第5章 网络编程

Node.js开发实战教程

内容导航

Contents

5.1 TCP服务器与客户端

- TCP基础
- net模块提供的API
- 创建TCP服务器和客户端
- 接收和处理物联网数据

5.2 UDP服务器与客户端

5.3 HTTP服务器和客户端

5.4 使用WebSocket实现浏览器与服务器的实时通信

5.5 实战演练——构建实时聊天室

Section 00

【学习目标】

（1）了解net模块；

（2）编程构建TCP服务器与客户端。

Section 01

【TCP基础】

- TCP数据分组称为段（Segment）。TCP将来自应用层的数据进行分块并封装成TCP段进行发送。
- TCP连接的建立采用三次握手方法。
- Socket之间的连接过程包括以下3个步骤。
  （1）服务器监听。服务器端Socket处于等待连接的状态，实时监控网络状态。
  （2）客户端请求。客户端Socket提出连接请求，要连接的目标是服务器端的Socket。
  （3）连接确认。
- net核心模块用于创建TCP连接和服务，封装了大部分底层的类和方法。导入该模块：

```
const net = require('net');
```

Section 02

【net模块提供的API】

net.Server类

- 可直接用于创建net.Server对象。

  const server = new net.Server([options][, connectionlistener])

- net.Server类实现的事件
  - close：当服务器关闭时被触发。如果有连接存在，则直到所有连接结束才会触发这个事件。
  - connection：当一个新的连接建立时被触发，Socket是一个net.Socket的实例对象。
  - error：发生错误时被触发。
  - listening：当服务被绑定后调用server.listen()方法。
- net.Server类提供的方法
  - server.address()方法用于返回绑定的ip地址、地址族和服务端口。
  - server.close()方法使服务器停止接受建立新的连接并保持现有的连接。
  - server.listen()方法用来启动一个服务器来监听连接。

Section 02

【net模块提供的API】

net.createServer()方法

- 返回一个net.Server类的实例（对象）。
- 该方法可以创建一个TCP服务器，是创建net.Server对象的快捷方式。
- 用法：

```
net.createServer([options][, connectionlistener])
```

Section 02

【net模块提供的API】

net.Socket类

- 可以直接创建net.Socket对象。

```
const client = net.Socket([options])
```

- net.Socket类实现的事件
  - close：当Socket完全关闭时发出该事件。
  - connect：当一个Socket连接成功建立时触发该事件。
  - data：当接收到数据时触发该事件。
  - drain：当写入缓冲区变为空时触发该事件。
  - end：当Socket的另一端发送一个FIN包时触发该事件，从而结束Socket的可读端。
  - error：当错误发生时触发该事件。close事件也会紧接着该事件被触发。
  - ready：当Socket准备使用时触发该事件。connect事件发生后立即触发该事件。
  - timeout：当Socket超时时触发该事件。
- net.Socket类提供的方法
  - socket.connect()方法用于在指定的Socket上启动一个连接。
  - socket.write()方法用于在Socket上发送数据。

Section 02

【net模块提供的API】

net.createConnection()方法

- 返回启动连接的net.Socket对象。
- 当连接建立之后，在返回的Socket上将触发一个connect事件。
- 语法格式：
  - net.createConnection(options[, connectListener])
  - net.createConnection(port[, host][, connectListener])

Section 03

【创建TCP服务器和客户端】

▶▶▶ 创建TCP服务器

```
const net = require('net');
var clientNo = 0;  //使用编号标识每个客户端
const server = net.createServer((client) => {
  // connection 监听器
  clientNo++;  //有客户端连接时，编号自动加1
  console.log(clientNo+'号客户端已连接');
  client.on('end', () => {
    console.log(clientNo+'号客户端已断开连接');
  });
  client.write(clientNo+'号客户端，你好！\r\n');
  client.pipe(client);//通过管道操作将客户端发来的数据返回给客户端
  client.on('data', (data) => {
    console.log(clientNo+'号客户端发来的数据：' + data.toString());
  });
});
```

Section 03

【创建TCP服务器和客户端】

▸▸▸ 创建TCP服务器

```
server.on('error', (err) => {
  throw err;
});
server.listen(8234, () => {
  console.log('TCP服务器已启动');
});
```

Section 03

【创建TCP服务器和客户端】

▶▶▶ 创建TCP客户端

```
const net = require('net');
var client = net.Socket(); // 创建TCP客户端
// 设置连接的服务器
client.connect(8234, '127.0.0.1', ()=> {
    console.log('连接到服务器');
    client.write('我是一个TCP客户端');//向服务器发送数据
});
// 监听服务器传来的数据
client.on('data', (data) =>{
    console.log('服务器返回的数据：' + data.toString());
});
// 监听end事件
client.on('end', ()=> {
    console.log('数据结束');
});
```

Section 04

【接收和处理物联网数据】

▶▶▶ 接收和处理设备上传数据的服务器

```
const net = require('net');
const server = net.createServer((client) => {
  console.log('设备已连接');
  client.on('end', () => {
    console.log('设备已断开连接');
  });
  client.setEncoding('hex');
```

