

项目 3-1 设计报告——通信协议原理设计

—— 第一组 高浩峻 2016110902023

李逢君 2016060601010

王雨田 2016110902030

一、小组分工情况	2
二、成帧、差错控制、信息表示（编码）等技术的完整解决方案	2
2.1 整体框架	2
2.2 成帧	4
2.3 差错控制	6
2.5 二维矩阵的表示形式及纠错方法	8
2.5.1 表示形式	8
2.5.2 纠错方法	9
2.5.3 从原理角度解释一下	10
三、信息表示（编码）	11
四、测试数据和对测试数据的分析（是否达到功能要求）	11
3.1 本机与本机测试	11
3.2 代码部分讲解	14
3.2.1 发送方	14
3.2.2 接收方	17
3.2.3 实验过程	18
3.2.4 本机与同局域网下的计算机测试	21
3.2.5 测试结果	27
五、反思：与教材相关知识的衔接，经验、教训、收获、体会等	28

一、 小组分工情况

在项目实现的前期，我们小组采取了纵向的分工方式，高浩峻和王雨田负责成帧、差错控制、信息表示（编码）等技术的完整解决方案，李逢君负责用 nodejs 的代码实现，李逢君在编程的过程中也对编码的方式进行了优化。

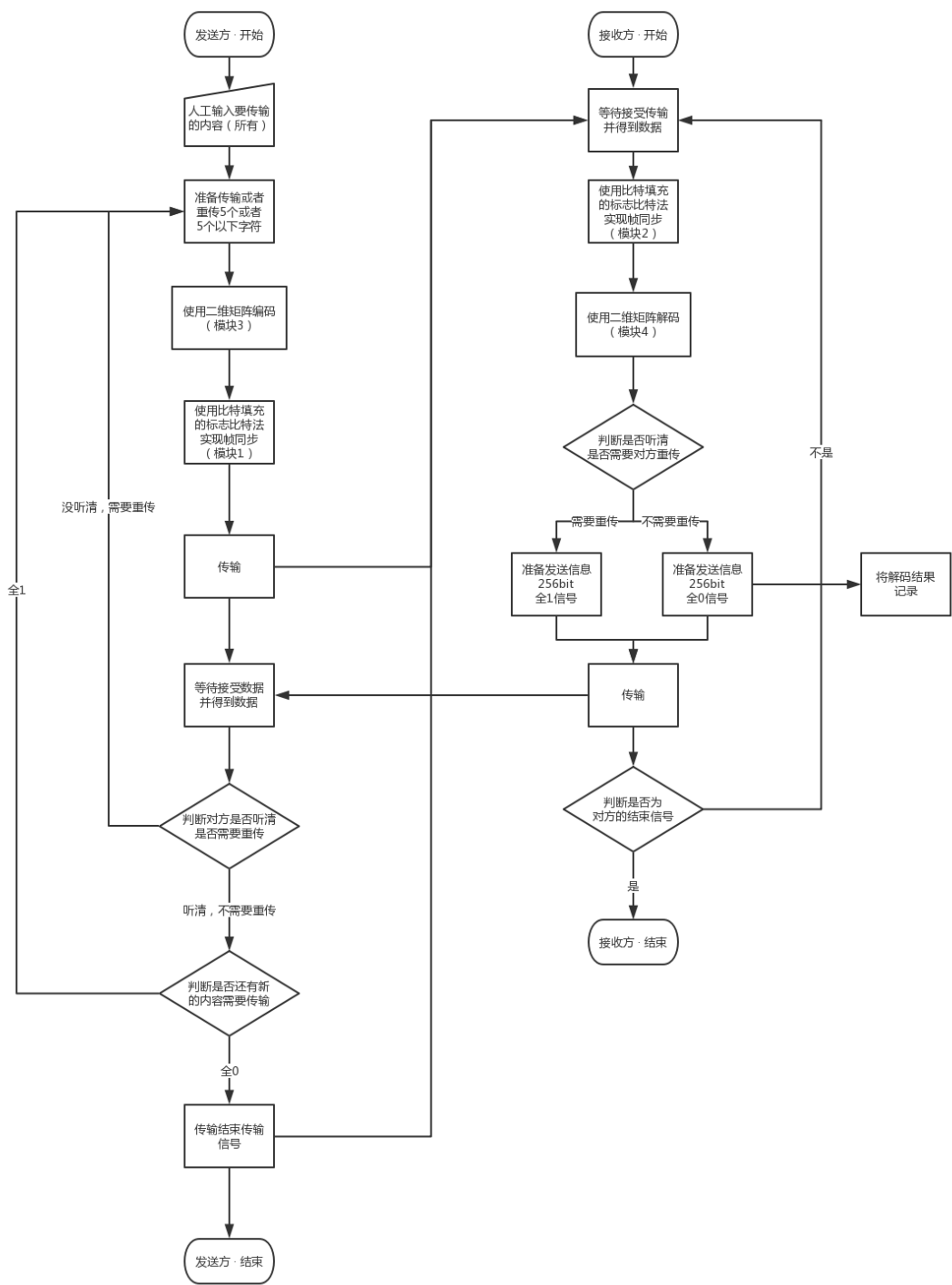
后期，我们一起完成了对测试数据的分析，寻找当前方案所出现的精度问题，并提出了更优化的解决方案。

二、 成帧、差错控制、信息表示（编码）等技术的完整解决方案

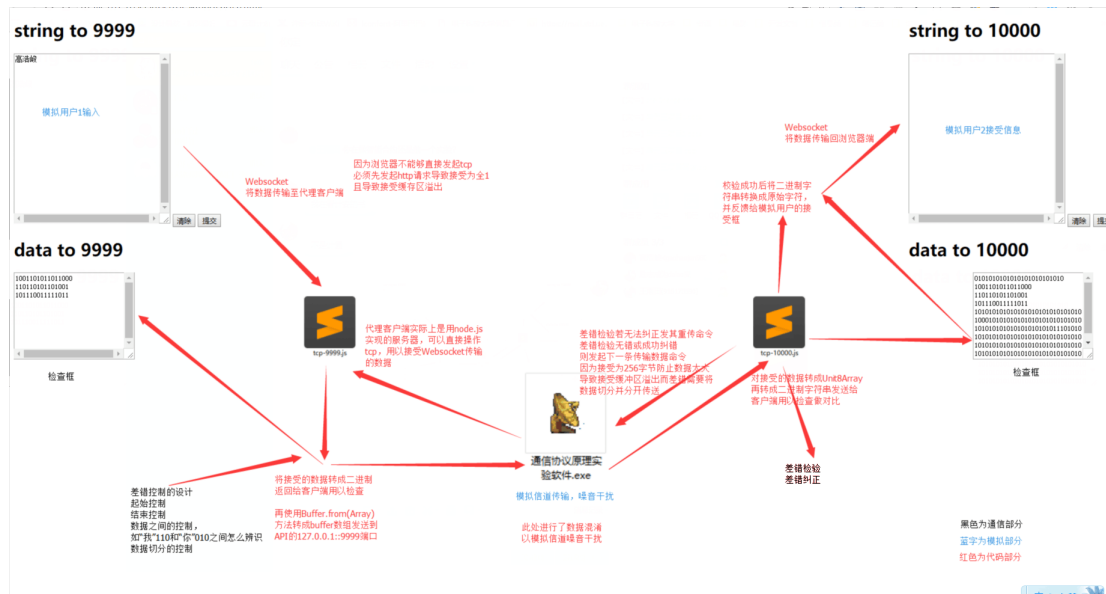
2.1 整体框架

首先，我们用流程图的方式将整体的实现过程表示出来，其中有四个模块，**模块 1&2** 分别是发送方和接收方的**成帧**过程的实现。**模块 3&4** 分别是发送方和接收方的**差错控制**的实现过程。

这个流程图其实只能显示单工的过程，但是我们在最后的实现过程发现，只要是接收方和发送方对于同一字段的定义不产生冲突（也就是不出现：发送方发送全 1bit 流表示发送结束，接收方发送全 1bit 流表示需要重传），就完全可以实现双工通信，也就是接收方和发送方跑的是同一段程序。



