

首先 helloworld

```
console.log("hello world");
```

保存在文件 helloworld.js

```
F:\node.js>node helloworld.js
hello world
```

然后是一个完整的基于 Node.js 的 web 应用

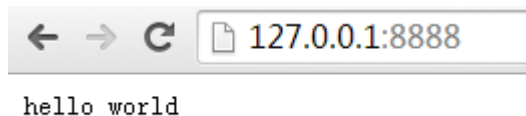
1. 一个基础的 HTTP 服务器

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

```
http.createServer(function(request,response)
{
    response.writeHead(200,{"Content-Type":"text/plain"});
    response.write("hello world");
    response.end();
}).listen(8888);
```

保存为 server.js

node server.js 启动，打开浏览器



hello world

一个简单的 HTTP 服务器就成功了。

下面分析一下这个服务器代码：

//require node.js 的 http 模块，并赋值给 http

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

//调用 http 模块的 createServer 函数

//它返回一个对象，然后对象调用 listen 侦听 8888 端口

```
http.createServer(
    //createServer 只有一个参数，而且参数为一个匿名函数
    function(request,response)
    {
        response.writeHead(200,{"Content-Type":"text/plain"});
        response.write("hello world");
        response.end();
    }
)
```

```
    }  
  ).listen(8888);
```

其实我们也可以用以下的代码(替换匿名函数):

```
var http=require("http");  
  
function onRequest(request,response)  
{  
  response.writeHead(200,{"Content-Type":"text/plain"});  
  response.write("hello world");  
  response.end();  
}  
  
http.createServer(onRequest).listen(8888);
```

那为什么 node.js 要用那种方式呢? 那就要说说 node.js 的基于事件驱动的回调。

node.js 是异步的, 任何时候请求都可能到达, 但是服务器程序是单进程的, 怎么才能在一个新的请求达到之后, 我们可以控制呢

上面的代码, 我们在创建服务器的时候, 传递了一个函数, 任何的请求达到, 这个函数都会被调用。这个方式其实就是回调。

下面我们修改代码来测试一下:

```
var http=require("http");  
  
function onRequest(request,response)  
{  
  console.log("请求到达");  
  response.writeHead(200,{"Content-Type":"text/plain"});  
  response.write("hello world");  
  response.end();  
}  
  
http.createServer(onRequest).listen(8888);  
console.log("服务器已启动");
```

A terminal window with a black background and white text. The first line shows the command 'F:\node.js>node server.js'. The second line shows the output '服务器已启动'. The next four lines show the output '请求到达' repeated four times, indicating incoming requests.

可以看到, 程序运行时, 已输出服务器已启动, 在浏览器打开之后, 输出请求到达。

在这里访问一个网页, 会出现两次请求达到是因为大部分的服务器都会在你访问

<http://127.0.0.1:8888> /时尝试读取 <http://127.0.0.1:8888/favicon.ico>

回调说清楚之后，再来看看服务器是怎么处理请求的。

当请求到达之后，onRequest 函数被触发之后，被传入了两个参数 request 和 response。这两个对象，你可以使用它们的方法来处理 HTTP 请求细节。

上面的代码，我们处理的细节是：

当收到请求时，使用 response.writeHead 返回一个 HTTP 状态和 HTTP 头，然后使用 response.write 在 HTTP 主体发送文本"hello world"

最后调用 response.end()完成响应。在这里，我们不在乎请求细节，所以没有使用 request 对象

编写自己的模块

我们修改一下上面的代码，把它写到一个函数里面：

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

function start()
{
    function onRequest(request,response)
    {
        console.log("请求到达");
        response.writeHead(200,{"Content-Type":"text/plain"});
        response.write("hello world");
        response.end();
    }

    http.createServer(onRequest).listen(8888);
    console.log("服务器已启动");
}

exports.start = start;
```


然后，我们创建另一个文件 index.js 请求里面的模块

index.js 代码如下：

```
var server = require("./server")

