

Node基础入门



2016年 10月

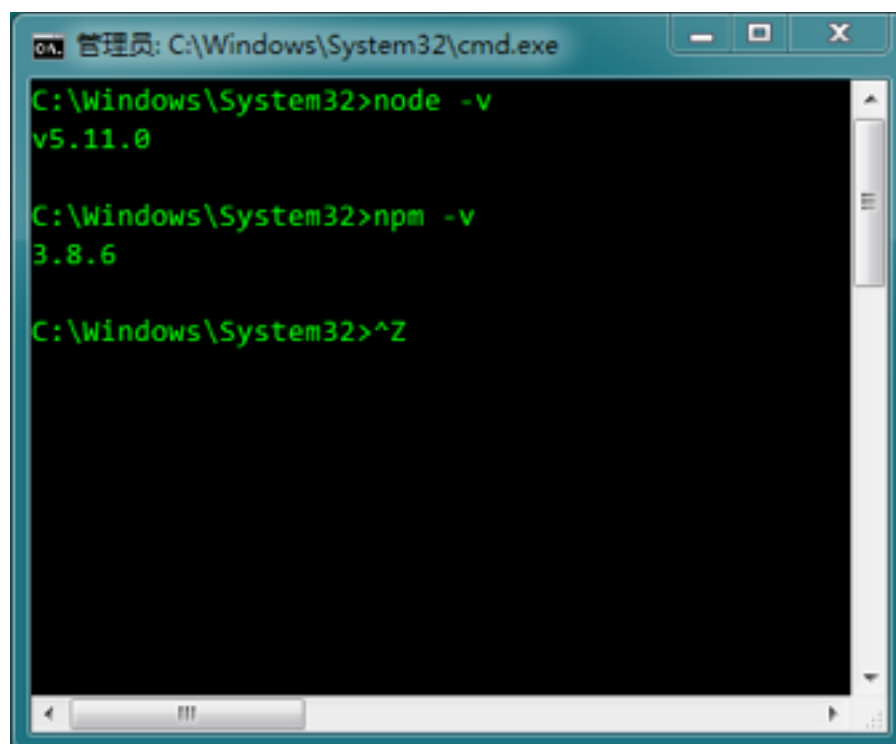
I. 环境搭建

Windows平台

1. 下载地址为<https://nodejs.org/en/download/>，如下图所示



2. 根据自己电脑分属的32/64位系统，选择对应的下载版本，这里以64-bit为例
3. 双击下载的得到的.msi文件，按照提示全部采用默认值，一路next，最后finish
4. 安装完成后，打开CMD，输入node -v和npm -v，如果出现版本号信息，则表示安装成功，入下图所示