下面这张图是将实现过程与前端结合（流程可视化）之后的逻辑图



2.2 成帧

对于成帧的方式，我们采取了 HDLC 中的比特填充的标志比特法：

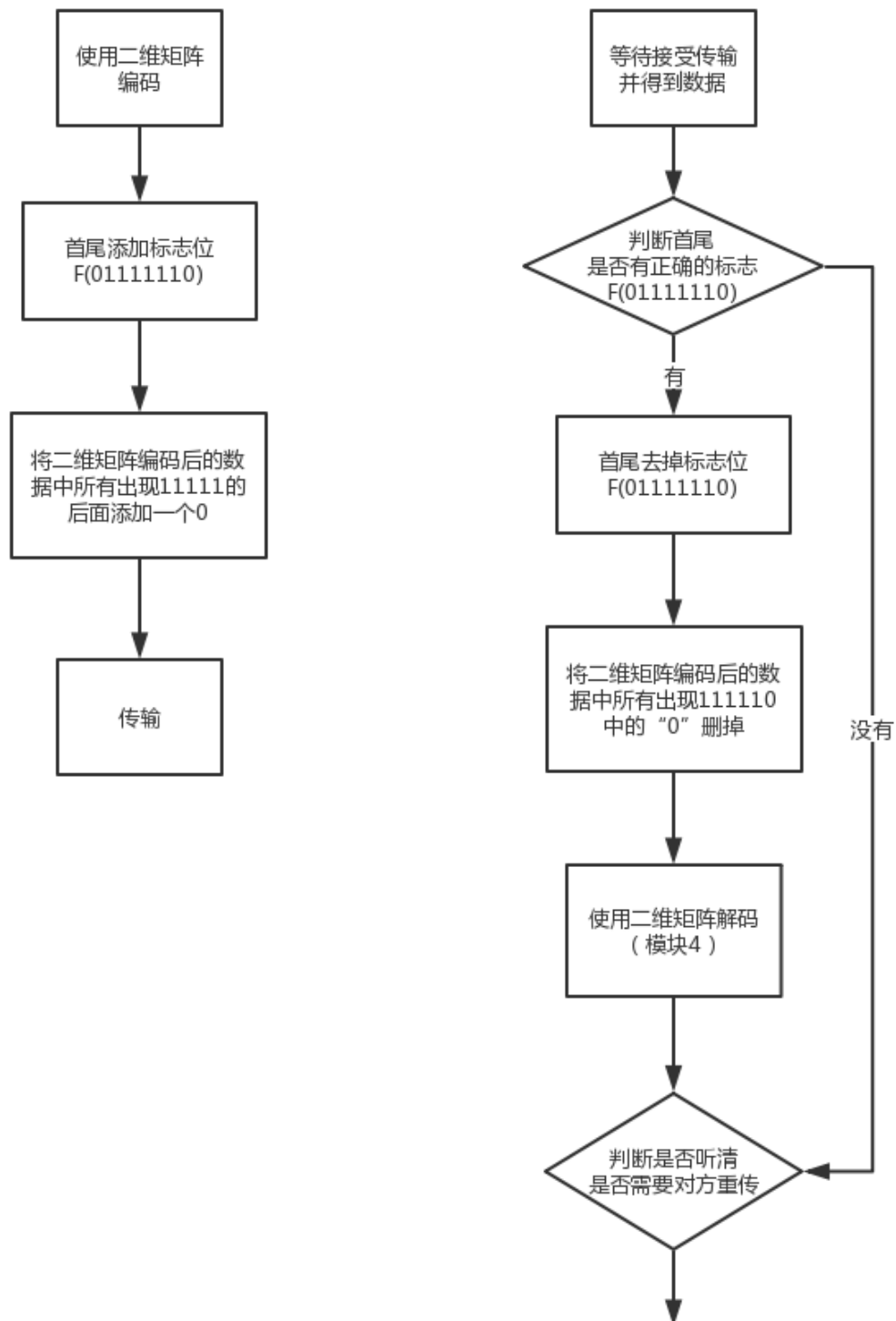
每个帧前、后均有一标志码 01111110，用作帧的起始、终止指示（也就是帧的同步）。又为了使标志码不出现在帧的内部，采用“0 比特插入法”来解决。也就是发送端遍历除标志码以外的所有字段，当发现有连续 5 个“1”出现时，便在其后添插一个“0”，然后继续发后继的比特流。在接收端，同样遍历起始标志码以外的所有字段。当连续发现 5 个“1”出现后，若其后一个比特“0”则自动删除它，以恢复原来的比特流。

接收方检测的时候，若发现连续 6 个“1”，则可能是插入的“0”发生差错变成“1”，也可能是收到了帧的终止标志码。如果我们固定每一帧所传输的 bit 位（在我们的项目中，不考虑成帧所产生的冗余，每帧 125bit），就可以通过检测中间的 bit 位是不是 125 来判定是否检测到了正确的标志位，如果是的话进行下一

步，如果不是的话则发送重传请求。

下面两张图是模块 1&2

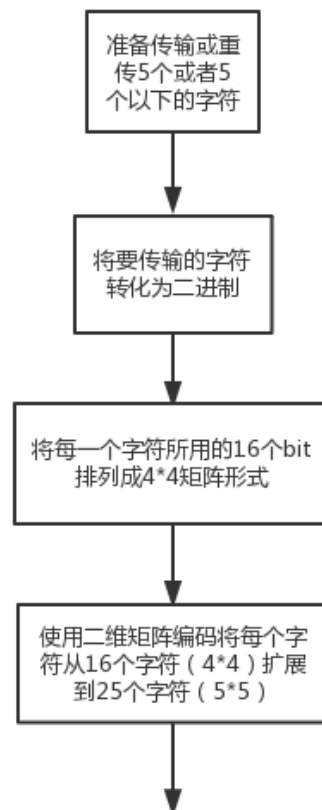
(左图为发送方的模块，右图为接收方的模块)



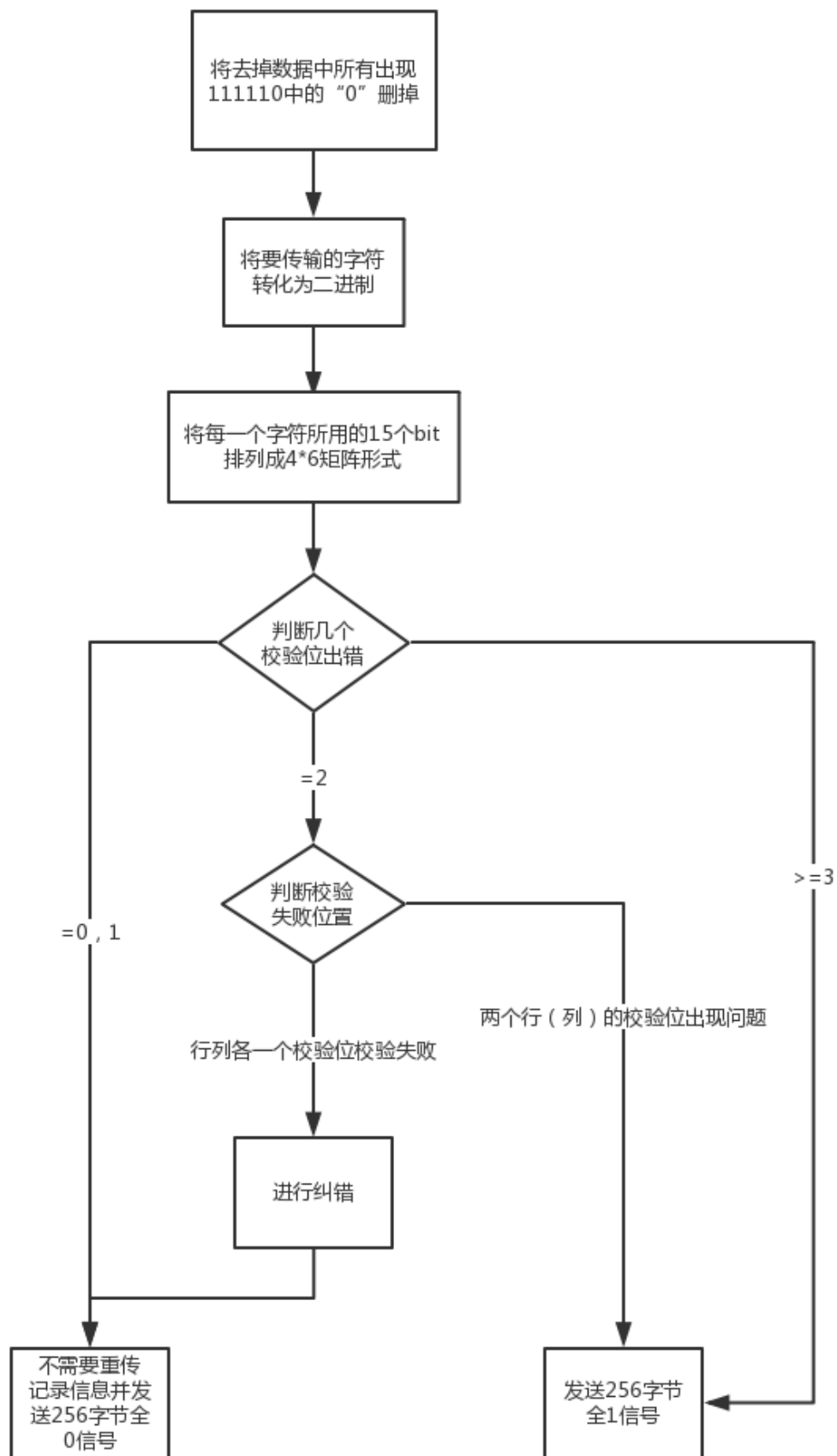
2.3 差错控制

最开始的时候，我们选择用汉明编码来实现差错控制，之后我们在看完课本之后，最终我们选择了使用方阵校验，也就是二维奇偶监督。

模块 3：二维矩阵编码



模块 4：二维矩阵解码



2.5 二维矩阵的表示形式及纠错方法

2.5.1 表示形式

1. 假设需传输的信息流为 1100 1011 1100 1001

2. 将信息流写成 4*4 的矩阵形式：

1	1	0	0
1	0	1	1
1	1	0	0
1	0	0	1

3. 使用偶校验，添加 5*5 如下：

1	1	0	0	0
1	0	1	1	1
1	1	0	0	0
1	0	0	1	0
0	0	1	0	1

偶检验方法是：

- a) 根据每行（每列）1 的个数来添加校验码。若该行中共奇数个 1，则在行的末尾添加一个 1，使该行内 1 的总数变成偶数个；若行中原本就有偶数个 1，则添加 0。
- b) 列同理。
- c) 最右下角添上的校验码，是根据整个数据矩阵中所有 1 的个数进行添加的，若所有 1 的个数是奇数，则右下角有 1（以让 1 的个数达到偶数）。

