link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=181 sentences=530, words=2721

1. 进程

- 在Node.js中每个应用程序都是一个进程类的实例对象。
- 使用 process对象代表应用程序,这是一个全局对象，可以通过它来获取Node.jsy应用程序以及运行该程序的用户、环境等各种信息的属性、方法和事件。

1.1 进程对象属性

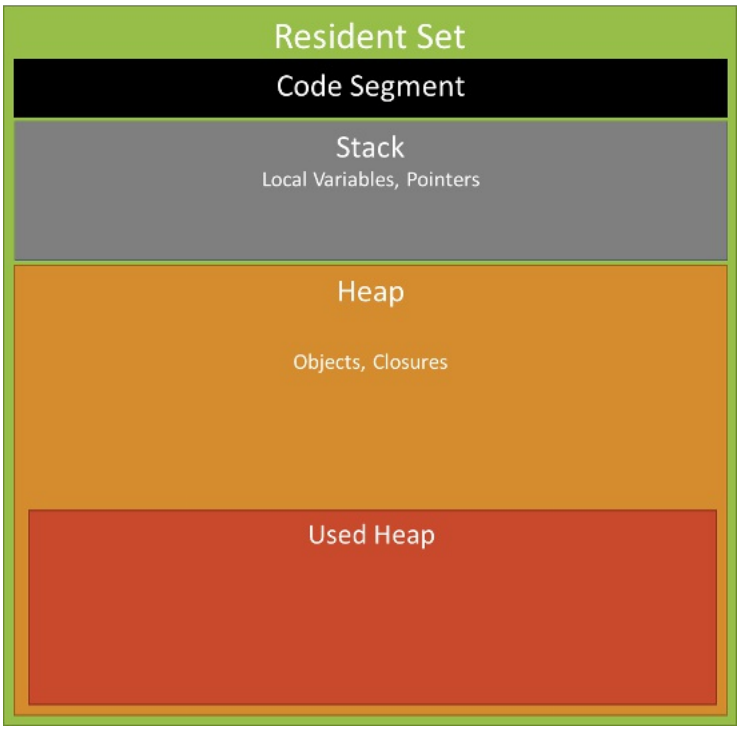
- execPath 可执行文件的绝对路径,如 /usr/local/bin/node
- version 版本号
- versions 依赖库的版本号
- platform 运行平台。 如 darwin、freebsd、linux、sunos、win32
- stdin 标准输入流可读流，默认暂停状态
- stdout 标准输出可写流，同步操作
- stderr 错误输出可写流，同步操作
- argv 属性值为数组
- env 操作系统环境信息
- pid 应用程序进程ID
- title 窗口标题
- arch 处理器架构 arm ia32 x64

```
process.stdin.resume();
process.stdin.on('data',function(chunk){
  process.stdout.write(`进程接收到数据： `+chunk);
});
```

```
process.argv.forEach((val,index,ary)=> console.log(index,val));
```

1.2 memoryUsage方法

```
process.memoryUsage()
```



- rss（resident set size）：所有内存占用，包括指令区和堆栈。
- heapTotal："堆"占用的内存，包括用到的和没用到的。
- heapUsed：用到的堆的部分。
- external： V8 引擎内部的 C++ 对象占用的内存。

1.3 nextTick方法

nextTick方法用于将一个函数推迟到代码中所书写的下一个同步方法执行完毕或异步方法的回调函数开始执行前调用

1.4 chdir

chdir方法用于修改Node.js应用程序中使用的当前工作目录，使用方式如下

```
process.chdir(directory);
```

1.5 cwd 方法

cwd方法用返回当前目录，不使用任何参数

```
console.log(process.cwd());
```

1.6 chdir 方法

改变当前的工作目录

```
console.log(`当前目录: ${process.cwd()}`);
process.chdir('..');
console.log(`上层目录: ${process.cwd()}`);
```

1.7 exit 方法

退出运行Node.js应用程序的进程

```
process.exit(0);
```

1.8 kill方法

用于向进程发送一个信号

- SIGINT 程序终止(interrupt)信号, 在用户键入INTR字符(通常是Ctrl-C)时发出，用于通知前台进程组终止进程。
- SIGTERM 程序结束(terminate)信号, 该信号可以被阻塞和处理。通常用来要求程序自己正常退出，shell命令kill缺省产生这个信号

```
process.kill(pid,[signal]);
```

