[SeaJs学习 2](#_Toc375161094)

[路径研究 2](#_Toc375161095)

[Node.js学习 2](#_Toc375161096)

[url 2](#_Toc375161097)

[非阻塞 *child\_process* 3](#_Toc375161098)

[以非阻塞操作进行请求响应 4](#_Toc375161099)

[Node.js不是单线程的吗？为什么又有child\_process、cluster等多线程模块？ 6](#_Toc375161100)

[重复启动服务器会报错 7](#_Toc375161101)

[Socket.IO学习 7](#_Toc375161102)

[客户端找不到socket.io.js 9](#_Toc375161103)

[问题描述 9](#_Toc375161104)

[原因分析 9](#_Toc375161105)

[解决方案 9](#_Toc375161106)

[测试服务器端 10](#_Toc375161107)

[测试node.js 10](#_Toc375161108)

[测试socket.io 10](#_Toc375161109)

# SeaJs学习

## 路径研究

1. seajs.use 中的相对路径始终相对当前页面来解析。

<!DOCTYPE html>

<html>

<head>

<title></title>

<script src="script/sea.js"></script>

</head>

<body>

<script>

seajs.config({

//相对标识永远相对当前模块的 URI 来解析

//此处使用相对路径，则是相对于当前页面的路径

base: './'

});

seajs.use("main", function(main){

main.init();

})

</script>

</body>

</html>

# Node.js学习

## url

但是为了解析这些数据，我们需要额外的Node.JS模块，它们分别是*url*和*querystring*模块。

url.parse(string).query

|

url.parse(string).pathname |

| |

| |

------ -------------------

http://localhost:8888/start?foo=bar&hello=world

--- -----

| |

| |

querystring(string)["foo"] |

|

querystring(string)["hello"]

当然我们也可以用*querystring*模块来解析POST请求体中的参数，稍后会有演示。

路由模块

## 非阻塞 child\_process

var exec = require("child\_process").exec;  
  
function start() {  
  console.log("Request handler 'start' was called.");  
  var content = "empty";  
  
  exec("ls -lah", function (error, stdout, stderr) {  
    content = stdout;  
  });  
  
  return content;  
}  
  
function upload() {  
  console.log("Request handler 'upload' was called.");  
  return "Hello Upload";  
}  
  
exports.start = start;  
exports.upload = upload;

上述代码中，我们引入了一个新的Node.js模块，child\_process。之所以用它，是为了实现一个既简单又实用的非阻塞操作：exec()。

exec()做了什么呢？它从Node.js来执行一个shell命令。在上述例子中，我们用它来获取当前目录下所有的文件（“ls -lah”）,然后，当/startURL请求的时候将文件信息输出到浏览器中。

上述代码是非常直观的： 创建了一个新的变量content（初始值为“empty”），执行“ls -lah”命令，将结果赋值给content，最后将content返回。

## 以非阻塞操作进行请求响应

到目前为止，我们的应用已经可以通过应用各层之间传递值的方式（请求处理程序 -> 请求路由 -> 服务器）将请求处理程序返回的内容（请求处理程序最终要显示给用户的内容）传递给HTTP服务器。

通过创建子进程来实现

为了使整个过程非阻塞，Node.js会将POST数据拆分成很多小的数据块，然后通过触发特定的事件，将这些小数据块传递给回调函数。这里的特定的事件有data事件（表示新的小数据块到达了）以及end事件（表示所有的数据都已经接收完毕）。

我们需要告诉Node.js当这些事件触发的时候，回调哪些函数。怎么告诉呢？ 我们通过在request对象上注册监听器（listener） 来实现。这里的request对象是每次接收到HTTP请求时候，都会把该对象传递给onRequest回调函数。

