

.NET网络通信模块与Unity网络开发

System.Net命名空间

- System.Net 是 .NET 框架中的一个命名空间，提供了用于网络编程的类和方法。它简化了许多常见网络协议的编程接口，使开发者能够轻松地创建网络应用程序

IPAddress

- 表示IP地址，有3种初始化方式

- 使用byte数组初始化IPAddress，IP地址由4个byte组成

```
1 byte[] ipAddress = new byte[] {118, 102, 11, 1};  
2 IPAddress ip1 = new IPAddress(ipAddress);
```

- 使用16进制长整型初始化IPAddress

```
1 IPAddress ip2 = new IPAddress(0x76666F0B);
```

- 使用IPAddress中的静态方法Parse做字符串转换

```
1 // public static IPAddress Parse(string ipString);  
2 IPAddress ip3 = IPAddress.Parse("118.102.111.11");
```

IPEndPoint

- 表示IP地址和端口号的组合，有2种初始化方式：

- 用长整型表示IP地址，整型表示端口号

```
1 // public IPEndPoint(long address, int port);  
2 IPEndPoint iPEndPoint1 = new IPEndPoint(0x76666F0B, 8080);
```

- 用IPAddress对象表示IP地址，整型表示端口号

```
1 // public IPEndPoint(IPAddress address, int port);
2 IPEndPoint iPEndPoint2 = new IPEndPoint(IPAddress.Parse("118.102.111.11"),
    8080);
```

IPHostEntry

- 返回域名解析的结果，包含IP地址、主机名等信息；该类一般不声明，而是作为某些方法的返回值
 - IPHostEntry类

```
1 public class IPHostEntry
2 {
3     public IPHostEntry();
4
5     public IPAddress[] AddressList { get; set; }
6     public string[] Aliases { get; set; }
7     public string HostName { get; set; }
8 }
```

Dns

- .NET中的静态类，提供一组方法用于与域名系统 (DNS) 服务器进行交互
 - 获取本地主机的名称

```
1 Debug.Log(Dns.GetHostName());
```

- 同步获取指定域名的IP信息，会阻塞主线程，有2个重载

```
1 // public static IPHostEntry GetHostEntry(IPAddress address);
2 // public static IPHostEntry GetHostEntry(string hostNameOrAddress);
3 IPHostEntry entry = Dns.GetHostEntry("www.baidu.com");
4 for(int i = 0; i < entry.AddressList.Length; i++) {
5     Debug.Log("IP: " + entry.AddressList[i]);
6 }
7 for(int i = 0; i < entry.Aliases.Length; i++) {
8     Debug.Log("Alias: " + entry.Aliases[i]);
9 }
10
11 Debug.Log("DNS服务器名称: " + entry.HostName);
```

- 异步获取指定域名的IP信息，在主线程中开启异步方法，有2个重载

```
1 // public static Task<IPHostEntry> GetHostEntryAsync(IPAddress address);
2 // public static Task<IPHostEntry> GetHostEntryAsync(string
  hostnameOrAddress);
3
4 // 异步方法
5 private async void GetHostEntry()
6 {
7     Task<IPHostEntry> task = Dns.GetHostEntryAsync("www.baidu.com");
8     await task;
9     for(int i = 0; i < task.Result.AddressList.Length; i++) {
10         Debug.Log("IP: " + task.Result.AddressList[i]);
11     }
12
13     for(int i = 0; i < task.Result.Aliases.Length; i++) {
14         Debug.Log("Alias: " + task.Result.Aliases[i]);
15     }
16
17     Debug.Log("DNS服务器名称: " + task.Result.HostName);
18 }
```

网络传输中的数据

- 序列化：把想要传递的类对象信息序列化为二进制数据传输给远端设备
- 反序列化：远端设备获取到二进制数据后反序列化成类对象
- BitConverter类，命名空间：System；作用是除字符串的其他常用类型和字节数组的相互转换
 - GetBytes方法，有多个重载，可以传入整型、浮点型、字符、布尔类型

```
1 byte[] bytes1 = BitConverter.GetBytes(1);
```

- Encoding类，命名空间：System.Text；作用是将字符串类型和字节数组相互转换，一般使用UTF-8编码类型
 - GetBytes方法，有多个重载，可以传入string或char[]

```
1 byte[] bytes2 = Encoding.UTF8.GetBytes("Test");
```

序列化

- 类对象序列化成字节数组的过程

- 在类中封装GetBytes方法，得到该类对象转换成的二进制数据
 - 明确字节数组的容量
 - 声明一个装载字节数组的容器
 - 将对象中的所有属性转换为字节数组并放入容器

```
1 public class PlayerInfo
2 {
3     public int level;
4     public string name;
5     public bool sex;
6
7     public PlayerInfo(int level, string name, bool sex)
8     {
9         this.level = level;
10        this.name = name;
11        this.sex = sex;
12    }
13
14    // 将类对象转换为二进制；字符串在存储时需要先存字符串的长度再存字符串本身，方便反序列化
15    public byte[] GetBytes()
16    {
17        // 1.明确字节数组的容量
18        int indexNum = sizeof(int) + // level: int的长度
19                      sizeof(int) + Encoding.UTF8.GetBytes(name).Length +
20                      // 记录name长度的整数: int的长度 + name: string本身的长度
21                      sizeof(bool); // sex: bool的长度
22        // 2.声明一个装载字节数组的容器
23        byte[] playerBytes = new byte[indexNum];
24
25        // 3.将对象中的所有属性转换为字节数组并放入容器
26        int index = 0;
27        BitConverter.GetBytes(level).CopyTo(playerBytes, index);
28        index += sizeof(int);
29
30        byte[] strBytes = Encoding.UTF8.GetBytes(name);
31        int num = strBytes.Length; // 字符串转换成字节数组的长度
32        BitConverter.GetBytes(num).CopyTo(playerBytes, index);
33        index += sizeof(int);
34        strBytes.CopyTo(playerBytes, index);
35        index += num;
36    }
```

```

37     BitConverter.GetBytes(sex).CopyTo(playerBytes, index);
38     index += sizeof(bool);
39
40     return playerBytes;
41 }
42 }

```

反序列化

- 将字节数组转换为原始的数据结构

- 转换成非字符串类型：使用BitConverter中的ToInt32、ToDouble、ToFloat、ToChar、ToBoolean等方法

```

1 byte[] bytes1 = BitConverter.GetBytes(10);
2 int result1 = BitConverter.ToInt32(bytes1, 0);
3 Debug.Log("转换结果：" + result1);

```

- 转换成字符串类型：使用Encoding中的GetString方法，有多个重载

```

1 byte[] bytes2 = Encoding.UTF8.GetBytes("Hello");
2 string result2 = Encoding.UTF8.GetString(bytes2);
3 string result3 = Encoding.UTF8.GetString(bytes2, 0, bytes2.Length);
4 Debug.Log("转换结果：" + result2);
5 Debug.Log("转换结果：" + result3);

```

- 将字节数组转换为转换成类对象

- 获取到字节数组playerBytes

```

1 PlayerInfo playerInfo = new PlayerInfo(10, "Alice", true);
2 byte[] playerBytes = playerInfo.GetBytes();

```

- 将字节数组按照类的成员变量序列化时的顺序反序列化，得到各成员变量的值

```

1 // PlayerInfo类的三个成员变量是level: int、name: string、sex: bool，需要按顺序反序列化
2 int index = 0;
3
4 int level = BitConverter.ToInt32(playerBytes, index);

```

```

5 index += sizeof(int);
6
7 int nameLength = BitConverter.ToInt32(playerBytes, index);
8 index += sizeof(int);
9
10 string name = Encoding.UTF8.GetString(playerBytes, index, nameLength);
11 index += nameLength;
12
13 bool sex = BitConverter.ToBoolean(playerBytes, index);
14 index += sizeof(bool);

```

- 重建类对象

```

1 PlayerInfo playerInfo2 = new PlayerInfo(level, name, sex);
2 Debug.Log("level: " + playerInfo2.level + "\n"
3     + "name: " + playerInfo2.name + "\n"
4     + "sex: " + playerInfo2.sex + "\n");

```

可序列化和反序列化类的基类

- 可以用一个抽象类作为可序列化类的基类，需包含以下方法
 - 获取字节数组容器的长度
 - 把成员变量序列化存入字节数组并返回
 - 读取字节数组反序列化到成员变量中
 - 各常用数据类型和该类对象的序列化和反序列化的方法

```

1 // 可序列化类的基类
2 public abstract class BaseData
3 {
4     /// <summary>
5     /// 获取字节数组容器的长度
6     /// </summary>
7     /// <returns>字节数组的长度</returns>
8     public abstract int GetBytesLength();
9
10    /// <summary>
11    /// 把成员变量序列化存入字节数组并返回
12    /// </summary>
13    /// <returns>成员变量序列化后的字节数组</returns>
14    public abstract byte[] ConverToByteArray();
15

```

```
16    /// <summary>
17    /// 从指定位置开始读取字节数组，反序列化到成员变量中
18    /// </summary>
19    /// <param name="bytes">字节数组</param>
20    /// <param name="beginIndex">开始读取的位置</param>
21    /// <returns>该类对象对应的字节数组的长度</returns>
22    public abstract int ReadFromByteArray(byte[] bytes, int beginIndex);
23
24    #region 将各类型的数据转成字节数组存入到指定容器中
25    protected void WriteInt(byte[] bytes, int value, ref int index)
26    {
27        BitConverter.GetBytes(value).CopyTo(bytes, index);
28        index += sizeof(int);
29    }
30
31    protected void WriteFloat(byte[] bytes, float value, ref int index)
32    {
33        BitConverter.GetBytes(value).CopyTo(bytes, index);
34        index += sizeof(float);
35    }
36
37    protected void WriteBool(byte[] bytes, bool value, ref int index)
38    {
39        BitConverter.GetBytes(value).CopyTo(bytes, index);
40        index += sizeof(bool);
41    }
42
43    protected void WritingString(byte[] bytes, string value, ref int index)
44    {
45        byte[] strBytes = Encoding.UTF8.GetBytes(value);
46        WriteInt(bytes, strBytes.Length, ref index);
47        strBytes.CopyTo(bytes, index);
48        index += strBytes.Length;
49    }
50
51    protected void WriteObject(byte[] bytes, BaseData data, ref int index)
52    {
53        data.ConvertToByteArray().CopyTo(bytes, index);
54        index += data.GetBytesLength();
55    }
56    #endregion
57
58    #region 将各类型数据对应的字节数组转换成原始数据结构或类对象
59    protected int ReadInt(byte[] bytes, ref int index)
60    {
61        int value = BitConverter.ToInt32(bytes, index);
62        index += sizeof(int);
63    }
```

```

63         return value;
64     }
65
66     protected float ReadFloat(byte[] bytes, ref int index)
67     {
68         float value = BitConverter.ToSingle(bytes, index);
69         index += sizeof(float);
70         return value;
71     }
72
73     protected bool ReadBool(byte[] bytes, ref int index)
74     {
75         bool value = BitConverter.ToBoolean(bytes, index);
76         index += sizeof(float);
77         return value;
78     }
79
80     protected string ReadString(byte[] bytes, ref int index)
81     {
82         int length = ReadInt(bytes, ref index); // 得到字符串对应的字节数组的长
            度
83         string value = Encoding.UTF8.GetString(bytes, index, length);
84         index += length;
85         return value;
86     }
87
88     protected T ReadObject<T>(byte[] bytes, ref int index) where T :
        BaseData, new()
89     {
90         T value = new T();
91         index += value.ReadFromByteArray(bytes, index);
92         return value;
93     }
94     #endregion
95 }

