```
Z:\> C:\Users\Mark\LearningNode\Chapter11\hello_world
Hello World!
```

11.1.3 脚本和参数

要想让脚本变得更加有趣，可以向脚本中传递参数。在这种情况下，一般需要关注两个问题：如何向脚本传递参数和如何在脚本中访问参数。

如果一直是通过调用解释器来运行Node脚本（在所有平台下都有效），就无需额外做任何事情——参数会直接传递给正在执行的脚本：

```
node script_name [args]*
```

在UNIX/Mac下传递参数

在类UNIX操作系统中，如果使用#!语法启动脚本，这些参数会直接传递给运行的脚本，因此，完全不需要做任何额外操作。

在Windows下传递参数

在Window平台下，如果使用带.bat扩展名的批处理文件运行Node脚本，可以将参数传递给这些批处理文件，然后相应地，将这些参数通过使用宏%*的方式传递给Node.js脚本。因此，批处理文件会如下所示：

```
@echo off
node %~d0\%~p0\params %*
```

在Node中访问参数

所有传递进来的参数都会被保存到全局对象process的argv数组中。数组的前两个元素一般是当前的node解释器路径和正在运行的脚本路径。因此，任何传递进来的参数都是从第三个元素，即索引下标2开始的。现在运行如下代码：

```
#!/usr/local/bin/node
console.log(process.argv);
```

可以看到如下输出结果：

```
Kimidori:01_running marcw$ ./params.js  1 2 3 4 5
[ '/usr/local/bin/node',
  '/Users/marcw/src/misc/LearningNode/Chapter11/01_running/params.js',
  '1',
  '2',
  '3',
  '4',
  '5' ]
```

11.2 同步处理文件

几乎所有的文件系统模块（fs）的每一个API都同时提供异步和同步两个版本。到目前为止，大部分都采用了异步版本的API。但是，一旦要编写命令行程序，同步版本的API就尤为重要了。基本上，fs中几乎每一个以func命名的API都有一个相应的叫做funcSync的API。

接下来的章节中，将会用些示例来演示和说明如何使用同步API。

11.2.1 基本文件API

fs模块不提供文件拷贝函数，因此我们可以自己写一个。open、read和write这些API都有对应的同步版本：openSync、readSync和writeSync。在同步版本中，一旦API出问题，就会抛出错误。当我们将文件从位置a拷贝到位置b的时候，可以使用一个缓冲对象来保存数据。在文件拷贝完成以后，一定要关闭文件接口。

```
var BUFFER_SIZE = 1000000;

function copy_file_sync (src, dest) {
    var read_so_far, fdsrc, fddest, read;
    var buff = new Buffer(BUFFER_SIZE);

    fdsrc = fs.openSync(src, 'r');
  fddest = fs.openSync(dest, 'w');
  read_so_far = 0;

  do {
      read = fs.readSync(fdsrc, buff, 0, BUFFER_SIZE, read_so_far);
      fs.writeSync(fddest, buff, 0, read);
      read_so_far += read;
  } while (read > 0);

  fs.closeSync(fdsrc);
  return fs.closeSync(fddest);
}
```

在调用该函数的时候，为了确保有足够的参数，可以写一个

file_copy.js脚本来调用该拷贝函数，从而可以处理调用过程中抛出的错误：

```
if (process.argv.length != 4) {
    console.log("Usage: " + path.basename(process.argv[1], '.js')
                + " [src_file] [dest_file]");
} else {
    try {
        copy_file_sync(process.argv[2], process.argv[3]);
    } catch (e) {
        console.log("Error copying file:");
        console.log(e);
        process.exit(-1);
    }

    console.log("1 file copied.");
}
```

可以看到上面的代码中使用到了一个新函数——process.exit。该函数会立即终止Node.js程序并将状态码返回给调用程序（一般会是shell解释器或者命令提示符）。Bourne shell（sh或者bash）的标准是：当返回的状态码为0时表明执行成功；非0的状态码则表明执行失败。当执行该拷贝函数出错时，就会返回-1。

这里，我们还可以稍作修改，将文件拷贝函数改造成文件移动函数。首先，执行文件拷贝操作，然后在确定目标文件已经完全写入并成功关闭以后，删除源文件。可以使用unlinkSync函数进行删除文件操作：

```
function move_file_sync (src, dest) {
    var read_so_far, fdsrc, fddest, read;
    var buff = new Buffer(BUFFER_SIZE);

    fdsrc = fs.openSync(src, 'r');
    fddest = fs.openSync(dest, 'w');
    read_so_far = 0;

    do {
        read = fs.readSync(fdsrc, buff, 0, BUFFER_SIZE, read_so_far);

        fs.writeSync(fddest, buff, 0, read);
        read_so_far += read;
    } while (read > 0);

    fs.closeSync(fdsrc);
    fs.closeSync(fddest);
    return fs.unlinkSync(src);
}
```