server.start()
```

运行如下：

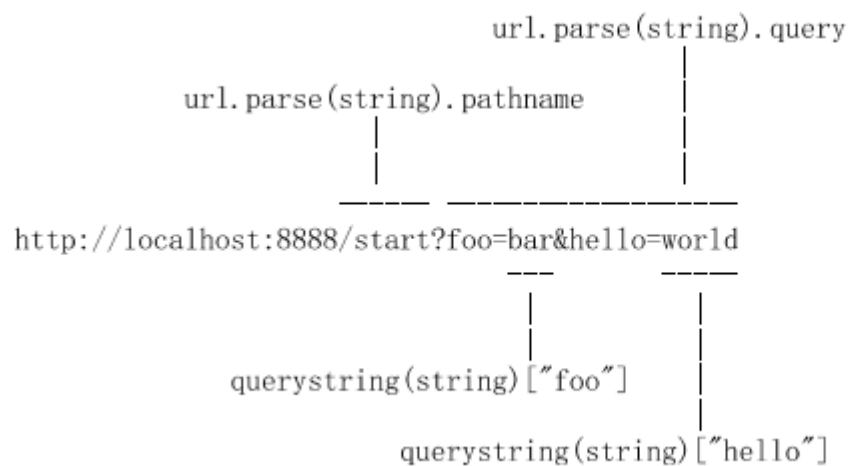


```
F:\node.js>node index.js
服务器已启动
请求到达
请求到达
```

说明自定义的 `server` 模块成功

然后我们要添加代码使得能够处理不同的 HTTP 请求，首先需要路由选择

为了得到请求的 URL，我们需要在 `request` 对象里面解析。这里，代码中需要添加 `url` 和 `querystring` 模块



修改 `onRequest` 函数中的代码：

```
function onRequest(request,response)
{
  var pathname = url.parse(request.url).pathname;
  console.log("请求:" + pathname + "到达");
  response.writeHead(200,{"Content-Type":"text/plain"});
  response.write("hello world");
  response.end();
}
```

记得在文件开头加入 `var url = require("url")`

运行：

```
F:\node.js>node index.js
服务器已启动
请求:/到达
请求:/favicon.ico到达
```

验证了为什么浏览一次为什么是两次请求。

现在我们创建一个 `router.js`，在里面编写我们根据不同的 URL 请求来区别处理的代码。

```
function route(pathname)
{
```

```
    console.log("路由请求:" + pathname);  
}
```

```
exports.route = route;
```

然后把 pathname 通过 start 传入，代码如下：

```
var server = require("./server")  
var router = require("./router")
```

```
server.start(router.route)
```

在 onRuquest 函数中加入 route(pathname);

运行：



```
P:\node.js>node index.js  
服务器已启动  
请求:/到达  
路由请求:/  
请求:/favicon.ico到达  
路由请求:/favicon.ico
```

现在我们并没有对 URL 请求区别处理，下面我们真正创建 requestHandlers 模块：

```
function start()  
{  
    console.log("请求处理函数： 处理 start");  
}  
  
function upload()  
{  
    console.log("请求处理函数： 处理 upload");  
}  
  
exports.start = start;  
exports.upload = upload;
```

修改 router 函数：

```
function route(handle,pathname)  
{  
    console.log("路由请求:" + pathname);  
    if (typeof handle[pathname] === "function")  
    {  
        handle[pathname]();  
    }  
    else  
    {  
        console.log("路由请求没找到 " + pathname)
```

```
}  
}
```

运行效果:

```
F:\node.js>node index.js  
服务器已启动  
请求:/start到达  
路由请求:/start  
请求处理函数: 处理start  
请求:/favicon.ico到达  
路由请求:/favicon.ico  
路由请求没找到 /favicon.ico  
请求:/upload到达  
路由请求:/upload  
请求处理函数: 处理upload  
请求:/favicon.ico到达  
路由请求:/favicon.ico  
路由请求没找到 /favicon.ico
```

现在的效果, 我们对于不同的请求都是输出 hello world, 那我们怎么才能根据不同的请求输出不同的字符呢

我们更改 start 和 upload 函数, 让它们执行之后返回一个字符串。

```
function start()  
{  
  console.log("请求处理函数: 处理 start");  
  return "hello start";  
}
```

```
function upload()  
{  
  console.log("请求处理函数: 处理 upload");  
  return "hello upload";  
}
```

然后我们还需更改 route 函数:

```
function route(handle,pathname)  
{  
  console.log("路由请求:" + pathname);  
  if (typeof handle[pathname] === "function")  
  {  
    return handle[pathname]();  
  }  
  else  
  {  
    console.log("路由请求没找到 " + pathname);  
    return "404 Not Found";  
  }  
}
```

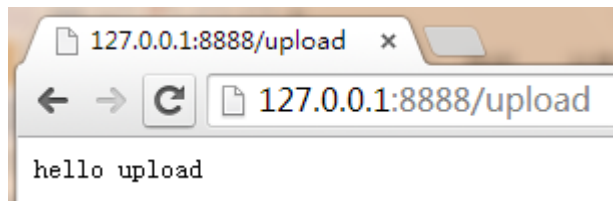
```
    }  
}
```

让它返回错误信息。

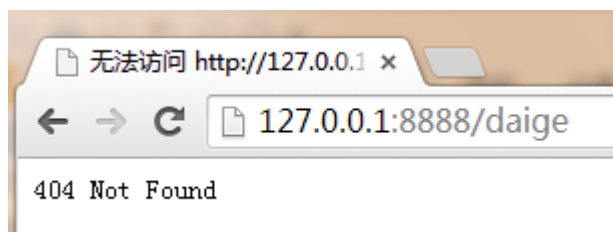
最后我们就要修改 `onRequest` 函数，让它打印出函数返回的信息。

```
var content = route(handle,pathname);  
response.write(content);
```

运行结果，会根据不同的 URL 请求打印出不同的信息：



找不到处理函数输出错误：

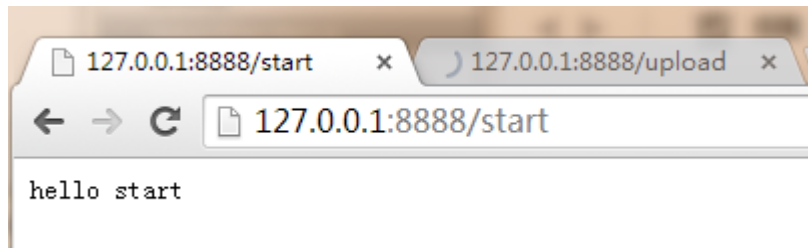


关于 `node.js` 的阻塞和非阻塞问题，我们通程序例子来说明：

我们实现一个 `sleep` 函数来让 `start` 等待 10 秒在返回，代码如下：

```
function start()  
{  
    console.log("请求处理函数： 处理 start");  
  
    function sleep(miliSeconds)  
    {  
        var startTime = new Date().getTime();  
        //等待 miliSeconds 毫秒  
        while(new Date().getTime < startTime + miliSeconds);  
    }  
  
    sleep(10000);  
    return "hello start";  
}
```

然后，我们开两个浏览器标签页，同时访问，观察结果：



发现，`upload()`函数也跟着阻塞。

这是为什么呢？

`node.js` 是单线程的，它是通过事件轮询来实现并行操作。所以，我们尽量避免阻塞操作。如果要用非阻塞操作，我们就需要回调，通过一个函数作为参数传递给其他需要时间的函数。`node.js` 会让会时间运行的函数继续运行，`node.js` 不等待它完成并且该函数需要提供一个回调函数，等它运行完了之后 `node.js` 会调用该回调函数。

我们先来看一种错误的非阻塞方法：

```
var exec = require("child_process").exec;
```

```
function start()
{
    console.log("请求处理函数： 处理 start");

    var content = "empty";

    exec("dir ",function(error,stdout,stderr)
    {
        content = stdout;
    });

    return content;
}

function upload()
{
    console.log("请求处理函数： 处理 upload");
    return "hello upload";
}

exports.start = start;
exports.upload = upload;
```

上面的代码，我们引入了一个新的模块，然后 `exec()`来执行一个 `dir` 命令。最后通过请求输出到浏览器上。

运行发现浏览器一直输出 `empty`



为什么呢？

因为在执行 `exec()` 的之后，`node.js` 会立即执行 `return content`，这个时候，`content` 的值为 `empty`。

而 `exec()` 的回调函数

还没有执行到——因为 `exec()` 的操作是异步的。

下面我们采用正确的方式：将 `response` 对象通过请求路由传递给请求处理程序。

`server.js`

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

function start(route,handle)
{
  function onRequest(request,response)
  {
    var pathname = url.parse(request.url).pathname;
    console.log("请求:" + pathname + "到达");

    route(handle,pathname, response);
  }

  http.createServer(onRequest).listen(8888);
  console.log("服务器已启动");
}

exports.start = start;
```

