**计算机网络**

学习完该课程，足够应付Unity客户端网络开发需求，如果作为***服务器程序员，还需学习更多的知识***。——***这可能意味着，我需要有后端开发的经验***。

网络的基本概念

**网络**

让设备之间实现互联。由若干设别和连接这些设备的链路构成，各个设备 间接或直接 通过介质相连。设备之间传递信息时候，将***传递的数据编码为2进制进行传输***。这些***数据是以电脉冲的形式传输***，线缆中的电压是在高低状态之间进行变化，所以***二进制的1通过产生一个正电压传输，二进制0通过产生负电压传输***。

局域网LAN

按照范围划分，是指***在某一个小区域由多台设别互联成的计算机组***。覆盖范围一般是方圆几千米内。

以太网

***一种计算机局域网技术，目前最普遍的局域网技术***。IEEE组织的***IEEE802.3标准制定了以太网技术标准***。它规定了包裹物理层的连线、电子信号和介质访问层协议的内容。

***网络连接的一种规则(协议)***。

以太网——网络拓扑结构

用传输媒介把计算机等各种设别相互连接起来的物理布局。

城域网MAN

在一个城市范围内所建立的网络，从几十公里到一百公里不等。***一个城域网由多个局域网组成***。

广域网WAN

又称 外网，公网。***是连接不同地区 局域网 或 城域网 设别通信的远程网***，范围几十公里到几千公里不等。能连接多个地区、城市和国家形成国际性的远程网络。

***广域网 不等于 互联网***。

互联网(internet)

***泛指彼此能够通信的设备组成的网络***，目前提到的互联网，大部分都是指因特网。

互联网将200多个国家和地区大部分设备连接起来，形成巨大的网络。***所以一定程度上，互联网等同于广域网，广域网包含了互联网。互联网上使用的技术，在广域网上一定有，但是某些广域网的技术，互联网上不一定有，比如军用的广域网***。

互联网的本质就是人为定义的一系列协议，总称为：***“互联网协议”***。主要功能：***定义计算机如何连接互联网，以及接入互联网的计算机的通信标准。也就是为我们的设备定义连入标准，并且为传输的2进制数据定义一些传输规则***。

因特网——Internet

国际最大的互联网，***它是全球最大的，开放的，有众多网络相互连接而成的特定的计算机网络。它采用TCP/IP协议簇作为通信的规则，提供了包裹万维网WWW, 文件传输FTP，电子邮件E-mail，远程登录Telnet等等服务***。

万维网WorldWideWeb

它是无数个网络站点和网页的集合，是构成因特网的主要部分。比如浏览器看到的内容就属于万维网。

***如果把因特网看做网络的基础，那么万维网就是因特网的应用，是利用因特网规则的一种信息传递和呈现的手段***。可以认为***万维网就是网站和页面的统称***。

IP、端口、Mac地址

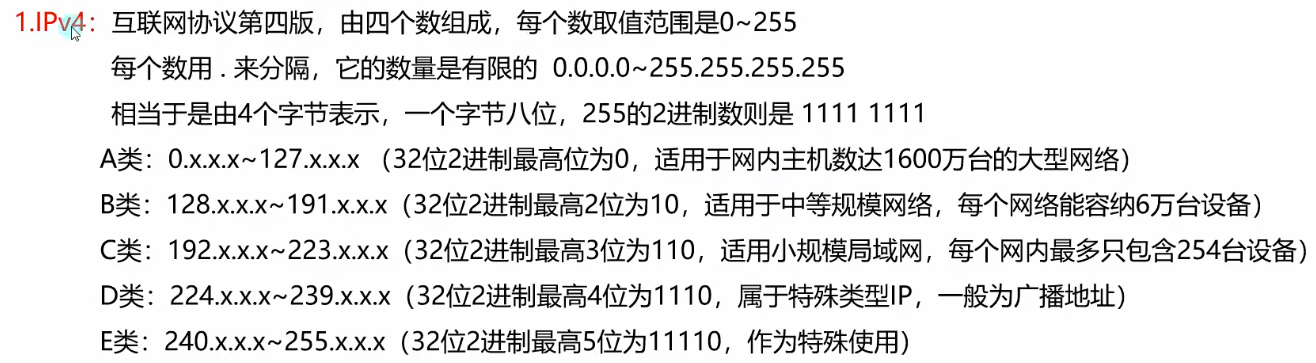
***解决问题——如何准确的将信息传递到目标设备***

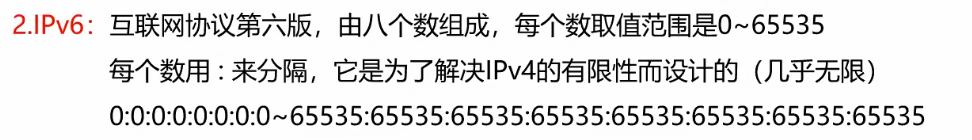
IP

IP地址是IP协议提供的一种统一的地址格式，***IP地址是设备在网络中的具体地址***。

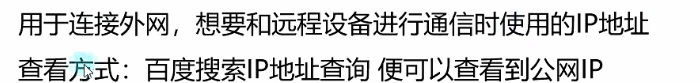
IP地址的分类

1. **IPv4**



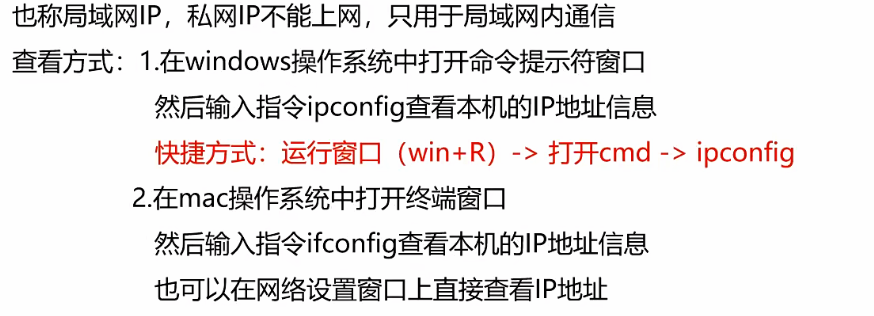
1. **IPv6**

**按照使用范围分类：**

1. **公网IP**

全球唯一的,比如一些知名网站baidu.com，任何人都可以访问。

1. **私网IP**

****是局域网内使用的，比如家里路由器分配的 192.168.1.xxx，仅家里设备之间能识别

端口号

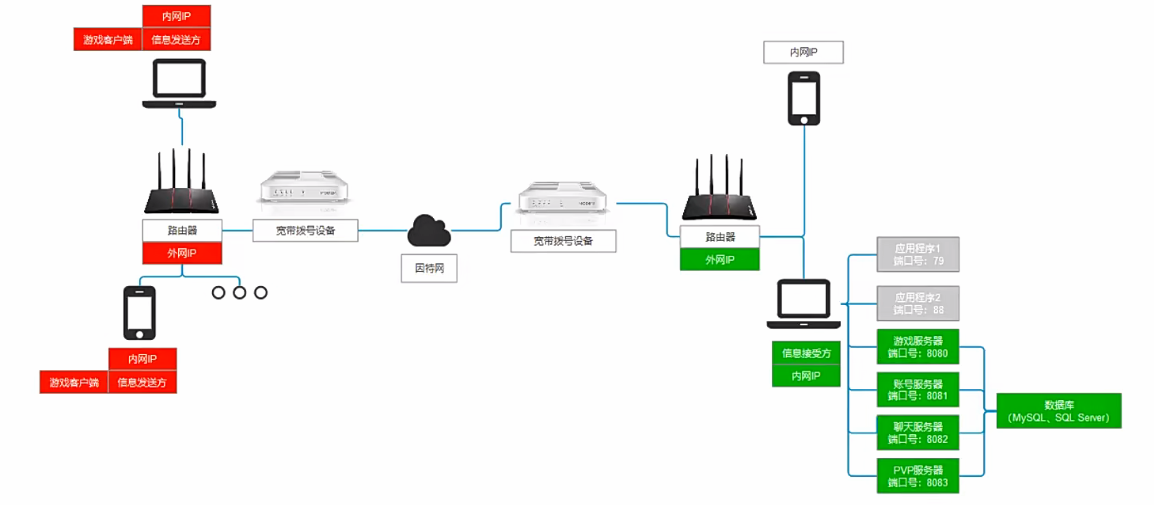
IP地址决定了设备在网络中的具体地址，而***端口是不同应用程序在该设备上的门牌号***，一台设备上不同的应用程序想要进行通信就必须对应一个唯一的端口号。

使用规则

端口号的取值范围是0-65535。

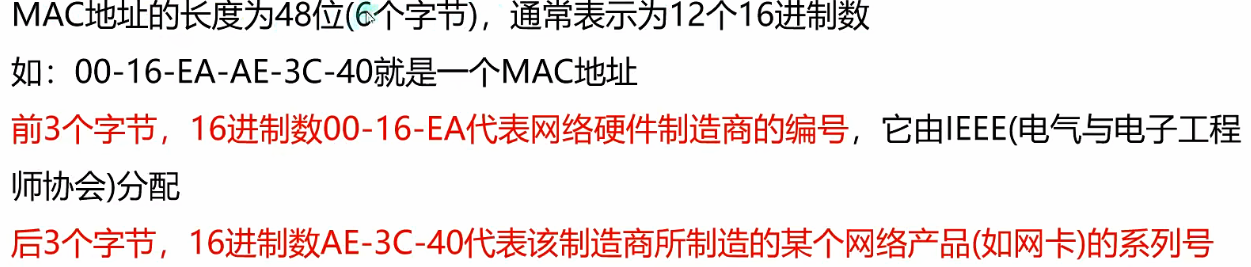
我们在进行网络程序开发时，需要自己为应用程序设置端口号。***端口号不能与其他应用程序相同，避免产生冲突***。一般选择1024以上的端口进行使用，1024以下的一般由IANA互联网数字分配机构管理。

通过 **IP : 端口号** 确定通信设备。

通信图

Mac地址

也称局域网地址，Mac地址，以太网地址。物理地址它用于确认网络设备的地址，在OSI模型中，第三层网络层负责IP地址，第二层数据链路层则负责Mac地址，***Mac地址是用于在网络中唯一标识一个网卡的，一台设备可以由多个网卡，每个网卡都会有一个唯一的Mac地址***。

Mac地址的构成

**Mac地址就像身份证，IP地址就像你的住址**——你一台设备的Mac地址是永远不会变的，但是当你连接不同地区的网络时，你的IP地址会发生变化。

通信流程

1. 你用手机（IP：192.168.1.5，MAC：00:1B:44:11:3A:B7）打开微信，发送消息给服务器（公网 IP：123.45.67.89）：
2. 你的消息会带上 “源 IP（192.168.1.5）+ 源端口（比如 56789，微信临时占用）+ 目标 IP（123.45.67.89）+ 目标端口（微信服务器的 8080 端口）”。
3. 消息先通过内网，路由器靠 MAC 地址识别你的手机，把消息转发到互联网。
4. 互联网上，数据靠目标 IP（123.45.67.89）找到微信服务器。
5. 服务器收到消息后，通过目标端口（8080）把数据交给微信服务程序，完成通信。

总结：

* IP：朋友的住址
* 端口：门牌号（具体应用程序）
* Mac地址：你朋友的身份证。

在网络通信中

通过***IP地址 以及 端口号 定位***想要通信的远端计算机中的某一个应用程序

* **IP地址 = 设备在网络中的位置——通常日常办公电脑的IP地址是内网地址**
* **端口 = 运行在该设备上的应用程序位置**
* **Mac地址 = 设备先进网络通信的唯一标识，设备真正进行物理信息传输用来定位的标识**。

**注意**：**内网**是小范围、私有可控的网络（如家庭/公司网络），**外网**是全球互联的公共网络（如互联网）

客户端和服务端

客户端

* **客户端Client**：用户使用的设备（计算机、手机、平板等等）
* **客户端应用程序**：用户使用的设别上安装的应用程序，用户会直接使用操作的内容。

**通常说的客户端会指客户端应用程序**。

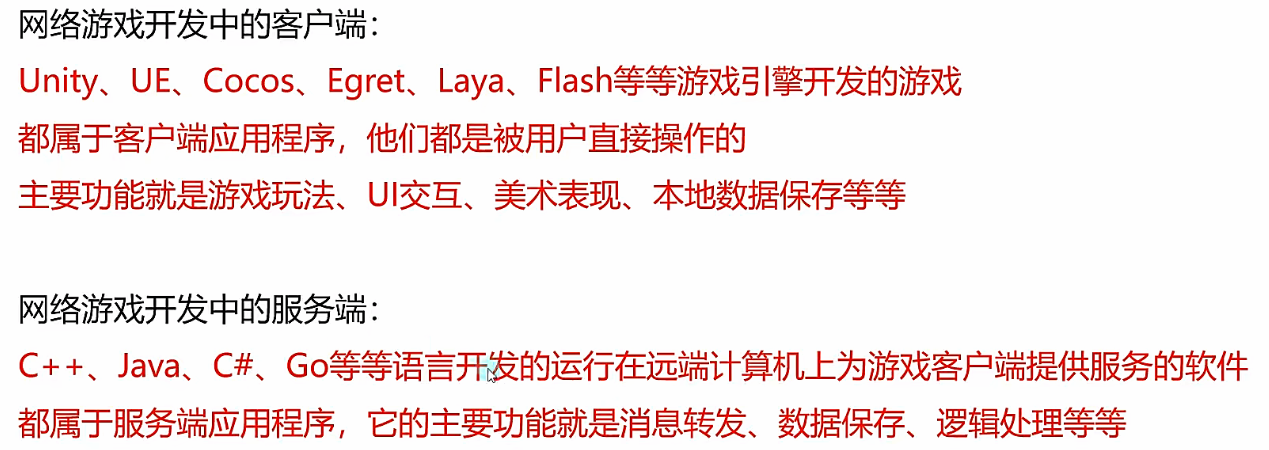
服务端

* **服务端Server**：为客户端提供服务的设备，一般是一台性能较好的计算机。
* **服务端应用程序**：为客户端提供服务的应用程序，该应用程序是运行在服务端设备上的。

