Node提供了net，http，dgram模块分别实现TCP/HTTP/UDP通信。

# 一 构建TCP服务器

## 1 基本构建

Node使用net模块的createServer方法构建TCP服务器。

const net = require('net');  
let server = net.createServer(function (socket) {  
 console.log('Connect...'); *//有访问才会输出* server.maxConnections = 3; *//可以用来设置服务器最大连接数* server.getConnections(function (err,count) { *//count为当前连接数* console.log(count);  
 });  
});  
server.listen(3000,function () {  
 console.log('Listen...'); *//服务器启动就会输出*});

server.listen(3000,function(){});方法其实是server下的listening事件：

server.on('listening',function () {  
 console.log('Listen...');  
 let address = server.address();  
 console.log(address.port);  
});

除了listening事件外，还支持connection，close，error事件。

当创建了一个TCP服务器后，可以通过server.address()方法来查看这个TCP服务器监听的地址，并返回一个JSON对象，该JSON对象的属性有：

port：TCP服务器监听的端口号；

family：说明TCP服务器监听的地址是PVV6还是IPV4；

address：TCP服务器监听的地址；

因为这个方法返回的是TCP服务器监听的地址信息，所以这个方法应该在使用了server.listen()方法或者绑定事件listening中的回调函数中使用。

## 2 与客户端进行数据交互

createSever()方法的回调函数参数是一个net.Socket对象（服务器所监听的端口对象），这个对象同样也有一个address()方法，用来获取TCP服务器绑定地址，同样也是返回一个含有port family address 属性的对象。

socket对象可以用来data事件接收客户端发送的流数据，支持的事件还有：connect，end，error，timeout。

利用socket.write()可以使TCP服务器发送数据，这个方法只有一个必须参数，就是需要发送的数据：第二个参数为编码格式，可选。同时，可以为这个方法设置一个回调函数。当有用户连接TCP服务器的时候，将发送数据给客户端：

const net = require('net');  
let server = net.createServer(function (socket) {  
 *//获取地址信息* let address = server.address();  
 let message = 'server address is ' + JSON.stringify(address);  
 socket.write(message,function () {  
 let writesize = socket.bytesWritten;  
 console.log(message + ' has end');  
 console.log('the size of message is ' + writesize);  
 });  
 socket.on('data',function (data) {  
 console.log('data is ' + data.toString());  
 let readsize = socket.bytesRead;  
 console.log('the size of data is ' + readsize);  
 });  
}).listen(3000);

可以使用telnet与服务器进行交互。上述代码中用到socket对象的bytesWritten和bytesRead属性，分别代表发送数据的字节数和接收数据的字节数。socket还有以下属性：

socket.localPort：本地端口的地址

socket.localAddress：本地IP地址

socket.remotePort：远程端口地址

socket.remoteFamily：远程IP协议地址

socket.remoteAddress：远程的IP地址

## 3 使用Node创建TCP客户端

Node创建TCP客户端只需要创建一个连接TCP客户端的socket对象即可，创建时候可以传入一个json对象，这个json对象有以下属性：

fd：置顶一个存在的文件描述，默认为null

readable：是否允许在这个socket上读，默认为false

writeable：是否允许在这个socket上写，默认为false

allowHalfOpen：该属性为false时，TCP服务器接收到客户端发送的一个FIN包后将会回发一个FIN包，该属性为true时，TCP服务器接收客户端发送的一个FIN包后不会回发FIN包。

let net = require('net');  
let client = new net.Socket();  
client.connect(3000,'localhost',function () {  
 console.log('connect...'); *//运行后输出该句，证明回调已经执行  
  
 //发送数据* client.write('msg of client');  
  
});  
*//新建的client对象（就是socket）用来支持数据交互  
  
//接收数据*client.on('data',function (data) {  
 console.log('the data come from server is ' + data.toString());  
});

# 二 UPD

## 1 简介

TCP数据传输是一种可靠的数据传输方式，在数据传输之前必须建立客户端与服务端之间的连接。UDP是一种面向非连接的协议，所以其传输速度比TCP更加快速。

Node使用dgram模块中的createSocket()方法创建一个UDP服务器，这个方法接收一个必须参数和一个可选参数，必须参数是一个标识UDP协议的类型，可指定为udp4或者udp6，可选参数是一个回调函数，即UDP服务器接收数据时触发的回调函数，回调函数有2个参数，一个是接收的数据，一个是存放发送者信息的对象。

const dgram = require('dgram');  
let socket = dgram.createSocket('udp4',function (msg,rinfo) {  
 *// code*});  
socket.bind(3000,'localhost',function () {  
 console.log('bind 3000....');  
});