把 `response` 当参数传给请求路由。

`router.js`

```
function route(handle,pathname,response)
{
  console.log("路由请求:" + pathname);

  if (typeof handle[pathname] === "function")
  {
    handle[pathname](response);
  }
  else
  {
    console.log("路由请求没找到 " + pathname);
    response.writeHead(404,{"Content-Type":"text/plain"});
```

```

        response.write("404 Not Found");
        response.end();
    }
}

```

```
exports.route = route;
```

```
exports.route = route;
```

在路由选择之后，才决定 HTTP 头等内容。

requestHandlers.js

```
var exec = require("child_process").exec;
```

```

function start(response)
{
    console.log("请求处理函数： 处理 start");

    exec("dir ",function(error,stdout,stderr)
    {
        response.writeHead(200,{"Content-Type":"text/plain"});
        response.write(stdout);
        response.end();
    });
}

```

```

function upload(response)
{
    console.log("请求处理函数： 处理 upload");
    response.writeHead(200,{"Content-Type":"text/plain"});
    response.write("hello upload");
    response.end();
}

```

```

exports.start = start;
exports.upload = upload;

```

根据路由选择和执行结果，设定 response 的内容。

运行：

执行 dir 成功



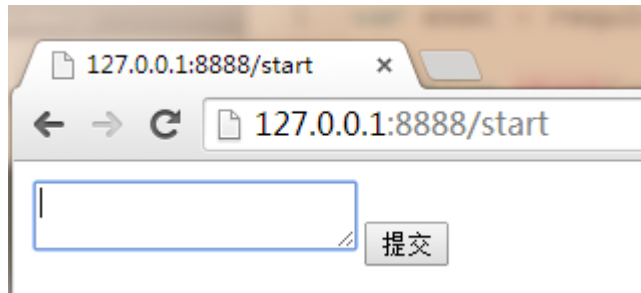
最后我们转到更有用的场景中来，假设用户和网站交互是这样的：  
 用户选择一个图片，上传一个图片，然后网站的应用显示这个图片。  
 修改 requestHandlers.js 中 start 函数：

```
function start(response)
{
    console.log("请求处理函数： 处理 start");

    var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="2" cols="20"></textarea>'+
    '<input type="submit" value="提交" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}
```

在浏览器中添加了一个文本域和提交按钮，点击提交会触发 upload



接下来，我们处理 POST 请求问题。

为了使整个过程非阻塞，node.js 会将 POST 的数据拆分成很多小的数据块，然后通过触发特定的事件，将小数据块传递给回调函数。

这些事件有 data 事件，end 事件等。

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 request 对象上注册监听器来实现。

修改 server.js

```
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("请求:" + pathname + "到达");
```

```
    //设置数据的编码为 utf-8
```

```
    request.setEncoding("utf8");
```

```
    //注册 data 事件的监听器，数据到达之后赋值给 postData
```

```
    request.addListener("data",function(postDataChunk)
```

```
    {
```

```
      postData += postDataChunk;
```

```
      console.log("收到新的 post 数据:"+postDataChunk + "。");
```

```
    });
```

```
    //注册 end 事件的监听器,数据接受完毕之后才触发请求路由
```

```
    request.addListener("end",function()
```

```
    {
```

```
      route(handle,pathname,response,postData);
```

```
    });
```

```
  }
```

```

    http.createServer(onRequest).listen(8888);
    console.log("服务器已启动");
}

```

exports.start = start;

再修改 router.js

```

function route(handle,pathname,response,postData)
{
    console.log("路由请求:" + pathname);

    if (typeof handle[pathname] === "function")
    {
        handle[pathname](response,postData);
    }
    else
    {
        console.log("路由请求没找到 " + pathname);
        response.writeHead(404,{"Content-Type":"text/plain"});

        response.write("404 Not Found");
        response.end();
    }
}

```

exports.route = route;

然后是 requestHandlers.js

```

var exec = require("child_process").exec;

function start(response,postData)
{
    console.log("请求处理函数: 处理 start");

    var body = '<html>'+
        '<head>'+
        '<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 />'+
        '</head>'+
        '<boby>'+
        '<form action="/upload" method="post">'+
        '<textarea name="text" rows="2" cols="20"></textarea>'+
        '<input type="submit" value="提交" />'+