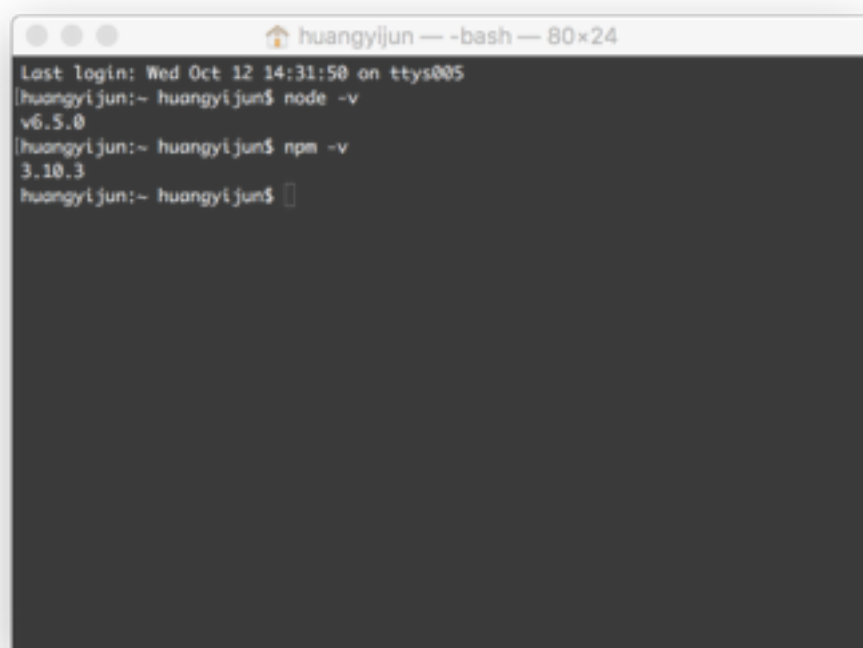


MAC平台

1. 下载地址为<https://nodejs.org/en/download/>，如下图所示



2. 选择macOS Installer（.pkg），点击下载
3. 双击下载得到的.pkg文件，一路点击“继续”，此步骤有可能需要输入sudo密码
4. 安装完成后，打开MAC的Terminal终端，输入node -v和npm -v，如果出现版本信息，则表示安装成功，如下图所示



II. 如何执行js

将Node环境搭建完毕后，我们就可以使用Node来执行一些js代码了

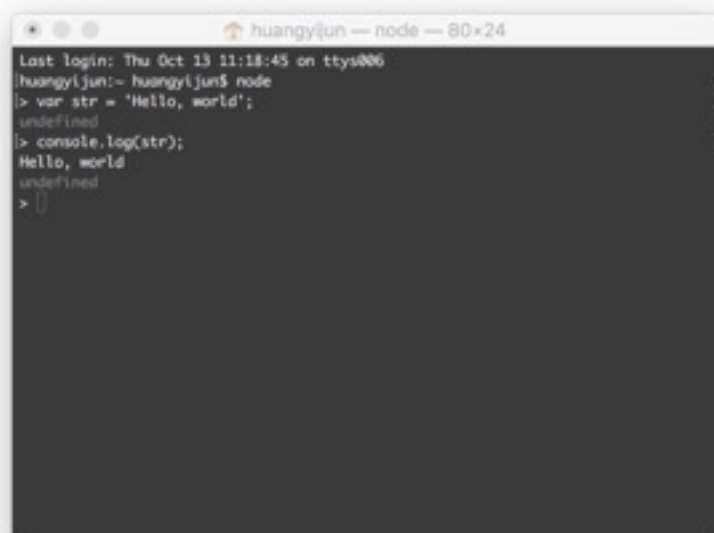
下面是两种Node执行JS的方式，让我们开启HelloWorld之旅

使用Node控制台执行JS

1. Windows下打开CMD，Mac下打开Terminal。输入node命令，按回车，如图：



2. 输入：var str = 'Hello, world'，回车
3. 输入：console.log(str)，回车，此时控制台会打印出Hello, world，如图：

