七天学会 NodeJS #4

不了解网络编程的程序员不是好前端，而 **NodeJS** 恰好提供了一扇了解网络编程的窗口。通过 **NodeJS**，除了可以编写一些服务端程序来协助前端开发和测试外，还能够学习一些 **HTTP** 协议与 **Socket** 协议的相关知识，这些知识在优化前端性能和 排查前端故障时说不定能派上用场。本章将介绍与之相关的 **NodeJS** 内置模块。

开门红

NodeJS 本来的用途是编写高性能 **Web** 服务器。我们首先在这里重复一下官方文档里的例子，使用 **NodeJS** 内置的 **http** 模块简单实现一个 **HTTP** 服务器。

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



http.createServer(function (request, response) {

    response.writeHead(200, { 'Content-Type': 'text-plain' });

    response.end('Hello World\n');

}).listen(8124);
```

以上程序创建了一个 **HTTP** 服务器并监听**8124**端口，打开浏览器访问该端口 **http://127.0.0.1:8124/**就能够看到效果。

豆知识： 在***nix** 系统下，监听**1024**以下端口需要 **root** 权限。因此，如果想监听**80**或**443**端口的话，需要使用 **sudo** 命令启动程序。

API 走马观花

我们先大致看看 **NodeJS** 提供了哪些和网络操作有关的 **API**。这里并不逐一介绍每个 **API** 的使用方法，官方文档已经做得很好了。

HTTP

官方文档： **http://nodejs.org/api/http.html**

'http'模块提供两种使用方式：

作为服务端使用时，创建一个 **HTTP** 服务器，监听 **HTTP** 客户端请求并返回响应。

作为客户端使用时，发起一个 **HTTP** 客户端请求，获取服务端响应。

首先我们来看看服务端模式下如何工作。如开门红中的例子所示，首先需要使用**.createServer**

方法创建一个服务器，然后调用.listen方法监听端口。之后，每当来了一个客户端请求，创建服务器时传入的回调函数就被调用一次。可以看出，这是一种事件机制。

HTTP请求本质上是一个数据流，由请求头（headers）和请求体（body）组成。例如以下是一个完整的HTTP请求数据内容。

```
POST / HTTP/1.1

User-Agent: curl/7.26.0

Host: localhost

Accept: */*

Content-Length: 11

Content-Type: application/x-www-form-urlencoded


Hello World
```

可以看到，空行之上是请求头，之下是请求体。HTTP请求在发送给服务器时，可以认为是按照从头到尾的顺序一个字节一个字节地以数据流方式发送的。而http模块创建的HTTP服务器在接收到完整的请求头后，就会调用回调函数。在回调函数中，除了可以使用request对象访问请求头数据外，还能把request对象当作一个只读数据流来访问请求体数据。以下是一个例子。

```
http.createServer(function (request, response) {

    var body = [];


    console.log(request.method);

    console.log(request.headers);


    request.on('data', function (chunk) {

        body.push(chunk);

    });
```

```
    request.on('end', function () {
        body = Buffer.concat(body);
        console.log(body.toString());
    });
}).listen(80);
```

```
POST
{ 'user-agent': 'curl/7.26.0',
  host: 'localhost',
  accept: '*/*',
  'content-length': '11',
  'content-type': 'application/x-www-form-urlencoded' }
Hello World
```

HTTP 响应本质上也是一个数据流，同样由响应头（headers）和响应体（body）组成。例如以下是一个完整的 HTTP 请求数据内容。

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 11
Date: Tue, 05 Nov 2013 05:31:38 GMT
Connection: keep-alive

Hello World
```

在回调函数中，除了可以使用 response 对象来写入响应头数据外，还能把 response 对象当作一个只写数据流来写入响应体数据。例如在以下例子中，服务端原样将客户端请求的请求体

数据返回给客户端。

```
http.createServer(function (request, response) {

    response.writeHead(200, { 'Content-Type': 'text/plain' });



    request.on('data', function (chunk) {

        response.write(chunk);

    });



    request.on('end', function () {

        response.end();

    });

}).listen(80);
```

