Node基础篇

最简短的开场白

Node是什么

- 用于编写服务器端应用
- JavaScript核心语法
- 只是操作的对象不同

前端	DOM	文档对象
	BOM/DOM	浏览器对象
	XMLHttpRequest/fetch	网络通讯
后端	os	操作系统
	process	进程
	fs	文件系统
	net	网络通讯

API哪里找

英文 https://nodejs.org/dist/latest-v10.x/docs/api/

中文 http://nodejs.cn/api/

运行/调试/模块 - 如何搭建万里长城

Helloworld

```
console.log('hello world')
```

bash运行

```
node helloworld/index.js
# 或
node helloworld
```

Nodemon自动重启

监视代码修改，自动重启

```
npm i nodemon -g
nodemon helloworld
```

Vscode调试debug

单元测试Jest

安装jest库

```
npm install jest -g
```

在__*tests*__文件夹中创建index.spec.ts

```
test("Hello world", () => {
    require('../index')
});
```

运行

```
jest helloworld
```

Exports与Require

```
const str = 'helloworld'
// module.exports = str
// 或
// module.exports.str = str
// 或
exports.str = str
```

测试程序

```
it('测试Export', () => {
    const string = require('../index')
    console.log('export', string)
})
```

测试代码生成工具

> - 掌握fs中的同步方法
> - path包

生成测试文件名

```
test('测试文件名称',() => {
    const src = new (require('../index'))()
    const ret =
src.getTestFileName('/abc/class.js')
    console.log('getSourceName',ret)
    expect(ret)
    .toBe('/abc/__test__/class.spec.js')
})
```

```
const path = require('path')
module.exports = class TestNow {
    /**
     * 生成测试文件名
     * @param {*} filename
     */
    getTestFileName(filename) {
        const dirName =
path.dirname(filename)
        const baseName =
path.basename(filename)
        const extname =
path.extname(filename)
        const testName =
baseName.replace(extname,
`.spec${extname}`)
```

```
        return path.format({
            root: dirName + '/__test__/',
            base: testName
        })
    }
}
```

生成测试代码

```
test('生成测试代码', () => {
    const src = new (require('../index'))()
    const ret =
src.getTestSource('fun','class')
    expect(ret)
        .toBe(
            `

test('TEST fun',() => {
    const  fun = require('../class')
    const ret = fun()
    // expect(ret)
    //      .toBe('test ret')
})
        `

        )
```

```
})
```

异步编程 - 如何控制好异步过程

> 参考资料 阮一峰 Javascript异步编程的4种方法)

- JS的执行环境是单线程（Single thread）
- I/O处理需要回调函数异步处理 (异步I/O)
- 前端异步IO可以消除UI阻塞，提高用户体验
- 而放在后端则可以提高CPU和内存里利用率

串联异步处理

异步操作队列化，按照期望的顺序执行。

Callback

> 回调地域太可怕

```
const logTime = (name) => {
    console.log(`Log....${name}    ` + new
Date().toLocaleTimeString())
}

exports.callback = () => {
    setTimeout(() => {
        logTime('callback 1')
        setTimeout(() => {
            logTime('callback 2')
        }, 100)
    }, 100)
}
```

测试代码

```
test('callback', done => {
  callback()
  // 延时4s结束
  setTimeout(done, 1000)
})
```

Promise

> The Promise object is used for asynchronous computations. A Promise represents a single asynchronous operation that hasn't completed yet, but is expected in the future.
>
> 译文：Promise对象用于异步操作，它表示一个尚未完成且预计在未来完成的异步操作。

说白了就是一个异步执行的状态机，异步执行的承诺。

```
const promise = (name, delay = 100) => new
Promise(resolve => {
    setTimeout(() => {
        logTime(name)
        resolve()
    }, delay)
})

exports.promise = () => {

    promise('Promise1')
        .then(promise('Promise2'))
        .then(promise('Promise3'))
        .then(promise('Promise4'))
}
```

Gennerator

> ES6 新引入了 Generator 函数，可以通过 yield 关键字，把函数的执行流挂起，为改变执行流程提供了可能，从而为异步编程提供解决方案。 基本

- function -> function* 称为Gennerator函数
- 函数内部有 yield 表达式。

```
function* func() {
    console.log("one");
    yield '1';
    console.log("two");
    yield '2';
    console.log("three");
    return '3';
}

const f = func()
f.next();
// one
// {value: "1", done: false}
f.next();
// two
// {value: "2", done: false}
f.next();
// three
```

```
// {value: "3", done: true}
f.next();
// {value: undefined, done: true}

// 或者通过迭代器
for (const [key,value] of func()) {
    console.log(`${key}: ${value}`);
}
```

逻辑代码

```
let co = function (gen, name) {
    var it = gen(name)
    var ret = it.next()
    ret.value.then(function (res) {
        it.next(res)
    })
}
exports.generator = () => {
    const generator = function* (name) {
        yield promise(name + 1)
        yield promise(name + 2)
        yield promise(name + 3)
        yield promise(name + 4)
```

```
    }
    let co = generator => {
        if (it = generator.next().value) {
            it.then(res => {
                co(generator)
            })
        } else {
            return
        }
    }
    co(generator('Co-Generator'))
}
```

async/await

> async/await是es7推出的一套关于异步的终极解决方案

- 任何一个await语句后面的 Promise 对象变为reject状态，那么整个async函数都会中断执行。
- async函数返回的 Promise 对象，必须等到内部所有await命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数。

```
exports.asyncAwait = async () => {
    await promise('Async/Await1')
    await promise('Async/Await2')
    await promise('Async/Await3')
    await promise('Async/Await4')
}
```