2.5.2 纠错方法

1. 先判断是否发生错误。

如果没有发生错误则发送全 0 表示接收成功，不需要重传，如果发生错误则进行接下来的判断。

2. 如果发生错误，需要分情况讨论。

1. 如果有 3 位或者 3 位以上的校验出现错误，说明至少发生了两次及以上次数的错误，这种情况直接发送全 1 表示需要重新传输。
2. 如果发生 1 位校验位不匹配的情况，说明校验位的位置出现错误，因为如果是校验位所在的行（列）中的源码发生错误，则源码所在列（行）的校验位也会校验失败。这种情况则不需要过多考虑，校验位本来就会在后面的阶段被去除。（也有可能是两个源码出错，这种情况在后面有说道，可能性比较小，所以不予考虑）
3. 如果发生 2 位校验位不符，则需要分情况讨论。
 - i. 如果是行列各一个校验位校验失败，则说明是原数据位出现问题。通过定位出错的校验位所在的行和列，其交叉点即为出错的位置，直接取反即可。（这种情况考虑不周，在反思部分有改进方案）
 - ii. 如果是两个行（列）的校验位出现问题，说明是原数据某一行（列）的两个源码出现错误，此时并不能确定是哪一个行（列）出现了问题，所以这种情况，就需要重传。

2.5.3 从原理角度解释一下

1、当发生单比特差错时，需要分情况讨论（发生单 bit 错误的时候，有可能会造成 1 位或者 2 位检验位不匹配的情况）

- a) 如果发生单 bit 错误, 且只有一位校验位不符, 则说明校验位出现错误, 则不需要过多考虑, 校验位本来就会在后面的阶段被去除。
- b) 如果发生单 bit 错误, 且有 2 位校验位不符, 则说明是原数据出现问题。通过定位出错的校验位所在的行和列, 其交叉点即为出错的位置, 直接取反即可。

2、当发生两比特差错时，需要分情况讨论（发生两 bit 错误的时候，可能会出现 1 位、2 位、3 位或者 4 位检验位不匹配的情况）

- a) 如果发生两 bit 错误, 且仅仅有 1 位校验位不符, 则说明一个源码出现了错误, 且对应的行列校验码中的一个也恰好出现了错误, 这种情况会导致纠错失败, 但是由于这种情况发生的可能较少, 所以暂时不予考虑。
- b) 如果发生两 bit 错误, 且仅有 2 位校验位不符, 则说明是原数据某一行（列）的两个源码出现错误, 此时并不能确定是哪一个行（列）出现了问题, 所以这种情况, 依旧需要重传。
- c) 如果发生两 bit 错误, 且有 3 位或 3 位以上校验位不符, 可能发生的情况很多, 而且能够确定出错位置的可能性很小, 所以当出现 3 位或 3 位以上校验位不符的时候, 我们选择重传。

三、 信息表示（编码）

这一部分，我们使用的方法是先将字符转换成 utf-8，再将 utf-8 转换成二进制编码。

但这个方法有一个问题就是每一个字符转换之后所对应的二进制码的位数是不固定的，但是这种编码方式有一个特点就是第一位全都是 1，而且经过测试，单个字符对应的二进制码最多是 16 个。

所以，我们的解决办法就是：将每一个字符对应的二进制码都用 16 个 bit 位表示，不足 16 位的在前面补 0，由于每一个字符都是以 1 开头，所以在解码的时候只需要将前面的 0 去掉就 OK 了。

四、 测试数据和对测试数据的分析（是否达到功能要求）

3.1 本机与本机测试

准备开始试验的之前，需要打开任务管理器结束 yundetectservice.exe 进程，原因是因为，如果不这样做运行软件就会出现 BUG：

任务管理器						
文件(F) 选项(O) 查看(V)						
进程 性能 应用历史记录 启动 用户 详细信息 服务						
名称	PID	状态	用户名	CPU	内存(专用...	描述
系统空闲进程	0	正在运行	SYSTEM	91	8 K	处理器空闲时间百分比
系统中断	-	正在运行	SYSTEM	00	K	延迟过程调用和中断服务例程
ZeroConfigService....	4128	正在运行	SYSTEM	00	3,552 K	Intel® PROSet/Wireless Zero Co...
yundetctservice.exe	10028	正在运行	user	00	1,704 K	yundetctservice.exe
WUDFHost.exe	984	正在运行	LOCAL SE...	00	996 K	Windows 驱动程序基础 - 用户模式...
WUDFHost.exe	564	正在运行	LOCAL SE...	00	1,176 K	Windows 驱动程序基础 - 用户模式...
WmiPrvSE.exe	6488	正在运行	SYSTEM	00	2,880 K	WMI Provider Host
WmiApSrv.exe	8740	正在运行	SYSTEM	00	956 K	WMI 性能反向适配器
wlanext.exe	3112	正在运行	SYSTEM	00	3,740 K	Windows Wireless LAN 802.11 E...
WINWORD.EXE	11624	正在运行	user	01	53,032 K	Microsoft Word
winlogon.exe	1136	正在运行	SYSTEM	00	1,328 K	Windows 登录应用程序
wininit.exe	752	正在运行	SYSTEM	00	964 K	Windows 启动应用程序
winguard_x64.exe	3880	正在运行	user	00	2,520 K	魔方守护
utility.exe	11116	正在运行	user	00	1,320 K	Lenovo Utility
unsecapp.exe	6384	正在运行	SYSTEM	00	1,184 K	Sink to receive asynchronous call...
TeamViewer_Servic...	3184	正在运行	SYSTEM	00	3,956 K	TeamViewer 13
Taskmgr.exe	12876	正在运行	user	00	26,764 K	任务管理器
taskhostw.exe	776	正在运行	user	00	2,712 K	Windows 任务的主机进程
SystemSettings.exe	364	已暂停	user	00	56 K	设置
System	4	正在运行	SYSTEM	00	20 K	NT Kernel & System

[简略信息\(D\)](#)
[结束任务\(E\)](#)

因为他占用了 127.0.0.1:10000, 这是本地的 10000 端口, 这会与跑在 10000 端口的 API 发生冲突, 导致无法启动 10000 端口的 API, 可以通过 cmd.exe 的命令 netstat -ano 查看端口占用情况:

选择C:\WINDOWS\system32\cmd.exe				
TCP	0.0.0.0:7000	0.0.0.0:0	LISTENING	13148
TCP	0.0.0.0:8000	0.0.0.0:0	LISTENING	6520
TCP	113.54.218.106:139	0.0.0.0:0	LISTENING	4
TCP	113.54.218.106:1551	111.221.29.156:443	ESTABLISHED	3228
TCP	113.54.218.106:1618	111.221.29.85:443	ESTABLISHED	11188
TCP	113.54.218.106:1702	182.254.34.77:80	CLOSE_WAIT	10832
TCP	113.54.218.106:1869	223.167.86.42:80	CLOSE_WAIT	10832
TCP	113.54.218.106:1873	182.254.76.104:80	CLOSE_WAIT	10832
TCP	113.54.218.106:1914	42.62.59.195:80	ESTABLISHED	6520
TCP	113.54.218.106:1973	14.18.234.254:80	ESTABLISHED	6520
TCP	113.54.218.106:5040	0.0.0.0:0	LISTENING	7684
TCP	127.0.0.1:1890	127.0.0.1:1891	ESTABLISHED	13148
TCP	127.0.0.1:1891	127.0.0.1:1890	ESTABLISHED	13148
TCP	127.0.0.1:1892	127.0.0.1:1893	ESTABLISHED	13148
TCP	127.0.0.1:1893	127.0.0.1:1892	ESTABLISHED	13148
TCP	127.0.0.1:1974	127.0.0.1:1975	ESTABLISHED	12652
TCP	127.0.0.1:1975	127.0.0.1:1974	ESTABLISHED	12652
TCP	127.0.0.1:1976	127.0.0.1:1977	ESTABLISHED	12652
TCP	127.0.0.1:1977	127.0.0.1:1976	ESTABLISHED	12652
TCP	127.0.0.1:4300	0.0.0.0:0	LISTENING	10832
TCP	127.0.0.1:4301	0.0.0.0:0	LISTENING	10832
TCP	127.0.0.1:5037	0.0.0.0:0	LISTENING	13112
TCP	127.0.0.1:5354	0.0.0.0:0	LISTENING	3660
TCP	127.0.0.1:5939	0.0.0.0:0	LISTENING	3184
TCP	127.0.0.1:9990	0.0.0.0:0	LISTENING	3780
TCP	127.0.0.1:10000	0.0.0.0:0	LISTENING	10028
TCP	127.0.0.1:31752	0.0.0.0:0	LISTENING	12012
TCP	169.254.178.154:139	0.0.0.0:0	LISTENING	4
TCP	169.254.178.154:5040	0.0.0.0:0	LISTENING	7684
TCP	192.168.100.19:139	0.0.0.0:0	LISTENING	4

本地的客户端是使用 nodejs 的来写的，nodejs 是一个能够运行 javascript 平台，而 javascript 是用于写网页的脚本语言，所以可以提供良好的数据可视化，提高人机交互体验。

同样的，实验之前需要配置好本机的 nodejs 环境，以及下载 git bash 命令行来运行 node。git bash 是基于 cmd，并且在 cmd 的基础上新增了一些命令和功能，用的是 ubuntu 的命令语法，更加方便。

解释一下为什么不直接在浏览器跑 javascript，因为 javascript 在浏览器的环境下只能操纵 http，并不能直接操纵 tcp，以 9999 端口作发送端，10000 端口作接收端为例，如果直接对本地的 9999 端口 API 发起 http 请求会导致接收处全为 1，并且报错接受缓冲区溢出，估计是 http 请求报文的内容全被解析成 1，因此不能直接在浏览器使用 javascript。

所以我使用的办法是在本地写了一个页面用来填写数据作数据可视化，然后把数据用 websocket(http 也可)转发到本地跑起 node.js 的服务器，然后再使用 node.js 的 net 模块操纵 tcp 向本地的(软件)挂起的 9999 端口发送数据，10000 端口收到数据后，再用另一个 node.js 服务器接受 10000 端口的数据，再把该数据转发到本地页面。