其余的脚本保持不变，只需要将函数名由copy_file_sync改为move_file_sync即可。

11.2.2 文件和状态

在Node中，可以使用文件系统模块中的mkdir函数创建文件夹，在下面的脚本中，使用了mkdirSync函数。现在，我们要写一个程序，其功能相当于UNIX shell中的mkdir–p命令：指定一个完整路径，然后创建该目录以及该路径中间所有缺失的目录。

该过程一共包含两步：

1）首先，拆分路径，然后从上至下判断每一级目录是否存在。如果在某一级路径下存在文件，却不是目录时（也就是说，想要使用mkdir a/b/c创建文件夹，但是a/b已经存在，而且只是一个普通文件），则抛出错误。要判断一个文件对象是否存在，则使用existsSync函数；而要确定该文件对象是否是一个目录，则可以调用statsSysnc函数，该函数会返回一个Stats对象，通过它可以判断是否为目录。

2）遍历所有的路径，并为所有缺失目录的路径创建目录。

下面是mkdirs函数的代码：

```
function mkdirs (path_to_create, mode) {
    if (mode == undefined) mode = 0777 & (~process.umask());

    // 1. What do we have already or not?
    var parts = path_to_create.split(path.sep);
    var i;
    for (i = 0; i < parts.length; i++) {
        var search;
        search = parts.slice(0, i + 1).join(path.sep);
        if (fs.existsSync(search)) {
            var st;
            if ((st = fs.statSync(search))) {
                if (!st.isDirectory())
                    throw new Error("Intermediate exists, is not a dir!");
            }
        } else {
            // doesn't exist. We can start creating now
            break;
        }
    }

    // 2. Create whatever we don't have yet.

    for (var j = i; j < parts.length; j++) {
        var build = parts.slice(0, j + 1).join(path.sep);
        fs.mkdirSync(build, mode);
    }
}
```

该函数的第一行是用来设置权限掩码的，可以用来设置刚创建的

目录的读写权限。函数在调用的时候就可以直接设置这些权限，或者如果当前shell用户不想赋予新文件（或者目录）某个权限，可以使用umask过滤掉。在Windows下，如果umask返回0，则表明没有屏蔽任何权限；Windows使用了和UNIX完全不一样的文件权限机制。

11.2.3 目录内容

要想列出某个目录中所有的内容，可以使用readdirSync函数，它会返回指定文件夹下的所有文件名数组，不包括“.”和“..”。

```
#!/usr/local/bin/node
var fs = require('fs');

var files = fs.readdirSync(".");
console.log(files);
```

11.3 用户交互：标准输入和输出

你可能会对所有进程的IO处理三元组很熟悉：stdin、stdout和stderr，分别代表了标准输入、标准输出和错误输出。同样，在Node.js脚本中也提供相同的功能，该功能被挂载到process对象中。在Node中，它们实际上都是Stream对象的实例（请参见第6章）。

实际上，Node中的console.log函数等价于

```
process.stdout.write(text + "\n");
```

而console.error则与

```
process.stderr.write(text + "\n");
```

等价。

但是，输入提供了很多选项，在下面的章节中，将会介绍到缓冲输入（逐行输入）和无缓冲输入（一旦输入一个字符，就会立刻输出出来）的对比。

11.3.1 基本缓冲输入和输出

默认情况下，每次只能从stdin数据流中读取和缓冲一行数据。因此，想要从stdin中读取数据，只有当用户按下回车（Enter）键，程序才会从输入流中读取整行数据。可以通过向stdin添加readable事件监听器实现，如下所示：

```
process.stdin.on('readable', function () {
    var data = process.stdin.read();

    // do something w input
});
```

但是默认情况下，stdin输入流处于暂停状态。因此，需要调用resume函数才能开始从输入流中接收数据：

```
process.stdin.resume();
```

下面写一个小程序，读取输入行，然后使用md5加密输入数据并将其打印出来，不断重复循环，直到按下Ctrl+C或者空行才退出：

```
process.stdout.write("Hash-o-tron 3000\n");
process.stdout.write("(Ctrl+C or Empty line quits)\n");
process.stdout.write("data to hash > ");

process.stdin.on('readable', function () {
    var data = process.stdin.read();
    if (data == null) return;
    if (data == "\n") process.exit(0);

    var hash = require('crypto').createHash('md5');
    hash.update(data);
    process.stdout.write("Hashed to: " + hash.digest('hex') + "\n");
    process.stdout.write("data to hash > ");
});

process.stdin.setEncoding('utf8');
process.stdin.resume();
```

上述代码在用户按下回车键的一瞬间就开始工作了：首先，检查输入是否为空，如果不为空，就用md5加密并打印出来；然后出现新的提示符，等待用户输入其他内容。由于stdin处于非暂停状态，因此Node程序不会退出（如果尝试暂停接收stdin输入流，并且没有其他任务需要执行，则程序会立即退出）。

11.3.2 无缓冲输入

在某些情况下，如果想让用户按下键盘以后程序立即响应，可以通过使用setRawMode函数来开启stdin输入流中的原始模式（raw mode），参数可以使用布尔值来设置是否开启（true为开启）原始模式。

