第八章：创建并管理外部进程

本章内容：

- 使用外部命令和子进程
- 与子进程发送和接收数据
- 发送信号和终结子进程

Node 被设计用来高效的处理 I/O 操作，但是你应该知道，有些类型的程序并不适合这种模式。比如，如果你打算用 Node 处理一个 CPU 密集的任务，你可能会堵塞事件循环，并因此降低了程序的响应。替代办法是，把 CPU 密集的任务分配给一个单独的进程来处理，从而释放事件循环。Node 允许你产生进程，并把这个新进程做为它父进程的子进程。在 Node 里，子进程可以和父进程进行双向通信，而且在某种程度上，父进程还可以监控和管理子进程。

另外一种需要使用子进程的情况是，当你想简单地执行一个外部命令，并让 Node 获取命令的返回值时。比如，你可以执行一个 UNIX 命令、脚本或者其他那些不能在 Node 里直接执行的命令。

本章将向你展示如何执行外部命令，创建，并和子进程通信，以及终结子进程。重点是让你了解如何在 Node 进程外完成一系列任务。

执行外部命令

当你需要执行一个外部 shell 命令或可执行文件时，你可以使用 child_process 模块，像这样导入它：

```
var child_process = require('child_process')
```

然后可以用模块内的 exec 函数来执行外部命令：

```
var exec = child_process.exec;
exec(command,callback);
```

exec 的第一个参数是你准备执行的 shell 命令字符串，第二个参数是一个回调函数。这个回调函数将会在 exec 执行完外部命令或者有错误发生时被调用。回调函数有三个参数：error,stdout,stderr，看下面的例子：

```
exec('ls',function(err,stdout,stderr){
    //译者注：如果使用 windows，可改为 windows 命令，比如 dir，后面不再赘述
});
```

如果有错误发生，第一个参数将会是一个 Error 类的实例，如果第一个参数不包含错误，那么第二个参数 stdout 将会包含命令的标准输出。最后一个参数包含命令相关的错误输出。

列表 8-1 展示了一个复杂些的执行外部命令的例子

LISTING 8-1：执行外部命令（源码：chapter8/01_external_command.js）

```
//导入 child_process 模块的 exec 函数
var exec = require('child_process').exec;
//调用 "cat *.js | wc -l" 命令
```

```
exec('cat *.js | wc –l ', function(err, stdout, stderr ){   //第四行
    //命令退出或者调用失败
    if( err ){
        //启动外部进程失败
        console.log('child_process  退出，错误码是：',err.code);
        return;
    }
}
```

第四行，我们把“cat *.js | wc -l”作为第一个参数传递给 exec，你也可以尝试任何其它命令，只要你在 shell 里使用过的命令都可以。

然后将一个回调函数作为第二个参数，它将会在错误发生或者子进程终结的时候被调用。

还可以在回调函数之前传递第三个可选参数，它包含一些配置选项，比如：

```
var exec = require('child_process').exec;
var options ={
    timeout: 1000,
    killSignal: 'SIGKILL'
};
exec('cat *.js | wc –l ', options, function(err,stdout,stderr){
    //...
});
```

可以使用的参数有：

- cwd —— 当前目录，可以指定当前工作目录。
- encoding —— 子进程输出内容的编码格式，默认值是”utf8”，也就是 UTF-8 编码。如果子进程的输出不是 utf8，你可以用这个参数来设置，支持的编码格式有：
  - ascii
  - utf8
  - ucs2
  - base64