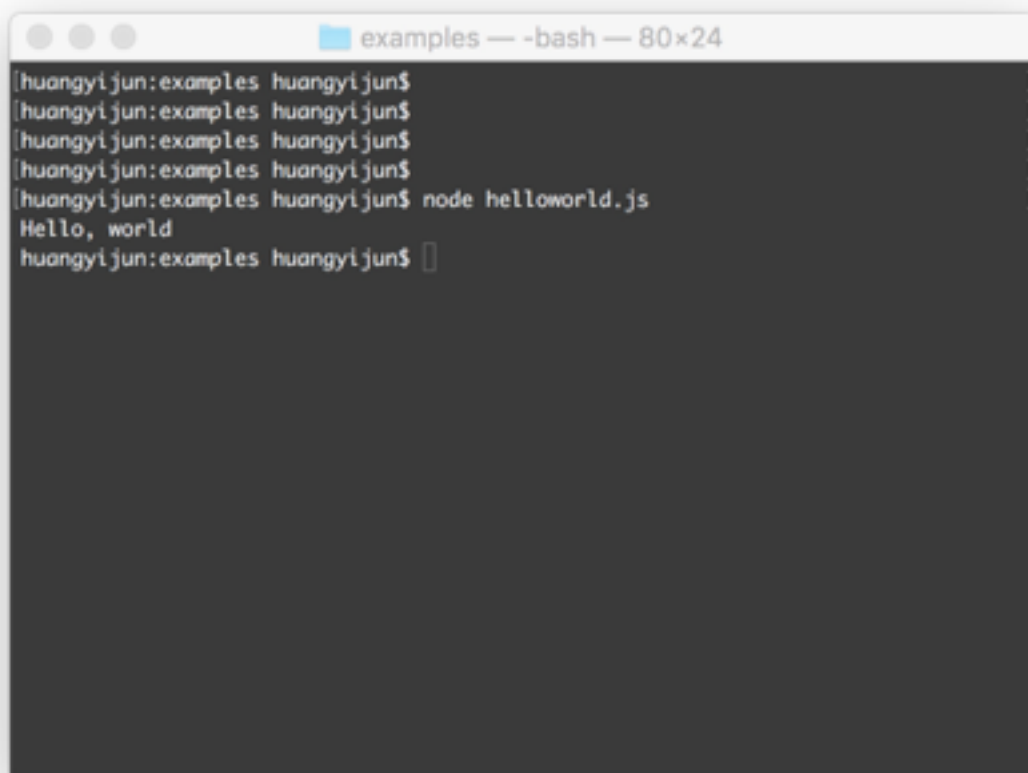


注意：图中的每一个undefined，是表示每输入一行后的返回值，这里的变量定义和控制台输出两句话显然没有返回值，所以会各显示一个undefined。

使用Node直接执行JS文件

1. 打开你最喜欢的编辑器，创建一个helloworld.js文件，文件中的内容如下：

```
var str = 'Hello, world';  
console.log(str);
```
2. Windows下打开CMD，Mac下打开Terminal，使用cd命令将当前盘符切换到文件helloworld.js所在的目录
3. 执行命令：node helloworld.js，可以看到控制台输出：Hello, world。如图

A screenshot of a terminal window titled 'examples — -bash — 80x24'. The terminal shows a series of commands and their outputs. The first four lines are empty prompts: 'huangyijun:examples huangyijun\$'. The fifth line shows the command 'node helloworld.js' being executed, followed by the output 'Hello, world'. The sixth line shows the prompt 'huangyijun:examples huangyijun\$' with a cursor. The terminal has a dark background and light-colored text.

```
huangyijun:examples huangyijun$  
huangyijun:examples huangyijun$  
huangyijun:examples huangyijun$  
huangyijun:examples huangyijun$  
huangyijun:examples huangyijun$ node helloworld.js  
Hello, world  
huangyijun:examples huangyijun$
```

III. 模块机制

I. 为什么需要引入模块机制

根据上一节中的说明，显然第二种使用`node xxx.js`的方式是我们来编写Node应用的主流方式，我们可以将逻辑js部分编写到一个以js为后缀的文件中，然后直接执行即可。似乎我们已经可以开始动手编写一个完整的Node应用了，那为什么还要来了解什么模块引入机制呢。

针对这个问题，我们可以思考下面的场景：

1. 我们项目由一个比较复杂主的逻辑构成，我们称之为Logic Total
2. Logic Total又由三个次一级逻辑构成，我们称之为LM1，LM2，LM3
3. 每一个上述的次一级逻辑又分别由更次一级的5个逻辑LN1, LN2, ..., LN5构成
- ...

好了，仅仅到第三步，我们就会发现，如果没有模块引入机制，我们将会把组成Logic Total的所有的逻辑代码都写到这一个js文件中，这些代码包含次一级的LM1，LM2，LM3，以及更次一级的LM1-LN1，LM2-LN1，LM3-LN1，..., LM1-LN5，LM2-LN5，LM3-LN...