接下来我们看看客户端模式下如何工作。为了发起一个客户端 **HTTP** 请求，我们需要指定目标服务器的位置并发送请求头和请求体，以下示例演示了具体做法。

```
var options = {

        hostname: 'www.example.com',

        port: 80,

        path: '/upload',

        method: 'POST',

        headers: {

            'Content-Type': 'application/x-www-form-urlencoded'

        }

    };
```

```
var request = http.request(options, function (response) {});

request.write('Hello World');

request.end();
```

可以看到，`.request`方法创建了一个客户端，并指定请求目标和请求头数据。之后，就可以把`request`对象当作一个只写数据流来写入请求体数据和结束请求。另外，由于**HTTP**请求中`GET`请求是最常见的一种，并且不需要请求体，因此`http`模块也提供了以下便捷**API**。

```
http.get('http://www.example.com/', function (response) {});
```

当客户端发送请求并接收到完整的服务端响应头时，就会调用回调函数。在回调函数中，除了可以使用`response`对象访问响应头数据外，还能把`response`对象当作一个只读数据流来访问响应体数据。以下是一个例子。

```
http.get('http://www.example.com/', function (response) {
    var body = [];

    console.log(response.statusCode);
    console.log(response.headers);

    response.on('data', function (chunk) {
        body.push(chunk);
    });

    response.on('end', function () {
        body = Buffer.concat(body);
        console.log(body.toString());
    });
});
```

```
--------------------------------

200

{ 'content-type': 'text/html',

  server: 'Apache',

  'content-length': '801',

  date: 'Tue, 05 Nov 2013 06:08:41 GMT',

  connection: 'keep-alive' }

<!DOCTYPE html>

...
```

HTTPS

官方文档： http://nodejs.org/api/https.html

https 模块与 http 模块极为类似，区别在于 https 模块需要额外处理 SSL 证书。

在服务端模式下，创建一个 HTTPS 服务器的示例如下。

```
var options = {

        key: fs.readFileSync('./ssl/default.key'),

        cert: fs.readFileSync('./ssl/default.cer')

    };


var server = https.createServer(options, function (request, response) {

        // ...

    });
```

可以看到，与创建 HTTP 服务器相比，多了一个 options 对象，通过 key 和 cert 字段指定了 HTTPS 服务器使用的私钥和公钥。

另外，NodeJS 支持 SNI 技术，可以根据 HTTPS 客户端请求使用的域名动态使用不同的证书，

因此同一个 **HTTPS** 服务器可以使用多个域名提供服务。接着上例，可以使用以下方法为 **HTTPS** 服务器添加多组证书。

```
server.addContext('foo.com', {
    key: fs.readFileSync('./ssl/foo.com.key'),
    cert: fs.readFileSync('./ssl/foo.com.cer')
});

server.addContext('bar.com', {
    key: fs.readFileSync('./ssl/bar.com.key'),
    cert: fs.readFileSync('./ssl/bar.com.cer')
});
```

在客户端模式下，发起一个 **HTTPS** 客户端请求与 `http` 模块几乎相同，示例如下。

```
var options = {
        hostname: 'www.example.com',
        port: 443,
        path: '/',
        method: 'GET'
    };

var request = https.request(options, function (response) {});

request.end();
```

但如果目标服务器使用的 **SSL** 证书是自制的，不是从颁发机构购买的，默认情况下 `https` 模块会拒绝连接，提示说有证书安全问题。在 `options` 里加入 `rejectUnauthorized: false` 字段可以禁用对证书有效性的检查，从而允许 `https` 模块请求开发环境下使用自制证书的 **HTTPS** 服务器。

URL

官方文档： http://nodejs.org/api/url.html

处理 HTTP 请求时 url 模块使用率超高，因为该模块允许解析 URL、生成 URL，以及拼接 URL。首先我们来看看一个完整的 URL 的各组成部分。

```
                                    href
 -----------------------------------------------------------------
                                  host          path
                               --------------- ----------------------------
 http: // user:pass @ host.com : 8080 /p/a/t/h ?query=string #hash
 -----    ---------   --------   ---- -------- ------------ -----
protocol     auth     hostname   port pathname     search     hash
                                                ------------
                                                 query
```