```

• 具体实现案例

- 类对象可以作为另一个类的成员变量，如TestInfo类中的player，序列化时直接调用该类对象的WriteObject方法（前提是作为成员变量的类也需要集成自BaseData）

```

1 public class TestInfo : BaseData
2 {
3     public int levelId;
4     public Player player;

```



```

5     public string levelName;
6
7     public override int GetBytesLength()
8     {
9         return sizeof(int) +
10             player.GetBytesLength() +
11             sizeof(int) + Encoding.UTF8.GetBytes(levelName).Length;
12     }
13
14     public override byte[] ConvertToArray()
15     {
16         int index = 0;
17         byte[] bytes = new byte[GetBytesLength()];
18         WriteInt(bytes, levelId, ref index);
19         WriteObject(bytes, player, ref index);
20         WritingString(bytes, levelName, ref index);
21
22         return bytes;
23     }
24 }
25
26 public class Player: BaseData
27 {
28     public bool sex;
29
30     public override int GetBytesLength()
31     {
32         return sizeof(int);
33     }
34
35     public override byte[] ConvertToArray()
36     {
37         int index = 0;
38         byte[] bytes = new byte[GetBytesLength()];
39         WriteBool(bytes, sex, ref index);
40
41         return bytes;
42     }
43 }

```

- 序列化和反序列化测试

- info1: 需要序列化的类对象
- bytes: info1序列化后的字节数组
- info2: 从bytes反序列化的类对象，用于重建info1

```

1 // 声明一个TestInfo的类对象info1, 并赋值成员变量
2 TestInfo info1 = new TestInfo();
3 info1.levelId = 100;
4 info1.player = new Player();
5 info1.player.sex = true;
6 info1.levelName = "EarthOnline";
7
8 // 序列化info1, 存入字节数组
9 byte[] bytes = info1.ConvertToByteArray();
10
11 // 声明一个TestInfo的类对象info2, 用于反序列化字节数组并重建info1
12 TestInfo info2 = new TestInfo();
13 info2.ReadFromByteArray(bytes, 0);
14
15 Debug.Log("levelId: " + info2.levelId + "\n" +
16         "playerSex: " + info2.player.sex + "\n" +
17         "levelName: " + info2.levelName + "\n");

```

Socket

- Socket是C#提供的用于网络通信的类（在其他语言中也有Socket类），位于System.Net.Sockets命名空间
- Socket套接字是支持TCP/IP网络通信的基本操作单位，包含
 - 本机的IP地址和端口
 - 对方主机的IP地址和端口
 - 使用的协议
- 一个Socket对象可被视为一个数据通道，连接客户端和服务端，用来收发消息
- C#中通过构造函数声明Socket的类型，构造函数有多个重载

```

1 public Socket(SocketInformation socketInformation);
2 public Socket(SocketType socketType, ProtocolType protocolType);
3 // 最常用的是网络寻址、Socket类型和协议类型的构造方式
4 public Socket(AddressFamily addressFamily,
5         SocketType socketType, ProtocolType protocolType);

```

Socket的分类

- 流套接字：用于实现TCP通信，提供面向连接的、可靠的、有序的、不丢失无重复的数据传输服务

```

1 Socket tcpSocket= new Socket(AddressFamily.InterNetwork,

```

```
2 SocketType.Stream, ProtocolType.Tcp);
```

- 数据报套接字：用于实现UDP通信，提供无连接的、不可靠的、不保证顺序的、可能出现丢失和重复的数据传输服务

```
1 Socket udpSocket = new Socket(AddressFamily.InterNetwork,  
2 SocketType.Dgram, ProtocolType.Udp);
```

- 原始套接字：用于实现IP数据报通信，直接访问协议的较低层，用于监听和分析数据报

TcpSocket常用方法

- 服务端专用方法

```
1 // 绑定IP和端口  
2 IPEndPoint ipEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8080);  
3 tcpSocket.Bind(ipEndPoint);  
4  
5 // 设置客户端连接的最大数量  
6 tcpSocket.Listen(999);  
7  
8 // 等待客户端接入  
9 Socket connectSocket = tcpSocket.Accept();
```

- 客户端专用方法

```
1 // 连接远端服务器  
2 tcpSocket.Connect(IPAddress.Parse("118.12.123.11"), 8080);
```

- 双端通用方法

```
1 // 发送数据，有同步和异步的方法  
2 byte[] byteArray1 = new byte[4] { 0x01, 0x02, 0x03, 0x04 };  
3 tcpSocket.Send(byteArray1);  
4  
5 // 接收数据，有同步和异步的方法  
6 byte[] byteArray2 = new byte[4];  
7 int length = tcpSocket.Receive(byteArray2);  
8
```

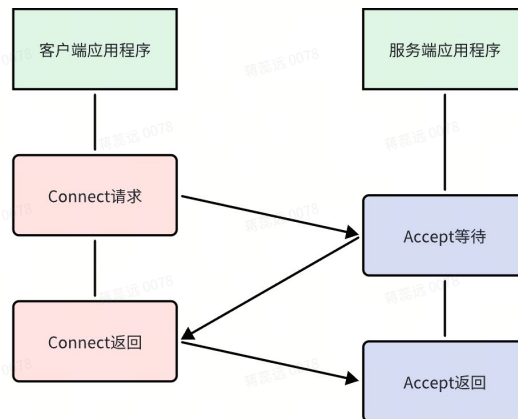
```

9 // 释放连接并关闭Socket, 需要在Close之前调用
10 tcpSocket.Shutdown(SocketShutdown.Both);
11
12 // 关闭连接, 释放所有的Socket资源
13 tcpSocket.Close();

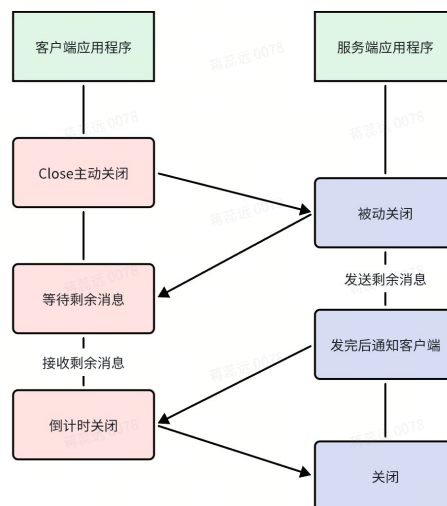
```

- TCP协议的三次握手和四次挥手封装在Socket内部, 无需额外处理

- TCP协议三次握手在Socket中的体现



- TCP协议四次挥手在Socket中的体现



- 简单的网络通信步骤

- 服务端

```

1 // 创建Tcp Socket
2 Socket tcpSocket = new Socket(AddressFamily.InterNetwork,
3     SocketType.Stream, ProtocolType.Tcp);

```

```

4
5 // 将Socket与本地IP地址和端口号绑定
6 try
7 {
8     IPEndPoint iPEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"),
9         8080);
10    tcpSocket.Bind(iPEndPoint);
11 }
12 catch(Exception ex)
13 {
14     Console.WriteLine("绑定报错: " + ex.Message);
15 }
16 // 监听客户端的连接请求
17 tcpSocket.Listen(10);
18 Console.WriteLine("服务端绑定监听结束, 等待客户端连入");
19 // 等待客户端连接, 返回一个新的客户端连接Socket (阻塞式)
20 Socket socketClient = tcpSocket.Accept();
21 Console.WriteLine("有客户端连入");
22
23 // 接收或发送消息
24 socketClient.Send(Encoding.UTF8.GetBytes("欢迎连入服务端"));
25 byte[] result = new byte[1024];
26 int length = socketClient.Receive(result);
27 Console.WriteLine("接收到了{0}发来的消息: {1}",
28     socketClient.RemoteEndPoint.ToString(),
29     Encoding.UTF8.GetString(result, 0, length));
30
31 // 释放连接
32 socketClient.Shutdown(SocketShutdown.Both);
33
34 // 关闭Socket
35 socketClient.Close();
36
37 Console.WriteLine("按任意键退出");
38 Console.ReadKey();

```

◦ 客户端

```

1 // 创建Tcp Socket
2 Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
3     ProtocolType.Tcp);
4 // 与服务端建立连接

```

```

5  IPEndPoint iPEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8080);
6  try
7  {
8      socket.Connect(iPEndPoint);
9  }
10 catch (SocketException ex)
11 {
12     if (ex.ErrorCode == 10061)
13     {
14         Debug.Log("服务器拒绝链接");
15     }
16     else
17     {
18         Debug.Log("连接服务器失败" + ex.ErrorCode);
19     }
20     return;
21 }
22
23 // 接收或发送数据
24 byte[] bytes = new byte[1024];
25 int length = socket.Receive(bytes);
26 Debug.Log("收到服务端发来的消息：" + Encoding.UTF8.GetString(bytes, 0,
    length));
27
28 socket.Send(Encoding.UTF8.GetBytes("客户端发送了一条消息"));
29
30 // 释放连接
31 socket.Shutdown(SocketShutdown.Both);
32
33 // 关闭Socket
34 socket.Close();

```

服务端TcpSocket

- 使用线程等待客户端连接，使用线程接收客户端消息，使用线程池处理客户端消息

```

1  using System.Net;
2  using System.Net.Sockets;
3  using System.Text;
4
5  namespace ServerProgram
6  {
7      internal class Program
8      {
9          static Socket socket;

```

```
10     static List<Socket> clientSockets = new List<Socket>();
11
12     static bool isClose = false;
13     static void Main(string[] args)
14     {
15         // 建立Socket, 绑定IP和端口并监听客户端连入
16         socket = new Socket(AddressFamily.InterNetwork,
17             SocketType.Stream, ProtocolType.Tcp);
18         IPEndPoint ipEndPoint = new
19 IPEndPoint(IPAddress.Parse("127.0.0.1"), 8080);
20         socket.Bind(ipEndPoint);
21         socket.Listen(10);
22
23         // 新开一个线程, 等待客户端连接, 不阻塞主线程
24         Thread acceptThread = new Thread(AcceptClientConnect);
25         acceptThread.Start();
26
27         // 收发消息
28         Thread receiveThread = new Thread(ReceiveMessage);
29         receiveThread.Start();
30
31         // 关闭连接和Socket
32         while (!isClose)
33         {
34             string input = Console.ReadLine();
35             if (input == "Quit")
36             {
37                 isClose = true;
38                 for (int i = 0; i < clientSockets.Count; i++)
39                 {
40                     clientSockets[i].Shutdown(SocketShutdown.Both);
41                     clientSockets[i].Close();
42                 }
43                 clientSockets.Clear();
44                 break;
45             }
46             else if(input.Substring(0, 2) == "B:")
47             {
48                 for (int i = 0; i < clientSockets.Count; i++)
49                 {
50                     clientSockets[i].Send(Encoding.UTF8.GetBytes(input.Substring(2)));
51                 }
52             }
53         }
54     }
```

```

55     /// <summary>
56     /// 接收客户端的连接请求，并把返回的socket对象存入列表
57     /// </summary>
58     static void AcceptClientConnect()
59     {
60         while(true)
61         {
62             Socket clientSocket = socket.Accept();
63             clientSockets.Add(clientSocket);
64             clientSocket.Send(Encoding.UTF8.GetBytes("欢迎连接服务端"));
65         }
66     }
67
68     /// <summary>
69     /// 遍历客户端Socket列表，接收消息
70     /// </summary>
71     static void ReceiveMessage()
72     {
73         Socket clientSocket;
74         byte[] result = new byte[1024 * 1024];
75         int length = 0;
76         int i = 0;
77         while (true)
78         {
79             for(i = 0; i < clientSockets.Count; i++)
80             {
81                 clientSocket = clientSockets[i];
82                 // 判断Socket是否有可以接收的消息
83                 if(clientSocket.Available > 0)
84                 {
85                     length = clientSocket.Receive(result);
86
87                     // 处理消息的方法放入线程池
88                     // 传入参数：客户端Socket的引用和反序列化后的消息
89                     ThreadPool.QueueUserWorkItem(HandleMessage,
90                                     (clientSocket, Encoding.UTF8.GetString(result, 0,
length)));
91                 }
92             }
93         }
94     }
95
96     /// <summary>
97     /// 处理消息
98     /// </summary>
99     /// <param name="obj">消息对象，包含客户端Socket和消息内容的元组</param>
100    static void HandleMessage(object obj)

