# 内网穿透开发文档

## 1. 引言

这个软件是我在课程设计中完成的一个作业。它的作用是帮助用户在私有网络（比如家里或办公室的网络）中的设备或服务与公共网络（比如互联网）进行通信(前提是你得获得公有服务端的允许)。通常情况下，私有网络中的设备是无法直接从外部访问的，但使用这个软件可以解决这个问题。

我们知道，有时候我们希望能够在外部远程访问家里的摄像头、家庭服务器或办公室的网络摄像头、打印机等设备。而NAT内网穿透软件正是为了解决这样的需求而设计的。

这个软件通过一种特殊的方式，让你能够像访问互联网上的公共设备一样，轻松地访问你私有网络中的设备。它会创建一个安全的连接，将你的请求从公共网络传递到私有网络，然后将响应返回给你。这样，你就可以在任何地方使用互联网连接访问你的设备，而不需要担心防火墙或路由器设置的限制。

这个软件还提供了一些额外的功能，比如可以让你在私有网络中搭建自己的私有云存储，或者在办公室的电脑上远程访问家庭电脑中的文件。它还可以帮助你轻松地进行文件共享或远程协助，无论你身在何处,此外，该软件还可以在局域网中进行使用，这样的话，就屏蔽了具体请求客户端的IP,保证了隐私安全。

总之，这个软件旨在让你能够方便地访问你私有网络中的设备或服务，解决内网访问的限制。它是为了提高你的网络使用体验而设计的，并且可以适用于各种常见的私有网络设置。

**注：此软件包含客户端和服务器两部分组成，用户可以根据选择进行使用，假设您有一台公有服务器，你可以将NAT的服务器部分部署上去，实现NAT穿透；当然您也可以将服务端部署到本地，实现局域的内网穿透。用户可以根据需要进行选择。**

## 2. 项目概述

### 2.1. 项目背景

在当今数字化时代，互联网的普及和技术的迅速发展使得人们越来越依赖于网络和互联设备。在家庭、办公室或其他场所，私有网络（LAN）的使用变得非常普遍。然而，由于网络安全和隐私的考虑，私有网络通常被配置为内部网络，无法直接从外部访问。

这就引发了一个常见的问题：如何在私有网络中的设备和服务与公共网络（比如互联网）之间进行安全、便捷的通信？这个问题尤其突出在需要远程访问家庭摄像头、私有服务器、打印机或其他设备的情况下。

为了解决这个问题，并提供一种简单、高效的解决方案，于是开发了这个NAT内网穿透软件。**该软件旨在克服私有网络中的限制，使用户能够在任何地方通过互联网访问他们的私有网络中的设备和服务。**

### 2.2. 目标

目标：

我们的NAT内网穿透软件的目标是提供一种简单、可靠的解决方案，使用户能够方便地远程访问他们的私有网络中的设备和服务

1. 远程访问：用户可以通过互联网连接，轻松地访问他们私有网络中的设备，如家庭摄像头、私有服务器、打印机等。
2. 私有云搭建：用户可以利用软件搭建自己的私有云存储，方便地存储和共享文件，实现远程访问和协作。
3. 文件共享：用户可以轻松地与他人共享文件和文件夹，无论他们身在何处。
4. 远程协助：软件提供远程协助功能，让用户能够远程协助家庭成员、同事或客户解决问题，提供技术支持或培训。

### 2.3. 关键特性和功能

1. 远程访问：用户可以通过互联网连接远程访问私有网络中的设备和服务，如家庭摄像头、私有服务器、打印机等。
2. 安全通信：软件提供安全的通信通道，使用加密协议保护数据传输的安全性和隐私(目前还没有集成该功能，后续会继续优化)。
3. 简单设置：用户可以通过简单的设置步骤和直观的界面，轻松地配置和启用内网穿透功能。
4. 网络穿透：软件通过网络穿透技术，允许用户在无需手动配置端口映射或路由器设置的情况下，实现对私有网络设备的访问。
5. 多设备支持：软件支持同时访问多个设备，允许用户管理和访问他们私有网络中的多个设备和服务。
6. 高性能和稳定性：软件设计优化，具有良好的性能和稳定性，确保远程访问的可靠性和流畅性。
7. 自定义配置：软件提供灵活的配置选项，允许用户根据他们的需求自定义设置和参数，以满足特定的网络环境和要求。
8. 日志记录和故障排除：软件记录关键事件和活动，以便用户进行故障排除和日志分析，帮助解决问题和提供技术支持。
9. 跨平台兼容：软件设计为跨平台兼容，可在不同的操作系统（如Windows、Mac、Linux等）上运行。

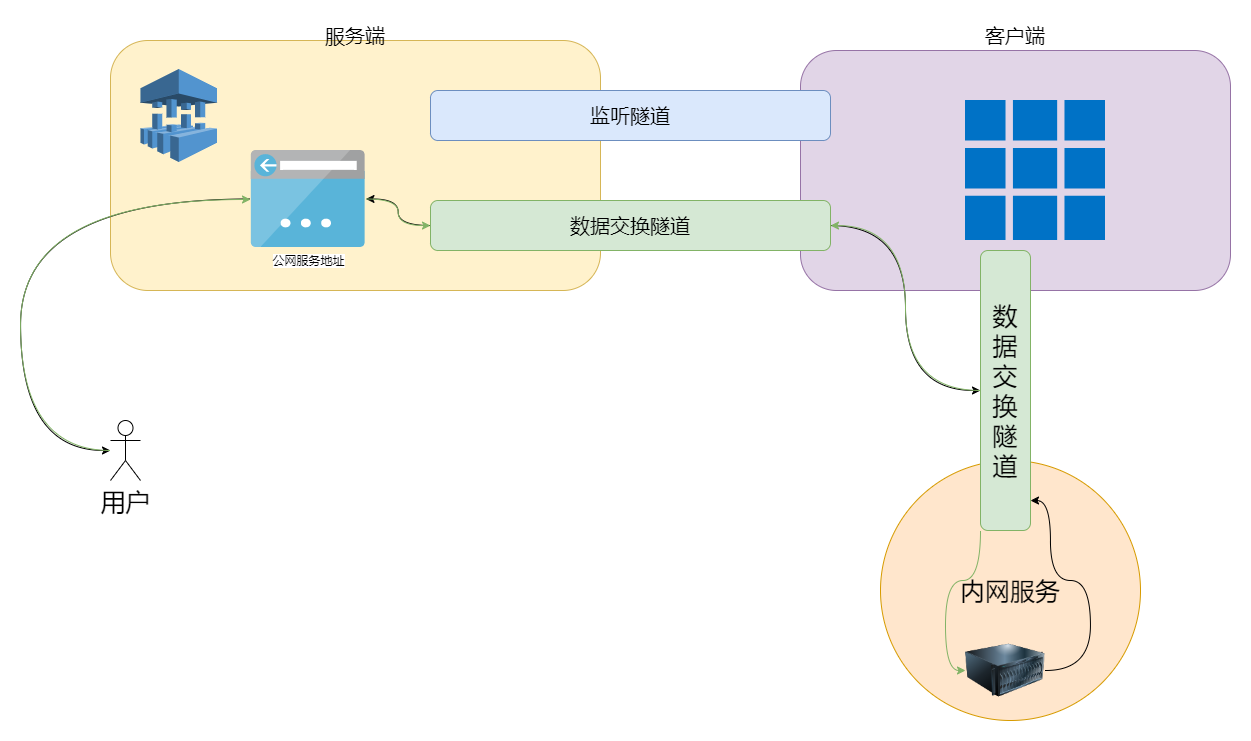
### 2.4. 术语和缩略词

1. **NAT**：网络地址转换（Network Address Translation），一种将私有IP地址转换为公共IP地址的网络技术。
2. **LAN**：局域网（Local Area Network），指在相对较小的范围内，如家庭、办公室或学校等地方连接的计算机网络。
3. **互联网**（Internet）：全球范围内相互连接的计算机网络，通过TCP/IP协议族进行数据传输。
4. **内网穿透**：通过网络隧道或代理等方式，将公共网络上的请求转发到私有网络中的设备或服务。
5. **端口映射**：将公共网络上的端口请求映射到私有网络中的特定设备或服务的过程。
6. **TCP**/IP：传输控制协议/互联网协议（Transmission Control Protocol/Internet Protocol），互联网通信的基本协议。
7. **API**：应用程序编程接口（Application Programming Interface），用于不同软件组件之间的通信和交互的一组定义。
8. **VPN**：虚拟专用网络（Virtual Private Network），通过公共网络建立加密通道，实现远程访问私有网络的安全连接。
9. **NAT**类型：对NAT设备进行分类的标识，如全锥形NAT、限制锥形NAT等，用于描述网络连接的可达性和访问限制。

## 3. 架构概述

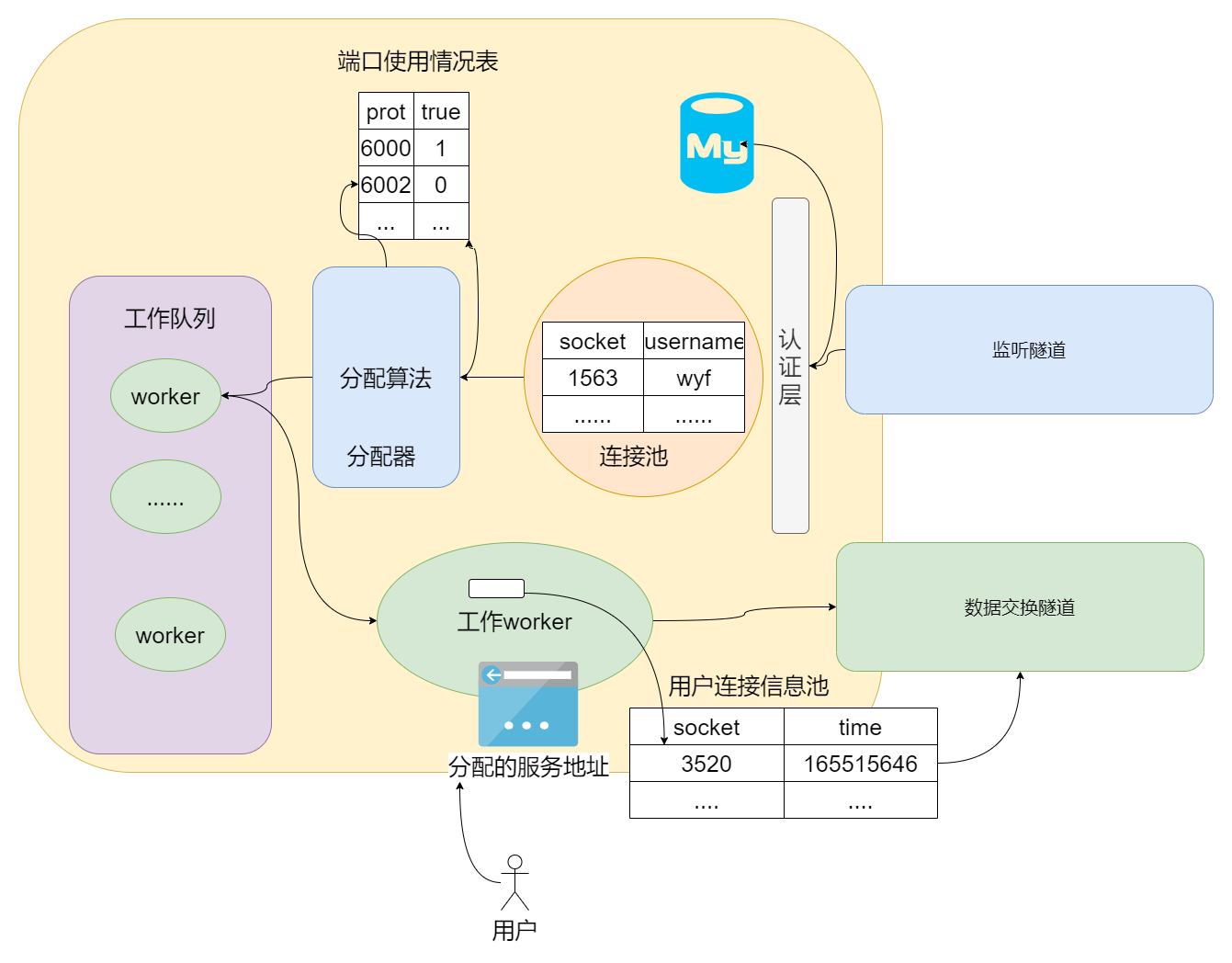
### 3.1. 系统架构

整体的系统架构为客户端-服务器模式，整体的视图如下所示：



整体的系统视图如上面所示，用户会根据公网服务地址访问到位于内网里的服务。当开发者想代理内部的服务时，只需要与服务端建立连接，服务端会根据实时使用情况向客户端分配一个限制空余的端口代理该服务；用户根据这个端口访问开发者所开发的内网服务。服务端和客户端详细的架构在下方展示。