**往往在软件开发中提到的服务端或服务器都是泛指服务端应用程序**。

***客户端通过网络和服务端进行通讯，服务端为客户端提供各种服务***。

网络游戏开发中的客户端和服务端

**静态(不变)数据存储在客户端，动态(要变)数据存储在服务端**。

***Unity程序员也被称为前端程序员，负责服务端开发的程序员一般被称为后端程序员程序员***。

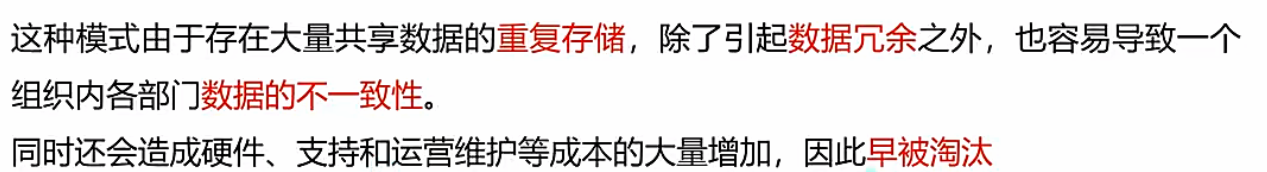
数据通信模型

**管理网络通信的方式**。

数据通信模型分类：

* 分散型
* 集中型
* 分布型

分散式

在分散式系统中，用户只负责管理自己的计算机系统，各自独立的系统之间没有资源或信息的交换和共享。就类似一台没有联网的设备。

集中式

在集中式环境中，用一台主计算机保存一个组织的全部数据，而用户通过设别连接到这台计算机系统并和它通信，从而达到访问数据的目的。

* 优点：方便数据共享，消除了数据的冗余和不一致性
* 缺点：可靠性不如分散式，主机故障所有系统全部瘫痪。

分布式

***分布式是分散式和集中式的混合***，是分散式的水平交互和集中式的垂直控制相结合的一种模式。

它兼具了分散式和集中式的优点：方便数据共享，消除了数据的冗余和不一致性，同样也加强了容错性。

**举个例子：**

比如一般的企业管理系统就是一个典型的集中式和分布式的综合体现。

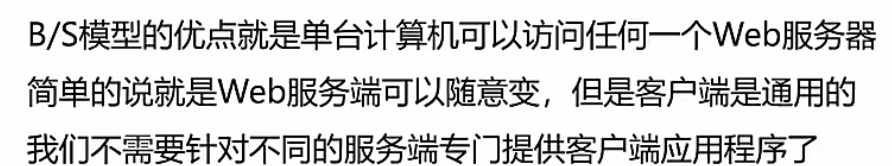
***所有数据用专用的数据库集中存储，属于集中式；而对数据的处理则有各个部门的软件分别控制，属于分布式***。

C/S 模型

C/S（Client/Server）模型也叫C/S模式，它是***目前大多数网络通信采用的模型***。

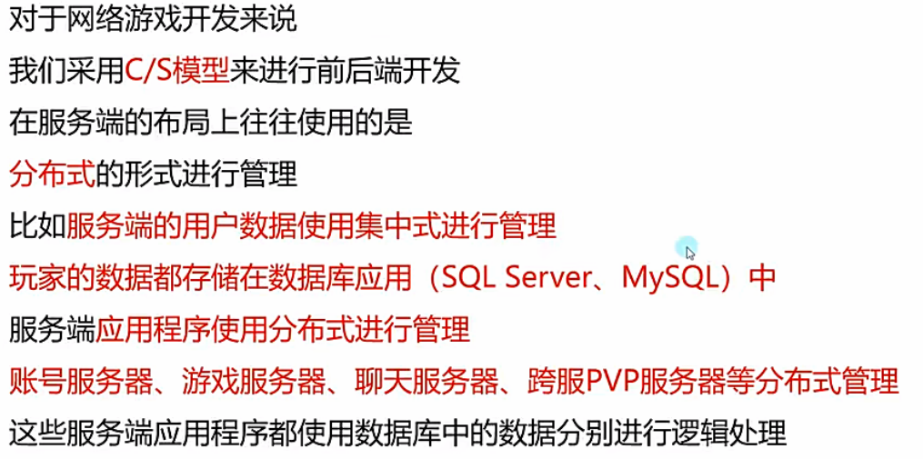
从程序的角度来说，客户端和服务器打交道，相当于是两个应用程序（进程）在打交道。

B/S模型（Browse/Server）

**一种基于Web的通信模型，使用HTTP超文本传输协议通信**。***B/S是一种特殊的C/S模型***，特殊之处就是这种模型的客户端不需要我们自己开发，一般是某种浏览器，比如IE，Chrome，QQ浏览器，360浏览器等等。

P2P模型(Peer to Peer)

每个联网的设备同时运行一个应用程序的客户端和服务端部分，也就是说一个应用程序即是客户端也是服务端。

总结

关于服务端部分，Unity前端程序员只做简单的了解即可，我们只需要学习：

1. 如何和服务器建立网络连接
2. 如何和服务端交换信息

***或许等我就业后，我才会开始学习后端，不过我真的很喜欢专精游戏开发——那种就像画出名画一样的艺术行为***

网络协议概述

计算机网络中进行数据交换而建立的规则，标准或约定的集合。指的是***计算机网络中相互通信的对等实体之间交换信息时所必须遵守的规则的集合***。

就是在 ***网络环境中进行通信需要遵守的规则***。

主要内容有：

* **OSI模型**
* **TCP/IP协议**

**OSI模型是国际组织定义的一套理论规则，主要用于定义网络通信的规则**。

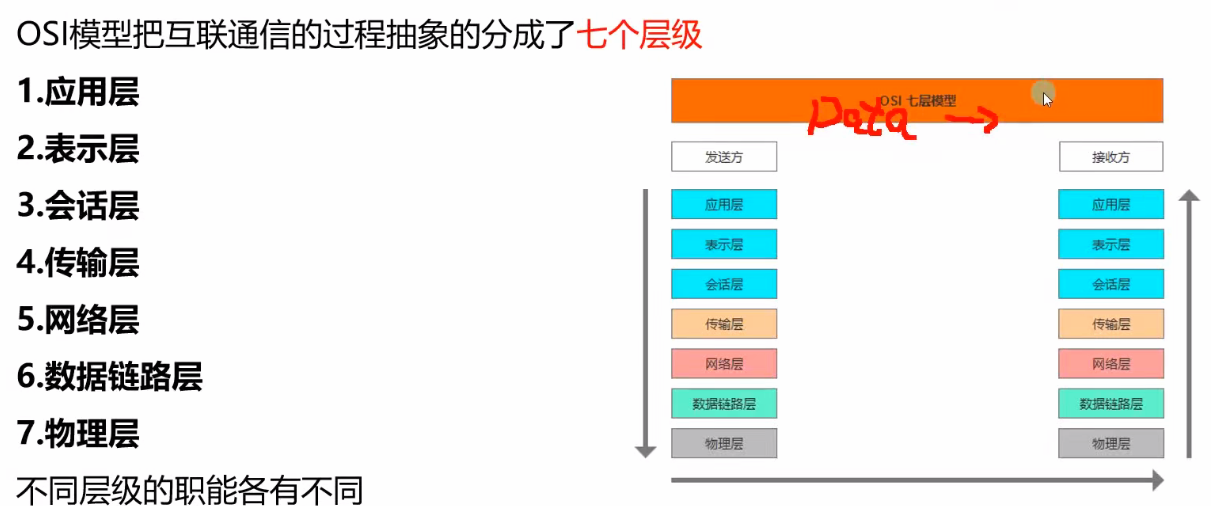
**TCP/IP 协议是基于OSI模型理论基础真正实现的通信规则——这也是课程学习的重点**。

接下来的学习内容，主要是：***2进制数据如何加工，如何准确的传递到目标设别当中，在数据传递过程中具体的规则标准是什么。而这些规则标准都是由网络协议指定的***。

OSI模型

开放式系统互联通信参考模型。

OSI模型的规则

OSI模型将复杂的互联网实现分成了好几个部分(层)。每一层都有自己的功能，每一层都要靠下一层的支持。

OSI模型每层的职责

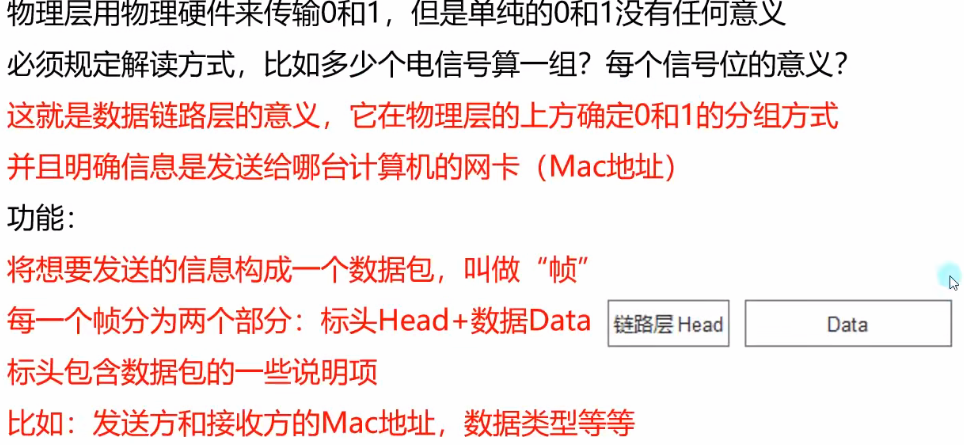
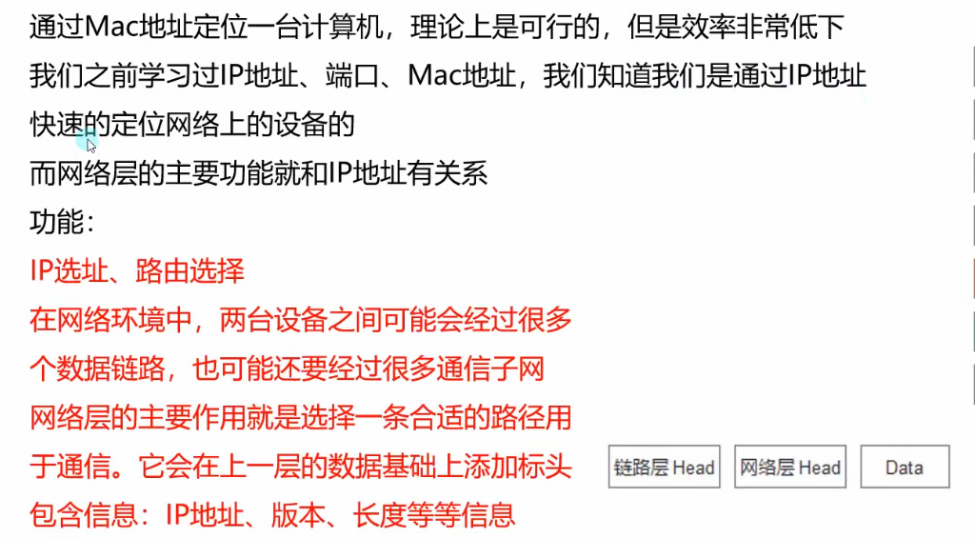
OSI模型分成两个部分：

* 第一部分：物理层、数据链路层、网络层、传输层（从下往上讲）
* 第二部分：应用层、表示层、会话层（从上往下讲）

物理层

把电脑连接起来的物理手段，主要规定了网络的一些电气特性，作用是负责传送0和1的电信号。***物理层将2进制数据利用电脉冲在物理媒介上实现比特流的传输***。

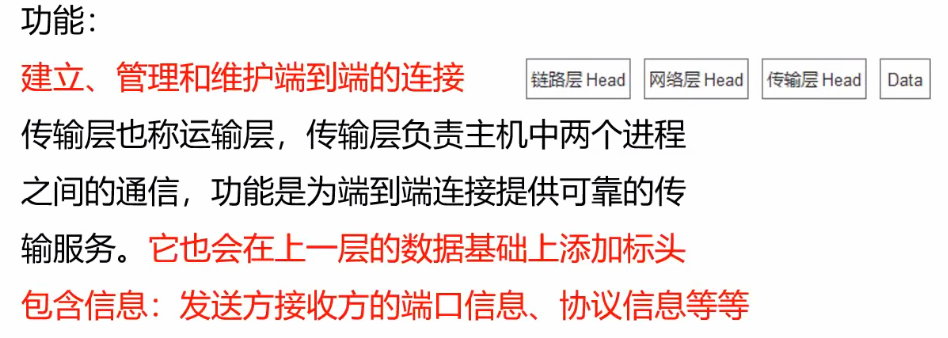
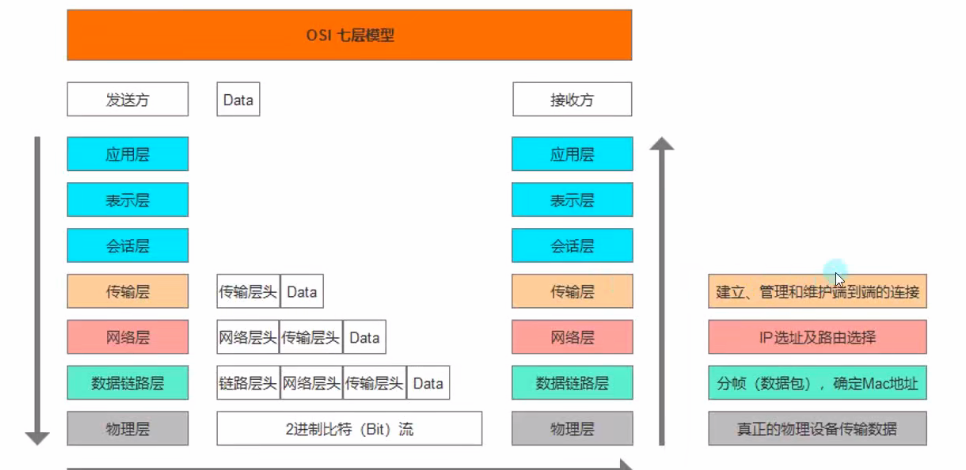
它的主要功能：定义传输模式、定义传输速率、比特同步、比特编码等。