  如果你想了解 Node 支持的这些编码格式的更多信息，请参考第 4 章“使用 Buffer 处理,编码,解码二进制数据”。
- timeout —— 以毫秒为单位的命令执行超时时间，默认是 0，即无限制，一直等到子进程结束。
- maxBuffer —— 指定 stdout 流和 stderr 流允许输出的最大字节数，如果达到最大值，子进程会被杀死。默认值是 200*1024。
- killSignal —— 当超时或者输出缓存达到最大值时发送给子进程的终结信号。默认值是“SIGTERM”，它将给子进程发送一个终结信号。通常都会使用这种有序的方式来结束进程。当用 SIGTERM 信号时，进程接收到以后还可以进行处理或者重写信号处理器的默认行为。如果目标进程需要，你可以同时向他传递其它的信号（比如 SIGUSR1）。你也可以选择发送一个 SIGKILL 信号，它会被操作系统处理并强制立刻结束子进程，这样的话，子进程的任何清理操作都不会被执行。

  如果你想更进一步的控制进程的结束，可以使用 child_process.spawn 命令，后面会介绍。
- evn —— 指定传递给子进程的环境变量，默认是 null，也就是说子进程会继承在它被创建之前的所有父进程的环境变量。

注意：使用 killSignal 选项，你可以以字符串的形式向目标进程发送信号。在 Node 里信号以字符串的形式存在，下面是 UNIX 信号和对应默认操作的列表：

NAME	DEFAULT ACTION	DESCRIPTION
SIGHUP	Terminate process	Terminal line hangup
SIGINT	Terminate process	Interrupt program
SIGQUIT	Create core image	Quit program
SIGILL	Create core image	Illegal instruction
SIGTRAP	Create core image	Trace trap
SIGABRT	Create core image	Abort program
SIGEMT	Create core image	Emulate instruction executed
SIGFPE	Create core image	Floating-point exception
SIGKILL	Terminate process	Kill program
SIGBUS	Create core image	Bus error
SIGSEGV	Create core image	Segmentation violation
SIGSYS	Create core image	Nonexistent system call invoked
SIGPIPE	Terminate process	Software termination signal
SIGALRM	Terminate process	Real-time timer expired
SIGTERM	Terminate process	Software termination signal
SIGURG	Discard signal	Urgent condition present on socket
SIGSTOP	Stop process	Stop (cannot be caught or ignored)
SIGTSTP	Stop process	Stop signal generated from keyboard
SIGCONT	Discard signal	Continue after stop
SIGCHLD	Discard signal	Child status has changed
SIGTTIN	Stop process	Background read attempted from control terminal
SIGTTOU	Stop process	Background write attempted to control terminal
SIGIO	Discard signal	I/O is possible on a descriptor

SIGXCPU	Terminate process	CPU time limit exceeded
SIGXFSZ	Terminate process	File size limit exceeded
SIGVTALRM	Terminate process	Virtual time alarm
SIGPROF	Terminate process	Profiling timer alarm
SIGWINCH	Discard signal	Window size change
SIGINFO	Discard signal	Status request from keyboard
SIGUSR1	Terminate process	User defined signal 1
SIGUSR2	Terminate process	User defined signal 2

你可能想为子进程提供一组可扩展的父级环境变量。如果直接去修改process.env对象，你会改变 Node 进程内所有模块的环境变量，这样会惹很多麻烦。替代方案是，创建一个新对象，复制 process.env 里的所有参数，见例子 8-2：

LISTING 8-2：使用参数化的环境变量来执行命令（源码：chapter8/02_env_vars_augment.js）

```
var env = process.env,
    varName,
    envCopy = {},
    exec = require('child_prcess').exec;
//将 process.env 复制到 envCopy
for( vaName in ev){
    envCopy[varName] = env[varName];
}

//设置一些自定义变量
envCopy['CUSTOM ENV VAR1'] = 'some value';
envCopy['CUSTOM ENV VAR2'] = 'some other value';

//使用 process.env 和自定义变量来执行命令
exec('ls–la',{env: envCopy}, function(err,stdout,stderr){
    if(err){ throw err; }
    console.log('stdout:', stdout);
    console.log('stderr:',stderr);
}
```

上面例子，创建了一个用来保存环境变量的 envCopy 变量，它首先从 process.env 那里复制 Node 进程的环境变量，然后又添加或替换了一些需要修改的环境变量，最后把 envCopy 作为环境变量参数传递给 exec 函数并执行外部命令。