- pid是一个整数，用于指定需要接收信号的进程ID
- signal 发送的信号，默认为 SIGTERM

1.9 uptime

返回当前程序的运行时间

```
process.uptime()
```

1.10 hrtime

测试一个代码段的运行时间,返回两个时间，第一个单位是秒，第二个单位是纳秒

```
let fs = require('fs);
let time = process.hrtime();
let data = fs.readFileSync('index.txt');
let diff = process.hrtime(time);
console.log(`读文件操作耗费的%d秒`,diff[0]);
```

1.11 exit事件

当运行Node.js应用程序进程退出时触发进程对象的exit事件。可以通过指定事件回调函数来指定进程退出时所执行的处理。

```
process.on('exit',function(){
    console.log('Node.js进程被推出);
});
process.exit();
```

1.12 uncaughtException事件

当应用程序抛出一个未被捕获的异常时触发进程对象的uncaughtException事件

```
process.on('uncaughtException',function(err){
  console.log('捕获到一个未被处理的错误:',err);
});
notExist();
```

1.13 信号事件

```
process.stdin.resume();
process.on('SIGINT',function(){
    console.log('接收到SIGINT信号');
});
```

2. 子进程

- 在Node.js中，只有一个线程执行所有操作，如果某个操作需要大量消耗CPU资源的情况下，后续操作都需要等待。
- 在Node.js中，提供了一个 child_process模块,通过它可以开启多个子进程，在多个子进程之间可以共享内存空间，可以通过子进程的互相通信来实现信息的交换。**2.1 spawn #****2.1.1 语法#**

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

- command 必须指定的参数，指定需要执行的命令
- args 数组，存放了所有运行该命令需要的参数
- options 参数为一个对象，用于指定开启子进程时使用的选项
    - cwd 子进程的工作目录
    - env 环境变量
    - detached 如果为true,该子进程魏一个进程组中的领头进程，当父进程不存在时也可以独立存在
    - stdio 三个元素的数组，设置标准输入/输出
        - pipe 在父进程和子进程之间创建一个管道，父进程可以通过子进程的stdio[0]访问子进程的标准输入，通过stdio[1]访问标准输出,stdio[2]访问错误输出
        - ipc 在父进程和子进程之间创建一个专用与传递消息的IPC通道。可以调用子进程的send方法向子进程发消息，子进程会触发 message事件
        - ignore 指定不为子进程设置文件描述符。这样子进程的标准输入、标准输出和错误输出被忽略
        - stream 子进程和父进程共享一个终端设备、文件、端口或管道
        - 正整数值 和共享一个steam是一样的
        - null或undefined 在子进程中创建与父进程相连的管道

默认情况下，子进程的stdin,stdout,stderr导向了ChildProcess这个对象的child.stdin,child.stdout,child.stderr流,

```
let spawn = require('child_process').spawn;
sapwn('prg',[],{stdio:['pipe','pipe',process.stderr]});
```

- ignore ['ignore','ignore','ignore'] 全部忽略
- pipe ['pipe','pipe','pipe'] 通过管道连接
- inherit [process.stdin,process.stdout,process.stderr]或[0,1,2] 和父进程共享输入输出

```
let spawn = require('child_process').spawn;
spawn('prg',[],{stdio:'inherit'});
```

- spawn方法返回一个隐式创建的代表子进程的ChildProcess对象
- 子进程对象同样拥有stdin属性值为一个可用于读入子进程的标准输入流对象
- 子进程对象同样拥有stdiout属性值和stderr属性值可分别用于写入子进程的标准输出流与标准错误输出流

** 2.1.2 close #**

- 当子进程所有输入输出都终止时，会触发子进程对象的close事件。

```
child.on('close',function(code,signal){});
```

- code 为0表示正常推出，为null表示异常退出
- 当在父进程中关闭子进程时，signal参数值为父进程发给子进程的信号名称

** 2.1.3 exit #**

- 当子进程退出时，触发子进程对象的exit事件
- 因为多个进程可能会共享i 个输入/输出，所以当子进程退出时，子进程的输入/输出可能并未终止

```
child.on('exit',function(code,signal){});
```

2.1.4 error # 如果子进程开启失败，那么将会触发子进程对象的error事件

```
child1.on('error', function (err) {
  console.log(err);
});
```