```



```

101     {
102         (Socket s, string str) info = ((Socket s, string str))obj;
103         Console.WriteLine("收到客户端{0}发来的消息: {1}",
104             info.s.RemoteEndPoint.ToString(), info.str);
105     }
106 }
107 }

```

• 问题和分析

- 在控制台输入"Quit"后，clientSocket = clientSockets[i]这一行报错：

```

1 System.ArgumentOutOfRangeException: "Index was out of range.
2     Must be non-negative and less than the size of the collection.
   Arg_ParamName_Name"

```

- 原因分析：在控制台输入"Quit"后，主线程执行了clientSockets.Clear(); 但线程池中的接收消息的方法仍在for循环遍历中，而clientSockets的列表元素个数已清零
- 解决方法：拷贝一份clientSockets副本；使用并发集合；标记并延迟删除...

封装

- ClientSocket模块：封装客户端Socket，管理单个客户端Socket的状态；提供发送、接收消息等功能

```

1 internal class ClientSocket
2 {
3     private static int CLIENT_BEGIN_ID = 1;
4     public int clientID;
5     public Socket socket;
6
7     public ClientSocket(Socket socket)
8     {
9         this.clientID = CLIENT_BEGIN_ID;
10        ++CLIENT_BEGIN_ID;
11        this.socket = socket;
12    }
13
14    /// <summary>
15    /// 是否是连接状态
16    /// </summary>
17    public bool Connected { get { return socket.Connected; } }
18

```

```
19    /// <summary>
20    /// 关闭Socket
21    /// </summary>
22    public void Close()
23    {
24        if (socket != null)
25        {
26            socket.Shutdown(SocketShutdown.Both);
27            socket.Close();
28            socket = null;
29        }
30    }
31
32    /// <summary>
33    /// 发送消息
34    /// </summary>
35    /// <param name="info">需要发送的字符串消息</param>
36    public void Send(string info)
37    {
38        if (socket == null)
39        {
40            return;
41        }
42        try
43        {
44            socket.Send(Encoding.UTF8.GetBytes(info));
45        }
46        catch (Exception ex)
47        {
48            Console.WriteLine("发消息出错: " + ex.Message);
49            this.Close();
50        }
51    }
52
53    /// <summary>
54    /// 接收消息
55    /// </summary>
56    public void Receive()
57    {
58        if (socket == null)
59        {
60            return;
61        }
62        try
63        {
64            if (socket.Available > 0)
65            {
```

```

66         byte[] result = new byte[1024 * 10];
67         int length = socket.Receive(result);
68         ThreadPool.QueueUserWorkItem(HandleMessage,
69             Encoding.UTF8.GetString(result, 0, length));
70     }
71 }
72 catch (Exception ex)
73 {
74     Console.WriteLine("收消息出错: " + ex.Message);
75     this.Close();
76 }
77 }
78
79 private void HandleMessage(object obj)
80 {
81     string str = obj as string;
82     Console.WriteLine("收到客户端{0}发来的消息: {1}",
83         this.socket.RemoteEndPoint.ToString(), str);
84 }
85 }

```

- ServerSocket模块：管理所有连入的客户端Socket；提供开启、监听连入、接收消息、发送消息等方法

```

1 internal class ServerSocket
2 {
3     // 服务端Socket
4     public Socket socket;
5     // 客户端连接的所有Socket, 用字典保存
6     public Dictionary<int, ClientSocket> clientDict = new Dictionary<int,
ClientSocket>();
7
8     // 是否停止
9     private bool isClose;
10
11     /// <summary>
12     /// 开启服务端Socket
13     /// </summary>
14     /// <param name="host">主机IP地址</param>
15     /// <param name="port">绑定的端口号</param>
16     /// <param name="num">最大连接数量</param>
17     public void Start(string host, int port, int num)
18     {
19         isClose = false;
20

```

```
21     socket = new Socket(AddressFamily.InterNetwork,
22         SocketType.Stream, ProtocolType.Tcp);
23     IPEndPoint ipEndPoint = new IPEndPoint(IPAddress.Parse(host), port);
24     socket.Bind(ipEndPoint);
25     socket.Listen(num);
26
27     // 新开线程监听客户端连入
28     ThreadPool.QueueUserWorkItem(Accept);
29     // 新开线程接收客户端消息
30     ThreadPool.QueueUserWorkItem(Receive);
31 }
32 // 关闭服务端
33 public void Close()
34 {
35     isClose = true;
36
37     foreach (ClientSocket client in clientDict.Values)
38     {
39         client.Close();
40     }
41
42     clientDict.Clear();
43
44
45     if (socket != null)
46     {
47         try
48         {
49             // 使用非阻塞模式或异步操作
50             socket.Blocking = false;
51             socket.Shutdown(SocketShutdown.Both);
52         }
53         catch (SocketException ex)
54         {
55             // 处理套接字异常
56             Console.WriteLine($"SocketException: {ex.Message}");
57         }
58         finally
59         {
60             socket.Close();
61             socket = null;
62         }
63     }
64 }
65
66 // 监听客户端连入
67 private void Accept(Object obj)
```

```
68     {
69         while(!isClose)
70         {
71             try
72             {
73                 Socket clientSocket = socket.Accept();
74                 ClientSocket client = new ClientSocket(clientSocket);
75                 // 给客户端发消息
76                 client.Send("欢迎连入服务端");
77                 // 把连入的客户端socket存入字典
78                 clientDict.Add(client.clientID, client);
79             }
80             catch(Exception ex)
81             {
82                 Console.WriteLine("客户端连入报错: " + ex.Message);
83             }
84         }
85     }
86
87     // 接收客户端消息
88     private void Receive(object obj)
89     {
90         while (!isClose)
91         {
92             if (clientDict.Count > 0)
93             {
94                 // 使用for循环代替foreach避免客户端断线重连报错
95                 var clients = clientDict.Values.ToList();
96                 for (int i = 0; i < clients.Count; i++)
97                 {
98                     clients[i].Receive();
99                 }
100             }
101         }
102     }
103
104     // 广播
105     public void Broadcast(string info)
106     {
107         foreach(ClientSocket clientSocket in clientDict.Values)
108         {
109             clientSocket.Send(info);
110         }
111     }
112 }
```

客户端TcpSocket

- 客户端Socket封装在网络模块（NetManager）中，提供开启连接、断开连接、发送消息、接收消息的方法
- 发送消息、接收消息各开一个线程，不阻塞主线程；子线程与主线程的数据交互通过消息队列进行

```
1 public class NetManager : MonoBehaviour
2 {
3     private static NetManager instance;
4     public static NetManager Instance => instance;
5
6     // 客户端Socket
7     private Socket socket;
8
9     // 用于发送消息的队列，主线程往里面放，发送线程从里面取
10    private Queue<string> sendMsgQueue = new Queue<string>();
11    // 用于接收消息的队列，子线程往里面放，主线程从里面取
12    private Queue<string> receiveMsgQueue = new Queue<string>();
13    // 用于单次接收消息的容器
14    private byte[] receiveBytes = new byte[1024 * 1024];
15    // 返回收到的字节数
16    int receiveNum;
17
18    // 发送消息的线程
19    private Thread sendMsgThread;
20    // 接收消息的线程
21    private Thread receiveThread;
22    // 是否连接
23    private bool isConnected;
24
25    private void Awake()
26    {
27        instance = this;
28        DontDestroyOnLoad(this.gameObject);
29    }
30
31    void Update()
32    {
33        // 处理消息
34        if(receiveMsgQueue.Count > 0)
35        {
36            Debug.Log(receiveMsgQueue.Dequeue());
37        }
38    }
39
```

```
40     void OnDestroy()
41     {
42         Close();
43     }
44
45     // 连接服务端
46     public void Connect(string ip, int port)
47     {
48         if (isConnected)
49         {
50             return;
51         }
52
53         if(socket == null)
54         {
55             socket = new Socket(AddressFamily.InterNetwork,
56                                 SocketType.Stream, ProtocolType.Tcp);
57         }
58
59         IPEndPoint iPEndPoint = new IPEndPoint(IPAddress.Parse(ip), port);
60
61         try
62         {
63             socket.Connect(iPEndPoint);
64             isConnected = true;
65             // 开启发送消息的线程
66             ThreadPool.QueueUserWorkItem(SendMsg);
67             // 开启接收消息的线程
68             ThreadPool.QueueUserWorkItem(ReceiveMsg);
69
70         }
71         catch(SocketException ex)
72         {
73             if(ex.ErrorCode == 10061)
74             {
75                 Debug.Log("服务器拒绝链接");
76             }
77             else
78             {
79                 Debug.Log("连接失败: " + ex.ErrorCode + ex.Message);
80             }
81         }
82     }
83
84     // 发送消息
85     public void Send(string info)
86     {
```

```

87         sendMsgQueue.Enqueue(info);
88     }
89
90     // 线程方法：发送消息
91     private void SendMsg(object obj)
92     {
93         while(isConnected)
94         {
95             if(sendMsgQueue.Count > 0)
96             {
97                 socket.Send(Encoding.UTF8.GetBytes(sendMsgQueue.Dequeue()));
98             }
99         }
100     }
101
102     // 线程方法：接收消息
103     private void ReceiveMsg(object obj)
104     {
105         while(isConnected)
106         {
107             if(socket.Available > 0)
108             {
109                 receiveNum = socket.Receive(receiveBytes);
110                 receiveMsgQueue.Enqueue(Encoding.UTF8.
111                     GetString(receiveBytes, 0, receiveNum));
112             }
113         }
114     }
115
116     // 关闭连接
117     public void Close()
118     {
119         if(socket != null)
120         {
121             isConnected = false;
122             socket.Shutdown(SocketShutdown.Both);
123             socket.Close();
124         }
125     }
126 }

```

区分消息类型

- 网络通信中，使用字节数组进行数据传递，接收到消息时需要知道数据类型，才能使用对应的类来反序列化

- 解决方法：为发送的信息添加标识(ID)，标识可以是int、short、bytes、long等...
 - 如果使用int类型作为消息ID，前4个字节是消息ID，后面的字节是消息内容
- 创建消息基类，基类继承自BaseData并添加消息ID的属性和获取方法

```
1 public class BaseMessage : BaseData
2 {
3     public override byte[] ConvertToArray()
4     {
5         throw new NotImplementedException();
6     }
7
8     public override int GetBytesLength()
9     {
10         throw new NotImplementedException();
11     }
12
13     public override int ReadFromByteArray(byte[] bytes, int beginIndex)
14     {
15         throw new NotImplementedException();
16     }
17
18     /// <summary>
19     /// 获取消息ID
20     /// </summary>
21     /// <returns>消息ID</returns>
22     public virtual int GetMessageID()
23     {
24         return 0;
25     }
26 }
```