记住，环境变量是通过操作系统在进程之间传递的，所有类型的环境变量值都是以字符串的形式到达子进程的。比如，如果父进程将数字 123 作为一个环境变量，子进程将会以字符串的形式接收“123”。

下面的例子，将在同一个目录里建立 2 个 Node 脚本：parent.js 和 child.js，第一个脚本将会调用第二个，下面我们来创建这两个文件：

LISTING 8-3：父进程设置环境变量(chapter8/03_environment_number_parent.js)

```
var exec = require('child_process').exec;
exec('node child.js', {env: {number: 123}}, function(err, stdout, stderr) {
    if (err) { throw err; }
    console.log('stdout:\n', stdout);
    console.log('stderr:\n', stderr);
});
```

把这段代码保存到 parent.js，下面是子进程的源码，把它们保存到 child.js（见例 8-4）

例 8-4: 子进程解析环境变量(chapter8/04_environment_number_child.js)

```
var number = process.env.number;
console.log(typeof(number)); // → "string"
number = parseInt(number, 10);
console.log(typeof(number)); // → "number"
```

当你把这个文件保存为child.js后，就可以在这个目录下运行下面的命令：

```
$ node parent.js
```

将会看到如下的输出：

```
sdtou:
 string
number

stderr:
```

可以看到，尽管父进程传递了一个数字型的环境变量，但是子进程却以字符串形式接收它（参见输出的第二行），在第三行你把这个字符串解析成了一个数字。

生成子进程

如你所见，可以使用 child_process.exec()函数来启动外部进程，并在进程结束的时候调用你的回调函数，这样用起来很简单，不过也有一些缺点：

- 除了使用命令行参数和环境变量，使用 exec()无法和子进程通信
- 子进程的输出是被缓存的，因此你无法流化它，它可能会耗尽内存

幸运的是，Node 的 child_process 模块允许更细粒度的控制子进程的启动，停止，及其它常规操作。你可以在应用程序里启动一个新的子进程，Node 提供一个双向的通信通道，可以让父进程和子进程相互收发字符串数据。父进程还可以有些针对子进程的管理操作，给子进程发送信号，以及强制关闭子进程。

创建子进程

你可以使用 child_process.spawn 函数来创建一个新的子进程，见例 8-5:

例 8-5: 生成子进程。 (chapter8/05_spawning_child.js)

```
// 导入child_process模块的spawn函数
var spawn = require('child_process').spawn;
// 生成用来执行 "tail -f /var/log/system.log"命令的子进程
var child = spawn('tail', ['-f', '/var/log/system.log']);
```

上面代码生成了一个用来执行tail命令的子进程，并将“-f”和“/bar/log/system.log”作

为参数。tail命令将会监控/var/log/system.og文件（如果存在的话），然后将所有追加的新数据输出到stdout标准输出流。spawn函数返回一个ChildProcess对象，它是一个指针对象，封装了真实进程的访问接口。这个例子里我们把这个新的描述符赋值给一个叫做child的变量。

监听来自子进程的数据

任何包含stdout属性的子进程句柄，都会将子进程的标准输出stdout作为一个流对象，你可以在这个流对象上绑定data事件，这样每当有数据块可用时，就会调用对应的回调函数，见下面的例子：

```
//将子进程的输出打印到控制台
child.stdout.on('data',function(data){
    console.log('tail output: ' + data);
});
```

每当子进程将数据输出到标准输出stdout时，父进程就会得到通知并把数据打印到控制台。

除了标准输出，进程还有另外一个默认输出流：标准错误流，通常用这个流来输出错误信息。

在这个例子里，如果/var/log/system.log文件不存在，tail进程将会输出类似下面的消息："/var/log/system.log：No such file or directory"，通过监听stderr流，父进程会在这种错误发生时得到通知。