同时，更新前一小节中的代码，让用户可以任意选择喜欢的哈希类型。在用户输入一行文本并按下回车键以后，程序会让用户按下数字键1到4以选择不同的哈希算法。该程序的完整代码如代码清单11.1所示。

代码清单11.1 使用标准输入中的RawMode（raw_mode.js）

```
process.stdout.write("Hash-o-tron 3000\n");
process.stdout.write("(Ctrl+C or Empty line quits)\n");
process.stdout.write("data to hash > ");

process.stdin.on('readable', function (data) {
    var data = process.stdin.read();
    if (!process.stdin.isRaw) {                          // 1.
        if (data == "\n") process.exit(0);
        process.stdout.write("Please select type of hash:\n");
        process.stdout.write("(1 - md5, 2 - sha1, 3 - sha256, 4 - sha512) \n");
        process.stdout.write("[1-4] > ");
        process.stdin.setRawMode(true);
    } else {
        var alg;
        if (data != '^C') {                               // 2.
            var c = parseInt(data);
            switch (c) {
                case 1: alg = 'md5'; break;
                case 2: alg = 'sha1'; break;
                case 3: alg = 'sha256'; break;
                case 4: alg = 'sha512'; break;
            }
            if (alg) {                                    // 3.
                var hash = require('crypto').createHash(alg);
                hash.update(data);
                process.stdout.write("\nHashed to: " + hash.digest('hex'));
                process.stdout.write("\ndata to hash > ");
                process.stdin.setRawMode(false);
            } else {
                process.stdout.write("\nPlease select type of hash:\n");
                process.stdout.write("[1-4] > ");
            }
        } else {
            process.stdout.write("\ndata to hash > ");
            process.stdin.setRawMode(false);
        }
    }
});

process.stdin.setEncoding('utf8')
process.stdin.resume();
```

由于该脚本从stdin中接收数据，既支持缓冲输入（当请求输入需要哈希的文本时），也支持无缓冲输入（当请求选择哈希算法时），因此，它稍微有点复杂。让我们看下它究竟是如何工作的。

1）首先，会检查接收的输入是缓存输入还是无缓存输入。如果是前者，则是一行需要哈希的输入文本，然后会打印出选择使用算法的请求。由于脚本希望用户只能按下一个1到4的数字键，因此将stdin切换到RawMode模式，现在，一旦输入任何一个键都会触发readable事件。

2）如果输入流是RawMode模式，这就意味着用户按下一个按键就会响应哈希算法的请求。这部分的第一行会检测输入的键值是否为Ctrl+C（注意，可以进入一个文本编辑器；如在Emacs中，可以使用Ctrl+Q，然后按下Ctrl+C，它会输出^C字符。每个编辑器都会稍有不同）。如果用户按下了Ctrl+C，脚本会中断请求，并返回到哈希提示符。

如果用户输入了其他键值，脚本会判断是否为有效键值（1到4），如果不是，脚本会提示让用户再重新输入一遍。

3）最后，根据选择的算法，脚本生成相应的哈希值并打印出来，然后返回到原先的请求输入数据的提示符。在这之前，一定要记得关闭RawMode模式，这样才能返回到正常的缓冲输入模式。

同时，由于在程序中调用了stdin的resume函数，所以只有在调用process.exit、用户在输入空行或者在缓冲输入模式下输入Ctrl+C（这会导致Node终止程序）时，程序才会正常退出。

11.3.3 Readline模块

另一种使用Node.js中输入流的方式就是使用readline模块。由于它仍然被标记为不稳定状态，极有可能更改API，因此，我不会花费大量的时间和精力在它上面，但是，它还是有一些精巧的特性值得我们在程序中使用。

要想使用readline模块，需要调用它的createInterface方法，指定参数选项中的输入流和输出流：

```
var readline = require('readline');

var rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
```

完成这些以后，程序只有在调用rl.close之后才会正常退出。

逐行提示

如果调用readline的prompt方法，该程序就会等待一行的输入（直到回车）。当程序捕捉到回车键按下，就会触发readline对象中的line事件，这样该事件就可以处理输入数据了：

```
rl.on("line", function (line) {
    console.log(line);
    rl.prompt();
});
```

如果需要继续监听事件，则需要再次调用prompt方法。readline接口最神奇的地方就是一旦用户按下Ctrl+C，SIGINT事件就会被调用，那么就可以选择关掉或者恢复状态，继续监听。这里，通过关闭readline接口，让程序停止监听输入流并退出。

```
rl.on("SIGINT", function ()
    rl.close();
});
```

现在，可以尝试使用readline模块来编写一个简易的逆波兰式计算器。计算器代码如清单11.2所示。

如果你从来没有听说过逆波兰式表示法或者忘记它是如何运行的，没有关系，它只是一种后缀数学符号格式。当计算1+2时，该表示法会写成1 2+；当计算5*（2+3）时，它会写成5 2 3+*，等等。这个简易计算器一次只接收一个字符串，并使用空格将字符分开，并做简单的计算。

