NodeJS 备忘

IO input/output 输入输出

JavaScript 是由ECMAScript 文档对象模型（DOM）和浏览器对象模型（BOM）组成

Node.js 中所谓的JavaScript 只是Core JavaScript，或者ECMAScript
不包含DOM、BOM 或者Client JavaScript。
因为Node.js 不运行在浏览器中 不需要使用浏览器中的许多特性

Node.js 内建了**HTTP** 服务器支持

Node.js 只是在事件队列中增加请求，等待操作系统的回应

采用异步式**I/O** 与事件驱动的架构设计

CommonJS

只需要访问http://nodejs.org

Next即可自动完成安装。

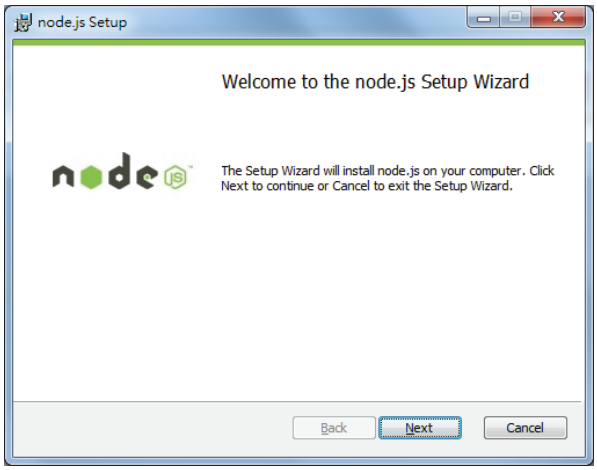


图2-1 在 Windows 上安装 Node.js

安装程序不会询问你安装路径，Node.js 会被自动安装到 C:\Program Files\nodejs 或 C:\Program Files (x86)\nodejs（64位系统）目录下，并且会在系统的 PATH 环境变量中增加该目录，因此我们可以在 Windows 的命令提示符中直接运行 `node`。

为了测试是否已经安装成功，我们在运行中输入 `cmd`，打开命令提示符，然后输入 `node`，将会进入 Node.js 的交互模式，如图2-2所示。

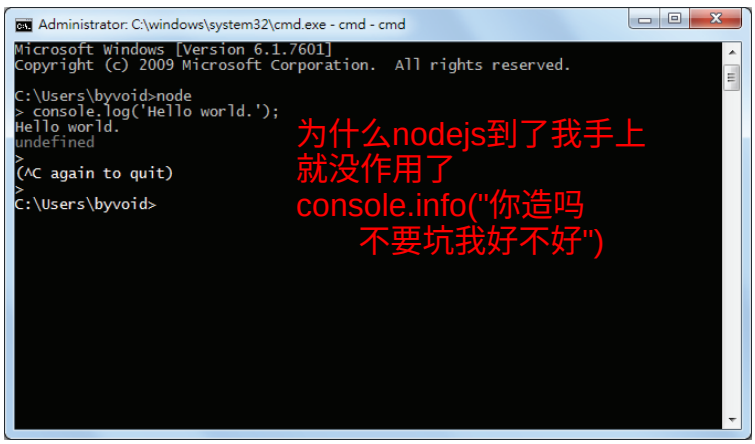


图2-2 Windows 命令提示符下的 Node.js

通过这种方式安装的 Node.js 还自动附带了 npm 图2-2，我们可以在命令提示符中直接输入 npm 来使用它。

实现第一个Node.js 程序吧

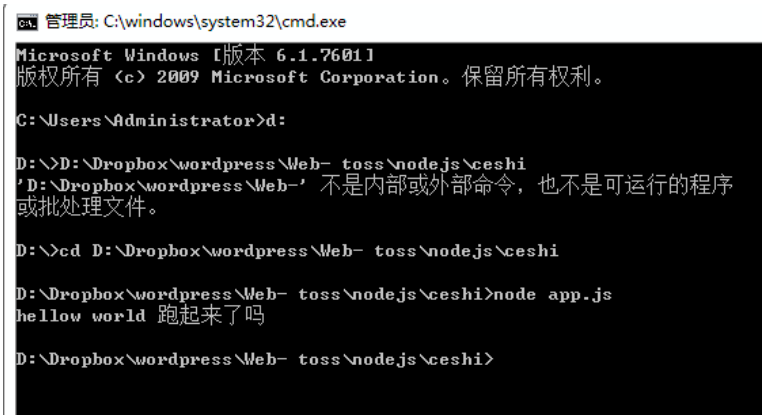
新建立 app.js 文件　console.log("hellow word")

