七天学会 NodeJS #1

什么是 NodeJS

JS 是脚本语言，脚本语言都需要一个解析器才能运行。对于写在 HTML 页面里的 JS，浏览器充当了解析器的角色。而对于需要独立运行的 JS，NodeJS 就是一个解析器。

每一种解析器都是一个运行环境，不但允许 JS 定义各种数据结构，进行各种计算，还允许 JS 使用运行环境提供的内置对象和方法做一些事情。例如运行在浏览器中的 JS 的用途是操作 DOM，浏览器就提供了 document 之类的内置对象。而运行在 NodeJS 中的 JS 的用途是操作磁盘文件或搭建 HTTP 服务器，NodeJS 就相应提供了 fs、http 等内置对象。

有啥用处

尽管存在一听说可以直接运行 JS 文件就觉得很酷的同学，但大多数同学在接触新东西时首先关心的是有啥用处，以及能带来啥价值。

NodeJS 的作者说，他创造 NodeJS 的目的是为了实现高性能 Web 服务器，他首先看重的 是事件机制和异步 IO 模型的优越性，而不是 JS。但是他需要选择一种编程语言实现他的想法，这种编程语言不能自带 IO 功能，并且需要能良好支持事件机制。 JS 没有自带 IO 功能，天生就用于处理浏览器中的 DOM 事件，并且拥有一大群程序员，因此就成为了天然的选择。

如他所愿，NodeJS 在服务端活跃起来，出现了大批基于 NodeJS 的 Web 服务。而另一方面，NodeJS 让前端众如获神器，终于可以让自己的能力覆盖范围跳出浏览器窗口，更大批的前端工具如雨后春笋。

因此，对于前端而言，虽然不是人人都要拿 NodeJS 写一个服务器程序，但简单可至使用命令交互模式调试 JS 代码片段，复杂可至编写工具提升工作效率。

NodeJS 生态圈正欣欣向荣。

如何安装

安装程序

NodeJS 提供了一些安装程序，都可以在 nodejs.org 这里下载并安装。

Windows 系统下，选择和系统版本匹配的.msi 后缀的安装文件。Mac OS X 系统下，选择.pkg 后缀的安装文件。

编译安装

Linux 系统下没有现成的安装程序可用，虽然一些发行版可以使用 apt-get 之类的方式安装，但不一定能安装到最新版。因此 Linux 系统下一般使用以下方式编译方式安装 NodeJS。

确保系统下 g++版本在4.6以上，python 版本在2.6以上。

从 nodejs.org 下载 tar.gz 后缀的 NodeJS 最新版源代码包并解压到某个位置。

进入解压到的目录，使用以下命令编译和安装。

```
$ ./configure

 $ make

 $ sudo make install
```

如何运行

打开终端，键入 node 进入命令交互模式，可以输入一条代码语句后立即执行并显示结果，例如：

```
$ node

> console.log('Hello World!');

Hello World!
```

如果要运行一大段代码的话，可以先写一个 JS 文件再运行。例如有以下 hello.js。

```
function hello() {

    console.log('Hello World!');

}

hello();
```

写好后在终端下键入 node hello.js 运行，结果如下：

```
$ node hello.js

Hello World!
```

权限问题

在 Linux 系统下，使用 NodeJS 监听80或443端口提供 HTTP(S)服务时需要 root 权限，有两种方式可以做到。

一种方式是使用 sudo 命令运行 NodeJS。例如通过以下命令运行的 server.js 中有权限使用80和443端口。一般推荐这种方式，可以保证仅为有需要的 JS 脚本提供 root 权限。

```
$ sudo node server.js
```

另一种方式是使用 chmod +s 命令让 NodeJS 总是以 root 权限运行，具体做法如下。因为这

种方式让任何 JS 脚本都有了 root 权限，不太安全，因此在需要很考虑安全的系统下不推荐使用。

```
$ sudo chown root /usr/local/bin/node

$ sudo chmod +s /usr/local/bin/node
```