此时，我们仅仅靠一个js文件，已经无法对整个项目进行很好的管理了。所有本项目后续功能的添加和维护更新已经变成了意大利面条。所以，这时候，引入模块机制就变成我们的首要任务。

II. CommonJS规范

正式为了解决上述问题，出现了CommonJS (<http://www.commonjs.org>) 规范，这个规范的目标正是为了构建JavaScript在包括Web服务器，桌面，命令行工具，及浏览器方面的生态系统。

CommonJS制定了解决这些问题的一些规范，然而光有规范还不行，Node在底层提供了该规范的一种实现方式。具体就是Node自身实现了`require`方法作为其引入模块的方法，同时NPM也基于CommonJS定义的包规范，实现了依赖管理和模块自动安装等功能，目前前端繁荣的Npm生态圈正是依托在这样的基础上。下面我们就来看看对于开发者来说，如何编写一个模块以及引入一个模块。

III. 模块的编写和导出

Node中的模块，非常容易编写，模块本身也是一个js文件，书写方法和普通的js文件一样。唯一不同的是我们要使用exports和module.exports两个关键字，将我们需要导出的变量或者方法导出来给别的js文件调用。

下面我们来以圆周长的计算为例，将如下代码保存为circular.js：

```
const PI = 3.1415926;

function area(r) {
    return PI * r * r;
}

exports.area = area;

function circumference(r) {
    return 2 * PI * r;
}

exports.circumference = circumference;
```

这样我们便得到了一个提供计算圆周长和圆面积的公共JS模块，该模块提供了：

1. 计算面积的方法：area函数，需要传入半径r
2. 计算周长的方法：circumference函数，需要传入半径r

很简单吧，下面我们来看下如何使用该模块。

IV. 模块的引用

使用III中JS模块，也是非常简单的。在circular.js同一级目录下，创建新的js文件，名称为testCircular.js，内容如下：

```
const circular = require('./circular.js');
const R = 2;
```

```
console.log(`This circular's area is: ${circular.area(R)}, and circumference is: $
{circular.circumference(R)}`);
```

文件编写完成后，使用node命令执行该testCircular.js文件，看到控制台输出：

```
This circular's area is: 12.5663704, and circumference is: 12.5663704
```

这样，就完成了一次模块的引用。

V. module.exports和exports

好了，经过III和IV两节，我们已经明白了在Node中如何编写和引入一个模块，但是在III节的开头，我们提到了可以使用module.exports和exports两个关键字导出模块，那么这两种方式有什么区别呢。

简单的说，一个JS模块，比如上述的circular.js，其真正导出的内容为module.exports部分，所有的exports.NAME = VALUE方法，最后都会属于module.exports对象中的一个元素，并且该元素的key就是NAME，value就是VALUE。

并且，如果我们想将整个模块导出一个固定的属性，比如整个模块导出一个函数，只能使用：

```
module.exports = function(){};
```

的方式来实现。

这样的原因，就是exports其实是module.exports对象的一个引用，当我们使用

```
exports.NAME = VALUE
```

时，module.exports对象内容同步发生改变

而当我们使用

```
exports = VALUE
```

时，我们仅仅是改变了exports变量的引用，并没有改变module.exports的内容。想要更详细的了解，可以参看《高阶之路——深入Node底层》中的《模块加载源码详解》一文，里面从node底层提供的module.js源代码角度，深入解析了Node实现CommonJS规范的方式。从中，我们可以更详细的了解到Node的模块载入策略。

VI. 第三方模块的安装和引用

上面5小节主要讲解了自己编写的模块的导出和引用。但是一个成熟的项目，肯定会使用到一些第三方的包，那么如何使用第三方开源包呢。

Node中很贴心的提供了Npm模块，该模块会随着用户的Node安装一起安装到OS中，Npm的使用方法也是相当简洁：

```
npm install 包名
```

即可，安装后，会在当前目录生成一个node_modules目录，这时候就可以在项目中，直接使用

```
require(包名)
```

的方式使用第三方包了。项目路径中如果有package.json文件时，直接使用npm install方法就可以根据package.json中的dependencies配置安装所有的依赖包，这样可以把node_modules这个文件目录从版本库中干掉。

几条常用的npm命令：

1. npm install <name> —save，安装npm包并且添加依赖到当前目录的package.json中
2. npm remove <name> —save，移除npm包并且从当前目录的package.json中移除依赖