结构相当于：

网页客户端-代理服务器 9999-通信协议原理实验软件(服务器)-代理服务器 10000-网页客户端

3.2 代码部分讲解

3.2.1 发送方

首先，并不是从网页客户端接受到的字符串数据就能够往服务器发送，因为 tcp 只能接受 0 或者 1 的比特流，因此需要代理服务器将接受到的字符串转化成比特数据，我们的做法是将字符串用 unicode 编码，然后转化成二进制；同样解码是将二进制转化成 unicode，然后将 unicode 转化成字符串。

实现代码如下：

```
//将字符串转换成二进制形式，中间用空格隔开
function strToBinary(str){
    var result = [];
    var list = str.split("");
    for(var i=0;i<list.length;i++){
        if(i != 0){
            result.push(" ");
        }
        var item = list[i];
        var binaryStr = item.charCodeAt().toString(2);
        result.push(binaryStr);
    }
    return result.join("");
}

//将二进制字符串转换成Unicode字符串
function binaryToStr(str){
    var result = [];
    var list = str.split(" ");
    for(var i=0;i<list.length;i++){
        var item = list[i];
        var asciiCode = parseInt(item,2);
        var charValue = String.fromCharCode(asciiCode);
        result.push(charValue);
    }
    return result.join("");
}
```

经过我们测试，使用这种方法进行编码的单个字符，最多会出现 16 个比特位的二进制。考虑到后期我们将使用二维奇偶校验做差错检验以及一定的差错纠正，因此为了使二维编码方便，我们将字符转化为二进制不足 16 个比特位的往比特字符串的高位补充 0 至 16 个比特位。

代码实现如下：

```
//为没满16位的字符二进制前加0
while(charArray[i].length < eachBytePerChar){
    charArray[i] = "0" + charArray[i];
}
```

尔后，需要将 16 位比特字符串转化成二维数组：

```
//将字符二进制拆分为数组
var charElement = charArray[i].split("");
//定义一个二维数组
var twoDimensionArray = new Array();
//组成一个5*4的数组，但是只有4*4有值
for(var k = 0 ; k < lines + 1; k++){
    twoDimensionArray[k] = new Array();
    for(var j = 0 ; j < columns; j++){
        if(k == lines){
            twoDimensionArray[k][j] = null;
        }else{
            twoDimensionArray[k][j] = charElement[k*lines+j];
            // console.log(twoDimensionArray[k][j]);
        }
    }
}
```

为行和列加上校验位：

```
//为行加上校验位
for(var k = 0 ; k < lines; k++){
    var oneNumber = 0;
    for(var j = 0 ; j < columns; j++){
        if(twoDimensionArray[k][j] == 1){
            oneNumber++;
        }
    }
    (oneNumber%2==0)?twoDimensionArray[k][columns]=0:twoDimensionArray[k][columns]=1;
    // console.log("line plus is" + twoDimensionArray[k][columns]);
    // console.log("oneNumber is:" + oneNumber);
}
//为列加上校验位
for(var j = 0 ; j < columns; j++){
    var oneNumber = 0;
    for (var k = 0 ; k < lines; k++){
        if(twoDimensionArray[k][j] == 1){
            totalOneNumberIn++;//4*4校验位
            oneNumber++;
        }
    }
    (oneNumber%2==0)?twoDimensionArray[lines][j]=0:twoDimensionArray[lines][j]=1;
    // console.log("oneNumber is:" + oneNumber);
    // console.log("column plus is:" + twoDimensionArray[lines][j]);
}
```

替 array[5][5]加上校验位：

```

//总校验位
for(var k = 0 ; k < lines+1; k++){
    for(var j = 0 ; j < columns+1; j++){
        if(twoDimensionArray[k][j] == 1){
            totalOneNumber++;
        }
    }
}
//为5-5加上校验位
(totalOneNumber%2==0)?twoDimensionArray[lines][columns]=0:twoDimensionArray[lines][columns]=1;

```

最后再将二维数组转化成字符串进行传输，此时的 1 个字符变成比特数组

是 25 位：

```

//将加上校验位的二维数组5*5转化为字符串
newCharArray[i] = " ";
for(var k = 0 ; k < lines + 1; k++){
    for(var j = 0 ; j < columns + 1; j++){
        newCharArray[i] += twoDimensionArray[k][j];
    }
}

```

传输部分先将字符串中的 11111 转化为 111110，以每 5 个字符为 1 帧进行发送，并且以 01111110 作字符串首尾的标志位进行拼接，最后转化为字节

数组(Buffer 的形式)进行发送：

```

function sendData(charStatus){
    var BeginAndEndFlag = "01111110";
    var sendData = "";
    //每帧发送个5字符串
    for (var i = 0; i < eachCharLengthPerFrame ; i++){
        if(newCharArray[i+charStatus]){
            sendData += newCharArray[i+charStatus];
        }
    }
    //添加起始和终止标志位，并且将11111替换成111110
    sendData = sendData.replace(/ /g,'').replace(/11111/g,"111110");
    sendData = BeginAndEndFlag + sendData + BeginAndEndFlag;
    // console.log("sendData is:" + sendData);
    //将要发送的数据变成比特数组
    var sendDataArray = sendData.split("");
    // console.log("length is:" + sendDataArray.length);
    var dataBuffer = Buffer.from(sendDataArray);
    // console.log("buffer is:" + dataBuffer);
    client.write(dataBuffer);
}

```

以上就是发送方的全部讲解。

3.2.2 接收方

首先，接收的时候需要新建一个八位(通过抓包可以发现)整数数组对象来接收传过来的数据：

```
var dataReceive = new UInt8Array(data);
```

并将数组转化成字符串：

```
var dataString = dataReceive.toString().replace(/,/g, "");
```

首先判断接收的字符串是否有首位标志位即 01111110，如果没有将无法提取出其中有意义的字符串，我们的方法是分别从字符串的头和尾来查找 01111110，如果两者的位置不一致则继续，否则说明标志位 01111110 发生变化，重传。

实现如下：

```
//如果标志位改变，则发送111111....重传
if(dataString.indexOf(BeginAndEndFlag)==dataString.lastIndexOf(BeginAndEndFlag)||dataString.indexOf(BeginAndEndFlag)==-1){
    reSend(i);
    return;
}else{//如果不是，则发送00000....继续
```

将有意义的数据从首尾标志位之间提取出来之后，需要将之前转化的 111110 变成 11111，然后判断转化后的字符串是否为 25 的倍数，如果不是则证明传输过程中 111110 发生变化，重传。

实现如下：

```
// 判断有多少个字符
var charNumber = dataString.length/((lines+1)*(columns+1));
// 如果不是整数则111110发生变化，重传
if(charNumber!=parseInt(charNumber)){
    reSend(i);
    return;
}
```

然后我们使用二维奇偶校验的方法对字符串进行差错检验及一定程度差错纠正。

由于此过程代码部分过于复杂，原理在本报告其他部分有做简短的介绍，就不在这部分详细解释，可阅读源码。

发送方与接收方的辨识，通过设置一个变量 sendData 为 true 或者 false 来辨识是否为发送方。如果发送方发送全 0 字节数组代表则发送完毕，如果接受方收到全 0 则停止接受。如果接受方发送全 1 字节数组则代表重传信号，发送全 0 字节数组则代表接着传信号。

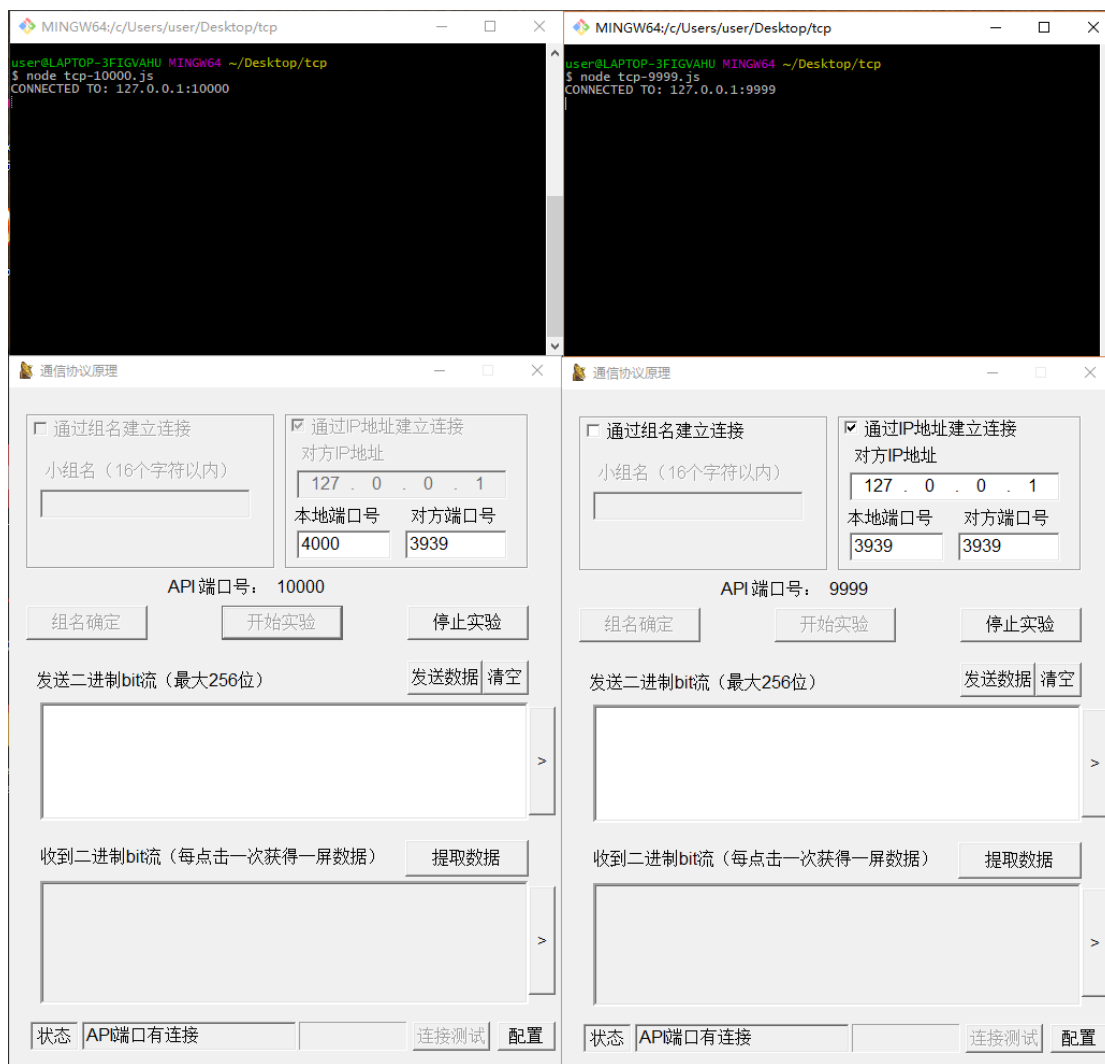
注全 1/0 字节数组并不是代表 256 个比特位全 1，如果超过 256 位则会导致接受缓冲区溢出，如果发送 256 位全 1 的数据服务器会分开两次发送数据，导致信息出错，因此我们设定 200 个元素为 1 的数组作全 1 数组，考虑到传输过程中有一定的几率 1 会置反，而且服务器会自动为数组两端添加 01010 的字段，因此另一方如果接受到超过一定阈值元素为 1 的数组即认定为全 1 数组。