### 3.2. 服务端架构

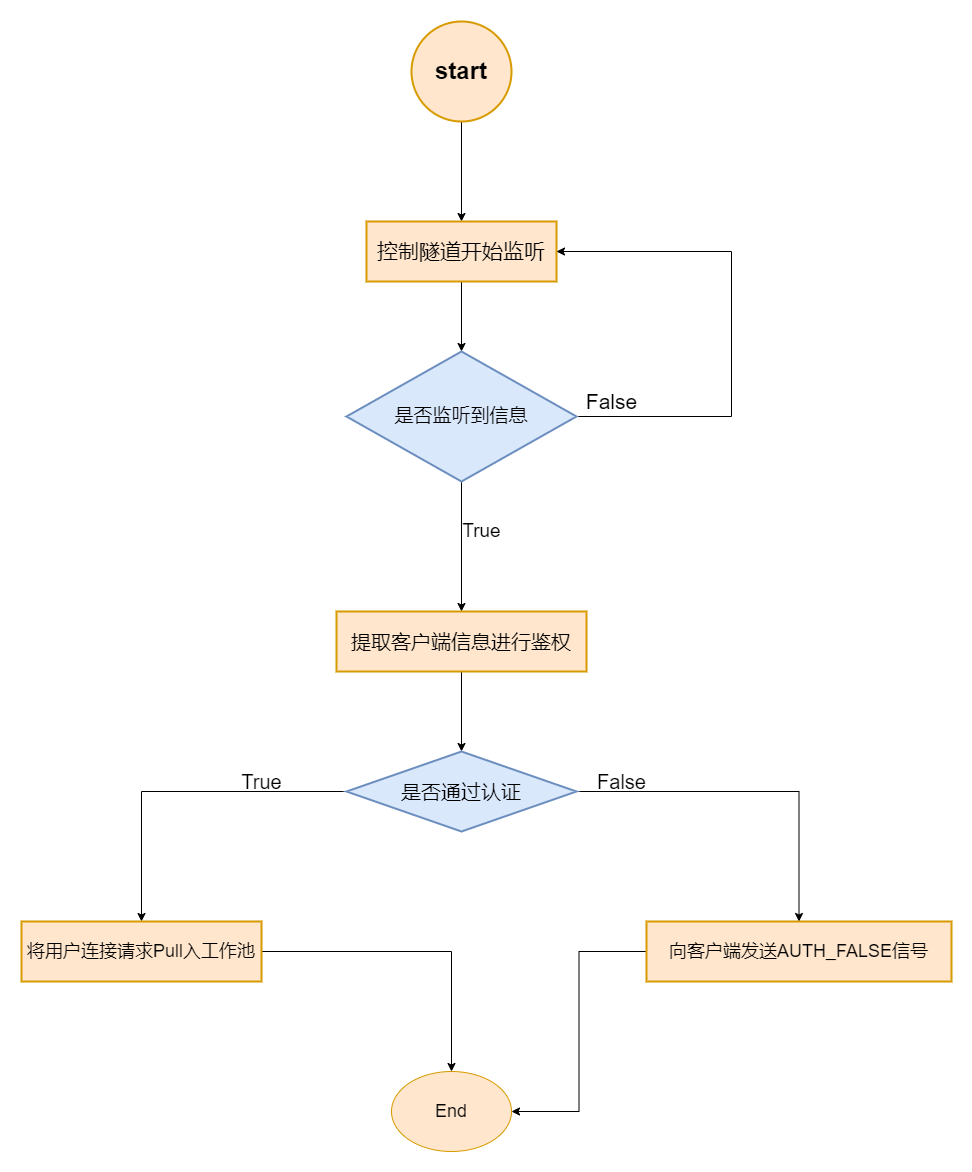


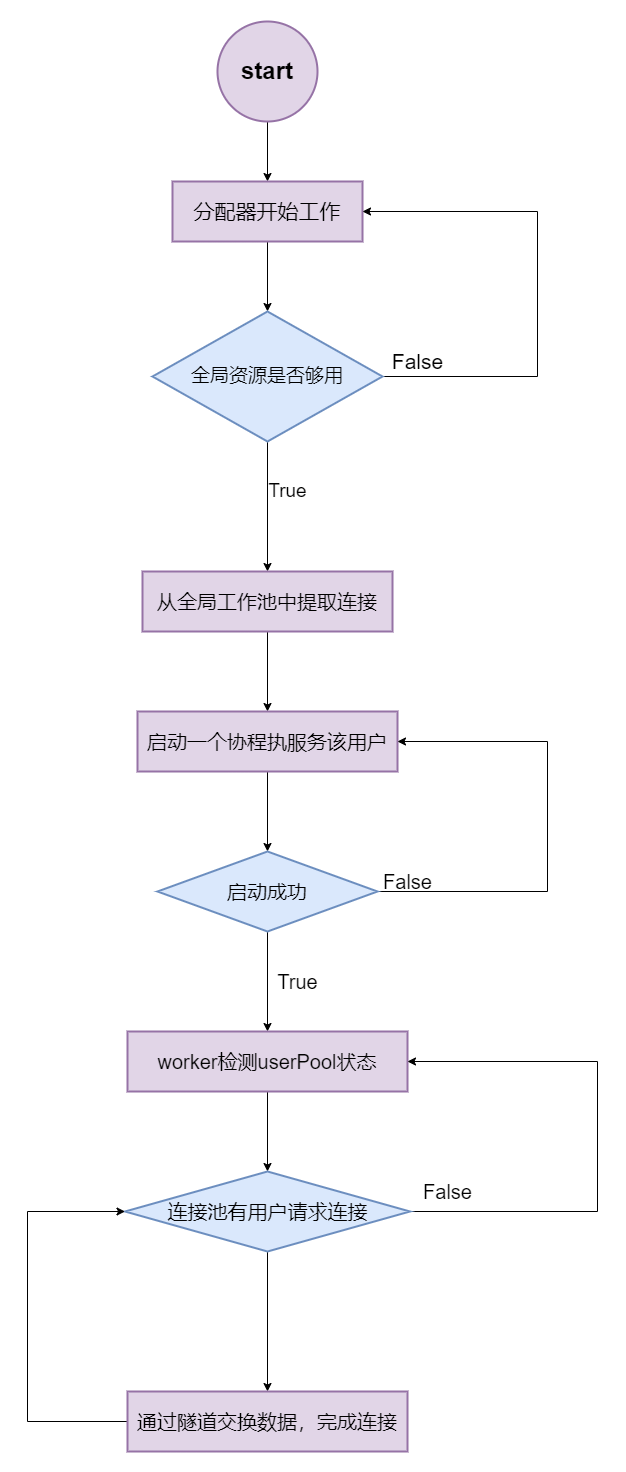
服务端的处理流程大致是这样的：

服务端启动一个控制接口，用于传递控制信息（类似于ftp协议），该控制接口有两个作用，**监听客户端发送的连接请求**和**传递控制信息**；当监听到用户发送过来的请求时，会从TCP stream流中先提取出用户信息(username，password),紧接着，服务端会根据用户信息进行校验，校验的内容包括用户是否注册以及用户账户是否过期;如果用户没有通过校验，则向客户端发送AUTH\_FAIL类型的信号，表示认证失败；如果通过了校验，则会将用户的信息pull入到连接池中去，至此，控制端的功能到此为止，紧接着就是整个服务端的核心内容了；

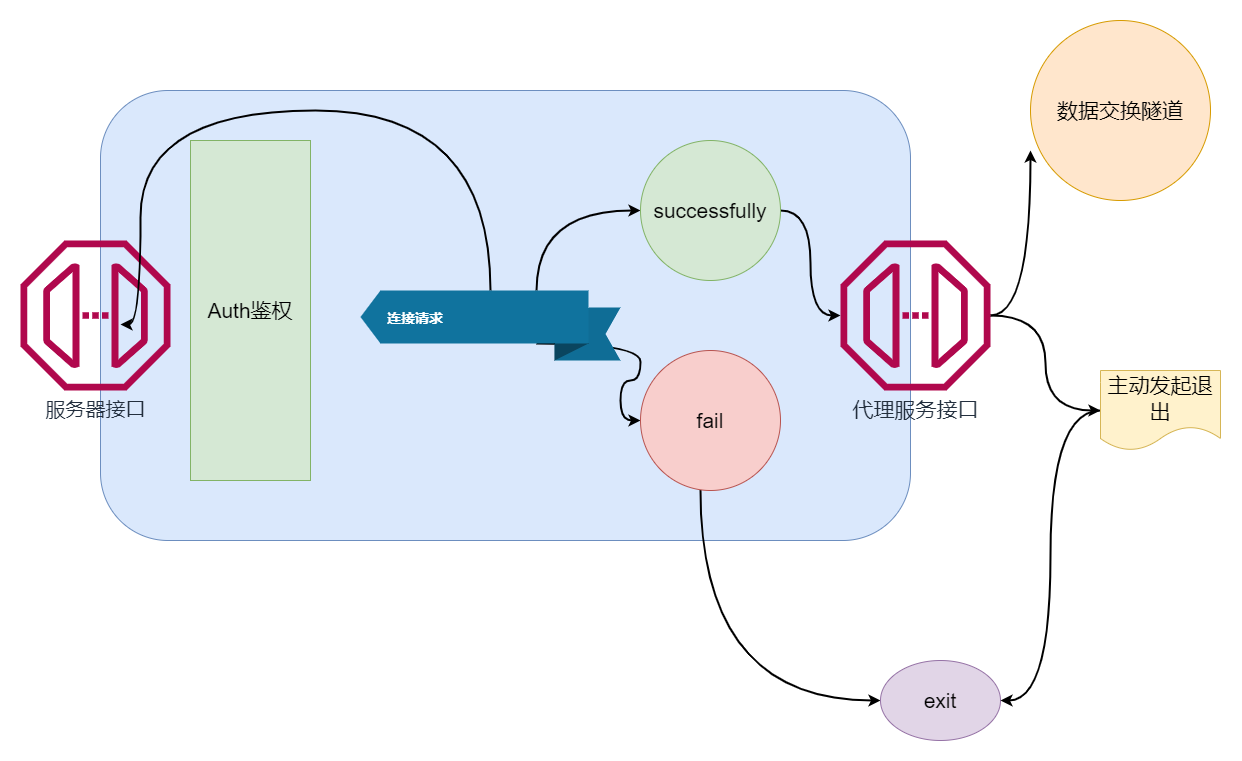
**分配器**会实时的检测整个服务端的资源的情况，如果服务端有充足的资源够客户端使用时，分配器会给该客户端分配一个代理port，并在全局记录下分配信息；与此同时开辟一个goroutine(协程)去处理该连接请求。