有了Npm，我们在开发Node项目时就不再是一个人在战斗了，npmjs.org上有大量的功能成熟的开源npm可供我们直接拿来使用。整个项目的可靠度也大大提升。

IV. 构造Web应用

I. 创建Http服务器

通过上面三节的学习，我们已经知道了使用Node编写一个项目的基本模式，本节，我们就来以一个web应用为例子，学习如何使用Node搭建web应用。

首先，我们显然是需要编写一个Http服务器。在Node中，编写http服务器相当的简单，原因就是Node底层将启动一个Http Server所需的所有复杂操作都帮我们封装好了，暴露给开发者的API非常简洁，我们可以创建一个server.js文件，内容如下：

```
const http = require("http");
```

```
http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write(`Hello World, From Path: ${request.url} and Method: ${request.method}`);
  response.end();
}).listen(8888, ()=>console.log(`Http Server start at 8888...`));
```

完成后，使用node命令启动该js文件，可以看到控制打印如下信息：

```
Http Server start at 8888...
```

此时，再打开浏览器，访问：http://127.0.0.1:8888，页面显示：

```
Hello World, From Path: / and Method: GET
```

访问：http://127.0.0.1:8888/start，页面显示：

```
Hello World, From Path: /start and Method: GET
```

可以看到，页面返回的From Path后面的路径和我们访问的路径保持一致，而Method则一直都是GET。

下一节我们来分析下这段代码，从中理解如何编写一个完整的web应用。

II. 侦听端口

I节中的代码，我们将createServer后面的函数去掉，简化为：

```
http.createServer().listen(8888, ()=>console.log(`Http Server start at 8888...`));
```

其中，listen方法的第一个入参8888显然就是我们启动的Http Server侦听的端口号。

第二个入参则是一个回调函数（ES6中的箭头函数，等价于function(){console.log(`Http Server start at 8888...`)}），本文为了简写方便单行回调函数均采用箭头函数形式），这个回调函数，会在http服务器启动完毕后再调用，作用就是打印出启动成功的日志。

III.Http请求处理函数

II中还遗留一个createServer中的匿名函数没有解释，本节来着重讲解下这个函数。

```
function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write(`Hello World, From Path: ${request.url} and Method: ${request.method}`);
  response.end();
}
```

该匿名函数是在服务器启动后，每次有http请求连接到服务器时，才会被触发的函数，通俗地讲，就是每一个http请求真正的处理函数。

这个函数的入参第一个request，为Http请求信息句柄；入参第二个response，为Http响应信息句柄。

通俗的说，request中包含了本次Http请求客户端的一些信息，比如我们可以使用request.url获取到本次Http请求的路径，使用request.method可以获取到本次Http请求的方法。这里就解释了I中页面From后面的路径总是和我们浏览器中访问路径保持一致的原因。

response作为响应信息句柄，提供了write方法给我们向缓冲区写数据，write方法可以多次写，最后调用end方法时再返回给客户端。也可以将要返回给客户端的数据直接作为end方法的参数直接返回。这里我们是采用write方法写数据再调用end方法发送的流程。response.writeHead则是添加Http头信息，这些格式都是由Http协议本身规定的，想要详细了解可以看下Http权威指南。

IV.路由的编写

I中搭建的Http服务器，虽然已经提供了简单的页面访问输出，但是要使用到真正的项目中，这是远远不够的。因为这个Http服务器没有一个Web应用中最重要路由模

块，即将客户端发出的不同的Http请求映射到不同的处理函数中进行数据处理，然后再将处理的结果以HTML或者JSON数据等形式返回给客户端。

而从III的讲解中，我们可以获取到用户每一次请求的路径以及对应的Http方法，有了这两个数据，结合上一节中讲解的Node模块编写和引用，自己构造一个Web应用中最重要的路由模块就变得顺理成章了。

