link: null
title: 珠峰架构师成长计划
description: 在Node.js中，提供了net模块用来实现TCP服务器和客户端的通信。
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=89 sentences=284, words=1663

1. TCP

在Node.js中，提供了net模块用来实现TCP服务器和客户端的通信。

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

- options.allowHalfOpen 是否允许单方面连接,默认值为false
- connectionListener参数用于指定当客户端与服务器建立连接时所要调用的回调函数，回调中有一个参数socket,指的是TCP服务器监听的socket端口对象

也可以通过监听connection事件的方式来指定监听函数

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

可以使用listen方法通知服务器开始监听客户端的连接

```
server.listen(port,[host],[backlog],[callback])
```

- port 必须指定的端口号
- host 指定需要监听的IP地址或主机名，如果省略的话服务器将监听来自于任何客户端的连接
- backlog指定位于等待队列中的客户端连接的最大数量，默认值为511

```
server.on('listening',function(){});
```

```
let net = require('net');
let server = net.createServer(function(socket){
  console.log('客户端已连接');
});
server.listen(8080,'localhost',function(){
    console.log('服务器开始监听');
});
```

```
server.address()
```

- port 端口号
- address TCP服务器监听的地址
- family 协议的版本

查看当前与TCP服务器建立连接的客户端的连接数量以及设置最大连接数量

```
server.getConnections(callback);
 server.maxConnections = 2;
```

使用close方法可以显式拒绝所有的客户端的连接请求,当所有已连接的客户端关闭后服务器会自动关闭,并触发服务器的close事件。

```
server.close();
server.on('close',callback);
```

net.Socket代表一个socket端口对象,它是一个可读可写流。

```
let net = require('net');
let util = require('util');
let server = net.createServer(function(socket){
  server.getConnections((err,count)=>{
      server.maxConnections = 1;
      console.log('最大连接数量%d,当前连接数量%d',server.maxConnections,count);
  });
  let address = socket.address();
  console.log('客户端地址 %s',util.inspect(address));
});
```

```
let server = net.createServer(function (socket) {
    socket.setEncoding('utf8');
    socket.on('data', function (data) {
        console.log('本次收到的内容为%s,累计收到的字节数是%d', data, socket.bytesRead);
    });
});
```

```
let server = net.createServer(function (socket) {
    socket.on('end', function () {
        console.log('客户端已经关闭');
    });
});
```

pipe方法可以将客户端发送的数据写到文件或其它目标中。

```
socket.pipe(destinatin,[options]);
```

- options.end 设置为false时当客户端结束写操作或关闭后并不会关闭目标对象，还可以继续写入数据

```
let net = require('net');
let path = require('path');
let ws = require('fs').createWriteStream(path.resolve(__dirname, 'msg.txt'));
let server = net.createServer(function (socket) {
    socket.on('data', function (data) {
        console.log(data);
    });
    socket.pipe(ws, { end: false });
    socket.on('end', function () {
        ws.end('over', function () {
            socket.unpipe(ws);
        });
    });
});
```

```
const net = require('net');
const path = require('path');
let file = require('fs').createWriteStream(path.join(__dirname, 'msg.txt'));
let server = net.createServer(function (socket) {
    socket.pipe(file, {
        end: false
    });
    setTimeout(function () {
        file.end('bye bye');
        socket.unpipe(file);
    }, 5000);

});
server.listen(8080);
```

pause可以暂停 data事件触发，服务器会把客户端发送的数据暂存在缓存区里

```
const net = require('net');
const net = require('net');
const path = require('path');
let file = require('fs').createWriteStream(path.join(__dirname, 'msg.txt'));
let server = net.createServer(function (socket) {
    socket.pause();
    setTimeout(function () {
        socket.resume();
        socket.pipe(file);
    }, 10 * 1000);
});
server.listen(8080);
```

```
let net = require('net');
let path = require('path');
let ws = require('fs').createWriteStream(path.resolve(__dirname, 'msg.txt'));
let server = net.createServer(function (socket) {
    socket.setTimeout(5 * 1000);
    socket.pause();
    socket.on('timeout', function () {
        socket.pipe(ws);
    });

});
server.listen(8080);
```

```
let socket = new net.Socket([options])
```

