Node.js 概述

来自《JavaScript 标准参考教程（alpha）》 (/)，by 阮一峰

目录

1. 简介
   1.1 安装与更新
   1.2 版本管理工具nvm
   1.3 基本用法
   1.4 REPL环境
   1.5 异步操作
   1.6 全局对象和全局变量
2. 模块化结构
   2.1 概述
   2.2 核心模块
   2.3 自定义模块
3. 异常处理

3.1 try…catch结构
3.2 回调函数
3.3 EventEmitter接口的error事件
3.4 uncaughtException事件
3.5 unhandledRejection事件
4. 命令行脚本
5. 参考链接

1. 简介

Node是JavaScript语言的服务器运行环境。

所谓“运行环境”有两层意思：首先，JavaScript语言通过Node在服务器运行，在这个意义上，Node有点像JavaScript虚拟机；其次，Node提供大量工具库，使得JavaScript语言与操作系统互动（比如读写文件、新建子进程），在这个意义上，Node又是JavaScript的工具库。

Node内部采用Google公司的V8引擎，作为JavaScript语言解释器；通过自行开发的libuv库，调用操作系统资源。

1.1 安装与更新

访问官方网站nodejs.org或者github.com/nodesource/distributions，查看Node的最新版本和安装方法。

官方网站提供编译好的二进制包，可以把它们解压到 /usr/local 目录下面。

```
$ tar -xf node-someversion.tgz
```

然后，建立符号链接，把它们加到$PATH变量里面的路径。

```
$ ln -s /usr/local/node/bin/node /usr/local/bin/node
$ ln -s /usr/local/node/bin/npm /usr/local/bin/npm
```

下面是Ubuntu和Debian下面安装Deb软件包的安装方法。

```
$ curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
$ sudo apt-get install -y nodejs

$ apt-get install nodejs
```

安装完成以后，运行下面的命令，查看是否能正常运行。

```
$ node --version
# 或者
$ node -v
```

更新node.js版本，可以通过node.js的 n 模块完成。

```
$ sudo npm install n -g
$ sudo n stable
```

上面代码通过 n 模块，将node.js更新为最新发布的稳定版。

n 模块也可以指定安装特定版本的node。

```
$ sudo n 0.10.21
```

1.2 版本管理工具nvm

如果想在同一台机器，同时安装多个版本的node.js，就需要用到版本管理工具nvm。

```
$ git clone https://github.com/creationix/nvm.git ~/.nvm
$ source ~/.nvm/nvm.sh
```

安装以后，nvm的执行脚本，每次使用前都要激活，建议将其加入~/.bashrc文件（假定使用Bash）。激活后，就可以安装指定版本的Node。

```
# 安装最新版本
$ nvm install node

# 安装指定版本
$ nvm install 0.12.1

# 使用已安装的最新版本
$ nvm use node

# 使用指定版本的node
$ nvm use 0.12
```

nvm也允许进入指定版本的REPL环境。

```
$ nvm run 0.12
```

如果在项目根目录下新建一个.nvmrc文件，将版本号写入其中，就只输入 `nvm use` 命令即可，不再需要附加版本号。

下面是其他经常用到的命令。

```
# 查看本地安装的所有版本
$ nvm ls

# 查看服务器上所有可供安装的版本。
$ nvm ls-remote

# 退出已经激活的nvm，使用deactivate命令。
$ nvm deactivate
```

1.3 基本用法

安装完成后，运行node.js程序，就是使用node命令读取JavaScript脚本。

当前目录的 demo.js 脚本文件，可以这样执行。

```
$ node demo
# 或者
$ node demo.js
```

使用 -e 参数，可以执行代码字符串。

```
$ node -e 'console.log("Hello World")'
Hello World
```

1.4 REPL环境

在命令行键入node命令，后面没有文件名，就进入一个Node.js的REPL环境（Read–eval–print loop，”读取-求值-输出”循环），可以直接运行各种JavaScript命令。

```
$ node
> 1+1
2
>
```