cmd 杠到目录里面　node app.js 就运行了

console 是Node.js 提供的控制台对象，其中包含了向标准输出写入的操作，如console.log、console.error 等

console.info("...")

这个是用不了的 我习惯用这个 被坑了 还以为node没有成功安装好



3.1.2 Node.js 命令行工具

如果不是 cmd里输入
C:\Program Files\nodejs 如果是这里 就是 .help

在前面的 Hello World 示例中，我们用到了命令行中的 node 命令，输入 node --help 可以看到详细的帮助信息：

```
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]

Options:
  -v, --version        print node's version
  -e, --eval script    evaluate script
  -p, --print          print result of --eval
  --v8-options         print v8 command line options
  --vars               print various compiled-in variables
  --max-stack-size=val set max v8 stack size (bytes)

Environment variables:
NODE_PATH              ';'-separated list of directories
                       prefixed to the module search path.
NODE_MODULE_CONTEXTS   Set to 1 to load modules in their own
                       global contexts.
NODE_DISABLE_COLORS    Set to 1 to disable colors in the REPL

Documentation can be found at http://nodejs.org/
```

其中显示了 node 的用法，运行 Node.js 程序的基本方法就是执行 node *script.js*，其中 *script.js*[①]是脚本的文件名。

除了直接运行脚本文件外，node --help 显示的使用方法中说明了另一种输出 Hello World 的方式：

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

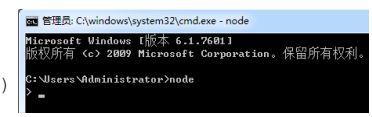
我们可以把要执行的语句作为 node -e 的参数直接执行。

使用 node 的 REPL 模式

REPL（Read-eval-print loop），即输入—求值—输出循环。如果你用过 Python，就会知道在终端下运行无参数的 python 命令或者使用 Python IDLE 打开的 shell，可以进入一个即时求值的运行环境。Node.js 也有这样的功能，运行无参数的 node 将会启动一个 JavaScript 的交互式 shell：

① 事实上脚本文件的扩展名不一定是 .js，例如我们将脚本保存为 script.txt，使用 node script.txt 命令同样可以运行。扩展名使用 .js 只是一个约定而已，遵循了 JavaScript 脚本一贯的命名习惯。

```
$ node
> console.log('Hello World');
Hello World
undefined
> consol.log('Hello World');
ReferenceError: consol is not defined
    at repl:1:1
    at REPLServer.eval (repl.js:80:21)
    at repl.js:190:20
    at REPLServer.eval (repl.js:87:5)
    at Interface.<anonymous> (repl.js:182:12)
    at Interface.emit (events.js:67:17)
    at Interface._onLine (readline.js:162:10)
    at Interface._line (readline.js:426:8)
    at Interface._ttyWrite (readline.js:603:14)
    at ReadStream.<anonymous> (readline.js:82:12)
```



进入 REPL 模式以后，会出现一个“>”提示符提示你输入命令，输入后按回车，Node.js 将会解析并执行命令。如果你执行了一个函数，那么 REPL 还会在下面显示这个函数的返回值，上面例子中的 undefined 就是 console.log 的返回值。如果你输入了一个错误的指令，REPL 则会立即显示错误并输出调用栈。在任何时候，连续按两次 Ctrl + C 即可推出 Node.js 的 REPL 模式。