- fd socket文件描述符
- type 客户端所有协议
- allowHalfOpen 是否允许半连接,服务器收到FIN包时不回发FIN包，可以使服务器可以继续向客户端发数据

```
socket.connect(port, host, callback);
socket.on('connect', callback);
```

- write表示向服务器写入数据
- end 用于结束连接
- error 连接发生错误
- destroy 销毁流
- close 表示连接关闭成功，hasError=true代表有可能有错误

```
socket.write(data,[encoding],[callback]);
```

```
let net = require('net');
let server = net.createServer(function (socket) {
    console.log("客户端已经连接");
    socket.setEncoding('utf8');
    socket.on('data', function (data) {
        console.log("已接收客户端发送的数据:%s", data);
        socket.write('服务器:' + data);
    })
    socket.on('error', function (err) {
        console.log('与客户端通信过程中发生了错误，错误编码为%s', err.code);
        socket.destroy();
    });
    socket.on('end', function (err) {
        console.log('客户端已经关闭连接');
        socket.destroy();
    });
    socket.on('close', function (hasError) {
        console.log(hasError ? '异常关闭' : '正常关闭');
    });
});
server.listen(808, function () {
    let client = new net.Socket();
    client.setEncoding('utf8');
    client.connect(808, '127.0.0.1', function () {
        console.log('客户端已连接');
        client.write('hello');
        setTimeout(function () {
            client.end('byebye');
        }, 5000);
    });
    client.on('data', function (data) {
        console.log('已经接收到客户端发过来的数据:%s', data);
    });
    client.on('error', function (err) {
        console.log('与服务器通信过程中发生了错误，错误编码为%s', err.code);
        client.destroy();
    });

});
```

停止server接受建立新的connections并保持已经存在的connections

```
 server.getConnections((err, count) => {
     if (count == 2) server.close();
 });
```

unref方法指定发客户端连接被全部关闭时退出应用程序 如果将allowHalfOpen方法，必须使用与客户端连接的socket端口对象的end 方法主动关闭服务器端连接

```
let net = require('net');
let server = net.createServer({ allowHalfOpen: true }, function (socket) {
    console.log("客户端已经连接");
    socket.setEncoding('utf8');
    socket.on('data', function (data) {
        console.log("已接收客户端发送的数据:%s", data);
        socket.write('服务器确认数据:' + data);
    })
    socket.on('error', function (err) {
        console.log('与客户端通信过程中发生了错误，错误编码为%s', err.code);
        socket.destroy();
    });
    socket.on('end', function (err) {
        console.log('客户端已经关闭连接');
        socket.end();
        server.unref();
    });
    socket.on('close', function (hasError) {
        if (hasError) {
            console.log('由于错误导致socket关闭');
            server.unref();
        } else {
            console.log('端口正常关闭');
        }
    });
    server.getConnections((err, count) => {
        if (count == 2) server.close();
    });
});
server.listen(808, function () { });
server.on('close', function () {
    console.log('服务器关闭');
});
```

write的返回值和bufferSize属性值

```
let server = net.createServer({ allowHalfOpen: true }, function (socket) {
    console.log("客户端已经连接");
    socket.setEncoding('utf8');
    let rs = fs.createReadStream(path.resolve(__dirname, 'a.txt'), { highWaterMark: 2 });
    rs.on('data', function (data) {
        let flag = socket.write(data);
        console.log("flag:", flag);
        console.log('缓存字节:' + socket.bufferSize);
        console.log('已发送字节:' + socket.bytesWritten);
    })
    socket.on('data', function (data) {
        console.log('data', data);
    });
    socket.on('drain', function (err) {
        "缓存区已全部发送"
    });
});
```