模块

编写稍大一点的程序时一般都会将代码模块化。在 NodeJS 中，一般将代码合理拆分到不同的 JS 文件中，每一个文件就是一个模块，而文件路径就是模块名。

在编写每个模块时，都有 require、exports、module 三个预先定义好的变量可供使用。

require

require 函数用于在当前模块中加载和使用别的模块，传入一个模块名，返回一个模块导出对象。模块名可使用相对路径（以./开头），或者是绝对路径（以/或 C:之类的盘符开头）。另外，模块名中的.js 扩展名可以省略。以下是一个例子。

```
var foo1 = require('./foo');

var foo2 = require('./foo.js');

var foo3 = require('/home/user/foo');

var foo4 = require('/home/user/foo.js');


// foo1至 foo4中保存的是同一个模块的导出对象。
```

另外，可以使用以下方式加载和使用一个 JSON 文件，模块名中.json 扩展名不可省略。

```
var data = require('./data.json');
```

exports

exports 对象是当前模块的导出对象，用于导出模块公有方法和属性。别的模块通过 require 函数使用当前模块时得到的就是当前模块的 exports 对象。以下例子中导出了一个公有方法。

```
exports.hello = function () {

    console.log('Hello World!');

};
```

module

通过 module 对象可以访问到当前模块的一些相关信息，但最多的用途是替换当前模块的导出对象。例如模块导出对象默认是一个普通对象，如果想改成一个函数的话，可以使用以下方式。

```
module.exports = function () {
    console.log('Hello World!');
};
```

以上代码中，模块默认导出对象被替换为一个函数。

模块初始化

一个模块中的 JS 代码仅在模块第一次被使用时执行一次，并在执行过程中初始化模块的导出对象。之后，缓存起来的导出对象被重复利用。

主模块

通过命令行参数传递给 NodeJS 以启动程序的模块被称为主模块。主模块负责调度组成整个程序的其它模块完成工作。例如通过以下命令启动程序时，main.js 就是主模块。

```
$ node main.js
```

完整示例

例如有以下目录。

```
- /home/user/hello/
    - util/
        counter.js
    main.js
```

其中 counter.js 内容如下：

```
var i = 0;

function count() {
    return ++i;
}
```

```
exports.count = count;
```

该模块内部定义了一个私有变量 i，并在 exports 对象导出了一个公有方法 count。

主模块 main.js 内容如下：

```
var counter1 = require('./util/counter');

var counter2 = require('./util/counter');


console.log(counter1.count());

console.log(counter2.count());

console.log(counter2.count());
```

运行该程序的结果如下：

```
$ node main.js

1

2

3
```

可以看到，counter.js 并没有因为被 require 了两次而初始化两次。

二进制模块

虽然一般我们使用 JS 编写模块，但 NodeJS 也支持使用 C/C++编写二进制模块。编译好的二进制模块除了文件扩展名是.node 外，和 JS 模块的使用方式相同。虽然二进制模块能使用操作系统提供的所有功能，拥有无限的潜能，但对于前端同学而言编写过于困难，并且难以跨平台使用，因此不在本教程的覆盖范围内。

小结

本章介绍了有关 NodeJS 的基本概念和使用方法，总结起来有以下知识点：

NodeJS 是一个 JS 脚本解析器，任何操作系统下安装 NodeJS 本质上做的事情都是把 NodeJS 执行程序复制到一个目录，然后保证这个目录在系统 PATH 环境变量下，以便终端下可以使用 node 命令。

终端下直接输入 node 命令可进入命令交互模式，很适合用来测试一些 JS 代码片段，比如正则表达式。

NodeJS 使用 **CMD** 模块系统，主模块作为程序入口点，所有模块在执行过程中只初始化一次。

除非 **JS** 模块不能满足需求，否则不要轻易使用二进制模块，否则你的用户会叫苦连天。