**worker**就是实际处理真实业务的线程，该worker里面存储着本次连接的所有信息如：ID(客户端ID)、ClientConn(客户端连接socket)、ServerListener(服务端代理socket)、Port、UserConnPool(用户连接池)、CurrentTransmitBytes(转发字节数)、single(信号);**worker**会根据userPool里面的连接信息，通过数据交换隧道实现代理的功能。





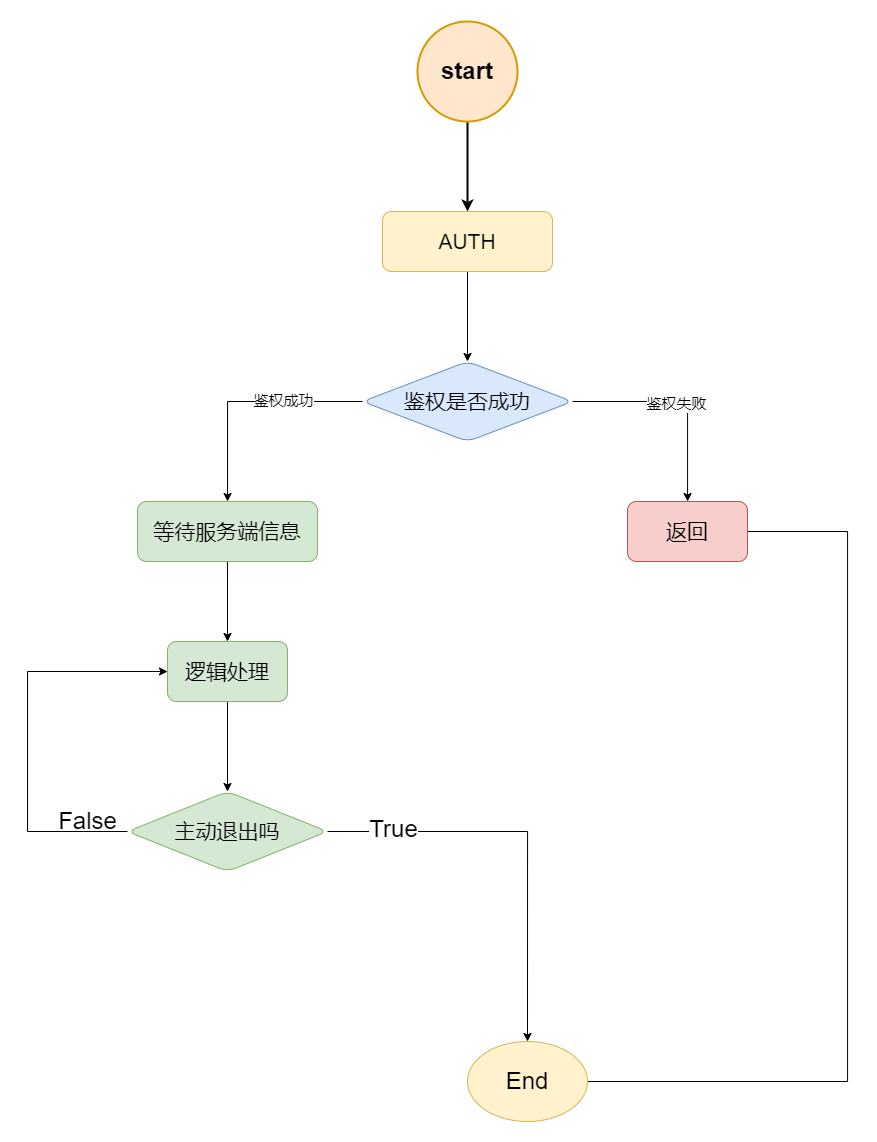
### 3.3. 客户端架构



客户端的流程就十分的简单了：

首先会将自己的基本信息封装好(username,password)；向服务端发送建立连接的请求，这时如果没有收到AUTH\_FAIL这个信号，那么代表着认证成功了，这时程序会一直阻塞到这里直到收到服务端发来的信号USER\_INFORMATION这代表着服务端已经准备好了这次会话所有的资源了，可以进行连接了。这时客户端就正确的启动了。

客户端的流程如下图所示：



### 3.4. 外部依赖关系

本项目依赖数据库来保存用户信息和实现鉴权功能。我们选择使用MYSQL 8.0版本作为存储解决方案。

## 4. 安装和配置

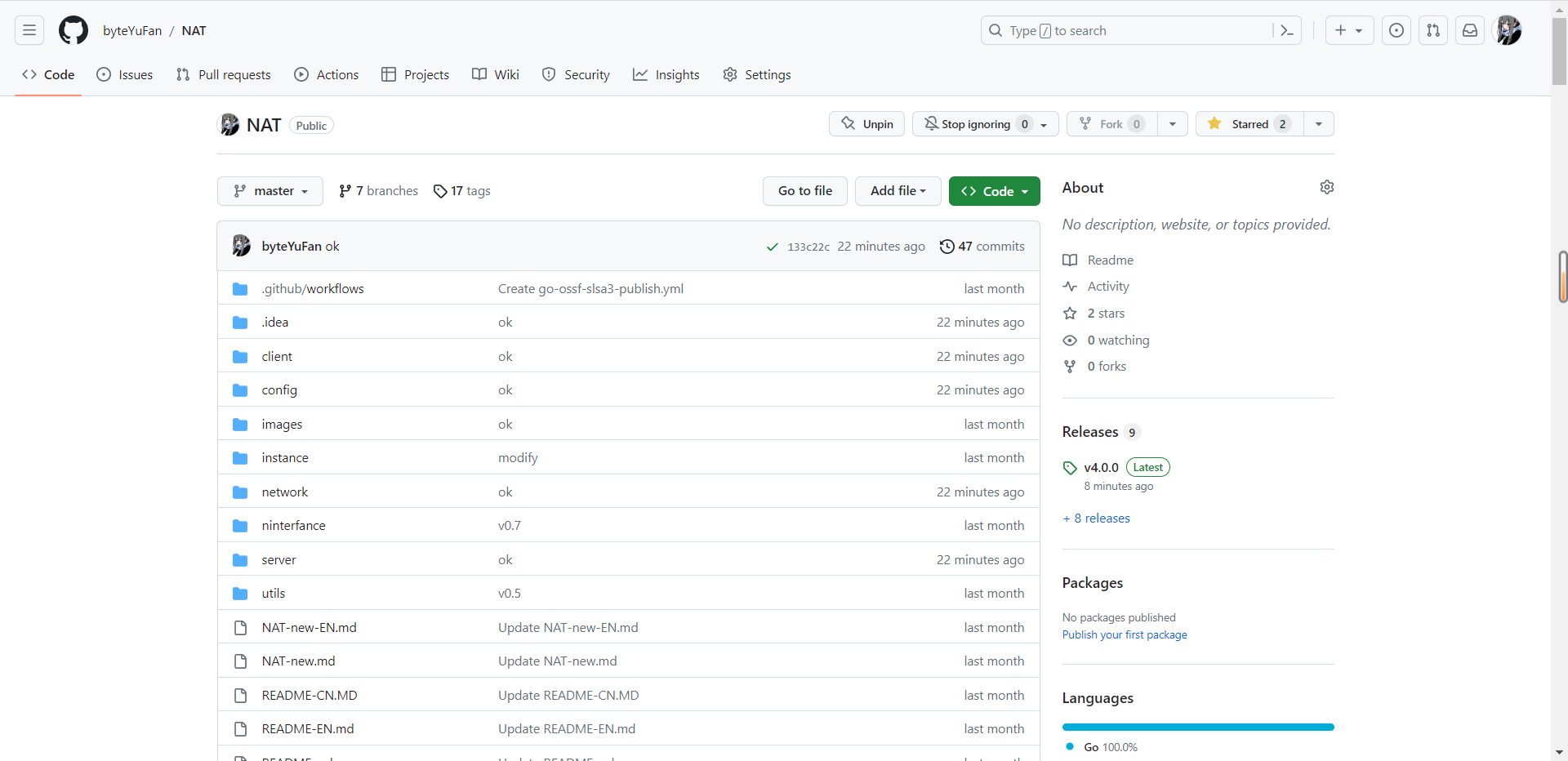
### 4.1. 安装教程

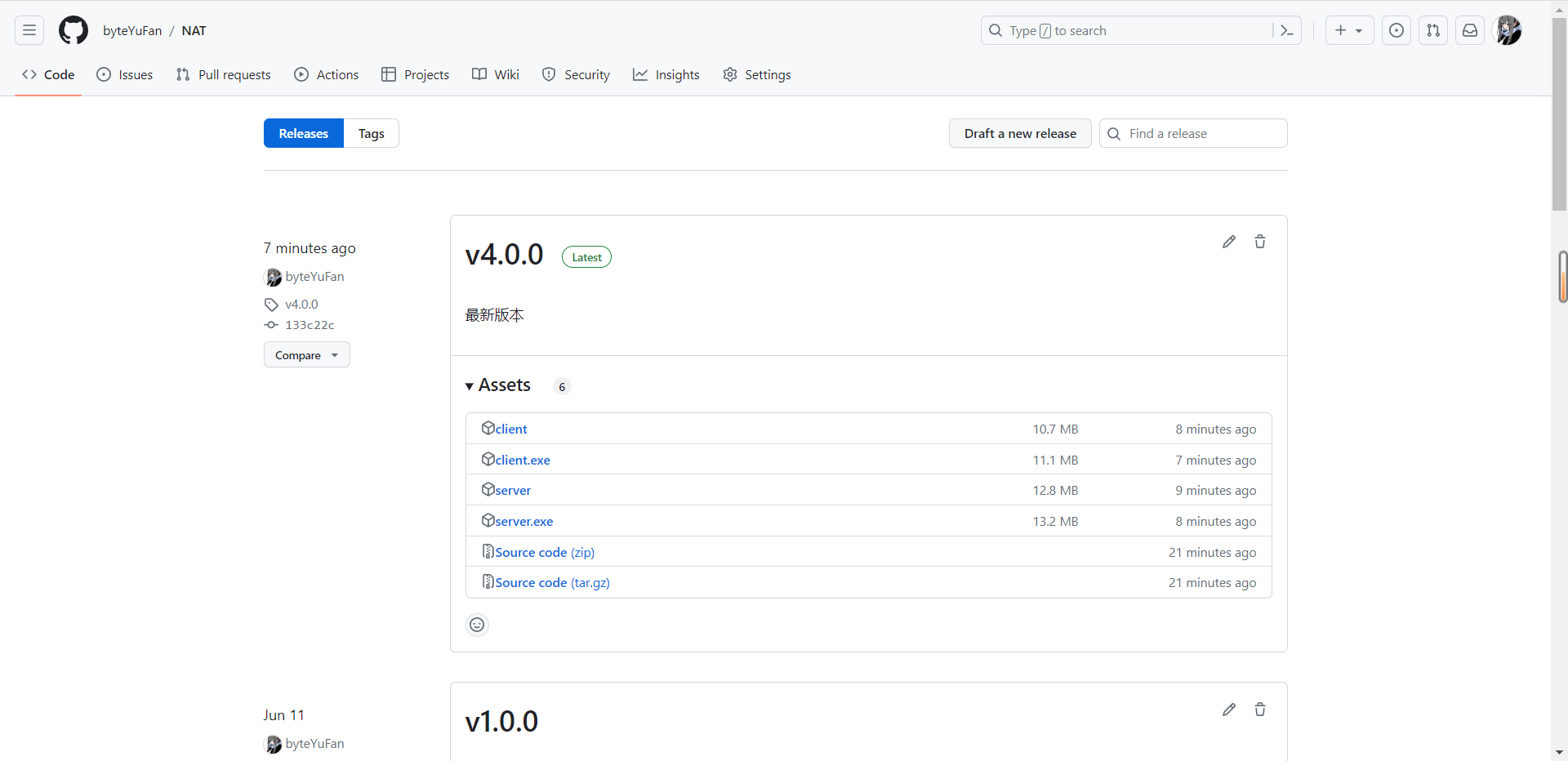
您可以通过项目的[github](https://github.com/byteYuFan/NAT)地址去下载最新版本的代码，并编译他。或者您也可以直接找release进行下载使用。