编写如下代码，保存到router.js中：

```
function index(req, res) {
  res.writeHead(200, {"Content-Type": "text/html"});
  res.write(`<h1>Hello World, This is index Page</h1>`);
  res.end();
}
```

```
function hello(req, res) {
  res.writeHead(200, {"Content-Type": "text/html"});
  res.write(`<h1>Hello World</h1>`);
  res.end();
}
```

```
function start(req, res) {
  res.writeHead(200, {"Content-Type": "text/html"});
  res.write(`<h1>Your Node Start From Here</h1>`);
  res.end();
}
```

```
function error(req, res, code) {
  res.writeHead(code, {"Content-Type": "text/plain"});
  res.write(`Can't ${req.method} ${req.url}`);
  res.end();
}
```

```
let Router = {
  GET: {
    '/': index,
```

```

    '/hello': hello,
    '/start': start
  },
  POST: {}
};

function router(req, res) {
  let path = req.url;
  let method = req.method;

  let methodMatch = Router[method.toUpperCase()];
  if (methodMatch) {
    let pathMatch = methodMatch[path];
    if (pathMatch) {
      pathMatch(req, res);
    } else {
      error(req, res, 404)
    }
  } else {
    error(req, res, 403)
  }
}

module.exports = router;

```

上面的内容里，我们提供了start，index和hello三个路由处理函数，分别对应的路由路径是/start，/和/hello；另外还提供了一个处理路由匹配不到的情况的函数error。

完成这个文件的编写后，我们将server.js中的代码修改为：

```

const http = require("http");
const router = require('./router.js')
http.createServer(router).listen(8888, ()=>console.log(`Http Server start at 8888...`));

```

同样使用node启动server.js后

访问：http://127.0.0.1:8888/，页面输出：

Hello World, This is index Page

访问: `http://127.0.0.1:8888/start`, 页面输出:

Your Node Start From Here

访问: `http://127.0.0.1:8888/hello`, 页面输出:

Hello World

访问: `http://127.0.0.1:8888/quick`, 页面输出:

Can't GET /quick

可以看到, 这样的一个简单的路由模块, 提供了Path映射和简单的映射失败错误处理。如果有更复杂的业务开发, 就是我们扩展路由函数文件的一个过程。

经过本章学习, 我们基本可以使用Node搭建一个包含路由模块的Http服务器, 如果你的应用不涉及到View层, 而是单纯的API接口, 几乎这样简单的router模块已经够用。

这样的服务器编写完成后, 我们使用`node server.js`可以保持其运行, 但是此时我们并没有服务器异常退出时的重启恢复机制, 以及后台运行机制。下一节, 我们将介绍下

Node中的进程守护工具: PM2

V. 守护工具PM2

I. 为什么需要PM2

经过上面四节内容，我们对Node开发应该已经具备了一个比较感性的认识。有了之前的知识，我们也可以使用Node进行一些业务开发了，但是这还不够。因为对于一个生产环境的应用来说，起码要具备两个要素：