3.2.3 实验过程

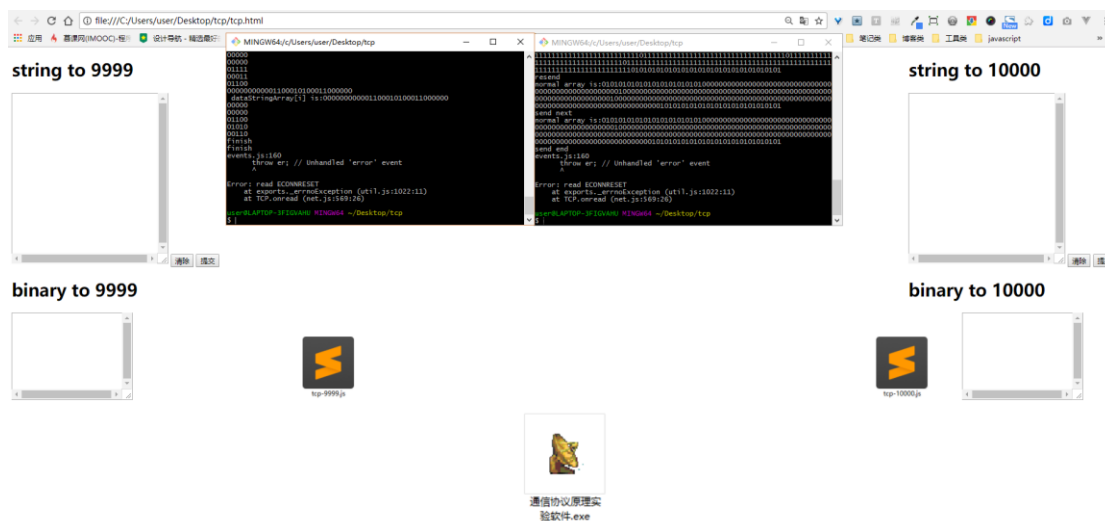
Node tcp-10000.js

Node tcp-9999.js

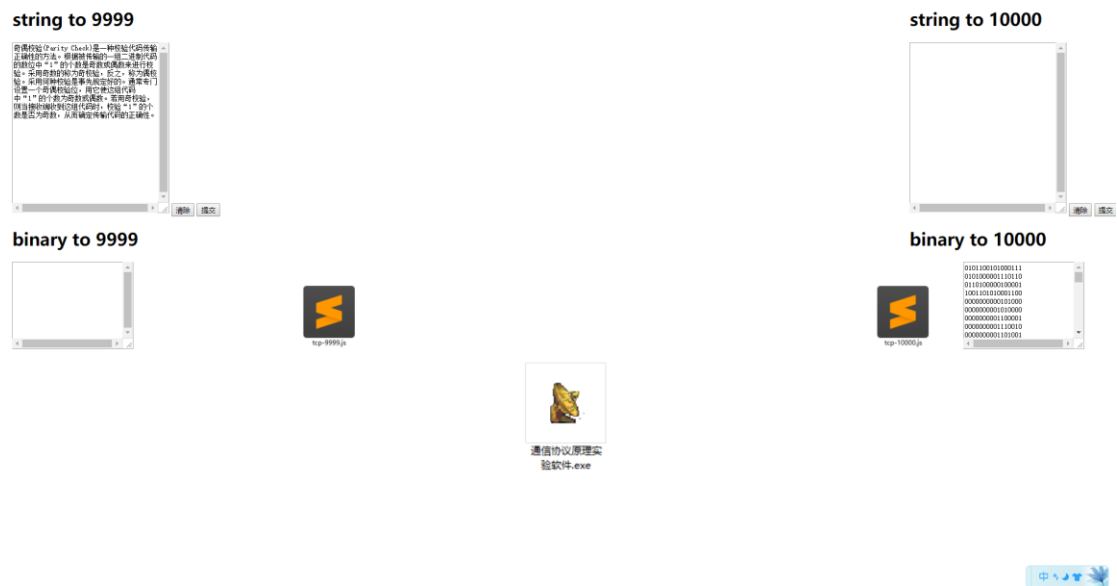
1. 启动服务



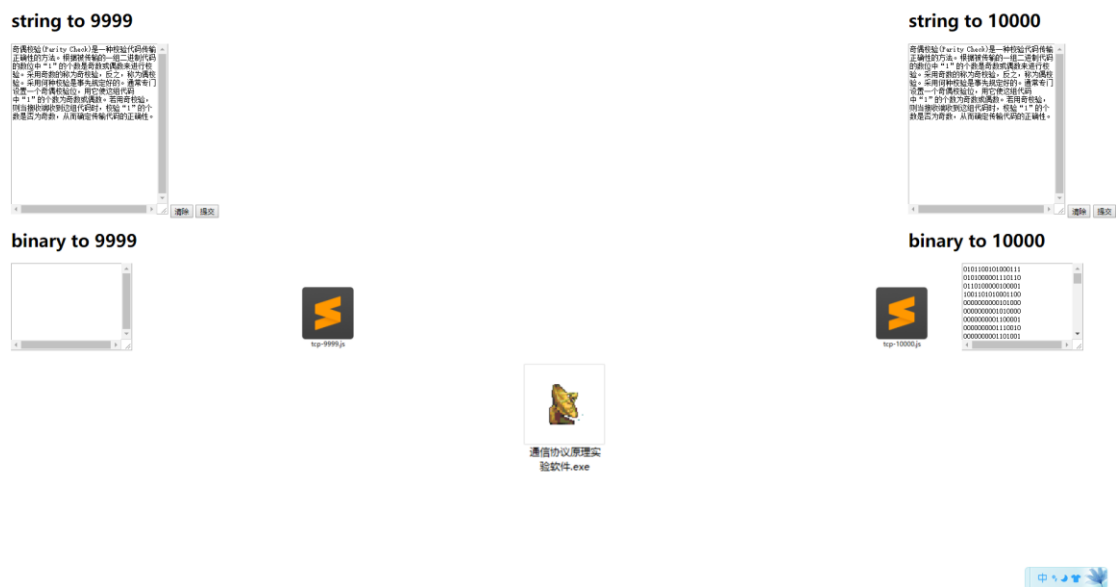
2. 打开网页客户端



3. 输入一段文字后点击提交



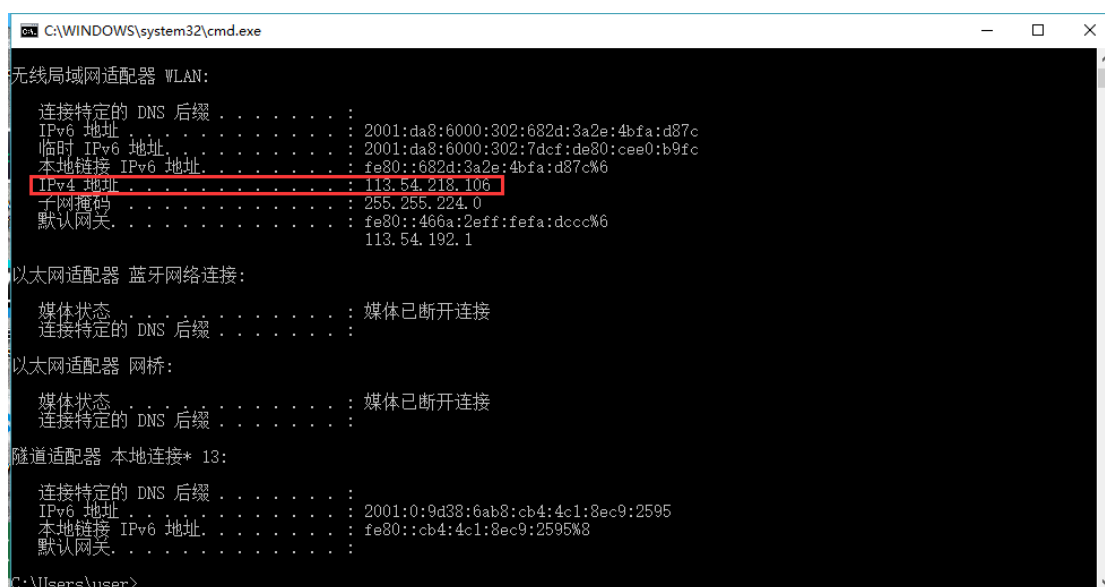
4. 点击提交后一段时间



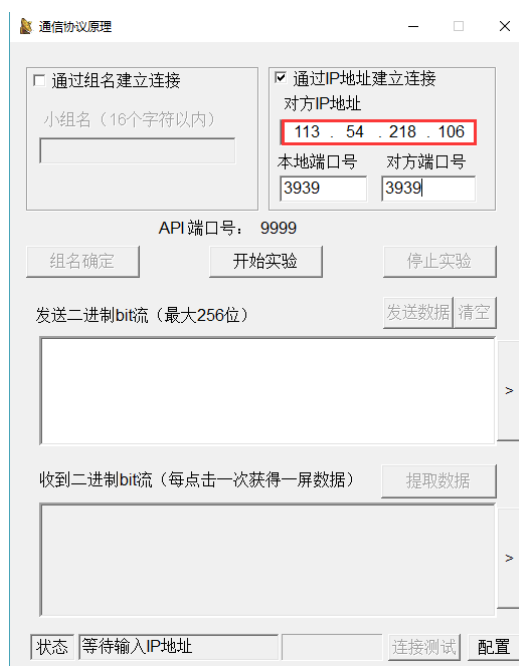
5. 数据传到了另外对接本地 10000 端口 API 的客户端窗口

3.2.4 本机与同局域网下的计算机测试

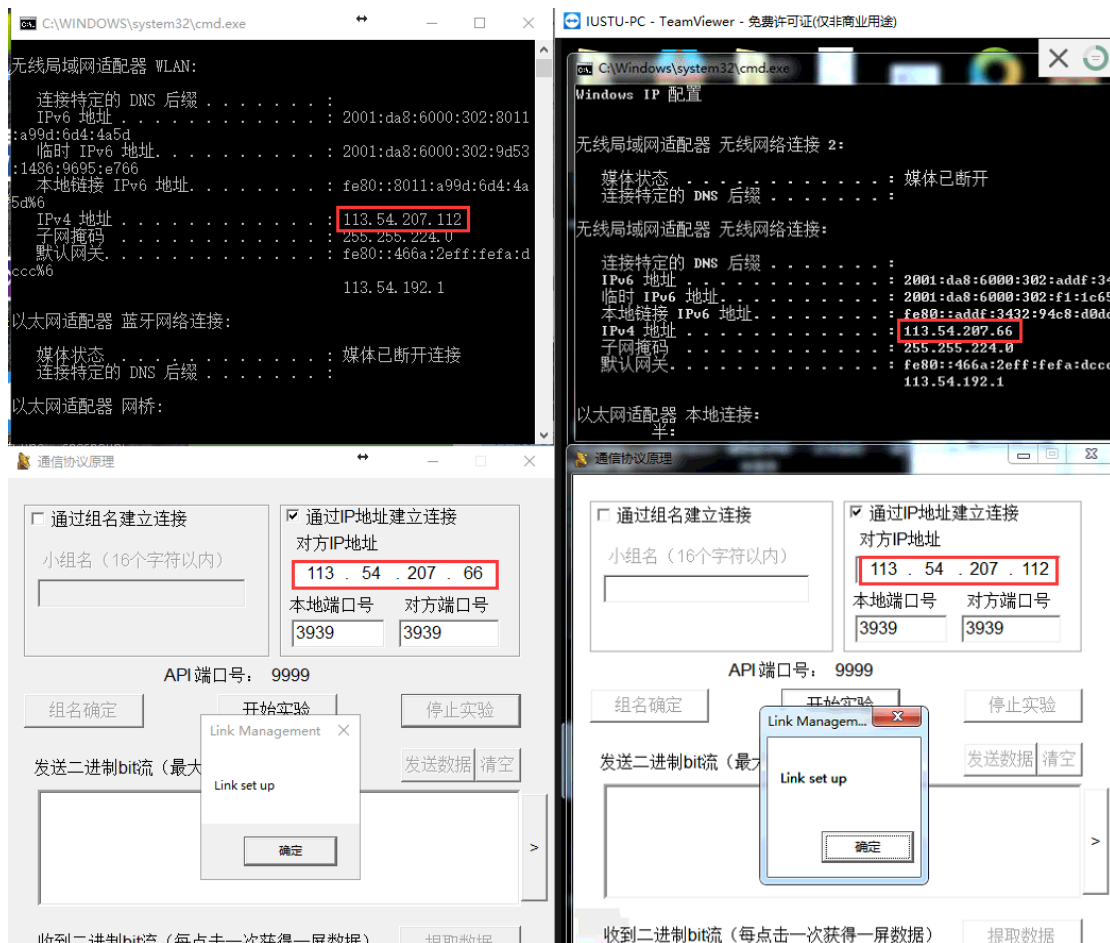
需要在 cmd.exe 中输入 ipconfig，拿到本机的内网的 ip 地址：



这个 ip 地址需要给通信的另一个计算机拿到并且填在通信协议原理实验软件中的“对方的 ip 地址”