地址附带如下：

github：<https://github.com/byteYuFan/NAT>

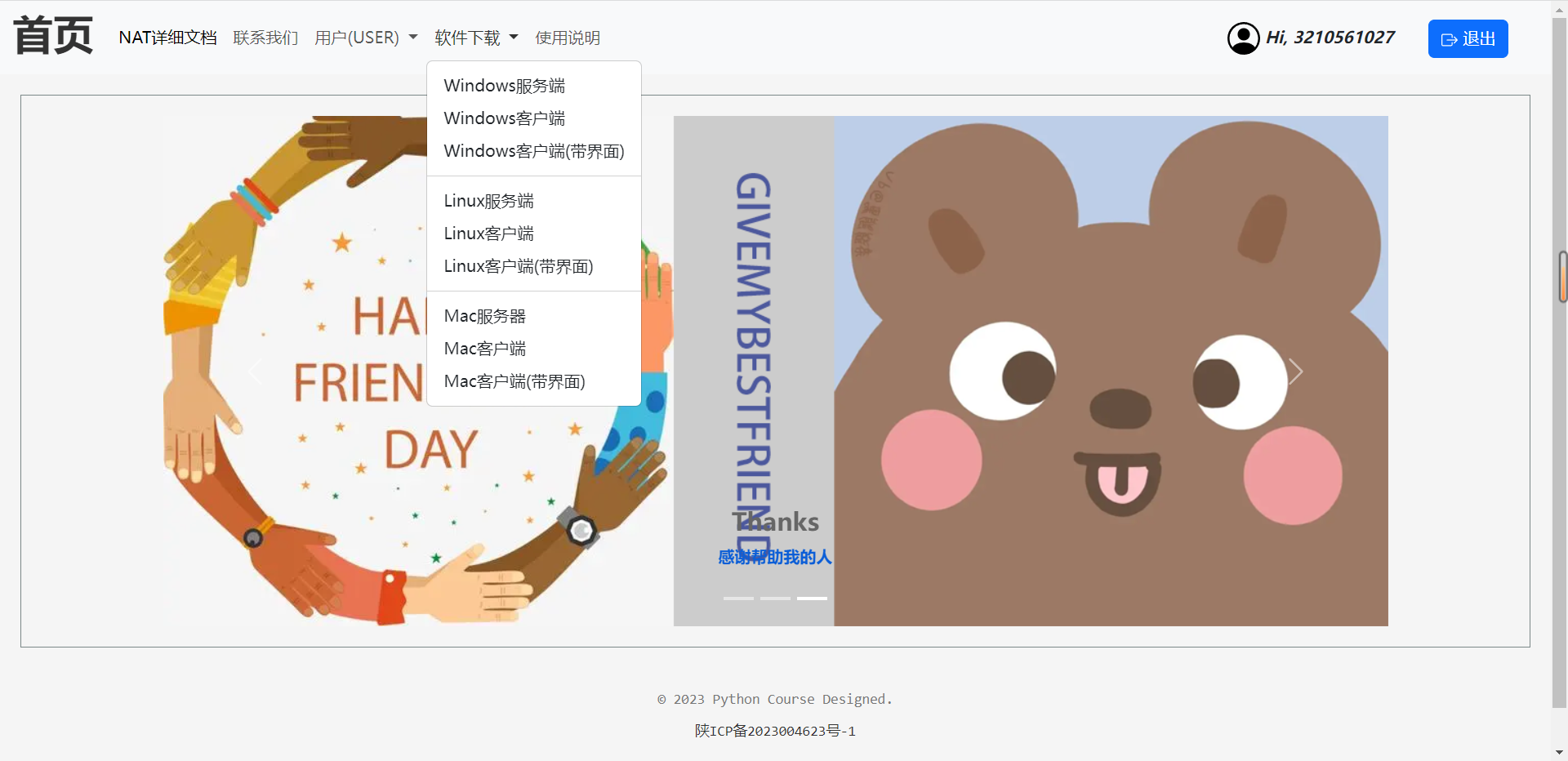
release：<https://github.com/byteYuFan/NAT/releases>





或者您也可以通过访问我们的官网进行下载使用：

官网地址pogf.com.cn:8000,你可以根据自己需要的版本进行下载使用



### 4.2. 数据库配置

您需要将这两张表转存到您的数据库中去，这样您就可以正常的使用认证功能了。

CREATE TABLE `userInfo\_userinfo` (   
 `user\_id` int NOT NULL AUTO\_INCREMENT,   
 `username` varchar(255) NOT NULL,   
 `password` varchar(255) NOT NULL,   
 `email` varchar(254) DEFAULT '00000000@test.com',   
 `flag` tinyint(1) NOT NULL DEFAULT '0',   
 `time` datetime(6) DEFAULT NULL ON UPDATE CURRENT\_TIMESTAMP(6),   
 PRIMARY KEY (`user\_id`)   
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_0900\_ai\_ci;

CREATE TABLE `bars` (   
 `id` bigint NOT NULL AUTO\_INCREMENT,   
 `bar` bigint NOT NULL,   
 `username` varchar(255) NOT NULL,   
 PRIMARY KEY (`id`)   
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_0900\_ai\_ci;

## 5. 使用说明

### 5.1. 操作说明

#### 1. 服务端

根据自己的需求将服务端上传到指定的为止，这里我使用的是腾讯云服务器，在root的模式下，给其增加执行权限并查看其详细的信息:

[root@VM-4-7-centos ~]# chmod a+x server   
[root@VM-4-7-centos ~]# ll server   
-rwxr-xr-x 1 root root 13379532 Jul 16 17:33 server

配置脚本：

本项目采用yaml文件的配置方式,如下所示：

Server:   
 Name: "Server-NAT"   
 ControllerAddr: "0.0.0.0:8080"   
 TunnelAddr: "0.0.0.0:8008"   
 VisitPort:   
 - 60000   
 - 60001   
 - 60002   
 - 60003   
 TaskQueueNum: 4   
 TaskQueueBuff: 32   
 MaxTCPConnNum: 4   
 MaxConnNum: 256   
 LogFilename: "server.log"   
 StartAuth: true   
 StartLog: true   
Database:   
 Username: "root"   
 Password: "123456"   
 Host: "0.0.0.0:3309"   
 DBName: "nat"

这个配置文件包含了服务器（Server）和数据库（Database）的相关配置信息。下面是每个配置项的含义：

Server配置：

* Name: 服务器的名称，此处设置为 "Server-NAT"。
* ControllerAddr: 控制器的地址，用于与客户端进行通信，此处设置为 "0.0.0.0:8080"，表示监听所有可用的网络接口和端口8080。
* TunnelAddr: 隧道地址，用于与其他服务器或设备建立隧道连接，此处设置为 "0.0.0.0:8008"。
* VisitPort可访问的端口列表，这里列出了一组端口号 [60000, 60001, 60002, 60003]，表示允许访问这些端口的连接。
* TaskQueueNum: 任务队列数量，指定用于处理请求的任务队列的数量，此处设置为 4。
* TaskQueueBuff: 任务队列缓冲大小，用于缓存待处理请求的数量，此处设置为 32。
* MaxTCPConnNum: 最大TCP连接数，限制允许的最大TCP连接数量，此处设置为 4。
* MaxConnNum: 最大连接数，限制允许的最大连接数量，此处设置为 256。
* LogFilename: 日志文件名，指定日志输出的文件名，此处设置为 "server.log"。
* StartAuth: 是否启用身份验证，设置为 true 表示启用身份验证功能。
* StartLog: 是否启用日志记录，设置为 true 表示启用日志记录功能。

Database配置：

* Username: 数据库用户名，设置为 "root"。
* Password: 数据库密码，设置为 "123456"。
* Host: 数据库主机地址，设置为 "0.0.0.0:3309"。
* DBName: 数据库名称，设置为 "nat"。

根据自己本地的需求之后配置好serber.yml问价之后，就可以启动程序了：

[root@VM-4-7-centos ~]# ./server   
 \_ \_ \_\_\_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_\_ \_\_\_\_\_ \_\_ \_\_ \_\_\_\_\_\_ \_\_\_\_\_   
 | \ | | /\ |\_\_ \_\_| / \_\_\_\_| | \_\_\_\_| | \_\_ \ \ \ / / | \_\_\_\_| | \_\_ \   
 | \| | / \ | | | (\_\_\_ | |\_\_ | |\_\_) | \ \ / / | |\_\_ | |\_\_) |   
 | . ` | / /\ \ | | \\_\_\_ \ | \_\_| | \_ / \ \/ / | \_\_| | \_ /   
 | |\ | / \_\_\_\_ \ | | \_\_\_\_) | | |\_\_\_\_ | | \ \ \ / | |\_\_\_\_ | | \ \   
 |\_| \\_| /\_/ \\_\ |\_| |\_\_\_\_\_/ |\_\_\_\_\_\_| |\_| \\_\ \/ |\_\_\_\_\_\_| |\_| \\_\   
   
   
[ServerName] Server-NAT   
[MaxServerConn] 4   
[服务端开启端口] [60000 60001 60002 60003]   
[ListenTaskQueue] 正在监听工作队列传来的信息……   
[服务器控制端开始监听]0.0.0.0:8080   
[Start Auth Successfully!] 服务器开启认证请求

#### 2. 客户端

这里我选择将客户端下载到本地Windows电脑上进行测试，已经下载成功，现在进行相关配置：

Auth:   
 Username: "3210561027"   
 Password: "00000000"   
Client:   
 Name: Client-NAT   
 PublicServerAddr: pogf.com.cn   
 TunnelServerAddr: pogf.com.cn:8008   
 ControllerAddr: pogf.com.cn:8080   
 LocalServerAddr: 0.0.0.0:8080