事件监听方式处理

采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

```
exports.event = async () => {
    const asyncFun = name => event => {
        setTimeout(() => {
            logTime(name)
            event.emit('end')
        }, 100)
        return event
    }

    const ary = [
        asyncFun('event1'),
        asyncFun('event2'),
        asyncFun('event3')
```

```
    ]

    const { EventEmitter } =
require('events')
    const event = new EventEmitter()
    let i = 0
    event.on('end', () => i < ary.length &&
ary[i++](event))
    event.emit('end')

}
```

eventEmmiter

```
const promise = (name, delay = 100) => new
Promise(resolve => {
    setTimeout(() => {
        logTime(name)
        resolve()
    }, delay)
})

exports.promise = () => {
```

```
    promise('Promise1')
        .then(promise('Promise2'))
        .then(promise('Promise3'))
        .then(promise('Promise4'))
}
```

扩展阅读eventEimitter源码解析 / 订阅发布机制

```
class EventEmitter {
    constructor() {
        this.handler = {};
    }
    on(eventName, callback) {
        if (!this.handles) {
            this.handles = {};
        }
        if (!this.handles[eventName]) {
            this.handles[eventName] = [];
        }

 this.handles[eventName].push(callback);
    }
    emit(eventName, ...arg) {
        if (this.handles[eventName]) {
```

```
            for (var i = 0; i <
this.handles[eventName].length; i++) {
                this.handles[eventName][i]
(...arg);
            }
        }

    }
}

const event = new EventEmitter();
event.on('some_event', num =>  {
    console.log('some_event 事件触发:'+num);
});
let num = 0
setInterval(() =>  {
    event.emit('some_event' , num ++ );
}, 1000);
```

异步处理并行

promisify

IO处理

同步与异步读取文件

> 补充资料 https://nodejs.org/dist/latest-v10.x/docs/api/fs.html

fs 方法	描述
fs.truncate	截断或者拓展文件到制定的长度
fs.ftruncate	和 truncate 一样，但将文件描述符作为参数
fs.chown	改变文件的所有者以及组
fs.fchown	和 chown 一样，但将文件描述符作为参数

fs.lchown	和 chown 一样，但不解析符号链接
fs.stat	获取文件状态
fs.lstat	和 stat 一样，但是返回信息是关于符号链接而不是它指向的内容
fs.fstat	和 stat 一样，但将文件描述符作为参数
fs.link	创建一个硬链接
fs.symlink	创建一个软连接
fs.readlink	读取一个软连接的值
fs.realpath	返回规范的绝对路径名
fs.unlink	删除文件
fs.rmdir	删除文件目录
fs.mkdir	创建文件目录
fs.readdir	读取一个文件目录的内容
fs.close	关闭一个文件描述符
fs.open	打开或者创建一个文件用来读取或者写入
fs.utimes	设置文件的读取和修改时间

fs.futimes	和 utimes 一样，但将文件描述符作为参数
fs.fsync	同步磁盘中的文件数据
fs.write	写入数据到一个文件
fs.read	读取一个文件的数据

Promisify

buffer - 操作二进制数据

生成Base64编码

```
exports.imageUrl = () => {
    const mime = 'image/png'
    const encoding = 'base64'
    const base64Data =
fs.readFileSync(`${__dirname}/image.png`).t
oString(encoding)
    const uri =
`data:${mime};${encoding},${base64Data}`
    // 

fs.writeFileSync(`${__dirname}/index.html`,
`<img src='${uri}' />`)
    // console.log(uri)
}
```

stream - 流的操作

流是基于事件的 API，用于管理和处理数据。

- 流是能够读写的
- 是基于事件实现的一个实例

理解流的最好方式就是想象一下没有流的时候怎么处理数据：

- `fs.readFileSync` 同步读取文件，程序会阻塞，所有数据被读到内存
- `fs.readFile` 阻止程序阻塞，但仍会将文件所有数据读取到内存中
- 希望少内存读取大文件，读取一个数据块到内存处理完再去索取更多的数据

进程/子进程

process

```
// 查看PATH
console.log(process.env.PATH.split(':').join('\n'));
console.log(`arch:${process.arch}`)
console.log(`platform:${process.platform}`)
console.log(`获取内存使用情况memoryUsage:${process.memoryUsage()}`)
console.log(`获取命令行参数argv:${process.argv}`)
```

自动重启工具（Nodemon）

```
const fs = require('fs')
const { spawn } = require('child_process')
const { resolve } = require('path')
function watch() {
    const [cmd, , source, ...argv] =
process.argv
    // console.log('abc', cmd, source,
argv)
    const childProcess = spawn(cmd,
[source, ...argv])

 childProcess.stdout.pipe(process.stdout)

 childProcess.stderr.pipe(process.stderr)
    const watcher =
fs.watch(resolve(__dirname, source), () =>
{
        console.log('File changed,
reloading.')
        childProcess.kill()
        watcher.close()
        watch()
    })
}
```

```
watch()
```