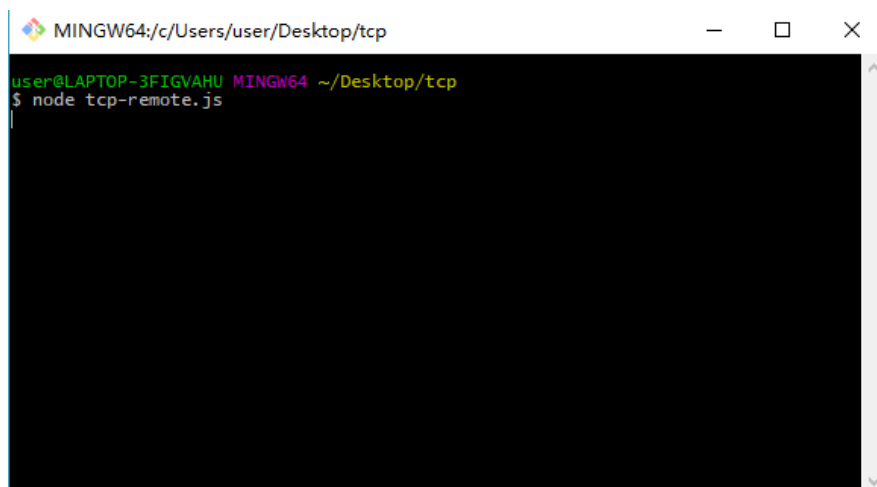


同样需要拿到另一个计算机内网的 ip 地址并填写在本机通信协议原理实验软件，下面为了截图方便，我们使用了 teamviewer 对另一台计算机进行连接操作。

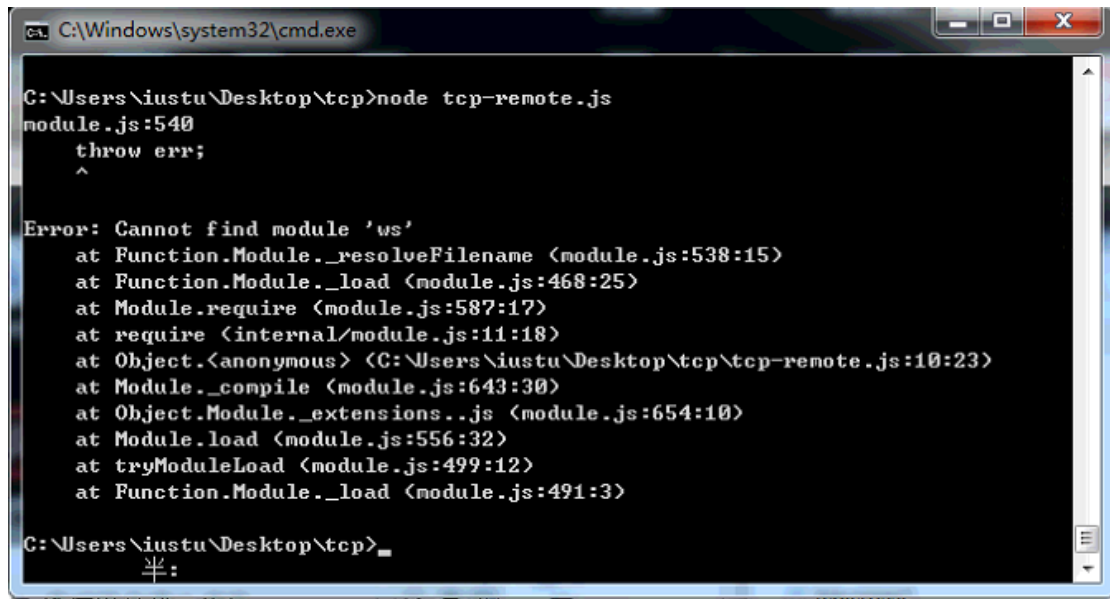


两台计算机需要用打开 git bash，输入 node tcp-remote.js

注：tcp-remote.js 在编码、传码、解码、验码等逻辑与上述实验的基本一致，只是添加了客户端输入本机 ip 与 port 的操作。



当然也可以使用 cmd.exe 挂起 node 服务器

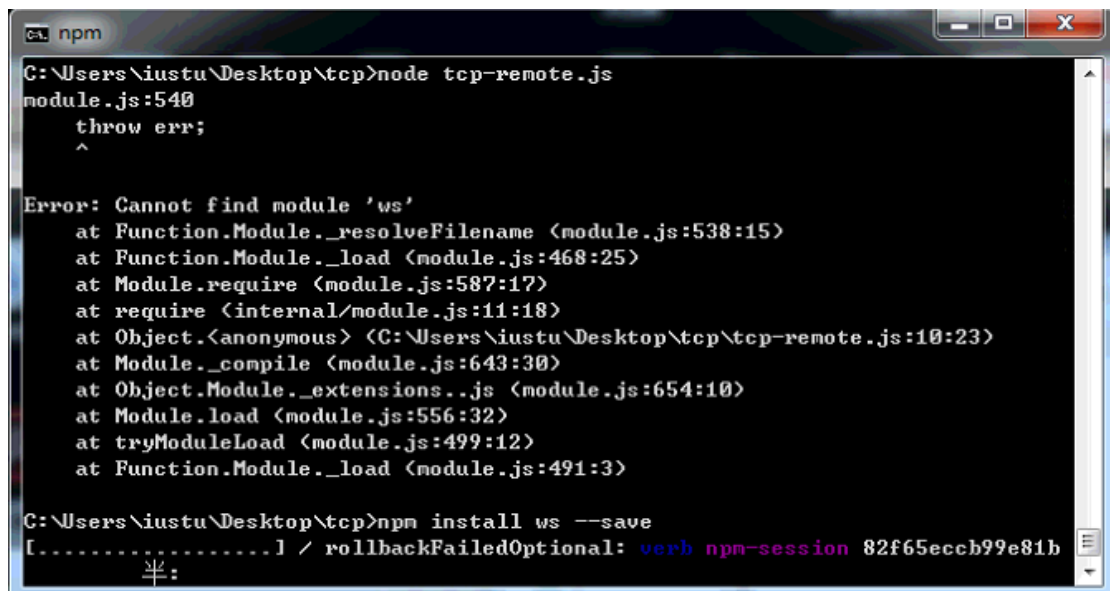


```
C:\Windows\system32\cmd.exe
C:\Users\iustu\Desktop\tcp>node tcp-remote.js
module.js:540
  throw err;
  ^

Error: Cannot find module 'ws'
    at Function.Module._resolveFilename (module.js:538:15)
    at Function.Module._load (module.js:468:25)
    at Module.require (module.js:587:17)
    at require (internal/module.js:11:18)
    at Object.<anonymous> (C:\Users\iustu\Desktop\tcp\tcp-remote.js:10:23)
    at Module._compile (module.js:643:30)
    at Object.Module._extensions..js (module.js:654:10)
    at Module.load (module.js:556:32)
    at tryModuleLoad (module.js:499:12)
    at Function.Module._load (module.js:491:3)
C:\Users\iustu\Desktop\tcp>
```

这里报错是因为没有安装相关依赖

输入命令 `npm install ws --save` 安装相关依赖，并重新执行 `node tcp-remote.js` 挂起代理服务器



```
npm
C:\Users\iustu\Desktop\tcp>node tcp-remote.js
module.js:540
  throw err;
  ^

Error: Cannot find module 'ws'
    at Function.Module._resolveFilename (module.js:538:15)
    at Function.Module._load (module.js:468:25)
    at Module.require (module.js:587:17)
    at require (internal/module.js:11:18)
    at Object.<anonymous> (C:\Users\iustu\Desktop\tcp\tcp-remote.js:10:23)
    at Module._compile (module.js:643:30)
    at Object.Module._extensions..js (module.js:654:10)
    at Module.load (module.js:556:32)
    at tryModuleLoad (module.js:499:12)
    at Function.Module._load (module.js:491:3)
C:\Users\iustu\Desktop\tcp>npm install ws --save