原来是这样子

node 提出的 REPL 在应用开发时会给人带来很大的便利，例如我们可以测试一个包能否正常使用，单独调用应用的某一个模块，执行简单的计算等。

3.1.3　建立 HTTP 服务器

nodejs 建立http fwq 很简单

前面的 Hello World 程序对于你来说可能太简单了，因为这个例子几乎可以在任何语言的教科书上找到对应的内容，既无聊又乏味，让我们来点儿不一样的东西，真正感受一下 Node.js 的魅力所在吧。

Node.js 是为网络而诞生的平台，但又与 ASP、PHP 有很大的不同，究竟不同在哪里呢？如果你有 PHP 开发经验，会知道在成功运行 PHP 之前先要配置一个功能强大而复杂的 HTTP 服务器，譬如 Apache、IIS 或 Nginx，还需要将 PHP 配置为 HTTP 服务器的模块，或者使用 FastCGI 协议调用 PHP 解释器。这种架构是“浏览器 – HTTP 服务器 – PHP 解释器”的组织方式，而Node.js采用了一种不同的组织方式，如图3-1 所示。

我们看到，Node.js 将“HTTP服务器”这一层抽离，直接面向浏览器用户。这种架构从某种意义上来说是颠覆性的，因而会让人心存疑虑：Node.js作为HTTP服务器的效率足够吗？会不会提高耦合程度？我们不打算在这里讨论这种架构的利弊，后面章节会继续说明。

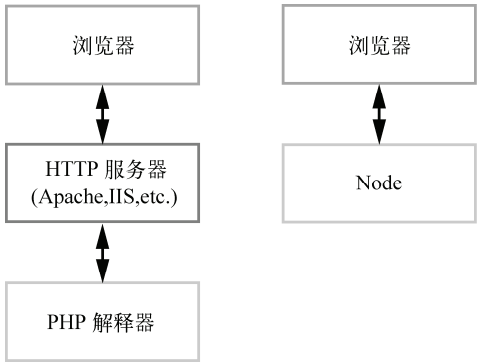


图3-1 Node.js 与 PHP 的架构

好了，回归正题，让我们创建一个 HTTP 服务器吧。建立一个名为 app.js 的文件，内容为：

```
//app.js

var http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<h1>Node.js</h1>');
  res.end('<p>Hello World</p>');
}).listen(3000);
console.log("HTTP server is listening at port 3000.");
```

接下来运行 `node app.js`命令，打开浏览器访问 http://127.0.0.1:3000，即可看到图3-2所示的内容。

我造出来啦

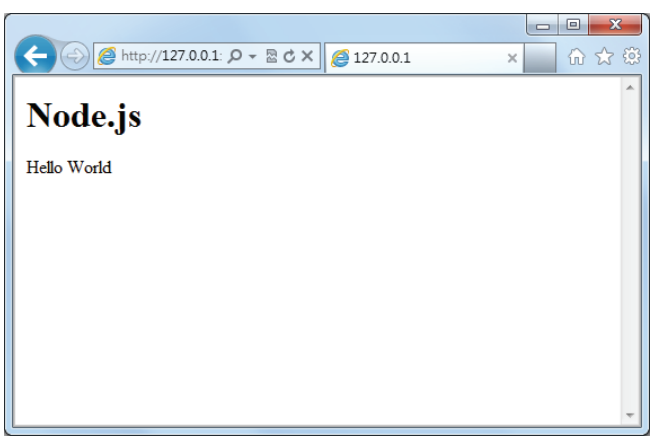


图3-2 用 Node.js 实现的 HTTP 服务器

用 Node.js 实现的最简单的 HTTP 服务器就这样诞生了。这个程序调用了 Node.js 提供的 `http` 模块，对所有 HTTP 请求答复同样的内容并监听 3000 端口。在终端中运行这个脚本时，我们会发现它并不像 Hello World 一样结束后立即退出，而是一直等待，直到按下 Ctrl＋C 才会结束。这是因为 `listen` 函数中创建了事件监听器，使得 Node.js 进程不会退出事件循环。我们会在后面的章节中详细介绍这其中的奥秘。

