[中英文 JSON 合并工具] Node 的文件操作能力 - fs

> 本节目标：【实现一个中英文JSON合并工具】- 一切皆文件，文件即数据，服务器作为数据的批发市场，文件读写能力成就了调度奇迹

使用 File System 的时候，就像其他的模块一样，我们把它导进来即可：

```
1 const fs = require('fs')
```

下文的示例代码均省略此句，以节省字数。

Node 的搬砖专业户

工程师经常自嘲是个搬砖的。在我的印象里面，Node 里面的搬砖伙计就是 fs ，也就是 File System 这货，一切跟文件相关的操作，都跟它脱不了关系。比如拷贝一个文件，在 Node 里面非常简单：

```
1 fs.copyFileSync('./1.txt', './2.txt')
```

无论是 Windows 还是 Linux 系统，它里面的视频、语音、图片、JSON 文件、二进制的程序压缩包等等，都是文件。文件散布在各个我们称之为文件夹的地方，而文件夹在本质上也是一种文件，所以一切皆文件。而文件本质上是带特殊属性的数据，所以我们往往提到跟操作系统交互，跟系统底层交互，这里面有相当一部分其实是在跟文件交互，跟数据交互。本质上是在操作系统里面，对于数据的阅读能力、输入输出能力，跟数据库的增删改查行为差不了太多。我们人作为一种物种，不也是如此么，被创建，被销毁，每天都在更新，还能被打上身份标签拎出来识别，只不过多了所谓灵魂、思想和情感这些很难量化的数据而已。

在 Node 里面有很多核心的能力，但最原始最基础最通用的能力，其实就是文件系统交互能力 - fs ，以及网络请求的处理能力 - http 。正是这两块能力彻底打破了前后端的边界，从前用 JS 是没办法操纵磁盘文件的，通过 JS 也没办法处理网络请求与返回，现在有了 Node 这个混搭各种底层的运行框架，搬砖就变得可能了。

那我们就进入这个能力里面一窥究竟吧。首先看看它的 File System API，扳扳指头算，密密麻麻百来个方法挂在 fs 上面，大部分都分为同步的和异步两个版本，比如创建文件夹有这样的写法：

```
fs.mkdir(path[,options], callback)
fs.mkdirSync(path[,options])
```

其实不光 mkdir，其他大部分的文件方法，都有 Sync 的版本。所以这百十个方法除去一半，其实也就剩下了几十个而已，无非一个是 callback 回调函数来异步获取结果，一个则是同步执行，阻滞它后面的代码执行，大家不用怕他们太多记不住，用的时候来查 API 就行。那到底同步和异步在底层执行上有什么区别呢，我们先挑选几个常用的 API，熟悉下它们的用法后，自然就有答案了。

文件能力

把数据从文件里面读出来，是一个刚需，比如一个项目在启动时候，需要把配置文件读进来，可以这样做：

```
fs.readFile('./config.json', function (err, data) {
  if (err) throw err
  loadServer(data)
}
// 或者使用同步的方式
const data = fs.readFileSync('./config.json')
loadServer(data)
```

既然活都一样干，那什么时候该用同步什么时候该用异步呢，我的建议是：

- 对于一次性加载进来，不会二次加载的，比如项目启动之初，或者过程中只读取一次配置文件的，可以用 Sync/同步的方法
- 对于体积较大的，可能需要多次读写的，有可能影响到业务流程的响应速度的，改用异步的方式做特殊情况就特殊处理，如果实在不知道选哪种，那就保险起见选异步的方式做。如果采用同步的方式做，需要注意捕获错误，避免程序崩溃，比如这样写：

```
let data
try{
  data = fs.readFileSync('./config.json')
} catch (err) {
  console.log('读取配置文件出错')
}
```

我们后面的示例代码都选用异步的方式来写，同步的不再赘述。

一旦我们用异步的方式来写代码，就会面临 Node 社区早些年一直头疼的问题，就是 callback hell - 回调地狱，表现就是这样的：

```
fs.readFile('./cfg1.json', function (err, data1) {
  fs.readFile(data1.cfgPath, function (err, data2) {
    fs.readFile(data2.cfgPath, function (err, data3) {
      fs.readFile(data3.cfgPath, function (err, data4) {
        fs.readFile(data3.cfgPath, function (err, data5) {
          //....
        })
      })
    })
  })
})
```