代码清单11.2　基于readline的简易后缀计算器（readline.js）

```
var readline = require('readline');
var rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

var p = "postfix expression > "
rl.setPrompt(p, p.length);
rl.prompt();                                          // 1.

rl.on("line", function (line) {                       // 2.
    if (line == "\n")  {
        rl.close(); return;
    }

    var parts = line.split(new RegExp("[ ]+"));
    var r = postfix_process(parts);
    if (r !== false)
        process.stdout.write("Result: " + r + "\n");
    else
        process.stdout.write("Invalid expression.\n");
    rl.prompt();                                      // 3.
});

rl.on("SIGINT", function () {                         // 4.
    rl.close();
});
// push numbers onto a stack, pop when we see an operator.
function postfix_process(parts) {
    var stack = [];
    for (var i = 0; i < parts.length; i++) {
        switch (parts[i]) {
          case '+': case '-': case '*': case '/':
            if (stack.length < 2) return false;
            do_op(stack, parts[i]);
            break;
          default:
            var num = parseFloat(parts[i]);
            if (isNaN(num)) return false;
            stack.push(num);

            break;
        }
    }
    if (stack.length != 1) return false;
    return stack.pop();
}

function do_op(stack, operator) {
    var b = stack.pop();
    var a = stack.pop();
    switch (operator) {
      case '+': stack.push(a + b); break;
      case '-': stack.push(a - b); break;
      case '*': stack.push(a * b); break;
      case '/': stack.push(a / b); break;
      default:  throw new Error("Unexpected operator");
    }
}
```

该程序工作流程如下：

1）首先，创建readline模块对象，设置默认的提示符文本，然后输出该提示符并等待输入。

2）当接收到一行输入的时候，检测它是否为空（如果为空，则关闭readline接口，退出整个程序），否则解析输入的字符串，并将其作为参数传递给计算函数。计算完成以后，打印计算结果（成功或失败）。

3）完成本次计算，告知readline打印提示符并继续等待下次输

入。

4）如果用户按下Ctrl+C，则程序会关闭readline接口实例，继而程序正常退出。

至此，可以完整测试该程序了：

```
Kimidori:03_stdinout marcw$ node readline_rpn.js
postfix expression > 1 2 +
Result: 3
postfix expression > 2 3 4 + *
Result: 14
postfix expression > cat
Invalid expression.
postfix expression > 1 2 4 cat dog  3 4 + - / *
Invalid expression.
postfix expression > 2 3 + 5 *
Result: 25
postfix expression >
```

问题

readline模块另一个重要功能就是可以提问，并直接在回调函数中接收答案。基本格式如下：

```
rl.question("hello? ", function (answer) {
    // do something
});
```

接下来，我们要写一个调查问卷程序：它包含一组问题（可以将问题放到文件中，这样问题是可配置的）。每次向用户问一个问题，然后使用fs模块中的appendFileSync函数将用户的答案写进answers.txt中。

由于question函数是是异步的，所以必须使用async.forEachSeries来迭代调查的每一个问题。调查程序如代码清单11.3所示。

代码清单11.3　调查问卷程序（question.js）

```
var readline = require('readline'),
    async = require("async"),
    fs = require('fs');

var questions = [ "What's your favorite color? ",
                  "What's your shoe size? ",
                  "Cats or dogs? ",
                  "Doctor Who or Doctor House? " ];

var rl = readline.createInterface({                    // 1.
    input: process.stdin,
    output: process.stdout
});
var output = [];
async.forEachSeries(
    questions,
    function (item, cb) {                              // 2.
        rl.question(item, function (answer) {
            output.push(answer);
            cb(null);
        });
    },
    function (err) {                                   // 3.
        if (err) {
            console.log("Hunh, couldn't get answers");
            console.log(err);
            return;
        }
        fs.appendFileSync("answers.txt", JSON.stringify(output) + "\n");
        console.log("\nThanks for your answers!");
        console.log("We'll sell them to some telemarketer immediately!");
        rl.close();
    }
);
```

该程序主要包含如下流程：

1）首先，初始化readline模块，设置stdin和stdout数据流。

2）然后，对于数组中的每一个问题，调用readline上的question函数（由于question是异步函数，因此需要使用async.forEachSeries）并将结果添加到输出数组中。

3）最后，所有问题回答结束以后，async会调用回调函数，当出错时，会打印出错误信息；或者将用户的答案附加到answers.txt文件中，然后关闭readline对象，退出程序。

11.4 进程处理

Node中还可以使用命令行（甚至在Web应用中）启动其他程序。使用child_process模块，有两种不同复杂程度的选项，我们将从简单的exec开始。

11.4.1 简单进程创建

Child_process模块中的exec函数会接收一个命令并在系统shell中执行（UNIX/Mac平台下的sh/bash，或者Windows下的cmd.exe）。因此，可以指定一个简单的命令程序如"date"来运行，或者稍微复杂点的命令，比如"echo'Mancy'|sed s/M/N/g"。所有的命令运行以后，将输出缓存，并会在命令执行完成以后返回给调用者。