小技巧——使用 supervisor

如果你有 PHP 开发经验，会习惯在修改 PHP 脚本后直接刷新浏览器以观察结果，而你在开发 Node.js 实现的 HTTP 应用时会发现，无论你修改了代码的哪一部份，都必须终止 Node.js 再重新运行才会奏效。这是因为 Node.js 只有在第一次引用到某部份时才会去解析脚本文件，以后都会直接访问内存，避免重复载入，而 PHP 则总是重新读取并解析脚本（如果没有专门的优化配置）。Node.js的这种设计虽然有利于提高性能，却不利于开发调试，因为我们在开发过程中总是希望修改后立即看到效果，而不是每次都要终止进程并重启。

supervisor 可以帮助你实现这个功能，它会监视你对代码的改动，并自动重启 Node.js。使用方法很简单，首先使用 npm 安装 supervisor：

不用这个

```
$ npm install -g supervisor
```

如果你使用的是 Linux 或 Mac，直接键入上面的命令很可能会有权限错误。原因是 npm 需要把 supervisor 安装到系统目录，需要管理员授权，可以使用 `sudo npm install -g supervisor` 命令来安装。

接下来，使用 `supervisor` 命令启动 app.js：

```
$ supervisor app.js

DEBUG: Running node-supervisor with
DEBUG:   program 'app.js'
DEBUG:   --watch '.'
DEBUG:   --extensions 'node|js'
DEBUG:   --exec 'node'

DEBUG: Starting child process with 'node app.js'
DEBUG: Watching directory '/home/byvoid/.' for changes.
HTTP server is listening at port 3000.
```

我本地电脑怎么是运行不了呢

我遇到 会不停的刷新

貌似是js文件写的不完整

当代码被改动时，运行的脚本会被终止，然后重新启动。在终端中显示的结果如下：

```
DEBUG: crashing child
DEBUG: Starting child process with 'node app.js'
HTTP server is listening at port 3000.
```

supervisor 这个小工具可以解决开发中的调试问题。

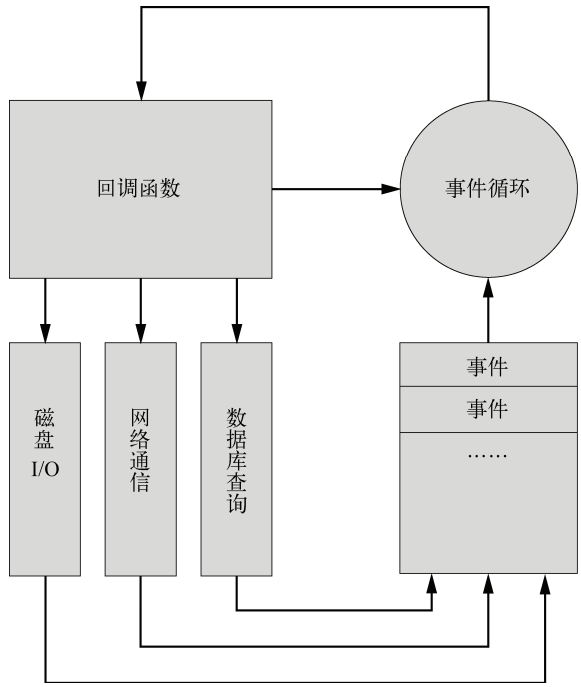


图3-5　事件循环

3.3　模块和包　seajs 也是这样子

模块（Module）和包（Package）是 Node.js 最重要的支柱。开发一个具有一定规模的程序不可能只用一个文件，通常需要把各个功能拆分、封装，然后组合起来，模块正是为了实现这种方式而诞生的。在浏览器 JavaScript 中，脚本模块的拆分和组合通常使用 HTML 的 `script` 标签来实现。Node.js 提供了 `require` 函数来调用其他模块，而且模块都是基于文件的，机制十分简单。

Node.js 的模块和包机制的实现参照了 CommonJS 的标准，但并未完全遵循。不过两者的区别并不大，一般来说你大可不必担心，只有当你试图制作一个除了支持 Node.js 之外还要支持其他平台的模块或包的时候才需要仔细研究。通常，两者没有直接冲突的地方。