这个配置文件包含了身份验证（Auth）和客户端（Client）的相关配置信息。下面是每个配置项的含义：

Auth配置：

* Username: 用户名，设置为 "3210561027"。
* Password: 密码，设置为 "00000000"。

Client配置：

* Name: 客户端的名称，此处设置为 "Client-NAT"。
* PublicServerAddr: 公共服务器地址，表示代理服务器的地址。
* TunnelServerAddr: 隧道服务器地址。
* ControllerAddr: 控制器地址，用于与服务器进行通信。
* LocalServerAddr: 本地服务器地址，用于提供本地服务，设置为 0.0.0.0:8000，表示监听所有可用的网络接口和端口8080。

这些配置项定义了身份验证和客户端的各种参数和选项，确保客户端能够按照指定的设置与服务器进行通信和访问服务。请注意，根据你的实际需求，你可能需要根据具体情况进行调整和修改配置文件中的值。

### 5.2. 内网穿透功能代理内部服务

测试说明：

在本地启动一个8080端口的服务，通过内网穿透软件代理该服务，分别使用curl，浏览器,安卓手机进行测试，看是否可以访问到该服务。

package main   
   
import (   
 "github.com/gin-gonic/gin"   
 "log"   
)   
   
func main() {   
 // 创建一个Gin的默认引擎   
 r := gin.Default()   
   
 // 定义一个路由处理函数   
 r.GET("/", func(c \*gin.Context) {   
 c.JSON(200, gin.H{   
 "message": "Hello, World!",   
 })   
 })   
   
 // 启动Web服务，监听在8080端口   
 err := r.Run(":8080")   
 if err != nil {   
 log.Fatal("启动Web服务失败: ", err)   
 }   
}

这段代码的作用是创建一个简单的Web服务，并定义了一个根路由处理函数，返回一个包含"Hello, World!"的JSON响应。

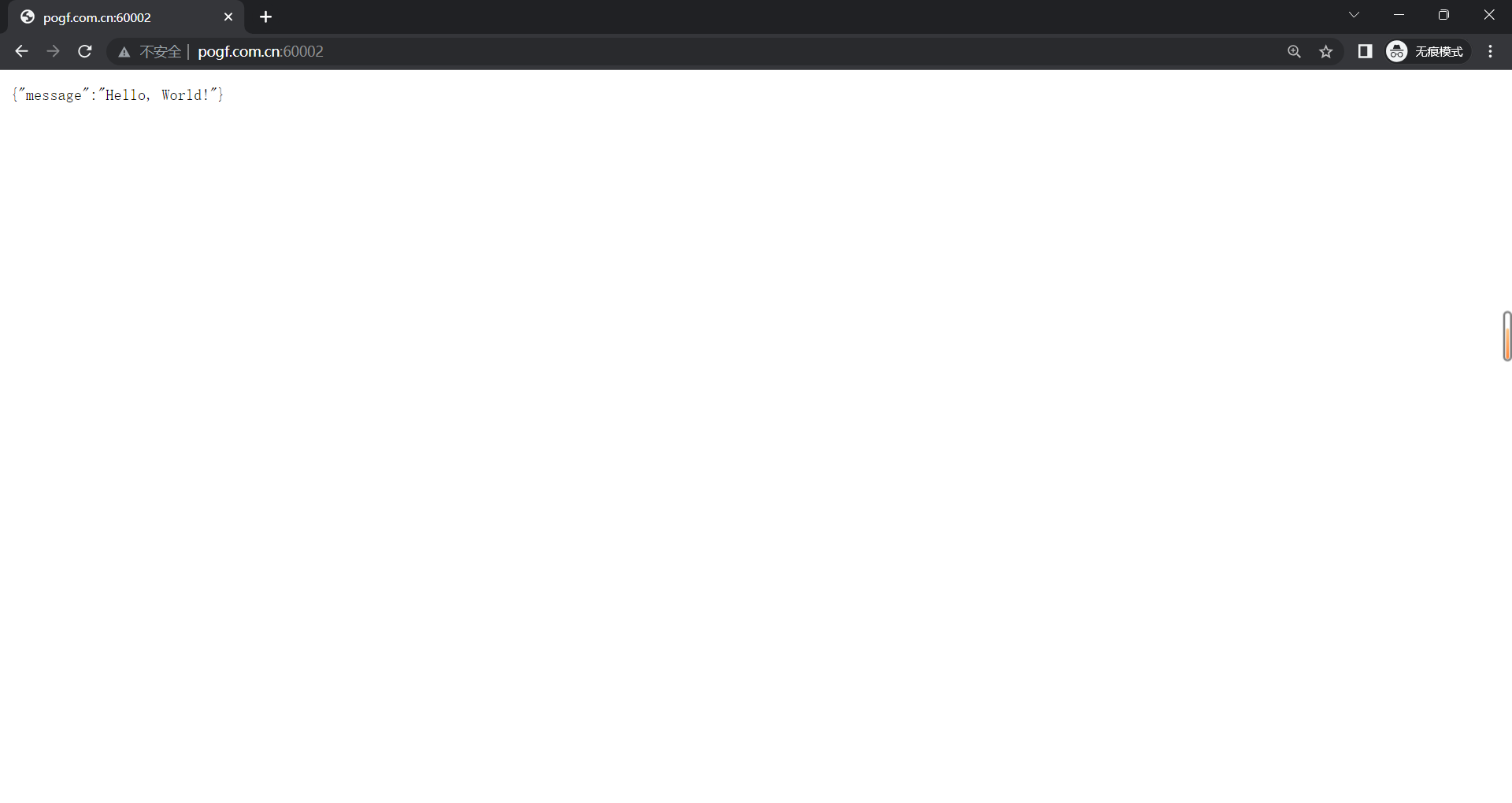
现在我们使用内网穿透软件代理其功能：

$ ./client.exe   
 \_ \_ \_\_\_\_\_\_\_ \_\_\_\_\_ \_ \_\_\_\_\_ \_\_\_\_\_\_ \_ \_ \_\_\_\_\_\_\_   
 | \ | | /\ |\_\_ \_\_| / \_\_\_\_| | | |\_ \_| | \_\_\_\_| | \ | | |\_\_ \_\_|   
 | \| | / \ | | | | | | | | | |\_\_ | \| | | |   
 | . ` | / /\ \ | | | | | | | | | \_\_| | . ` | | |   
 | |\ | / \_\_\_\_ \ | | | |\_\_\_\_ | |\_\_\_\_ \_| |\_ | |\_\_\_\_ | |\ | | |   
 |\_| \\_| /\_/ \\_\ |\_| \\_\_\_\_\_| |\_\_\_\_\_\_| |\_\_\_\_\_| |\_\_\_\_\_\_| |\_| \\_| |\_|   
   
   
[Client Running Successfully!]   
[PublicAddress] pogf.com.cn   
[TunnelAddress] pogf.com.cn:8008   
[LocalAddress] 127.0.0.1:8080   
[Conn Successfully]pogf.com.cn:8080   
[auth] 发送认证消息成功。   
[接收到USER\_INFORMATION]   
[ClientInfoUID] 1   
[VisitAddress] pogf.com.cn:60002

我们使用curl工具去测试pogf.com.cn:60002:

C:\Users\wangyufan>curl pogf.com.cn:60002   
{"message":"Hello, World!"}

发现没有任何问题，我们通过谷歌浏览器无痕窗口访问它:



也没有任何问题，我们使用手机端进行访问：



完全正确。

### 5.3. 测试代理Redis服务

测试说明：

在本地启动Redis服务，通过内网穿透软件进行代理，通过服务端的地址向redis插入一条数据，然后在本地进行查看，如果在本地看到了这条消息，则说明代理成功：

初始redis情况

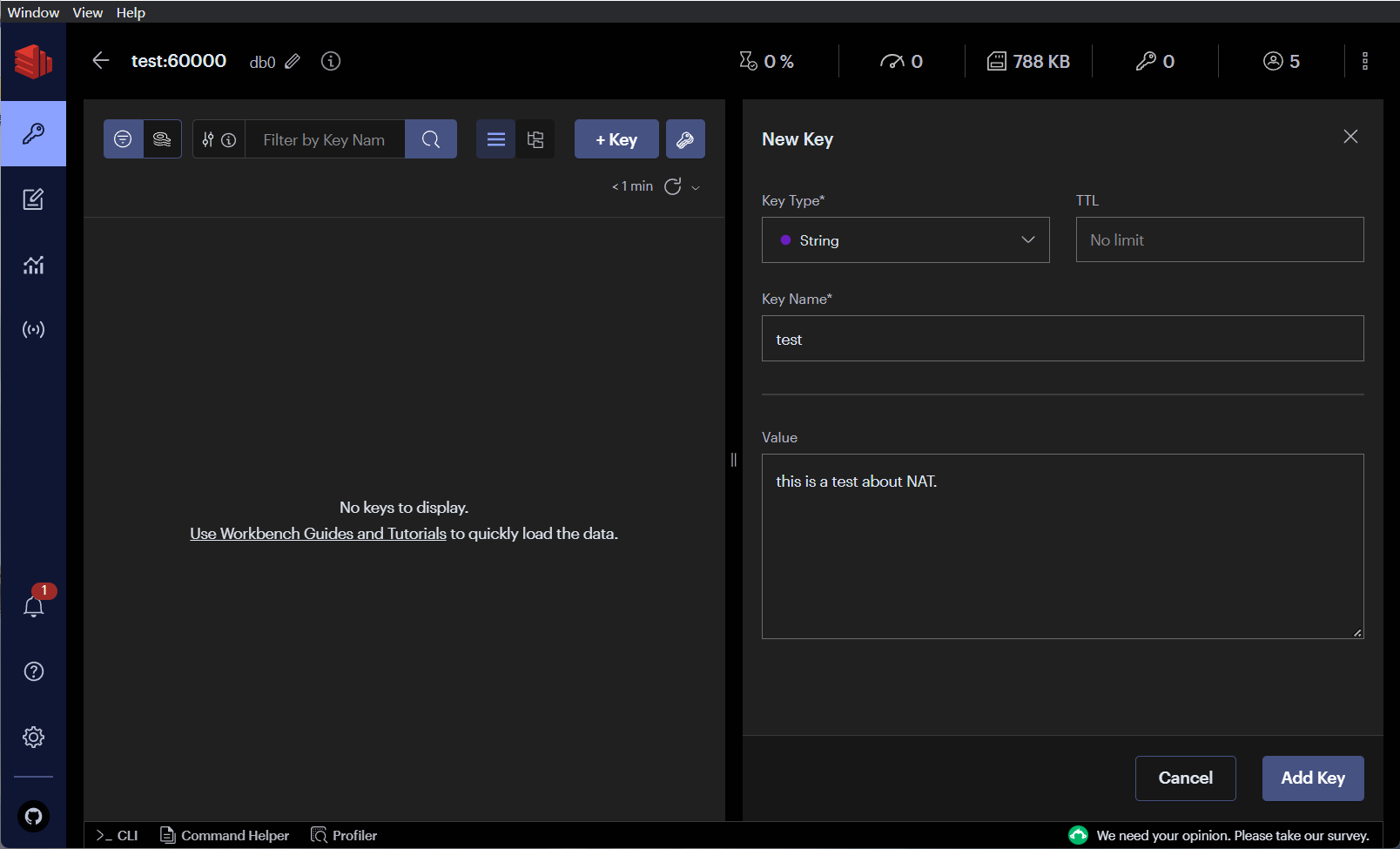
127.0.0.1:6379> keys \*   
(empty list or set)   
127.0.0.1:6379>

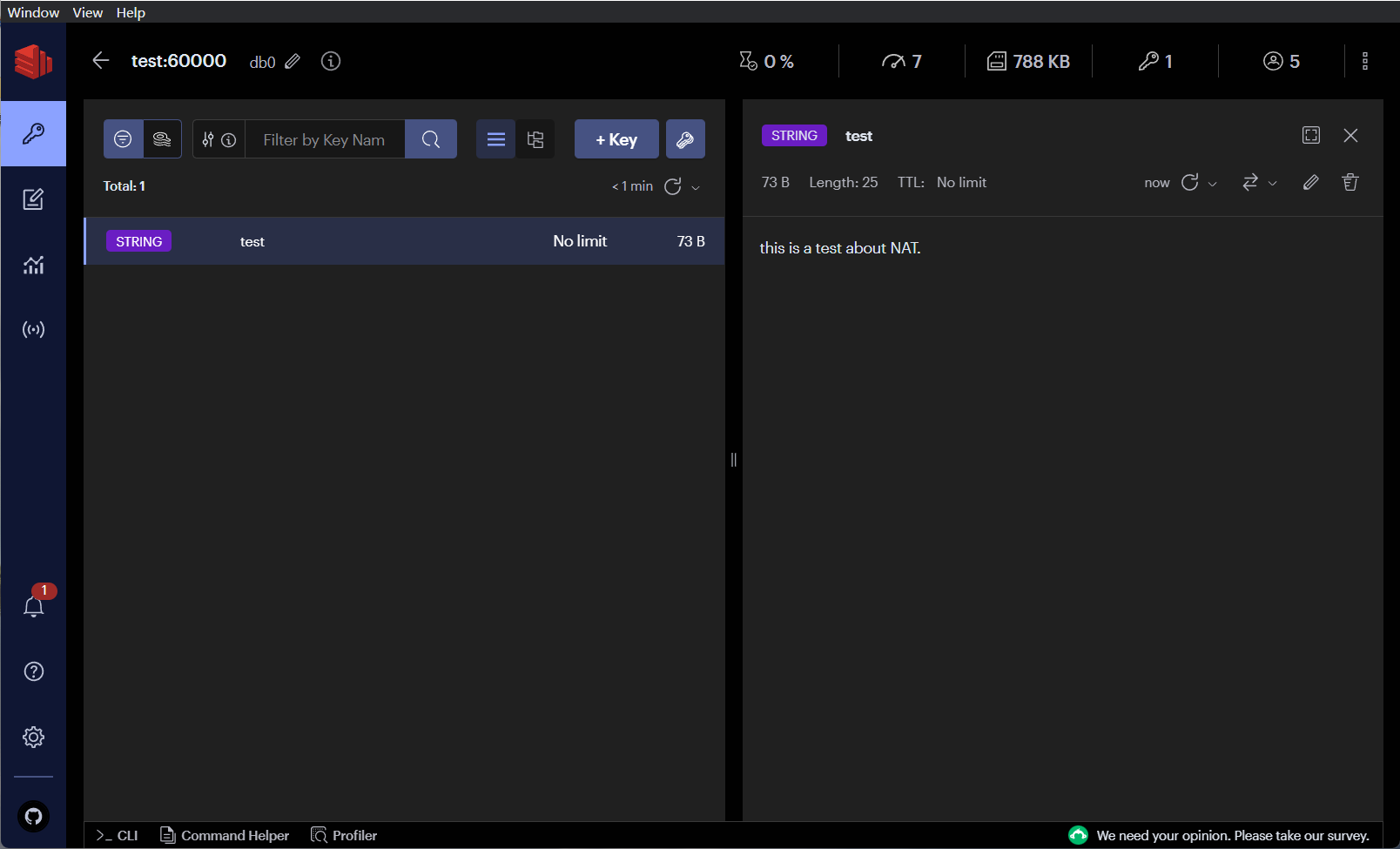
更改客户端配置并启动客户端:

LocalServerAddr: "127.0.0.1:6379"

[Client Running Successfully!]   
[PublicAddress] pogf.com.cn   
[TunnelAddress] pogf.com.cn:8008   
[LocalAddress] 127.0.0.1:6379   
[Conn Successfully]pogf.com.cn:8080   
[auth] 发送认证消息成功。   
[接收到USER\_INFORMATION]   
[ClientInfoUID] 1   
[VisitAddress] pogf.com.cn:60000

通过redis客户端工具连接pogf.com.cn:60000这个地址，并向里面插入一条数据：





成功插入，现在我们在本地查看刚刚插入的这条数据：

127.0.0.1:6379> keys \*   
1) "test"   
127.0.0.1:6379> get test   
"this is a test about NAT."

完全正确。

### 5.4. 局域网内屏蔽实际地址

通过上述的例子，我们不难明白，在局域网中，我们可以通过一台主机作为服务器，使用该主机的IP作为我们服务的访问地址，可以屏蔽我们服务的实际地址。

## 6. 开发核心

以下内容将是整个系统的核心开发过程。主要分为**协议概述**、**服务器**、**客户端**这三个部分组成，我们也从这三个角度进行介绍NAT项目。

### 6.1. 协议概述

我们知道，TCP传输数据的特定是字节流的，因此，当我们发送一些信息时，有可能这些包会被黏在一起，客户端收到了不该出现的信息，这是我们万万不希望出现的，因此我们在这块定义相关的发包和收包的协议以避免发生这些问题。

黏包问题通常是由应用层协议设计不合理造成的，而不是TCP协议本身引起的。

在应用层协议中，黏包问题可能出现的原因包括：

1. **消息边界模糊**：应用层协议没有明确定义消息的边界，导致接收方无法准确切分和解析消息。
2. **数据的粘连**：发送方连续发送多个消息，但这些消息在传输过程中被合并为一个或部分粘连在一起，导致接收方无法正确判断消息的边界。
3. **消息长度不固定**：应用层协议中的消息长度不固定，导致发送方和接收方在切分和解析消息时出现不一致。
4. **处理延迟**：接收方在处理接收到的消息时存在延迟，而发送方持续发送消息，导致已接收和待接收的消息粘连在一起。

为了解决应用层协议的黏包问题，可以采取以下一些常见的解决方案：

1. **消息长度字段**：在应用层协议中，引入消息长度字段，使接收方能够根据长度字段准确切分消息。
2. **消息分隔符**：定义特定的消息分隔符，在消息中插入分隔符以标识消息的边界。
3. **消息头标识**：在消息头中添加特定的标识符，用于标识消息的开始和结束。
4. **消息序号**：在每个消息中添加序号，接收方可以根据序号判断消息的完整性。
5. **应用层缓冲区**：接收方使用应用层缓冲区，将接收到的数据存储起来，然后根据应用层协议的定义进行消息的切分和处理。

#### 1. 消息接口定义

// IMessage 将请求的消息封装到一个Message中，定义抽象的接口   
type IMessage interface {   
 // GetMsgID 获取消息的ID   
 GetMsgID() uint32   
 // GetMsgDataLen 获取消息长度   
 GetMsgDataLen() uint32   
 // GetMsgData 获取消息内容   
 GetMsgData() []byte   
 // SetMsgID 设置消息的ID   
 SetMsgID(uint32)   
 // SetMsgLen 设置消息长度   
 SetMsgLen(uint32)   
 // SetMsgData 设置消息内容   
 SetMsgData([]byte)   
}

该接口IMessage定义了封装请求消息到Message的方法，并提供了一些抽象的操作消息的方法。以下是对该接口的描述：

* GetMsgID() uint32：获取消息的ID，返回一个uint32类型的消息ID。
* GetMsgDataLen() uint32：获取消息的长度，返回一个uint32类型的消息长度。
* GetMsgData() []byte：获取消息的内容，返回一个[]byte类型的消息内容。
* SetMsgID(uint32)：设置消息的ID，接受一个uint32类型的参数，用于设置消息的ID。
* SetMsgLen(uint32)：设置消息的长度，接受一个uint32类型的参数，用于设置消息的长度。
* SetMsgData([]byte)：设置消息的内容，接受一个[]byte类型的参数，用于设置消息的内容。

#### 2. 消息接口实现

// Message 该结构体为消息类型，其中包括消息的ID编号，  
// 消息的长度 消息的内容  
// 其中 ID 和 DataLen 为固定长度大小 为 4 + 4 = 8 字节  
// Data 为消息的具体内容 其中DataLen为Data的大小  
type Message struct {  
 // ID 消息ID  
 ID uint32  
 // DataLen 消息长度  
 DataLen uint32  
 // Data 消息的内容  
 Data []byte  
}  
// NewMsgPackage 新建一个Message消息的包 传入的参数为消息的type-id 消息的实际内容  
func NewMsgPackage(id uint32, data []byte) \*Message {  
 return &Message{  
 ID: id,  
 DataLen: uint32(len(data)),  
 Data: data,  
 }  
}  
  
// GetMsgID 获取消息的ID  
func (m \*Message) GetMsgID() uint32 {  
 return m.ID  
}  
  
// GetMsgDataLen 获取消息长度  
func (m \*Message) GetMsgDataLen() uint32 {  
 return m.DataLen  
}  
  
// GetMsgData 获取消息内容  
func (m \*Message) GetMsgData() []byte {  
 return m.Data  
}  
  
// SetMsgID 设置消息的ID  
func (m \*Message) SetMsgID(id uint32) {  
 m.ID = id  
}  
  
// SetMsgLen 设置消息长度  
func (m \*Message) SetMsgLen(len uint32) {  
 m.DataLen = len  
}  
  
// SetMsgData 设置消息内容  
func (m \*Message) SetMsgData(data []byte) {  
 m.Data = data  
}

* ID uint32：消息的ID编号，占据固定长度的4字节。
* DataLen uint32：消息的长度，占据固定长度的4字节。
* Data []byte：消息的具体内容，长度由DataLen指定。

此外，代码还提供了一些方法来操作Message结构体：

* NewMsgPackage(id uint32, data []byte) \*Message：创建一个新的Message消息对象，传入消息的ID和实际内容作为参数。
* GetMsgID() uint32：获取消息的ID。
* GetMsgDataLen() uint32：获取消息的长度。
* GetMsgData() []byte：获取消息的内容。
* SetMsgID(id uint32)：设置消息的ID。
* SetMsgLen(len uint32)：设置消息的长度。
* SetMsgData(data []byte)：设置消息的内容。

#### 3. 传递消息接口

// IDataPack 抽象层封装包解决TCP黏包问题的拆包封装包的模块  
// 针对Message进行TLV格式的封装  
// 针对Message进行TLV格式的拆包  
// 先读取固定长度的head-->消息的长度和消息的类型  
// 在根据消息内容的长度，在读取内容  
// 直接面向TCP连接的数据流 TCP stream  
type IDataPack interface {  
 // GetHeadLen 获取包的头长度  
 GetHeadLen() uint32  
 // Pack 封包  
 Pack(msg IMessage) ([]byte, error)  
 // Unpack 拆包  
 Unpack([]byte) (IMessage, error)  
}

上述代码定义了一个名为IDataPack的接口，用于抽象层封装包解决TCP黏包问题的拆包封装包的模块。该接口提供以下方法：

* GetHeadLen() uint32：获取包的头长度。用于确定消息头部的固定长度。
* Pack(msg IMessage) ([]byte, error)：封包方法，将消息进行封装成字节流的形式，用于发送。
* Unpack([]byte) (IMessage, error)：拆包方法，将接收到的字节流进行解析，还原成消息对象。