我们可以使用.parse 方法来将一个 URL 字符串转换为 URL 对象，示例如下。

```
url.parse('http://user:pass@host.com:8080/p/a/t/h?query=string#hash');

/* =>

{ protocol: 'http:',

  auth: 'user:pass',

  host: 'host.com:8080',

  port: '8080',

  hostname: 'host.com',

  hash: '#hash',

  search: '?query=string',

  query: 'query=string',
```

```
  pathname: '/p/a/t/h',

  path: '/p/a/t/h?query=string',

  href: 'http://user:pass@host.com:8080/p/a/t/h?query=string#hash' }

*/
```

传给`.parse`方法的不一定要是一个完整的 **URL**，例如在 **HTTP** 服务器回调函数中，`request.url`不包含协议头和域名，但同样可以用`.parse`方法解析。

```
http.createServer(function (request, response) {

    var tmp = request.url; // => "/foo/bar?a=b"

    url.parse(tmp);

    /* =>

    { protocol: null,

      slashes: null,

      auth: null,

      host: null,

      port: null,

      hostname: null,

      hash: null,

      search: '?a=b',

      query: 'a=b',

      pathname: '/foo/bar',

      path: '/foo/bar?a=b',

      href: '/foo/bar?a=b' }

    */

}).listen(80);
```

.parse 方法还支持第二个和第三个布尔类型可选参数。第二个参数等于 true 时，该方法返回的 URL 对象中，query 字段不再是一个字符串，而是一个经过 querystring 模块转换后的参数对象。第三个参数等于 true 时，该方法可以正确解析不带协议头的 URL，例如 //www.example.com/foo/bar。

反过来，format 方法允许将一个 URL 对象转换为 URL 字符串，示例如下。

```
url.format({
    protocol: 'http:',
    host: 'www.example.com',
    pathname: '/p/a/t/h',
    search: 'query=string'
});
/* =>
'http://www.example.com/p/a/t/h?query=string'
*/
```

另外，.resolve 方法可以用于拼接 URL，示例如下。

```
url.resolve('http://www.example.com/foo/bar', '../baz');
/* =>
http://www.example.com/baz
*/
```

Query String

官方文档： http://nodejs.org/api/querystring.html

querystring 模块用于实现 URL 参数字符串与参数对象的互相转换，示例如下。

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge');
/* =>
{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }
*/
```

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' });

/* =>

'foo=bar&baz=qux&baz=quux&corge='

*/
```

Zlib

官方文档： http://nodejs.org/api/zlib.html

zlib 模块提供了数据压缩和解压的功能。当我们处理 HTTP 请求和响应时，可能需要用到这个模块。

首先我们看一个使用 zlib 模块压缩 HTTP 响应体数据的例子。这个例子中，判断了客户端是否支持 gzip，并在支持的情况下使用 zlib 模块返回 gzip 之后的响应体数据。

```
http.createServer(function (request, response) {

    var i = 1024,

        data = '';



    while (i--) {

        data += '.';

    }



    if ((request.headers['accept-encoding'] || '').indexOf('gzip') !== -1) {

        zlib.gzip(data, function (err, data) {

            response.writeHead(200, {

                'Content-Type': 'text/plain',

                'Content-Encoding': 'gzip'

            });
```

```
            response.end(data);
        });
    } else {
        response.writeHead(200, {
            'Content-Type': 'text/plain'
        });
        response.end(data);
    }
}).listen(80);
```

接着我们看一个使用 zlib 模块解压 HTTP 响应体数据的例子。这个例子中，判断了服务端响应是否使用 gzip 压缩，并在压缩的情况下使用 zlib 模块解压响应体数据。

```
var options = {
        hostname: 'www.example.com',
        port: 80,
        path: '/',
        method: 'GET',
        headers: {
            'Accept-Encoding': 'gzip, deflate'
        }
    };

http.request(options, function (response) {
    var body = [];
```

```
    response.on('data', function (chunk) {
        body.push(chunk);
    });

    response.on('end', function () {
        body = Buffer.concat(body);

        if (response.headers['content-encoding'] === 'gzip') {
            zlib.gunzip(body, function (err, data) {
                console.log(data.toString());
            });
        } else {
            console.log(data.toString());
        }
    });
}).end();
```

