nodejs命令行程序开发

### process对象

process是个全局对象，不需要require即可在任何地方使用。

#### 常用属性

* + 1. process.argv

返回一个数组

第一个元素是 process.execPath。 如果需要访问 argv[0] 的原始值，参阅 process.argv0。 第二个元素将是正在执行的 JavaScript 文件的路径。 其余元素将是任何其他命令行参数。

|  |
| --- |
| $ node process-args.js one two=three four  输出：  0: /usr/local/bin/node  1: /Users/mjr/work/node/process-args.js  2: one  3: two=three  4: four |

* + 1. process.env

返回一个对象，包含了当前 Shell 的所有环境变量；

例如：NODE\_ENV=production app.js

在程序中可以读取process.env.NODE\_ENV 值为production

* + 1. stdin/stdout

process. stdin 指向标准输入（键盘到缓冲区里的东西），返回一个可读的流；返回连接到 stdin (fd 0) 的流 ,它是一个双工流；

所以可以使用Stream的所有方法。

process.stdout 指向标准输出（向用户显示内容），返回一个可写的流

process.stdout.write相当于console.log

|  |
| --- |
| var fs = require('fs');  // 将一个文件导向标准输出  //**由于process.stdout和process.stdin与其他进程的通信，都是流（stream）形式，所以必须通过pipe管道命令中介。**  fs.createReadStream('wow.txt').**pipe**(process.stdout); |

* + 1. process.cwd()

返回运行 Node 的工作目录（绝对路径），即在哪个目录下运行的node

区别于\_\_dirname： \_\_dirname指执行文件时该文件在文件系统中所在的目录

* + 1. process.on()

process 对象部署了 **EventEmitter** 接口，可以使用 process.on() 方法监听各种事件，并指定回调函数

data事件：数据输出输入时触发

SIGINT事件：接收到系统信号SIGINT时触发，主要是用户按Ctrl + c时触发。

SIGTERM事件：系统发出进程终止信号SIGTERM时触发

exit事件：进程退出前触发

* + 1. process.exit()

会让 Node 立即终止当前进程（同步），参数为一个退出状态码，0 表示成功，大于 0 的任意整数表示失败。

### Child Process 模块

const cp = require('child\_process')

#### exec(command[, options][, callback])

方法用于执行 shell 命令

command: 要运行的命令 ，带空格

options:

cwd <string> 子进程的当前工作目录。默认值: null。

env <Object> 环境变量的键值对。默认值: null。

encoding <string> 默认值: 'utf8'。

shell <string> 用于执行命令的 shell。参阅 shell 的要求与 Windows 默认的 shell。 默认值: UNIX 上是 '/bin/sh'，Windows 上是 process.env.ComSpec。

timeout <number> 默认值: 0。

maxBuffer <number> stdout 或 stderr 上允许的最大字节数。如果超过限制，则子进程将终止。参阅 maxBuffer 与 Unicode。默认值: 200 \* 1024。

callback <Function> 当进程终止时调用。

error <Error>

stdout <string> | <Buffer> 子进程的 stdout

stderr <string> | <Buffer> 子进程的 stderr

encoding用于指定解码stdout和stderr；

|  |
| --- |
| const { exec } = require('child\_process');  exec('cat \*.js missing\_file | wc -l', (error, stdout, stderr) => {  if (error) {  console.error(`执行出错: ${error}`);  return;  }  console.log(`stdout: ${stdout}`);  console.log(`stderr: ${stderr}`);  }); |

#### spawn(command[, args][, options])

创建一个子进程执行特定命令，（返回的是个process对象）与 exec() 的区别是它没有回调函数，只能通过监听事件来获取运行结果，它适用于子进程长时间运行的情况，可以实时输出结果。它适用于子进程长时间运行的情况，可以实时输出结果。

|  |
| --- |
| const ls = cp.spawn('ls', ['-l']);  ls.stdout.on('data', (data) => {  console.log(`stdout: ${data}`);  });  ls.stderr.on('data', (data) => {  console.log(`stderr: ${data}`);  });  ls.on('close', (code) => {  console.log(`child process exited with code ${code}`);  }); |

