Node.js 路由 Node.js 常用工具

Node.js 全局对象

JavaScript 中有一个特殊的对象，称为全局对象（Global Object），它及其所有属性都可以在程序的任何地方访问，即全局变量。

在浏览器 JavaScript 中，通常 window 是全局对象， 而 Node.js 中的全局对象是 global，所有全局变量（除了 global 本身以外）都是 global 对象的属性。

在 Node.js 我们可以直接访问到 global 的属性，而不需要在应用中包含它。

全局对象与全局变量

global 最根本的作用是作为全局变量的宿主。按照 ECMAScript 的定义，满足以下条 件的变量是全局变量：

- 在最外层定义的变量；
- 全局对象的属性；
- 隐式定义的变量（未定义直接赋值的变量）。

当你定义一个全局变量时，这个变量同时也会成为全局对象的属性，反之亦然。需要注 意的是，在 Node.js 中你不可能在最外层定义变量，因为所有用户代码都是属于当前模块的， 而模块本身不是最外层上下文。

注意： 永远使用 var 定义变量以避免引入全局变量，因为全局变量会污染 命名空间，提高代码的耦合风险。

__filename

__filename 表示当前正在执行的脚本的文件名。它将输出文件所在位置的绝对路径，且和命令行参数所指定的文件名不一定相同。 如果在模块中，返回的值是模块文件的路径。

实例

创建文件 main.js ，代码如下所示：

```
// 输出全局变量 __filename 的值
console.log( __filename );
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
/web/com/runoob/nodejs/main.js
```

__dirname

__dirname 表示当前执行脚本所在的目录。

实例

创建文件 main.js ，代码如下所示：

```
// 输出全局变量 __dirname 的值
console.log( __dirname );
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
/web/com/runoob/nodejs
```

---

setTimeout(cb, ms)

setTimeout(cb, ms) 全局函数在指定的毫秒(ms)数后执行指定函数(cb)。：setTimeout() 只执行一次指定函数。

返回一个代表定时器的句柄值。

实例

创建文件 main.js ，代码如下所示：

```
function printHello(){
   console.log( "Hello, World!");
}
// 两秒后执行以上函数
setTimeout(printHello, 2000);
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
Hello, World!
```

---

clearTimeout(t)

clearTimeout(t) 全局函数用于停止一个之前通过 setTimeout() 创建的定时器。 参数 **t** 是通过 setTimeout() 函数创建的定时器。

实例

创建文件 main.js ，代码如下所示：

```
function printHello(){
   console.log( "Hello, World!");
}
// 两秒后执行以上函数
var t = setTimeout(printHello, 2000);
```

```
// 清除定时器
clearTimeout(t);
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
```

---

setInterval(cb, ms)

setInterval(cb, ms) 全局函数在指定的毫秒(ms)数后执行指定函数(cb)。

返回一个代表定时器的句柄值。可以使用 **clearInterval(t)** 函数来清除定时器。

setInterval() 方法会不停地调用函数，直到 clearInterval() 被调用或窗口被关闭。

实例

创建文件 main.js ，代码如下所示：

```
function printHello(){
   console.log( "Hello, World!");
}
// 两秒后执行以上函数
setInterval(printHello, 2000);
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
```

Hello, World! Hello, World! Hello, World! Hello, World! Hello, World! ……
以上程序每隔两秒就会输出一次"Hello, World!"，且会永久执行下去，直到你按下 **ctrl + c** 按钮。

---

console

console 用于提供控制台标准输出，它是由 Internet Explorer 的 JScript 引擎提供的调试工具，后来逐渐成为浏览器的实施标准。

Node.js 沿用了这个标准，提供与习惯行为一致的 console 对象，用于向标准输出流（stdout）或标准错误流（stderr）输出字符。

console 方法

以下为 console 对象的方法:

序号	方法 & 描述
1	**console.log([data][, ...])** 向标准输出流打印字符并以换行符结束。该方法接收若干 个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则 以类似于C 语言 printf() 命令的格式输出。

2	**console.info([data][, ...])** 该命令的作用是返回信息性消息，这个命令与console.log差别并不大，除了在chrome中只会输出文字外，其余的会显示一个蓝色的惊叹号。
3	**console.error([data][, ...])** 输出错误消息的。控制台在出现错误时会显示是红色的叉子。
4	**console.warn([data][, ...])** 输出警告消息。控制台出现有黄色的惊叹号。
5	**console.dir(obj[, options])** 用来对一个对象进行检查（inspect），并以易于阅读和打印的格式显示。
6	**console.time(label)** 输出时间，表示计时开始。
7	**console.timeEnd(label)** 结束时间，表示计时结束。
8	**console.trace(message[, ...])** 当前执行的代码在堆栈中的调用路径，这个测试函数运行很有帮助，只要给想测试的函数里面加入 console.trace 就行了。
9	**console.assert(value[, message][, ...])** 用于判断某个表达式或变量是否为真，接收两个参数，第一个参数是表达式，第二个参数是字符串。只有当第一个参数为false，才会输出第二个参数，否则不会有任何结果。

console.log()：向标准输出流打印字符并以换行符结束。
console.log 接收若干 个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则 以类似于C 语言 printf () 命令的格式输出。
第一个参数是一个字符串，如果没有 参数，只打印一个换行。

```
console.log('Hello world');
console.log('byvoid%diovyb');
console.log('byvoid%diovyb', 1991);
```