还等什么，马上来实现。先从server.js开始：

var http = require("http");  
var url = require("url");  
  
function start(route, handle) {  
  function onRequest(request, response) {  
    var postData = "";  
    var pathname = url.parse(request.url).pathname;  
    console.log("Request for " + pathname + " received.");  
  
    request.setEncoding("utf8");  
  
    request.addListener("data", function(postDataChunk) {  
      postData += postDataChunk;  
      console.log("Received POST data chunk '"+  
      postDataChunk + "'.");  
    });  
  
    request.addListener("end", function() {  
      route(handle, pathname, response, postData);  
    });  
  
  }  
  
  http.createServer(onRequest).listen(8888);  
  console.log("Server has started.");  
}  
  
exports.start = start;

上述代码做了三件事情： 首先，我们设置了接收数据的编码格式为UTF-8，然后注册了“data”事件的监听器，用于收集每次接收到的新数据块，并将其赋值给postData 变量，最后，我们将请求路由的调用移到end事件处理程序中，以确保它只会当所有数据接收完毕后才触发，并且只触发一次。我们同时还把POST数据传递给请求路由，因为这些数据，请求处理程序会用到。

## Node.js不是单线程的吗？为什么又有child\_process、cluster等多线程模块？

NodeJS中的JavaScript也确实是在单线程上执行，但是作为宿主的NodeJS，它本身并非是单线程的，

child\_process模块可以创建子进程

cluster通过child\_process.fork的方式创建出多个子进程实例，这些实例会自动共享相同的侦听端口。它貌似是对child\_process的封装

对于cpu密集型运算，可以分解为小任务，通过process.netxtTick延迟，错开执行；或者使用child\_process、cluster等模块创建子进程

对于Node.js多线程这块还需要进一步研究！

NodeJS 封装了内部的异步实现后，导致程序员无法直接操作线程，也就造成所有的业务逻辑运算都会丢到JavaScript的执行线程上，这也就意味着，在高并发请求的时候，I/O的问题是很好的解决了，但是所有的业务逻辑运算积少成多地都运行在JavaScript线程上，形成了一条拥挤的JavaScript运算线程。NodeJS的弱点在这个时候会暴露出来，单线程执行运算形成的瓶颈，拖慢了I/O的效率。这大概可以算得上是密集运算情况下无法很好利用多核 CPU的缺点。这条拥挤的JavaScript线程，给I/O形成了性能上限。

但是，事情又并非绝对的。回到前端浏览器中，为了解决线程拥挤的情况，Web Worker应运而生。而同样，Node也提供了child\_process.fork来创建Node的子进程。在一个Node进程就能很好的解决密集 I/O的情况下，fork出来的其余Node子进程可以当作常驻服务来解决运算阻塞的问题（将运算分发到多个Node子进程中上去，与Apache创建多个子进程类似）。当然child\_process/Web Worker的机制永远只能解决单台机器的问题，大的Web应用是不可能一台服务器就能完成所有的请求服务的。拜NodeJS在I/O上的优势，跨OS的多Node之间通信的是不算什么问题的。解决NodeJS的运算密集问题的答案其实也是非常简单的，就是将运算分发到多个CPU上。请参考文章后的multi-node的性能测试，可以看到在多Node进程的情景下，响应请求的速度被大幅度提高（感谢CNode社区的snoopy友情测试）。

在文章的写作过程中，Node最新发布的0.6.0版本，新增了cluster模块。该模块的作用是可以通过fork的方式创建出多个子进程实例，这些实例会自动共享相同的侦听端口。你可以根据当前计算机上的CPU数量来创建相应的实例数，以此达到分发请求，充分利用CPU的目的。详情请参阅[官方文档](http://http/nodejs.org/docs/v0.5.10/api/cluster.html)。在之前的解决运算密集问题中，工程师需要multi-node这样的库或者其他方案去手动分发请求，在cluster模块的支持下，可以释放掉工程师在解决此问题上的大部分精力。