使用 spawn 可以实现一个简单的守护进程，在工作进程不正常退出时重启工作进程：

|  |
| --- |
| /\* daemon.js \*/  function spawn(mainModule) {  const worker = cp.spawn('node', [ mainModule ]);  worker.on(**'exit'**, function (code) {  if (code !== 0) {  spawn(mainModule);  }  });  }  spawn('worker.js'); |

#### fork(modulePath[, args][, options])

fork() 用来创建一个子进程执行 node 脚本，fork('./child.js') 相当于 spawn('node', ['./child.js'])，区别在于 fork 会在父子进程之间建立一个通信管道（fork() 的返回值），用于进程间通信。对该通信管道对象可以监听 **message** 事件，用来获取子进程返回的信息，也可以向子进程发送信息。

|  |
| --- |
| /\* main.js \*/  const cp = require('child\_process');  const proc = cp.fork('./child.js');  proc.on('message', function(msg) {  console.log(`parent got message: ${msg}`);  });  proc.send({ hello: 'world' });  /\* child.js \*/  process.on('message', function(msg) {  console.log(`child got message: ${msg}`);  });  process.send({ foo: 'bar' }); |

### Cluster 模块

Node.js 默认单进程执行，但这样就无法利用多核计算机的资源，cluster 模块的出现就是为了解决这个问题的。在开发服务器程序时，可以通过 cluster 创建一个主进程和多个 worker 进程，让每个 worker 进程运行在一个核上，统一通过主进程监听端口和分发请求。

|  |
| --- |
| const cluster = require('cluster');  const http = require('http');  const numCPUs = require('os').cpus().length;  if (cluster.isMaster) {  console.log(`Master ${process.pid} is running`);  // Fork workers.  for (let i = 0; i < numCPUs; i++) {  **cluster.fork();**  }  cluster.on('exit', (worker, code, signal) => {  console.log(`worker ${worker.process.pid} died`);  });  } else {  // Workers can share any TCP connection  // In this case it is an HTTP server  http.createServer((req, res) => {  res.writeHead(200);  res.end(`hello world from ${process.pid}`);  }).listen(8000);  console.log(`Worker ${process.pid} started`);  } |

常用属性和方法：

#### isMaster/isWorker

cluster.isMaster 用来判断当前进程是否是主进程，cluster.isWorker 用来判断当前进程是否是工作进程，两者返回的都是布尔值

#### workers

cluster.workers 是一个包含所有 worker 进程的对象，key 为 worker.id，value 为 worker 进程对象。

#### fork([env])

cluster.fork() 方法用来新建一个 worker 进程，默认上下文复制主进程，只有主进程可调用。

常用事件：

#### listening

在工作进程调用 listen 方法后，会触发一个 listening 事件，这个事件可以被 cluster.on('listening') 监听。

#### exit

在工作进程挂掉时，会触发一个 exit 事件，这个事件可以被 cluster.on('exit') 监听。

|  |
| --- |
| cluster.on('exit', (worker, code, signal) => {  console.log('worker %d died (%s). restarting...',  worker.process.pid, signal || code);  cluster.fork(); //重启工作进程  }); |

#### worker对象：

worker 对象是 cluster.fork() 的返回值，代表一个 worker 进程。

worker.id：当前 worker 的唯一标识，也是保存在 cluster.workers 中的 key 值。

worker.process： 所有的 worker 进程都是通过 **child\_process.fork()** 生成的，这个进程对象保存在 worker.process 中。

worker.send()：用在主进程给子进程发送消息，在子进程中，使用 process.on() 监听消息并使用 process.send() 发送消息。

|  |
| --- |
| const cluster = require('cluster');  console.log(cluster.isMaster);  if (cluster.isMaster) {  const worker = cluster.fork();  worker.send('Hello world);  } else if (cluster.isWorker) {  process.on('message', (msg) => {  console.log(msg);  process.send(msg);  });  }  运行：node test.js  输出：  True  False  Hello world  首次执行时是主进程，会走if，此时fork了一个子进程，等同于重新执行了一遍node test.js 此时创建的是个子进程，所以走else，且子进程监听message事件，  在第5行，主进程向子进程send了一条消息，被子进程监听到，所以把message打印出来。 |

process.send() 用来给父进程发送消息;

worker.send():