Net

官方文档： http://nodejs.org/api/net.html

net 模块可用于创建 Socket 服务器或 Socket 客户端。由于 Socket 在前端领域的使用范围还不是很广，这里先不涉及到 WebSocket 的介绍，仅仅简单演示一下如何从 Socket 层面来实现 HTTP 请求和响应。

首先我们来看一个使用 Socket 搭建一个很不严谨的 HTTP 服务器的例子。这个 HTTP 服务器不管收到啥请求，都固定返回相同的响应。

```
net.createServer(function (conn) {
    conn.on('data', function (data) {
        conn.write([
```

```
            'HTTP/1.1 200 OK',
            'Content-Type: text/plain',
            'Content-Length: 11',
            '',
            'Hello World'
        ].join('\n'));
    });
}).listen(80);
```

接着我们来看一个使用 **Socket** 发起 **HTTP** 客户端请求的例子。这个例子中，**Socket** 客户端在建立连接后发送了一个 **HTTP GET** 请求，并通过 **data** 事件监听函数来获取服务器响应。

```
var options = {
        port: 80,
        host: 'www.example.com'
    };

var client = net.connect(options, function () {
        client.write([
            'GET / HTTP/1.1',
            'User-Agent: curl/7.26.0',
            'Host: www.baidu.com',
            'Accept: */*',
            '',
            ''
        ].join('\n'));
```

```
    });


client.on('data', function (data) {
    console.log(data.toString());
    client.end();
});
```

灵机一点

使用 **NodeJS** 操作网络，特别是操作 **HTTP** 请求和响应时会遇到一些惊喜，这里对一些常见问题做解答。

问： 为什么通过 `headers` 对象访问到的 **HTTP** 请求头或响应头字段不是驼峰的？

答： 从规范上讲，**HTTP** 请求头和响应头字段都应该是驼峰的。但现实是残酷的，不是每个 **HTTP** 服务端或客户端程序都严格遵循规范，所以 **NodeJS** 在处理从 别的客户端或服务端收到的头字段时，都统一地转换为了小写字母格式，以便开发者能使用统一的方式来访问头字段，例如 `headers['content-length']`。

问： 为什么 `http` 模块创建的 **HTTP** 服务器返回的响应是 `chunked` 传输方式的？

答： 因为默认情况下，使用`.writeHead` 方法写入响应头后，允许使用`.write` 方法写入任意长度的响应体数据，并使用`.end` 方法结束一个响应。由于响应体数据长度不确定，因此 **NodeJS** 自动在响应头里添加了 `Transfer-Encoding: chunked` 字段，并采用 `chunked` 传输方式。但是当响应体数据长度确定时，可使用`.writeHead` 方法在响应头里加上 `Content-Length` 字段，这样做之后 **NodeJS** 就不会自动添加 `Transfer-Encoding` 字段和使用 `chunked` 传输方式。

问： 为什么使用 `http` 模块发起 **HTTP** 客户端请求时，有时候会发生 `socket hang up` 错误？

答： 发起客户端 **HTTP** 请求前需要先创建一个客户端。`http` 模块提供了一个全局客户端 `http.globalAgent`，可以让我们使用`.request` 或`.get` 方法时不用手动创建客户端。但是全局客户端默认只允许**5**个并发 **Socket** 连接，当某一个时刻 **HTTP** 客户端请求创建过多，超过这个数字时，就会发生 `socket hang up` 错误。解决方法也很简单，通过 `http.globalAgent.maxSockets` 属性把这个数字改大些即可。另外，`https` 模块遇到这个问题时也一样通过 `https.globalAgent.maxSockets` 属性来处理。

小结

本章介绍了使用 **NodeJS** 操作网络时需要的 **API** 以及一些坑回避技巧，总结起来有以下几点：

`http` 和 `https` 模块支持服务端模式和客户端模式两种使用方式。

request 和 response 对象除了用于读写头数据外，都可以当作数据流来操作。

url.parse 方法加上 request.url 属性是处理 HTTP 请求时的固定搭配。

使用 zlib 模块可以减少使用 HTTP 协议时的数据传输量。

通过 net 模块的 Socket 服务器与客户端可对 HTTP 协议做底层操作。

小心踩坑。