```

```

'</form>' +
'</boby>' +
'</html>';

response.writeHead(200, {"Content-Type": "text/html"});
response.write(body);
response.end();
}

```

```

function upload(response, postData)
{
    console.log("请求处理函数： 处理 upload");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("你正在发送:" + postData);
    response.end();
}

```

```

exports.start = start;
exports.upload = upload;

```

然后呢，我们需要把整个消息传递给处理程序

```

var querystring = require("querystring");

```

```

function start(response, postData)
{
    console.log("请求处理函数： 正在处理 start");

    var body = '<html>' +
'<head>' +
'<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 />' +
'</head>' +
'<boby>' +
'<form action="/upload" method="post">' +
'<textarea name="text" rows="2" cols="20"></textarea>' +
'<input type="submit" value="提交" />' +
'</form>' +
'</boby>' +
'</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
}

```

```
    response.end();  
  }  
}
```

```
function upload(response,postData)  
{  
  console.log("请求处理函数: 正在处理 upload");  
  response.writeHead(200,{"Content-Type":"text/plain"});  
  response.write("你正在发送的文本:"+ querystring.parse(postData).text);  
  response.end();  
}
```

```
exports.start = start;  
exports.upload = upload;
```

最后的最后，实现上传图片并展示图片：  
安装

```
F:\node.js>npm install formidable  
formidable@1.0.15 node_modules\formidable
```

修改几个文件，代码中有说明：

server.js

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

```
function start(route,handle)  
{  
  function onRequest(request,response)  
  {  
    var pathname = url.parse(request.url).pathname;  
    console.log("请求:" + pathname + "到达");  
  
    route(handle,pathname,response,request);  
  
  }  
  
  http.createServer(onRequest).listen(8888);  
  console.log("服务器已启动");  
}
```

```
exports.start = start;
```

index.js

```

var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}

handle[""]      = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;
handle["/show"]  = requestHandlers.show;

server.start(router.route,handle);

requestHandlers.js
var querystring = require("querystring");

//本地文件读取模块
var fs          = require("fs");

//使用 formidable 模块
var formidable   = require("formidable");

function start(response,request)
{
    console.log("请求处理函数： 正在处理 start");

    var body = '<html>'+
        '<head>'+
        '<meta http-equiv="Content-Type" content="text/html;charset=UTF-8 />'+
        '</head>'+
        '<boby>'+
        '<form action="/upload" enctype="multipart/form-data" method="post">'+
        '<input type="file" name="upload" multiple="multiple">'+
        '<input type="submit" value="文件上传" />'+
        '</form>'+
        '</boby>'+
        '</html>';

    response.writeHead(200,{"Content-Type":"text/html"});
    response.write(body);
    response.end();
}

```



```

function upload(response,request)
{
    console.log("请求处理函数： 正在处理 upload");

    var form = new formidable.IncomingForm();
    console.log("开始解析");
    form.parse(request,function(error,fields,files)
    {
        console.log("解析完成");
        fs.rename(files.upload.path,"D:/test.png",function(err)
        {
            if(err)
            {
                fs.unlink("D:/test.png");
                fs.rename(files.upload.path,"D:/test.png")
            }
        }
    });

    response.writeHead(200,{"Content-Type":"text/plain"});
    response.write("接收到图片:<br/>");
    response.write("<img src='/show' />")
    response.end();
}
);

```

```

}

```

```

function show(response)
{
    console.log("请求处理函数： 正在处理 show");
    response.writeHead(200,{"Content-Type":"image/png"});
    //打开文件
    fs.createReadStream("D:/test.png").pipe(response);
}

```

```

exports.start = start;
exports.upload = upload;
exports.show = show;

```

```

router.js

```

```

function route(handle,pathname,response,request)
{
    console.log("路由请求:" + pathname);

    if (typeof handle[pathname] === "function")
    {
        handle[pathname](response,request);
    }
    else
    {
        console.log("路由请求没找到 " + pathname);
        response.writeHead(404,{"Content-Type":"text/plain"});

        response.write("404 Not Found");
        response.end();
    }
}

exports.route = route;

```

效果:

