浙江傳媒學院
COMMUNICATION
UNIVERSITY
OF ZHEJIANG

设计文档

(2025 届)

项目名称 Keep Jumpinig

学 生 姓 名: 史龙星 学 号: 210207340

二级学院名称: 媒体工程学院 专 业: 数字媒体技术

指 导 教 师: 王忠 职 称: 教学老师

郑 重 声 明

我谨在此郑重声明：本人制作项目《Keep Jumpinig》均系本人独立完成，没有抄袭行为，部分客户端代码与室友合作完成，网络模块代码框架由课程实例提供，音乐来自网络 (侵必删)。其余均为本人制作。若有不实，后果由本人承担。

承诺人（签名）：史龙星

2024 年　　6 月　　9 日

Keep Jumpinig

摘要：这是我在学期某中基于课程提供的网络通信模块示例将小游戏改造为联网游戏的一个项目。实现了 xxx。

关键词：结课作业；服务器搭建；客户端制作

目 录

1 绪论

1.1 工程文件运行要求

Unity2021.3.14f1c1 版本及以上。使用 URP 渲染管线。

> **提示：**本程序使用 Unity2021.3.14f1c1 版本制作。Unity 版本过低可能无法运行。Unity 有多个渲染管线 (Build-in, URP, HDRP)。我们使用 URP 管线制作，若使用其它渲染管线可能因为库函数的差异导致显示 Bug。

1.2 运行文件

服务端在局域网与公网上都能跑，输入本地 IP 与一个未被使用的 Port 即可点击启动服务器。客户端输入服务端的 IP(服务器部署在公网就输入服务器公网地址，服务端署在局域网就输入本地地址)，然后输入服务端端口点击开始连接即可。

1.3 问题反馈

本人尚为程序入门学习者，才疏学浅，如果有任何错误或者纰漏，望各位大佬不吝赐教。

1.4 项目文件下载

https://github.com/Target12TA/OnlineGameCourseWork

2 网络模块

2.1 通讯模式

2.1.1 客户端通讯模式

客户端所有的网络通讯均采用异步传输。传输逻辑如下图 2-1。

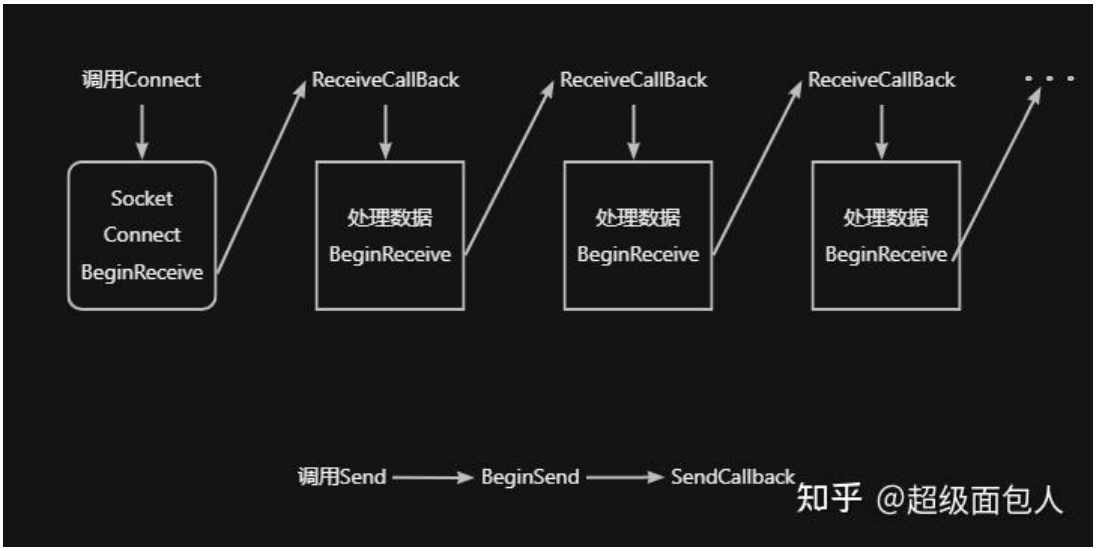

图 2-1

代码段 2-1 NetManager

```
//客户端所有的网络通讯均采用异步传输

//定义套接字
static Socket

public static void Connect(string ip, int port)
{
    ...
    socket.BeginConnect(.., ConnectCallback,..);
}

//Connect回调
private static void ConnectCallback(IAsyncResult ar)
{
    ...
    socket.EndConnect(ar);
    socket.BeginReceive(.., ReceiveCallback, ..);
}

//Receive回调
private static void ReceiveCallback(IAsyncResult ar)
{
```

```
    int buffCount = socket.EndReceive(ar);
    if (buffCount == 0)
    {
        //四次挥手
        Close();
        return;
    }
    socket.BeginReceive(.., ReceiveCallback, ..);
}
//*************************Send*********************************************
//Send同理都是由一个Callback函数进行回调
//在解决粘包半包时再进行简介
```

代码段 2-2 Main

```
void Start
{
    //与服务端进行连接 第三次握手
    //ps:
    NetManager.Connect(ip, port);
}

void Update
{
    NetManager.MsgUpdate();
}
```

2.1.2 服务端通讯模式

客户端所有的网络通讯均采用异步传输。同时检测多个 Socket 的状态，在设置要监听的列表后，如果有一个或多个 Socket 可读或可写，那就返回这些 Socket，如果没有可读的那就继续堵塞。

代码段 2-3 Main

```
private void Update()
{
    if (recvStr != "")
    {
        GameObject newItem = Instantiate(item, item.GetComponent<Transform>().parent);
        newItem.GetComponent<Text>().text = recvStr;
        newItem.SetActive(true);
        recvStr = "";
    }

    if (socket == null)
    {
        return;
    }
```

```
    checkRead.Clear();
    checkRead.Add(socket);
    Socket.Select(checkRead, null, null, 0);
    foreach (Socket s in checkRead)
    {
        byte[] readBuffer = new byte[1024];
        int count = socket.Receive(readBuffer);
        string receiveMessage = System.Text.Encoding.UTF8.GetString(readBuffer, 0, count);
        GameObject newItem = Instantiate(item, item.GetComponent<Transform>().parent);
        newItem.GetComponent<Text>().text = receiveMessage;
        newItem.SetActive(true);
    }
}
```

2.2 正确收发消息

2.2.1 解决粘包半包问题

粘包半包现象：本质就是收发数据不同步的问题。如下图 2-2、2-3。

图 2-2

图 2-3

使用长度信息法解决粘包半包：在消息头每次都加上消息的长度。

代码段 2-4 ByteArray

```
//模拟成缓冲区，包装成类便于拓展

//缓冲区
public byte[] bytes

//读写位置
public int readIdx = 0;
public int writeIdx = 0;
//容量
private int capacity = 0;
//剩余空间
public int remain { get { return capacity - writeIdx; } }
//数据长度
public int length { get { return writeIdx - readIdx; } }
//移动数据
public void MoveBytes()
{
    Array.Copy(bytes, //原数组
        readIdx,    //原数组的起始位置
        bytes,      //目标数组
        0,        //目标数组的起始位置
        length);    //复制的消息长度
    writeIdx = length;
    readIdx = 0;
}
```

代码段 2-5 NetManager

```
//静态 模拟的读取缓冲区本就只有一个
static ByteArray receiveBuff;
private static void ConnectCallback(IAsyncResult ar)
{
    receiveBuff.writeIdx += socket.EndReceive(ar);
    //缓冲区大于协议字节数
    while(receiveBuff.length > 4)
    {
        //初始化协议，在其中读取长度信息
        int msgTotalLen = 0;
        //对协议(数据)进行解码
        MsgBase msgBase = MsgBase.FromBytes(.....,out msgTotalLen);
        if(长度 > 缓冲区长度 && 一些不符合协议解码的条件) break;
        //添加协议
        msgList.Add(msgBase);
        //更新缓冲区读写位置
        receiveBuff.readIdx += msgTotalLen;
    }
    //移动数据
    if (receiveBuff.readIdx > 0) receiveBuff.MoveBytes();

    socket.BeginReceive(receiveBuff.bytes,   //缓冲区
```

```
        receiveBuff.writeIdx,               //开始位置
        receiveBuff.remain,          //最多读取多少数据
        0,                   //标志位，设成0即可
        ReceiveCallback,           //回调函数
        socket);               //状态
}
```

2.2.2 解决大小端问题

大端小端现象：字节存储/读取在地址上的顺序不同。大小端问题其实可以归类到编解码主题中。

代码段 2-6 MsgType

```
//处理一条消息的逻辑：解析协议头判断是否可以创建消息对象然后进行下一步的处理。
//协议头与协议体分开编解码：
//协议头占四个字节(长度 协议类型)由我们自己处理  协议体通过序列化工具处理

//编码器
public static byte[] ToBytes(MsgBase msgBase)
{
    //编码协议体
    string s = JsonUtility.ToJson(msgBase);
    byte[] strBytes = System.Text.Encoding.UTF8.GetBytes(s);
    msgBase.msgLen = (Int16)strBytes.Length;

    //编码协议头
    byte[] buf = new byte[strBytes.Length + 4];
    WriteInt16(buf, 0, 2, msgBase.msgLen);
    WriteInt16(buf, 2, 2, (Int16)msgBase.msgType);

    //拼接后返回进行传输
    strBytes.CopyTo(buf, 4);
    return buf;
}
//实现对Int16类型字段的编码
private static void WriteInt16(byte[] buf, int offset, int count, Int16 val)
{
    if (BitConverter.IsLittleEndian)
    {
        buf[offset] = (byte)(val & 0xff);
        buf[offset + 1] = (byte)(val >> 8);
    }
    else
    {
        buf[offset] = (byte)(val >> 8);
        buf[offset + 1] = (byte)(val & 0xff);
    }
}
```

```
36
37  //解码器
38  public static MsgBase FromBytes(byte[] bytes, int offset, int count, out int msgTotalLen)
39  {
40      msgTotalLen = 0;
41      if (count < 4) return null;
42
43      Int16 msgLen = ReadInt16(bytes, offset, count);
44      offset += 2;
45      count -= 2;
46      if (msgLen > count+2) return null;
47
48      MsgType type = (MsgType)ReadInt16(bytes, offset, count);
49      offset += 2;
50      count -= 2;
51      string s = System.Text.Encoding.UTF8.GetString(bytes, offset, msgLen);
52      msgTotalLen = msgLen + 4;
53      MsgBase msgBase = (MsgBase)JsonUtility.FromJson(s, Type.GetType(type.ToString()));
54      msgBase.msgLen = msgLen;
55      return msgBase;
56  }
57  //实现对Int16类型字段的解码
58  public static Int16 ReadInt16(byte[] bytes, int offset, int count)
59  {
60      //必须大于2字节
61      if (count < 2)
62      {
63          throw new Exception("Need more data");
64      }
65      Int16 val;
66      if (BitConverter.IsLittleEndian)
67      {
68          val = (Int16)((bytes[offset + 1] << 8) | bytes[offset]);
69      }
70      else
71      {
72          val = (Int16)((bytes[offset] << 8) | bytes[offset + 1]);
73      }
74
75      return val;
76  }
```

2.2.3 解决发送不完整数据问题

发送不完整数据现象：在网络波动的环境下、操作系统的发送缓冲区剩余大小小于 Send 方法写入的字节大小等情况可能导致数据丢失、发送了不完整的数据。解决方式也很简单：Send 方法把要发送的数据存入了操作系统的发送缓冲区，然后返回成功写入的字节数。所以我们需要在发送前将数据保存起来，如果发送不完整，在 Send 回调

函数中继续发送数据。

代码段 2-7 ByteArray

```
//缓冲区
public byte[] bytes;
//数据长度
public int length { get { return writeIdx - readIdx; } }
//读写位置
public int readIdx = 0;
public int writeIdx = 0;
```

代码段 2-8 NetManager

```
//设计写入队列 :
//调用BeginSend之后，可能需要隔一段时间才会调用回调函数，在这一段时间再次调用BeginSend。
static Queue<ByteArray> writeQueue;

public static void Send(byte[] sendBytes)
{
    ByteArray ba = new ByteArray(sendBytes);
    writeQueue.Enqueue(ba);
    if (writeQueue.Count == 1)
        socket.BeginSend(sendBytes, 0, sendBytes.Length, 0,
            SendCallback, socket);
}


private static void SendCallback(IAsyncResult ar)
{
    ByteArray ba = writeQueue.First();
    //实际发送的字节数
    ba.readIdx += socket.EndSend(ar);
    //剩余长度等于0  即发送完整
    if (ba.length == 0)
    {
        writeQueue.Dequeue();
        ba = (writeQueue.Count <= 0) ? null :
            writeQueue.First();
    }
    //如果发送不完整，在Send回调函数中继续发送数据
    if (ba != null)
    {
        socket.BeginSend(ba.bytes, ba.readIdx, ba.length, 0,
            SendCallback, socket);
    }
}
```

2.2.4 线程冲突问题

线程冲突：客户端采用异步通讯模式，多个线程可能同时访问并操作同一个对象(我们自定义的缓冲区)，最终导致收发数据异常。连续两次发送消息时，在第二次发送时，第一次发送的回调函数刚好被调用。t2 时刻理应该 writeQueue.count == 2 不会进入 if(writeQueue.count == 1) 分支去发送数据。但是在 t3 时刻执行了 Dequeue 导致在 t4 时刻进入 if(writeQueue.count == 1) 分支去发送数据。最总导致在第二次 Send 中的数据会被发送两次。如下图 2-4。可以通过加锁 (lock) 的方式进行处理，当多个线程争夺同一

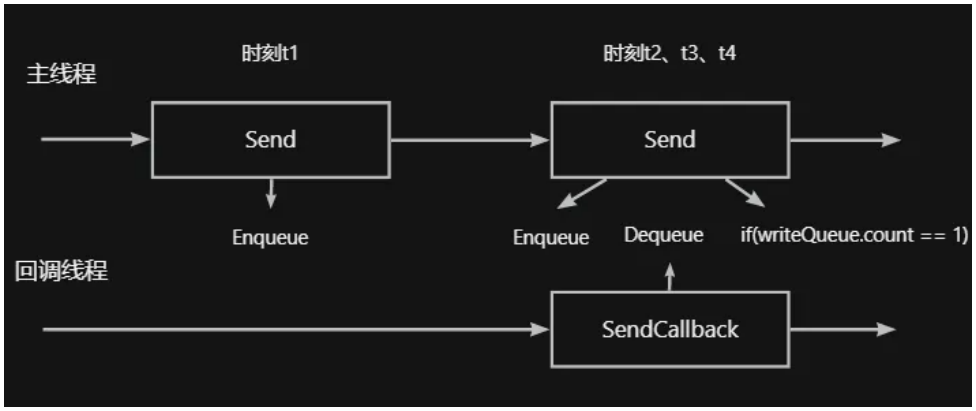

图 2-4

个锁 (同时访问同一个对象时)，最先访问的线程进行操作，其它线程进行等待。由于这是一个产生程序阻塞的地方我们需要做到性能的极致。

代码段 2-9 NetManager

```
//对writeQueue对象进行加锁处理
//当有线程访问并操作writeQueue对象时，防止其它线程去操作writeQueue对象
public static void Send(byte[] sendBytes)
{
    ByteArray ba = new ByteArray(sendBytes);
    lock (writeQueue)
    {
        writeQueue.Enqueue(ba);
        if (writeQueue.Count == 1)
        {
            socket.BeginSend(sendBytes, 0, sendBytes.Length, 0, SendCallback, socket);
        }
    }
}

private static void SendCallback(IAsyncResult ar)
{
    ....
    lock (writeQueue)
    {
        ba = writeQueue.First();
```

```
    }

    ba.readIdx += count;
    if (ba.length == 0)
    {
        lock (writeQueue)
        {
            writeQueue.Dequeue();
            ba = (writeQueue.Count <= 0) ? null : writeQueue.First();
        }
    }
    ....
}
```

2.2.5 心跳机制

心跳机制的本质就是每隔一段时间让客户端与服务端通信一次，若指定时间内又没发生通信则认为客户端已经断开连接。服务端认为此客户端网络不通，关闭此 socket。流程如图 2-5。

