Node.js RESTful API Node.js JXcore 打包

Node.js 多进程

我们都知道 Node.js 是以单线程的模式运行的，但它使用的是事件驱动来处理并发，这样有助于我们在多核 cpu 的系统上创建多个子进程，从而提高性能。

每个子进程总是带有三个流对象：child.stdin, child.stdout 和child.stderr。他们可能会共享父进程的 stdio 流，或者也可以是独立的被导流的流对象。

Node 提供了 child_process 模块来创建子进程，方法有：

- **exec** - child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。
- **spawn** - child_process.spawn 使用指定的命令行参数创建新进程。
- **fork** - child_process.fork 是 spawn()的特殊形式，用于在子进程中运行的模块，如 fork('./son.js') 相当于 spawn('node', ['./son.js']) 。与spawn方法不同的是，fork会在父进程与子进程之间，建立一个通信管道，用于进程之间的通信。

---

exec() 方法

child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。

语法如下所示：

```
child_process.exec(command[, options], callback)
```

参数

参数说明如下：

command： 字符串， 将要运行的命令，参数使用空格隔开

options ：对象，可以是：

- cwd ，字符串，子进程的当前工作目录
- env，对象 环境变量键值对
- encoding ，字符串，字符编码（默认： 'utf8'）
- shell ，字符串，将要执行命令的 Shell（默认: 在 UNIX 中为`/bin/sh`， 在 Windows 中为`cmd.exe`， Shell 应当能识别`-c`开关在 UNIX 中，或 `/s /c` 在 Windows 中。 在Windows 中，命令行解析应当能兼容`cmd.exe`）
- timeout，数字，超时时间（默认： 0）
- maxBuffer，数字， 在 stdout 或 stderr 中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死 （默认: 200*1024）
- killSignal ，字符串，结束信号（默认：'SIGTERM'）

- uid，数字，设置用户进程的 ID
- gid，数字，设置进程组的 ID

callback ：回调函数，包含三个参数error, stdout 和 stderr。

exec() 方法返回最大的缓冲区，并等待进程结束，一次性返回缓冲区的内容。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');
for(var i=0; i<3; i++) {
var workerProcess = child_process.exec('node support.js '+i, function (error, stdout, stderr) {
if (error) {
console.log(error.stack);
console.log('Error code: '+error.code);
console.log('Signal received: '+error.signal);
}
console.log('stdout: ' + stdout);
console.log('stderr: ' + stderr);
});
workerProcess.on('exit', function (code) {
console.log('子进程已退出，退出码 '+code);
});
}
```

执行以上代码，输出结果为：

```
$ node master.js
子进程已退出，退出码 0
stdout: 进程 1 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 0 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 2 执行。

stderr:
```

spawn() 方法

child_process.spawn 使用指定的命令行参数创建新进程，语法格式如下：

```
child_process.spawn(command[, args][, options])
```

参数

参数说明如下：

command： 将要运行的命令

args： Array 字符串参数数组

options Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- stdio Array|String 子进程的 stdio 配置
- detached Boolean 这个子进程将会变成进程组的领导
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

spawn() 方法返回流 (stdout & stderr)，在进程返回大量数据时使用。进程一旦开始执行时 spawn() 就开始接收响应。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');
for(var i=0; i<3; i++) {
var workerProcess = child_process.spawn('node', ['support.js', i]);
workerProcess.stdout.on('data', function (data) {
console.log('stdout: ' + data);
});
workerProcess.stderr.on('data', function (data) {
console.log('stderr: ' + data);
});
workerProcess.on('close', function (code) {
console.log('子进程已退出，退出码 '+code);
});
}
```

执行以上代码，输出结果为：

```
$ node master.js stdout: 进程 0 执行。

子进程已退出，退出码 0
stdout: 进程 1 执行。
```

```
子进程已退出，退出码 0
stdout: 进程 2 执行。

子进程已退出，退出码 0
```

fork 方法

child_process.fork 是 spawn() 方法的特殊形式，用于创建进程，语法格式如下：

```
child_process.fork(modulePath[, args][, options])
```

参数

参数说明如下：

modulePath： String，将要在子进程中运行的模块

args： Array 字符串参数数组

options：Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- execPath String 创建子进程的可执行文件
- execArgv Array 子进程的可执行文件的字符串参数数组（默认： process.execArgv ）
- silent Boolean 如果为`true`，子进程的`stdin`，`stdout`和`stderr`将会被关联至父进程，否则，它们将会从父进程中继承。（默认为：`false`）
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

返回的对象除了拥有ChildProcess实例的所有方法，还有一个内建的通信信道。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');
for(var i=0; i<3; i++) {
var worker_process = child_process.fork("support.js", [i]);
worker_process.on('close', function (code) {
console.log('子进程已退出，退出码 ' + code);
});
}
```

执行以上代码，输出结果为：

```
$ node master.js
进程 0 执行。
子进程已退出，退出码 0
进程 1 执行。
子进程已退出，退出码 0
进程 2 执行。
子进程已退出，退出码 0
```

← Node.js RESTful API

Node.js JXcore 打包 →