如果使用参数 –use_strict，则REPL将在严格模式下运行。

```
$ node --use_strict
```

REPL是Node.js与用户互动的shell，各种基本的shell功能都可以在里面使用，比如使用上下方向键遍历曾经使用过的命令。

特殊变量下划线（_）表示上一个命令的返回结果。

```
> 1 + 1
2
> _ + 1
3
```

在REPL中，如果运行一个表达式，会直接在命令行返回结果。如果运行一条语句，就不会有任何输出，因为语句没有返回值。

```
> x = 1
1
> var x = 1
```

上面代码的第二条命令，没有显示任何结果。因为这是一条语句，不是表达式，所以没有返回值。

1.5 异步操作

Node采用V8引擎处理JavaScript脚本，最大特点就是单线程运行，一次只能运行一个任务。这导致Node大量采用异步操作（asynchronous opertion），即任务不是马上执行，而是插在任务队列的尾部，等到前面的任务运行完后再执行。

由于这种特性，某一个任务的后续操作，往往采用回调函数（callback）的形式进行定义。

```
var isTrue = function(value, callback) {
  if (value === true) {
    callback(null, "Value was true.");
  }
  else {
    callback(new Error("Value is not true!"));
  }
}
```

上面代码就把进一步的处理，交给回调函数callback。

Node约定，如果某个函数需要回调函数作为参数，则回调函数是最后一个参数。另外，回调函数本身的第一个参数，约定为上一步传入的错误对象。

```
var callback = function (error, value) {
  if (error) {
    return console.log(error);
  }
  console.log(value);
}
```

上面代码中，callback的第一个参数是Error对象，第二个参数才是真正的数据参数。这是因为回调函数主要用于异步操作，当回调函数运行时，前期的操作早结束了，错误的执行栈早就不存在了，传统的错误捕捉机制try…catch对于异步操作行不通，所以只能把错误交给回调函数处理。

```
try {
  db.User.get(userId, function(err, user) {
    if(err) {
      throw err
    }
    // ...
  })
} catch(e) {
  console.log('Oh no!');
}
```

上面代码中，db.User.get方法是一个异步操作，等到抛出错误时，可能它所在的try…catch代码块早就运行结束了，这会导致错误无法被捕捉。所以，Node统一规定，一旦异步操作发生错误，就把错误对象传递到回调函数。

如果没有发生错误，回调函数的第一个参数就传入null。这种写法有一个很大的好处，就是说只要判断回调函数的第一个参数，就知道有没有出错，如果不是null，就肯定出错了。另外，这样还可以层层传递错误。

```
if(err) {
  // 除了放过No Permission错误意外，其他错误传给下一个回调函数
  if(!err.noPermission) {
    return next(err);
  }
}
```

1.6 全局对象和全局变量

Node提供以下几个全局对象，它们是所有模块都可以调用的。

- **global**：表示Node所在的全局环境，类似于浏览器的window对象。需要注意的是，如果在浏览器中声明一个全局变量，实际上是声明了一个全局对象的属性，比如 `var x = 1` 等同于设置 `window.x = 1`，但是Node不是这样，至少在模块中不是这样（REPL环境的行为与浏览器一致）。在模块文件中，声明 `var x = 1`，该变量不是 `global` 对象的属性，`global.x` 等于undefined。这是因为模块的全局变量都是该模块私有的，其他模块无法取到。
- **process**：该对象表示Node所处的当前进程，允许开发者与该进程互动。
- **console**：指向Node内置的console模块，提供命令行环境中的标准输入、标准输出功能。

Node还提供一些全局函数。

- **setTimeout()**：用于在指定毫秒之后，运行回调函数。实际的调用间隔，还取决于系统因素。间隔的毫秒数在1毫秒到2,147,483,647毫秒（约24.8天）之间。如果超过这个范围，会被自动改为1毫秒。该方法返回一个整数，代表这个新建定时器的编号。
- **clearTimeout()**：用于终止一个setTimeout方法新建的定时器。
- **setInterval()**：用于每隔一定毫秒调用回调函数。由于系统因素，可能无法保证每次调用之间正好间隔指定的毫秒数，但只会多于这个间隔，而不会少于它。指定的毫秒数必须是1到2,147,483,647（大约24.8天）之间的整数，如果超过这个范围，会被自动改为1毫秒。该方法返回一个整数，代表这个新建定时器的编号。
- **clearInterval()**：终止一个用setInterval方法新建的定时器。
- **require()**：用于加载模块。
- **Buffer()**：用于操作二进制数据。