这里与TCP的createServer一样，返回的都是socket对象，且回调函数就是监听message事件，因此使用createServer也可以不指定回调函数，直接显示的监听message事件就可以。

const dgram = require('dgram');  
let socket = dgram.createSocket('udp4');  
socket.bind(3000,'localhost',function () {  
 console.log('bind 3000....');  
});  
  
socket.on('message',function (msg,rinfo) {  
 console.log(msg.toString());  
});

## 2 与客户端数据交互

socket对象主要有以下方法：

bind(); 绑定端口号

send(); 发送数据

address(); 获取该socket端口对象相关的地址信息；

close(); 关闭socket对象

const dgram = require('dgram');  
let socket = dgram.createSocket('udp4',function (msg,rinfo) {  
 console.log(msg.toString());  
 let message = new Buffer('hello,I am server');  
 socket.send(message,0,message.length,rinfo.port,rinfo.address,function (err,bytes) {  
 if(err){  
 console.log('error is ' + err);  
 } else {  
 console.log('send ' + bytes + ' message');  
 }  
 });  
}).bind(3000,'localhost');

send方法有很多参数：

buf：需要发送的消息，可以是缓存对象或者字符串

offset：是个整数，代表消息在缓存中的偏移量

length：是个整数，代表消息的比特数

## 3 UDP客户端

const dgram = require('dgram');  
let message = new Buffer('msg from client');  
let socket = dgram.createSocket('udp4');  
socket.send(message,0,message.length,3000,'localhost',function (err,bytes) {  
 if(err){  
 console.log(err);  
 return;  
 } else {  
 console.log('client send ' + bytes + 'message')  
 }  
});  
socket.on('message',function (msg,rinfo) {  
 console.log('msg from server');  
});

# 三 加密模块crypto

加密模块需要底层OpenSSL支持，该模块提供了一套针对OpenSLL的hash（哈希）、hmac（密钥哈希）、cipher（编码）、decipher（解码）、sign（签名），以及verify（验证）等方法的封装。不可逆加密一般处理用户登录，可逆加密一般处理数据存储，如用户关键数据。目前最安全的加密方式是RSA。

## 1 不可逆加密hash

let crypto = require('crypto');  
/\*  
调用hash便拿，算法实现可以是：md5，sha1，sha256，sha512，ripemd160

没有参数，将会抛出异常  
 \*/  
let hash = crypto.createHash('md5');  
  
*//hash编码方式实现加密，binary省略后输出结果一致*

*/hash.update(‘123’);//可以分批多次分别将1,2,3传入*hash.update(new Buffer('tesstxt','binary'));  
*//返回加密后的字符串*let encode = hash.digest('hex');  
console.log(encode);

hash.digest()函数可以传入3个类型的参数：hex（十六进制）、binary（二进制、默认）、base64。

注意：hash在调用digest方法后会被清空，如果要使用多次加密，需要多次创建。

## 2 不可逆加密HMAC

HMAC是密钥相关的哈希运算消息认证码，其利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出，这样的加密方式由于密钥存在，更为安全复杂。

let crypto = require('crypto');

/\*  
 创建hamc对象：crypto.createHmac(algorithm,key);  
 参数一是一个指定的算法，参数二是密钥字符串  
 \*/  
let hmac = crypto.createHmac('md5','ruyue');  
hmac.update(new Buffer('testtxt','binary'));  
let encode = hmac.digest('hex');

注意：同hash一样，多次加密需要多次创建该对象。

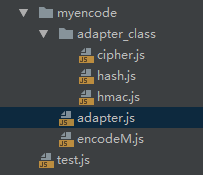
## 3 可逆加密Cipher和Decipher

let crypto = require('crypto')  
 ,key = 'ruyue'  
 ,text = 'testtext'  
 ,cipher = crypto.createCipher('aes-256-cbc',key)  
 ,decipher = crypto.createDecipher('aes-256-cbc',key);  
  
cipher.update(text,'utf8','hex');  
let ecryptedPWD = cipher.final('hex'); *//加密数据*decipher.update(ecryptedPWD,'hex','utf8');  
let PWD = decipher.final('utf8'); *//解密数据*