[.....] / rollbackFailedOptional: verb npm-session 82f65eccb99e81b
C:\Users\iustu\Desktop\tcp>
```

```
C:\Windows\system32\cmd.exe - node tcp-remote.js

at Module.load (module.js:556:32)
at tryModuleLoad (module.js:499:12)
at Function.Module._load (module.js:491:3)

C:\Users\iustu\Desktop\tcp>npm install ws --save
npm WARN saveError ENOENT: no such file or directory, open 'C:\Users\iustu\Desktop\tcp\package.json'
npm WARN enoent ENOENT: no such file or directory, open 'C:\Users\iustu\Desktop\tcp\package.json'
npm WARN tcp No description
npm WARN tcp No repository field.
npm WARN tcp No README data
npm WARN tcp No license field.

+ ws@4.0.0
added 4 packages in 4.564s

C:\Users\iustu\Desktop\tcp>node tcp-remote.js
半:
```

下面是网页客户端：

your Ip:

ip:127.0.0.1 port:9999 提交 清除

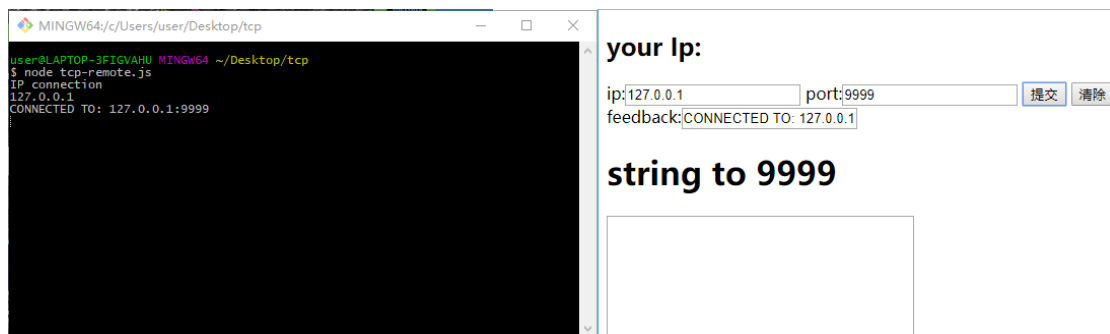
feedback:

string to 9999

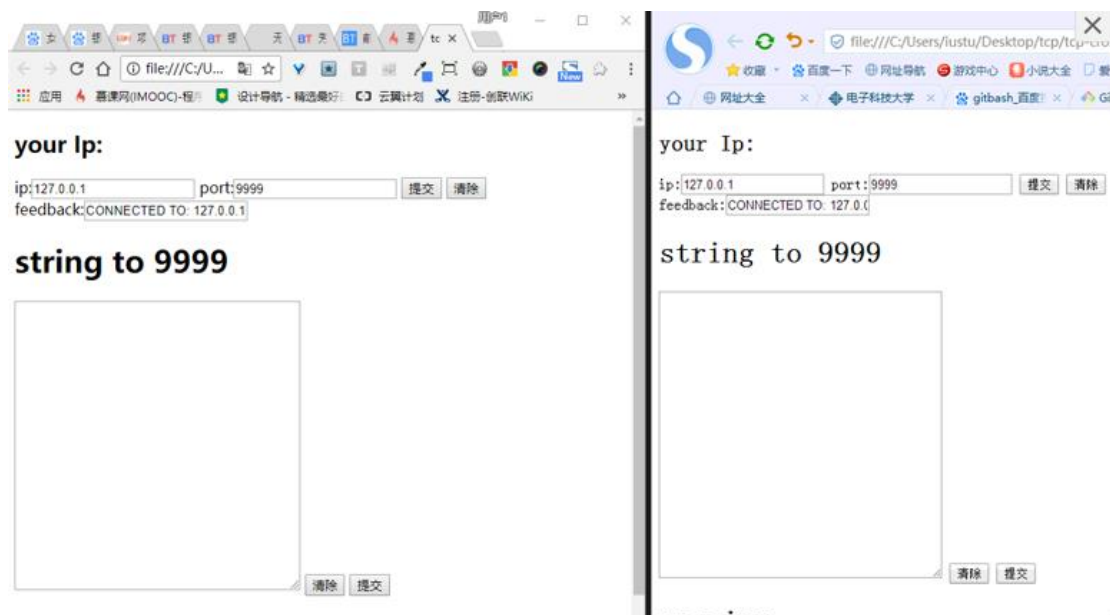
清除 提交

receive

首先输入本机的 ip(可以写 127.0.0.1)与 API 端口点击提交后会收到反馈：



同样对另一台计算机也是如此：



我们知道 127.0.0.1 是一个本机的回送地址，因此网页客户端将连接上(这里使用 Websocket)本机 9999 端口挂起的服务，也就是 113.54.207.66:9999 或者 113.54.207.112:9999，我们将使用本地网页客户端与本地服务器将数据完成在应用到端口上的传输，而实验软件会提供数据在内网下不同 ip 的端口上的传输。

输入一段数据，并且点击提交，另一台计算机收到本机发来的消息：

另一台计算机收到本机发来的消息

your Ip:

ip:127.0.0.1

port:9999

提交

清除

feedback:CONNECTED TO: 127.0.0.1

string to 9999

清除 提交

二十一世纪初，泰勒公司先进机器人发展到了连锁阶段，那些机器人实际上和人类完全相同，被称为复制人，这些复制人在体力、敏捷度和智慧上都不错，被人类用于外世界从事危险的劳动、危险的探险工作及其他星球的殖民任务上，经过外界殖民地连锁六号战斗组的血腥暴动后，地球上宣布复制人为违法物——必须处死。特勤小组——银翼杀手受命侦查任何入侵复制人，并予以击毙。那不叫做处决，而是称之为退休。Rick Deckard (Harrison Ford 饰) 就是银翼杀手之一，某天，他奉命追踪潜入泰勒公司的复制人Roy Batty (Rutger Hauer 饰)、Zhora (Joanna Cassidy 饰)、Leon (Brion James 饰) 和 Pris (Daryl Hannah 饰)，和他搭档的是泰勒公司的连锁六号复制人Rachael (Sean Young 饰)。在追踪的过程中，他和瑞秋产生了感情，明白了复制人们为了延长自身的机器寿命而做出的努力，渐渐地开始反思人类的命运。