Node提供两个全局变量，都以两个下划线开头。

- `__filename`：指向当前运行的脚本文件名。
- `__dirname`：指向当前运行的脚本所在的目录。

除此之外，还有一些对象实际上是模块内部的局部变量，指向的对象根据模块不同而不同，但是所有模块都适用，可以看作是伪全局变量，主要为module, module.exports, exports等。

2. 模块化结构

2.1 概述

Node.js采用模块化结构，按照CommonJS规范定义和使用模块。模块与文件是一一对应关系，即加载一个模块，实际上就是加载对应的一个模块文件。

require命令用于指定加载模块，加载时可以省略脚本文件的后缀名。

```
var circle = require('./circle.js');
// 或者
var circle = require('./circle');
```

require方法的参数是模块文件的名字。它分成两种情况，第一种情况是参数中含有文件路径（比如上例），这时路径是相对于当前脚本所在的目录，第二种情况是参数中不含有文件路径，这时Node到模块的安装目录，去寻找已安装的模块（比如下例）。

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

有时候，一个模块本身就是一个目录，目录中包含多个文件。这时候，Node在package.json文件中，寻找main属性所指明的模块入口文件。

```
{
  "name" : "bar",
  "main" : "./lib/bar.js"
}
```

上面代码中，模块的启动文件为lib子目录下的bar.js。当使用 `require('bar')` 命令加载该模块时，实际上加载的是 `./node_modules/bar/lib/bar.js` 文件。下面写法会起到同样效果。

```
var bar = require('bar/lib/bar.js')
```

如果模块目录中没有package.json文件，node.js会尝试在模块目录中寻找index.js或index.node文件进行加载。

模块一旦被加载以后，就会被系统缓存。如果第二次还加载该模块，则会返回缓存中的版本，这意味着模块实际上只会执行一次。如果希望模块执行多次，则可以让模块返回一个函数，然后多次调用该函数。

2.2 核心模块

如果只是在服务器运行JavaScript代码，用处并不大，因为服务器脚本语言已经有很多种了。Node.js的用处在于，它本身还提供了一系列功能模块，与操作系统互动。这些核心的功能模块，不用安装就可以使用，下面是它们的清单。

- **http**：提供HTTP服务器功能。
- **url**：解析URL。
- **fs**：与文件系统交互。
- **querystring**：解析URL的查询字符串。
- **child_process**：新建子进程。
- **util**：提供一系列实用小工具。
- **path**：处理文件路径。
- **crypto**：提供加密和解密功能，基本上是对OpenSSL的包装。

上面这些核心模块，源码都在Node的lib子目录中。为了提高运行速度，它们安装时都会被编译成二进制文件。

核心模块总是最优先加载的。如果你自己写了一个HTTP模块， `require('http')` 加载的还是核心模块。

2.3 自定义模块

Node模块采用CommonJS规范。只要符合这个规范，就可以自定义模块。

下面是一个最简单的模块，假定新建一个foo.js文件，写入以下内容。

```
// foo.js

module.exports = function(x) {
    console.log(x);
};
```

上面代码就是一个模块，它通过module.exports变量，对外输出一个方法。

这个模块的使用方法如下。

```
// index.js

var m = require('./foo');

m("这是自定义模块");
```

上面代码通过require命令加载模块文件foo.js（后缀名省略），将模块的对外接口输出到变量m，然后调用m。这时，在命令行下运行index.js，屏幕上就会输出“这是自定义模块”。

```
$ node index
这是自定义模块
```

module变量是整个模块文件的顶层变量，它的exports属性就是模块向外输出的接口。如果直接输出一个函数（就像上面的foo.js），那么调用模块就是调用一个函数。但是，模块也可以输出一个对象。下面对foo.js进行改写。

```
// foo.js

var out = new Object();

function p(string) {
  console.log(string);
}

out.print = p;

module.exports = out;
```