数据链路层

网络层

传输层

有了Mac地址和IP地址，已经可以为两台计算机建立通通信了，但是没有端口号，就无法确定在哪个有用程序之间进行通信。传输层的功能，就是建立端口到端口的通信。

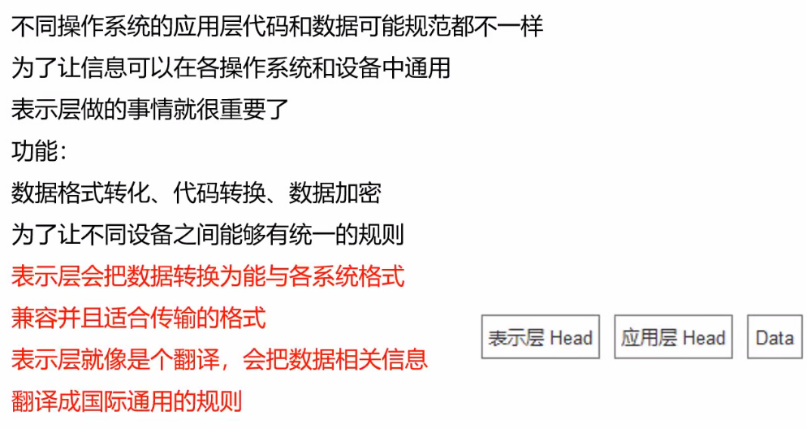
第一部分的总结

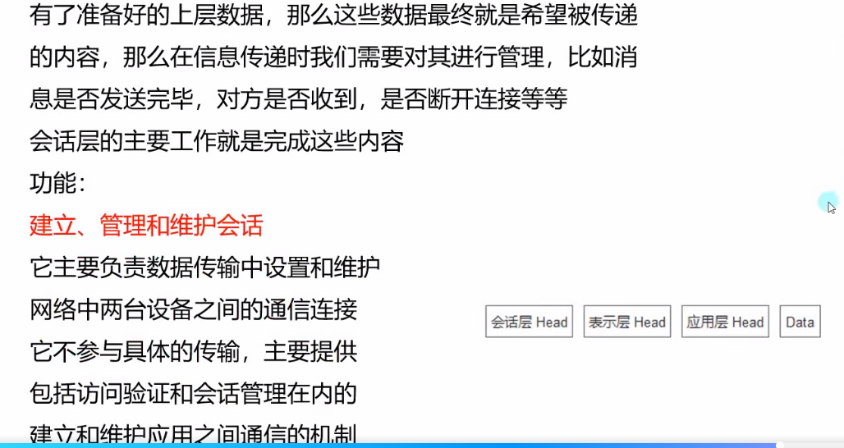
应用层

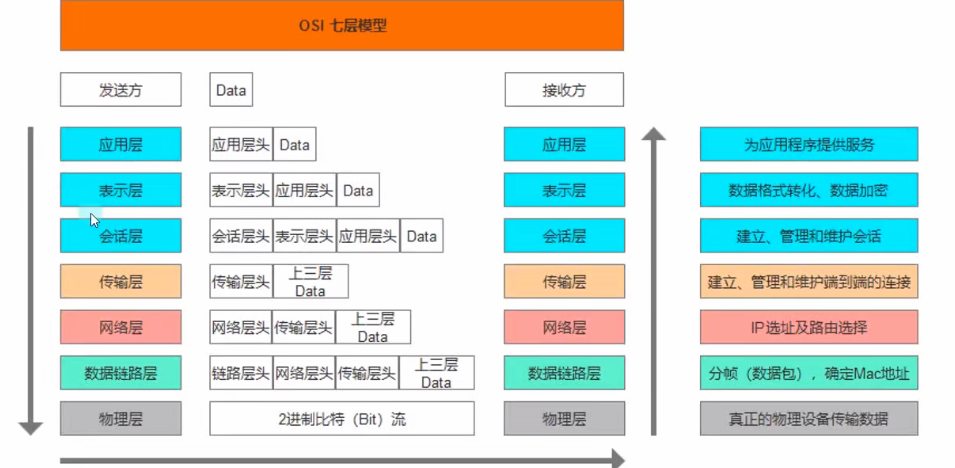
和用户直接打交道的层级，比如浏览器，游戏等等。

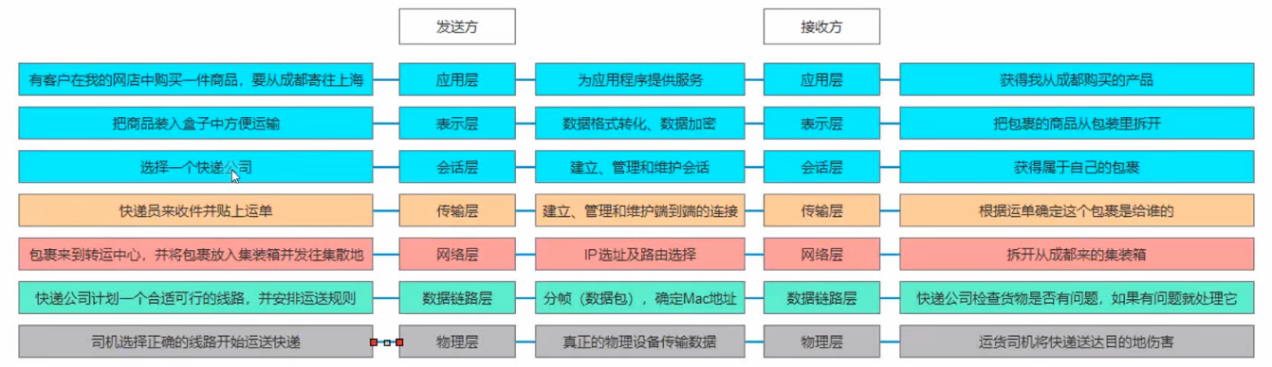
功能：为应用程序提供服务，根据要传递的信息决定使用哪种协议来处理数据来进行传输。**我们程序员主要开发的也是这一层，它会在原始的数据的基础上添加标头，包含信息：协议信息等等**。

常见的协议有：FTP、HTTP、SMTP等。

表示层

会话层

第二部分总结

OSI每层的职责

总结：

OSI模型主要还是理论，学习它能帮助我们理解网络通信的流程，日常开发也接触不到几层。

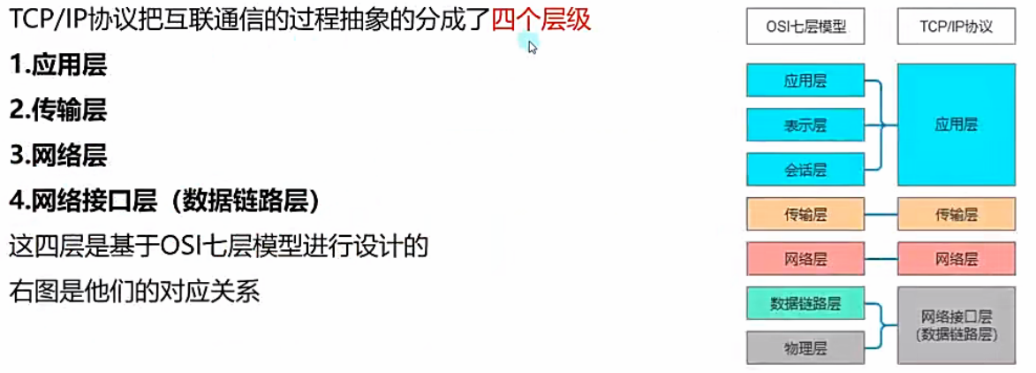
TCP/IP协议

TCP/IP协议不仅仅指的是TCP和IP协议，而是指一个由FTP, SMTP, TCP,UDP,IP等等协议构成的协议簇，只是因为TCP.IP协议最具代表性。

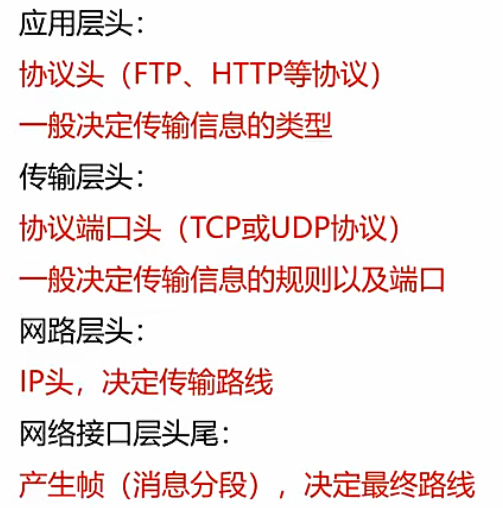
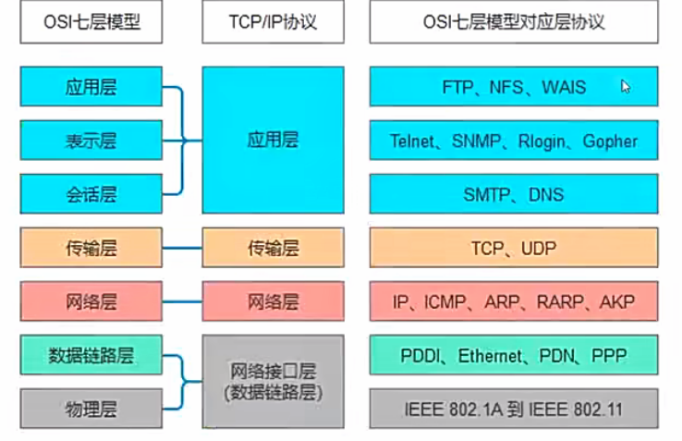
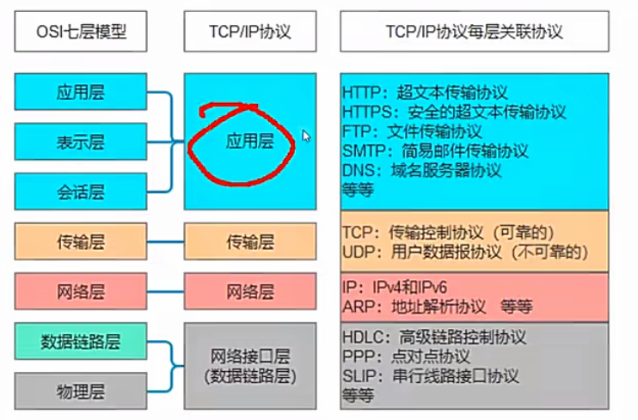
它是用于互联网的通信协议，是一系列规则的统称，他们定义了消息在网络间进行传输的规则。

TCP/IP协议的规则

TCP/IP网络结构体实际上是基于OSI七层模型设计出来的。OSI七层模型只是一个概念模型，它主要用于描述，讨论和理解单个网络功能。而TCP/IP协议是为了解决一组特定的问题而设计的，它是基于互联网开发的标准协议(规则)。

****** ***OSI模型只是一个基本概念，而Tcp/IP协议是基于这个概念的具体实现。***

***在进行网络开发时，我们就是基于TCP/IP协议来进行网络通信的。***

TCP/IP协议每层职能

* ***应用层的相关协议会在实际开发中去学习***
* ***传输层的TCP和UDP对于我们来说是非常重要的两个规则。***

TCP和UDP

Tcp/IP协议中的重要协议

应用层协议：

* HTTP协议： 超文本传输协议
* HTTPS协议：加密的超文本传输协议
* FTP协议：文件传输协议
* DNS: 域名系统

传输层协议：

* ***TCP协议：传输控制协议***
* ***UDP协议：用户数据报协议***

网络层协议：

* IP协议



TCP与UDP的区别

TCP协议

面向连接的协议，在收发数据前，必须和对方建立可靠的连接。在消息传送过程中是有顺序的，并且是不会丢包的。如果某一条消息在传送过程中失败了，会重新发送消息，直到成功。

特点：

* **面向连接**：两者必须建立可靠连接
* **一对一**：只能1对1的建立连接
* **可靠性高**：消息传送失败会重新发送，不运行丢包
* **有序的**: 是按照顺序进行消息发送的

TCP协议的重要概念： ***三次握手，四次挥手***

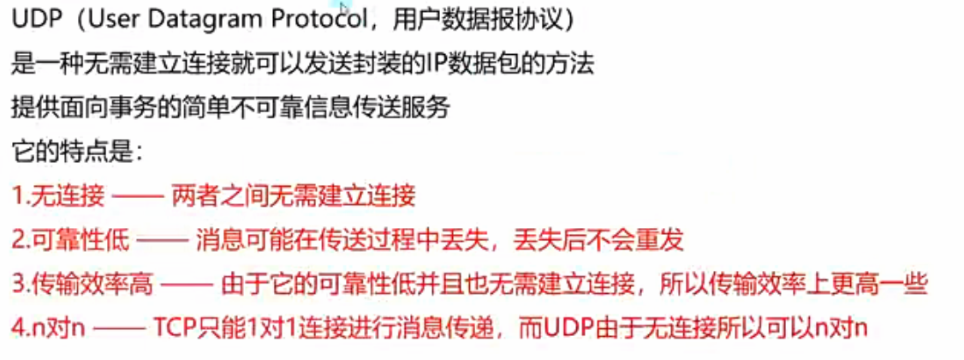
**三次握手**

1. 第一次握手(C->S)：TCP连接请求，告诉服务器我要和你建立连接
2. 第二次握手(S->C)：TCP授予连接，告诉客户端可以连接
3. 第三次握手(C->S)：TCP确认连接，告诉服务器

**四次挥手**

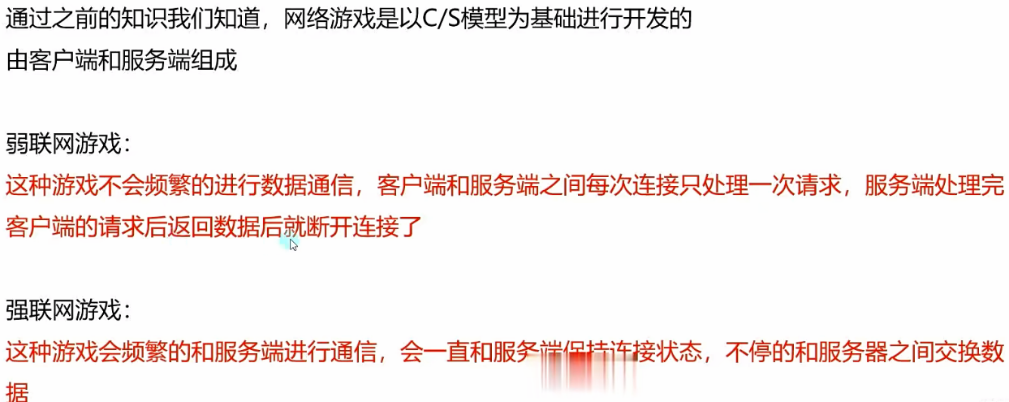
1. 第一次挥手(C -> S): 告诉服务器数据发完了，如果还有消息就快发。
2. 第二次挥手(S ->C): 告诉客户端我知道，请继续等待我的消息。
3. 第三次挥手(S -> C)：告诉客户端消息发完了，你可以断开连接了。
4. 第四次挥手(C->S)：告诉服务器等我一会，如果没有收到你回复我就断开了。

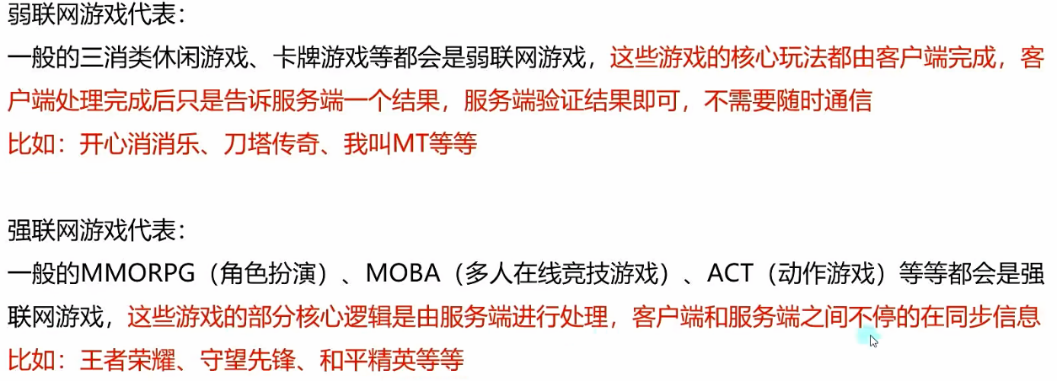
TCP有了这三次握手，四次挥手的规则，可以提供可靠的服务，通过TCP连接传送的数据，可以做到无差错，不丢失，不重复，且按顺序到达。

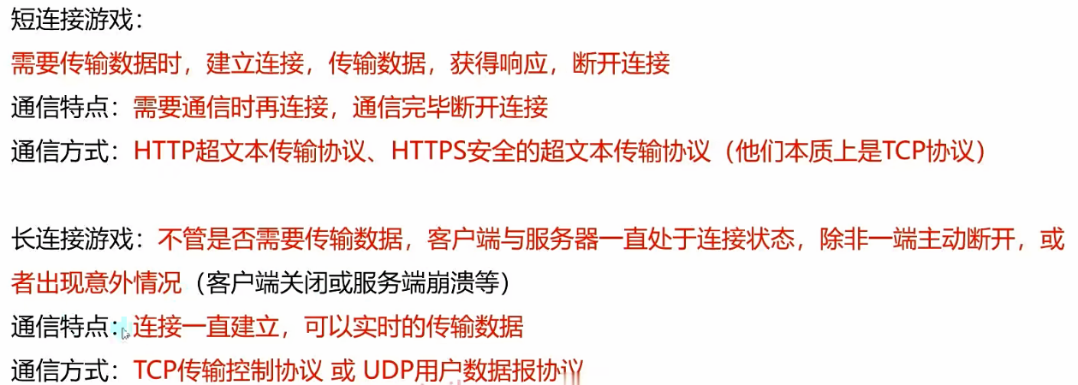
UDP协议

它具有资源消耗小，处理速度快的特点。

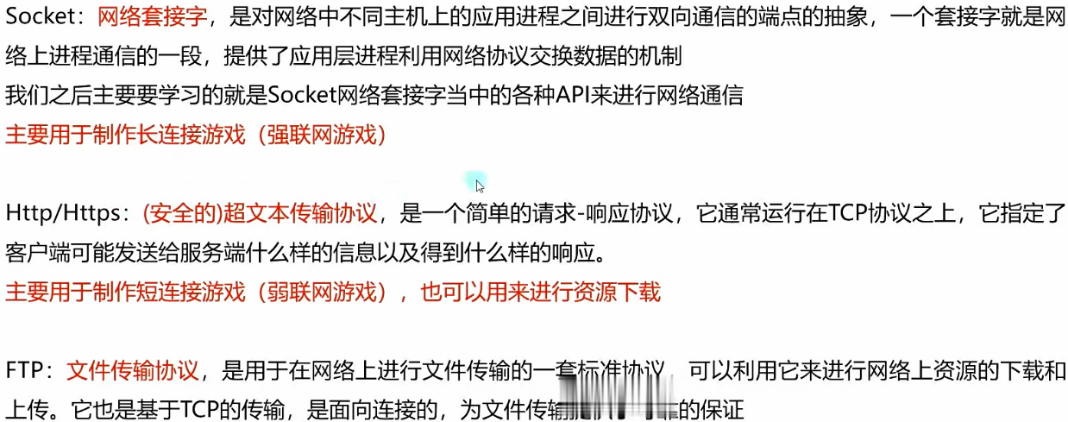
游戏通信方案概述

弱联网和强联网游戏



长连接和短连接游戏

Socket、HTTP 、FTP

总结：

网络游戏的通信方案大体上可以根据游戏的实际情况情况分为两种 ***长连接(强联网)游戏***和**短连接(弱联网)游戏**。

我们的主要学习方法：

* **Socket网络嵌套字** 主要用来完成 长连接网络游戏需求。
* **Http超文本传输协议** 主要用来完成，短连接网络游戏需求(或者资源下载相关)。
* **FTP文件传输协议**：主要用来完成 资源的下载和上传等需求。

IP地址和端口号

网络中通过IP和端口号确认网络设备。C#提供了对应的IP和端口相关的类，来声明对应的信息。

IPAddress类

命名空间: System.Net

类名: IPAddress

**用btye数组进行初始化**

|  |
| --- |
| byte[] ipAddress = new byte[] {118, 102, 111, 11};  IPAddress ip1 = new IPAddress(ipAddress); |

**用long长整型进行初始化(不推荐，4字节对应的长整型，可以使用16进制存储)**

|  |
| --- |
| IPAddress ip2 = new IPAddress(0x79666F0B); //与上面的IP地址等价 |

**推荐使用的方式 使用字符串转换**

|  |
| --- |
| IPAddress ip3 = IPAddress.Parse(“118,102,111,11”); |

特殊IP地址

127.0.0.1 代表本机IP

一些静态成员，获取可用的IPv6地址,自动帮你获取可以使用的IPv6地址

|  |
| --- |
| IPAddress.IPv6Any |

IPEndPoint类

命名空间： System.Net

类名: IPEndPoint

IPEndPoint类将网络端点表示为IP地址和端口号，表现为IP地址和端口号的组合

**初始化**

|  |
| --- |
| IPEndPoint ipPoint = new IPEndPoint(0x79666F0B, 8080);  //第二种方式  IPEndPoint ipPoint2 = new IPEndPoint(IPAddress.Parse(“118.102.111.11”), 8080); |

* 第二个参数传递的是一个IPAddress对象

域名解析

将域名解析为IP，比如www.baidu.com就是一个域名。域名的解析工作由DNS服务器来完成。同时C#提供了域名解析相关的类。

IPHostEntry类

命名空间System.Net

类名：IPHostEntry

主要作用：域名解析后的返回值，可以通过该对象获取IP地址，主机名等等信息。

***一般不会主动声明该类，都是作为某些方法的返回值返回信息，我们主要通过该类对象获取返回值***。

IPHostEntry host = new IPHostEntry(); **//这样做没有意义**

* 获取关联IP 成员变量AddressList
* 获取主机别名列表 成员变量 Aliases
* 获取DNS名称 成员变量 HostName

Dns类

命名空间System.Net

类名：Dns

主要作用：Dns是一个静态类，提供了很多静态方法，可以使用它来根据域名获取IP地址。

1. 获取本地系统的主机名

print(Dns.GetHostName()); **//默认是计算机的名称**

1. 获取指定域名的IP信息

根据域名获取

2.1同步获取

注意：由于获取远程主机信息是需要进行网络通信的，所以可能会阻塞主线程

|  |
| --- |
| IPHostEntry entry = Dns.GetHostEntry(“www.baidu.com”);  for(int i = 0 ; i <entry.AddressList.Length,;i++ ){  print(“IP地址” + entry.AddressList[i]);  }  for(int i = 0 ; i <entry.Aliases.Length,;i++ ){ //别名不是每个域名都有，有的域名没有  print(“主机别名：” + entry.Aliases[i]);  }  print(“DNS名称” + entry.HostName); |

2.2异步获取

|  |
| --- |
| GetHostEntry()  **//调用异步方法**  private async void GetHostEntry()  {  Task<IPHostEntry> task = Dns.GetHostEntryAsync(“www.baidu.com”);  await task;  for(int i = 0 ; i <task.Result.AddressList.Length,;i++ ){  print(“IP地址” + task.Result.AddressList[i]);  }  for(int i = 0 ; i <task.Result.Aliases.Length,;i++ ){ //别名不是每个域名都有，有的域名没有  print(“主机别名：” + task.Result.Aliases[i]);  }  print(“DNS名称” + task.Result.HostName);  } |

序列化和反序列化2进制数据

网络通信的目的就是传输数据，在网络通信过程中，一般会将数据转为2进制数据发送。

网络通信中，我们把***想要传递的类对象信息序列化2进制数据(一般为byte字节数组)***，再将二进制数据通过网络传输发给远程设备，***远程设备获取到2进制数据后再讲其反序列化为对应的类对象***。

和网络通信相关的重要知识点(会在网络通信中频繁使用)：

1. ***BitConverter类 ： 主要用于处理各类型的字节数组间的相互转换***
2. ***Encoding类： 主要用于处理字符串类型和字节数组间的相互转换。***
3. ***加密相关: 了解2进制数据加密的常用手段和思路。***

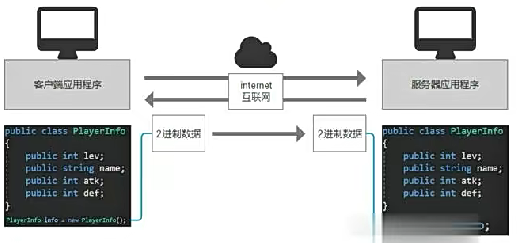
本地持久化知识点

1. File类： 文件操作类，用于操作文件
2. FileStream类：文件流类，以流的形式进行文件存储读取操作。
3. MemoryStream： 内存流对象
4. ***BinaryFormatter: 2进制格式化对象——不会在网络开发中使用***

BinaryFormatter类可以快速的将C#对象转化为字节数组数据，但是在网络开发中，我们不会使用。***因为客户端和服务端的开发语言大多数情况下是不同的，BinaryFormatter类序列化的数据无法兼容其他语言***。

网络通信中传输的数据

在网络通信中***，我们想要传递的类对象信息序列化为2进制数据***(一般为byte数组)，再将该2进制数据通过网络传输给远端设备

 远端设备获取到该2进制数据后，再将其反序列为对应的类对象。

和网络通信相关的重要知识点

***BitConverter类： 主要用于处理各类型的字节数组间的相互转换***

***Encoding类：主要用于处理字符串类型和字节数组间的相互转换。***

***加密相关：了解2进制数据加密的常用手段和思路***

本地持久化知识点：

File类：文件操作类，用于操作文件

FileStream类， 文件流类，以流的形式进行文件存储读取操作。

MemoryStream 内存流对象

BinaryFormatter: 2进制格式化对象。

***注意***：

2进制知识点有BinaryFormatter类，它可以快速的将C#类对象转换为字节数组数据，但是在网络开发中，***我们不会使用它***，因为客户端和服务端的开发语言大多数情况是不同的。***BinaryFormatter类序列化的数据无法兼容其他语言***。

序列化

非字符串类型转字节数组

关键类: BitConverter

所在命名空间: System

主要作用： 除了字符串的其他常用类型的字节数组相互转换。

|  |
| --- |
| byte[] byte1 = BitConverter.GetBytes(1);  byte[] byte2 = BitConverter.GetBytes(3.14f);  byte[] byte3 = BitConverter.GetBytes(true);  byte[] byte4 = BitConverter.GetBytes(‘a’); //这里是char类型 |

字符串转字节数组

关键类：Encoding

所在命名空间: System.Text

主要作用：将字符串类型和字节数组相互转换，并且决定转换时使用的字符编码类型，网路通信建议使用UTF-8类型。

byte[] byte2 = Encoding.UTF8.GetBytes(“我是王宝强，大的来了！”);

将类对象转换为2进制数据

注意：不能使用BinaryFormatter， 原因我说过。

单纯的转换一个变量为字节数组非常简单

但是我们如何将一个类对象携带的所有信息放入到一个字节数组中呢？

1. 明确字节数组的容量

2. 声明一个装载信息的字节数组容器

3. 将对象中的所有信息转为字节数组并放入该容器中(可以利用数组中的CopyTo方法转存字节数组)

CopyTo方法的第二个参数代表 从容器的第几个位置开始存储。

|  |
| --- |
| public class PlayerInfo  {  public int lev;  public string name;  public short atk;  public bool sex;  }  PlayerInfo info = new PlayerInfo()  info.lev = 10;  info.name = “杨”;  info.atk = 88;  info.sex = false;  //第一步  int indexNum = sizeof(int) + //lev int类型 4  sizeof(int) + //代表name字符串转换成字节数组后，数组的长度  Encoding.UTF8.GetBytes(info.name).Length + //字符串具体字节数组的长度  sizeof(short) + //atk short类型 2  sizeof(bool); //sex bool 类型1  //第二步  byte[] playerBytes = new byte[indexNum];  //第三步  int index = 0;  //等级  BitConverter.GetBytes(info.lev).CopyTo(playerBytes.index);  index += sizeof(int);  //姓名  byte[] strBytes = Encoding.UTF8.GetBytes(info.name);  int num = strBytes.Length;  BitConverter.GetBytes(num).CopyTo(playerBytes, index);  index += sizeof(int);  strBytes.CopyTo(playerBytes, index);  index += num；  //攻击力  BitConverter.GetBytes(info.atk).CopyTo(playerBytes, index);  index += sizeof(short);  //性别  BitConverter.GetBytes(info.sex).CopyTo(playerBytes, index);  index += sizeof(bool); |

优化：

我们按照类型，使不同的字段使用不同方法得到字节数组。

我们可以将该方法，行为封装到对应类里面，变为一种方法，这样得到对象，他就会自己封装自己的字段从而得到存储的字节数组。

|  |
| --- |
| public class PlayerInfo  {  public int lev;  public string name;  public short atk;  public bool sex;    public byte[] GetBytes(){执行逻辑…}  } |

反序列化

字节数组转非字符串类型

关键类: BitConverter

所在命名空间: System

主要作用：除字符串的其他常用类型和字节数组相互转换

|  |
| --- |
| byte[] bytes = BitConverter.GetBytes(99);  int i = BitConverter.ToInt32(bytes, 0);  print(i); |

字节数组转字符串类型

关键类：Encoding

所在命名空间: System.Text

主要作用: 将字符串类型和字节数组想换转换，并且决定转换时使用的字符编码类型，网络通信时建议使用UTF-8编码

|  |
| --- |
| byte[] bytes2 = Encoding.UTF8.GetBytes(“1312ddada1哈哈哈”);  string str = Encoding.UTF8.GetString(bytes2, 0, bytes2.Length);  print(str); |

如何将二进制数据转为一个类对象

* + - 1. 获取到对应的字节数组

PlayerInfo info = new PlayerInfo(); //这里使用序列化案例中的代码

info.lev = 10;

info.name = “杨”;

info.atk = 88;

info.sex = false;

byte[] playerBytes = info.GetBytes();

* + - 1. 将字节数组按照序列化时的顺序进行反序列化

|  |
| --- |
| PlayerInfo info2 = new PlayerInfo();  int index = 0;  //等级  info2.lev = BitConverter.ToInt32(PlayerBytes, index);  index += 4;  //姓名  int length = BitConverter.ToInt32(playerBytes, index);  index += 4;  info2.name = Encoding.UTF8.GetString(playerBytes, index, length);  index += length;  //攻击力  info2.atk = BitConverter.ToInt16(playerBytes, index);  index += 2；  //性别0  info2.sex = BitConverter.ToBoolean(playerBytes, index);  index += 1; |