Section 04

【接收和处理物联网数据】

▶▶▶ 接收和处理设备上传数据的服务器

```
client.on('data', (data) => {
    const buf = Buffer.from(data);
    var str = buf.toString();
    console.log('设备发来的数据：' +str);
    var sixteen = str.substring(18,22);
    var ten = parseInt(sixteen,16);//将十六进制数转换为十进制数
    console.log('待转换十六进制数：' +sixteen);
    console.log('转换为十进制数：'+ ten );
  });
});
server.on('error', (err) => {
  throw err;
});
server.listen(8234, () => {
  console.log('服务器已启动');
});
```

Section 04

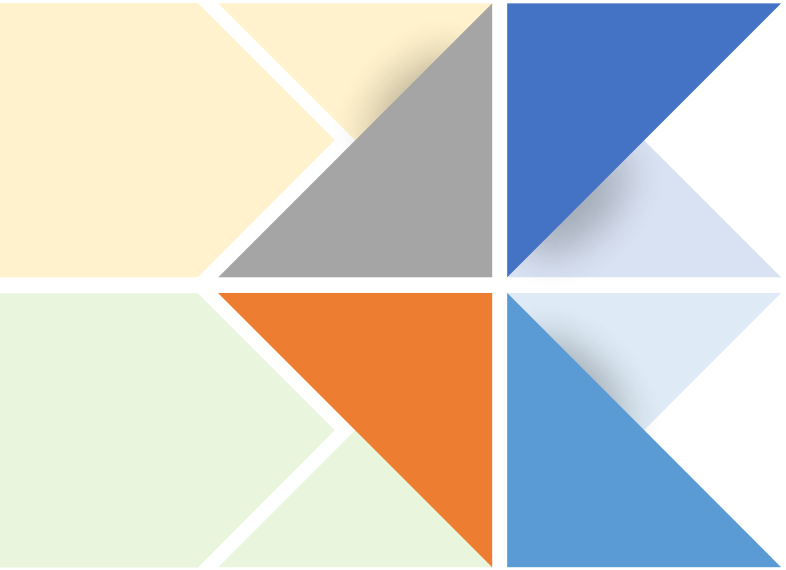
【接收和处理物联网数据】

▸▸▸ 模拟设备上传数据

```
const net = require('net');
var client = net.Socket(); // 创建TCP客户端来模拟设备
// 设置连接的服务器
client.connect(8234, '127.0.0.1',  ()=> {
  console.log('连接到服务器');
  client.write('000316003400000001012702520042 00','hex');//向服务器上传十六进制数据
});
// 监听服务器返回的数据
client.on('data', (data) =>{
  console.log('服务器返回的数据：' + data.toString());
});
// 监听end事件
client.on('end',  ()=> {
    console.log('数据通信结束');
});
```

内容导航

Contents



5.1 TCP服务器与客户端

5.2 UDP服务器与客户端

- UDP基础
- dgram模块提供的API
- 创建UDP服务器和客户端
- 通过UDP实现文件上传

5.3 HTTP服务器和客户端

5.4 使用WebSocket实现浏览器与服务器的实时通信

5.5 实战演练——构建实时聊天室

Section 00

【学习目标】

（1）了解dgram模块；
（2）编程构建UDP服务器与客户端。

Section 01

【UDP基础】

- UDP提供的是一种无连接的、不可靠的数据传输方式。
- UDP的不可靠特性不代表它不可靠，其可靠性保证和流控制可以由UDP用户（即应用程序）决定。
- UDP传输给IP的数据单元称作UDP数据报（Datagram）。
- UDP使用端口号为不同的应用保留其各自的数据传输通道。UDP使用Socket，只不过是无连接的数据报Socket。
- dgram核心模块用于实现UDP通信。导入该模块：

```
const dgram = require('dgram');
```

Section 02

【dgram模块提供的API】

dgram.Socket类

- dgram.Socket类提供实现UDP应用的基本框架。dgram.Socket对象是一个封装了数据报功能的事件触发器。dgram.Socket实例由dgram.createSocket()方法创建。
- dgram.Socket类实现的事件
  - close事件：使用close()方法关闭一个Socket之后触发该事件。
  - error：发生任何错误都会触发该事件。
  - listening：当一个Socket开始监听数据报信息时触发该事件。
  - message：当有新的数据报被Socket接收时触发该事件。
- dgram.Socket类提供的方法
  - socket.bind()方法用于设置Socket在指定的端口和地址上监听数据报信息。
  - socket.send()方法用于在Socket上发送一个数据报。

Section 02

【dgram模块提供的API】

dgram.createSocket()方法

- 该方法用于创建dgram.Socket对象。一旦创建了Socket，调用socket.bind()方法会指示Socket开始监听数据报消息。
- 基本用法

  dgram.createSocket(options[, callback])
- 另一种用法：创建一个特定类型的dgram.Socket对象

  dgram.createSocket(type[, callback])

Section 03

【创建UDP服务器和客户端】

▶▶▶ UDP服务器

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');//创建dgram.Socket对象
server.on('error', (err) => {
  console.log(`服务器异常：\n${err.stack}`);
  server.close();
});
server.on('message', (msg, rinfo) => {
  var strmsg = msg.toString();
  server.send(strmsg, rinfo.port, rinfo.address); //将接收到的消息返回给客户端
  console.log(`服务器接收到来自 ${rinfo.address}:${rinfo.port} 的 ${strmsg}`);
});
server.on('listening', () => {
  const address = server.address();
  console.log(`服务器监听 ${address.address}:${address.port}`);
});
server.bind(41234);
```

Section 03

【创建UDP服务器和客户端】

▸▸▸ UDP客户端

```
const dgram = require('dgram');
const message = Buffer.from('你好！我是一个UDP客户端');
const client = dgram.createSocket('udp4');
client.on('close',()=>{
    console.log('socket已关闭');
});
client.on('error',(err)=>{
    console.log(err);
});
client.on('message',(msg,rinfo)=>{
    if(msg=='exit') client.close();
    var strmsg = msg.toString();
    console.log(`接收到来自：${rinfo.address}:${rinfo.port} 的消息：  ${msg}`);
});
client.send(message, 41234, 'localhost', (err) => {//向服务器发送消息
    if(err) client.close();
});
```

Section 04

【通过UDP实现文件上传】

▶▶▶ 文件上传服务器

```
const dgram = require('dgram');
const fs = require('fs');
var port = 41234;
var server = dgram.createSocket('udp4');
var writeStream = fs.createWriteStream('upfile');//创建一个可写流
server.on('error', (err) => {
    console.log(`服务器异常：\n${err.stack}`);
    server.close();
  });
server.on('message', function(msg, rinfo) {//message事件触发
    process.stdout.write(msg.toString());//将接收到的数据输出到终端
    writeStream.write(msg.toString()); //将接收到的数据写入可写流
});
server.on('listening', function() {
    console.log('文件传输服务器已准备好：', server.address());
});
server.bind(port);
```

Section 04

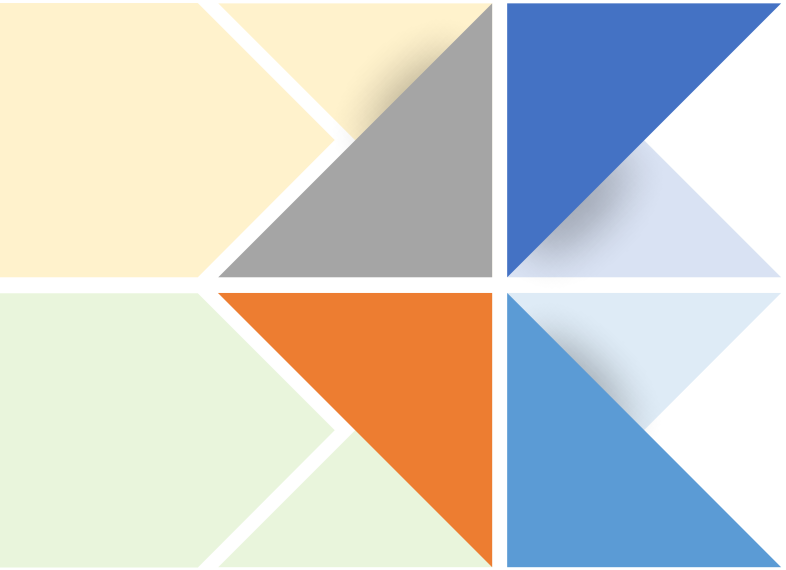
【通过UDP实现文件上传】

▶▶▶ 文件上传客户端

```
const dgram = require('dgram');
const fs = require('fs');
var remoteIP = '127.0.0.1';
var port = 41234;
var defaultSize = 16;//流读取的默认块大小
var readStream = fs.createReadStream(__filename);//从当前文件创建一个可读流
var client = dgram.createSocket('udp4');
readStream.on('readable', function() {//可读流准备好之后就开始发送数据
    sendData();
});
function sendData() {
  var message = readStream.read(defaultSize);
  if (!message) {
    return client.unref(); //完成文件传输之后，关闭Socket
  }
  client.send(message,0,message.length, port, remoteIP, function (err, bytes) {
    sendData();
  });
}
```

内容导航

Contents



Section 00 【学习目标】

（1）了解http模块；
（2）编程构建HTTP服务器与客户端。

Section 01 【HTTP基础】

- HTTP是一种通用的、无状态的、与传输数据无关的应用层协议。
- 客户端与服务器之间的HTTP交互过程
  （1）客户端与服务器建立TCP连接。
  （2）客户端向服务器发送HTTP请求报文。
  （3）服务器向客户端返回HTTP响应报文。
  （4）关闭HTTP连接。
- http核心模块用于实现HTTP通信。导入该模块：

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

Section 02

【实现HTTP服务器】

http.Server类

- http.Server类提供了实现HTTP服务器的基本框架。
- http.Server类实现的事件
  - request：当有请求时会触发该事件，提供两个参数request和response，分别为http.IncomingMessage对象和http.ServerResponse对象，表示请求和响应的信息。
  - connect：客户端请求HTTP的CONNECT方法时触发该事件。
  - connection：当TCP建立连接时触发该事件，提供参数socket通常表示net.Socket对象。
  - close：当服务器关闭时触发该事件。
  - clientError：如果客户端连接触发error事件，则会在此处转发。
- http.Server类提供的方法
  - server.listen() 用于启动HTTP服务器监听连接。
  - server.close()用于停止服务器接受新连接。
- http.Server类主要实现的功能
  - 基于TCP连接建立一个网络监听器。
  - 监听自身的request请求事件。

Section 02

【实现HTTP服务器】

▸▸▸ http.IncomingMessage类——传入的信息

- http.IncomingMessage对象（该类实例）由http.Server或http.ClientRequest创建。
- http.IncomingMessage类实现的事件
  - aborted：当请求中止时被触发。
  - close：表明底层连接已关闭。
- http.IncomingMessag属性

属性	说明
complete	收到并成功解析的完整HTTP消息
headers	请求或响应的消息头对象，包括消息头的名称和值的键值对。注意消息头的名称都是小写的
rawHeaders	原始请求头/响应头的列表，与接收到的完全一致。
method	HTTP请求方法，用字符串表示，如'GET'、'POST'
url	请求的URL字符串，仅包含实际HTTP请求中存在的URL。
socket	与连接关联的net.Socket对象
statusCode	3位HTTP响应状态码，例如404。仅对从http.ClientRequest对象获取的响应有效
statusMessage	HTTP响应状态消息（原因短语），例如OK或Internal Server Error

Section 02

【实现HTTP服务器】

▸▸▸ http.ServerResponse类——响应

- 指定要发送到客户端的响应。
- http.ServerResponse类实现的事件
  - close：调用response.end()方法，或者能够刷新之前已终止的底层连接。
  - finish：响应发送后触发该事件。
- 关于响应头的额外方法
  - response.setHeader(name, value)：设置一个特定的响应头。
  - getHeader(name)：获取已在响应中设置的某个响应头。
  - removeHeader(name)：移除已在响应中设置的某个响应头。
  - response.addTrailers(headers)：将HTTP尾部响应头添加到响应中。
  - response.writeHead(statusCode,[reasonPhrase],[headers])：将某个响应头写入请求。

Section 02

【实现HTTP服务器】

▸▸▸ http.ServerResponse类——响应

➢ 关于响应体的方法

- response.write(data,[encoding])：发送响应体。
- response.writeContinue()：向客户端发送HTTP/1.1 100 Continue消息，表示应发送请求体。
- response.end([data][,encoding][,callback])：结束响应，向服务器表明已发送所有的响应头和响应体。

➢ 属性

- response.statusCode：刷新响应头时将发送到客户端的状态码。
- response.statusMessage：刷新响应头时将发送到客户端的状态消息。

Section 02

【实现HTTP服务器】

▸▸▸ http.createServer()方法

- 要启动HTTP服务器，需要使用http.createServer()方法创建一个http.Server对象。
- 该方法的用法：

```
http.createServer([options][, requestlistener])
```

Section 02

【实现HTTP服务器】

▶▶▶ 创建HTTP服务器

```
const http = require('http');
const server = http.createServer((req,res)=>{
    console.log('Web客户端URL：'+req.url);
    //设置响应头信息
    res.writeHead(200,{'Content-Type':'text/html;charset=utf-8'});
    res.write('Hello!<br>');
    res.end('响应完毕\n');
}).listen(8080,()=>{
    console.log('服务器正在8080端口上监听！');
});
```

Section 03

【实现HTTP客户端】

http.ClientRequest类

- http.ClientRequest类提供了实现HTTP客户端的基本框架。
- 通过http.request()方法创建并返回一个http.ClientRequest对象，作为HTTP客户端，启动、监控和处理来自服务器的响应。
- 实现的事件
  - abort：当请求被客户端中止时被触发。
  - connect：每次服务器使用CONNECT方法响应请求时都会触发该事件。
  - continue：当服务器发送“100 Continue”HTTP响应时被触发。
  - response：当收到此请求的响应时被触发。此事件仅触发一次。
  - socket：将Socket分配给此请求后被触发。
  - timeout：当底层Socket因处于不活动状态而超时时被触发。

Section 03

【实现HTTP客户端】

▸▸▸ http.ClientRequest类

➢ 用于请求头的方法

- request.setHeader(name, value)：设置一个特定的请求头。
- request.getHeader(name)：读取请求中的一个请求头。
- request.flushHeaders()：刷新请求头。

➢ 用于请求体的方法

- request.write(chunk[, encoding][, callback])#：发送一个请求体的数据块。
- request.end([data[, encoding]][, callback])：完成发送请求。
- request.abort()：终止当前的请求。
- request.setTimeout(timeout,[callback])：为请求设置Socket超时时间。

Section 03

【实现HTTP客户端】

http.request()方法

- 构建一个HTTP客户端，需使用http.request()方法创建一个ClientRequest对象。
- 该方法的两种用法：

```
http.request(options[, callback])
http.request(url[, options][, callback])
```

Section 03

【实现HTTP客户端】

▸▸▸ 创建HTTP客户端

```
const http=require('http');
const querystring = require('querystring');
const postData = querystring.stringify({'msg': '你好！我是HTTP客户端'});
const options = {
  hostname: '127.0.0.1',
  port: 8080,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded;charset=utf-8',
    'Content-Length': Buffer.byteLength(postData)
  }
};
```

Section 03

【实现HTTP客户端】

▶▶▶ 创建HTTP客户端

```
const req = http.request(options, (res) => {
  console.log(`状态码: ${res.statusCode}`);
  console.log(`响应头: ${JSON.stringify(res.headers)}`);
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(`响应体: ${chunk}`);
  });
  res.on('end', () => {
    console.log('响应中已无数据');
  });
});
req.on('error', (e) => {
  console.error(`请求遇到问题: ${e.message}`);
});
// 将数据写入请求体
req.write(postData);
req.end();
```

Section 04

【HTTP服务器获取并解析请求内容】

▸▸▸ 获取并解析GET请求内容

- GET请求服务器

```
const http = require('http');
const url  = require('url');
var server = http.createServer((req,res)=>{
    let reqUrl = decodeURI(req.url);//对url进行解码（url会对中文进行编码）
    reqUrl = url.parse(reqUrl);//解析URL字符串并创建一个URL对象
    console.log(reqUrl);
    res.end('提交成功！');//回应客户端
}).listen(8080);
```

Section 04

【HTTP服务器获取并解析请求内容】

▸▸▸ 获取并解析GET请求内容

➢ GET请求客户端

```
const http=require('http');
//注意使用encodeURI()方法对含有中文字符的URL进行编码
var options = {
    hostname: '127.0.0.1',
    port: 8080,
    path: encodeURI('/index.html?page=12&stats=打开')
};
http.get(options, (res) => {
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(chunk.toString());
  });
}).on('error', (e) => {
  console.error(`出现错误: ${e.message}`);
});
```

Section 04

【HTTP服务器获取并解析请求内容】

▸▸▸ 获取并解析POST请求内容

➢ 服务器接收POST请求内容的流程

（1）服务器收到数据时，通过监听request对象的data事件来获取数据。

（2）对于多次发送的情况，每收到一部分数据，request对象的data事件会被触发一次，同时通过回调函数可以获取该部分的数据。完整的数据需要服务器端拼接。

（3）当接收数据完毕之后，会执行request的on事件。

Section 04

【HTTP服务器获取并解析请求内容】

▸▸▸ 获取并解析POST请求内容

- POST请求服务器

```
const http = require('http');
const querystring = require('querystring');
var server = http.createServer((req,res)=>{
    let data = '';//创建空字符串用于累加获取的数据
    req.on('data',(chunk)=>{
        data += chunk; //chunk默认是一个二进制数据，与字符串拼接时其会自动转换为字符串
    });
    req.on('end',()=>{
        data = decodeURI(data);//对url进行解码（url会对中文进行编码）
        console.log(data);
        let dataObj = querystring.parse(data); //使用querystring将数据转换为对象
        console.log(dataObj);
        res.end('提交成功！');//回应客户端
    });
}).listen(8080);
```

Section 04

【HTTP服务器获取并解析请求内容】

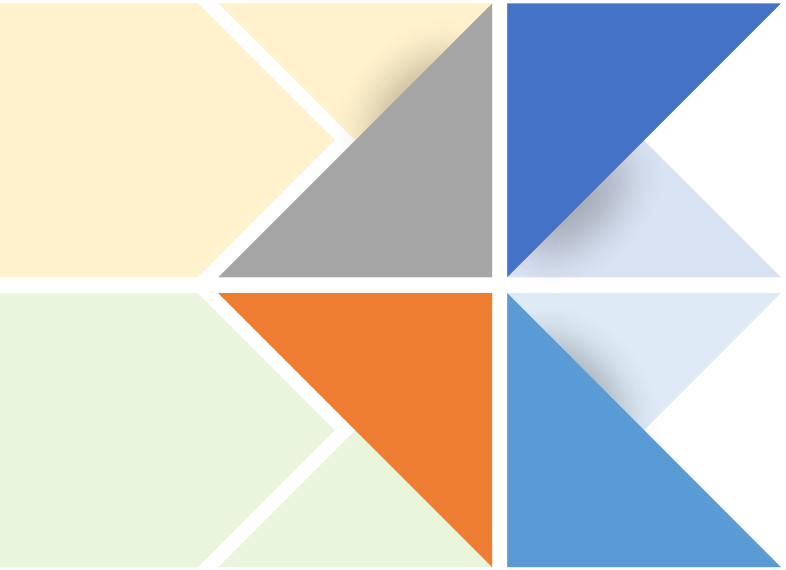
▸▸▸ 获取并解析POST请求内容

- POST请求客户端