- 需要传递消息的类继承该基类，序列化时需要加上消息ID，反序列化时无需解析消息ID

```
1 public class PlayerMessage : BaseMessage
2 {
3     public int playerID;
4     public PlayerData playerData;
5
6     public override byte[] ConvertToArray()
7     {
8         int index = 0;
9         byte[] bytes = new byte[GetBytesLength()];
10     }
```

```

11     WriteInt(bytes, GetMessageID(), ref index);
12     WriteInt(bytes, playerID, ref index);
13     WriteObject(bytes, playerData, ref index);
14
15     return bytes;
16 }
17
18 public override int ReadFromByteArray(byte[] bytes, int beginIndex = 0)
19 {
20     // 反序列化不需要解析消息ID
21     int index = beginIndex;
22     playerID = ReadInt(bytes, ref index);
23     playerData = ReadObject<PlayerData>(bytes, ref index);
24
25     return index - beginIndex;
26 }
27
28 public override int GetBytesLength()
29 {
30     return sizeof(int) + // 消息ID的长度
31           sizeof(int) + playerData.GetBytesLength();
32 }
33
34 /// <summary>
35 /// 自定义的玩家消息ID
36 /// </summary>
37 /// <returns>玩家消息ID: 1</returns>
38 public override int GetMessageID()
39 {
40     return 1;
41 }
42 }

```

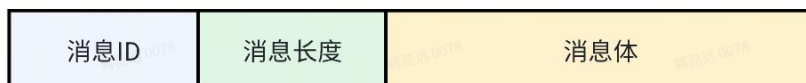
分包和粘包

- 定义：分包、粘包指网络通信中由于各种因素造成的消息与消息之间出现的两种状态，可能同时发生
 - 分包：一个消息分成多个消息进行发送
 - 粘包：一个消息和另一个消息黏在一起
- 出现原因：
 - 分包：当一个数据包过大，超过了网络传输的最大传输单元或TCP发送缓冲区的大小时，数据包会被拆分成多个小包进行传输。

- 粘包：当多个小数据包在发送时被合并成一个大包进行传输，或者接收端在一次读取操作中接收到多个数据包时，数据包会粘在一起。

解决方法

- 固定长度封装：将每个数据包封装成固定长度，不足部分用填充字符填充，这种方法简单但可能浪费带宽。
- 使用分隔符：在每个数据包的末尾使用固定的分隔符（如 \r\n），接收端通过分隔符来区分不同的数据包，这种方法适用于文本数据传输。
- 消息头包含长度信息：在数据包的头部包含消息长度信息，接收端根据长度信息读取完整的数据包，这是最常用的方法，适用于大多数应用场景；方法如下
 - 判断字节数组的状态的依据：消息长度
 - 为消息添加头部，头部记录消息的长度
 - 接收消息时通过消息长度来判断是否分包、粘包
 - 从而对消息进行合并、拆分处理
 - 一个消息由以下部分构成：



- 自定义协议：设计自定义协议来处理分包和粘包问题。例如，使用Netty框架中的解码器来处理。

消息长度方法的实践

- 消息类中的给消息的头部添加消息长度：
 - GetBytesLength返回的字节数组长度新增消息体长度（一般是Int）的长度
 - ConvertToByteArray把消息体长度写入字节数组

```
1 public class PlayerMessage : BaseMessage
2 {
3     public int playerId;
4     public PlayerData playerData;
5
6     public override byte[] ConvertToByteArray()
7     {
8         int index = 0;
9         int bytesLength = GetBytesLength();
10        byte[] bytes = new byte[bytesLength];
11    }
```

```

12      // 写入消息ID
13      WriteInt(bytes, GetMessageID(), ref index);
14      // 写入消息体长度 (不包含表示消息ID和消息体长度的前2个int)
15      WriteInt(bytes, bytesLength - 2 * sizeof(int), ref index);
16      // 写入消息体
17      WriteInt(bytes, playerId, ref index);
18      WriteObject(bytes, playerData, ref index);
19
20      return bytes;
21  }
22
23  public override int ReadFromByteArray(byte[] bytes, int beginIndex = 0)
24  {
25      // 反序列化不需要解析消息ID
26      int index = beginIndex;
27      playerId = ReadInt(bytes, ref index);
28      playerData = ReadObject<PlayerData>(bytes, ref index);
29
30      return index - beginIndex;
31  }
32
33  public override int GetBytesLength()
34  {
35      return sizeof(int) + // 消息ID
36             sizeof(int) + // 消息体的长度
37             sizeof(int) + playerData.GetBytesLength(); // 消息体本身
38  }
39
40  /// <summary>
41  /// 自定义的玩家消息ID
42  /// </summary>
43  /// <returns>玩家消息ID: 1</returns>
44  public override int GetMessageID()
45  {
46      return 1;
47  }
48  }

```

- 接收单条消息的方法，在线程方法中调用

```
1 HandleReceiveMessage(receiveBytes, receiveNum);
```

- 声明用于缓存消息的容器

```
1 // 消息缓存的容器
2 private byte[] cacheBytes = new byte[1024 * 1024];
3 // 缓存的字节数
4 private int cacheNum;
```

- 单条消息的处理方法

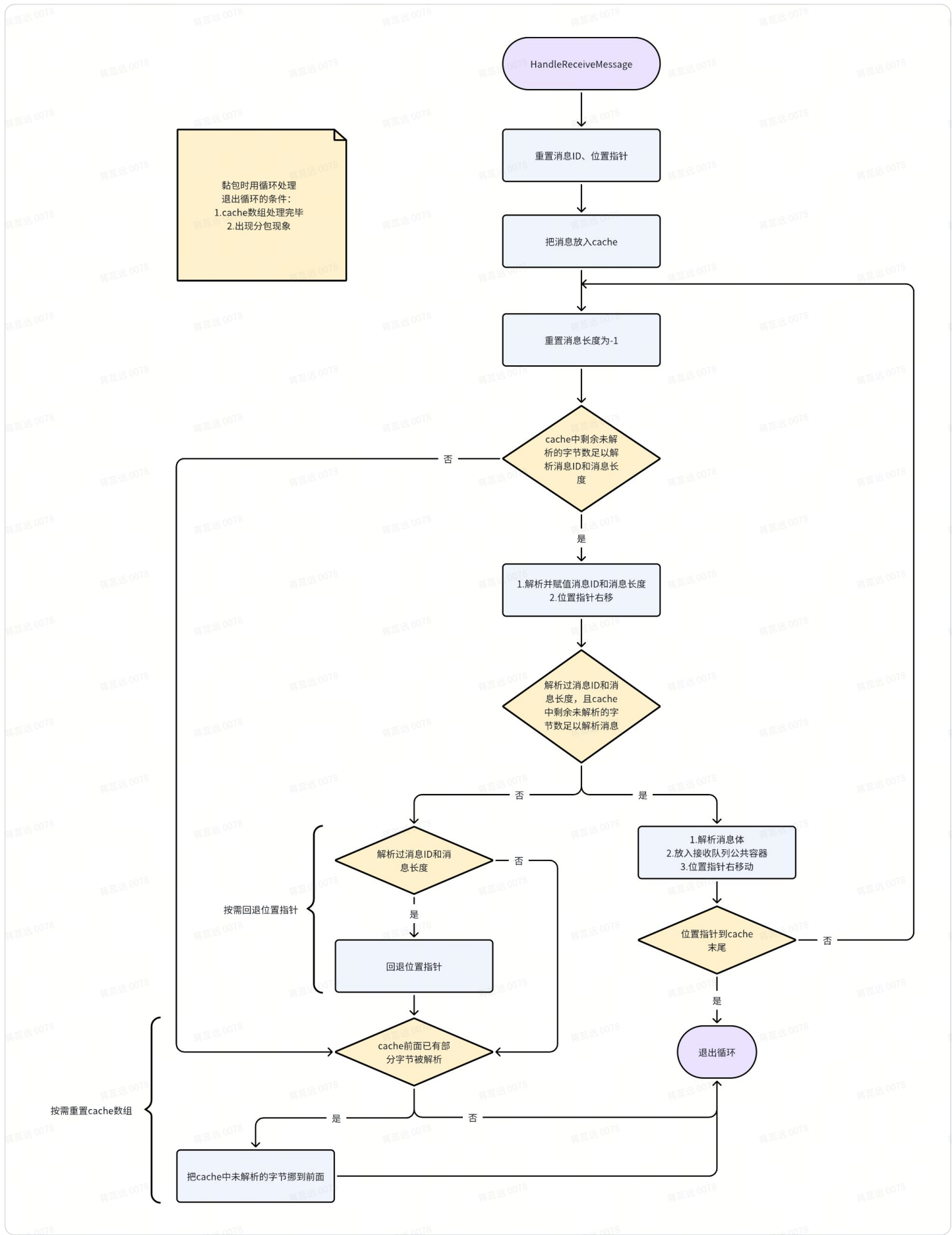
```
1 private void HandleReceiveMessage(byte[] receiveBytes, int receiveNum)
2 {
3     int msgID = 0;
4     int msgLength = 0;
5     int nowIndex = 0;
6
7     // 收到消息时先放到缓存容器中
8     receiveBytes.CopyTo(cacheBytes, cacheNum);
9     cacheNum += receiveNum;
10
11     while(true)
12     {
13         // 置成-1, 用于标识有没有解析过消息ID和消息长度
14         msgLength = -1;
15         // 能解析消息ID和消息长度
16         if (cacheNum - nowIndex >= 2 * sizeof(int))
17         {
18             // 解析消息ID
19             msgID = BitConverter.ToInt32(cacheBytes, nowIndex);
20             nowIndex += sizeof(int);
21
22             // 解析消息长度
23             msgLength = BitConverter.ToInt32(cacheBytes, nowIndex);
24             nowIndex += sizeof(int);
25         }
26
27         // 前面解析过消息ID和消息长度, 且能解析完整的消息体, 说明没出现分包现象
28         if (msgLength != -1 && cacheNum - nowIndex >= msgLength)
29         {
30             // 解析消息体
31             BaseMessage baseMessage = null;
32             // 判断消息类型, 转换成不同的对象
33             switch (msgID)
34             {
35                 case 1:
36                     baseMessage = new PlayerMessage();
37                     baseMessage.ReadFromByteArray(cacheBytes, nowIndex);
38                     break;
```

```

39         default:
40             break;
41     }
42     // 收到消息，解析为对象，并放入接收队列公共容器
43     if(baseMessage != null)
44     {
45         receiveMsgQueue.Enqueue(baseMessage);
46     }
47     nowIndex += msgLength;
48
49     // 解析完消息体后，判断解析到的位置是否已到cache末尾
50     if (nowIndex == cacheNum) {
51         // 清空缓存容器
52         cacheNum = 0;
53         break;
54     }
55 }
56 // 剩余包体长度小于消息体长度，出现了分包现象
57 else
58 {
59     // 解析了消息ID和长度但是没有解析消息体，需要回退nowIndex（都没解析）
60     if(msgLength != -1)
61     {
62         nowIndex -= 2 * sizeof(int);
63     }
64
65     // 重置缓存容器，把前面解析过的字节从缓存容器中去除，把没解析过的移上来
66     if (nowIndex > 0)
67     {
68         Array.Copy(cacheBytes, nowIndex, cacheBytes, 0, cacheNum -
nowIndex);
69         cacheNum = cacheNum - nowIndex;
70     }
71
72     break;
73 }
74 }
75 }