当然现实中不会这么变态（其实也不排除比这还要变态），这些异步操作有先后依赖关系，按照顺序执行，这种代码写起来维护起来都很恶心。除了用同步（Sync）的方式做，社区也衍生了很多解决方案，甚至直接推动了整个 Javascript 语言特性的发展，比如 GeneratorFunction 迭代执行的函数，比如 Async 异步函数，比如 Promise 规范以及实现了 Promise 规范的一系列库等等。而 Node 里面，新版里面 Promise 已经挂载到了 Global 下面可以直接使用。那我们用 Promise 改写一下：

```
// 先把 readFile 封装到一个 Prmoise 里面
const readFile = filePath => new Promise((resolve, reject) => {
  fs.readFile(filePath, (err, data) => resolve(data))
}
// 然后通过 Promise.then 链式调用
readFile('./cfg1.json')
  .then(data => readFile(data.cfgPath))
  .then(data => readFile(data.cfgPath))
  .then(data => readFile(data.cfgPath))
  .then(data => {
    console.log('data4.cfgPath:', data)
   })
```

改成了 Promise 的链式调用后，代码嵌套好多了，但是一个冗长的链条，依然看着不太舒服。想要获取链路头部的一些数据也需要额外的包装，不是很方便。干脆通过 Async function 再来重构下：

```
// 先把 readFile 封装到一个 Prmoise 里面
const readFile = filePath => new Promise((resolve, reject) => {
  fs.readFile(filePath, (err, data) => resolve(data))
}
async function readCfg() {
  const data1 = await readFile('./cfg1.json')
  const data2 = await readFile(data1.cfgPath)
  const data3 = await readFile(data2.cfgPath)
```

```
  const data4 = await readFile(data3.cfgPath)
  console.log(data4)
}
```

这样写，就看着自然多了，所有的异步 callback 都可以用同步的方式来写，不光 readFile，fs 里面任何的异步方法，都可以用这种方式包装。

fs.readFile(path[, options], callback)

这是 readFile 的 API，其实我最早接触 path[, options], callback 这种参数形式的时候，我是真心看不懂的，不明觉厉，直到后来才意识到，原来这里的 [, options] 意思是这个参数可以省略不写。所以 readFile 接收三个参数，文件路径、配置和回调，在 options 这里，可以直接用 ‘utf8’ 来替代，通过指定编码，就可以拿到的 指定编码解析后的字符串，也就是：

fs.readFile('./1.txt', 'utf8', function(err, data) {})

那如果是第二个参数留空，也不指定 encoding，那么返回的 data 就是一个 Buffer 形式表示的二进制数据了，如果指定的话，就是这样来写：

```
fs.readFile('./1.txt', {
  flag: 'r+',
  encoding: 'utf8'
}, function(err, data) {})
```

这个 options 可以是一个对象，除了 encoding，还可以配置 flag 的值，flag 按我的理解就是标志位，表示文件的打开模式，标识是否具有某个权限。

熟悉 UNIX 系统的童鞋就会知道，UNIX 的文件权限标志位用 9 个码位标识，3 个码位一组，共 3 组。3 个码位依次标识可读，可写，可执行，1 表示有权限，0 表示没权限。3 组依次表示登录用户，同组用户，其他用户的权限。

比如 r 是对打开的文件进行只读, 带上加号表示的是对打开的文件进行读写，如果该文件不存在，会抛出错误，如果是 w+ 的话，同样是表示读写，但是文件不存在的话，会创建一个，并不会抛出错误，这些标志位可以参考如下列表：

- 'a' : 打开文件用于追加。 如果文件不存在，则创建该文件。
- 'ax' : 类似于 'a' ，但如果路径存在，则失败。
- 'a+' : 打开文件用于读取和追加。 如果文件不存在，则创建该文件。
- 'ax+' : 类似于 'a+' ，但如果路径存在，则失败。
- 'as' : 打开文件用于追加（在同步模式中）。 如果文件不存在，则创建该文件。
- 'as+' : 打开文件用于读取和追加（在同步模式中）。 如果文件不存在，则创建该文件。
- 'r' : 打开文件用于读取。 如果文件不存在，则会发生异常。

- 'r+' : 打开文件用于读取和写入。 如果文件不存在，则会发生异常。
- 'rs+' : 打开文件用于读取和写入（在同步模式中）。 指示操作系统绕过本地的文件系统缓存。

这对于在 NFS 挂载上打开文件时非常有用，因为它可以跳过可能过时的本地缓存。 它对 I/O 性能有非常实际的影响，因此不建议使用此标志（除非真的需要）。

这不会把 fs.open() 或 fsPromises.open() 变成同步的阻塞调用。 如果需要同步的操 作， 则应使用 fs.openSync() 之类的。

- 'w' : 打开文件用于写入。 如果文件不存在则创建文件，如果文件存在则截断文件。
- 'wx' : 类似于 'w' ，但如果路径存在，则失败。
- 'w+' : 打开文件用于读取和写入。 如果文件不存在则创建文件，如果文件存在则截断文件。
- 'wx+' : 类似于 'w+' ，但如果路径存在，则失败。