```
const http=require('http');
const querystring = require('querystring');
//将要提交的数据转换为查询字符串
const postData = querystring.stringify({
    'name': 'Gates',
    'title': '博士'
});
const options = {
  hostname: '127.0.0.1',
  port: 8080,
  method: 'POST',
};
const req = http.request(options, (res) => {
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(chunk.toString());
  });
});
req.write(postData);// 将数据写入请求体
req.end();
```

内容导航

Contents



Section 00 【学习目标】

（1）了解WebSocket协议；
（2）编程实现WebSocket服务器与客户端。

Section 01 【WebSocket简介】

▸▸▸ 为什么需要WebSocket协议

- HTTP存在的问题
  - 每次客户端和服务器端的交互都是一次HTTP的请求和应答的过程，增加了每次传输的数据量，浪费带宽资源。
  - 不是真正的实时技术，只是模拟实时的效果，轮询会造成同步延迟。
  - 编程比较复杂，尤其是要模拟比较真实的实时效果时。
- WebSocket协议突破HTTP的局限性，节省服务器资源和带宽，解决两端的实时数据传输问题。

Section 01

【WebSocket简介】

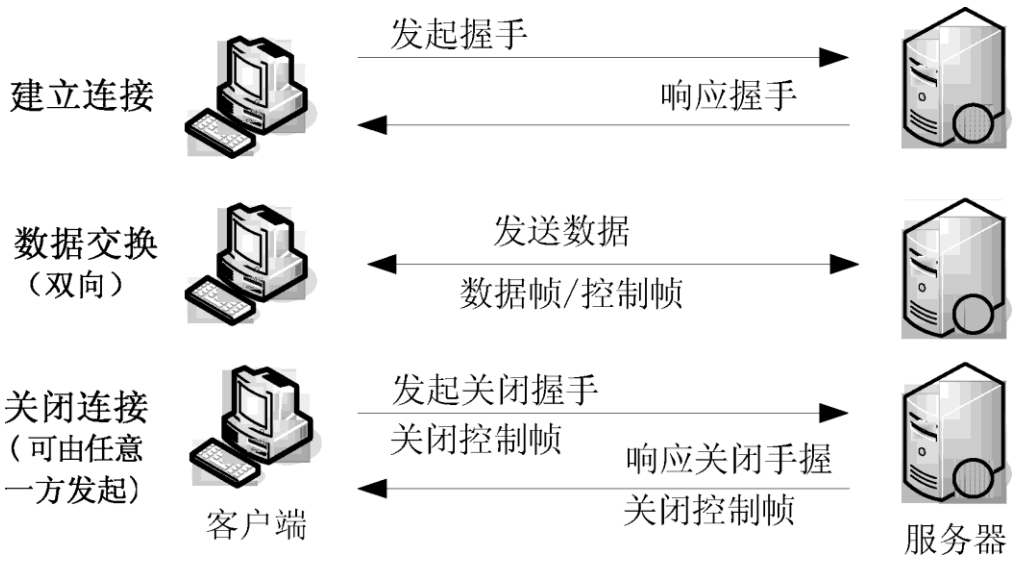
▸▸▸ WebSocket协议的实现机制

（1）由客户端发起握手，建立连接阶段必须依赖HTTP进行一次握手。两端之间的WebSocket连接建立的同时服务器完成了协议升级，由HTTP升级为WebSocket。连接会一直保持，直到客户端或者服务器任何一方主动关闭。

（2）进入数据交换阶段，客户端与服务端可以互相主动发送消息。此阶段数据直接通过TCP通道传输，不再依赖HTTP。WebSocket的数据传输是以帧（Frame）的形式传输的。

（3）关闭连接，可以由任意一端发起关闭连接的命令。

WebSocket通信过程



Section 01

【WebSocket简介】

▸▸▸ WebSocket协议的应用场合

- 实时通信：聊天应用。
- 实时展示和分析：典型的有实时计数器、图表、日志客户端等。
- 文档协同：允许多个用户同时编辑一个文档，且用户能够看到每个用户做出的修改。

WebSocket协议不适合那些不支持HTML5或对HTML5支持不够充分的浏览器。

Section 02

【使用Node.js实现WebSocket服务器和客户端】

▸▸▸ 实现WebSocket服务器

- WebSocket库：ws、WebSocket-Node、faye-websocket-node和socket.io。
- 创建一个WebSocket服务器实例

  new WebSocket.Server(options[, callback])
- WebSocket服务器内置事件
  - close：服务器关闭时被触发。
  - connection：成功握手连接时触发。
  - error：发生错误时被触发，可注入一个Error对象。
  - headers：握手前被触发，允许在发送HTTP头之前检查和修改标题。
  - listening：绑定端口时被触发。

Section 02

【使用Node.js实现WebSocket服务器和客户端】

▸▸▸ 实现WebSocket服务器

➢ 示例

```
const WebSocket = require('ws');  //加载ws模块
// 创建 Websocket服务器并在8080端口监听
const wss = new WebSocket.Server({ port: 8080 });
console.log('服务器正在8080端口上监听！');
// 当有客户端连接到服务器
wss.on('connection', function connection(ws) {
// 通过ws（客户端）对象获取客户端发送的信息
  ws.on('message', function incoming(message) {
    console.log('收到: %s', message);
    //主动推送信息给客户端
    ws.send(message);
  });
});
```

Section 02

【使用Node.js实现WebSocket服务器和客户端】

▸▸▸ 实现WebSocket客户端

- 创建一个WebSocket客户端实例

  new WebSocket(address[, protocols][, options])

- WebSocket客户端内置事件
  - close：连接关闭时被触发，有两个参数code（状态码）和reason（原因）。
  - error：发生错误时被触发，有一个参数error（错误）。
  - message：接收到服务器消息时被触发，有一个参数data，表示接收到的数据，类型可以是字符串、Buffer、ArrayBuffer。
  - open：连接建立时被触发。

Section 02

【使用Node.js实现WebSocket服务器和客户端】

▸▸▸ 实现WebSocket客户端

➢ 示例

```
const WebSocket = require('ws');
const ws = new WebSocket('ws://localhost:8080');
ws.on('open', function open() {
  console.log('已连接上服务器');
  ws.send(Date.now());
});
ws.on('close', function close() {
  console.log('连接已断开');
});
ws.on('message', function incoming(data) {
  console.log(`往返时延: ${Date.now() - data} ms`);
  setTimeout(function timeout() {
    ws.send(Date.now());
  }, 1000);
});
```

Section 03 【浏览器客户端】

▸▸▸ 实现WebSocket客户端

- 支持HTML5的浏览器都可作为WebSocket客户端。
- 将网页作为WebSocket客户端的示例