*/6522f9780cccde550d26d766f1886e1c*console.log(ecryptedPWD); */*console.log(PWD); *//testtext*

## 4 手动设计加密模块

设计一个私人加密模块：提供encode decode API接口。



test.js 为测试程序

adapter.js 兼容三种加密接口的适配器

encode.js 外部调用模块

一般情况下，我们在调用该模块时，只需要传递参数即可，如下所示 test.js：

const encodeM = require('./myencode/encodeM');  
let hashEncodeStr = encodeM.encode('hash','md5','ruyue','hex');  
console.log(hashEncodeStr);

调用到模块encodeM.js：

const AdapterM = require('./adapter');  
exports.encode = function () {  
 *//获取加密模块名，例如使用hash时，模块名为hash* let encodeM = arguments[0] ? arguments[0] :null  
 *//算法类型，如md5，sha1* ,algorith = arguments[1] ? arguments[1] : null  
 *//需要加密的字符串或者二进制数据流* ,enstring = arguments[2] ? arguments[2] : ''  
 *//输出的返回类型，如hex* ,returnType = arguments[3] ? arguments[3] : ''  
 *//密钥，可选参数* ,enkey = arguments[4] ? arguments[4] : ''  
 *//加密编码：binary，ascii，utf8* ,enType = arguments[5] ? arguments[5] : '';   
 *//调用Adapter中的encode方法* return AdapterM.encode(encodeM,algorith,enstring,returnType,enkey,enType);  
};  
exports.decode = function () {  
 let encodeM = arguments[0] ? arguments[0] :null   
 ,algorith = arguments[1] ? arguments[1] : null   
 ,enstring = arguments[2] ? arguments[2] : ''   
 ,returnType = arguments[3] ? arguments[3] : ''   
 ,dekey = arguments[4] ? arguments[4] : ''   
 ,deType = arguments[5] ? arguments[5] : '';  
 return AdapterM.decode(encodeM,algorith,enstring,returnType,dekey,deType);  
};

encodeM.js将数据传递给 适配器adapter.js

class Adapter{  
 constructor(){}  
 encode() {  
 let encodeM = arguments[0] ? arguments[0] :null  
 ,algorith = arguments[1] ? arguments[1] : null  
 ,enstring = arguments[2] ? arguments[2] : ''  
 ,returnType = arguments[3] ? arguments[3] : ''  
 ,enkey = arguments[4] ? arguments[4] : ''  
 ,enType = arguments[5] ? arguments[5] : ''  
 ,AdapteeClass = require('./adapter\_class/' + encodeM)  
 ,adapteeObj = new AdapteeClass();  
 *//调用Adapter中的encode方法* return adapteeObj.encode(algorith,enstring,returnType,enkey,enType);  
 };  
  
 decode() {  
 let encodeM = arguments[0] ? arguments[0] :null  
 ,algorith = arguments[1] ? arguments[1] : null  
 ,enstring = arguments[2] ? arguments[2] : ''  
 ,returnType = arguments[3] ? arguments[3] : ''  
 ,dekey = arguments[4] ? arguments[4] : ''  
 ,deType = arguments[5] ? arguments[5] : ''  
 ,AdapteeClass = require('./adapter\_class/' + encodeM)  
 ,adapteeObj = new AdapteeClass();  
 return adapteeObj.decode(algorith,enstring,returnType,dekey,deType);  
 };  
}  
module.exports = new Adapter();

适配器选择具体的执行模块，比如hash

const crypto = require('crypto');  
module.exports = function () {  
 this.encode = function () {  
 let algorithm = arguments[0] ? arguments[0] : null  
 ,enstring = arguments[1] ? arguments[1] : ''  
 ,returnType = arguments[2] ? arguments[2] : '';  
 let hash = crypto.createHash(algorithm);  
 hash.update(enstring);  
 return hash.digest(returnType);  
 };  
 this.decode = function () {  
 console.log('decode is not completed');  
 };  
};