global

> 本文介绍Node中的全局对象，包括global本身以及控制台输出对象Console、require函数、定时器相关方法以及__filename和__dirname等全局变量。

1.0 global全局对象

在Node中定义了一个 `global` 对象，代表全局命名空间，所有的全局变量、函数或对象都作为 `global全局对象` 的成员。我们可以在REPL环境中直接通过变量来进行查看，下面列出核心成员(细节有省略)。

```
wendingding$ node
> global
{ console: [Getter],
  DTRACE_NET_SERVER_CONNECTION: [Function],
  DTRACE_NET_STREAM_END: [Function],
  DTRACE_HTTP_SERVER_REQUEST: [Function],
  DTRACE_HTTP_SERVER_RESPONSE: [Function],
  DTRACE_HTTP_CLIENT_REQUEST: [Function],
  DTRACE_HTTP_CLIENT_RESPONSE: [Function],
  global: [Circular],
  process:
    process { ···省略··· },
  Buffer: { ···省略··· },
  clearImmediate: [Function],
  clearInterval: [Function],
  clearTimeout: [Function],
  setImmediate: { [Function: setImmediate] [Symbol(util.promisify.custom)]: [Function] },
  setInterval: [Function],
  setTimeout: { [Function: setTimeout] [Symbol(util.promisify.custom)]: [Function] },
  module:
    Module { ···省略··· },
  require: { ···省略··· }
```

通过打印输出我们发现， `global全局对象` 中包含很多的成员，比如用于控制台输出的console对象、处理底层网络请求的一系列函数以及相对复杂的process对象等，这里先给出整体结构图然后再分别介绍。

下面列出的是全局对象跟网络连接相关的一系列方法(这里不做深入)。

```
DTRACE_NET_STREAM_END: [Function],
DTRACE_NET_SERVER_CONNECTION: [Function],
DTRACE_HTTP_SERVER_REQUEST: [Function],
DTRACE_HTTP_SERVER_RESPONSE: [Function],
DTRACE_HTTP_CLIENT_REQUEST: [Function],
DTRACE_HTTP_CLIENT_RESPONSE: [Function],
```

在 `global全局对象` 中有一个 `global` 属性，该属性指向的是自身，而且全局对象中所有的成员都可以直接通过成员的名称来进行访问(这点跟前端开发中的 `window` 对象类似)，下面给出简单示例代码。

```
wendingding$ node
> global.global == global
true
> global == this
true
> global.module
Module {
  id: '<repl>',
  exports: {},
  parent: undefined,
  filename: null,
  loaded: false,
  children: [],
  paths:
   [ '/Users/文顶顶/Desktop/global/repl/node_modules',
     '/Users/文顶顶/Desktop/global/node_modules',
     '/Users/文顶顶/Desktop/node_modules',
     '/Users/文顶顶/node_modules',
     '/Users/node_modules',
     '/node_modules',
     '/Users/文顶顶/.node_modules',
     '/Users/文顶顶/.node_libraries',
     '/usr/local/lib/node' ] }
> module.id
'<repl>'
```

2.0 Console控制台输出

`Console` 对象主要用于控制台输出，该对象中拥有诸多方法，作用各异但差别不大。

```
wendingding$ node
> console.log(console)
Console {
  log: [Function: bound consoleCall],
  info: [Function: bound consoleCall],
  warn: [Function: bound consoleCall],
  error: [Function: bound consoleCall],
  dir: [Function: bound consoleCall],
  time: [Function: bound consoleCall],
  timeEnd: [Function: bound consoleCall],
  clear: [Function: bound consoleCall],
```

```
  group: [Function: bound consoleCall],
  groupCollapsed: [Function: bound consoleCall],
  groupEnd: [Function: bound consoleCall],
  Console: [Function: Console],
  ...省略...
  context: [Function: context],
  [Symbol(counts)]: Map {} }
```