基本格式如下：

```
exec(command, function (error, stdout, stderr) {
    // error is if an error occurred
    // stdout and stderr are buffers
});
```

当命令执行完成以后，回调函数会被调用。当出错时，第一个参数为非空。否则，所有的内容将会被写到对应输出流stdout和stderr的Buffer对象中。

现在，我们尝试写一个程序使用exec函数来运行cat程序（即Windows中的type程序），它需要指定一个文件名称。该程序使用exec函数启动cat/type程序，并在执行完成以后打印所有输出信息：

```
var exec = require('child_process').exec,
    child;

if (process.argv.length != 3) {
    console.log("I need a file name");
    process.exit(-1);
}

var file_name = process.argv[2];
var cmd = process.platform == 'win32' ? 'type' : "cat";
child = exec(cmd + " " + file_name, function (error, stdout, stderr) {
    console.log('stdout: ' + stdout);
    console.log('stderr: ' + stderr);

    if (error) {
        console.log("Error exec'ing the file");
        console.log(error);

          process.exit(1);
      }
  });
```

11.4.2 使用Spawn创建进程

另一种高级的创建进程的方式是使用child_process模块中的spawn函数。该函数拥有其创建出来的子进程的stdin和stdout的完整控制权，这样可以使用一些奇妙的功能，如将输出从一个子进程通过管道传入另一个子进程中。

下面的程序需要传入一个JavaScript文件的名称，并使用node程序运行该脚本：

```
var spawn = require("child_process").spawn;
var node;

if (process.argv.length != 3) {
    console.log("I need a script to run");
    process.exit(-1);
}

var node = spawn("node", [ process.argv[2] ]);
node.stdout.on('readable', print_stdout);
node.stderr.on('readable', print_stderr);
node.on('exit', exited);

function print_stdout() {
    var data = process.stdout.read();
    console.log("stdout: " + data.toString('utf8'));
}
function print_stderr(data) {
    var data = process.stderr.read();
    console.log("stderr: " + data.toString('utf8'));
}
function exited(code) {
    console.error("--> Node exited with code: " + code);
}
```

当调用spawn时，第一个参数是需要执行的命令的名称，第二个参数是传进来的参数数组。可以看到子进程中任何输出流写入到stdout和stderr时，都会立即触发对应数据流上的事件，并可以实时看到发生的一切。

现在，我们可以尝试写一些更高级的功能。在前一章中，可以看到利用shell脚本或者命令行提示符实现下述功能非常高效：

```
while 1
    node script_name
end
```

而要使用JavaScript实现相同的功能，可以使用spawn函数。当然，这比shell脚本要复杂一些。但是这是有益的，它可以让我们做一些额外的工作，以便得到任何想要的监控数据。新的启动器如代码清单11.4所示。

代码清单11.4　全Node执行程序（node_runner.js）

```
var spawn = require("child_process").spawn;
var node;

if (process.argv.length < 3) {
    console.log("I need a script to run");
    process.exit(-1);
}

function spawn_node() {
    var node = spawn("node", process.argv.slice(2));
    node.stdout.on('readable', print_stdout);
    node.stderr.on('readable', print_stderr);
    node.on('exit', exited);
}

function print_stdout() {
    var data = process.stdout.read();
    console.log("stdout: " + data.toString('utf8'));
}
function print_stderr(data) {
    var data = process.stderr.read();
    console.log("stderr: " + data.toString('utf8'));
}
function exited(code) {
    console.error("--> Node exited with code: " + code + ". Restarting");
    spawn_node();
}

spawn_node();
```

该程序监听新创建的子进程中的exit事件，当该事件被触发时，程序会使用node解释器重启想要运行的脚本。

现在给你留一份练习作业：更新上述脚本，让其功能与前一章中的node_ninja_runner一样。使用exec函数获得ps aux命令执行以后的所有输出内容；该输出结果可以使用JavaScript解析。最后，如果检测到输出内容过大，可以使用kill方法来结束子进程。

11.5 小结

现在，我们见识到Node不仅擅长编写网络应用，而且在同步的命令行应用中也有独到之处。在经过简单的创建、运行脚本并传递参数给脚本这一系列的旅程后，你应该对Node程序操作输入流和输出流得心应手，甚至在必要的时候能在缓冲输入流和无缓冲输入流中来回切换。最后，我们学习了如何使用exec和spawn创建和运行Node.js脚本程序。

在结束这一章的Node.js编程学习之后，我们将会把注意力集中到最重要的脚本和应用测试上面。

第12章　测试

程序写到现在，我们准备看一下如何测试以保证程序的正常运行。现在有很多成熟的测试模型和范式，而Node.js支持其中的绝大部分。在本章中，我们会集中了解其中最常用的一些测试模型，然后学习如何进行功能测试——不仅仅只测试同步API，还要测试Node的异步代码。最后，为相册应用添加相应的测试代码。