Socket的重要API

Socket嵌套字的作用

C#提供给我们用于网络通信的一个类

类名: Socket

命名空间: System.Net.Sockets

Socke嵌套字是支持TCP/IP网络通信的基本操作单位

一个嵌套字包含以下关键信息

本机的IP地址和端口

对方主机的IP地址和端口

双方通信的协议

一个Socket对象表示一个本地或者远程套接字信息，它可以被视为一个数据通道，这个通道连接客户端与服务端，数据的发送和接受均通过这个通道进行。

一般在制作长连接游戏时，会使用Socket作为网络通信方案。

Socket类型

Socket套接字有3种不同类型

* + - 1. 流套接字

主要用于实现TCP通信，提供了面向连接，可靠的，有序的，数据无差错且无重复的数据传输服务。

* + - 1. 数据报套接字

主要用于实现UDP通信，提供了无连接的通信服务，数据包的长度不能大于32KB，不提供正确性检查，不保证顺序，可能出现重发，丢失等情况。

* + - 1. 原始套接字(不常用，不深入讲解)

主要用于实现IP数据包通信，用于直接访问协议的较低层，常用于侦听和分析数据包。

通过Socket的构造函数，我们可以声明不同类型的套接字

Socket s = new Socket();

**参数一：AddressFamily 网络寻址 枚举类型，决定寻址方案**

常用：

* InterNetWork IPv4寻址
* InterNetWork IPv6寻址

做了解：

* UNIX UNIX本地到主机地址
* ImpLink ARPANETIMP地址
* Ipx IPX或SPX地址
* Iso ISO协议的地址
* Osi OSI协议的地址
* NetBios NetBios地址
* Atm 本机ATM服务地址
* 还有很多等等

**参数二：SocketType 套接字枚举类型，决定使用的套接字类型**

常用：

* Dgram 支持数据报，最大长度固定的无连接，不可靠的消息(主要用于UDP通信)
* Stream 支持可靠，双向，基于连接的字节流(主要用于TCP通信)

做了解:

* Raw 支持对基础传输协议的访问
* Rdm 支持无连接，面向消息，以可靠方式发送的消息
* Seqpacket 提供排序字节流的面向连接且可靠的双向传输

**参数三： ProtocelType 协议类型枚举类型，决定套接字使用的通信协议**

常用:

* Tcp TCP传输控制协议
* Udp UDP用户数据报协议

做了解：

* IP IP网络协议
* Icmp Icmp网际消息控制协议
* Igmp Igmp网际组管理协议
* Ggp 网关到网关协议
* IPv4 Internet协议版本4
* Pup PARC通用数据包协议
* Idp Internet数据包交换协议
* Raw 原始IP数据包协议
* Ipx Internet数据包交换协议
* Spx 顺序包交换协议
* IcmpV6 用于IPv6的Internet控制消息协议

**2,3参数的常用搭配**

* SocketType.Dgram + ProtocolType.Udp = UDP协议通信(主要学习)
* SocketType.Stream + ProtocolType.Tcp = TCP协议通信(主要学习)
* SocketType.Raw + ProtocolType.Icmp = Internet控制报文协议(了解)
* SocketType.Raw + ProtocolType.Raw = 简单的IP包通信(了解)

**TCP套接字**

Socket sokcetTcp = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

**UDP套接字**

Socket sokcetTcp = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);

Socket的常用属性（目前是大概的认识）

套接字的连接状态

if(socketTcp.Connected) {…}

获取套接字类型

Debug.Log(socketTcp.SocketType);

获取套接字的协议类型

Debug.Log(socketTcp.ProtocolType);

获取套接字的寻址方案

Debug.Log(socketTcp.AddressFamily);

从网络中获取准备读取的数据数据量

Debug.Log(socket.Available); //将会接受多少个字节（Int）

获取本机EndPoint对象(注意: IPEndPoint继承EndPoint)

socketTcp.LocalEndPoint as IPEndPoint; //这样就可以获取IP地址和端口号

获取远程EndPoint对象

socketTcp.RemoteEndPoint as IPEndPoint;

Socket的常用方法(了解)

用于服务端

绑定IP和端口

IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(“127.0.0.1”), 8080);

socketTcp.Bind(ipPoint); //将主机IP作为服务器

设置客户端连接的最大数量

socketTcp.Listen(10);

等待客户端接入

socketTcp.Accept(); //这些方法具有同步和异步方法

用于客户端

连接远程服务端

socketTcp.Connect(IPAddress.Parse(“128.12.123.11”), 8080); //传入的远程服务器IP

用于客户端和服务端

同步发送和接受数据

socketTcp.Send(); //TCP使用

socketTcp.SenTo(); //UDP使用

socketTcp.Recrive(); //接受数据

//进入类中去了解，就会看到同步和异步方法，这里只做了解，后面会慢慢用到

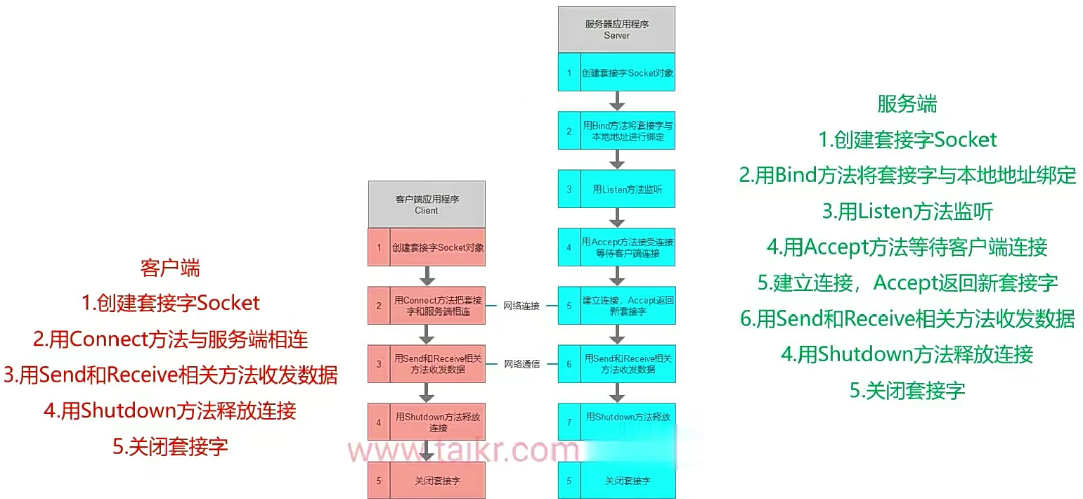
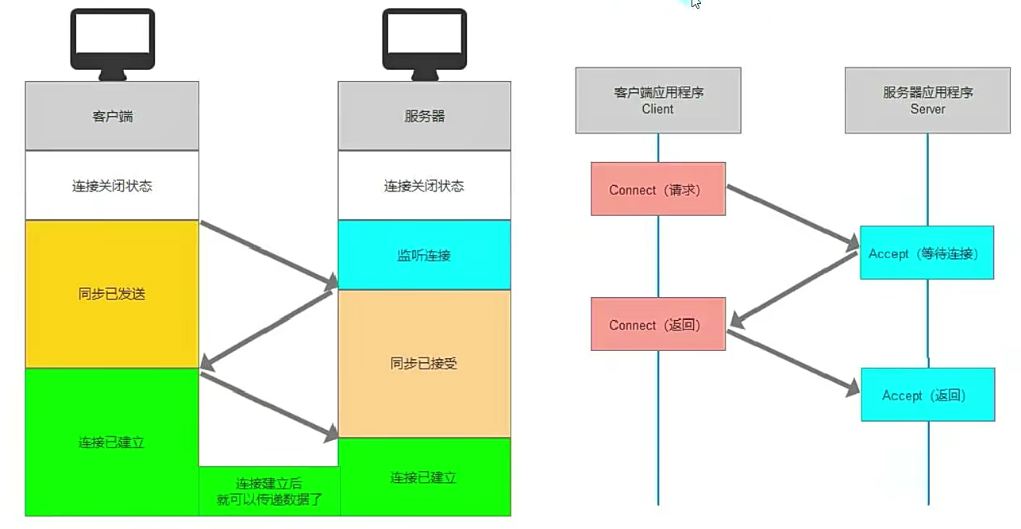
异步发送和接受数据

释放连接并关闭Socket, 先与Close调用

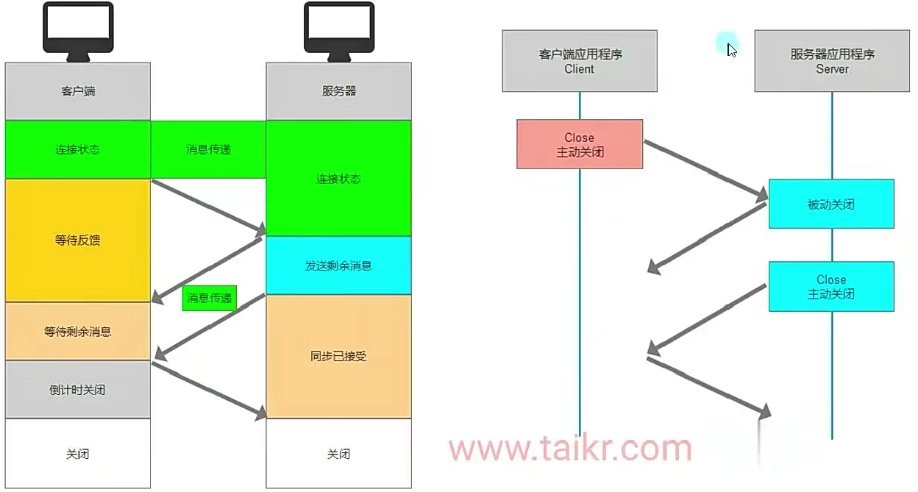
socketTcp.Shutdown(SocketShutdown.Both);

关闭连接，释放所有Socket关联资源

socketTcp.Close();

Socket套接字TCP通信概述

三次握手的体现

四次挥手的体现

***TCP协议的三次握手和四次挥手被Socket封装在内部，不需要我们额外处理***。

服务端

在VisualStudio中创建C#控制台应用

|  |
| --- |
| using System;  using System.Net;  using System.Net.Sockets;  using System.Text;  namespace ServerSide  {  class Program {  static void Main(string[] args) {  //服务端要做的事情  //1.创建套接字  Socket socketTcp = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);  //2.用Bind方法将套接字与本地绑定  try  {  IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 50001);  socketTcp.Bind(ipPoint);  }  catch (Exception e){  Console.WriteLine("绑定报错" + e.Message);  return;  }  //3.用Listen方法监听  socketTcp.Listen(1024); //最大可以有1024台设备进行连接  Console.WriteLine("服务端绑定监听结束，等待客户端连入");  //4.用Accept方法等待客户端链接  //5.建立连接，Accept返回新套接字  Socket socketClient = socketTcp.Accept(); //Accept是一个堵塞方法，如果有返回值就代表有客户端连入  Console.WriteLine("有客户端连入");  //6.用Send和Receive相关方法接受数据  //发送  socketClient.Send(Encoding.UTF8.GetBytes("欢迎连入服务端"));  //接收数据  byte[] result = new byte[1024];  int receiveNum = socketClient.Receive(result); //返回值为接受到的字节数  Console.WriteLine($"接收到了{socketClient.RemoteEndPoint.ToString()}发来的消息: {Encoding.UTF8.GetString(result, 0, receiveNum)}");  //7.用Shutdown 方法释放连接  socketClient.Shutdown(SocketShutdown.Both);  //8.关闭套接字  socketClient.Close();  Console.WriteLine("按任意键退出");  Console.ReadKey();  }  }  } |

客户端

在Unity中，创建脚本，并将脚本挂载到物体上，运行Unity。

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using System.Net;  using System.Net.Sockets;  using System.Text;  using UnityEngine;  public class Lesson22\_ClientSide : MonoBehaviour  {  // Start is called before the first frame update  void Start()  {  //客户端要做的事情  //1.创建套接字Socket  Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);  //2.用Connect方法与服务端相连  IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 50001); //这里填写的远程连接服务端的IP和端口号  try  {  socket.Connect(ipPoint);  }  catch(SocketException e) {  if (e.ErrorCode == 10061)  {  print("服务器拒绝连接");  }  else {  print("连接失败"+ e.ErrorCode);  }  return;  }  //3. 用Send和Receive相关方法接受数据  byte[] receiveBytes = new byte[1024];  int receiveNum = socket.Receive(receiveBytes);  print("收到服务端发来的消息:" + Encoding.UTF8.GetString(receiveBytes,0, receiveNum));  socket.Send(Encoding.UTF8.GetBytes("你好，我是客户端"));  //4. 用Shutdown方法释放连接  socket.Shutdown(SocketShutdown.Both);  //5. 关闭套接字  socket.Close();  //客户端的 Connect、Send、Receive都是会堵塞主线程。  }  // Update is called once per frame  void Update()  {    }  } |

注意

现在的一切，都只是些鸡毛蒜皮，***你只能用你的电脑连接到你的服务器***。

***朋友在连接你的热点的情况下，需要修改代码，才能连接你的服务器。***

***当朋友未和你在同一局域网下，你需要将你的IP(你在电脑上看见的IP，默认路由器分配的内网IP)转为公网IP，并配置为服务器，才能让别人连接进来***。

区分消息类型

如何发送之前自定义类的2进制信息

我们已经写过BaseData类，第一步发送的数据要继承BaseData类

使用其序列化方法，反序列化方法。

***问题***: 将序列化的2进制数据发送给对象时，对方该如何区分？

* PlayerInfo: 玩家信息
* ChatInfo: 聊天信息
* LoginInfo: 登录信息

等等数据

这些数据对象序列化后是长度不同的字节数组，将他们发送给对象后，***对方如何区分出他们分别是什么消息***。

如何区分消息

解决方案：

为发送的消息添加标识，比如添加消息ID。

举例：

比如我们规定，接收到的消息前4个字节为ID(使用了int类型作为ID)，后面的才是数据部分。

ID

数据部分

当然我们可以用任何类型作为头部，比如int,short,byte,long…这些都可以。

实践

1. 创建消息基类，基类继承BaseData，基类添加获取消息ID的方法或者属性
2. 让想要被发送的消息继承该类，实现序列化反序列化方法。
3. 修改客户端和服务端收发消息逻辑

BaseMsg 基础消息类

|  |
| --- |
| public class BaseMsg : BaseData  {  public override int GetBytesNum(){}  public override int Reading(byte[] bytes, int beginIndex = 0){}  public override byte[] Writing(){}  public virtual int GetID(){  return 0;  }  } |

PlayerMsg 消息类

|  |
| --- |
| public class PlayerMsg: BaseMsg{  public int playerID;  public PlayerData playerData;  public override byte[] Writing(){  int index = 0;  byte[] bytes = new byte[GetBytesNum()];  //先写消息ID  WriteInt(bytes, GetID(), ref index);  //写这个消息的成员变量  WriteInt(bytes，playerID, ref index);  WriteData(bytes, playerData, ref index);  return bytes;  }  public override int Reading(byte[] bytes, int beginIndex =0){  //反序列化不需要去解析ID，因为在这一步之前，就应该把ID反序列化出来  //用来判断到底使用哪一个自定义类来反序列化  int index = beginIndex;  playerID = ReadInt(bytes, ref index);  playerData = ReadData<PlayerData>(bytes, ref index);  return index – beginIndex;  }  public override int GetBytesNum(){  return 4+ //消息ID的长度  4+ //playerID的字节数组长度  playerData.GetBytesNum(); //playerData的字节数组长度    }  //自定义消息ID，主要用于区分是哪一个消息类  public override int GetID(){  return 1001;  }  } |