父进程可以这样监听标准错误流：

```
child.stderr.on('data', function(data) {
    console.log('tail error output:', data);
});
```

stderr属性和stdout一样，也是只读流，每当子进程往标准错误流里输出数据时，父进程就会得到通知，并输出数据。

发送数据到子进程

除了从子进程的输出流里接收数据，父进程还可以通过 childPoces.stdin 属性往子进程的标准输入里写入数据，以此来往子进程发送数据。

子进程可以通过 process.stdin 只读流来监听标准输入的数据，但是注意你首先必须得恢复（resume）标准输入流，因为它默认处于暂停（paused）状态。

例 8-6 将会创建一个包含如下功能的程序：

- +1 应用：一个简单的应用程序，可以从标准输入接收整型，然后相加，再把相加以后的结果输出到标准输出流。这个应用作为一个简单的计算服务， 把Node进程模拟成一个可以执行特定工作的外部服务。
- 测试+1应用的客户端，发送随机整型，然后输出结果。用来演示Node进程如何生成一个子进程然后让它执行特定的任务。

用下面例8-6的代码创建一个名为plus_one.js的文件：

例 8-6: +1 应用程序(chapter8/06_plus_one.js)

```
// 恢复默认是暂停状态的标准输入流
process.stdin.resume();
```

```
process.stdin.on('data', function(data) {
    var number;
    try {
        // 将输入数据解析为整型
        number = parseInt(data.toString(), 10);
        Spawning Child Processes  /  71
        // +1
        number += 1;
        // 输出结果
        process.stdout.write(number + "\n");
    } catch(err) {
        process.stderr.write(err.message + "\n");
    }
});
```

上面代码里，我们等待来自stdin标准输入流的数据，每当有数据可用，就假设它是个整型并把它解析到一个整型变量里，然后加1，并把结果输出到标准输出流。

可以通过下面命令来运行这个程序：

```
$ node plus_one.js
```

运行后程序就开始等待输入，如果你输入一个整数然后按回车，就会看到一个被加1以后的数字被显示到屏幕上。

可以通过按Ctrl-C来退出程序。

一个测试客户端

现在你要创建一个 Node 进程来使用前面的“+1 应用”提供的计算服务。

首先创建一个名为 plus_one_test.js 的文件，内容见例 8-7:

例 8-7: 测试+1应用(chapter8/07_plus_one_test.js)

```
var spawn = require('child_process').spawn;
// 生成一个子进程来执行+1应用
var child = spawn('node', ['plus_one.js']);
// 每一秒调用一次函数
setInterval(function() {
    // Create a random number smaller than 10.000
    var number = Math.floor(Math.random() * 10000);
    // Send that number to the child process:
    child.stdin.write(number + "\n");
    // Get the response from the child process and print it:
    child.stdout.once('data', function(data) {
        console.log('child replied to ' + number + ' with: ' + data);
    });
}, 1000);
child.stderr.on('data', function(data) {
    process.stdout.write(data);
```

```
});
```

从第一行到第四行启动了一个用来运行“+1应用”的子进程，然后使用setInterval函数每秒钟执行一次下列操作：

- 新建一个小于10000的随机数
- 将这个数字作为字符串传递给子进程
- 等待子进程回复一个字符串

因为你想每次只接收1个数字的计算结果，因此需要使用child.stdout.once而不是child.stdout.on。如果使用了后者，会每隔1秒注册一个data事件的回调函数，每个被注册的回调函数都会在子进程的stdout接收到数据时被执行，这样你会发现同一个计算结果会被输出多次，这种行为显然是错的。

在子进程退出时接收通知

当子进程退出时，exit 事件会被触发。例 8-8 展示了如何监听它：

例 8-8: 监听子进程的退出事件 (chapter8/09_listen_child_exit.js)

```
var spawn = require('child_process').spawn;
// 生成子进程来执行 "ls -la"命令
var child = spawn('ls', ['-la']);
child.stdout.on('data', function(data) {
    console.log('data from child: ' + data);
});
// 当子进程退出：
child.on('exit', function(code) {
    console.log('child process terminated with code ' + code);
});
```