12.1 测试框架选择

现今，有很多流行的测试模型，包括测试驱动开发（test-driven development，TDD）、行为驱动开发（behavior-driven development，BDD）等。前者是用来确保所有的代码都拥有合适的测试接口（实际上，很多情况下，必须先有测试用例，后有开发代码）；而后者则是专注于某个单元或者代码模块的业务需求，要求测试得更全面，而不仅仅是简单的单元测试。

不考虑使用哪一种测试模型（或者打算使用，如果这是你第一次写测试的话），将测试代码添加到代码库中是个不错的主意，因为不仅仅要确保现在的代码库没有问题，而且要保证将来代码库修改以后不会带来其他问题。在Node应用中添加测试是需要一些挑战的，因为经常需要将同步、异步和RESTful服务API功能混合到一起。但是，不用担心，因为Node.js平台足够健壮和优秀，它已经准备了一些不错的选择来满足我们所有的需求。

在拥有琳琅满目的测试框架的今天，有三款出色的测试框架广受欢迎，脱颖而出：

- nodeunit——这是一款简单易用的测试框架，同时支持Node和浏览器端测试。它极易使用，并且在定制测试框架方面极为灵活。

- Mocha——这款测试框架基于一个名叫Expresso的旧测试框架，Mocha是一款功能齐全的TDD Node测试框架，专注于易用性和愉悦编程的理念。它拥有一些非常棒的异步测试的功能，并提供了格式化输出的API。

- VowsJS——它是Node.js平台下最为卓越的BDD测试框架，VowsJS不仅仅包含非常描述性的语法和结构，能将测试和BDD理念完美结合；更能让测试用例并行，从而提高执行效率。

尽管它们都非常完美，并且有各自的定位，但是在本章中，我们

将专注于简单的nodeunit，很大一部分原因是，它使用起来非常简单且容易演示。

安装Nodeunit

现在，可以在项目的根目录中创建test/子文件夹，并将项目中与测试相关的文件、示例、数据文件等都放进去。第一个放进该文件夹的是package.json文件，如下所示：

```
{
  "name": "API-testing-demo",
  "description": "Demonstrates API Testing with nodeunit",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "nodeunit": "0.7.x"
  }
}
```

当运行npm update命令时，nodeunit模块就会自动安装，然后就可以开始编写和运行测试用例了。

12.2 编写测试用例

Nodeunit将测试集成到模块内部，即把每一个暴露的函数都当成一个测试，而每一个暴露的对象都会被当成一个测试组。对于每一个测试，都会赋予一个对象参数，它会帮助执行测试用例，并在完成测试时通知nodeunit：

```
exports.test1 = function (test) {
    test.equals(true, true);
    test.done();
}
```

在每一个测试的最后都需要调用test.done；否则，nodeunit就无法知道测试是否已经完成。要想运行该测试用例，可以将其保存到名叫trivial.js的文件中，并运行node_modules/.bin文件夹下的脚本。在Unix/Mac平台或者Windows平台下，可以运行如下命令（当然，在Windows平台下，可以将“/”字符替换成“\”字符）：

```
node_modules/.bin/nodeunit trivial.js
```

执行结束以后，可以看到如下结果：

```
C:\Users\Mark\a> node_modules\.bin\nodeunit.cmd trivial.js

trivial.js
✓ test1

OK: 1 assertions (0ms)

C:\Users\Mark\a>
```

12.2.1 简单功能测试

每写一个测试，都需要做三件事：

1）调用参数test中的expect方法，以确认nodeunit在该测试下期望验证的“条件”次数。这一步是是可选的，但如果偶尔在一些测试中需要跳过一些测试代码，那么这会是一个不错的选择。

2）对于每一个需要验证的条件，需要用到以下一些断言函数（见表12.1）来验证期望的结果。前面第一个示例中的test.equals就是其中之一。

3）在每一个测试的最后调用test.done来通知nodeunit结束测试。

现在，将前一章中编写的逆波兰式计算器代码放进一个叫做rpn.js的文件中（见代码清单12.1）。

代码清单12.1　rpn.js文件

```
// push numbers onto a stack, pop when we see an operator.
exports.version = "1.0.0";

exports.compute = function (parts) {
    var stack = [];
    for (var i = 0; i < parts.length; i++) {
        switch (parts[i]) {
          case '+': case '-': case '*': case '/':
            if (stack.length < 2) return false;
            do_op(stack, parts[i]);
            break;
        default:
            var num = parseFloat(parts[i]);
            if (isNaN(num)) return false;
            stack.push(num);
            break;
        }
    }
    if (stack.length != 1) return false;
    return stack.pop();
}

function do_op(stack, operator) {
    var b = stack.pop();
    var a = stack.pop();
    switch (operator) {
      case '+': stack.push(a + b); break;
      case '-': stack.push(a - b); break;
      case '*': stack.push(a * b); break;
      case '/': stack.push(a / b); break;

      default:  throw new Error("Unexpected operator");
    }
}
```