PlayerData 玩家数据类

|  |
| --- |
| public class PlayerData : BaseData  {  public string name;  public int atk;  public int lev;    public override byte[] Writing(){  int index = 0;  byte[] bytes = new byte[GetBytesNum()];  WriteString(bytes, name, ref index);  WriteInt(bytes, atk, ref index);  WriteInt(bytes, lev, ref index);  return bytes;  }  public override int Reading(byte[] bytes, int beginIndex =0){  int index = beginIndex;  name = ReadString(bytes, ref index);  atk = ReadInt(bytes, ref index);  lev = ReadInt(bytes, ref index);  return index – beginIndex;  }  public override int GetBytesNum(){  return 4 + 4 + 4 + Encoding.UTF8.GetBytes(name).Length;  }  } |

然后，就可以修改客户端和服务端收发消息的逻辑

ClientSide 客户端代码

|  |
| --- |
| public class Lesson22\_ClientSide : MonoBehaviour  {  // Start is called before the first frame update  void Start()  {  //客户端要做的事情  //1.创建套接字Socket  Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);  //2.用Connect方法与服务端相连  IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 50001); //这里填写的远程连接服务端的IP和端口号  try  {  socket.Connect(ipPoint);  }  catch(SocketException e) {  if (e.ErrorCode == 10061)  {  print("服务器拒绝连接");  }  else {  print("连接失败"+ e.ErrorCode);  }  return;  }  //3. 用Send和Receive相关方法接受数据  byte[] receiveBytes = new byte[1024];  int receiveNum = socket.Receive(receiveBytes);  int msgID = BitConverter.ToInt32(receiveBytes, 0);  switch(msgID){  case 1001:  PlayerMsg msg = new PlayerMsg();  msg.Reading(reciveBytes, 4); //因为已经读了4个字节  print(msg.playerID);  print(msg.playerData.name);  print(msg.playerData.atk);  print(msg.playerData.lev);  }  print("收到服务端发来的消息:" + Encoding.UTF8.GetString(receiveBytes,0, receiveNum));  socket.Send(Encoding.UTF8.GetBytes("你好，我是客户端"));  //4. 用Shutdown方法释放连接  socket.Shutdown(SocketShutdown.Both);  //5. 关闭套接字  socket.Close();  //客户端的 Connect、Send、Receive都是会堵塞主线程。  }  } |

ServerSide

在修改之前，将Unity工程的BaseData,BaseMsg,PlayerMsg,PlayerData拖入该工程中

然后在该工程中把using UnityEngine全部都删掉

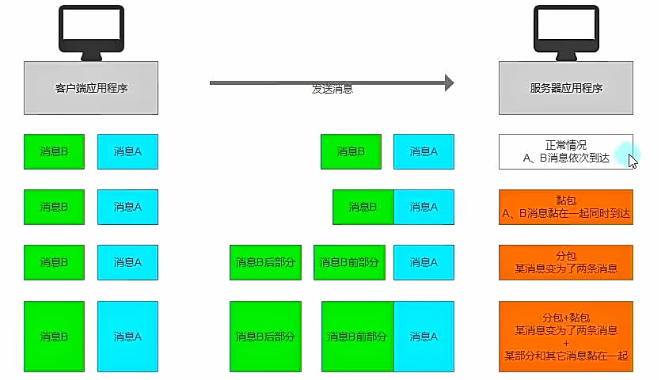
|  |
| --- |
| namespace ServerSide  {  class Program {  static void Main(string[] args) {  //服务端要做的事情  //1.创建套接字  Socket socketTcp = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);  //2.用Bind方法将套接字与本地绑定  try  {  IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 50001);  socketTcp.Bind(ipPoint);  }  catch (Exception e){  Console.WriteLine("绑定报错" + e.Message);  return;  }  //3.用Listen方法监听  socketTcp.Listen(1024); //最大可以有1024台设备进行连接  Console.WriteLine("服务端绑定监听结束，等待客户端连入");  //4.用Accept方法等待客户端链接  //5.建立连接，Accept返回新套接字  Socket socketClient = socketTcp.Accept(); //Accept是一个堵塞方法，如果有返回值就代表有客户端连入  Console.WriteLine("有客户端连入");  //6.用Send和Receive相关方法接受数据  //发送  PlayerMsg msg = new PlayerMsg();  msg.playerID = 666;  msg.playerData = new PlayerData();  msg.playerData.name = “我是服务端”;  msg.playerData.atk = 99;  msg.playerData.lev = 50;  socketClient.Send(msg.Writing())  **//socketClient.Send(Encoding.UTF8.GetBytes("欢迎连入服务端"));**  //接收数据  byte[] result = new byte[1024];  int receiveNum = socketClient.Receive(result); //返回值为接受到的字节数  Console.WriteLine($"接收到了{socketClient.RemoteEndPoint.ToString()}发来的消息: {Encoding.UTF8.GetString(result, 0, receiveNum)}");  //7.用Shutdown 方法释放连接  socketClient.Shutdown(SocketShutdown.Both);  //8.关闭套接字  socketClient.Close();  Console.WriteLine("按任意键退出");  Console.ReadKey();  }  }  } |

这里用服务端发送消息，客户端接收消息。

分包、黏包

什么是分包，黏包

分包、黏包在网络通信中由于各种因素(网络环境，API规则等)造成的消息与消息之间出现的两种状态。

* 分包：一个消息分成了多个消息进行发送。
* 黏包：一个消息和另一个消息黏在了一起

**注意**：分包和黏包可能同时发生。

如何解决分包，黏包问题

目前的处理中：

我们收到的消息都是以字节数组的形式在程序中体现：

我们的处理规则是默认传过来的消息就是正常情况，前4个字节是消息ID，后面的字节数组全部用来反序列化。如果出现分包，黏包会导致我们反序列化报错。

那么通过接收到的字节数组我们应该如何判断收到的字节数组处于以下状态？

1. 正常
2. 分包
3. 黏包

**解决方案**：***消息长度***

数据部分

长度

ID

实践解决

1. 为所有的消息添加头部信息，用于存储其消息长度
2. 根据分包，黏包的表现情况，修改接收消息处的逻辑

现在我们来修改PlayerMsg的代码

PlayerMsg

|  |
| --- |
| public class PlayerMsg : BaseMsg  {  public int playerID;  public PlayerData playerData;  public override byte[] Writing()  {  int index = 0;  int bytesNum = GetBytesNum();  byte[] bytes = new byte[bytesNum];  //先写消息ID  WriteInt(bytes, GetID(), ref index);  //写消息体的长度  WriteInt(bytes, bytesNum - 8, ref index);  //写这个消息的成员变量  WriteInt(bytes, playerID, ref index);  WriteData(bytes, playerData, ref index);  return bytes;  }  public override int Reading(byte[] bytes, int beginIndex = 0)  {  //反序列化不需要去解析ID，因为在这一步之前，就应该把ID反序列化出来  //用来判断到底使用哪一个自定义类来反序列化  int index = beginIndex;  playerID = ReadInt(bytes, ref index);  playerData = ReadData<PlayerData>(bytes, ref index);  return index - beginIndex;  }  public override int GetBytesNum()  {  return 4 + //消息ID的长度  4 + //消息体的长度  4 + //playerID的字节数组长度  playerData.GetBytesNum(); //playerData的字节数组长度  }  //自定义消息ID，主要用于区分是哪一个消息类  public override int GetID()  {  return 1001;  }  } |

然后修改 NetMgr

|  |
| --- |
| public class NetMgr : MonoBehaviour  {  private static NetMgr instance;  public static NetMgr Instance => instance;  private Socket socket; //客户端Socket  //用于发送消息的队列公共容器，主线程往里面放，发送线程从里面取  private Queue<BaseMsg> sendMsgQueue = new Queue<BaseMsg>();  //用于接收消息的队列公共容器，子线程往里面放，主线程从里面取  private Queue<BaseMsg> receiveQueue = new Queue<BaseMsg>();  //private byte[] receiveBytes = new byte[1024 \* 1024]; //用于接收消息的水桶  //private int receiveNum = 0; //返回收到的字节数  //用于处理分包时，缓存的 字节数组 和 字节数组长度  private byte[] cacheBytes = new byte[1024 \* 1024];  private int cacheNum = 0;  private bool isConnected = false; //是否是连接状态  private void Awake()  {  instance = this;  DontDestroyOnLoad(gameObject);  }  // Start is called before the first frame update  void Start()  {    }  // Update is called once per frame  void Update()  {  if (receiveQueue.Count > 0) {  BaseMsg msg = receiveQueue.Dequeue();  if (msg is PlayerMsg) {  PlayerMsg playerMsg = (msg as PlayerMsg);  print(playerMsg.playerID);  print(playerMsg.playerData.name);  print(playerMsg.playerData.lev);  print(playerMsg.playerData.atk);  }  }  }  /// <summary>  /// 连接服务器  /// </summary>  /// <param name="ip"></param>  /// <param name="port"></param>  public void Connect(string ip, int port)  {  if (isConnected) //避免多次连接,报错  return;  if(socket == null)  socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);  IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(ip), port);  try  {  socket.Connect(ipPoint);  isConnected = true;  //开启线程  ThreadPool.QueueUserWorkItem(SendMsg);  ThreadPool.QueueUserWorkItem(ReceiveMsg); //接受消息线程  }  catch (SocketException e) {  if (e.ErrorCode == 10061)  {  print("服务器拒绝连接");  }  else {  print("连接失败");  }  }  }  //发送消息  public void Send(BaseMsg msg) {  sendMsgQueue.Enqueue(msg);  }  private void SendMsg(object obj) {  while (isConnected) {  if (sendMsgQueue.Count > 0) {  socket.Send(sendMsgQueue.Dequeue().Writing());  }  }  }  //接收消息  private void ReceiveMsg(object obj) {  while (isConnected) {  if (socket.Available > 0) {  byte[] receiveBytes = new byte[1024 \* 1024];  int receiveNum = socket.Receive(receiveBytes);  HandleReceiveMsg(receiveBytes, receiveNum);  //首先读出ID  //int msgID = BitConverter.ToInt32(receiveBytes, 0);  //BaseMsg baseMsg = null;  //switch (msgID) {  // case 1001:  // PlayerMsg msg = new PlayerMsg();  // msg.Reading(receiveBytes, 4);  // baseMsg = msg;  // break;  //}  //if (baseMsg == null) {  // continue;  //}  //receiveQueue.Enqueue(baseMsg);  }  }  }  //处理接收消息 分包、黏包的方法  private void HandleReceiveMsg(byte[] receiveBytes, int receiveNum) {  int msgID = 0;  int msgLength = 0;  int nowIndex = 0;  //收到消息时 应该看看 之前有没有缓存的 如果有，我们就直接拼接  receiveBytes.CopyTo(cacheBytes, cacheNum);  cacheNum += receiveNum;  //通过循环解决黏包问题  while (true) {  msgLength = -1; //将msgLength置为-1，是为了防止分包和黏包同时出现的情况下，上一次黏包解析的数据还残留  //可能出现第一个if没进去，但是第二个if是上一个数据进去了。  if (cacheNum - nowIndex >= 8) {  //解析ID  msgID = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  //解析长度  msgLength = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  }  if (cacheNum - nowIndex >= msgLength && msgLength != -1)  {  //解析消息体  BaseMsg baseMsg = null;  switch (msgID)  {  case 1001:  PlayerMsg msg = new PlayerMsg();  msg.Reading(cacheBytes, nowIndex);  baseMsg = msg;  break;  }  if (baseMsg != null)  receiveQueue.Enqueue(baseMsg);  nowIndex += msgLength;  //缓存解决完毕  if (nowIndex == cacheNum) {  cacheNum = 0;  break;  }  }  else {  //如果不满足，证明有分包  //那么我们需要把当前收到的内容，记录下来  //有待下次接收到消息后，再做处理  //receiveBytes.CopyTo(cacheBytes, 0);  //cacheNum = receiveNum;  //如果进行了id和长度的解析，但是没有成功解析消息体 就需要减去nowIndex移动的位置  if (msgLength != -1) {  nowIndex -= 8;  }  //把剩余没有解析的字节数组内容移动到前面来，用于缓存下次接续解析  Array.Copy(cacheBytes, nowIndex, cacheBytes, 0, cacheNum - nowIndex);  cacheNum = cacheNum - nowIndex;  break;  }  }  }  public void Close() {  if (socket != null) {  socket.Dispose();  socket.Close();  isConnected = false;  }  }  private void OnDestroy() {  Close();  }  } |