```
<script type="text/javascript">
    var ws = new WebSocket("ws://127.0.0.1:8080");
    ws.onopen = function(){
        console.log('websocket open');
        document.getElementById("stat").innerHTML = "已连接上";
    }
    ws.onclose = function(){
        console.log('websocket close');
        document.getElementById("stat").innerHTML = "连接断开";
    }
    ws.onmessage = function(event){
        console.log(event.data);
        document.getElementById("recvTxt").value = event.data;
    }
    document.getElementById("sendBtn").onclick = function(){
        var txt = document.getElementById("sendTxt").value;
        ws.send(txt);
    }
</script>
```

Section 04 【Socket.IO】

▸▸▸ 概述

- Socket.IO是一个支持客户端与服务器之间实时、双向、基于事件的通信的库。
- Socket.IO包括Node.js服务器API和JavaScript客户端库。
- Socket.IO并非WebSocket的完全实现。
- Socket.IO不仅简化了API接口，使得操作更容易，而且让那些不支持WebSocket协议的浏览器会将WebSocket连接自动降为Ajax连接，最大限度地保证了兼容性。
- Socket.IO的目标是统一通信机制，使所有浏览器和移动设备都可以进行实时通信，为开发者提供客户端与服务器端一致的编程体验。

▸▸▸ 安装Socket.IO

- Socket.IO服务器API需要安装socket.io包

```
npm install --save socket.io
```

- JavaScript客户端默认由服务器的/socket.io/socket.io.js提供。也可以通过CDN来引用。
- 如果使用Node.js编译，则可以安装socket.io-client包

```
npm install --save socket.io-client
```

Section 04 【Socket.IO】

▸▸▸ Socket.IO服务器

- Socket.IO服务器必须绑定一个http.Server实例。

```
const server = require('http').createServer();
const io = require('socket.io')(server);
server.listen(8000);
```

- 在Express框架中的用法

```
const app = require('express')();
const server = require('http').Server(app);
const io = require('socket.io')(server);
server.listen(8000);
```

Section 04 【Socket.IO】

▸▸▸ Socket.IO客户端

- 客户端通常使用的是标准化的JavaScript库，需对外公开io这一命名空间。

```
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io('http://localhost');
 socket.on('news', function (data) {
   console.log(data);
   socket.emit('my other event', { my: 'data' });
 });
</script>
```

Section 04 【Socket.IO】

▶▶▶ 触发和接收事件

```
// 这里是隐式绑定http.Server，io(<port>)将自动创建一个HTTP服务器
var io = require('socket.io')(8000);
io.on('connection', function (socket) {
  io.emit('this', { will: 'be received by everyone'});//触发this事件发送消息
  socket.on('primsg', function (from, msg) {//接收由primsg事件发送的消息
    console.log('I received a private message by ', from, ' saying ', msg);
  });
  socket.on('disconnect', function () {
    io.emit('user disconnected');
  });
});
```

Section 04 【Socket.IO】

▶▶▶ 发送和接收数据

- 服务器（app.js）代码

```
var io = require('socket.io')(80);
io.on('connection', function (socket) {
  socket.on('ferret', function (name, fn) {//传递一个fn函数
    fn('woot');//指定用于响应的确认信息
  });
});
```

- 客户端（index.html）代码

```
<script>
  var socket = io(); //没有任何参数的io()会自动发现HTTP服务
  socket.on('connect', function () {
  //可以避免监听connect事件，而是直接监听ferret事件
    socket.emit('ferret', 'tobi', function (data) {
      console.log(data); // 数据将为'woot'
    });
  });
</script>
```

Section 04 【Socket.IO】

▶▶▶ 广播消息

```
var io = require('socket.io')(8000);
io.on('connection', function (socket) {
  socket.broadcast.emit('user connected');
});
```

Section 04 【Socket.IO】

▸▸▸ 命名空间

- 命名空间可以为Socket指派不同的端点或路径，有助于最小化TCP连接的数量，从而通过引入通信通道之间的分离在应用中进行隔离。
- 默认的命名空间称为/。以下两个语句将事件发送到连接到/路径的所有Socket：

```
io.sockets.emit('hi', 'everyone');
io.emit('hi', 'everyone'); // 短格式
```

- 示例：一个connection事件发送给每个命名空间，每个Socket实例接收该事件发送的消息和数据。

```
io.on('connection', function(socket){
  socket.on('disconnect', function(){ });
});
```

- 要设置自定义名称空间，必须在服务器端调用of()方法：

```
const nsp = io.of('/my-namespace');
nsp.on('connection', function(socket){
  console.log('someone connected');
});
nsp.emit('hi', 'everyone!');
```

Section 04 【Socket.IO】

▸▸▸ 命名空间

➢ 客户端通知Socket.IO客户端连接到该命名空间：

```
const socket = io('/my-namespace');
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io('http://localhost');
  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'data' });
  });
</script>
```

Section 04 【Socket.IO】

▸▸▸ 房间

- 每个命名空间中可以定义多个频道，每个频道就是一个房间（Room）。
- 使用方法join()加入一个房间：