我们经常把 Node.js 的模块和包相提并论，因为模块和包是没有本质区别的，两个概念也时常混用。如果要辨析，那么可以把包理解成是实现了某个功能模块的集合，用于发布和维护。对使用者来说，模块和包的区别是透明的，因此经常不作区分。本节中我们会详细介绍：

创建一个 module.js 的文件

```
//module.js
var name;
exports.setName = function(thyName){
    name = thyName;
};
exports.sayHello = function() {
    console.log('Hello ' + name);
};
```

在同一目录下创建getmodule.js

```
var myModule = require('./module');

    myModule.setName('lake');
    myModule.sayHello();
```

没什么特别的和seajs 差不多

创建包

npm来解决包的发布和获取需求

符合CommonJS 规范的包应该具备以下特征：
package.json 必须在包的顶层
目录下；
二进制文件应该在bin 目录下；
JavaScript 代码应该在lib 目录下；
文档应该在doc 目录下；
单元测试应该在test 目录下。

模块与文件是一一对应的　　最简单的包，就是一个作为文件夹的模块

我创建 somepage 的文件夹 里面有 index.js

exports.Hellow 暴漏一个接口

然后我在 somepage 之外建立getPage.js

require("./somePage")

这样子就和 1个模块(包) 关联上了

Node.js 在调用某个包时，会首先检查包中package.json 文件的main 字段，
将其作为包的接口模块，如果package.json 或 main 字段不存在，
会尝试寻找index.js 或index.node 作 为包的接口。

somepage 模块 暴漏的是 index.js 里面的接口
模块中的其他xxx.js 不会暴漏

在 index.js 可以 require('..') 他们

包是在模块基础上更深一步的抽象，

它将某个独立的功能封装起来，用于发布、更新、依赖管理和版本控制。

Node.js 根据CommonJS 规范实现了包机制

开发了npm来解决包的发布和获取需求。

Node.js 的包是一个目录，其中包含一个JSON 格式的包说明文件package.json。

也就是说每个module 里面都有个 package.json

运行 node getpackage.js，控制台将输出结果 `Hello.`。

我们使用这种方法可以把文件夹封装为一个模块，即所谓的包。包通常是一些模块的集合，在模块的基础上提供了更高层的抽象，相当于提供了一些固定接口的函数库。通过定制package.json，我们可以创建更复杂、更完善、更符合规范的包用于发布。

2. package.json

在前面例子中的somepackage文件夹下，我们创建一个叫做package.json的文件，内容如下所示：

```
{
  "main" : "./lib/interface.js"
}
```

然后将index.js重命名为interface.js并放入lib子文件夹下。以同样的方式再次调用这个包，依然可以正常使用。

Node.js在调用某个包时，会首先检查包中package.json文件的 `main` 字段，将其作为包的接口模块，如果package.json或 `main` 字段不存在，会尝试寻找index.js或index.node作为包的接口。

package.json是CommonJS规定的用来描述包的文件，完全符合规范的package.json文件应该含有以下字段。

- `name`：包的名称，必须是唯一的，由小写英文字母、数字和下划线组成，不能包含空格。
- `description`：包的简要说明。
- `version`：符合语义化版本识别[①]规范的版本字符串。
- `keywords`：关键字数组，通常用于搜索。
- `maintainers`：维护者数组，每个元素要包含 `name`、`email`（可选）、`web`（可选）字段。
- `contributors`：贡献者数组，格式与`maintainers`相同。包的作者应该是贡献者数组的第一个元素。
- `bugs`：提交bug的地址，可以是网址或者电子邮件地址。
- `licenses`：许可证数组，每个元素要包含 `type`（许可证的名称）和 `url`（链接到许可证文本的地址）字段。
- `repositories`：仓库托管地址数组，每个元素要包含 `type`（仓库的类型，如 `git`）、`url`（仓库的地址）和 `path`（相对于仓库的路径，可选）字段。

① 语义化版本识别（Semantic Versioning）是由Gravatars和GitHub创始人Tom Preston-Werner提出的一套版本命名规范，最初目的是解决各式各样版本号大小比较的问题，目前被许多包管理系统所采用。