1. 能够在后台运行，即启动命令执行后，关闭shell，应用依旧能够运行
2. 进程出现异常时，有自动重启恢复机制

前面四节，我们都是在介绍直接使用node启动进程的方式，这种方式显然是不能满足上面两个要求的。

这时候，我们在生产部署时，就需要用到Node的守护工具——PM2。

II. PM2的安装

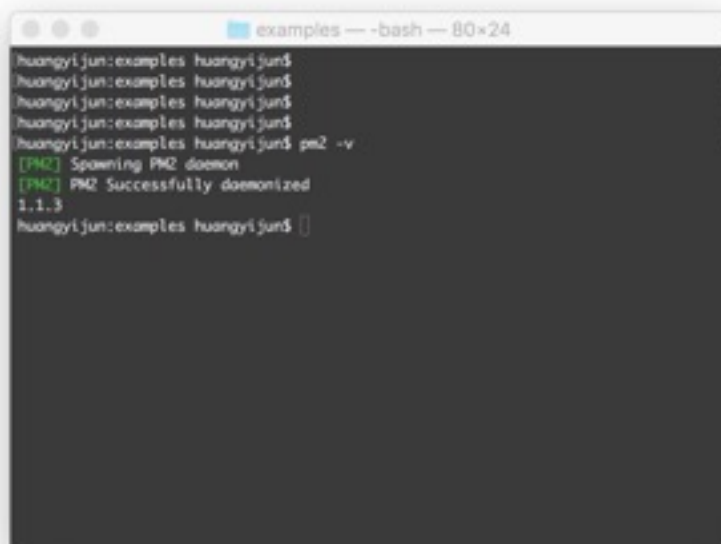
由于PM2是一个全局命令，所以我们要使用全局安装的方式，具体如下：

1. Windows下打开CMD，Mac下打开Terminal终端
2. Windows下输入：npm install pm2 -g
3. Mac下输入：sudo npm install pm2 -g

即可将PM2工具安装到全局。要检测是否安装成功，步骤如下：

1. Windows下打开CMD，Mac下打开Terminal终端
2. 输入：pm2 -v

如果得到版本信息，则表示PM2安装成功：

A terminal window titled 'examples -- bash -- 80x24' showing the installation and verification of PM2. The user runs 'pm2 -v' and receives the output: '[PM2] Spawning PM2 daemon', '[PM2] PM2 Successfully daemonized', and '1.1.3'.

```
examples -- bash -- 80x24
huangyijun:examples huangyijun$
huangyijun:examples huangyijun$
huangyijun:examples huangyijun$
huangyijun:examples huangyijun$
huangyijun:examples huangyijun$ pm2 -v
[PM2] Spawning PM2 daemon
[PM2] PM2 Successfully daemonized
1.1.3
huangyijun:examples huangyijun$
```

III.PM2的使用

一. 启动

以上一节的中搭建的Http Server为例，使用pm2启动server.js，方法如下：

1.使用cd命令进入到server.js的当前目录

2.pm2 start server.js

此时server.js已经由PM2在后台启动了。但是这时候其实PM2只启动了一个server进程，在多核机器上，单进程无法有效利用多核CPU，所以我们可以启动多个子进程来提高服务器的性能，以四核CPU机器为例，方法如下：

1.使用cd命令进入到server.js的当前目录

2.pm2 start server.js -i 4

其中，-i 4表示启动4个子进程，子进程数量一般来说和CPU逻辑核数保持一致，能达到最大的性能利用。

另外，我们可以给进程设置别名，用作后续管理用，方法如下：

```
pm2 start server.js -i 4 --name httpServer
```

这样，我们就给server.js这个四个子进程设置了别名：httpServer

二. 重启

启动后，如果进程需要重启，方法如下：

```
pm2 restart <processName | |processID | |all>
```

其中，processName为子进程的名字，processID为子进程在PM2中分配到的ID，all表示重启全部子进程，三个参数选填一个即可。

例如，上述的子进程别名被设置为httpServer，我们可以按照如下方式重启：

```
pm2 restart httpServer
```

另外，PM2还提供了专门给WEB页面应用的0秒宕机重载功能，方法如下：

```
pm2 reload <processName | |processID | |all>
```

参数含义和restart一致，这里简单讲述下restart和reload的不同：

1.restart执行后子进程会按照ID顺序依次立即被杀死，并且杀死后立即重启

2.reload执行后，第一个子进程被完全杀死并且重启成功后，才会杀死第二个子进程并且等到第二个子进程重启成功后，才会杀死第三个子进程...

这样的效果就是，当我们的server.js启动四个子进程实例后，restart方式的重启，在生产环境可能会造成500错误（四个子进程都被杀死，且重启的过程中有客户端请求到达）；而采用reload方法后，生产环境永远都会有起码一个子进程在线提供服务给用户，所以对于WEB页面应用来说，近似于0秒宕机的重载方式。

三. 停止

最后就是进程的停止，我们可以用如下命令来停止子进程运行：

```
pm2 stop <processName || processID || all>
```

参数含义和二节中描述的一致，stop后的子进程，会在PM2列表中保留，需要启动只需要再次执行：

```
pm2 start <processName || processID || all>
```

另外，PM2还提供了删除子进程并且在PM2列表中不保留的方式：

```
pm2 delete <processName || processID || all>
```