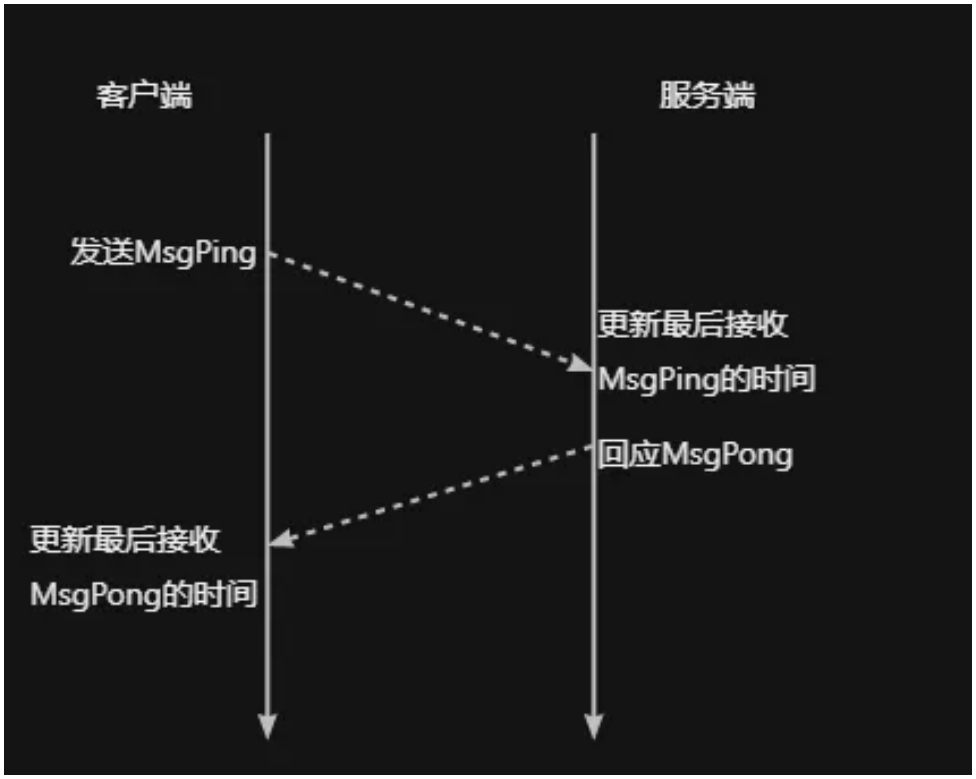

图 2-5

3 业务逻辑与设计

3.1 基于同步理论的移动消息

之前使用的是状态同步中的直接状态同步，经过思考使用跟随算法进行解决即可异步角色瞬移的问题：客户方每过 0.1s 发送玩家位置信息，然后服务端同步到所有客户端上，最后异步角色在在 0.1s 将位置和旋转信息插值计算至同步信息。

代码段 3-1 Message

```
public class MsgMove : MsgBase
{
    public string desc = "";
    public float x;
    public float y;
    public float z;
    public float eulY;

    public MsgMove() { msgType = MsgType.MsgEnter; }
    public MsgMove(string desc, float x, float y, float z, float elulY)
    {
        msgType = MsgType.MsgMove;
        this.desc = desc;
        this.x = x;
        this.y = y;
        this.z = z;
        this.eulY = elulY;
    }
}
```

代码段 3-2 Main

```
void OnMove(MsgBase msgBase)
{
  MsgMove msg = (MsgMove)msgBase;
  if (msg.desc == NetManager.GetDesc())
    return;
  BasePlayer ctrlPlayer = syncPlayers[msg.desc];
  Vector3 pos = new Vector3(msg.x, msg.y, msg.z);
  Vector3 eul = new Vector3(-90, 0, msg.eulY);
  //这里也可以通过记录下移动点，然后再Update中插值移动 应该会更平滑
  Debug.Log(msg.eulY);
  ctrlPlayer.transform.DOMove(pos, 0.03f);
  ctrlPlayer.transform.DORotate(eul, 0.03f);
}
```