清除 提交

receive

receive

二十一世纪初，泰勒公司先进机器人发展到了连锁阶段，那些机器人实际上和人类完全相同，被称为复制人，这些复制人在体力、敏捷度和智慧上都不错，被人类用于外世界从事危险的劳动、危险的探险工作及其他星球的殖民任务上，经过外界殖民地连锁六号战斗组的血腥暴动后，地球上宣布复制人为违法物——必须处死。特勤小组——银翼杀手受命侦查任何入侵复制人，并予以击毙。那不叫做处决，而是称之为退休。Rick Deckard (Harrison Ford 饰) 就是银翼杀手之一，某天，他奉命追踪潜入泰勒公司的复制人Roy Batty (Rutger Hauer 饰)、Zhora (Joanna Cassidy 饰)、Leon (Brion James 饰) 和 Pris (Daryl Hannah 饰)，和他搭档的是泰勒公司的连锁六号复制人Rachael (Sean Young 饰)。在追踪的过程中，他和瑞秋产生了感情，明白了复制人们为了延长自身的机器寿命而做出的努力，渐渐地开始反思人类的命运。

同样，另一边的信息也能够传输到本地上来：

your Ip:

ip:127.0.0.1

port:9999

提交

清除

feedback:CONNECTED TO: 127.0.0.1

string to 9999

二十一世纪初，泰勒公司先进机器人发展到了连锁阶段，那些机器人实际上和人类完全相同，被称为复制人，这些复制人在体力、敏捷度和智慧上都不错，被人类用于外世界从事危险的劳动、危险的探险工作及其他星球的殖民任务上，经过外界殖民地连锁六号战斗组的血腥暴动后，地球上宣布复制人为违法物——必须处死。特勤小组——银翼杀手受命侦查任何入侵复制人，并予以击毙。那不叫做处决，而是称之为退休。Rick Deckard (Harrison Ford 饰) 就是银翼杀手之一，某天，他奉命追踪潜入泰勒公司的复制人Roy Batty (Rutger Hauer 饰)、Zhora (Joanna Cassidy 饰)、Leon (Brion James 饰) 和 Pris (Daryl Hannah 饰)，和他搭档的是泰勒公司的连锁六号复制人Rachael (Sean Young 饰)。在追踪的过程中，他和瑞秋产生了感情，明白了复制人们为了延长自身的机器寿命而做出的努力，渐渐地开始反思人类的命运。

清除 提交

receive

这是科幻史上最有诚意的致敬 专访《银翼杀手2049》主创维伦纽瓦&高斯林
城会玩！隔空“亲热”低技版《银翼杀手2049》曝“AI女友”正片片段
“银翼杀手”导演有意执导《那湾25》忙于科幻电影《沙丘》拍摄或分身乏术
“雷神3”《银翼杀手2049》内地定档11月上映间隔一周 漫威奇幻遭遇索尼科幻
《银翼杀手2049》内地定档11月10日 雷德利·斯科特现身解读 北美口碑一片叫好
“银翼杀手2049”口碑解禁 一片叫好 众外媒被视效和手法征服 赞欧导演是“神”
高司令：从未参与过这么神秘的电影 《银翼杀手2049》曝光特辑 主创解读电影
“银翼杀手2049”发第二款中字预告短片“野猫”巴蒂斯塔铁汉柔情 为救女孩暴露身份

清除 提交

receive

二十一世纪初，泰勒公司先进机器人发展到了连锁阶段，那些机器人实际上和人类完全相同，被称为复制人，这些复制人在体力、敏捷度和智慧上都不错，被人类用于外世界从事危险的劳动、危险的探险工作及其他星球的殖民任务上，经过外界殖民地连锁六号战斗组的血腥暴动后，地球上宣布复制人为违法物——必须处死。特勤小组——银翼杀手受命侦查任何入侵复制人，并予以击毙。那不叫做处决，而是称之为退休。Rick Deckard (Harrison Ford 饰) 就是银翼杀手之一，某天，他奉命追踪潜入泰勒公司的复制人Roy Batty (Rutger Hauer 饰)、Zhora (Joanna Cassidy 饰)、Leon (Brion James 饰) 和 Pris (Daryl Hannah 饰)，和他搭档的是泰勒公司的连锁六号复制人Rachael (Sean Young 饰)。在追踪的过程中，他和瑞秋产生了感情，明白了复制人们为了延长自身的机器寿命而做出的努力，渐渐地开始反思人类的命运。

3.2.5 测试结果

测试结果，我们一次性发送了 10000 个“1”来测试会出现多少错误，实验结果是：9944 个字符传输正确，56 个字符传输错误。

也就是说我们的成功率为：99.44%。

之后我们又在原先基础上做了很多新的尝试，比如更改校验位数，增加冗余校验方法，这些都能够再次提升我们的准确率，但是又会出现新的问题，就是运行速度变慢。

所以我们又针对运行速度变慢又改进了传输方案：比如说，原先是一帧传 5 个字符，这一帧出错有整帧重传，改进方案就是重传时从出错的那个字符开始继续传输接下去的 5 个字符。

五、 反思：与教材相关知识的衔接，经验、教训、收获、体会等

在测试数据的过程中，我们追踪的传输失败的字符，研究了传输失败的原因（也就是方案上的漏洞），绝大部分传输失败的原因有两个：

一、传输过程中，源码中的一位与总校验位同时出错，或者行列各一个校验位出错的情况

当源码中的一个和总校验位同时出错之后，会导致有两个校验位的出错，而且是行列各一个错误，按照原来的思路：如果是行列各一个校验位校验失败，则说明是原数据位出现问题。通过定位出错的校验位所在的行和列，其交叉点即为出错的位置，直接取反即可。

这样就忽略了源码中的一位与总校验位同时出错，或者行列各一个校验位出错的情况。

1	1	0	0	0		1	1	0	0	0
1	0	1	1	1		1	1	1	1	1
1	1	0	0	0		1	1	0	0	0
1	0	0	1	0		1	0	0	1	0
0	0	1	0	1		0	0	1	0	0

传输之后：

1	1	0	0	0		1	1	0	0	0
1	1	1	1	1		1	1	1	1	1
1	1	0	0	0		1	1	0	0	0
1	0	0	1	0		1	0	0	1	0
0	0	1	0	1		0	0	1	0	0

上述这种情况就导致：如果出现行列各一个校验位出错（且不考虑总校验码），这时候分三种情况讨论：

- 1、 一个源码出错
- 2、 源码中的一位与总校验位同时出错
- 3、 行列各一个校验码出错

这时我们发现，2&3 的时候总校验码没有出现错误，而之后 1 的时候总校验码出现了错误。在之前的时候，我们从来没有用到过的总校验码就起作用了。

所以修改后的规则，如果出现行列各一个校验位出错，看总校验位有没有出错：

1. 如果总校验位出错，则将出错校验位所在行列交叉点取反以纠错。
2. 如果没有总校验位出现错误，则不能确定是什么位置出错，则选择重传。

二、传输过程中，一个矩阵中 3 位出现错误，且 3 位连线成直角三角形，也就是说再加一位可以构成矩形。

1	1	0	0	0
1	0	1	1	1
1	1	0	0	0
1	0	0	1	0
0	0	1	0	1

传输之后：

1	1	0	0	0
1	1	1	1	1
1	1	0	0	0
1	1	0	0	0
0	0	1	0	1

传输之后：

1	1	0	0	0
1	1	1	1	1
1	1	0	0	0
1	0	0	1	0
0	1	1	0	0

因为这种情况，虽然错了 3 个 bit，但是只有两个检验位不符（不包括总校验位）。

这个时候又要分两种情况来讨论了，3 位出现错误中，有没有出现在总校验位。这两种情况都会只产生两个校验位的错误。按照上面的结论，这时候需要判断总校验位符不符合，如果不符则纠正，如果符合则发送重传请求。

最要命的是，这里出现的这两种情况，总校验位一种是符合（校验位出错），一种是总校验位不符（总校验位没有出错）。

也就是说按照原来的方法，当出错的 3 位没有总校验位，所以总校验位校验失败，这个时候会去纠正一个 bit 使得这个方阵成立，而改动的这个就是以出错

的 3 位为顶点构成的矩形中剩下那个没有出错的顶点。

这个时候最后的结果肯定是失败的，所以这种情况我们是无法在不大量损失效率的情况继续优化的。

只有一个办法可以继续优化，就是出现两个和两个以上校验位出错就重传，这样能够继续精确我们的结果，但是付出的代价就是重传的概率大大提升，因为一般来说出错一个及以上的情况都会导致两个和两个以上校验位校验失败。也就是如果每 bit 出错概率在 0.3%-1% 浮动，那么将近有四分之一的帧需要重传。

总的来说，这个题目是一个非常有意思的题目，每一个环节都值得我们花很长的时间去思考和实现。每一个小的环节都会有很多种不同的实现方法，而每一种实现方法又有着很大的提升空间。

很多时候，我们会发现鱼和熊掌不可兼得：想要传输的内容更准确，就要牺牲冗余率，反之亦然；想要纠错的更多，就要牺牲处理的速度，反之亦然。不同的目的有不同的选择，不同的情况有不同的方案。我觉得这就是通信有意思的地方。