我们在开发中使用控制台输出常用的是 Console.log 方法，该方法用于进行标准输出流的输出，也就是在控制台中打印和显示N行字符串信息，使用方式非常简单而且灵活。

```
console.log("001 我是字符串");
console.log("002 我是需要传递参数的字符串，参数为%s"," XXX");
console.log("002 我是需要传递参数的字符串，参数为%s"," XXX","and Other");

console.log("003 控制输出数字=> %d",10.123);
console.log("003 控制输出符号=> %%",30);

console.log("004 控制输出简单计算的结果=> ",3 + 3);
var a = 10,b = 10;
console.log(a == b);

console.info("005 我也是字符串");
```

列出上面代码的输出。

BASH

```
wendingding$ node Console.js
001 我是字符串
002 我是需要传递参数的字符串，参数为 XXX
002 我是需要传递参数的字符串，参数为 XXX and Other
003 控制输出数字=> 10.123
003 控制输出符号=> % 30
004 控制输出简单计算的结果=>  6
true
005 我也是字符串
wendingding$ node Console.js > log.text
```

console 对象的 log、info、error、warn 方法在使用上几乎没有任何的差别，而且都支持对输出流进行重定向操作(使用>符号)，而 dir 方法则可以查看并打印对象的详细内容，在需要查看对象或函数细节的时候会比较有用，下面给出简单示例。

```
//语法  console.dir(obj , [ options ] )
//参数
//obj               要打印的目标对象
//options           用来控制打印的可选配置对象，主要配置项如下
//[1]  colors:      布尔类型值，输出的信息是否有颜色
//[2]  depth:       告诉内部的util.inspect()格式化对象时要递归多少次，默认为2，null则无限递归
//[3]  showHidden   设置为true则显示不可枚举属性和 symbol 属性。
wendingding$ node
> var obj = {name:"zs",show:function(){console.log(this.name)}};
undefined
> obj
{ name: 'zs', show: [Function: show] }
```

```
{ name: 'zs', show: [Function: show] }

> console.dir(obj,{showHidden:true,depth:1})
{ name: 'zs',
  show:
   { [Function: show]
     [length]: 0,
     [name]: 'show',
     [arguments]: null,
     [caller]: null,
     [prototype]: show { [constructor]: [Circular] } } }
```

console.time()和console.endTime() 方法用以计算一个操作的持续时间，两个方法需要配对使用，它们接收同一个字符串作为参数以标记开始和结束时间线，下面给出简短示例。

```
//开始计时(任务标记为for-10000)
console.time("for-10000");

for(var i = 0; i<10000;i++){}
//结束计时
console.timeEnd("for-10000");

//REPL环境中执行代码:
wendingding$ node Console.js
for-10000: 0.209ms
```

console.group()和console.groupEnd() 方法用来设置后面的输出缩进增加或减少两个空格，在8.5.0版本中新增的 groupCollapsed 方法是 group 方法的别名，另外 clear 方法用来清空控制台消息。

```
bogon:fs wendingding$ node
>  console.log("参照字符串");
参照字符串
> console.group("将后续行的缩进增加两个空格")
将后续行的缩进增加两个空格
>  console.log("参照字符串");
  参照字符串
> console.groupEnd("将后续行的缩进减少两个空格")
>  console.log("参照字符串");
参照字符串
>console.clear();
```

除了全局的 console 实例外，console对象(模块)还在内部提供了Console 类可用来创建一个具有可配置的输出流的简单记录器，代码中通过 require("console").Console 或 console.Console 使用，具体的使用方式请参考官方文档，这里简单列出该类的结构。

```
wendingding$ node
> console.dir(console.Console,{depth:1,showHidden:true})
{ [Function: Console]
  [length]: 2,
  [name]: 'Console',
  [prototype]:
   Console {
     [constructor]: [Circular]
```

```
    [constructor]: [Circular],
    log: [Object],
    info: [Object],
    warn: [Object],
    error: [Object],
    dir: [Object],
    time: [Object],
    timeEnd: [Object],
    trace: [Object],
    assert: [Object],
    clear: [Object],
    count: [Object],
    countReset: [Object],
    group: [Object],


    groupCollapsed: [Object],
    groupEnd: [Object] } }
```