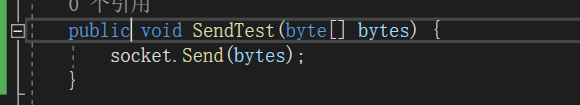
目前解决的客户端分包黏包问题，服务端也需要解决以上问题

修改服务端的ClientSocket类

|  |
| --- |
| namespace ServerSideExercises2  {  internal class ClientSocket  {  private static int CLIENT\_BEGIN\_ID = 1;  public int clientID;  public Socket socket;  //用于处理分包时，缓存的 字节数组 和 字节数组长度  private byte[] cacheBytes = new byte[1024 \* 1024];  private int cacheNum = 0;  public bool Connected => this.socket.Connected; //是否是连接状态  public ClientSocket(Socket socket)  {  this.clientID = CLIENT\_BEGIN\_ID;  this.socket = socket;  CLIENT\_BEGIN\_ID++;  }  //关闭  public void Close() {    if (socket != null) {  socket.Shutdown(SocketShutdown.Both);  socket.Close();  socket = null;  }  }  //发送  //public void Send(string info) {  // if(socket != null)  // socket.Send(Encoding.UTF8.GetBytes(info));  //}  public void Send(BaseMsg info) {  if (socket != null)  {  try  {  socket.Send(info.Writing());  }  catch (Exception e){  Console.WriteLine("发消息出错" + e.Message);  Close();  }  }  }  //接收  public void Receive()  {  if (socket == null)  {  return;  }  if (socket.Available > 0) {  byte[] result = new byte[1024 \* 5];  int receiveNum = socket.Receive(result);  HandleReceiveMsg(result, receiveNum);  //收到数据后，先读取4个字节转为ID  //int msgID = BitConverter.ToInt32(result, 0);  //BaseMsg msg = null;  //switch (msgID) {  // case 1001:  // PlayerMsg playermsg = new PlayerMsg();  // playermsg.Reading(result, 4);  // msg = playermsg;  // break;  //}  ////ThreadPool.QueueUserWorkItem(MsgHandle, Encoding.UTF8.GetString(result, 0, receiveNum));  //if (msg == null) {  // return;  //}  //ThreadPool.QueueUserWorkItem(MsgHandle, msg);  }    }  //处理接收消息 分包、黏包的方法  private void HandleReceiveMsg(byte[] receiveBytes, int receiveNum)  {  int msgID = 0;  int msgLength = 0;  int nowIndex = 0;  //收到消息时 应该看看 之前有没有缓存的 如果有，我们就直接拼接  receiveBytes.CopyTo(cacheBytes, cacheNum);  cacheNum += receiveNum;  //通过循环解决黏包问题  while (true)  {  msgLength = -1; //将msgLength置为-1，是为了防止分包和黏包同时出现的情况下，上一次黏包解析的数据还残留  //可能出现第一个if没进去，但是第二个if是上一个数据进去了。  if (cacheNum - nowIndex >= 8)  {  //解析ID  msgID = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  //解析长度  msgLength = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  }  if (cacheNum - nowIndex >= msgLength && msgLength != -1)  {  //解析消息体  BaseMsg baseMsg = null;  switch (msgID)  {  case 1001:  PlayerMsg msg = new PlayerMsg();  msg.Reading(cacheBytes, nowIndex);  baseMsg = msg;  break;  }  if (baseMsg != null)  ThreadPool.QueueUserWorkItem(MsgHandle, baseMsg);  nowIndex += msgLength;  //缓存解决完毕  if (nowIndex == cacheNum)  {  cacheNum = 0;  break;  }  }  else  {  //如果不满足，证明有分包  //那么我们需要把当前收到的内容，记录下来  //有待下次接收到消息后，再做处理  //receiveBytes.CopyTo(cacheBytes, 0);  //cacheNum = receiveNum;  //如果进行了id和长度的解析，但是没有成功解析消息体 就需要减去nowIndex移动的位置  if (msgLength != -1)  {  nowIndex -= 8;  }  //把剩余没有解析的字节数组内容移动到前面来，用于缓存下次接续解析  Array.Copy(cacheBytes, nowIndex, cacheBytes, 0, cacheNum - nowIndex);  cacheNum = cacheNum - nowIndex;  break;  }  }  }  private void MsgHandle(object obj) {  //string str = obj as string;  //Console.WriteLine($"收到客户端{this.socket.RemoteEndPoint}发来的消息: {str}");  BaseMsg msg = obj as BaseMsg;  if (msg is PlayerMsg)  {  PlayerMsg playerMsg = msg as PlayerMsg;  Console.WriteLine(playerMsg.playerID);  Console.WriteLine(playerMsg.playerData.name);  Console.WriteLine(playerMsg.playerData.lev);  Console.WriteLine(playerMsg.playerData.atk);  }  }  } |

做完这些，我们还需要技术测试，我们可以手动发送分包，黏包。

1.在NetMsg中添加SendTest方法，用于发送数组

2.然后在ClientButton脚本中添加各种行为

|  |
| --- |
| btn.onClick.AddListener(() =>  {  //由于对代码进行了修改，这里不再传输文本  //if (input.text != "") {  // NetMgr.Instance.Send(input.text);  //}  PlayerMsg ms = new PlayerMsg();  ms.playerID = 1111;  ms.playerData = new PlayerData();  ms.playerData.name = "Yang";  ms.playerData.lev = 10;  ms.playerData.atk = 20;  NetMgr.Instance.Send(ms);  });  //黏包测试  btn1.onClick.AddListener(() =>  {  PlayerMsg ms1 = new PlayerMsg();  ms1.playerID = 1001;  ms1.playerData = new PlayerData();  ms1.playerData.name = "Yang1";  ms1.playerData.lev = 10;  ms1.playerData.atk = 20;  PlayerMsg ms2 = new PlayerMsg();  ms2.playerID = 1002;  ms2.playerData = new PlayerData();  ms2.playerData.name = "Yang2";  ms2.playerData.lev = 20;  ms2.playerData.atk = 40;  byte[] bytes = new byte[ms1.GetBytesNum() + ms2.GetBytesNum()];  ms1.Writing().CopyTo(bytes, 0);  ms2.Writing().CopyTo(bytes, ms1.GetBytesNum());  NetMgr.Instance.SendTest(bytes);  });  //分包测试  //如果两个包发的很快，也会自动粘起来，所以可以使用异步  btn2.onClick.AddListener(async () =>  {  PlayerMsg msg = new PlayerMsg();  msg.playerID = 1003;  msg.playerData = new PlayerData();  msg.playerData.name = "Yang3";  msg.playerData.lev = 3;  msg.playerData.atk = 3;  byte[] bytes = msg.Writing();  //分包  byte[] bytes1 = new byte[10];  byte[] bytes2 = new byte[bytes.Length - 10];  //第一个包  Array.Copy(bytes, 0 , bytes1, 0, 10);  //第二个包  Array.Copy(bytes, 10, bytes2, 0, bytes.Length - 10);  NetMgr.Instance.SendTest(bytes1);  await Task.Delay(500);  NetMgr.Instance.SendTest(bytes2);  });  //分包，黏包测试  btn3.onClick.AddListener(async () =>  {  PlayerMsg ms1 = new PlayerMsg();  ms1.playerID = 1001;  ms1.playerData = new PlayerData();  ms1.playerData.name = "Yang1";  ms1.playerData.lev = 10;  ms1.playerData.atk = 20;  PlayerMsg ms2 = new PlayerMsg();  ms2.playerID = 1002;  ms2.playerData = new PlayerData();  ms2.playerData.name = "Yang2";  ms2.playerData.lev = 20;  ms2.playerData.atk = 40;  byte[] bytes1 = ms1.Writing(); //消息A  byte[] bytes2 = ms2.Writing(); //消息B  byte[] bytes2\_1 = new byte[10];  byte[] bytes2\_2 = new byte[bytes2.Length - 10];  //分成第一个包  Array.Copy(bytes2, 0, bytes2\_1,0,10);  //分成第二个包  Array.Copy(bytes2, 10, bytes2\_2, 0, bytes2.Length - 10);  //消息A和消息B1的黏包  byte[] bytes = new byte[bytes1.Length + bytes2\_1.Length];  bytes1.CopyTo(bytes, 0);  bytes2\_1.CopyTo(bytes, bytes1.Length);  NetMgr.Instance.SendTest(bytes);  await Task.Delay(500);  NetMgr.Instance.SendTest(bytes2\_2);  }); |

心跳消息——客户端主动断开连接

目前的客户端主动断开连接

目前的客户端主动退出，客户端会调用socket的ShutDown和Close方法

但是通过调用这两个方法后，服务端无法得知客户端已经主动断开。

之前写的脚本是通过socket.Connected字段得到客户端是否与服务器处于连接的状态。

但是这个字段有个弊端：***它并不是主动去检测连接状态，而是进行I/O操作后才去判断是否在连接——它返回的是最后一个I/O操作的连接状态***。

解决目前断开不及时的问题

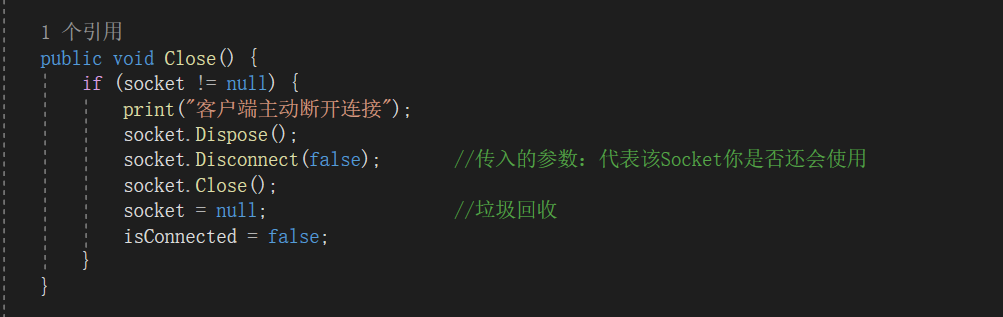
1. 客户端尝试使用Disconnect方法主动断开连接

Socket当中有一个专门在客户端使用的方法

Disconnect()

客户端调用该方法和服务端断开连接

看是否是因为之前直接Close而没有调用Disconnect造成服务端无法及时获取状态。

NetMgr

**服务端处理删除记录的socket的相关逻辑**

ClientSocket

|  |
| --- |
| namespace ServerSideExercises2  {  internal class ClientSocket  {  private static int CLIENT\_BEGIN\_ID = 1;  public int clientID;  public Socket socket;  //用于处理分包时，缓存的 字节数组 和 字节数组长度  private byte[] cacheBytes = new byte[1024 \* 1024];  private int cacheNum = 0;  public bool Connected => this.socket.Connected; //是否是连接状态  public ClientSocket(Socket socket)  {  this.clientID = CLIENT\_BEGIN\_ID;  this.socket = socket;  CLIENT\_BEGIN\_ID++;  }  //关闭  public void Close() {    if (socket != null) {  socket.Shutdown(SocketShutdown.Both);  socket.Close();  socket = null;  }  }  //发送  //public void Send(string info) {  // if(socket != null)  // socket.Send(Encoding.UTF8.GetBytes(info));  //}  public void Send(BaseMsg info) {  if (Connected)  {  try  {  socket.Send(info.Writing());  }  catch (Exception e)  {  Console.WriteLine("发消息出错" + e.Message);  Program.socket.AddDelSocket(this);  Close();  }  }  else {  Program.socket.AddDelSocket(this);  }  }  //接收  public void Receive()  {  if (!Connected)  {  Program.socket.AddDelSocket(this);  return;  }  if (socket.Available > 0) {  byte[] result = new byte[1024 \* 5];  int receiveNum = socket.Receive(result);  HandleReceiveMsg(result, receiveNum);  //收到数据后，先读取4个字节转为ID  //int msgID = BitConverter.ToInt32(result, 0);  //BaseMsg msg = null;  //switch (msgID) {  // case 1001:  // PlayerMsg playermsg = new PlayerMsg();  // playermsg.Reading(result, 4);  // msg = playermsg;  // break;  //}  ////ThreadPool.QueueUserWorkItem(MsgHandle, Encoding.UTF8.GetString(result, 0, receiveNum));  //if (msg == null) {  // return;  //}  //ThreadPool.QueueUserWorkItem(MsgHandle, msg);  }    }  //处理接收消息 分包、黏包的方法  private void HandleReceiveMsg(byte[] receiveBytes, int receiveNum)  {  int msgID = 0;  int msgLength = 0;  int nowIndex = 0;  //收到消息时 应该看看 之前有没有缓存的 如果有，我们就直接拼接  receiveBytes.CopyTo(cacheBytes, cacheNum);  cacheNum += receiveNum;  //通过循环解决黏包问题  while (true)  {  msgLength = -1; //将msgLength置为-1，是为了防止分包和黏包同时出现的情况下，上一次黏包解析的数据还残留  //可能出现第一个if没进去，但是第二个if是上一个数据进去了。  if (cacheNum - nowIndex >= 8)  {  //解析ID  msgID = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  //解析长度  msgLength = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  }  if (cacheNum - nowIndex >= msgLength && msgLength != -1)  {  //解析消息体  BaseMsg baseMsg = null;  switch (msgID)  {  case 1001:  PlayerMsg msg = new PlayerMsg();  msg.Reading(cacheBytes, nowIndex);  baseMsg = msg;  break;  }  if (baseMsg != null)  ThreadPool.QueueUserWorkItem(MsgHandle, baseMsg);  nowIndex += msgLength;  //缓存解决完毕  if (nowIndex == cacheNum)  {  cacheNum = 0;  break;  }  }  else  {  //如果不满足，证明有分包  //那么我们需要把当前收到的内容，记录下来  //有待下次接收到消息后，再做处理  //receiveBytes.CopyTo(cacheBytes, 0);  //cacheNum = receiveNum;  //如果进行了id和长度的解析，但是没有成功解析消息体 就需要减去nowIndex移动的位置  if (msgLength != -1)  {  nowIndex -= 8;  }  //把剩余没有解析的字节数组内容移动到前面来，用于缓存下次接续解析  Array.Copy(cacheBytes, nowIndex, cacheBytes, 0, cacheNum - nowIndex);  cacheNum = cacheNum - nowIndex;  break;  }  }  }  private void MsgHandle(object obj) {  //string str = obj as string;  //Console.WriteLine($"收到客户端{this.socket.RemoteEndPoint}发来的消息: {str}");  BaseMsg msg = obj as BaseMsg;  if (msg is PlayerMsg)  {  PlayerMsg playerMsg = msg as PlayerMsg;  Console.WriteLine(playerMsg.playerID);  Console.WriteLine(playerMsg.playerData.name);  Console.WriteLine(playerMsg.playerData.lev);  Console.WriteLine(playerMsg.playerData.atk);  }  }  }    } |

ServerSocket

|  |
| --- |
| namespace ServerSideExercises2  {  internal class ServerSocket  {  //服务端Socket  public Socket socket;  //客户端连接的所有Socket  public Dictionary<int, ClientSocket> clientDic = new Dictionary<int, ClientSocket>();  //有待移除的客户端socket 这些socket实际上已经断开连接了  private List<ClientSocket> delList = new List<ClientSocket>();  private bool isClose;  //开启服务器端  public void Start(string ip, int port, int num) {  isClose = false;  socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);  IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(ip), port);  socket.Bind(ipPoint);  socket.Listen(num);  ThreadPool.QueueUserWorkItem(Accept);  ThreadPool.QueueUserWorkItem(Receive);  }  //关闭服务器端  public void Close() {  isClose = true;  foreach (ClientSocket client in clientDic.Values) {  client.Close();  }  clientDic.Clear();  socket.Shutdown(SocketShutdown.Both);  socket.Close();  socket = null; //回收  }  //接收客户端连入  private void Accept(object obj) {  while (!isClose) {  try  {  //连入一个客户端  Socket clientSocket = socket.Accept();  ClientSocket client = new ClientSocket(clientSocket);  //client.Send("欢迎连入服务器");  lock (clientDic)  {  clientDic.Add(client.clientID, client);  }  }  catch (Exception e)  {  Console.WriteLine("客户端连入报错" + e.Message);  }  }  }  //接收客户端消息  private void Receive(object obj) {  while (!isClose) {  if (clientDic.Count > 0) {  lock (clientDic) {  foreach (ClientSocket client in clientDic.Values)  {  client.Receive();  }  //判断有没有 断开连接的 把其移除  CloseDelListSocket();  }  }  }  }  //广播消息  //public void Broadcast(string info) {  // foreach (ClientSocket client in clientDic.Values) {  // client.Send(info);  // }  //}  public void Broadcast(BaseMsg info)  {  lock (clientDic) {  foreach (ClientSocket client in clientDic.Values)  {  client.Send(info);  }  }  }  //添加待移除的 socket内容  public void AddDelSocket(ClientSocket socket) {  if (!delList.Contains(socket)) {  delList.Add(socket);  }  }  public void CloseDelListSocket() {  for (int i = 0; i < delList.Count; i++)  {  CloseClientSocket(delList[i]);  }  delList.Clear();  }  //关闭客户端的连接的 从字典移除  public void CloseClientSocket(ClientSocket socket) {  lock (clientDic) //线程锁：当我在使用该方法时候，别的线程无法访问指定变量的内存空间  {  socket.Close();  if (clientDic.ContainsKey(socket.clientID))  {  clientDic.Remove(socket.clientID);  Console.WriteLine($"客户端{socket.clientID}主动断开连接!");  }  }  }  }  } |