这些在官方的文档里都有介绍，其实大家只要理解了 read write add 这些模式的意义，等到具体使用的时候，根据场景来设置这个 flag 就可以了。

打开文件

读文件的时候，文件并不一定存在，我们有时候会直接来打开这个文件，可以通过 fs 的 open 方法对文件进行打开操作：

```
fs.open('./a.txt', 'w', function(err, fd) {
  console.log(fd)
  // fs.open 打开了文件，使用后通过fs.close关闭文件
  fs.close(fd, callback)
})
```

打开文件后，就可以通过文件描述符对文件进行读写操作了。open 接收的参数，第一个是文件名，第二个是标志位，最后一个参数就是回调函数。回调函数中第二个参数 fd 表示打开文件的文件描述符，我们要的就是这货。

等到文件打开后，就可以使用 fs.read() 方法进行更加精细的读取。所谓精细的控制，就是说 read 可以传入一大堆参数：

```
fs.read(fd, buffer, offset, length, position, callback)
```

fd 参数，文件描述符，通过 fs.open 拿到，buffer 参数，是把读取的数据写入这个对象，是个 Buffer 对象，offset 参数，写入 buffer 的起始位置，length 参数，写入 buffer 的长度，position 参数，从文

件的什么位置开始读。然后 callback(err, bytesRead, buffer) 回调方法传入三个参数，第一个参数是出现异常抛出的 err，第二个参数是读取了多少 bytes，最后一个是 buffer 读取到的数据。

在读取前需要创建一个用于保存文件数据的缓冲区，缓冲区数据最终会被传递到回调函数中：

```
fs.open('./a.txt', 'w', function(err, fd) {
  var buf = Buffer.alloc(1024)
  var offset = 0
  var len = buf.length
  var pos = 101
  fs.read(fd, buf, offset, len, pos, function(err, bytes, buffer) {
    console.log('读取' + bytes + ' bytes')
    console.log(buf.slice(0, bytes).toString())
  }
})
```

写文件

对应读文件，就是写文件，第一个参数是文件名，第二个是要写入的 buffer 数据，第三个是可省略的参数，跟 readFile 方法一样，配置读写权限和编码格式，设置 flag 为 w, 如果文件不存在会创建一个文件，这种写入方式会全部删除旧有的数据，然后再写入数据，最后一个是回调方法。

```
const data = Buffer.from('Hello World')
fs.writeFile('./b.txt', data, {
  flag: 'w',
  encoding: 'utf8'
}, function(err){})
```

写文件还可以更加精细控制，通过 write 方法，类似 read 方法，write 方法也接受许多参数。

fs.write(fd, buffer, offset, length, position, callback)

首先第一个参数是 fd，通过 fs.open 可以拿到文件描述符。write 方法通过这个找到文件的所在位置。第二个参数是 buffer 缓冲区，也就是即将被写入到这个文件的二进制数据， 第三个参数是 offset，也就是 buffer 写入的偏移量，一般默认从 0 开始写，每写一次，就修改一下这个偏移量，从而保证数据的连续和完整性，第四个参数是 length，也就是要写入的 buffer 的字节数长度。通过 offset 和 length 的结合，来指定哪些数据应该被写入到文件中。第五个参数是 position，用来指定写入到文件的什么位置，如果是 null，将会从当前文件指针的位置开始写入，最后跟着的参数就是一个回调函数 callback，里面传递了三个参数，第一个毫无疑问是错误对象 err，第二个是写入了多少 bytes，第三个是缓冲区写入的数据。

```
fs.open('./c.txt', 'a', function(err, fd) {
  const buf = Buffer.from('Hello World')
  const offset = 0
  const len = buf.length
  const pos = 100

  fs.write(fd, buf, offset, len, pos, function(err, bytes, buffer) {
    console.log('写入了 ' + bytes + ' bytes')
    // 数据已被填充到 buf 中
    console.log(buf.slice(0, bytes).toString())
    fs.close(fd, function(err){})
  })
})
```

这样竟然可以把 buffer 数据给存储到了文件中，write 还有第二种用法，就是不直接写入 buffer 数据，而是写入字符串，怎么做呢?

```
fs.open('./c.txt', 'a', function(err, fd) {
  const data = 'Hello World'

  // 回调函数第一个参数是异常，第二个是指定多少字符数将被写入到文件，最后一个是返回的字符串
  fs.write(fd, data, 0, 'utf8', function(err, wtitten, string) {
    console.log(wtitten)
    console.log(string)
    fs.close(fd, function(err){})
  })
})
```