现在，为它编写测试代码（千万不要忘记在测试文件中使用rpn.js文件中的require）：

```
exports.addition = function (test) {
    test.expect(4);
    test.equals(rpn.compute(prep("1 2 +")), 3);
    test.equals(rpn.compute(prep("1 2 3 + +")), 6);
    test.equals(rpn.compute(prep("1 2 + 5 6 + +")), 14);
    test.equals(rpn.compute(prep("1 2 3 4 5 6 7 + + + + + +")), 28);
    test.done();
};
```

prep函数用来将提供的字符串分割成数组：

```
function prep(str) {
    return str.trim().split(/[ ]+/);
}
```

我们可以在测试中添加任何计算器中支持的运算符（减法、乘法和除法），甚至还可以添加小数的测试。

```
exports.decimals = function (test) {
    test.expect(2);
    test.equals(rpn.compute(prep("3.14159 5 *")), 15.70795);
    test.equals(rpn.compute(prep("100 3 /")), 33.333333333333336);
    test.done();
}
```

到目前为止，只使用了test.equals断言来验证期望的值。然而，nodeunit还使用了一个叫做assert的模块，它提供了其他一些方法，如表12.1所示。

表 12.1　测试断言

方　法	测试说明
ok（value）	测试 value 值是否为真（true）
equal（value，expected）	测试 value 值是否为 expected 值（仅使用 == 进行值比较，而非 ===）
notEqual（value，expected）	确保 value 值不为 expected 值（仅使用 == 进行值比较）
deepEqual（value，expected）	确保 value 值与 expected 值相等，如果需要，会比较子值，使用 == 进行比较
notDeepEqual（value，expected）	确保 value 值不为 expected 值，如果需要，会比较子值，使用 == 进行比较
strictEqual（value，expected）	测试值是否相等，使用 === 操作符
throws（code，[Error]）	确保代码段抛出错误，并可选择是否为指定错误类型
doesNotThrow（code，[Error]）	确保代码段不抛出错误（可选，指定错误类型）

现在，可以添加一个新测试，确保计算器不接受空表达式：

```
exports.empty = function (test) {
    test.expect(1);
    test.throws(rpn.compute([]));
    test.done();
};
```

测试失败以后，nodeunit会报错，并将失败条件和引起错误的完整调用栈信息打印出来：

```
01_functional.js
✖ addition

AssertionError: 28 == 27
    at Object.assertWrapper [as equals]
       [...]tests/node_modules/nodeunit/lib/types.js:83:39)
    at Object.exports.addition [...]tests/01_functional.js:9:10)
   (etc)
✓ subtraction
✓ multiplication
✓ division
✓ decimals
✓ empty

FAILURES: 1/17 assertions failed (5ms)
```

通过打印信息，可以查看代码中哪个测试导致了错误，分析错误原因并修复。

12.2.2 异步功能测试

在Node.js中会用到很多异步编程，因此很多测试也需要是异步的。Nodeunit将这种理念融于设计之中：无论测试执行多长时间或者异步与否，都没有问题，只要在执行结束时调用test.done即可。例如，现在编写两个异步测试：

```
exports.async1 = function (test) {
    setTimeout(function () {
        test.equal(true, true);
        test.done();
    }, 2000);
};

exports.async2 = function (test) {
    setTimeout(function () {
        test.equal(true, true);
        test.done();
    }, 1400);
};
```

执行上面的测试模块会得到如下测试结果（注意：运行时一般会将这两个测试合并起来顺序执行）：

```
Kimidori: functional_tests marcw$ node_modules/.bin/nodeunit 02_async.js

02_async.js
✓ async1
✓ async2

OK: 2 assertions (3406ms)
```

12.3 RESTful API测试

现在可以给相册应用添加两种不同的测试：第一种是功能测试，保证前面编写的各类模块能正常运行；另一种是确保服务器提供一致的功能性REST API。

只要添加一个名叫request的npm模块，就可以使用nodeunit实现第二种测试，request允许使用不同的HTTP方法调用远程URL，并将结果返回给Buffer对象。

例如，要想调用服务器上的/v1/albums.json，可以调用：

```
request.get("http://localhost:8080/v1/albums.json", function (err, resp, body) {
    var r = JSON.parse(body);
    // do something
});
```

上述回调函数的参数包括：

- 错误对象，当调用过程中发生错误时，会将错误信息返回给该对象。
- HttpResponse对象，从中可以获得返回的头信息和状态码。
- Buffer对象，用来保存服务器返回的数据（如果有数据的话）

还记得在第9章中编写的MySQL版相册应用么？把它拷贝到一个新的地方，将数据库清理干净，并重新运行数据库初始脚本schema.sql：

```
mysql -u root < schema.sql
```

然后在根目录下创建一个新的叫做test/的目录：

```
+ photo_album/
  + static/
  + app/
  + test/
```

在test文件夹下，创建一个包含nodeunit和request的package.json文件：

```
{
  "name": "API-testing-demo",
  "description": "Demonstrates API Testing with request and nodeunit",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "nodeunit": "0.7.x",
    "request": "2.x"
  }
}
```