当服务器和客户端建立连接后，当一方主机突然断电、重启、系统崩溃等意外情况时，将来不及向另一方发送FIN包，这样另一方将永远处于连接状态。 可以使用setKeepAlive方法来解决这一个问题

```
socket.setKeepAlive([enaable],[initialDelay]);
```

- enable 是否启用嗅探，为true时会不但向对方发送探测包，没有响应则认为对方已经关闭连接，自己则关闭连接
- initialDelay 多久发送一次探测包，单位是毫秒

1.2.7 聊天室1.0

```

let net = require('net');
let util = require('util');
let clients = {};
let server = net.createServer(function (socket) {
    server.getConnections(function (err, count) {
        socket.write(`weclome,there is ${count} users now,please input your username\r\n`);
    });
    let nickname;
    socket.setEncoding('utf8');
    socket.on('data', function (data) {
        data = data.replace(/\r\n/, '');
        if (data == 'byebye') {
            socket.end();
        } else {
            if (nickname) {
                broadcast(nickname, `${nickname}:${data}`);
            } else {
                nickname = data;
                clients[nickname] = socket;
                broadcast(nickname, `welcome ${nickname} joined us!`);
            }
        }

    });
    socket.on('close', function () {
        socket.destroy();
    });
}).listen(8088);

function broadcast(nickname, msg) {
    for (let key in clients) {
        if (key != nickname) {
            clients[key].write(msg + '\r\n');
            clients[nickname].destroy();
            delete clients[nickname];
        }
    }
}
```

1.2.8 聊天室2.0

```
var key = scoket.remoteAddress+':'+socket.remotePort;
users[key] = {name:'匿名',socket};

socket.on('data',function(){
    parse(data);
});
function parse(msg){
  swtich(msg.type){
   case 'secret':
     secret(msg.user,msg.text);
     break;
  }
  case 'boardcast':
     boardcast(message.text);
     break;
  case 'cname':
     cname(messsage.text);
     break;
  case 'list':
     list();
     break;
  default:
     socket.write('不能识别命令');
     break;
}
function secret(user,text){

}
function boardcast(text){

}
function cname(text){

}
function list(){

}
```

```
b:text 广播
c:nickname:text 私聊
n:nickname 改名
l 列出在线用户列表

on('data',function(data){
   if(data == 'quit){

   }else if(data == 'help'){

   }else(){
     write(data);
   }
});
function convert(){

}
```

- isIP 判断字符串是否是IP
- isIPv4 判断字符串是否是IPv4地址
- isIPv6 判断字符串是否是IPv6地址

2. UDP

```
let socket = dgram.createSocket(type,[callback]);
socket.on('messsage',function(msg,rinfo){});
```

- type 必须输入，制定时udp4还是udp6
- callback 从该接口接收到数据时调用的回调函数
    - msg 接收到的数据
    - rinfo 信息对象
        - address 发送着的地址
        - family ipv4还是ipv6
        - port 发送者的socket端口号
        - size 发送者所发送的数据字节数

```
socket.bind(port,[address],[callback]);
socket.on('listening',callabck;
```

- port 绑定的端口号
- address 监听的地址
- callback 监听成功后的回调函数

如果发送数据前还没有绑定过地址和端口号，操作系统将为其分配一个随机端口并可以接收任何地址的数据

```
 socket.send(buf,offset,length,port,address,[callback]);
```

- buffer 代表缓存区
- offset 从缓存区第几个字节开始发
- length 要发送的字节数
- port 对方的端口号
- address 接收数据的socket地址
- callback 制定当数据发送完毕时所需要的回调函数
    - err 错误对象
    - byets 实际发送的字节数**2.3 address** 获取此socket相关的地址信息

```
let address = socket.address();
```