学会了读写的这些方法，我们就可以来实现一个小工具了，来检查一张图片是不是 PNG 格式，因为 PNG 头部 8 bytes是固定的，所以拿到文件前 8 bytes就可以作为判断的条件。

```
// png.js
fs.open('11.png', 'r', function(err, fd) {
  var header = new Buffer([137, 80, 78, 71, 13, 10, 26, 10])
  var buf = new Buffer(8)
  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes, buffer) {
    if (header.toString() === buffer.toString()){
       console.log('是 PNG 图片')
    } else {
      console.log('不是 PNG 图片')
    }

```

```
  })
})
```

先是用 fs.open 打开 png 文件，然后 header 数据是 PNG 图片标识数据，位于 PNG 图片前 8 个 bytes，只要读取文件前 8 bytes 数据，然后对比一下数据是否一致就可以了。

那么关于写，有时候我们就想在文件结尾追加一些内容，比如每次服务器访问，我们就对日志文件增加一行记录，这时候，fs 的 appendFile 方法就非常顺手：

```
fs.appendFile('./c.txt', 'Hello World', {
  encoding: 'utf8'
}, function(err) {
  console.log('done')
})
```

目录读写能力

除了文件，文件夹也可以读写，也即目录可以创建，删除，支持遍历。创建非常简单，命令行里面我们通过是 makeDir 来创建，fs 模块也有一个 makedir 方法，传入目录的完整路径和路径名就可以了，默认的权限是 0777。这个就不演示了，我们直接看如何读取目录，也是直接上一个代码，实现一个小功能，把某个目录下的 js 文件都给遍历出来，然后放到一个数组里。

```
// 使用 fs.readdir 读取目录，重点是其回调函数files对象
/**
  * path，要读取目录的完整路径及目录名
  * [callback(err, files)]，读取目录回调函数；err错误对象，files数组，存放读取的目录中的所有文件名
*/
const walk = function(path) {
  fs.readdirSync(path)
    .forEach(function(file) {
      const newPath = path + '/' + file
      const stat = fs.statSync(newPath)
      if (stat.isFile()) {
        if (/\.js/.test(file)){
          files.push(file)
        }
      } else if(stat.isDirectory()) {
        walk(newPath)
      }
    })
}
```

```
walk(filesPath)
console.log(files.join('\r\n'))
```

编程练习 - 中英文 JSON 合并工具

当我们开发一个国际网站时候，有时候需要处理 i18n 的内容，而页面比较多的时候，我们不方便用一个大而全的 JSON 文件来囊括所有的翻译内容，我们可能会把文案按照页面区分后，最终再把它们拼成一份，比如有一个 pagea.json:

```
{
  "signup": "注册"
}
```

有一个 pageb.json:

```
{
  "menu": "菜单"
}
```

希望把它们合并成 data.json

```
{
  "signup": "注册",
  "menu": "菜单"
}
```

代码可以这样写：

```
const fs = require('fs')
const path = require('path')
// 判断目标路径的文件存在与否
const exists = filePath => fs.existsSync(filePath)
const jsonPath = process.argv[2]
if(!jsonPath) {
  console.log('没有传 JSON 目录参数哦! ')
  process.exit(1)
}
const rootPath = path.join(process.cwd(), jsonPath)
```

```
// 遍历所有文件
const walk = (path) => fs
  .readdirSync(path)
  .reduce((files, file) => {
    const filePath = path + '/' + file
    const stat = fs.statSync(filePath)
    if (stat.isFile()) {
      if(/(.*)\.(json)/.test(file)) {
        return files.concat(filePath)
      }
    }
    return files
  }, [])
// 合并文件内容
const mergeFileData = () => {
  const files = walk(rootPath)

  if(!files.length) process.exit(2)

  const data = files
    .filter(exists)
    .reduce(total, file => {
      const fileData = fs.readFileSync(file)
      const basename = path.basename(file, '.json')
      let fileJson

      try {
         fileJson = JSON.parse(fileData)
       } catch(err) {
         console.log('读取出错', file)
         console.log(err)
       }
       total[basename] = fileJson
       return total
    }, {})
  fs.writeFileSync('./data.json', JSON.stringify(data, null, 2))
}
mergeFileData()
```

总结

这一节的内容略显枯燥，我们对读写总结一下。无论读写，都有两种方式，一种粗矿的，一种精细化的。精细化的控制，需要先 open 一个文件，然后操作读写，但需要手工调用 close 方法关闭文件。这种方式适合于多次写入或读取。粗狂的读写是一次性服务的，直接调用 writeFile/appendFile/readFile 方法，只会写入或读取一次，在它的内部自动调用了 close 方法。另

外呢，对于 write 方法，因为多次对同一文件进行 write 并不安全，为了保证写入的安全性，就必须以 callback 的方式来保证下次写入顺序的正确性才可以保证文件写入内容的正确性，避免写时的冲突。官方推荐是使用 stream 方式替代，也就是 createWriteStream，关于 Stream 我们就放到后面的小节中学习。