现在可以编写第一个RESTful API的测试了：

```
var request = require('request');

var h = "http://localhost:8080";
exports.no_albums = function (test) {
    test.expect(5);
    request.get(h + "/v1/albums.json", function (err, resp, body) {

        test.equal(err, null);
        test.equal(resp.statusCode, 200);
        var r = JSON.parse(body);
        test.equal(r.error, null);
        test.notEqual(r.data.albums, undefined);
        test.equal(r.data.albums.length, 0);
        test.done();
    });
};
```

一开始，数据库中并没有相册信息，因此在获取所有相册信息的时候会返回一个空数组。

要测试相册的创建功能，需要使用PUT方法，将数据发送到服务器中，测试返回的结果。Request模块可以指定JSON数据格式来传递数据，返回的结果中会自动指定Content-Type:application/json响应头，返回结果为JSON格式数据：

```
exports.create_album = function (test) {
    var d = "We went to HK to do some shopping and spend New Year's. Nice!";
    var t = "New Year's in Hong Kong";
    test.expect(7);
    request.put(
        { url: h + "/v1/albums.json",
          json: { name: "hongkong2012",
                  title: t,
                  description: d,
                  date: "2012-12-28" } },
        function (err, resp, body) {
            test.equal(err, null);
            test.equal(resp.statusCode, 200);
            test.notEqual(body.data.album, undefined);
            test.equal(body.data.album.name, "hongkong2012"),
            test.equal(body.data.album.date, "2012-12-28");
            test.equal(body.data.album.description, d);
            test.equal(body.data.album.title, t);
            test.done();
        }
    );
};
```

当请求结束以后，可以测试所有的内容，包括HTTP响应的状态码（在成功请求以后，一般会返回200）；可以分解JSON对象，确认数据是否与期望值一致。在一开始，你会注意到上述代码指定必须检测七次条件，这会帮助nodeunit更好地执行测试。最后，不仅需要确认API是否按预期一样正常工作，而且需要确保出错是否在预期之内。因此当一个函数期望返回错误时，一定要进行检测！事实上，当我在写这本书的时候，下面的测试代码帮我检测出handlers/albums.js中的一些参数验证的错误：

```
exports.fail_create_album = function (test) {
    test.expect(4);
    request.put(
        { url: h + "/v1/albums.json",

          headers: { "Content-Type" : "application/json" },
          json: { name: "Hong Kong 2012",                    // no spaces allowed!
                  title: "title",
                  description: "desc",
                date: "2012-12-28" } },
        function (err, resp, body) {
            test.equal(err, null);
            test.equal(resp.statusCode, 403);
            test.notEqual(body.error, null);
            test.equal(body.error, "invalid_album_name");
            test.done();
        }
    );
};
```

因此，不仅要检测HTTP返回的状态码是否为403，而且要检测错误的内容是否与预期一致。

测试受保护的资源

当使用用户名和密码保护服务器资源时（可以使用HTTPS协

议，这样就无法查看用户名和密码了），如果需要测试API，利用URL可以内置HTTP基本身份验证的原理，在request的URL中包含用户名和密码，如下所示：

```
https://localhost:username@password:8080/v1/albums.json
```

因此，要测试站点中的安全部分，需要写如下所示的nodeunit测试：

```
var h = "localhost:username@secret:8080";
exports.get_user_info = function (test) {
    test.expect(5);
    request.get(h + "/v1/users/marcwan.json", function (err, resp, body) {
        test.equal(err, null);
        test.notEqual(resp.statusCode, 401);
        test.equal(resp.statusCode, 200);
        var r = JSON.parse(body);
        test.equal(r.data.user.username, "marcwan");
        test.equal(r.data.user.password, undefined);
        test.done();
    });
};
```

12.4 小结

通过npm可以找到很多可用的测试框架来测试Node.js应用和脚本，既简单又快速。在本章中，我不仅展示了如何使用目前最流行的TDD框架nodeunit测试应用中的同步和异步代码，还演示了如何与request模块结合，完整地测试JSON服务器提供的API。

学会了这些知识，我们可以结束这次奇妙的Node.js之旅了。我希望已经原汁原味地将Node.js非凡而有趣之处传递给你，并让你在读书的过程中，体验到Node平台的魅力。

请坐下来，开始编写代码。如果你有什么好的关于网站的点子，就开始动手实现它吧！如果你是移动端开发者，请想想如何将页面展现给移动用户，并且如何使用Node实现。即使你只是想学习一下服务器端脚本，也还是动手亲身体验下吧。因为想要拥有更好的编程技术的唯一途径就是使用这些技术。

如果编程过程中遇到了困难和问题，别忘记Node.js社区可是非常活跃和乐于助人的！在如此多的热心积极的开发者中，你一定能找到志同道合的朋友，并得到所需的资源和帮助，来编写有趣实用的应用。