- port
- address
- family

```
var dgram = require('dgram');
var socket = dgram.createSocket('udp4');
socket.on('message',function(msg,rinfo){
  console.log(msg.toString());
  console.log(rinfo);
   socket.send(msg,0,msg.length,rinfo.port,rinfo.address);
});
socket.bind(41234,'localhost');
```

```
var dgram = require('dgram');
var socket = dgram.createSocket('udp4');
socket.on('message',function(msg,rinfo){
    console.log(msg.toString());
    console.log(rinfo);
});
socket.setTTL(128);
socket.send(new Buffer('珠峰培训'),0,6,41234,'localhost',function(err,bytes){
    console.log('发送了个%d字节',bytes);
});
socket.on('error',function(err){
    console.error(err);
});
```

创建一个UDP服务器并通过该服务器进行数据的广播

```
let dgram = require('dgram');
let server = dgram.createSocket('udp4);
server.on('message',function(msg){
  let buf = new Bufffer('已经接收客户端发送的数据'+msg);
  server.setBroadcast(true);
  server.send(buf,0,buf.length,41235,"192.168.1.255");
});
server.bind(41234,'192.168.1.100');
```

```
let dgram = require('dgram');
let client = dgram.createSocket('udp4);
client.bind(41235,'192.168.1.102);
let buf = new Buffer('hello');
client.send(buf,0,buf.length,41234,'192.168.1.100');
client.on('message',function(msg,rinfo){
  console.log('received : ',msg);
});
```

- 所谓的组播，就是将网络中同一业务类型进行逻辑上的分组，从某个socket端口上发送的数据只能被该组中的其他主机所接收，不被组外的任何主机接收。
- 实现组播时，并不直接把数据发送给目标地址，而是将数据发送到组播主机，操作系统将把该数据组播给组内的其他所有成员。
- 在网络中，使用D类地址作为组播地址。范围是指 224.0.0.0 ~ 239.255.255.255,分为三类
    - 局部组播地址: 224.0.0.0 ~ 224.0.0.255 为路由协议和其他用途保留
    - 预留组播地址: 224.0.1.0 ~ 238.255.255.255 可用于全球范围或网络协议
    - 管理权限组播地址： 239.0.0.0 ～ 239.255.255.255 组织内部使用，不可用于Internet

把该socket端口对象添加到组播组中。

```
socket.addMembership(multicastAddress,[multicastInterface]);
```

- multicastAddress 必须指定，需要加入的组播组地址
- multicastInterface 可选参数，需要加入的组播组地址

```
socket.dropMembership(multicastAddress,[multicastInterface]);
socket.setMulticastTTL(ttl);
socket.setMulticastLoopback(flag);
```

```
let dgram = require('dgram');
let server = dgram.createSocket('udp4');
server.on('listening',function(){
  server.MulticastTTL(128);
  server.setMulticastLoopback(true);
  server.addMembership('230.185.192.108');
});
setInterval(broadcast,1000);
function broadcast(){
  let buffer = Buffer.from(new Date().toLocaleString());
  server.send(buffer,0,buffer.length,8080,"230.185.192.108");
}
```

```
let dgram = require('dgram');
let client = dgram.createSocket('udp4');
client.on('listening',function(){
    client.addMembership('230.185.192.108');
});
client.on('message',function(message,remote){
  console.log(message.toString());
});
client.bind(8080,'192.168.1.103');
```

附录

```
FF FB 1F FF FB 20 FF FB 18 FF FB 27 FF FD 01 FF FB 03 FF FD 03
FF FB 1F window size
FF FB 20 terminal speed
FF FB 18 terminal type
FF FB 27 Telnet Environment Option
FF FD 01 echo
FF FB 03 suppress go ahead
FF FD 03 suppress go ahead
&#x5982;&#x679C;&#x4E0D;&#x9700;&#x8981;&#x9019;&#x4E9B;, &#x6539;&#x7528;RAW&#x6A21;&#x5F0F;&#x5C31;&#x53EF;&#x4EE5;&#x4E86;
```

打开telnet功能