```

- 处理流程



自定义退出消息

- 解决的问题：

- 服务端的socket.Connect变量表示的是上一次收发消息是否成功，因此无法准确判断客户端状态
- 因此需要自定义退出消息，由客户端断开时主动发送，服务端接收到后把对应socket从字典中移除
- 客户端的处理：
 - 新增QuitMessage类表示自定义退出消息（服务端同步添加此类）

```
1 public class QuitMessage : BaseMessage
2 {
3     public override byte[] ConvertToByteArray()
4     {
5         int index = 0;
6         byte[] bytes = new byte[GetBytesLength()];
7         WriteInt(bytes, GetMessageID(), ref index);
8         WriteInt(bytes, 0, ref index);
9
10        return bytes;
11    }
12
13    public override int GetBytesLength()
14    {
15        return sizeof(int) + sizeof(int);
16    }
17
18    public override int ReadFromByteArray(byte[] bytes, int beginIndex)
19    {
20        return 0;
21    }
22
23    public override int GetMessageID()
24    {
25        return 88;
26    }
27 }
```

- 在NetManager->Close方法中调用

```
1 public void Close()
2 {
3     if(socket != null)
4     {
5         Debug.Log("客户端主动断开连接");
```



```

6      // 主动发送关闭连接的消息给服务端
7      QuitMessage quitMessage = new QuitMessage();
8      socket.Send(quitMessage.ConvertToByteArray());
9
10     socket.Shutdown(SocketShutdown.Both);
11     socket.Disconnect(false);
12     socket.Close();
13     socket = null;
14     isConnected = false;
15 }
16 }

```

- 服务端的处理:

- ServerSocket中新增待删除的Socket列表delList和添加列表的方法

```

1  // 待移除的客户端Socket, 避免在foreach时在字典中移除引起的报错
2  private List<ClientSocket> delList = new List<ClientSocket>();
3  // 添加待移除的Socket
4  public void AddDelSocket(ClientSocket clientSocket)
5  {
6      if(!delList.Contains(clientSocket))
7      {
8          delList.Add(clientSocket);
9      }
10 }

```

- ServerSocket中新增关闭客户端Socket并从clientDict中移除的方法

```

1  // 关闭客户端Socket
2  private void CloseClientSocket(ClientSocket socket)
3  {
4      lock (clientDict)
5      {
6          socket.Close();
7          // 从字典中移除
8          if (clientDict.ContainsKey(socket.clientID))
9          {
10             clientDict.Remove(socket.clientID);
11             Console.WriteLine("客户端{0}主动断开连接", socket.clientID);
12          }
13      }
14 }

```

- ServerSocket->Receive中遍历字典调用每个客户端Socket的接收方法之后，要移除待移除列表中的Socket

```
1 // 接收客户端消息
2 private void Receive(object obj)
3 {
4     while (!isClose)
5     {
6         if (clientDict.Count > 0)
7         {
8             lock(clientDict)
9             {
10                 foreach(ClientSocket client in clientDict.Values)
11                 {
12                     client.Receive();
13                 }
14
15                 // 检测有没有断开连接 (delList中) 的Socket, 把它移除
16                 for(int i = 0; i < delList.Count; i++)
17                 {
18                     CloseClientSocket(delList[i]);
19                 }
20                 delList.Clear();
21             }
22         }
23     }
24 }
```

- ClientSocket->HandleReceiveMessage中新增自定义退出消息的ID判断

```
1 // 判断消息类型, 转换成不同的对象
2 switch (msgID)
3 {
4     case 1:
5         baseMessage = new PlayerMessage();
6         baseMessage.ReadFromByteArray(cacheBytes, nowIndex);
7         break;
8     case 88:
9         // 自定义退出消息的ID是88
10        baseMessage = new QuitMessage();
11        break;
12    default:
13        break;
```

```
14 }
```

- ClientSocket->HandleMessage中如果判断类型为QuitMessage，把自身加入待删除列表

```
1 private void HandleMessage(object obj)
2 {
3     BaseMessage msg = obj as BaseMessage;
4     if (msg is PlayerMessage)
5     {
6         PlayerMessage playerMessage = msg as PlayerMessage;
7         Console.WriteLine("playerID: " + playerMessage.playerID + "\n" +
8                             "name: " + playerMessage.playerData.name + "\n"
9                             + "atk: " + playerMessage.playerData.atk +
10                             "\n" + "sex: " + playerMessage.playerData.sex);
11     }
12     else if(msg is QuitMessage)
13     {
14         // 收到断开连接的消息，把自己添加到待移除的列表中
15         Program.socket.AddDelSocket(this);
16     }
17 }
```

心跳消息

- 定义：心跳信息是长连接中，客户端和服务端之间定期发送的一种特殊的数据包；用于通知对方自己还在线，以确保长连接的有效性
- 为什么需要心跳消息：
 - 避免非正常关闭客户端时，服务器无法正常收到退出消息
 - 自定义超时判断，如果超出某一时间没收到客户端的消息，证明客户端已断开连接
 - 避免客户端长期不发送消息，防火墙或路由器会断开连接，用心跳消息保持活跃状态
- 客户端实现：
 - 自定义心跳消息类

```
1 public class HeartMessage : BaseMessage
2 {
3     public override int GetBytesLength()
4     {
5         return sizeof(int) + sizeof(int);
```

```

6    }
7
8    public override byte[] ConvertToArray()
9    {
10       int index = 0;
11       byte[] bytes = new byte[GetBytesLength()];
12       WriteInt(bytes, GetMessageID(), ref index);
13       WriteInt(bytes, 0, ref index);
14
15       return bytes;
16    }
17
18    public override int ReadFromArray(byte[] bytes, int beginIndex)
19    {
20       return 0;
21    }
22
23    public override int GetMessageID()
24    {
25       return 999;
26    }
27 }

```

- NetManager中用Invoke定时发送心跳消息

```

1  // 发送心跳消息的间隔时间
2  private int SEND_HEART_MSG_INTERVAL = 2;
3  // 心跳消息对象
4  private HeartMessage heartMessage = new HeartMessage();
5
6  private void Awake()
7  {
8      instance = this;
9      DontDestroyOnLoad(this.gameObject);
10
11     // 定时发送心跳消息 (使用主线程)
12     InvokeRepeating("SendHeartMessage", 0, SEND_HEART_MSG_INTERVAL);
13 }
14
15 private void SendHeartMessage()
16 {
17     if (isConnected)
18     {
19         Send(heartMessage);
20     }

```

```
21 }
```

- 服务端实现：

- ClientSocket->HandleReceiveMessage新增心跳消息类型的判断

```
1 case 999:
2     baseMessage = new HeartMessage();
3     break;
```

- ClientSocket->HandleMessage更新上次收到心跳消息的时间

```
1 else if(msg is HeartMessage)
2 {
3     // 记录收到心跳消息的时间
4     lastTime = DateTime.Now.Ticks / TimeSpan.TicksPerSecond;
5     Console.WriteLine("收到客户端的{0}的心跳消息", this.clientID);
6 }
```

- ClientSocket->CheckTimeOut检测是否超时，若超时就把自身加入待删除列表

```
1 // 上一次收到心跳消息的时间
2 private long lastTime = -1;
3 // 判定超时的时间间隔
4 private static int TIME_OUT_TIME = 10;
5
6 /// <summary>
7 /// 检测超时，如果客户端超时，就加入待删除列表
8 /// </summary>
9 private void CheckTimeOut()
10 {
11     if (lastTime != -1 && DateTime.Now.Ticks / TimeSpan.TicksPerSecond -
        lastTime > TIME_OUT_TIME)
12     {
13         Program.socket.AddDelSocket(this);
14     }
15 }
```


- ClientSocket->Receive新增检测是否超时的方法，而Receive本身就在ServerSocket的帧循环中调用


```

1 try
2 {
3     if (socket.Available > 0)
4     {
5         byte[] result = new byte[1024 * 10];
6         int length = socket.Receive(result);
7         HandleReceiveMessage(result, length);
8     }
9
10    // 检测是否超时
11    CheckTimeOut();
12 }

```

异步通信常用方法

 同步方法和异步方法的区别：同步方法是阻塞模式+顺序执行，调用者必须等待方法执行完成后才能继续执行后续操作；异步方法是非阻塞模式+并发执行，调用者可以在方法执行过程中执行其他操作。

 C#网络通信模块中有两种异步方案，内部开多线程，通过回调形式返回结果

Begin开头的API：通过AsyncCallback参数传入回调函数，需要与End开头的方法配合使用

Async结尾的API：依赖SocketAsyncResult对象，通过在Complete参数中添加回调函数

Begin

- 服务端专用：BeginAccept和EndAccept

```

1 public IAsyncResult BeginAccept(AsyncCallback callback, object state);
2 public Socket EndAccept(IAsyncResult asyncResult);

```

```

1 socket.BeginAccept(AcceptCallBack, socket);
2 private void AcceptCallBack(IAsyncResult result)
3 {
4     try
5     {
6         Socket s = result.AsyncState as Socket;

```

```

7      Socket clientSocket = s.EndAccept(result);
8      // 把clientSocket加入字典
9      // 连入一个之后接着监听其他客户端连入
10     s.BeginAccept(AcceptCallBack, s);
11 }
12 catch(SocketException ex)
13 {
14     Debug.Log(ex.SocketErrorCode);
15 }
16 }

```

- 客户端专用：BeginConnect和EndConnect

```

1 public IAsyncResult BeginConnect(EndPoint remoteEP, AsyncCallback callback,
   object state);
2 public void EndConnect(IAsyncResult asyncResult);

```

```

1 socket.BeginConnect(ipEndPoint, (result) => {
2     Socket s = result.AsyncState as Socket;
3     try
4     {
5         s.EndConnect(result);
6         Debug.Log("客户端连接成功");
7     }
8     catch(SocketException ex)
9     {
10        Debug.Log("连接出错: " + ex.SocketErrorCode + ex.Message);
11    }
12 }, socket);

```

- 服务端和客户端通用

- 接收消息：BeginReceive和EndReceive

```

1 public IAsyncResult BeginReceive(byte[] buffer, int offset, int size,
   SocketFlags socketFlags, AsyncCallback callback, object state);
2 public int EndReceive(IAsyncResult asyncResult);

```

```

1 socket.BeginReceive(receiveBytes, index, receiveBytes.Length,

```

```

2     SocketFlags.None, ReceiveCallback, socketTcp);
3 private ReceiveCallback(IAsyncResult result)
4 {
5     Socket s = result.AsyncState as Socket;
6     try
7     {
8         int receiveLength = s.EndReceive(result);
9         // 进行消息处理
10
11         // 继续接收消息
12         s.BeginReceive(receiveBytes, index, receiveBytes.Length,
13             SocketFlags.None, ReceiveCallback, s);
14     }
15     catch(SocketException ex)
16     {
17         Debug.Log("接收消息出错: " + ex.SocketErrorCode + ex.Message);
18     }
19 }

```

- 发送消息：BeginSend和EndSend

```

1 public IAsyncResult BeginSend(byte[] buffer, int offset, int size, SocketFlags
   socketFlags, AsyncCallback callback, object state);
2 public int EndSend(IAsyncResult asyncResult);

```

```

1 socket.BeginSend(bytes, 0, bytes.Length, SocketFlags.None, (result) => {
2     Socket s = result.AsyncState as Socket;
3     try
4     {
5         int num = s.EndSend(result);
6         Debug.Log("成功发送{0}个字节", num);
7     }
8     catch(SocketException ex)
9     {
10         Debug.Log("发送消息出错: " + ex.SocketErrorCode + ex.Message);
11     }
12 }, socket);