```
io.on('connection', function(socket){
  socket.join('someroom');
});
```

- 广播或发出事件时也可以简单地使用to()或in()方法，这两个方法的功能相同：

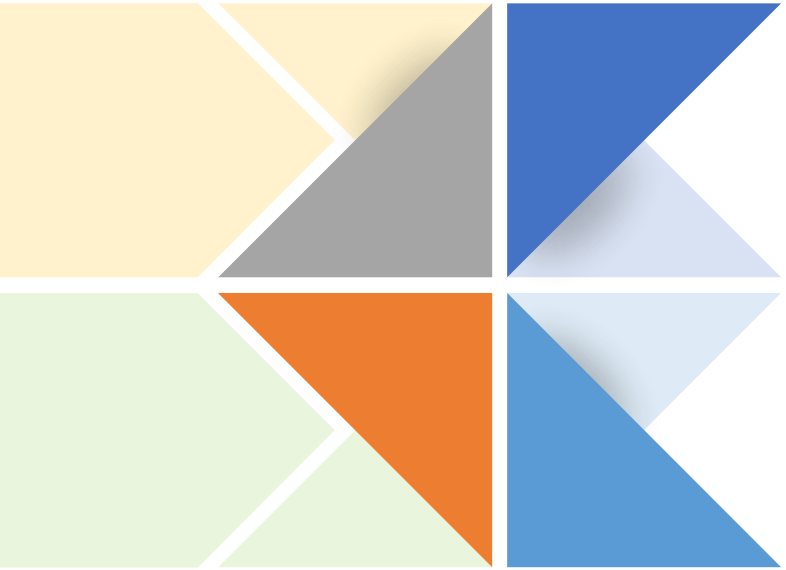
```
io.to('someroom').emit('someevent');
```

- 使用leave()方法离开房间，用法同join()方法。
- Socket.IO中的每个Socket由一个随机的唯一的标识符Socket#id标识。

```
io.on('connection', function(socket){
  socket.on('say to someone', function(id, msg){
    socket.broadcast.to(id).emit('my message', msg);
  });
});
```

内容导航

Contents



Section 00

【学习目标】

（1）掌握Socket.IO的基本使用；

（2）使用Socket.IO编写实时聊天程序。

Section 01

【准备Web框架】

▶▶▶ 创建项目

```
{
  "name": "chatapp",
  "version": "1.0.0",
  "description": "socket.io app",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Section 01

【准备Web框架】

▸▸▸ 简单的Web应用

```
const app = require('express')();
const server = require('http').Server(app);
app.get('/', function(req, res){
  res.send('<h1>Hello world</h1>');
});
server.listen(8000, function(){
  console.log('监听端口 *:8000');
});
```

使用浏览器访问进行实测

Section 01

【准备Web框架】

▸▸▸ 发布HTML网页

- index.html网页文件

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
   <section id="input_zone">
     <input id="message" class="vertical-align"
type="text" />
     <button id="send_message" class="vertical-
align" type="button">发送</button>
   </section>
  </body>
</html>
```

- index.js文件中改用app.sendFile()方法向客户端发送网页文件

```
app.get('/', function(req, res){
   res.sendFile(__dirname + '/index.html');
});
```

访问新的网页

Section 02

【编写服务器端程序】

```
const express = require('express');
const app = express();
const server = require('http').Server(app);
//通过中间件函数托管静态文件
app.use(express.static('static'));
//显示网页客户端文件index.html
app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});
server.listen(8000, function(){
  console.log('监听端口 *:8000');
});
//创建socket.io实例
const io = require("socket.io")(server);
```

Section 02

【编写服务器端程序】

```
//当有连接打开时
io.on('connection', (socket) => {
    console.log('有新用户接入！');
    //设置默认用户名
    socket.username = "匿名";
    //监听到change_username事件时，实时修改用户名
    socket.on('change_username', (data) => {
        socket.username = data.username;
    });
    //监听到new_message事件时，广播客户端发送的新消息
    socket.on('new_message', (data) => {
        io.sockets.emit('new_message', {message : data.message, username : socket.
username});
    });
});
```

Section 03

【编写客户端程序】

```
<script>
  $(function(){
  //建立连接
  var socket = io.connect('http://localhost:8000');
  //按钮和输入区
  var message = $("#message");
  var username = $("#username");
  var send_message = $("#send_message");
  var send_username = $("#send_username");
  var chatroom = $("#chatroom");
  //发出new_message事件，发送消息
  send_message.click(function(){
    socket.emit('new_message', {message : message.val()})
  });
```

Section 03

【编写客户端程序】

```
//监听new_message事件
    socket.on("new_message", (data) => {
      message.val('');
      chatroom.append("<p class='message'>" + data.username + ": " + data.message +
"</p>");
    });
    //发出change_username事件，发送用户名信息
    send_username.click(function(){
      socket.emit('change_username', {username : username.val()})
    });
  });
  </script>
```

Section 04

【测试】

本章小结

Summary

Node.js提供的net、dgram和http模块分别用来实现基本的TCP、UDP和HTTP应用程序，属于比较低层次的，主要提供简单的后端服务，如果要实现复杂的应用，则工作量相当大，因此实际的应用程序开发大都采用更高层次的模块或框架。如http模块不提供处理路由、cookie、缓存等的调用，而Express有助于快速实现完整的Web服务。Node.js的核心模块中关于网络的还有dns、http2、https。WebSocket协议解决了浏览器与服务器之间的实时通信问题，而Socket.IO又是类WebSocket的解决方案，目标是使所有浏览器和移动设备都可以进行实时通信。

Thank you