简化为计算 **N** 乘以 **N-1** 的阶乘。当 **N** 等于**1**时，问题达到最小规模，不需要再简化，因此直接返回**1**。

陷阱： 使用递归算法编写的代码虽然简洁，但由于每递归一次就产生一次函数调用，在需要优先考虑性能时，需要把递归算法转换为循环算法，以减少函数调用次数。

遍历算法

目录是一个树状结构，在遍历时一般使用深度优先+先序遍历算法。深度优先，意味着到达一个节点后，首先接着遍历子节点而不是邻居节点。先序遍历，意味着首次到达了某节点就算遍历完成，而不是最后一次返回某节点才算数。因此使用这种遍历方式时，下边这棵树的遍历顺序是 `A > B > D > E > C > F`。

同步遍历

了解了必要的算法后，我们可以简单地实现以下目录遍历函数。

```
function travel(dir, callback) {
    fs.readdirSync(dir).forEach(function (file) {
        var pathname = path.join(dir, file);

        if (fs.statSync(pathname).isDirectory()) {
            travel(pathname, callback);
        } else {
            callback(pathname);
        }
    });
}
```

可以看到，该函数以某个目录作为遍历的起点。遇到一个子目录时，就先接着遍历子目录。遇到一个文件时，就把文件的绝对路径传给回调函数。回调函数拿到文件路径后，就可以做各种判断和处理。因此假设有以下目录：

```
- /home/user/
    - foo/
        x.js
    - bar/
        y.js
    z.css
```

使用以下代码遍历该目录时，得到的输入如下。

```
travel('/home/user', function (pathname) {
    console.log(pathname);
```

```
});
```

/home/user/foo/x.js

/home/user/bar/y.js

/home/user/z.css

异步遍历

如果读取目录或读取文件状态时使用的是异步 **API**，目录遍历函数实现起来会有些复杂，但原理完全相同。**travel** 函数的异步版本如下。

```
function travel(dir, callback, finish) {
    fs.readdir(dir, function (err, files) {
        (function next(i) {
            if (i < files.length) {
                var pathname = path.join(dir, files[i]);

                fs.stat(pathname, function (err, stats) {
                    if (stats.isDirectory()) {
                        travel(pathname, callback, function () {
                            next(i + 1);
                        });
                    } else {
                        callback(pathname, function () {
                            next(i + 1);
                        });
```

```
                }

            });

        } else {

            finish && finish();

        }

    }(0));

});

}
```

这里不详细介绍异步遍历函数的编写技巧，在后续章节中会详细介绍这个。总之我们可以看到异步编程还是蛮复杂的。

文本编码

使用 **NodeJS** 编写前端工具时，操作得最多的是文本文件，因此也就涉及到了文件编码的处理问题。我们常用的文本编码有 **UTF8**和 **GBK** 两种，并且 **UTF8**文件还可能带有 **BOM**。在读取不同编码的文本文件时，需要将文件内容转换为 **JS** 使用的 **UTF8**编码字符串后才能正常处理。

BOM 的移除

BOM 用于标记一个文本文件使用 **Unicode** 编码，其本身是一个 **Unicode** 字符（**"\uFEFF"**），位于文本文件头部。在不同的 **Unicode** 编码下，**BOM** 字符对应的二进制字节如下：

```
Bytes      Encoding
----------------------------
FE FF      UTF16BE
FF FE      UTF16LE
EF BB BF   UTF8
```

因此，我们可以根据文本文件头几个字节等于啥来判断文件是否包含 **BOM**，以及使用哪种 **Unicode** 编码。但是，**BOM** 字符虽然起到了标记文件编码的作用，其本身却不属于文件内容的一部分，如果读取文本文件时不去掉 **BOM**，在某些使用场景 下就会有问题。例如我们把几个 **JS** 文件合并成一个文件后，如果文件中间含有 **BOM** 字符，就会导致浏览器 **JS** 语法错误。因此，使用 **NodeJS** 读取文本文件 时，一般需要去掉 **BOM**。例如，以下代码实现了识别和去除 **UTF8 BOM** 的功能。

```
function readText(pathname) {

    var bin = fs.readFileSync(pathname);



    if (bin[0] === 0xEF && bin[1] === 0xBB && bin[2] === 0xBF) {

        bin = bin.slice(3);

    }



    return bin.toString('utf-8');

}
```

GBK 转 UTF8

NodeJS 支持在读取文本文件时，或者在 Buffer 转换为字符串时指定文本编码，但遗憾的是，GBK 编码不在 NodeJS 自身支持范围内。因此，一般我们借助 iconv-lite 这个三方包来转换编码。使用 NPM 下载该包后，我们可以按下边方式编写一个读取 GBK 文本文件的函数。

```
var iconv = require('iconv-lite');



function readGBKText(pathname) {

    var bin = fs.readFileSync(pathname);



    return iconv.decode(bin, 'gbk');

}
```

单字节编码

有时候，我们无法预知需要读取的文件采用哪种编码，因此也就无法指定正确的编码。比如我们要处理的某些 CSS 文件中，有的用 GBK 编码，有的用 UTF8编码。虽然可以一定程度可以根据文件的字节内容猜测出文本编码，但这里要介绍的是有些局限，但是要简单得多的一种技术。

首先我们知道，如果一个文本文件只包含英文字符，比如 Hello World，那无论用 GBK 编码或是 UTF8编码读取这个文件都是没问题的。这是因为在这些编码下，ASCII0~128范围内字符

都使用相同的单字节编码。

反过来讲，即使一个文本文件中有中文等字符，如果我们需要处理的字符仅在 ASCII0~128 范围内，比如除了注释和字符串以外的 JS 代码，我们就可以统一使用单字节编码来读取文件，不用关心文件的实际编码是 GBK 还是 UTF8。以下示例说明了这种方法。

1. GBK 编码源文件内容：

```
var foo = '中文';
```

2. 对应字节：

```
76 61 72 20 66 6F 6F 20 3D 20 27 D6 D0 CE C4 27 3B
```

3. 使用单字节编码读取后得到的内容：

```
var foo = '{乱码}{乱码}{乱码}{乱码}';
```

4. 替换内容：

```
var bar = '{乱码}{乱码}{乱码}{乱码}';
```

5. 使用单字节编码保存后对应字节：

```
76 61 72 20 62 61 72 20 3D 20 27 D6 D0 CE C4 27 3B
```

6. 使用 GBK 编码读取后得到内容：

```
var bar = '中文';
```

这里的诀窍在于，不管大于 0xEF 的单个字节在单字节编码下被解析成什么乱码字符，使用同样的单字节编码保存这些乱码字符时，背后对应的字节保持不变。

NodeJS 中自带了一种 binary 编码可以用来实现这个方法，因此在下例中，我们使用这种编码来演示上例对应的代码该怎么写。

```
function replace(pathname) {
    var str = fs.readFileSync(pathname, 'binary');
    str = str.replace('foo', 'bar');
    fs.writeFileSync(pathname, str, 'binary');
}
```

小结

本章介绍了使用 **NodeJS** 操作文件时需要的 **API** 以及一些技巧，总结起来有以下几点：

学好文件操作，编写各种程序都不怕。

如果不是很在意性能，`fs` 模块的同步 **API** 能让生活更加美好。

需要对文件读写做到字节级别的精细控制时，请使用 `fs` 模块的文件底层操作 **API**。

不要使用拼接字符串的方式来处理路径，使用 `path` 模块。

掌握好目录遍历和文件编码处理技巧，很实用。