最后几行加黑的代码，父进程使用子进程的 exit 事件来监听它的退出事件，当事件发生时，控制台显示相应的输出。子进程的退出码会被作为第一个参数传递给回调函数。有些程序使用一个非 0 的退出码来代表某种失败状态。比如，如果你尝试执行命令“ls –al filename.txt”，但是当前目录没有这个文件，你就会得到一个值为 1 的退出码，见例 8-9：

例8-9：获得子进程的退出码 (chapter8/10_child_exit_code.js)

```
var spawn = require('child_process').spawn;
// 生成子进程，执行"ls does_not_exist.txt" 命令
var child = spawn('ls', ['does_not_exist.txt']);
// 当子进程退出
child.on('exit', function(code) {
    console.log('child process terminated with code ' + code);
});
```

这个例子里，exit 事件触发了回调函数，并把子进程的退出码作为第一个参数传递给它。如果子进程是被信号杀死而导致的非正常退出，那么相应的信号代码会被当作第二个参数传递给回调函数，如例 8-10：

LISTING 8-10: 获得子进程的退出信号(chapter8/11_child_exit_signal.js)

```
var spawn = require('child_process').spawn;
// 生成子进程，运行"sleep 10"命令
```

```
var child = spawn('sleep', ['10']);
setTimeout(function() {
    child.kill();
}, 1000);
child.on('exit', function(code, signal) {
    if (code) {
        console.log('child process terminated with code ' + code);
    } else if (signal) {
        console.log('child process terminated because of signal ' + signal);
    }
});
```

这个例子里，启动一个子进程来执行 sleep 10 秒的操作，但是还没到 10 秒就发送了一个 SIGKILL 信号给子进程，这将会导致如下的输出：

```
child process terminated because of signal SIGTERM
```

发送信号并杀死进程

在这部分，你将学习如何使用信号来管理子进程。信号是父进程用来跟子进程通信，甚至杀死子进程的一种简单方式。

不同的信号代码代表不同的含义，有很多信号，其中最常见的一些是用来杀死进程的。如果一个进程接收到一个它不知道如何处理的信号，程序就会被异常中断。有些信号会被子进程处理，而有些只能由操作系统处理。

一般情况下，你可以使用 child.kill 方法来向子进程发送一个信号，默认发送 SIGTERM 信号：

```
var spawn = require('child_process').spawn;
var child = spawn('sleep', ['10']);
setTimeout(function() {
    child.kill();
}, 1000);
```

还可以通过传入一个标识信号的字符串作为kill方法的唯一参数，来发送某个特定的信号：

```
child.kill('SIGUSR2');
```

需要注意的是，虽然这个方法的名字叫kill，但是发送的信号并不一定会杀死子进程。如果子进程处理了信号，默认的信号行为就会被覆盖。用Node写的子进程可以像下面这样重写信号处理器的定义：

```
process.on('SIGUSR2', function() {
    console.log('Got a SIGUSR2 signal');
});
```

现在，你定义了SIGUSR2的信号处理器，当你的进程再收到SIGUSR2信号的时候就不会被杀死，而是输出“Got a SIGUSR2 signal”这句话。使用这种机制，你可以设计一种简单的方式来跟子进程沟通甚至命令它。虽然不像使用标准输入功能那么丰富，但是这方式要简单很多。

小结

这一章，学习了使用 child_process.exec 方法来执行外部命令，这种方式可以不使用命令行参数，而是通过定义环境变量的方式把参数传递给子进程。

还学习了通过调用 child_process.spawn 方法生成子进程的方式来调用外部命令，这种方式你可以使用输入流，输出流来跟子进程通信，或者使用信号来跟子进程通信以及杀死进程。