运行结果为：

```
Hello world
byvoid%diovyb
byvoid1991iovyb
```

- console.error()：与console.log() 用法相同，只是向标准错误流输出。
- console.trace()：向标准错误流输出当前的调用栈。

```
console.trace();
```

运行结果为：

```
Trace:
at Object.<anonymous> (/home/byvoid/consoletrace.js:1:71)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

实例

创建文件 main.js ，代码如下所示：

```
console.info("程序开始执行: ");

var counter = 10;
console.log("计数: %d", counter);

console.time("获取数据");
//
// 执行一些代码
//
console.timeEnd('获取数据');

console.info("程序执行完毕。")
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
程序开始执行:
计数: 10
获取数据: 0ms
程序执行完毕
```

process

process 是一个全局变量，即 global 对象的属性。

它用于描述当前Node.js 进程状态的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要和它打交道。下面将会介绍 process 对象的一些最常用的成员方法。

序号	事件 & 描述
1	**exit** 当进程准备退出时触发。
2	**beforeExit** 当 node 清空事件循环，并且没有其他安排时触发这个事件。通常来说，当没有进程安排时 node 退出，但是 'beforeExit' 的监听器可以异步调用，这样 node 就会继续执行。
3	**uncaughtException** 当一个异常冒泡回到事件循环，触发这个事件。如果给异常添加了监视器，默认的操作（打印堆栈跟踪信息并退出）就不会发生。
4	**Signal 事件** 当进程接收到信号时就触发。信号列表详见标准的 POSIX 信号名，如 SIGINT、SIGUSR1 等。

实例

创建文件 main.js ，代码如下所示：

```
process.on('exit', function(code) {

  // 以下代码永远不会执行
  setTimeout(function() {
    console.log("该代码不会执行");
  }, 0);

  console.log('退出码为:', code);
});
console.log("程序执行结束");
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
程序执行结束
退出码为: 0
```

退出状态码

退出状态码如下所示：

状态码	名称 & 描述

1	**Uncaught Fatal Exception** 有未捕获异常，并且没有被域或 uncaughtException 处理函数处理。
2	**Unused** 保留
3	**Internal JavaScript Parse Error** JavaScript的源码启动 Node 进程时引起解析错误。非常罕见，仅会在开发 Node 时才会有。
4	**Internal JavaScript Evaluation Failure** JavaScript 的源码启动 Node 进程，评估时返回函数失败。非常罕见，仅会在开发 Node 时才会有。
5	**Fatal Error** V8 里致命的不可恢复的错误。通常会打印到 stderr ，内容为： FATAL ERROR
6	**Non-function Internal Exception Handler** 未捕获异常，内部异常处理函数不知为何设置为on-function，并且不能被调用。
7	**Internal Exception Handler Run-Time Failure** 未捕获的异常， 并且异常处理函数处理时自己抛出了异常。例如，如果 process.on('uncaughtException') 或 domain.on('error') 抛出了异常。
8	**Unused** 保留
9	**Invalid Argument** 可能是给了未知的参数，或者给的参数没有值。
10	**Internal JavaScript Run-Time Failure** JavaScript的源码启动 Node 进程时抛出错误，非常罕见，仅会在开发 Node 时才会有。
12	**Invalid Debug Argument** 设置了参数--debug 和/或 --debug-brk，但是选择了错误端口。
128	**Signal Exits** 如果 Node 接收到致命信号，比如SIGKILL 或 SIGHUP，那么退出代码就是128 加信号代码。这是标准的 Unix 做法，退出信号代码放在高位。

Process 属性

Process 提供了很多有用的属性，便于我们更好的控制系统的交互：

序号.	属性 & 描述
1	**stdout**

	标准输出流。
2	**stderr** 标准错误流。
3	**stdin** 标准输入流。
4	**argv** argv 属性返回一个数组，由命令行执行脚本时的各个参数组成。它的第一个成员总是node，第二个成员是脚本文件名，其余成员是脚本文件的参数。
5	**execPath** 返回执行当前脚本的 Node 二进制文件的绝对路径。
6	**execArgv** 返回一个数组，成员是命令行下执行脚本时，在Node可执行文件与脚本文件之间的命令行参数。
7	**env** 返回一个对象，成员为当前 shell 的环境变量
8	**exitCode** 进程退出时的代码，如果进程优通过 process.exit() 退出，不需要指定退出码。
9	**version** Node 的版本，比如v0.10.18。
10	**versions** 一个属性，包含了 node 的版本和依赖.
11	**config** 一个包含用来编译当前 node 执行文件的 javascript 配置选项的对象。它与运行 ./configure 脚本生成的 "config.gypi" 文件相同。
12	**pid** 当前进程的进程号。
13	**title** 进程名，默认值为"node"，可以自定义该值。
14	**arch** 当前 CPU 的架构：'arm'、'ia32' 或者 'x64'。

15	**platform** 运行程序所在的平台系统 'darwin', 'freebsd', 'linux', 'sunos' 或 'win32'
16	**mainModule** require.main 的备选方法。不同点，如果主模块在运行时改变，require.main可能会继续返回老的模块。可以认为，这两者引用了同一个模块。

实例

创建文件 main.js ，代码如下所示：