3.0 定时器相关方法

Node中的 `timer模块` 提供了全局的定时器API，这些方法的作用和Web浏览器提供的定时器方法类似。
在实现上，Node中的定时器方法基于Node事件循环来进行构建。

> **定时器相关方法·语法介绍**
>
> ```
> setImmediate(callback[, ...args])          预定立即执行的回调函数，它在I/O事件回调之后被触发
> setInterval(callback, delay[, ...args])    每间隔固定时间就执行一次回调函数
> setTimeout(callback, delay[, ...args])     固定时间后执行一次回调函数
> clearImmediate(immediate)
> clearInterval(timeout)                     取消定时器
> clearTimeout(timeout)                      取消定时器
> ```

在上面列出的这些方法中， `callback` 均表示回调函数，如果该参数不是函数类型，那么在执行代码的时候将抛出TypeError。 `delay` 表示时间(计量单位为毫秒)，需注意delay的取值范围为1~2147483647，如果超出该范围那么delay的值将会会被设为 1。 `args` 表示的是当调用回调函数时传递给回调函数的实际参数，需注意 `args` 的传参格式是参数列表而非数组。

`clearxxx` 系列方法均用来取消定时器，这些函数接收一个 `timeout` 类型的实例对象(该类型的实例对象是内部创建的，作为 `setTimeout()` 或 `setInterval()` 的返回值)。

```
//001 开启定时器
//2秒后执行回调函数，10和20作为回调函数的参数
var timer1 = setTimeout(function(a,b){
  console.log("setTimeout方法=>%d",(a +b));
},2000,10,20)

//002 开启定时器
//每隔1秒就执行一次回调函数，Nice作为回调函数的参数
var timer2 = setInterval(function(param){
  console.log("setInterval=>参数: ",param);
} 1000,"Nice!")
```

```
},1000, Nice! )

//003 取消定时器
//5秒之后执行回调函数，取消time2定时器
setTimeout(function(){
  console.log("取消定时器timer2");
  clearInterval(timer2)
},5000)

//timer2.unref()
//当调用该方法时，活动的定时器对象对象不要求 Node.js 事件循环保持活动。
//如果没有其他活动保持事件循环运行，则进程可能在定时器对象的回调函数被调用之前退出。
//注意：多次调用定时器对象的unref() 方法没有效果。

//timer2.ref()
//当调用该方法时，只要定时器对象处于活动状态就要求Node事件循环不要退出。
//注意：多次调用定时器对象的ref()方法没有效果。
//注意：默认所有的定时器对象都是"ref"的，通常不需要调用ref()方法，除非之前调用了unref()。
```

说明 定时器对象的 `unref` 方法能够取消回调函数的调用。当指定定时器实例对象的回调函数被取消后，可以通过对应的 `ref` 方法来恢复调用，下面列出代码的执行情况：

```
wendingding$ node timer.js
setInterval=>参数:  Nice!
setTimeout方法=>30
setInterval=>参数:  Nice!
setInterval=>参数:  Nice!
setInterval=>参数:  Nice!
取消定时器timer2
```

4.0 __filename和__dirname

`__filename` 和 `__dirname` 是Node中预定义的两个变量，分别用来获取当前模块的文件名(全路径)以及当前的目录名，这两个变量在任何模块文件的内部均可使用。

> ❏ **__filename** 获取当前模块文件的完整绝对路径(文件名)
> ❏ **__dirname** 获取当前模块所在目录的完整绝对路径(目录名)

我们可以在JavaScript文件中打印 `__filename` 和 `__dirname` 变量，然后运行查看结果。

```
//备注：在timer.js文件中打印变量
console.log(__dirname);
console.log(__filename);

//REPL环境执行timer.js文件结果
wendingding$ node timer.js
/Users/文顶顶/Desktop/fs
/Users/文顶顶/Desktop/fs/timer.js
```

- Posted by 博客园·文顶顶 | 花田半亩
- 联系作者 简书·文顶顶 新浪微博·Coder_文顶顶
-