node基础

- 使用node来实现第一个http服务器

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

http.createServer(function(request, response) {
    // 发送 HTTP 头部
    // HTTP 状态值: 200 : OK
    // 内容类型: text/plain
    response.writeHead(200, { "Content - Type": "text / plain" });
    console.log("欢迎来到慧点科技");
    // 发送响应数据 “Hello World”
    response.end("Hello World\ n");
}).listen(8888);
```

- 学习资料
  - 菜鸟教程: http://www.runoob.com/nodejs/nodejs-tutorial.html
  - nodejs参考手册: http://nodeapi.ucdok.com/#/api/
  - 模块
    - require是啥?
    - 模块化开发
      - 为什么会有模块?
        - 在JavaScript发展初期就是为了实现简单的页面交互逻辑，寥寥数语即可；如今CPU、浏览器性能得到了极大的提升，很多页面逻辑迁移到了客户端（表单验证等），随着web2.0时代的到来，Ajax技术得到广泛应用，jQuery等前端库层出不穷，前端代码日益膨胀
          这时候JavaScript作为嵌入式的脚本语言的定位动摇了，JavaScript却没有为组织代码提供任何明显帮助，甚至没有类的概念，更不用说模块（module）了，JavaScript极其简单的代码组织规范不足以驾驭如此庞大规模的代码
      - 模块的好处
        - 既然JavaScript不能handle如此大规模的代码，我们可以借鉴一下其它语言是怎么处理大规模程序设计的，在Java中有一个重要带概念——package，逻辑上相关的代码组织到同一个包内，包内是一个相对独立的王国，不用担心命名冲突什么的，那么外部如果使用呢？直接import对应的package即可
          import java.util.ArrayList;
        - 遗憾的是JavaScript在设计时定位原因，没有提供类似的功能，开发者需要模拟出类似的功能，来隔离、组织复杂的JavaScript代码，我们称为模块化。
          一个模块就是实现特定功能的文件，有了模块，我们就可以更方便地使用别人的代码，想要什么功能，就加载什么模块。模块开发需要遵循一定的规范，各行其是就都乱套了
        - commonjs规范
          - CommonJS就是为JS的表现来制定规范，因为js没有模块的功能所以CommonJS应运而生，它希望js可以在任何地方运行，不只是浏览器中。
          - CommonJS能有一定的影响力，我觉得绝对离不开Node的人气，不过喔，Node，CommonJS，浏览器甚至是W3C之间有什么关系呢，
          - |--------------浏览器----- ------------------| |-------------------------CommonJS----------------------------------|

            | BOM | | DOM | | ECMAScript | | FS | | TCP | | Stream | | Buffer | |........|

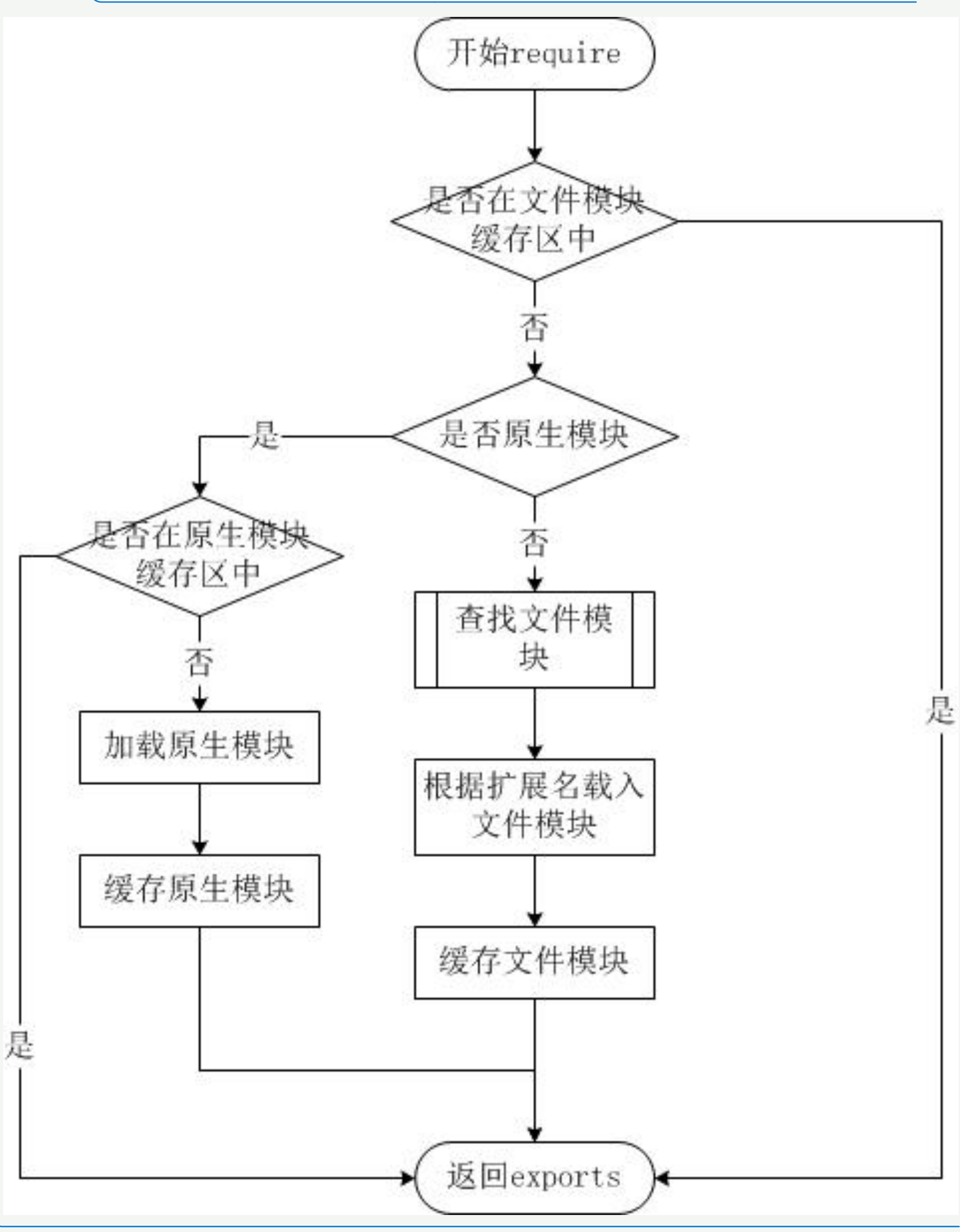
            |-------W3C-----------| |---------------------------------------Node--|
      - 案例：引入url模块解析路径