## 重复启动服务器会报错

# Socket.IO学习

1. 启动服务器还是交给node，打开命令行窗口，定位到server.js所在文件夹，输入node server.js启动服务器。

    在上面的index.html中，注意这行：<script src="/socket.io/socket.io.js"></script>。如果不想使用本地的socket.io脚本，可以直接使用下面这个公开的脚本：

|  |
| --- |
| <script src="<http://cdn.socket.io/stable/socket.io.js>"></script> |

 此外需要注意这行：socket = io.connect(null)。这里的null代表连接本地服务，可以换成"localhost"，效果也是一样的。

2. 可以使用socket.io直接启动http服务。例如：

|  |
| --- |
| var io = require('socket.io').listen(80);    io.sockets.on('connection', function (socket) {      io.sockets.emit('this', { will: 'be received by everyone'});  }); |

 3. socket.io可以直接通过send方法发送消息，使用message事件接收消息，例如：

|  |
| --- |
| //server.js  var io = require('socket.io').listen(80);    io.sockets.on('connection', function (socket) {      socket.on('message', function () { });  });    //index.html  <script>  var socket = io.connect('<http://localhost/>');  socket.on('connect', function () {      socket.send('hi');      socket.on('message', function (msg) {          // my msg      });  });  </script> |

 4. 发送和处理数据

    两端可以互发事件，互发数据，相互通信。发送事件的代码为：socket.emit(action, data, function)，其中action为事件的名称，data为数据，function为回调函数;处理事件代码 为：socket.on(action,function)，如果emit发送的时候有数据data，则function中参数包含了这个数据。 socket.io除了发送和处理内置事件，如connect, disconnect, message。还允许发送和处理自定义事件，例如:

|  |
| --- |
| //服务端：  io.sockets.on('connection', function (socket) {      socket.emit('news', { hello: 'world' });      socket.on('my other event', function (data) {          console.log(data);      });  });    //客户端：  <script src="/socket.io/socket.io.js"></script>  <script>  var socket = io.connect('[http://localhost](http://localhost/)');  socket.on('news', function (data) {      console.log(data);      socket.emit('my other event', { my: 'data' });  });  </script> |

 5. 从上面可以看出来，发送数据的时候，send和emit是都可以使用的。只不过emit更是强化了自定义事件的处理。

6. 可以在服务端使用socket的get/set方法存储客服端的相关数据，例如：

|  |
| --- |
| //服务端  var io = require('socket.io').listen(80);    io.sockets.on('connection', function (socket) {      socket.on('set nickname', function (name) {          socket.set('nickname', name, function () { socket.emit('ready'); });      });        socket.on('msg', function () {          socket.get('nickname', function (err, name) {              console.log('Chat message by ', name);          });      });  });  //客户端  <script>  var socket = io.connect('[http://localhost](http://localhost/)');    socket.on('connect', function () {      socket.emit('set nickname', confirm('What is your nickname?'));      socket.on('ready', function () {          console.log('Connected !');          socket.emit('msg', confirm('What is your message?'));      });  });  </script> |

## 客户端找不到socket.io.js

### 问题描述

我先从网上下载了一个socket.io.js，结果客户端io没有connect方法！这估计是因为下载的是0.6版本，而最新的是0.9版本。

官网上的示例是直接在客户端引入：

<script src="/socket.io/socket.io.js"></script>

但是这样却找不到socket.io.js！

### 原因分析

这是因为socket.io.js是包含在node.js下载的socket.io模块中的，它位于（如果监听的是8080端口）<http://localhost:8080/socket.io/socket.io.js>

### 解决方案

1. 硬编码为：<script src="<http://localhost:8080/socket.io/socket.io.js>"></script>

这样每次更改端口时，都需要更改src。且在部署时也很麻烦（如要手动改域名等）。

1. 在浏览器中打开<http://localhost:8080/socket.io/socket.io.js>文件，将该文件保存为静态文件socket.io.js，再在客户端引用。

# 测试服务器端

## 测试node.js

使用jasmine-node测试node.js

## 测试socket.io

使用jasmine测试socket.io