2.1.5 kill #

- 父进程还可以使用kill方法向子进程发送信号,参数为描述该信号的字符串，默认参数值为 SIGTERM
- SIGTERM 程序结束(terminate)信号, 与SIGKILL不同的是该信号可以被阻塞和处理. 通常用来要求程序自己正常退出

```
child.kill([signal]);
```

** 2.1.6 案例 #**

1. spawn.js

```
let path = require('path');
let {
 spawn
} = require('child_process');

let p1 = spawn('node', ['test1.js', 'a'], {
 cwd: path.join(__dirname, 'test1')
});
let p2 = spawn('node', ['test3.js'], {
 cwd: path.join(__dirname, 'test3'),
 stdio: 'pipe'
});

p1.stdout.on('data', function (data) {
 console.log('P1:子进程的标准输出:' + data);
 p2.stdin.write(data);
});
p1.on('error', function () {
 console.log('p1:子进程1开启失败');
});
p2.on('error', function () {
 console.log('p2:子进程2开启失败');
});
```

2. test1.js

```
process.stdout.write('p1:子进程当前工作目录为:' + process.cwd() + '\r\n');
process.stdout.write('p1:' + process.argv[2] + ' \r\n');
```

3. test2.js

```
let fs = require('fs');
let path = require('path');
let out = fs.createWriteStream(path.join(__dirname, 'msg.txt'));
process.stdin.on('data', function (data) {
 out.write(data);
});
process.stdin.on('end', function () {
 process.exit();
});
```

** 2.1.7 detached #**

- 在默认情况下，只有在子进程全部退出后，父进程才能退出。为了让父进程可以先退出，而让子进程继续进行I/O操作,可以在spawn方法中使用options参数，把detached属性值设置为true
- 默认情况下父进程会等待所有的子进程退出后才可以退出，使用subprocess.unref方法可以让父进程不用等待子进程退出就可以直接退出

```
let cp = require('child_process');
let fs = require('fs');
let path = require('path');
let out = fs.openSync(path.join(__dirname, 'msg.txt'), 'w', 0o666);
let sp = cp.spawn('node', ['4.detached.js'], {
  detached: true,
  stdio: ['ignore', out, 'ignore']
});
sp.unref();
```

```
let count = 10;
let $timer = setInterval(() => {
    process.stdout.write(new Date().toString() + '\r\n');
    if (--count == 0) {
        clearInterval($timer);
    }
}, 500);
`
```

** 2.2 fork开启子进程 #**

- 衍生一个新的 Node.js 进程，并通过建立一个 IPC 通讯通道来调用一个指定的模块，该通道允许父进程与子进程之间相互发送信息
- fork方法返回一个隐式创建的代表子进程的ChildProcess对象
- 子进程的输入/输出操作执行完毕后，子进程不会自动退出，必须使用 process.exit() 方法显式退出

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

- args 运行该文件模块文件时许哟啊使用的参数
- options 选项对象
    - cwd 指定子进程当前的工作目录
    - env 属性值为一个对象，用于以"键名/键值"的形式为子进程指定环境变量
    - encoding 属性值为一个字符串，用于指定输出及标准错误输出数据的编码格式，默认值为'utf8'
    - silent 属性值为布尔值，当属性值为false时，子进程和父进程对象共享标准(输入/输出),true时不共享

** 2.2.1 发送消息 #**

```
child.send(message,[sendHandle]);
process.send(message,[sendHandle]);
```

- message是一个对象，用于指定需要发送的消息
- sendHandle是一个 net.Socket 或 net.Server 对象
- 子进程可以监听父进程发送的message事件

```
process.on('message',function(m,setHandle){});
```

- m 参数值为子进程收到的消息
- sendHandle为服务器对象或socket端口对象

当父进程收到子进程发出的消息时，触发子进程的message事件

```
child.on('message',function(m,setHandle){

});
```

5.fork.js

```
let {
    fork
} = require('child_process');
let path = require('path');
let child = fork(path.join(__dirname, 'fork.js'));
child.on('message', function (m) {
    console.log('父进程接收到消息:', m);
    process.exit();
});
child.send({
    name: 'zfpx'
});
child.on('error', function (err) {
    console.error(arguments);
});
```

fork.js

```
process.on('message', function (m, setHandle) {
    console.log('子进程收到消息:', m);
    process.send({
        age: 9
    });
})
```