3.2 Player 相关协议设计

此项目使用使用相对简单的状态同步，每当 Player 状态发生改变时就发送相应消息。主要涉及三个协议：

MsgJump —> 同步状态给其它客户端进入 Jump 状态

MsgHurt —> 同步状态给其它客户端

MsgSprint —> 同步给其它客户端播放 Sprint 特效

MsgDead —> 同步给其它客户端进入观战模式

为了用户体验，所有的负面消息 (对玩家产生挫败感消息) 都在客户端进行判断后在进行消息广播。如下：

SyncPlayer 的受伤状态基于其它客户端的 MsgHurt 消息，SyncPlayer 不与 Item 产生碰撞伤害。

SyncPlayer 的死亡状态基于其它客户端的 MsgDead 消息，删除此 SyncPlayer。CtrlPlayer.hp <= 0 删除 CtrlPlayer 将视角移至随机一个存活的 SyncPlayer 上，当没有 CtrlPlayer 与 SyncPlayer 时，结算游戏统计存活时间。

代码段 3-3 Main

```
public class MsgJump : MsgBase
{
    public string desc = "";
    public MsgJump() { msgType = MsgType.MsgJump; }

    public MsgJump(string desc)
    {
        msgType = MsgType.MsgJump;
        this.desc = desc;
    }
}

public class MsgHurt : MsgBase
{
    public string desc = "";
    public MsgHurt() { msgType = MsgType.MsgSprint; }

    public MsgHurt(string desc)
    {
        msgType = MsgType.MsgHurt;
        this.desc = desc;
    }
}

```

```
public class MsgDead : MsgBase
{
    public string desc = "";
    public MsgDead() { msgType = MsgType.MsgDead; }

    public MsgDead(string desc)
    {
        msgType = MsgType.MsgDead;
        this.desc = desc;
    }
}
```

代码段 3-4 Main

```
void OnSprint(MsgBase msgBase)
{
  MsgSprint msg = (MsgSprint)msgBase;
  if (msg.desc == NetManager.GetDesc())
    return;
  (syncPlayers[msg.desc] as SyncPlayer).Sprint();
}
void OnJump(MsgBase msgBase)
{
  Debug.Log($"{NetManager.GetDesc()}: 发送Jump消息");
  MsgJump msg = (MsgJump)msgBase;
  if (msg.desc == NetManager.GetDesc())
    return;
  (syncPlayers[msg.desc] as SyncPlayer).Jump();
}
void OnHurt(MsgBase msgBase)
{
  MsgHurt msg = (MsgHurt)msgBase;
  if (msg.desc == NetManager.GetDesc())
    return;
  (syncPlayers[msg.desc] as SyncPlayer).Hurt();
}
void OnDead(MsgBase msgBase)
{
  MsgDead msg = (MsgDead)msgBase;
  if (msg.desc == NetManager.GetDesc())
    return;
  (syncPlayers[msg.desc] as SyncPlayer).Dead();
  syncPlayers.Remove(msg.desc);
  LevelManager.Instance.CheckPlayerDead();
}
```

3.3 开始 UI 界面以及协议设计

代码段 3-5 服务端

```
//收到MsgJoin消息
if(!ItemManager.isActiveGenerate)
    send(MsgJoin);
else
    send(MsgRefuse);
```

代码段 3-6 客户端

```
//收到MsgRefuse消息
close()
提示：服务器已满请30s后再尝试
将开始连接失活 30s后激活

//收到MsgJoin消息
SceneMgr.Load(FightScene);

//发送MsgJoin消息
if(点击开始连接)
    connect
    send(MsgJoin)
```

3.4 Item 与 Tree 相关协议设计

由服务端实现生成/激活 Item 逻辑，客户端实现失活 Item 逻辑。协议设计：

MsgGenerateItem：id pos type

MsgRemoveItem：id

代码段 3-7 服务端

```
stack<int> useableID;
int itemNum++;
//收到MsgRemoveItem消息
useableID.Push(id);
Send(MsgRemoveItem: id);

//发送MsgGenerateItem消息
while(一段时间)
{
    if(useableID.count!=0)
        id = useableID.Pop();
    else
        id = itemNum++;
    Send(MsgGenerateItem: id、GetPos()、GetType())
}
```

代码段 3-8 客户端

```
Dictionary<int, Item> itemDic
//收到MsgGenerateItem消息
if(itemDic.ContainKey(id))
{
    item[id].setActive = true;
    基于消息给pos type 赋值...
}

//收到MsgRemoveItem消息
item[id].setActive = false;

//发送MsgRemoveItem消息
if(item.hp<=0)
{
    Send(MsgRemoveItem: id);
    item[id].setActive = false;
}
```