接口的实现类针对Message消息进行了TLV格式的封装和拆包操作。TLV格式指的是消息头部包含类型（Type）、长度（Length），再加上实际的消息内容（Value）。

具体的封包过程如下：

1. 根据消息对象的内容，构建消息头部，包含消息的长度和类型信息。
2. 将消息头部和消息内容按照一定的规则拼接在一起，形成完整的封包数据。

具体的拆包过程如下：

1. 先读取固定长度的头部数据，获取消息的长度和类型信息。
2. 根据消息长度信息，读取对应长度的消息内容。
3. 根据读取到的头部和内容，还原成完整的消息对象。

通过实现IDataPack接口的具体类，可以对TCP连接中的数据流进行封包和拆包操作，解决TCP黏包问题，确保消息的完整传输。

#### 4. 传递消息实体

// DataPackage 封包解包的结构体  
type DataPackage struct {  
}  
  
// NewDataPackage 创建一个封包拆包的实例  
func NewDataPackage() \*DataPackage {  
 return &DataPackage{}  
}  
  
// GetHeadLen 获取包头的长度 根据我们的协议定义直接返回8就可以了  
func (dp \*DataPackage) GetHeadLen() uint32 {  
 return uint32(8)  
}  
  
// Pack 将 ninterfance.IMessage 类型的结构封装为字节流的形式  
// 字节流形式 [ 数据长度 + ID + 真实数据 ]  
func (dp \*DataPackage) Pack(msg ninterfance.IMessage) ([]byte, error) {  
 // 创建一个字节流的缓存，将msg的信息一步一步的填充到里面去  
 dataBuff := bytes.NewBuffer([]byte{})  
 if err := binary.Write(dataBuff, binary.LittleEndian, msg.GetMsgDataLen()); err != nil {  
 return nil, err  
 }  
 if err := binary.Write(dataBuff, binary.LittleEndian, msg.GetMsgID()); err != nil {  
 return nil, err  
 }  
 if err := binary.Write(dataBuff, binary.LittleEndian, msg.GetMsgData()); err != nil {  
 return nil, err  
 }  
 return dataBuff.Bytes(), nil  
}  
  
func (dp \*DataPackage) Unpack(data []byte) (ninterfance.IMessage, error) {  
 // 创建一个从data里面读取的ioReader  
 dataBuffer := bytes.NewBuffer(data)  
 msg := &Message{}  
 if err := binary.Read(dataBuffer, binary.LittleEndian, &msg.DataLen); err != nil {  
 return nil, err  
 }  
 if err := binary.Read(dataBuffer, binary.LittleEndian, &msg.ID); err != nil {  
 return nil, err  
 }  
 return msg, nil  
}

这段代码实现了一个数据包的封包（Pack）和拆包（Unpack）的功能，用于将消息（Message）封装为字节流形式，并将字节流解析还原为消息对象。

1. DataPackage 结构体是封包和拆包的实现结构体，其中没有存储任何状态信息。
2. NewDataPackage() 函数用于创建一个 DataPackage 实例，返回指针。
3. GetHeadLen() 方法用于获取包头的长度，这里直接返回固定值 8，表示包头长度为 8 字节。
4. Pack(msg ninterfance.IMessage) ([]byte, error) 方法将传入的 IMessage 对象封装为字节流形式。具体实现如下：
   1. 创建一个字节流缓存 dataBuff。
   2. 使用 binary.Write 方法按照大端字节序将消息的数据长度、ID和真实数据依次写入 dataBuff。
   3. 最后，返回 dataBuff.Bytes()，即封装好的字节流。
5. Unpack(data []byte) (ninterfance.IMessage, error) 方法用于将字节流解析为消息对象。具体实现如下：
   1. 创建一个从 data 中读取的 dataBuffer。
   2. 创建一个空的 Message 对象 msg。
   3. 使用 binary.Read 方法按照大端字节序从 dataBuffer 读取数据长度和ID，并将其赋值给 msg 的对应字段。
   4. 最后，返回解析得到的 msg 对象。

#### 5. 协议约束

const (   
 USER\_AUTHENTICATION\_SUCCESSFULLY = 10001   
)   
const (   
 NEW\_CONNECTION = 20001   
 USER\_INFORMATION = 20002   
 KEEP\_ALIVE = 20003   
 CONNECTION\_IF\_FULL = 20004   
)   
   
const (   
 USER\_NOT\_EXIST = 30001   
 USER\_ALREADY\_EXIST = 30002   
 USER\_EXPIRED = 30003   
 PASSWORD\_INCORRET = 30004   
 AUTH\_FAIL = 30005   
)   
   
const (   
 USER\_REQUEST\_AUTH = 60001   
)

1. 在第一个常量块中，USER\_AUTHENTICATION\_SUCCESSFULLY 的值为 10001，表示用户身份验证成功。
2. 在第二个常量块中，定义了以下常量：
   * NEW\_CONNECTION：值为 20001，表示新的连接。
   * USER\_INFORMATION：值为 20002，表示用户信息。
   * KEEP\_ALIVE：值为 20003，表示保持连接。
   * CONNECTION\_IF\_FULL：值为 20004，表示连接已满。
3. 在第三个常量块中，定义了以下常量：
   * USER\_NOT\_EXIST：值为 30001，表示用户不存在。
   * USER\_ALREADY\_EXIST：值为 30002，表示用户已存在。
   * USER\_EXPIRED：值为 30003，表示用户已过期。
   * PASSWORD\_INCORRET：值为 30004，表示密码不正确。
   * AUTH\_FAIL：值为 30005，表示认证失败。
4. 在第四个常量块中，定义了以下常量：
   * USER\_REQUEST\_AUTH：值为 60001，表示用户请求认证。

在后续的过程中，将会更具系统的设计和相关业务不断进行更新操作。

var ProtocolMap map[uint32]interface{}  
  
func init() {  
 ProtocolMap = make(map[uint32]interface{})  
 // 添加创建协议  
 ProtocolMap[NEW\_CONNECTION] = "NEW\_CONNECTION"  
 ProtocolMap[USER\_INFORMATION] = ClientConnInfo{}  
 ProtocolMap[KEEP\_ALIVE] = "ping"  
 ProtocolMap[CONNECTION\_IF\_FULL] = "the-connection-is-full."  
  
 ProtocolMap[USER\_ALREADY\_EXIST] = "user already exist!"  
 ProtocolMap[USER\_NOT\_EXIST] = "user not exist."  
 ProtocolMap[USER\_EXPIRED] = "user not expired."  
 ProtocolMap[PASSWORD\_INCORRET] = "The password is incorrect."  
  
 ProtocolMap[USER\_AUTHENTICATION\_SUCCESSFULLY] = "Authentication successful!"  
}

这段代码定义了一个名为 ProtocolMap 的变量，它是一个映射（map）类型，用于存储消息协议的映射关系。

在 init() 函数中，首先使用 make() 函数创建了一个空的 ProtocolMap，然后通过键值对的方式将不同的消息协议与对应的值进行关联。

具体的映射关系如下：

* NEW\_CONNECTION 对应的值是字符串 "NEW\_CONNECTION"，表示新连接的协议。
* USER\_INFORMATION 对应的值是 ClientConnInfo{}，表示用户信息的协议，ClientConnInfo 是一个结构体类型。
* KEEP\_ALIVE 对应的值是字符串 "ping"，表示保持连接的协议。
* CONNECTION\_IF\_FULL 对应的值是字符串 "the-connection-is-full."，表示连接已满的协议。
* USER\_ALREADY\_EXIST 对应的值是字符串 "user already exist!"，表示用户已存在。
* USER\_NOT\_EXIST 对应的值是字符串 "user not exist."，表示用户不存在。
* USER\_EXPIRED 对应的值是字符串 "user not expired."，表示用户未过期。
* PASSWORD\_INCORRET 对应的值是字符串 "The password is incorrect."，表示密码不正确
* USER\_AUTHENTICATION\_SUCCESSFULLY 对应的值是字符串 "Authentication successful!"，表示通过这个 ProtocolMap，可以根据特定的协议值获取对应的消息或状态信息，方便在代码中处理不同的协议情况。

### 6.2. 通用模块配置

#### 1. 交换数据函数

这一步是整个内网穿透的核心过程，不管以后整个程序写的如何，此处都是十分重要的。这一步呢，我们要封装一个交换数据的函数，即就是，将一个tcpConn的数据转发到另一个tcpConn里面去，在实现这些内容之前，我们先看看go提供的一个函数io.Copy:

func Copy(dst Writer, src Reader) (written int64, err error) {  
 return copyBuffer(dst, src, nil)  
}

参数说明：

* dst 是目标 io.Writer 接口，用于接收拷贝的数据。
* src 是源 io.Reader 接口，用于提供要拷贝的数据。

返回值：

* written 是拷贝的字节数。
* err 是可能发生的错误。如果拷贝操作成功，该值为 nil。

io.Copy 函数会不断地从源 src 中读取数据，并将其写入目标 dst，直到源 src 的数据结束或发生错误。它会自动处理数据的缓冲和复制过程，简化了数据拷贝的操作。

看了这个函数我们是不是一下子就明白了，交换数据是如此的Easy!

TCPConn这个类型实现了Write(p []byte) (n int, err error)和Read(p []byte) (n int, err error)方法，那么这个类型也就实现了Writer和Reader接口，因此可以直接调用这个函数。（go 语言特性，我非常喜欢）。

于是就有了我们封装好的函数：

// SwapConnDataEachOther 通讯双方相互交换数据  
func SwapConnDataEachOther(local, remote \*net.TCPConn) {  
 go swapConnData(local, remote)  
 go swapConnData(remote, local)  
}  
  
// SwapConnData 这个函数是交换两个连接数据的函数  
func swapConnData(local, remote \*net.TCPConn) {  
 // 关闭本地和远程连接通道  
 defer local.Close()  
 defer remote.Close()  
 // 将remote的数据拷贝到local里面  
 \_, err := io.Copy(local, remote)  
 if err != nil {  
 return  
 }  
}

#### 2. 创建TCP监听和连接

func CreateTCPListener(addr string) (\*net.TCPListener, error) {  
 tcpAddr, err := net.ResolveTCPAddr("tcp", addr)  
 if err != nil {  
 return nil, err  
 }  
 tcpListener, err := net.ListenTCP("tcp", tcpAddr)  
 if err != nil {  
 return nil, err  
 }  
 return tcpListener, nil  
}  
  
// CreateTCPConn 连接指定的TCP  
func CreateTCPConn(addr string) (\*net.TCPConn, error) {  
 tcpAddr, err := net.ResolveTCPAddr("tcp", addr)  
 if err != nil {  
 return nil, err  
 }  
 tcpConn, err := net.DialTCP("tcp", nil, tcpAddr)  
 if err != nil {  
 return nil, err  
 }  
 return tcpConn, nil  
}

这两个函数比较简单，我就不在赘述了，CreateTCPListener是新建一个tcp的listener,CreateTCPConn是新建一个tcp的connecter，非常的简单。

#### 3. 客户端连接信息模块

此模块是客户端和服务器约定好如何传输用户的数据，如何将用户信息转化为字节流，以及如何从字节流中解封装为用户信息，也可以叫做协议吧。

// ClientConnInfo 客户端连接信息  
type ClientConnInfo struct {  
 UID int64  
 Port int32  
}  
  