主进程调用这个方法会发送消息给具体的工作进程。还有一个等价的方法是ChildProcess.send()。

工作进程调用这个方法会发送消息给主进程。还有一个等价方法是process.send()。

### yargs模块

npm install --save yargs

#### argv对象

|  |
| --- |
| #!/usr/bin/env node  var argv = require('yargs').argv;  console.log('hello ', argv.name); |

使用时，可以使用一下两种：

|  |
| --- |
| $ hello --name=tom  hello tom  $ hello --name tom  hello tom |

也就是说process.argv原始返回值如下：

|  |
| --- |
| $ node hello --name=tom  [ 'node',  '/path/to/myscript.js',  '--name=tom' ] |

yargs将上面的结果构建成一个对象，每个参数都是一个键值对：

|  |
| --- |
| var argv = require('yargs').argv;  // $ node hello --name=tom  // argv = {  // name: tom  // }; |

可以使用alias，指定n是name的别名：

|  |
| --- |
| #!/usr/bin/env node  var argv = require('yargs')  .alias('n', 'name')  .argv;  console.log('hello ', argv.n); |

这样 ，长参数和短参数都可以用了：

|  |
| --- |
| $ hello -n tom  hello tom  $ hello --name tom  hello tom |

### commander

#### version

.version('x.y.z') 设置命令程序的版本号

#### option

.option('-n, --name <name>', 'your name', 'GK')

* 第一个参数是选项定义，分为短定义和长定义。用 | 或 , 连接。

参数可以用<>或者[]修饰；

<> 表示option后面必须跟参数，[] 表示option后面可以不必跟参数

**当<>和[]都不使用时，使用该参数时，参数值为true，否则为false**

* 第二个参数为选项描述
* 第三个参数为选项参数默认值，可选。

|  |
| --- |
| #!/usr/bin/env node  var program = require('commander');  program  .option('-r, --resume', '简历')  .option('-l, --language <lang>', '这个语言是我擅长的语言。')  .parse(process.argv);  if (program.resume) {  console.log('简历'  + '-'  + '这个是我的简历！' + program.resume  );  }  // 用program.xxx可以拿到参数的值，xxx必须是长参数  if (program.resume) console.log(**program.resume**); // output: true  if (program.language) console.log(**program.language**); // output: 英语 |

.parse可以讲参数值赋予program.参数名

#### Unknown option

当接收到未定义的option时，程序会自动抛出错误

.allowUnknownOption() 取消自动报错机制

|  |
| --- |
| #!/usr/bin/env node  program  .**allowUnknownOption**()  .version('0.0.1')  .option('-r, --resume', '简历');  program.parse(process.argv); |

#### command

.command('init <path>', 'description')

第一个参数表示命令，可以使用<>或者[]修饰命令参数, <> 表示必传，[]表示可传

第二个参数可选，当不填时，将返回Command对象，若存在，将返回原型对象。

若无第二个参数，搭配使用action

若有第二个参数，则需要制定可执行文件！！

#### description

.description('command description') 用于描述命令

#### arguments

制定参数，<>为必传 []为可选，可以在action回调里获取到；

#### action

.action(fn)

用于设置命令执行的相关回调。fn可以接受命令的参数为函数形参，顺序与command()中定义的顺序一致。

#### parse

program.parse(process.argv)

此api一般是最后调用，用于解析process.argv。

#### outputHelp

program.outputHelp()

一般用于未录入参数时自动打印帮助信息。

### chalk

终端输出色彩化

### progress

渲染进度条