通过唯一 ID 字段的设计，保证了服务器与客户端 Item 的唯一性索引。利用集合管理 Item 对象进行精确的控制达到通过重复利用已经创建的对象，避免频繁的创建和销毁过程，从而减少系统的内存分配和垃圾回收带来的开销。

服务端就像是一个存储垃圾的罐子，需要时取，对顺序没要求采用 Stack 集合。客户端失活的对象时要保证与其它客户端执行失活的对象一致，必须有一个查询操作使用 Dictionary 集合

一段时间如何控制：通过获取世界时间、开始游戏时的时间、游戏经历的时间、每一次循环经历的时间后，按需实现很简单。

随机 Pos 的生成：随机数相关的内容。在实现这里的时候，深深感受到：相当一部分网络游戏都需要在服务器端进行物理逻辑判断 (不能使用 Unity 引擎里的东西了，更获取不到当时场景里没被记录的对象)，想要高效、稳定、精确的计算势必要在服务器端实现一套自己的算法（物理、地形等），在服务器端进行的计算还可以有效防止修改本地数据的外挂。

3.5 云服务器的搭建

将服务端程序打包至云服务器上运行即可。

云服务器的优点：

1、有静态 IP、不会被运营商给动态修改。

2、24 小时不断电、不断网。

服务器一定要部署在静态 IP 下不然可能会导致数据收发异常。可以通过以下流程

解决客户端 IP 被动态修改的问题：客户端 Connect –> 服务端 Accept 并记录客户端的公网 IP 与 Port 不再改变直到客户端发送 GameOver/Leave 消息并给客户端发送其连接时的 IP 与 Port 不在改变–> 客户端收到将其记录作为发消息的唯一标识。

这时为了解决在计网中网络传输会有以下流程如图 3-1。

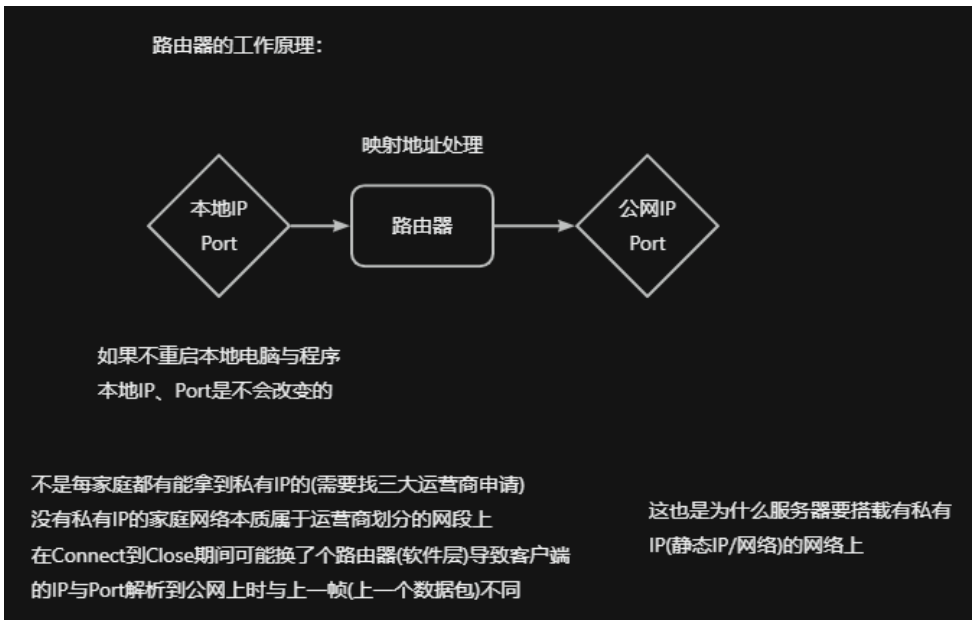

图 3-1

致谢

在本研究中，我们得到了许多人的帮助和支持，在此向他们表示感谢。首先，我要感谢我的教师，他在整个研究过程中给予了我耐心的指导和宝贵的建议。同时，我还要感谢 Google 搜索引擎以及在网络上分享自己经验的友人，他们在我学习的过程中给予了我很多帮助。最后，我要感谢我的家人和朋友，他们在我学习和研究的过程中一直支持和鼓励我。