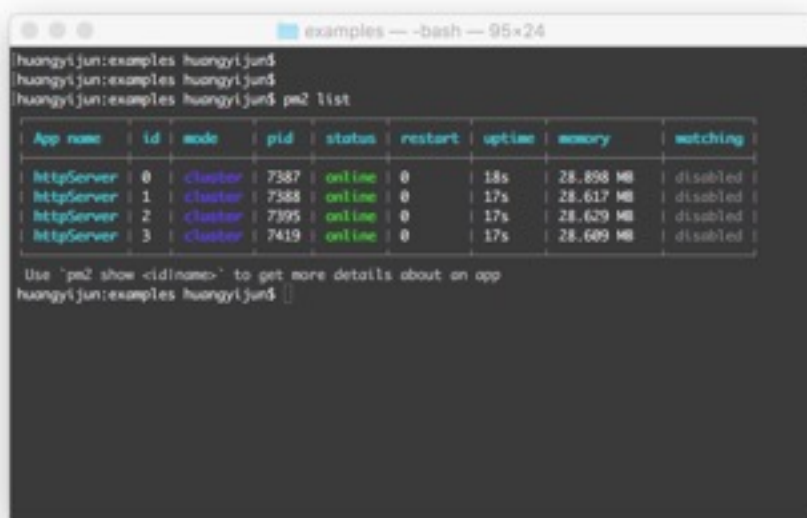
参数含义同样，和stop的区别在于会把子进程从PM2列表中删除掉，如果需要再次启动，只能按照一节中的方法进入到js文件目录，再使用pm2 star fileName的方式启动。

四. 状态

查看当前pm2启动的子进程状态方法：

```
pm2 list
```

显示如下：



```
examples -- bash -- 95x24
huangyijun:examples huangyijun$
huangyijun:examples huangyijun$
huangyijun:examples huangyijun$ pm2 list
```

App name	id	mode	pid	status	restart	uptime	memory	watching
httpServer	0	cluster	7387	online	0	18s	28.898 MB	disabled
httpServer	1	cluster	7388	online	0	17s	28.617 MB	disabled
httpServer	2	cluster	7395	online	0	17s	28.629 MB	disabled
httpServer	3	cluster	7419	online	0	17s	28.609 MB	disabled

```
Use 'pm2 show <id|name>' to get more details about an app
huangyijun:examples huangyijun$
```

我们来解释下相关字段：

1. App name：子进程的别名，如果启动时按照一节中带了—name参数，则会显示用户设置的，如果没有带该参数，则以启动的文件名为默认name
- 2.id：子进程在PM2中分配到的ID，按照0，1，2...顺序递增
- 3.mode：cluster模式还是fork模式的方式启动的子进程
- 4.pid：每一个子进程在系统中的进程ID
- 5.status：online表示运行正常，errored表示子进程已经异常退出
- 6.restart：表示重启次数（restart和reload都算是重启）
- 7.uptime：表示子进程从启动到现在运行的时间
- 8.memory：表示子进程占用的系统内存大小（堆内+堆外内存）
- 9.watching：表示是否开启文件变化监控，开启后文件变更后自动重启

五. 其余实用命令

以上四小节讲解了PM2最常用的几个命令，本小节来介绍下其余的一些比较常用的命令给大家：

- 1.pm2 monit：在控制台实时监控子进程的内存和CPU开销
- 2.pm2 logs：在控制台实时观看子进程日志信息
- 3.pm2 start server.js -l /opt/logs/server.log：将日志信息输出到/opt/logs/server.log文件中，这里的日志文件
- 4.pm2 start server.js -e /opt/logs/server-error.log -o /opt/logs/server-out.log：将错误日志信息输出到/opt/logs/server-error.log文件中，将stdout日志信息输出到/opt/logs/server-out.log中，即将错误日志和普通的日志信息分开到两个文件中
- 5.pm2 start helloworld.js --log-date-format="YYYY-MM-DD HH:mm:ss.SSS"：日志前面加上格式化后的事件前缀，格式化内容按照log-date-format参数的值来定