- dependencies：包的依赖，一个关联数组，由包名称和版本号组成。

下面是一个完全符合 CommonJS 规范的 package.json 示例：

```
{
  "name": "mypackage",
  "description": "Sample package for CommonJS. This package demonstrates the required
     elements of a CommonJS package.",
  "version": "0.7.0",
  "keywords": [
     "package",
     "example"
  ],
  "maintainers": [
     {
        "name": "Bill Smith",
        "email": "bills@example.com",
     }
  ],
  "contributors": [
     {
        "name": "BYVoid",
        "web": "http://www.byvoid.com/"
     }
  ],
  "bugs": {
     "mail": "dev@example.com",
     "web": "http://www.example.com/bugs"
  },
  "licenses": [
     {
        "type": "GPLv2",
        "url": "http://www.example.org/licenses/gpl.html"
     }
  ],
  "repositories": [
     {
        "type": "git",
        "url": "http://github.com/BYVoid/mypackage.git"
     }
  ],
  "dependencies": {
     "webkit": "1.2",
     "ssl": {
        "gnutls": ["1.0", "2.0"],
        "openssl": "0.9.8"
     }
  }
}
```

创建一个包

npm 可以非常方便地发布一个包，比pip、gem、pear 要简单得多。在发布之前，首先需要让我们的包符合npm 的规范，npm 有一套以CommonJS 为基础包规范，但与CommonJS
并不完全一致，其主要差别在于必填字段的不同。通过使用 npm init 可以根据交互式问答
产生一个符合标准的 package.json，例如创建一个名为byvoidmodule 的目录，然后在这个
目录中运行npm init

一个符合npm 规范的package.json 文件。创建一个
index.js 作为包的接口，一个简单的包就制作完成了。
在发布前，我们还需要获得一个账号用于今后维护自己的包，使用 npm adduser 根据提示输入用户名、密码、邮箱，等待账号创建完成。完成后可以使用 npm whoami 测验是否已经取得了账号。
接下来，在package.json 所在目录下运行npm publish，稍等片刻就可以完成发布了。
打开浏览器，访问http://search.npmjs.org/ 就可以找到自己刚刚发布的包了。现在我们可以在
世界的任意一台计算机上使用 npm install byvoidmodule 命令来安装它。

这个我还没试过 大约和npm 的发布 差不多

node 模块分2类
1 node 提供核心模块　　加载速度快 经次于缓存加载
2 由用户编写的模块

核心模块 在node 源码编译过程中 就开始 跑了
直接 require("..") 就可以了　不需要 路径了

文件模块 则是在运行的时候 动态加载　需要 路径来定位路径

node 核心模块

核心模块是Node.js 的心脏，它由一些精简而高效的库组成，为Node.js 提供了基本的API。本章中，我们挑选了一部分最常用的核心模块加以详细介绍，主要内容包括：
全局对象；
常用工具；
事件机制；
文件系统访问；
HTTP 服务器与客户端。

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。

console 输出

核心模块 pdf里面有　我这里就不拿出来了

放置在当前目录的node_modules 子目录下

C:\Users\Administrator\AppData\Roaming\npm\node_modules

架，多数功能只是对 HTTP 协议中常用操作的封装，更多的功能需要插件或者整合其他模块来完成。

下面用 Express 重新实现前面的例子：

express 框架

npm express -g

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

var app = express.createServer();
app.use(express.bodyParser());
app.all('/', function(req, res) {
  res.send(req.body.title + req.body.text);
});

app.listen(3000);
```

可以看到，我们不需要手动编写 req 的事件监听器了，只需加载 express.bodyParser() 就能直接通过 req.body 获取 POST 的数据了。

5.2　快速开始

在上一小节我们已经介绍了 Web 开发的典型架构，我们选择了用 Express 作为开发框架来开发一个网站，从现在开始我们就要真正动手实践了。

5.2.1　安装 Express

首先我们要安装 Express。如果一个包是某个工程依赖，那么我们需要在工程的目录下使用本地模式安装这个包，如果要通过命令行调用这个包中的命令，则需要用全局模式安装（关于本地模式和全局模式，参见 3.3.4节），因此按理说我们使用本地模式安装 Express 即可。但是Express 像很多框架一样都提供了 Quick Start（快速开始）工具，这个工具的功能通常是建立一个网站最小的基础框架，在此基础上完成开发。当然你可以完全自己动手，但我还是推荐使用这个工具更快速地建立网站。为了使用这个工具，我们需要用全局模式安装 Express，因为只有这样我们才能在命令行中使用它。运行以下命令：