** 2.2.2 silent #**

在默认情况下子进程对象与父进程对象共享标准输入和标准输出。如果要让子进程对象用独立的标准输入输出，可以将silent属性值设置为 true forkslient.js

```
let {
    fork
} = require('child_process');
let path = require('path');

let p1 = fork('node', [path.join(__dirname, 'fork1.js')], {
    silent: true
});
let p2 = fork('node',path.join(__dirname, 'fork2.js'));
p1.stdout.on('data', function (data) {
    console.log('子进程1标准输出: ' + data);
    p2.send(data.toString());
});
p1.on('exit', function (code, signal) {
    console.log('子进程退出，退出代码为:' + code);
});
p1.on('error', function (err) {
    console.log('子进程开启失败:' + err);
    process.exit();
});
```

fork1.js

```
process.argv.forEach(function (item) {
    process.stdout.write(item + '\r\n');
});
```

fork2.js

```
let fs = require('fs');
let out = fs.createWriteStream(path.join(__dirname, 'msg.txt'));
process.on('message', function (data) {
    out.write(data);
});
```

** 2.2.3 子进程与父进程共享HTTP服务器 #**

```
let http = require('http');
let {
    fork
} = require('child_process');
let fs = require('fs');
let net = require('net');
let path = require('path');
let child = fork(path.join(__dirname, '8.child.js'));
let server = net.createServer();
server.listen(8080, '127.0.0.1', function () {
    child.send('server', server);
    console.log('父进程中的服务器已经创建');
    let httpServer = http.createServer();
    httpServer.on('request', function (req, res) {
        if (req.url != '/favicon.ico') {
            let sum = 0;
            for (let i = 0; i < 100000; i++) {
                sum += 1;
            }
            res.write('客户端请求在父进程中被处理。');
            res.end('sum=' + sum);
        }
    });
    httpServer.listen(server);
});
```

```
let http = require('http');
process.on('message', function (msg, server) {
    if (msg == 'server') {
        console.log('子进程中的服务器已经被创建');
        let httpServer = http.createServer();
        httpServer.on('request', function (req, res) {
            if (req.url != '/favicon.ico') {
                sum = 0;
                for (let i = 0; i < 10000; i++) {
                    sum += i;
                }
                res.write('客户端请求在子进程中被处理');
                res.end('sum=' + sum);
            }
        });
        httpServer.listen(server);
    }
});
```

```
let http = require('http');
let options = {
    hostname: 'localhost',
    port: 8080,
    path: '/',
    method: 'GET'
}
for (let i = 0; i < 10; i++) {
    let req = http.request(options, function (res) {
        res.on('data', function (chunk) {
            console.log('响应内容:' + chunk);
        });
    });
    req.end();
}
```

** 2.2.4 子进程与父进程共享socket对象 #**

```
let {
    fork
} = require('child_process');
let path = require('path');
let child = fork(path.join(__dirname, '11.socket.js'));
let server = require('net').createServer();
server.on('connection', function (socket) {
    if (Date.now() % 2 == 0) {
        child.send('socket', socket);
    } else {
        socket.end('客户端请求被父进程处理!');
    }
});
server.listen(41234, );
```

```
process.on('message', function (m, socket) {
    if (m === 'socket') {
        socket.end('客户端请求被子进程处理.');
    }
});
```

```
let net = require('net');
let client = new net.Socket();
client.setEncoding('utf8');
client.connect(41234, 'localhost');
client.on('data', function (data) {
    console.log(data);
});
```

** 2.3 exec开启子进程 #**

- exec方法可以开启一个用于运行某个命令的子进程并缓存子进程的输出结果
- spawn是一个异步方法，exec是一个同步方法
- 衍生一个 shell 并在 shell 上运行命令

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

- command 需要执行的命令
- options 选项对象
    - cwd 子进程的当前工作目录
    - env 指定子进程的环境变量
    - encoding 指定输出的编码

    - timeout 子进程的超时时间
    - maxbuffer 指定缓存标准输出和错误输出的缓存区最大长度
    - killSignal 指定关闭子进程的信号，默认值为 "SIGTERM"