```

Async

- 服务端专用：AcceptAsync

- 在委托Completed中添加回调函数


```
1 public event EventHandler<SocketAsyncEventArgs> Completed;
2 public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

```
1 SocketAsyncEventArgs e = new SocketAsyncEventArgs();
2 e.Complete += (socket, args) => {
3     if(args.SocketError == SocketError.Success) {
4         // 连接成功, 获取连入的客户端Socket
5         Socket clientSocket = args.AcceptSocket;
6         // 继续监听其他客户端连入
7         (socket as Socket).AcceptAsync(args);
8     } else {
9         Debug.Log("连入客户端失败" + args.SocketError);
10    }
11 }
12 socket.AcceptAsync(e);
```

- 客户端专用: ConnectAsync

- 在委托Completed中添加回调函数

```
1 SocketAsyncEventArgs e = new SocketAsyncEventArgs();
2 e.RemoteEndPoint = ipEndPoint;
3 e.Complete += (socket, args) => {
4     if(args.SocketError == SocketError.Success) {
5         Debug.Log("连接成功");
6     } else {
7         Debug.Log("连接失败");
8     }
9 }
10 socket.ConnectAsync(e);
```

- 服务端和客户端通用

- 发送消息: SendAsync

```
1 SocketAsyncEventArgs e = new SocketAsyncEventArgs();
2 // 设置需要发送的字节数组
3 e.SetBuffer(bytes, 0, bytes.Length);
4 e.Complete += (socket, args) => {
5     if(args.SocketError == SocketError.Success) {
6         Debug.Log("发送成功");
```

```

7     } else {
8         Debug.Log("发送失败" + args.SocketError);
9     }
10 }
11
12 socket.SendAsync(e);

```

- 接收消息：ReceiveAsync

```

1 SocketAsyncEventArgs e = new SocketAsyncEventArgs();
2 // 设置接收数据的容器
3 e.SetBuffer(new byte[1024*1024], 0, 1024*1024);
4 e.Complete += (socket, args) => {
5     if(args.Socket.Error == SocketError.Success){
6         // Buffer: 装字节的容器; BytesTransferred: 收取到的字节数个数
7         Encoding.UTF8.GetString(args.Buffer, 0, args.BytesTransferred);
8         // 重置容器
9         args.SetBuffer(0, args.Buffer.Length);
10        // 继续接收新的消息
11        (socket as Socket).ReceiveAsync(args);
12    } else {
13
14    }
15 }
16
17 socket.ReceiveAsync(e);

```

最大传输单元

- 最大传输单元（Maximum Transmission Unit, MTU）是通知通信对方能接受的数据服务单元的最大尺寸；以太网和802.3协议对数据帧的长度限制最大值分别为1500字节和1492字节
- 为了避免自动分包，需要限制最大传输单元；由于UDP包本身带有头部信息28，建议：
 - 局域网环境下，1472字节以内
 - 互联网环境下，548字节以内

Udp通信的简单应用

- Udp通信中客户端和服务端的收发逻辑一致

客户端

- Udp客户端通信实现收发字符串

```

1 // 创建一个Udp Socket并绑定本地的IP和端口
2 Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
    ProtocolType.Udp);
3 IPEndPoint iPEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8080);
4 socket.Bind(iPEndPoint);
5
6 // 指定服务端的IP和端口
7 IPEndPoint remoteIPEndPoint1 = new IPEndPoint(IPAddress.Parse("127.0.0.1"),
    8081);
8 // 发送消息
9 socket.SendTo(Encoding.UTF8.GetBytes("客户端给服务端发消息"), remoteIPEndPoint1);
10
11 // 接收消息
12 byte[] receiveBytes = new byte[512];
13 // 消息来源服务端的IP和端口
14 EndPoint remoteIPEndPoint2 = new IPEndPoint(IPAddress.Any, 0);
15 int receiveNum = socket.ReceiveFrom(receiveBytes, ref remoteIPEndPoint2);
16
17 Debug.Log((remoteIPEndPoint2 as IPEndPoint).Address.ToString() + "发来了消息: "
    + Encoding.UTF8.GetString(receiveBytes, 0, receiveNum));
18
19 // 释放连接, 关闭Socket
20 socket.Shutdown(SocketShutdown.Both);
21 socket.Close();

```

服务端

- Udp服务端通信实现收发字符串

```

1 // 创建一个Udp Socket并绑定本地的IP和端口
2 Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
    ProtocolType.Udp);
3 IPEndPoint iPEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8081);
4 socket.Bind(iPEndPoint);
5
6 // 接收消息
7 byte[] receiveBytes = new byte[512];
8 // 消息来源服务端的IP和端口
9 EndPoint remoteIPEndPoint2 = new IPEndPoint(IPAddress.Any, 0);
10 int receiveNum = socket.ReceiveFrom(receiveBytes, ref remoteIPEndPoint2);
11 Console.WriteLine((remoteIPEndPoint2 as IPEndPoint).Address.ToString() + "发来了
    消息: " + Encoding.UTF8.GetString(receiveBytes, 0, receiveNum));
12
13 // 发送消息
14 socket.SendTo(Encoding.UTF8.GetBytes("服务端给客户端发消息"), remoteIPEndPoint2);

```

```
15
16 // 释放连接, 关闭Socket
17 socket.Shutdown(SocketShutdown.Both);
18 socket.Close();
```

Udp通信异步方法

Begin

- 发送消息：BeginSendTo和EndSendTo

```
1 byte[] bytes = Encoding.UTF8.GetBytes("123");
2 EndPoint ipEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8080);
3 socket.BeginSendTo(bytes, 0, bytes.Length,
4     SocketFlags.None, ipEndPoint, SendCallback, socket);
5
6 // 异步发送回调函数
7 private void SendCallback(IAsyncResult result)
8 {
9     try
10     {
11         Socket s = result.AsyncState as Socket;
12         // End和Begin配合使用
13         s.EndSendTo(result);
14         print("发送成功");
15     }
16     catch(SocketException ex)
17     {
18         print("发送失败: " + ex.SocketErrorCode);
19     }
20 }
```

- 接收消息：BeginReceiveFrom和EndReceiveFrom

```
1 socket.BeginReceiveFrom(cacheBytesm 0, cacheBytes.Length,
2     SocketFlags.None, ref ipEndPoint, ReceiveCallback, (socket, ipEndPoint));
3
4 // 异步接收回调函数
5 private void ReceiveCallback(IAsyncResult result)
6 {
7     try
8     {
```

```

9      (Socket s, EndPoint ipEndPoint) info = ((Socket,
EndPoint))result.AsyncState;
10     // 记录了接收的字节数个数
11     int num = info.s.EndReceiveFrom(result, ref info.ipEndPoint);
12     // 处理消息
13
14     // 继续接收消息
15     info.s.BeginReceiveFrom(cacheBytesm 0, cacheBytes.Length,
16     SocketFlags.None, ref info.ipEndPoint, ReceiveCallback, info);
17 }
18 catch(SocketException ex)
19 {
20     print("接收消息失败: " + ex.SocketErrorCode);
21 }
22 }

```

Async

- 发送消息：SendToAsync

```

1 SocketAsyncEventArgs e = new SocketAsyncEventArgs();
2 e.SetBuffer(bytes, 0, bytes.length);
3 e.RemoteEndPoint = ipEndPoint;
4 e.Completed += SendCallback;
5
6 // 发送消息回调函数
7 private void SendCallback(object s, SocketAsyncEventArgs args)
8 {
9     if(args.SocketError == SocketError.Success)
10     {
11         print("发送成功");
12     }
13     else
14     {
15         print("发送失败: " + args.SocketError);
16     }
17 }
18
19 socket.SendToAsync(e);

```

- 接收消息：ReceiveFromAsync

```

1 SocketAsyncEventArgs e = new SocketAsyncEventArgs();

```

```

2 e.SetBuffer(cacheBytes, 0, cacheBytes.Length);
3 e.RemoteEndPoint = new IPEndPoint(IPAddress.Any, 0);
4 e.Completed += ReceiveCallback;
5
6 // 接收消息回调函数
7 private void ReceiveCallback(object s, SocketAsyncEventArgs args)
8 {
9     if(args.SocketError == SocketError.Success)
10    {
11        int num = args.BytesTransferred;
12        print("接收成功, 接收的字节数: " + num);
13        // 处理消息
14
15        // 重置容器
16        args.SetBuffer(0, cacheBytes.Length);
17        Socket socket = s as Socket;
18        socket.ReceiveFromAsync(args);
19    }
20    else
21    {
22        print("接收失败: " + args.SocketError);
23    }
24 }
25
26 socket.ReceiveFromAsync(e)

```

FTP

- FTP是文件传输协议，使用户可以把文件从一台主机拷贝到另一台主机上，除此之外，还提供登录、目录查询和绘画控制等功能。
- FTP的本质是TCP通信：通过FTP传输文件，双方至少需要建立两个TCP连接，一个是**控制连接**，用于传输FTP命令；一个是**数据连接**，用于传输文件数据。
- FTP有两种传输模式
 - 主动模式（Port模式）：服务器主动连接客户端并传输文件
 - 被动模式（Passive模式）：客户端主动连接服务器（控制连接和数据连接都由客户端发起）
- FTP有两种数据传输方式
 - ASCII传输方式：以ASCII编码方式传输数据，适用于传输仅包含英文的命令和文件
 - 二进制传输方式：可以指定编码方式传输命令和文件数据，应用较广

关键类

- NetworkCredential：通信凭证类

```
1 NetworkCredential n = new NetworkCredential("Alice", "Alice123");
```

- FtpWebRequest: Ftp文件传输协议客户端操作类
 - Create 创建新的WebRequest, 用于Ftp相关操作

```
1 FtpWebRequest req = FtpWebRequest.Create(new Uri("ftp://127.0.0.1/test.txt"))  
2     as FtpWebRequest;
```

- Abort: 终止正在传输的文件

```
1 req.Abort();
```

- GetRequestStream: 获取用于上传的流

```
1 Stream reqStream = req.GetRequestStream();
```

- GetResponse: 返回FTP服务器的响应

```
1 FtpWebResponse res = req.GetResponse() as FtpWebResponse;
```

- 成员变量

```
1 // Credentials 通信凭证  
2 req.Credentials = n;  
3 // KeepAlive 请求完成时是否保持与FTP服务器的连接  
4 req.KeepAlive = false;  
5 // Method 操作命令设置  
6 req.Method = WebRequestMethods.Ftp.DownloadFile;  
7 // UseBinary 是否使用二进制传输  
8 req.UseBinary = true;  
9 // RenameTo 重命名  
10 req.RenameTo = "myTest.txt";
```

- FtpWebResponse: FTP服务器对请求的响应

- Close: 释放所有资源