```
var http = require("http");
//引入一个http的模块
var url = require("url");
//引入路径模块
var server = http.createServer(function(req, res) {
    //parse(路径,第二个参数为将请求参数解析为对象)
    var urlObj = url.parse(req.url, true);
    console.log(urlObj);
    res.end();
});
server.listen(8080);
```

      - 创建自己的模块
        - 加载当前目录的模块
          - 不加var 的变量是全局变量，在模块内部最好加var
          - 要导出一个属性或者方法使用:module.exports
            - 如果是模块下的方法和属性使用:module.exports={xx:xx}
            - 如果是想让模块直接接受方法就导出函数
        - 加载非当前目录的模块
          - 首先会看当前目录是否有一个node_modules，如果有就看进入里面查找是否有叫模块名称的文件或者是否有叫模块名称的文件夹
          - 如果是模块名称的文件夹需要看一下文件夹内部是否有index.js或者package.json中是否有设置mian属性

```
{
    "name": "home",
    "version": "0.0.1",
    "main": "home2.js"
}
```

          - 如果当前目录中没有node_modules，会向上查找
            - 可以通过：console.log(module)输出
        - 加载原生模块



      - 扩展
        - 非node项目中也可以使用模块化开发技术，用得比较多的模块加载器有：requirejs(CMD),seaJs(AMD)
  - npm
    - npm
      - 包（一组模块的集合）
      - 举例：安装express
      - NPM作为Node的包管理器
      - 案例1：安装一个express模块,看他的结构
  - 参考资料: http://www.cnblogs.com/dolphinX/p/4381855.html
- repl(**交互式解释器**)
  - REPL(Read Eval Print Loop:交互式解释器)概念
    - 表示一个电脑的环境，类似 Window 系统的终端或 Unix/Linux shell，我们可以在终端中输入命令，并接收系统的响应。
    - 读取 – 读取用户输入，解析输入了Javascript 数据结构并存储在内存中。

      执行 – 执行输入的数据结构

      打印 – 输出结果

      循环 – 循环操作以上步骤直到用户两次按下 ctrl–c 按钮退出。
    - 进入repl方式：终端中键入node
  - repl处理:
    - 表达式运算

```
$ node
> 1 +4
5
> 5 / 2
2.5
> 3 * 6
18
> 4 – 1
3
> 1 + ( 2 * 3 ) – 4
3
>
```

    - 使用变量

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log(“Hello World”)
Hello World
undefined
```

    - **多行表达式**

```
$ node
> var x = 0
undefined
> do {
… x++;
… console.log(“x: “ + x);
… } while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

    - 下划线
      - 你可以使用下划线(_)获取表达式的运算结果

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

    - **REPL 命令**
      - trl + c – 退出当前终端。

        ctrl + c 按下两次 – 退出 Node REPL。

        ctrl + d – 退出 Node REPL.

        向上/向下 键 – 查看输入的历史命令

        tab 键 – 列出当前命令

        .help – 列出使用命令

        .break – 退出多行表达式

        .clear – 退出多行表达式