// NewClientConnInstance 新建一个实体  
func NewClientConnInstance(id int64, port int32) \*ClientConnInfo {  
 return &ClientConnInfo{  
 UID: id,  
 Port: port,  
 }  
}  
  
// ToBytes 将 ClientConnInfo 结构体转换为字节流  
func (info \*ClientConnInfo) ToBytes() ([]byte, error) {  
 buf := new(bytes.Buffer)  
  
 // 使用 binary.Write 将字段逐个写入字节流  
 if err := binary.Write(buf, binary.BigEndian, info.UID); err != nil {  
 return nil, err  
 }  
 if err := binary.Write(buf, binary.BigEndian, info.Port); err != nil {  
 return nil, err  
 }  
  
 return buf.Bytes(), nil  
}  
  
// FromBytes 从字节流中恢复 ClientConnInfo 结构体  
func (info \*ClientConnInfo) FromBytes(data []byte) error {  
 buf := bytes.NewReader(data)  
 // 使用 binary.Read 从字节流中读取字段值  
 if err := binary.Read(buf, binary.BigEndian, &info.UID); err != nil {  
 return err  
 }  
 if err := binary.Read(buf, binary.BigEndian, &info.Port); err != nil {  
 return err  
 }  
 return nil  
}

单元测试：

func TestClientConnInfo\_FromBytes(t \*testing.T) {  
 ci := ClientConnInfo{  
 UID: 1,  
 Port: 8080,  
 }  
 data, err := ci.ToBytes()  
 if err != nil {  
 fmt.Println("[ToBytes Err]", err)  
 } else {  
 fmt.Println("[ToBytes Successfully]", data)  
 }  
 nci := new(ClientConnInfo)  
 err = nci.FromBytes(data)  
 if err != nil {  
 fmt.Println("[FromBytes Err]", err)  
 } else {  
 fmt.Println("[FromBytes Successfully]", nci)  
 }  
}

=== RUN TestClientConnInfo\_FromBytes  
[ToBytes Successfully] [0 0 0 0 0 0 0 1 0 0 31 144]  
[FromBytes Successfully] &{1 8080}  
--- PASS: TestClientConnInfo\_FromBytes (0.00s)

#### 4. 用户信息模块

这些内容比较类似，后续会考虑将这些封装为一个接口。

// UserInfo 用户信息模块  
type UserInfo struct {  
 // UserName  
 UserName string  
 // Password  
 Password string  
 // ExpireTime  
 ExpireTime time.Time  
}  
  
// NewUserInfoInstance 新建一个实体  
func NewUserInfoInstance(username, password string) \*UserInfo {  
 return &UserInfo{  
 UserName: username,  
 Password: password,  
 }  
}  
  
func (info \*UserInfo) ToBytes() ([]byte, error) {  
 buf := new(bytes.Buffer)  
  
 // 将用户名长度编码到字节流中  
 userNameLen := len(info.UserName)  
 if err := binary.Write(buf, binary.BigEndian, int32(userNameLen)); err != nil {  
 return nil, err  
 }  
  
 // 将用户名内容编码到字节流中  
 if err := binary.Write(buf, binary.BigEndian, []byte(info.UserName)); err != nil {  
 return nil, err  
 }  
  
 // 将密码长度编码到字节流中  
 passwordLen := len(info.Password)  
 if err := binary.Write(buf, binary.BigEndian, int32(passwordLen)); err != nil {  
 return nil, err  
 }  
  
 // 将密码内容编码到字节流中  
 if err := binary.Write(buf, binary.BigEndian, []byte(info.Password)); err != nil {  
 return nil, err  
 }  
  
 // 将过期时间编码到字节流中  
 if err := binary.Write(buf, binary.BigEndian, info.ExpireTime.Unix()); err != nil {  
 return nil, err  
 }  
  
 return buf.Bytes(), nil  
}  
  
func (info \*UserInfo) FromBytes(data []byte) error {  
 buf := bytes.NewReader(data)  
  
 // 从字节流中读取用户名长度  
 var userNameLen int32  
 if err := binary.Read(buf, binary.BigEndian, &userNameLen); err != nil {  
 return err  
 }  
  
 // 从字节流中读取用户名内容，并根据长度进行截取  
 userNameBytes := make([]byte, userNameLen)  
 if err := binary.Read(buf, binary.BigEndian, userNameBytes); err != nil {  
 return err  
 }  
 info.UserName = string(userNameBytes)  
  
 // 从字节流中读取密码长度  
 var passwordLen int32  
 if err := binary.Read(buf, binary.BigEndian, &passwordLen); err != nil {  
 return err  
 }  
  
 // 从字节流中读取密码内容，并根据长度进行截取  
 passwordBytes := make([]byte, passwordLen)  
 if err := binary.Read(buf, binary.BigEndian, passwordBytes); err != nil {  
 return err  
 }  
 info.Password = string(passwordBytes)  
  
 // 从字节流中读取过期时间  
 var expireTimeUnix int64  
 if err := binary.Read(buf, binary.BigEndian, &expireTimeUnix); err != nil {  
 return err  
 }  
 info.ExpireTime = time.Unix(expireTimeUnix, 0)  
  
 return nil  
}

这段代码定义了一个 UserInfo 结构体，表示用户信息模块。它包含了用户名、密码和过期时间等字段。

NewUserInfoInstance 函数是一个构造函数，用于创建 UserInfo 结构体的实例，并初始化用户名和密码。

ToBytes 方法将 UserInfo 结构体的字段编码为字节流。首先，创建一个 bytes.Buffer 缓冲区。然后，将用户名长度以 int32 类型的大端序编码写入缓冲区。接下来，将用户名内容以字节流形式写入缓冲区。然后，将密码长度和密码内容写入缓冲区。最后，将过期时间的 UNIX 时间戳以 int64 类型的大端序编码写入缓冲区。最终，返回缓冲区中的字节流。

FromBytes 方法将字节流解码为 UserInfo 结构体的字段。首先，创建一个 bytes.Reader 读取器，读取字节流。然后，从读取器中读取用户名长度，并根据长度创建字节切片。接着，从读取器中读取用户名内容，并将其转换为字符串赋值给 UserName 字段。然后，从读取器中读取密码长度，并根据长度创建字节切片。接着，从读取器中读取密码内容，并将其转换为字符串赋值给 Password 字段。最后，从读取器中读取过期时间的 UNIX 时间戳，并使用 time.Unix 函数将其转换为 time.Time 类型赋值给 ExpireTime 字段。

这样，通过 ToBytes 和 FromBytes 方法，可以实现 UserInfo 结构体与字节流之间的相互转换，方便在网络传输或存储中使用。

单元测试：

func TestNewUserInfoInstance(t \*testing.T) {  
 ui := NewUserInfoInstance("wyfld", "yfw123456789")  
 ui.ExpireTime = time.Now()  
 d, err := ui.ToBytes()  
 if err != nil {  
 fmt.Println(err)  
 return  
 }  
 fmt.Println(d)  
 un := new(UserInfo)  
 un.FromBytes(d)  
 fmt.Println(un)  
}

=== RUN TestNewUserInfoInstance  
[0 0 0 5 119 121 102 108 100 0 0 0 12 121 102 119 49 50 51 52 53 54 55 56 57 0 0 0 0 100 132 113 19]  
&{wyfld yfw123456789 2023-06-10 20:48:19 +0800 CST}  
--- PASS: TestNewUserInfoInstance (0.02s)  
PASS

#### 5. 发送信息模块

// SendAndReceiveInstance 实体   
type SendAndReceiveInstance struct {   
 Conn \*net.TCPConn   
}   
   
// NewSendAndReceiveInstance 新建一个控制层实例对象   
func NewSendAndReceiveInstance(conn \*net.TCPConn) \*SendAndReceiveInstance {   
 return &SendAndReceiveInstance{   
 Conn: conn,   
 }   
}   
   
// SendDataToClient 向客户端发送消息，此处应该指定协议   
func (csi \*SendAndReceiveInstance) SendDataToClient(dataType uint32, msg []byte) (int, error) {   
 msgInstance := NewMsgPackage(dataType, msg)   
 pkgInstance := NewDataPackage()   
 dataStream, err := pkgInstance.Pack(msgInstance)   
 if err != nil {   
 return 0, err   
 }   
 count, err := csi.Conn.Write(dataStream)   
 if err != nil {   
 return 0, err   
 }   
 return count, nil   
}   
func (csi \*SendAndReceiveInstance) ReadHeadDataFromClient() (ninterfance.IMessage, error) {   
 // 根据我们协议规定的内容每次数据来的时候，先读取头部的8个字节的数据   
 headData := make([]byte, 8)   
 // 从Conn中读取数据到headData中去   
 if \_, err := io.ReadFull(csi.Conn, headData); err != nil {   
 return nil, err   
 }   
 // 先创建一个解包的实例   
 dp := NewDataPackage()   
 // 解封装这个包   
 return dp.Unpack(headData)   
}   
func (csi \*SendAndReceiveInstance) ReadRealDataFromClient(msg ninterfance.IMessage) (ninterfance.IMessage, error) {   
 if msg.GetMsgDataLen() < 0 {   
 return msg, nil   
 }   
 // 新建一个data，长度为msg头部的长度   
 data := make([]byte, msg.GetMsgDataLen())   
 if \_, err := io.ReadFull(csi.Conn, data); err != nil {   
 return nil, err   
 }   
 msg.SetMsgData(data)   
 return msg, nil   
}

这段代码定义了一个名为 SendAndReceiveInstance 的结构体，表示发送和接收消息的实例对象。它包含了一个 Conn 字段，用于存储与客户端的 TCP 连接。

NewSendAndReceiveInstance 函数是一个构造函数，用于创建 SendAndReceiveInstance 结构体的实例，并初始化 Conn 字段。

SendDataToClient 方法用于向客户端发送消息。它接受一个 dataType 参数表示消息的类型，以及一个 msg 参数表示要发送的消息内容。首先，使用 NewMsgPackage 函数创建一个消息包实例，该实例包含了消息类型和消息内容。然后，使用 NewDataPackage 函数创建一个封包解包实例。接下来，将消息实例打包为字节流，并获取打包后的数据流。最后，使用 Conn 字段的 Write 方法将数据流写入连接，并返回写入的字节数和可能发生的错误。

ReadHeadDataFromClient 方法用于从客户端读取消息头数据。根据协议规定，每次数据到达时，首先需要读取8个字节的头部数据。该方法创建一个长度为8的字节切片 headData，然后使用 io.ReadFull 函数从 Conn 字段中读取数据并存储到 headData 中。接着，创建一个数据包解析实例 dp，并使用其 Unpack 方法解包 headData。最后，返回解包后的消息和可能发生的错误。

ReadRealDataFromClient 方法用于从客户端读取真实数据。根据协议规定，消息头部中记录了真实数据的长度，因此需要根据长度读取相应的数据。首先，检查消息的数据长度，如果长度小于0，则直接返回消息。否则，根据消息的数据长度创建一个字节切片 data，然后使用 io.ReadFull 函数从 Conn 字段中读取数据并存储到 data 中。最后，将读取到的数据设置为消息的数据部分，并返回消息和可能发生的错误。

这些方法组合起来，实现了从客户端发送消息和接收消息的功能，通过封包解包的过程，将消息转换为字节流进行传输，并在接收端进行解析和处理。

## 7. 性能和优化

## 8. 附录