```
1 res.Close();
```

- GetResponseStream: 返回从FTP服务器下载数据的流

```
1 Stream resStream = res.GetResponseStream();
```

- 成员变量

```
1 // ContentLength 接收到的数据的长度
2 print(res.ContentLength);
3 // ContentType 接收到的数据的类型
4 print(res.ContentType);
5 // StatusCode 状态码
6 print(res.StatusCode);
7 // StatusDescription 状态文本
8 print(res.StatusDescription);
9 // BannerMessage FTP建立连接时登录前服务器发送的消息
10 print(res.BannerMessage);
11 // ExitMessage FTP会话结束时服务器发送的消息
12 print(res.ExitMessage);
13 // LastModified FTP服务器上的文件的上次修改时间
14 print(res.LastModified);
```

HTTP

- HTTP是超文本传输协议，是因特网上应用最广泛的一种网络传输协议；最初设计HTTP的目的是为了提供一种发布和接收由文本文件组成的HTML页面的方法，后来发展到除了文本数据外，还可以传输图片、音频、视频、压缩文件以及各种程序文件。
- HTTP的本质是TCP通信，因此不会丢包、乱序；它定义了Web客户端（一般指浏览器）如何从Web服务器请求Web页面，以及服务器如何把Web页面传给客户端。
- HTTP的工作原理
 - 以TCP方式工作
 - 是无状态的
 - 使用元信息作为标头

- HTTP/1.0仅支持短连接，HTTP/1.1支持持久连接，可以连续发送请求，并在上一个请求应答之前发送多个请求；目前使用的基本都是HTTP/1.1。
- HTTP协议的请求类型
 - GET：请求获取特定的资源，比如请求一个Web页面或请求获取一个资源
 - POST：请求提交数据进行处理，比如请求上传一个文件
 - HEAD：请求获取特定的资源，但是不会返回具体内容，只返回消息头
 - PUT：向指定位置上传最新内容
 - DELETE：删除指定资源
 - OPTIONS：返回服务器针对特定资源支持的HTTP请求方法
 - TRACE：回显服务器收到的请求
 - CONNECT：预留给能够将连接改为管道方式的代理服务器
- HTTP的请求格式
 - <request line>：请求行，说明请求类型、HTTP版本
 - <headers>：标头，服务端会使用的附加信息
 - <blank line>：空行，表明标头结束
 - <request body>：消息主体，可以包含任何数据
- HTTP的响应格式
 - <status line>：状态行，提供一个状态码来说明请求情况
 - <headers>：标头，客户端会使用的附加信息
 - <blank line>：空行，表面标头结束
 - <request body>：消息主体，可以包含任何数据

关键类

- HttpRequest：发送客户端请求的类
 - Create：创建新的WebRequest，用于进行HTTP相关操作

```
1 HttpRequest req = HttpRequest.Create(new
  Uri("http://10.0.71.145:8000/HttpServer/"))
2   as HttpRequest;
```

- Abort：如果正在进行文件传输，用此方法可终止传输

```
1 req.Abort();
```

- **GetRequestStream**: 获取用于上传的流

```
1 Stream s = req.GetRequestStream();
```

- **GetResponse**: 返回HTTP服务器响应

```
1 HttpResponseMessage res = req.GetResponse() as HttpResponseMessage;
```

- **成员变量**

```
1 // Credentials: 通信凭证
2 req.Credentials = new NetworkCredential("Alice", "Alice123");
3 // PreAuthenticate: 是否随请求发送一个身份验证标头, 需要身份验证时设置为true
4 req.PreAuthenticate = true;
5 // Headers: 构成标头的键值对的集合
6 req.Headers = new WebHeaderCollection();
7 // ContentLength: 发送的消息的字节数, 发送时需要先设置内容长度
8 req.ContentLength = 100;
9 // ContentType: 消息的类型
10 req.ContentType = "";
11 // Method: 操作命令设置
12 /*
13 WebRequestMethods.Http类中的操作命令属性
14 Get: 获取请求
15 Post: 提交请求
16 Head: 获取请求, 只返回消息头
17 Put: 向指定位置上传最新内容
18 Connect: 表示与代理一起使用的HTTP CONNECT协议方法, 该代理可以动态切换到隧道
19 MkCol: 请求在URI指定的位置新建集合
20 */
21 req.Method = WebRequestMethods.Http.Get;
```

- **HttpWebResponse**: 获取服务器返回信息的类

- **Close**: 释放资源

```
1 res.Close();
```

- `GetResponseStream`: 返回从FTP服务器下载数据的流

```
1 Stream s = res.GetResponseStream();
```

- 成员变量

```
1 // ContentLength: 接收的消息的字节数
2 long receiveLength = res.ContentLength;
3 // ContentType: 接收的消息的类型
4 string receiveType = res.ContentType;
5 // StatusCode: HTTP服务器下发的最新状态码
6 HttpStatusCode statusCode = res.StatusCode;
7 // StatusDescription: HTTP服务器下发的状态码的描述文本
8 string StatusDescription = res.StatusDescription;
9 // LastModified: HTTP服务器上的文件的上次修改时间
10 DateTime lastModified = res.LastModified;
```

Get和Post请求

- GET和POST是HTTP协议中最常用的两种请求方法，它们在数据传输方式、用途和安全性等方面有显著差异
 - 数据传输方式:
 - GET请求: 参数附加在URL后，以查询字符串的形式出现，如: `example.com?name=value`。数据通过URL传输，浏览器会记录访问的URL。
 - POST请求: 参数包含在请求体 (body) 中，而不是URL中。数据不会显示在URL中，相对更隐蔽。
 - 用途:
 - GET请求: 用于从服务器获取数据，不会改变服务器的状态。适用于获取资源或数据，例如搜索查询、获取用户信息等。
 - POST请求: 用于向服务器提交数据，可能会改变服务器的状态。适用于提交表单、上传文件或创建资源。
 - 安全性:
 - GET请求: 因为参数在URL中，容易被浏览器历史记录、服务器日志等记录，不适合传输敏感数据。

- POST请求：参数不在URL中，减少了暴露风险，适合传输敏感数据，但仍需通过HTTPS加密传输以确保数据安全。

- 缓存：

- GET请求：响应可以被浏览器缓存，提高响应速度。
- POST请求：响应通常不会被缓存，确保数据的实时性。

常用类

- HttpRequest：发送客户端请求的类

```
1 // Create: 创建新的WebRequest, 用于进行HTTP相关操作
2 HttpRequest req = HttpRequest.Create(new
    Uri("http://10.0.71.145:8000/HttpServer/")) as HttpRequest;
3 // Abort: 如果正在进行文件传输, 用此方法可终止传输
4 req.Abort();
5 // GetRequestStream: 获取用于上传的流
6 Stream s = req.GetRequestStream();
7 // GetResponse: 返回HTTP服务器响应
8 HttpResponse res = req.GetResponse() as HttpResponse;
9
10 // Credentials: 通信凭证
11 req.Credentials = new NetworkCredential("Alice", "Alice123");
12 // PreAuthenticate: 是否随请求发送一个身份验证标头, 需要身份验证时设置为true
13 req.PreAuthenticate = true;
14 // Headers: 构成标头的键值对的集合
15 req.Headers = new WebHeaderCollection();
16 // ContentLength: 发送的消息的字节数, 发送时需要先设置内容长度
17 req.ContentLength = 100;
18 // ContentType: 消息的类型
19 req.ContentType = "";
20
21 // Method: 操作命令设置
22 /*
23 WebRequestMethods.Http类中的操作命令属性
24 Get: 获取请求
25 Post: 提交请求
26 Head: 获取请求, 只返回消息头
27 Put: 向指定位置上传最新内容
28 Connect: 表示与代理一起使用的HTTP CONNECT协议方法, 该代理可以动态切换到隧道
29 MkCol: 请求在URI指定的位置新建集合
30 */
31 req.Method = WebRequestMethods.Http.Get;
```

- **HttpWebResponse**: 服务器返回的信息的类

```
1 // Close: 释放资源
2 res.Close();
3 // GetResponseStream: 返回从FTP服务器下载数据的流
4 Stream s = res.GetResponseStream();
5
6 // ContentLength: 接收的消息的字节数
7 long receiveLength = res.ContentLength;
8 // ContentType: 接收的消息的类型
9 string receiveType = res.ContentType;
10 // StatusCode: HTTP服务器下发的最新状态码
11 HttpStatusCode statusCode = res.StatusCode;
12 // StatusDescription: HTTP服务器下发的状态码的描述文本
13 string StatusDescription = res.StatusDescription;
14 // LastModified: HTTP服务器上的文件的上次修改时间
15 DateTime lastModified = res.LastModified;
```

Post请求内容类型

- MIME类型（也称为MIME类型或Content-Type）用于描述传输数据的类型和格式。

- 文本类型

- text/html
- text/css
- text/javascript
- text/plain: 用于发送纯文本数据。

| This is plain text data.

- 图片类型

- image/gif
- image/png
- image/jpeg
- image/bm
- image/webp
- image/x-icon
- image/vnd.microsoft.icon

- 音频类型

- audio/midi
- audio/mpeg
- audio/webm
- audio/ogg
- audio/wave
- 视频类型
 - video/webm
 - video/ogg
- 二进制类型（application类型）
 - application/octet-stream：通用的二进制类型，常用于文件下载或上传。
 - application/x-www-form-urlencoded：用于发送表单数据，数据以键值对形式编码，并通过URL编码特殊字符

```
| name=Tom&age=25
```
 - application/json：用于发送JSON格式的数据，适合传输结构化数据。

```
| {  
|   "name": "Tom",  
|   "age": 25  
| }
```
 - application/xml：用于发送XML格式的数据。

```
| <person>  
|   <name>Tom</name>  
|   <age>25</age>  
| </person>
```
- 复合类型（multipart类型）
 - multipart/form-data：用于上传文件或二进制数据，请求体以多部分形式编码。

```
| --boundary  
| Content-Disposition: form-data; name="field1"  
  
| value1  
| --boundary  
| Content-Disposition: form-data; name="file"; filename="example.txt"
```

Content-Type: text/plain

(file content)

--boundary--

UnityEngine命名空间下的WWW类

- WWW类是Unity提供的用于访问网页的类，可以通过该类下载和上传一些数据。
- WWW类在使用HTTP协议时，默认的请求类型是Get。
- 支持HTTP协议、FTP协议（仅限于匿名下载）、FILE协议（可以异步加载本地文件）。
- 一般配合协同程序使用。

常用方法

```
1 // 创建一个WWW请求
2 WWW www = new WWW("http://10.0.71.145:8000/HttpServer/文件.txt");
3 // 从下载数据返回一个音效切片AudioClip对象
4 www.GetAudioClip();
5 // 从下载数据返回一个MovieTexture对象
6 www.GetMovie();
7 // 用下载数据中的图像来替换现有的Texture2D对象
8 Texture2D tex = new Texture2D(100, 100);
9 www.LoadImageIntoTexture(tex);
10 // 从缓存加载AB包对象，若该包不在缓存，则自动下载储存到缓存中
11 WWW.LoadFromCacheOrDownload("http://192.168.50.109:8000/Http_Server/test.assetbundle", 1);
```

常用变量

```
1 www.assetBundle // 如果下载的数据是AB包，获取下载结果
2 www.audioClip // 如果下载的数据是音效切片，获取下载结果
3 www.bytes // 以字节数组的形式获取下载的内容
4 www.bytesDownloaded // 过去已下载的字节数
5 www.error // 错误信息
6 www.isDone // 判断下载是否已经完成
7 www.movie // 获取一个MovieTexture的下载结果
8 www.text // 如果下载的数据是字符串，获取下载结果
9 www.texture // 获取一个Texture2D的下载结果
```

UnityEngine命名空间下的WWWForm类

- WWWForm类使用HTTP协议进行上传，用到的请求类型是Post。

常用方法

```
1 WWWForm data = new WWWForm();
2 // 添加二进制数据
3 data.AddBinaryData(fieldName, contents);
4 // 添加字段的键和值
5 data.AddField(fieldName, value);
```

常用变量

UnityEngine命名空间下的UnityWebRequest类

- UnityWebRequest是一个Unity提供的模块化的系统类，用于构成HTTP请求和处理HTTP响应
- 主要目的是让Unity客户端与Web服务器进行交互
- 使用IMultipartFormSection接口将数据组合成多部分表单，用于通过 UnityWebRequest API 将复杂数据序列化成格式正确的字节，以便进行网络请求

常用方法