```
// 输出到终端
process.stdout.write("Hello World!" + "\n");

// 通过参数读取
process.argv.forEach(function(val, index, array) {
   console.log(index + ': ' + val);
});

// 获取执行路径
console.log(process.execPath);


// 平台信息
console.log(process.platform);
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
Hello World!
0: node
1: /web/www/node/main.js
/usr/local/node/0.10.36/bin/node
darwin
```

方法参考手册

Process 提供了很多有用的方法，便于我们更好的控制系统的交互：

序号	方法 & 描述
1	**abort()** 这将导致 node 触发 abort 事件。会让 node 退出并生成一个核心文件。
2	**chdir(directory)** 改变当前工作进程的目录，如果操作失败抛出异常。

3	**cwd()** 返回当前进程的工作目录
4	**exit([code])** 使用指定的 code 结束进程。如果忽略，将会使用 code 0。
5	**getgid()** 获取进程的群组标识（参见 getgid(2)）。获取到得时群组的数字 id，而不是名字。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
6	**setgid(id)** 设置进程的群组标识（参见 setgid(2)）。可以接收数字 ID 或者群组名。如果指定了群组名，会阻塞等待解析为数字 ID 。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
7	**getuid()** 获取进程的用户标识(参见 getuid(2))。这是数字的用户 id，不是用户名。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
8	**setuid(id)** 设置进程的用户标识（参见setuid(2)）。接收数字 ID或字符串名字。果指定了群组名，会阻塞等待解析为数字 ID 。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
9	**getgroups()** 返回进程的群组 iD 数组。POSIX 系统没有保证一定有，但是 node.js 保证有。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
10	**setgroups(groups)** 设置进程的群组 ID。这是授权操作，所以你需要有 root 权限，或者有 CAP_SETGID 能力。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
11	**initgroups(user, extra_group)** 读取 /etc/group ，并初始化群组访问列表，使用成员所在的所有群组。这是授权操作，所以你需要有 root 权限，或者有 CAP_SETGID 能力。 注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
12	**kill(pid[, signal])** 发送信号给进程. pid 是进程id，并且 signal 是发送的信号的字符串描述。信号名是字符串，比如 'SIGINT' 或 'SIGHUP'。如果忽略，信号会是 'SIGTERM'。
13	**memoryUsage()**

	返回一个对象，描述了 Node 进程所用的内存状况，单位为字节。
14	**nextTick(callback)** 一旦当前事件循环结束，调用回调函数。
15	**umask([mask])** 设置或读取进程文件的掩码。子进程从父进程继承掩码。如果mask 参数有效，返回旧的掩码。否则，返回当前掩码。
16	**uptime()** 返回 Node 已经运行的秒数。
17	**hrtime()** 返回当前进程的高分辨时间，形式为 [seconds, nanoseconds]数组。它是相对于过去的任意事件。该值与日期无关，因此不受时钟漂移的影响。主要用途是可以通过精确的时间间隔，来衡量程序的性能。 你可以将之前的结果传递给当前的 process.hrtime() ，会返回两者间的时间差，用来基准和测量时间间隔。

实例

创建文件 main.js ，代码如下所示：

```
// 输出当前目录
console.log('当前目录: ' + process.cwd());

// 输出当前版本
console.log('当前版本: ' + process.version);

// 输出内存使用情况
console.log(process.memoryUsage());
```

执行 main.js 文件，代码如下所示:

```
$ node main.js
当前目录: /web/com/runoob/nodejs
当前版本: v0.10.36
{ rss: 12541952, heapTotal: 4083456, heapUsed: 2157056 }
```

← Node.js 路由 Node.js 常用工具 →