```
$ npm install -g express
```

我遇到 express不是内部或外部命令

等待数秒后安装完成，我们就可以在命令行下通过 express 命令快速创建一个项目了。在这之前先使用 express --help 查看帮助信息：

```
Usage: express [options] [path]

Options:
  -s, --sessions           add session support
  -t, --template <engine>  add template <engine> support (jade|ejs). default=jade
```

查看版本: express -V

注意express -V中的V要大写,不然很多版本中会不识别

卸载: 命令 npm uninstall -g express

最新express4.0 版本 安装命令工具 npm install -g express-generator

```
  -c, --css <engine>      add stylesheet <engine> support (stylus). default=plain css
  -v, --version           output framework version
  -h, --help              output help information
```

Express 在初始化一个项目的时候需要指定模板引擎，默认支持Jade和ejs，为了降低学习难度我们推荐使用 ejs[①]，同时暂时不添加 CSS 引擎和会话支持。

5.2.2 建立工程

首先要 / 到目录里面去这里

通过以下命令建立网站基本结构：

是 express -e ejs miao5cm.org

```
express -t ejs microblog
```

当前目录下出现了子目录 microblog，并且产生了一些文件：

```
   create : microblog
   create : microblog/package.json
   create : microblog/app.js
   create : microblog/public
   create : microblog/public/javascripts
   create : microblog/public/images
   create : microblog/public/stylesheets
   create : microblog/public/stylesheets/style.css
   create : microblog/routes
   create : microblog/routes/index.js
   create : microblog/views
   create : microblog/views/layout.ejs
   create : microblog/views/index.ejs

dont forget to install dependencies:
$ cd microblog && npm install
```

这一步很重要，漏掉了 不可以
类似 spm build
会自动加载 package.json

它还提示我们要进入其中运行 `npm install`，我们依照指示，结果如下：

```
ejs@0.6.1 ./node_modules/ejs
express@2.5.8 ./node_modules/express
-- qs@0.4.2
-- mime@1.2.4
-- mkdirp@0.3.0
-- connect@1.8.5
```

根据paceage.json 里面的东西
来下载 资源

它自动安装了依赖 ejs 和 express。这是为什么呢？检查目录中的 package.json 文件，内容是：

① ejs（Embedded JavaScript）是一个标签替换引擎，其语法与 ASP、PHP 相似，易于学习，目前被广泛应用。Express 默认提供的引擎是 jade，它颠覆了传统的模板引擎，制定了一套完整的语法用来生成 HTML 的每个标签结构，功能强大但不易学习。

```
{
    "name": "microblog"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.8"
    , "ejs": ">= 0.0.1"
  }
}
```

其中 dependencies 属性中有express 和ejs。无参数的 npm install 的功能就是检查当前目录下的 package.json，并自动安装所有指定的依赖。

5.2.3　启动服务器

用 Express 实现的网站实际上就是一个 Node.js 程序，因此可以直接运行。我们运行 node app.js，看到 Express server listening on port 3000 in development mode。

接下来，打开浏览器，输入地址 http://localhost:3000，你就可以看到一个简单的 Welcome to Express 页面了。如果你能看到如图5-2 所示的页面，那么说明你的设定正确无误。

图5-2　Express 初始欢迎页面

要关闭服务器的话，在终端中按 Ctrl + C。注意，如果你对代码做了修改，要想看到修改后的效果必须重启服务器，也就是说你需要关闭服务器并再次运行才会有效果。如果觉得有些麻烦，可以使用 supervisor 实现监视代码修改和自动重启，具体使用方法参见 3.1.3 节。