```
1 // 使用Get请求获取文本或二进制数据
2 IEnumerator LoadText()
3 {
4     UnityWebRequest req =
        UnityWebRequest.Get("http://10.0.71.145:8000/HttpServer/text.txt");
5
6     // 等待服务器响应，断开连接后再执行
7     yield return req.SendWebRequest();
8
9     if(req.result == UnityWebRequest.Result.Success)
10    {
11        // 返回文本
12        print(req.downloadHandler.text);
13        // 返回二进制字节数组
14        byte[] bytes = req.downloadHandler.data;
15        print("字节数组长度: " + bytes.Length);
16    }
17    else
18    {
```



```
19         print("资源获取失败: " + req.result + req.responseCode + req.error);
20     }
21 }
22 // 使用Get请求获取纹理数据
23 IEnumerator LoadTexture()
24 {
25     UnityWebRequest req =
26     UnityWebRequestTexture.GetTexture("http://10.0.71.145:8000/HttpServer/上
27     传.png");
28     // 等待服务器响应, 断开连接后再执行
29     yield return req.SendWebRequest();
30     if (req.result == UnityWebRequest.Result.Success)
31     {
32         image.texture = (req.downloadHandler as
33         DownloadHandlerTexture).texture;
34         image.texture = DownloadHandlerTexture.GetContent(req);
35     }
36     else
37     {
38         print("获取失败: " + req.result + req.responseCode + req.error);
39     }
40 }
41 // 使用Get请求获取AB包数据
42 IEnumerator LoadTexture()
43 {
44     UnityWebRequest req =
45     UnityWebRequestTexture.GetTexture("http://10.0.71.145:8000/HttpServer/上
46     传.png");
47     req.SendWebRequest();
48     // 每帧获取一下下载进度和下载字节数
49     while(!req.isDone)
50     {
51         print(req.downloadProgress);
52         print(req.downloadedBytes);
53         yield return null;
54     }
55     if (req.result == UnityWebRequest.Result.Success)
56     {
57         image.texture = (req.downloadHandler as
58         DownloadHandlerTexture).texture;
59         image.texture = DownloadHandlerTexture.GetContent(req);
60     }
```

```
60     else
61     {
62         print("获取失败: " + req.result + req.responseCode + req.error);
63     }
64 }
65
66 // 使用Post请求发送数据
67 IEnumerator PostData()
68 {
69     List<IMultipartFormSection> data = new List<IMultipartFormSection>();
70     data.Add(new MultipartFormDataSection("Name", "Hello World"));
71     data.Add(new MultipartFormFileSection("upload.png",
72     File.ReadAllBytes(Application.streamingAssetsPath + "/test.png")));
73     data.Add(new MultipartFormFileSection("Hello World", "text.txt"));
74
75     UnityWebRequest req =
76     UnityWebRequest.Post("http://10.0.71.145:8000/HttpServer", data);
77     req.SendWebRequest();
78
79     while(!req.isDone)
80     {
81         print(req.uploadProgress);
82         print(req.uploadedBytes);
83         yield return null;
84     }
85
86     if(req.result == UnityWebRequest.Result.Success)
87     {
88         print("上传成功");
89     }
90     else
91     {
92         print("上传失败: " + req.result + req.responseCode + req.error);
93     }
94 }
95
96 // 使用Put发送数据
97 IEnumerator PutData()
98 {
99     UnityWebRequest req =
100     UnityWebRequest.Put("http://10.0.71.145:8000/HttpServer",
101     File.ReadAllBytes(Application.streamingAssetsPath + "/test.png"));
102     yield return req.SendWebRequest();
103
104     if(req.result == UnityWebRequest.Result.Success)
105     {
106         print("上传成功");
107     }
```

```

103     }
104     else
105     {
106         print("上传失败: " + req.result + req.responseCode + req.error);
107     }
108 }

```

DownloadHandler类

- DownloadHandlerBuffer: 将下载的数据存储在内存中, 适用于简单的数据存储。

```

1 IEnumerator Start()
2 {
3     UnityWebRequest www = new UnityWebRequest("https://example.com");
4     www.downloadHandler = new DownloadHandlerBuffer();
5     yield return www.SendWebRequest();
6
7     if (www.result != UnityWebRequest.Result.Success)
8     {
9         Debug.Log(www.error);
10    }
11    else
12    {
13        Debug.Log(www.downloadHandler.text);
14    }
15 }

```

- DownloadHandlerFile: 将下载的文件直接保存到磁盘, 适用于大文件下载。

```

1 IEnumerator Start()
2 {
3     string url = "https://example.com/file.zip";
4     string path = Path.Combine(Application.persistentDataPath, "file.zip");
5     UnityWebRequest uwr = new UnityWebRequest(url);
6     uwr.downloadHandler = new DownloadHandlerFile(path);
7     yield return uwr.SendWebRequest();
8
9     if (uwr.result != UnityWebRequest.Result.Success)
10    {
11        Debug.LogError(uwr.error);
12    }
13    else
14    {

```

```
15     Debug.Log("File successfully downloaded and saved to " + path);
16 }
17 }
```

- DownloadHandlerTexture: 下载图像并将其存储为 Unity 的 Texture 对象。

```
1 IEnumerator Start()
2 {
3     string url = "https://example.com/image.png";
4     UnityWebRequest uwr = new UnityWebRequest(url);
5     uwr.downloadHandler = new DownloadHandlerTexture();
6     yield return uwr.SendWebRequest();
7
8     if (uwr.result != UnityWebRequest.Result.Success)
9     {
10         Debug.LogError(uwr.error);
11     }
12     else
13     {
14         Texture2D texture = DownloadHandlerTexture.GetContent(uwr);
15         GetComponent<Renderer>().material.mainTexture = texture;
16     }
17 }
```

- DownloadHandlerAssetBundle: 下载并处理 AssetBundle。

```
1 IEnumerator Start()
2 {
3     string url = "https://example.com/assetbundle";
4     UnityWebRequest uwr = new UnityWebRequest(url);
5     uwr.downloadHandler = new DownloadHandlerAssetBundle(url, 0);
6     yield return uwr.SendWebRequest();
7
8     if (uwr.result != UnityWebRequest.Result.Success)
9     {
10         Debug.LogError(uwr.error);
11     }
12     else
13     {
14         AssetBundle bundle = DownloadHandlerAssetBundle.GetContent(uwr);
15         // 使用 AssetBundle
16     }
17 }
```

- DownloadHandlerAudioClip: 下载音频数据并将其存储为 AudioClip 对象。

```
1 IEnumerator Start()
2 {
3     string url = "https://example.com/audio.mp3";
4     UnityWebRequest uwr = new UnityWebRequest(url);
5     uwr.downloadHandler = new DownloadHandlerAudioClip(url, AudioType.MPEG);
6     yield return uwr.SendWebRequest();
7
8     if (uwr.result != UnityWebRequest.Result.Success)
9     {
10         Debug.LogError(uwr.error);
11     }
12     else
13     {
14         AudioClip clip = DownloadHandlerAudioClip.GetContent(uwr);
15         // 使用 AudioClip
16     }
17 }
```

- DownloadHandlerScript: Unity 中的一个特殊类，用于处理自定义下载数据；它本身并不执行任何操作，但可以被用户定义类继承，以便在数据从网络到达时执行完全自定义的数据处理。

```
1 public class CustomDownloadHandler : DownloadHandlerScript
2 {
3     private byte[] data;
4     private int dataLength;
5
6     public CustomDownloadHandler(byte[] buffer) : base(buffer)
7     {
8         data = buffer;
9         dataLength = 0;
10    }
11
12    protected override bool ReceiveData(byte[] byteArray, int dataLength)
13    {
14        if (byteArray == null || byteArray.Length < 1)
15        {
16            Debug.LogError("CustomDownloadHandler :: ReceiveData - received a null/empty buffer");
17            return false;
18        }
19    }
```

```

20     System.Buffer.BlockCopy(byteArray, 0, data, this.dataLength,
dataLength);
21     this.dataLength += dataLength;
22     return true;
23 }
24
25 protected override void CompleteContent()
26 {
27     Debug.Log("Download complete! Data length: " + dataLength);
28 }
29
30 protected override float GetProgress()
31 {
32     return dataLength / (float)data.Length;
33 }
34 }
35
36 public class Example : MonoBehaviour
37 {
38     IEnumerator Start()
39     {
40         byte[] buffer = new byte[1024 * 1024]; // 1 MB buffer
41         UnityWebRequest www = new UnityWebRequest("https://example.com");
42         www.downloadHandler = new CustomDownloadHandler(buffer);
43         yield return www.SendWebRequest();
44
45         if (www.result != UnityWebRequest.Result.Success)
46         {
47             Debug.Log(www.error);
48         }
49         else
50         {
51             Debug.Log("Download successful!");
52         }
53     }
54 }

```

UploadHandler类

- UploadHandlerRaw: 用于上传字节数组

```

1  IEnumerator UploadBytes()
2  {
3      UnityWebRequest req = new
UnityWebRequest("http://10.0.71.145:8000/HttpServer/",

```

```

UnityWebRequest.kHttpVerbPOST);
4   byte[] bytes = Encoding.UTF8.GetBytes("Hello World");
5   req.uploadHandler = new UploadHandlerRaw(bytes);
6
7   yield return req.SendWebRequest();
8   print(req.result);
9 }

```

- UploadHandlerFile：用于上传文件

```

1 IEnumerator UploadFile()
2 {
3     UnityWebRequest req = new
    UnityWebRequest("http://10.0.71.145:8000/HttpServer/",
    UnityWebRequest.kHttpVerbPOST);
4     req.uploadHandler = new UploadHandlerFile(Application.streamingAssetsPath
    + "/test.png");
5
6     yield return req.SendWebRequest();
7     print(req.result);
8 }

```

协议生成工具

- 协议生成工具是一种根据通信协议规则自动生成消息定义的工具。它大大降低了维护协议的难度，使得一些协议（例如 Protobuf 和 Thrift）得以流行。
- 协议生成工具的主要作用包括：
 - 自动生成消息定义：根据预先设定的协议规则，自动生成消息的定义，减少手动编写的工作量。
 - 提高效率：通过自动化生成，减少了人为错误，提高了开发和维护的效率。
 - 减少故障：自动生成的消息定义更一致，减少了因手动编写导致的故障。
- 一个典型的 `.proto` 文件包含以下几个部分：
 - 语法声明：指定使用 proto3 语法。
 - 命名空间声明（可选）：避免命名冲突。
 - 导入其他 `.proto` 文件（可选）：使用其他文件定义的消息类型。
 - 消息类型定义：定义数据结构。
 - 枚举类型定义（可选）：定义一组命名的整数常量。
 - 服务定义（可选）：用于 RPC 服务。

```
1 syntax = "proto3";
2
3 package example;
4
5 import "google/protobuf/timestamp.proto";
6
7 message Person {
8     string name = 1;
9     int32 id = 2;
10    oneof contact {
11        string email = 3;
12        PhoneNumber phone = 4;
13    }
14
15    enum PhoneType {
16        MOBILE = 0;
17        HOME = 1;
18        WORK = 2;
19    }
20
21    message PhoneNumber {
22        string number = 1;
23        PhoneType type = 2;
24    }
25
26    map<string, string> attributes = 5;
27    google.protobuf.Timestamp create_time = 6;
28 }
```