- callback 指定子进程终止时调用的回调函数

```
function(err,stdout,stderr){}
```

- err 错误对象
- stdout 标准输出
- stderr 错误输出

```
let {
    exec
} = require('child_process');
let path = require('path');
let p1 = exec('node test1.js a b c', {
    cwd: path.join(__dirname, 'test3')
}, function (err, stdout, stderr) {
    if (err) {
        console.log('子进程开启失败:' + err);
        process.exit();
    } else {
        console.log('子进程标准输出\r\n' + stdout.toString());
        p2.stdin.write(stdout.toString());
    }
});
let p2 = exec('node test2.js', {
    cwd: path.join(__dirname, 'test3')
}, function (err, stdout, stderr) {
    process.exit();
});
```

```
let path = require('path');
process.argv.forEach(function (item) {
    process.stdout.write(item + '\r\n');
});
```

```
let fs = require('fs');
let path = require('path');
let out = fs.createWriteStream(path.join(__dirname, 'msg.txt'));
process.stdin.on('data', function (data) {
    out.write(data);
    process.exit();
})
```

** 2.4 execFile开启子进程 #**

- 可以使用 execFile开启一个专门用于运行某个可执行文件的子进程
- 类似 child_process.exec()，但直接衍生命令，且无需先衍生一个 shell

```
child_process.execFile(file,[args],[optioins],[callback]);
```

- file 指定需要运行的可执行文件路径及文件名
- args 运行该文件所需要的参数
- options 开启子进程的选项
- callback 指定子进程终止时调用的回调函数

```
let {
    execFile
} = require('child_process');
let path = require('path');

let p1 = execFile('node', ['./test1.js'], {
    cwd: path.join(__dirname, 'test4')
}, function (err, stdout, stderr) {
    if (err) {
        console.log('子进程1开启失败:' + err);
        process.exit();
    } else {
        console.log('子进程标准输出:' + stdout.toString());
        p2.stdin.write(stdout.toString());
    }
});
let p2 = execFile('node', ['./test2.js'], {
    cwd: path.join(__dirname, 'test4')
}, function (err, stdout, stderr) {
    if (err) {
        console.log('子进程2开启失败:' + err);
        process.exit();
    } else {
        console.log('子进程标准输出:' + stdout.toString());
    }
});
```

```
let path = require('path');
process.argv.forEach(function (item) {
    process.stdout.write(item + '\r\n');
});
```

```
let fs = require('fs');
let path = require('path');
let out = fs.createWriteStream(path.join(__dirname, 'msg.txt'));
process.stdin.on('data', function (data) {
    out.write(data);
    process.exit();
})
```

3. cluster

为了利用多核CPU的优势，Node.js提供了一个cluster模块允许在多个子进程中运行不同的Node.js应用程序。
** 3.1 fork方法创建work对象 #**

- 可以使用fork方法开启多个子进程，在每个子进程中创建一个Node.js应用程序的实例，并且在该应用程序中运行一个模块文件。

- fork方法返回一个隐式创建的work对象
- 在cluster模块中，分别提供了一个isMaster属性与一个isWork属性，都是布尔值

```
cluster.fork([env]);
```

- env 为子进程指定环境变量

** 3.1.1 获取所有的worker #**

```
for(let index in cluster.workers){
    console.log(cluster.workers[index]);
}
```

** 3.1.2 获取当前的worker和id #**

```
if(cluster.isMaster){
  cluster.fork()
}else if(cluster.isWorker){
  console.log('I am worker #'+cluster.worker.id);
}
```

** 3.1.3 服务器 #**

```
let cluster = require('cluster');
let http = require('http');
if (cluster.isMaster) {
    cluster.fork();
    console.log('这段代码运行在主进程里');
} else {
    http.createServer(function (req, res) {
        if (req.url != '/favicon.ico') {
            res.end('hello');
            console.log('这段代码运行在子进程里');
        }
    }).listen(8080);
}
```

** 3.1.4 fork事件 #**

当使用fork方法开启子进程时，将同时触发fork事件

```
cluster.on('fork',function(worker){
  console.log('子进程 '+workder.id+'被开启');
});
```

** 3.1.5 online事件 #**

- 在使用fork方法开启一个新的用于运行Node.js应用程序的子进程后，该应用程序将通过向主进程发送反馈信息，当主进程接收到该反馈信息后，触发online事件

```
cluster.on('online',function(worker){
  console.log('已经收到子进程#'+workder.id+"的消息");
});
```