上面的代码表示模块输出out对象，该对象有一个print属性，指向一个函数。下面是这个模块的使用方法。

```
// index.js

var m = require('./foo');

m.print("这是自定义模块");
```

上面代码表示，由于具体的方法定义在模块的print属性上，所以必须显式调用print属性。

3. 异常处理

Node是单线程运行环境，一旦抛出的异常没有被捕获，就会引起整个进程的崩溃。所以，Node的异常处理对于保证系统的稳定运行非常重要。

一般来说，Node有三种方法，传播一个错误。

- 使用throw语句抛出一个错误对象，即抛出异常。
- 将错误对象传递给回调函数，由回调函数负责发出错误。
- 通过EventEmitter接口，发出一个error事件。

3.1 try…catch结构

最常用的捕获异常的方式，就是使用try…catch结构。但是，这个结构无法捕获异步运行的代码抛出的异常。

```
try {
  process.nextTick(function () {
    throw new Error("error");
  });
} catch (err) {
  //can not catch it
  console.log(err);
}

try {
  setTimeout(function(){
    throw new Error("error");
  },1)
} catch (err) {
  //can not catch it
  console.log(err);
}
```

上面代码分别用process.nextTick和setTimeout方法，在下一轮事件循环抛出两个异常，代表异步操作抛出的错误。它们都无法被catch代码块捕获，因为catch代码块所在的那部分已经运行结束了。

一种解决方法是将错误捕获代码，也放到异步执行。

```
function async(cb, err) {
  setTimeout(function() {
    try {
      if (true)
        throw new Error("woops!");
      else
        cb("done");
    } catch(e) {
      err(e);
    }
  }, 2000)
}

async(function(res) {
  console.log("received:", res);
}, function(err) {
  console.log("Error: async threw an exception:", err);
});
// Error: async threw an exception: Error: woops!
```

上面代码中，async函数异步抛出的错误，可以同样部署在异步的catch代码块捕获。

这两种处理方法都不太理想。一般来说，Node只在很少场合才用try/catch语句，比如使用 `JSON.parse` 解析JSON文本。

3.2 回调函数

Node采用的方法，是将错误对象作为第一个参数，传入回调函数。这样就避免了捕获代码与发生错误的代码不在同一个时间段的问题。

```
fs.readFile('/foo.txt', function(err, data) {
  if (err !== null) throw err;
  console.log(data);
});
```

上面代码表示，读取文件 foo.txt 是一个异步操作，它的回调函数有两个参数，第一个是错误对象，第二个是读取到的文件数据。如果第一个参数不是null，就意味着发生错误，后面代码也就不再执行了。

下面是一个完整的例子。

```
function async2(continuation) {
  setTimeout(function() {
    try {
      var res = 42;
      if (true)
        throw new Error("woops!");
      else
        continuation(null, res); // pass 'null' for error
    } catch(e) {
      continuation(e, null);
    }
  }, 2000);
}

async2(function(err, res) {
  if (err)
    console.log("Error: (cps) failed:", err);
  else
    console.log("(cps) received:", res);
});
// Error: (cps) failed: woops!
```

上面代码中，async2函数的回调函数的第一个参数就是一个错误对象，这是为了处理异步操作抛出的错误。

3.3 EventEmitter接口的error事件

发生错误的时候，也可以用EventEmitter接口抛出error事件。

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

emitter.emit('error', new Error('something bad happened'));
```

使用上面的代码必须小心，因为如果没有对error事件部署监听函数，会导致整个应用程序崩溃。所以，一般总是必须同时部署下面的代码。