** 3.1.6 listening #**

当在子进程运行的Node.js应用程序中调用服务器的listen方法后，该服务器开始对指定地址及端口进行监听，同时触发listening事件。

```
let cluster = require('cluster');
let http = require('http');
if (cluster.isMaster) {
    cluster.fork();
    console.log('这段代码运行在主进程里');
} else {
    http.createServer(function (req, res) {
        if (req.url != '/favicon.ico') {
            res.end('hello');
            console.log('这段代码运行在子进程里');
        }
    }).listen(8080, 'localhost');
}
cluster.on('online', function (worker) {
    console.log('已经收到子进程#' + worker.id + "的消息");
});
cluster.on('listening', function (worker, address) {
    console.log('子进程中的服务器开始监听,地址为:' + address.address + ":" + address.port);
});
```

** 3.1.7 setupMaster #**

子进程中的Node.js应用程序默认运行当前正在运行的Node.js应用程序中的主模块文件。可以使用setupMaster方法修改子进程中运行的模块文件

```
cluster.setupMaster([settings]);
```

- settings设置子进程中运行的Node.js应用程序各种默认行为的对象
    - exec 子进程运行的模块文件名称的完整路径及文件名
    - args 属性为一个数组，其中存放了所有运行子进程的Node.js运行程序所需要的参数
    - silent 布尔值。当属性值为false时，子进程对象与主进程对象共享标准输入/输出

```
let cluster = require('cluster');
cluster.setupMaster({
    exec: 'subtask.js'
});
cluster.fork();
console.log('这段代码被运行子主进程中');
console.log('cluster.settings属性值: %j', cluster.settings);
```

```
let http = require('http');
http.createServer(function (req, res) {
    if (req.url != '/favicon.ico') {
        res.writeHead(200);
        res.end('ok');
        console.log('这段代码被运行在子进程中');
    }
}).listen(8080);
```

** 3.1.8 在子进程里运行服务器 #**

当在子进程里运行服务器时，客户端总是先被主进程接收，然后转发给子进程中的服务器。如果在多个子进程中运行服务器，当主进程接收到客户端请求后，将会自动分配给一个当前处于空闲状态的子进程。

```
let cluster = require('cluster');
let http = require('http');
if (cluster.isMaster) {
    cluster.fork();
    cluster.fork();
} else {
    http.createServer(function (req, res) {
        if (req.url !== '/favicon.ico') {
            let sum = 0;
            for (let i = 0; i < 1000000; i++) {
                sum += i;
            }
            res.writeHead(200);
            res.write(`客户端请求在子进程${cluster.worker.id}中被处理`);
            res.end(`子进程${cluster.worker.id}中的计算结果=${sum}`);
        }
    }).listen(8080);
}
```

** 3.1.9 在子进程使用单独的输出 #**

```
let cluster = require('cluster');
let http = require('http');
if (cluster.isMaster) {
    cluster.setupMaster({
        silent: true
    });
    let worker = cluster.fork();
    worker.process.stdout.on('data', function (data) {
        console.log('接收到来自客户端的请求，目标地址:' + data);
    });
} else {
    http.createServer(function (req, res) {
        if (req.url !== '/favicon.ico') {
            let sum = 0;
            for (let i = 0; i < 1000000; i++) {
                sum += i;
            }
            res.writeHead(200);
            console.log(`客户端请求在子进程${cluster.worker.id}中被处理`);
            res.write(`客户端请求在子进程${cluster.worker.id}中被处理`);
            res.end(`子进程${cluster.worker.id}中的计算结果=${sum}`);
        }
    }).listen(8080);
}
```

** 3.2 worker对象 #**** 3.2.1 online #**

当新建一个工作进程后，工作进程应当响应一个online消息给主进程。当主进程收到online消息后触发这个事件

```
let cluster = require('cluster');
let http = require('http');
if (cluster.isMaster) {
    let worker = cluster.fork();
    console.log('这段代码被运行在主进程里');
    worker.on('online', function () {
        console.log(`已经收到子进程${worker.id}的运行信息`);
    });
} else {
    http.createServer(function (req, res) {
        if (req.url !== '/favicon.ico') {
            let sum = 0;
            for (let i = 0; i < 1000000; i++) {
                sum += i;
            }
            res.end('ok');
            console.log(`这段代码被运行在子进程中。`);
        }
    }).listen(8080);
}
```