```
emitter.on('error', function(err) {
  console.error('出错：' + err.message);
});
```

3.4 uncaughtException事件

当一个异常未被捕获，就会触发uncaughtException事件，可以对这个事件注册回调函数，从而捕获异常。

```
var logger = require('tracer').console();
process.on('uncaughtException', function(err) {
  console.error('Error caught in uncaughtException event:', err);
});

try {
  setTimeout(function(){
    throw new Error("error");
  },1);
} catch (err) {
  //can not catch it
  console.log(err);
}
```

只要给uncaughtException配置了回调，Node进程不会异常退出，但异常发生的上下文已经丢失，无法给出异常发生的详细信息。而且，异常可能导致Node不能正常进行内存回收，出现内存泄露。所以，当uncaughtException触发后，最好记录错误日志，然后结束Node进程。

```
process.on('uncaughtException', function(err) {
  logger.log(err);
  process.exit(1);
});
```

3.5 unhandledRejection事件

iojs有一个unhandledRejection事件，用来监听没有捕获的Promise对象的rejected状态。

```
var promise = new Promise(function(resolve, reject) {
  reject(new Error("Broken."));
});

promise.then(function(result) {
  console.log(result);
})
```

上面代码中，promise的状态变为rejected，并且抛出一个错误。但是，不会有任何反应，因为没有设置任何处理函数。

只要监听unhandledRejection事件，就能解决这个问题。

```
process.on('unhandledRejection', function (err, p) {
  console.error(err.stack);
})
```

需要注意的是，unhandledRejection事件的监听函数有两个参数，第一个是错误对象，第二个是产生错误的promise对象。这可以提供很多有用的信息。

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

http.createServer(function (req, res) {
  var promise = new Promise(function(resolve, reject) {
    reject(new Error("Broken."))
  })

  promise.info = {url: req.url}
}).listen(8080)

process.on('unhandledRejection', function (err, p) {
  if (p.info && p.info.url) {
    console.log('Error in URL', p.info.url)
  }
  console.error(err.stack)
})
```

上面代码会在出错时，输出用户请求的网址。

```
Error in URL /testurl
Error: Broken.
  at /Users/mikeal/tmp/test.js:9:14
  at Server.<anonymous> (/Users/mikeal/tmp/test.js:4:17)
  at emitTwo (events.js:87:13)
  at Server.emit (events.js:169:7)
  at HTTPParser.parserOnIncoming [as onIncoming] (_http_server.js:471:12)
  at HTTPParser.parserOnHeadersComplete (_http_common.js:88:23)
  at Socket.socketOnData (_http_server.js:322:22)
  at emitOne (events.js:77:13)
  at Socket.emit (events.js:166:7)
  at readableAddChunk (_stream_readable.js:145:16)
```

4. 命令行脚本

node脚本可以作为命令行脚本使用。

```
$ node foo.js
```

上面代码执行了foo.js脚本文件。

foo.js文件的第一行，如果加入了解释器的位置，就可以将其作为命令行工具直接调用。

```
#!/usr/bin/env node
```

调用前，需更改文件的执行权限。

```
$ chmod u+x foo.js
$ ./foo.js arg1 arg2 ...
```

作为命令行脚本时， `console.log` 用于输出内容到标准输出， `process.stdin` 用于读取标准输入， `child_process.exec()` 用于执行一个shell命令。

5. 参考链接

[1] Cody Lindley, Package Managers: An Introductory Guide For The Uninitiated Front-End Developer

[2] Stack Overflow, What is Node.js?

[3] Andrew Burgess, Using Node's Event Module

[4] James Halliday, task automation with npm run- Romain Prieto, Working on related Node.js modules locally

[5] Alon Salant, Export This: Interface Design Patterns for Node.js Modules

[6] Node.js Manual & Documentation, Modules

[7] Brent Ertz, Creating and publishing a node.js module

[8] Fred K Schott, "npm install –save" No Longer Using Tildes

[9] Satans17, Node稳定性的研究心得

[10] Axel Rauschmayer, Write your shell scripts in JavaScript, via Node.js

留言

comments powered by Disqus (http://disqus.com)