** 3.2.2 send #**

在使用fork发放开启子进程后，可以使用fork方法所返回的worker对象的send方法在主进程中向子进程发送消息。

```
worker.send(message,[sendHandle]);
process.send(message,[sendHandle]);
process.on('message',function(m,setHandle){});
```

```
let cluster = require('cluster');
cluster.setupMaster({
    exec: 'child.js'
});
let worker = cluster.fork();
worker.on('message', function (m) {
    console.log('父进程接收到消息:', m);
    process.exit();
});
worker.send({
    name: 'zfpx'
});
```

** 3.2.3 共享socket #**

```
let http = require('http');
let cluster = require('cluster');
let net = require('net');
cluster.setupMaster({
    exec: '22.subsocket.js'
});
let worker = cluster.fork();
let server = require('net').createServer();
server.on('connection', function (socket) {
    if (Date.now() % 2 == 0) {
        worker.send('socket', socket);
    } else {
        socket.end('客户端的请求在主进程中处理');
    }

});
server.listen(41234, 'localhost');
worker.on('message', function (m, socket) {
    console.log(m);

});
```

```
process.on('message', function (msg, socket) {
    if (msg == 'socket') {
        socket.end('子进程中返回消息:' + msg);
        process.send('告诉父进程我处理了一个消息');
    }
});
```

** 3.2.4 kill #**

当使用fork方法开启子进程后，可以使用fork方法返回的worker对象的kill方法强制关闭子进程

```
worker.kill([signal]);
```

- signal 强制关闭子进程的信号字符串。默认参数值为 "SIGTERM"

** 3.2.5 exit #**

当子进程退出时，将会触发worker对象的exit事件

```
worker.on('exit',function(code,signal));
```

- code 退出代码。正常退出为0，异常退出为null
- worker.exitedAfterDisconnect可以用于区分自发退出还是被动退出，主进程可以根据这个值决定是否重新衍生新的工作进程。

```
let cluster = require('cluster');
let http = require('http');
if (cluster.isMaster) {
    cluster.setupMaster({
        silent: true
    });
    let worker = cluster.fork();
    worker.process.stdout.on('data', function (data) {
        console.log('接收到来自客户端的请求,目标地址为:' + data);
        worker.kill();
    });
    worker.on('exit', function (code, signal) {
        console.log(arguments)

        if (worker.exitedAfterDisconnect  == true) {
            console.log(`子进程${worker.id}自动退出`);
        } else if (worker.exitedAfterDisconnect  == false) {
            console.log(`子进程${worker.id}异常退出，退出代码为${code}`);
        }
        if (signal) {
            console.log('退出信号为 %s', signal);
        }
    });
} else {
    let server = http.createServer(function (req, res) {
        if (req.url != '/favicon.ico') {
            res.end('hello');
            process.stdout.write(req.url);
        }
    }).listen(8080);
}
```

** 3.2.6 disconnect #**

可以使用worker对象的disconnect方法使该子进程不再接收外部连接

```
let cluster = require('cluster');
let http = require('http');
if (cluster.isMaster) {
    cluster.setupMaster({ silent: true });
    let worker = cluster.fork();
    worker.process.stdout.on('data', function (data) {
        console.log('接收到来自客户端的请求,目标地址为:' + data);
        setTimeout(function () {
            worker.disconnect();
            worker.send('disconnect');
        }, 2000);
    });

    worker.on('disconnect', function () {
        console.log(`子进程${worker.id}断开连接`);
    });
    worker.on('exit', function (code, signal) {
        if (worker.exitedAfterDisconnect) {
            console.log(`子进程${worker.id}正常退出`);
        } else {
            console.log(`子进程${worker.id}异常退出`);
        }
        if (signal) {
            console.log(' 退出信号为' + signal);
        }
    });

} else {
    let http = require('http');
    let server = http.createServer(function (req, res) {
        if (req.url != '/favicon.ico') {
            res.end('hello');
            process.stdout.write(req.url);
        }
    }).listen(8080);
    process.on('message', function (msg) {
        if (msg == 'disconnect') {

        }

    });
}
```