Program

|  |
| --- |
| namespace ServerSideExercises2  {  internal class Program  {  public static ServerSocket socket;  static void Main(string[] args)  {  socket = new ServerSocket();  socket.Start("127.0.0.1", 50001, 1024);  Console.WriteLine("服务器开启成功");  while (true) {  string input = Console.ReadLine();  if (input == "Quit")  {  socket.Close();  }  else if (input.Substring(0, 2) == "B:") {  //socket.Broadcast(input.Substring(2));  PlayerMsg msg = new PlayerMsg();  msg.playerID = 999;  msg.playerData = new PlayerData();  msg.playerData.name = "服务器发来的消息";  msg.playerData.lev = 99;  msg.playerData.atk = 80;  socket.Broadcast(msg);  }  }  }  }  } |

***经过测试，该方法没有作用。服务端任然不知道客户端是否断开连接***。

1. 自定义退出消息

让服务器端收到该消息就知道是客户端想要主动断开连接，然后服务器端主动处理掉socket相关工作

在客户端处，创建QuitMsg代码

|  |
| --- |
| public class QuitMsg : BaseMsg  {  public override int GetBytesNum()  {  return 4 + 4;  }  public override int Reading(byte[] bytes, int beginIndex = 0)  {  return 0;  }  public override byte[] Writing()  {  int index = 0;  byte[] bytes = new byte[GetBytesNum()];  WriteInt(bytes, GetID(), ref index);  WriteInt(bytes, 0, ref index);  return bytes;  }  public override int GetID() {  return 1003;  }  } |

修改NetMgr脚本的Close方法

|  |
| --- |
| public void Close() {  if (socket != null) {  print("客户端主动断开连接");  QuitMsg msg = new QuitMsg(); //发送断开连接的消息  socket.Send(msg.Writing());  socket.Shutdown(SocketShutdown.Both);  socket.Disconnect(false); //传入的参数：代表该Socket你是否还会使用  socket.Close();  socket = null; //垃圾回收  isConnected = false;  }  } |

然后将QuitMsg脚本复制到服务器工程文件夹中

修改ClientSocket脚本的HandleReceiveMsg方法和MsgHandle方法

|  |
| --- |
| private void HandleReceiveMsg(byte[] receiveBytes, int receiveNum)  {  int msgID = 0;  int msgLength = 0;  int nowIndex = 0;  //收到消息时 应该看看 之前有没有缓存的 如果有，我们就直接拼接  receiveBytes.CopyTo(cacheBytes, cacheNum);  cacheNum += receiveNum;  //通过循环解决黏包问题  while (true)  {  msgLength = -1; //将msgLength置为-1，是为了防止分包和黏包同时出现的情况下，上一次黏包解析的数据还残留  //可能出现第一个if没进去，但是第二个if是上一个数据进去了。  if (cacheNum - nowIndex >= 8)  {  //解析ID  msgID = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  //解析长度  msgLength = BitConverter.ToInt32(cacheBytes, nowIndex);  nowIndex += 4;  }  if (cacheNum - nowIndex >= msgLength && msgLength != -1)  {  //解析消息体  BaseMsg baseMsg = null;  switch (msgID)  {  case 1001:  PlayerMsg msg = new PlayerMsg();  msg.Reading(cacheBytes, nowIndex);  baseMsg = msg;  break;  case 1003:  baseMsg = new QuitMsg();  //由于该消息都没有消息体，所以可以不用反序列化它  break;  }  if (baseMsg != null)  ThreadPool.QueueUserWorkItem(MsgHandle, baseMsg);  nowIndex += msgLength;  //缓存解决完毕  if (nowIndex == cacheNum)  {  cacheNum = 0;  break;  }  }  else  {  //如果不满足，证明有分包  //那么我们需要把当前收到的内容，记录下来  //有待下次接收到消息后，再做处理  //receiveBytes.CopyTo(cacheBytes, 0);  //cacheNum = receiveNum;  //如果进行了id和长度的解析，但是没有成功解析消息体 就需要减去nowIndex移动的位置  if (msgLength != -1)  {  nowIndex -= 8;  }  //把剩余没有解析的字节数组内容移动到前面来，用于缓存下次接续解析  Array.Copy(cacheBytes, nowIndex, cacheBytes, 0, cacheNum - nowIndex);  cacheNum = cacheNum - nowIndex;  break;  }  }  }  private void MsgHandle(object obj) {  //string str = obj as string;  //Console.WriteLine($"收到客户端{this.socket.RemoteEndPoint}发来的消息: {str}");  BaseMsg msg = obj as BaseMsg;  if (msg is PlayerMsg)  {  PlayerMsg playerMsg = msg as PlayerMsg;  Console.WriteLine(playerMsg.playerID);  Console.WriteLine(playerMsg.playerData.name);  Console.WriteLine(playerMsg.playerData.lev);  Console.WriteLine(playerMsg.playerData.atk);  }  else if (msg is QuitMsg) {  Program.socket.AddDelSocket(this);  }  } private void MsgHandle(object obj) {  //string str = obj as string;  //Console.WriteLine($"收到客户端{this.socket.RemoteEndPoint}发来的消息: {str}");  BaseMsg msg = obj as BaseMsg;  if (msg is PlayerMsg)  {  PlayerMsg playerMsg = msg as PlayerMsg;  Console.WriteLine(playerMsg.playerID);  Console.WriteLine(playerMsg.playerData.name);  Console.WriteLine(playerMsg.playerData.lev);  Console.WriteLine(playerMsg.playerData.atk);  }  else if (msg is QuitMsg) {  Program.socket.AddDelSocket(this);  }  } |

经过测试，该方法有效

总结

客户端可以用过Disconnect方法主动和服务器断开连接

服务端可以通过Connected属性判断连接状态决定是否释放Socket

但是由于服务器端Connected变量表示的是上一次收发消息是否成功，所以服务端无法准确判断客户端的连接状态。因此我们***需要自定义一条退出消息，当客户端退出连接时，向服务器发送该消息，服务器接受到消息后确认是退出消息，主动与客户端断开连接***。

目前学习的这一项，在为心跳消息做铺垫。

心跳消息——实现心跳消息

所谓心跳消息，就是在长连接中，客户端和服务端之间定期发送一种特殊的数据包，用于通知对方自己还在线，以确保长连接的有效性。由于发送的时间间隔往往是固定的持续的，就像是心跳一样一直存在。

为什么需要心跳消息？

1. 避免非正常关闭客户端时，服务器无法正常收到关闭连接消息
2. 避免客户端长期不发送消息，防火墙或者路由器会断开连接，我们可以通过心跳消息一直保持活跃状态。

实现心跳消息

**客户端**

主要功能: 定时发送消息

创建新的脚本 HeartMsg

|  |
| --- |
| public class QuitMsg : BaseMsg  {  public override int GetBytesNum()  {  return 4 + 4;  }  public override int Reading(byte[] bytes, int beginIndex = 0)  {  return 0;  }  public override byte[] Writing()  {  int index = 0;  byte[] bytes = new byte[GetBytesNum()];  WriteInt(bytes, GetID(), ref index);  WriteInt(bytes, 0, ref index);  return bytes;  }  public override int GetID() {  return 1003;  }  } |

NetMsg修改

|  |
| --- |
| private int SEND\_HEART\_MSG\_TIME = 2; //发送心跳消息间隔  private HeartMsg heartMsg = new HeartMsg();  private void Awake()  {  instance = this;  DontDestroyOnLoad(gameObject);  InvokeRepeating("SendHeartMsg", 0, SEND\_HEART\_MSG\_TIME); //启动 心跳消息  }  private void SendHeartMsg() {  if (isConnected) {  Send(heartMsg);  }  } |

**服务器**

主要功能: 不停检测上次收到某客户端消息的时间，如果超时则认为连接已经断开

将心跳消息HeartMsg拷贝到服务器文件夹

在ClientSokcet中的HandleReciveMsg方法中添加新的消息处理

|  |
| --- |
| switch (msgID)  {  case 1001:  PlayerMsg msg = new PlayerMsg();  msg.Reading(cacheBytes, nowIndex);  baseMsg = msg;  break;  case 1003:  baseMsg = new QuitMsg();  //由于该消息都没有消息体，所以可以不用反序列化它  break;  case 999:  baseMsg = new HeartMsg();  break;  } |

在MsgHandle方法中，添加新的消息处理

|  |
| --- |
| private void MsgHandle(object obj) {  //string str = obj as string;  //Console.WriteLine($"收到客户端{this.socket.RemoteEndPoint}发来的消息: {str}");  BaseMsg msg = obj as BaseMsg;  if (msg is PlayerMsg)  {  PlayerMsg playerMsg = msg as PlayerMsg;  Console.WriteLine(playerMsg.playerID);  Console.WriteLine(playerMsg.playerData.name);  Console.WriteLine(playerMsg.playerData.lev);  Console.WriteLine(playerMsg.playerData.atk);  }  else if (msg is QuitMsg)  {  Program.socket.AddDelSocket(this);  }  else if (msg is HeartMsg) {  frontTime = DateTime.Now.Ticks / TimeSpan.TicksPerSecond; //记录上一次发送心跳消息的时间 秒  }  } |

ClientSocket

|  |
| --- |
| private long frontTime = -1; //上一次收到心跳消息的时间  private static int TIME\_OUT\_TIME = 10; //心跳超时时间  public ClientSocket(Socket socket)  {  this.clientID = CLIENT\_BEGIN\_ID;  this.socket = socket;  CLIENT\_BEGIN\_ID++;  ThreadPool.QueueUserWorkItem(CheckTimeout);  }  /// <summary>  /// 检测心跳超时方法  /// </summary>  /// <param name="obj"></param>  private void CheckTimeout(Object obj) {  while (Connected) {  if (frontTime != -1 && DateTime.Now.Ticks / TimeSpan.TicksPerSecond - frontTime >= TIME\_OUT\_TIME)  {  Program.socket.AddDelSocket(this); //待移除客户端  break;  }  Thread.Sleep(5000); //线程休息5秒  }  } |

注意：***这里通过开线程池去检查心跳是否超时，其实是非常浪费性能的行为，因为连接客户端数量多了，就要开多个线程，只会随着客户端的连入不断变卡。目前只是为了学习心跳包，实际开发遇到此问题要学会优化***。

TCP异步通信

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*我是分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

之前学习的都是同步通信，从现在开始，你将进入Tcp异步通信课程

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*我是分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

异步通信常用的方法

举例说明异步方法原理

1. 线程回调——不影响主线程逻辑

|  |
| --- |
| CountDownAsync(5, ()=>{  print(“倒计时结束”);  })  print(“异步执行后的逻辑”); //该语句不会等待倒计时结束  public void CountDownAsync(int second，UnityAction callBack)  {  Thread t = new Thread(()=>{  while(true){  print(second);  Thread.Sleep(1000);  --second;  if(second == 0)  break;  }  callback ?.Invoke();  });  t.Start();  print(“开始倒计时”);  } |

1. async和await会等待线程执行完毕，继续执行后面的逻辑

|  |
| --- |
| CountDownAsync(5);  print(“异步执行后的逻辑”); //该语句不会等待倒计时结束  public async void CountDownAsync(int second){  print(“倒计时开始”);  await Task.Run(() =>{  while(true){  print(second);  Thread.Sleep(1000);  --second;  if(second == 0){  break;  }  }  })  print(“倒计时结束”);  } |

Socket Tcp通信中的异步方法(Begin开头方法)

回调函数参数IAsyncResult

AsyncState 调用异步方法时传入的参数，需要转化

AsyncWaitHandle 用于同步等待

|  |
| --- |
| Socket socketTcp = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp); |

**服务器相关**

**BeginAccept**: 启动异步操作来接受一个传入的连接请求。

* 参数一：指定连接请求完成时要调用的回调函数(当有客户端成功连接时，自动执行这个方法)
* 参数二: 传递给回调函数的“状态对象”(可以是任意类型，这里传入Socket，方便回调函数继续使用)

**EndAccept:**必须调用的方法，用于 “结束” 异步接受操作，获取到与客户端通信的clientSocket（后续通过这个 Socket 和客户端收发数据）。如果不调用EndAccept，会导致 Socket 资源泄漏。

**执行逻辑**

**调用BeginAccept方法，该方法会返回一个IAsyncResult对象(用户跟踪异步操作状态)，异步操作完成后，该方法会调用传入的回调函数，需要在回调函数中调用EndAccept接收连接结果、错误信息、结束异步操作的资源占用(这很关键)。（其他类似方法也是同理）**

BeginAccept函数有3个重载，我们用的最多的是第一个重载，传入两个参数——异步回调函数，一个object参数。第二个参数一般将当前的socket传入进去就行了。

|  |
| --- |
| socketTcp.BeginAccept(AcceptCallBack, socketTcp); //连入一个客户端就会执行以下异步方法  private void AcceptCallBack(IAsyncResult result){  try{  //从异步结果中取出之前传递的 **主监听Socket**  Socket s = result.AsyncState as Socket;  //得到连入的**客户端的Socket**  Socket clientSocket = s.EndAccept(result);  //再次启动异步监听，确保能接受下一个客户端连接  s.BeginAccept(AcceptCallBack, s);  }  catch(SocketException e){  print(e.SocketErrorCode);  }  } |

该代码会一直等待下一个客户端连接，一旦有一个客户端连接，就会执行该方法，然后继续等待下一个客户端连接。

**客户端相关**

BeginConnect

EndConnect

|  |
| --- |
| IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(“127.0.0.1”),8080);  socketTcp.BeginConnect(ipPoint, (result) =>{  Socket s =result.AsyncState as Socket;  try{  s.EndConnect(result); //完成连接  }  catch(SocketException e){  print(“连接出错” + e.SocketErrorCode + e.Message);  //TODO: 断线重连操作  }  }) |

客户端通过异步方式连接指定IP和端口的服务器，使用Lambda表达式作为回调函数处理连接结果。这里只要连接成功了，就可以直接通信，无需多次连接。

**服务器客户端通用**

**接收消息**

BeginReceive

EndReceive

|  |
| --- |
| private byte[] resultBytes = new byte[1024]; //存储字节数组的容器  socketTcp.BeginReceive(resultBytes, 0, resultBytes.Length,SocketFlags.None, (result)=>{  try{  //这个返回值是你收到了多少个字节  int num = socketTcp.EndRecive(result);  //进行消息处理 假设  Encoding.UTF8.GetString(resultBytes, 0, num);  }  catch(SocketException e){  print(“接受消息出问题” + e.SocketErrorCode + e.Message );  }  }, socketTcp); |



如果要重复利用的话

|  |
| --- |
| private void ReceiveCallBack(IAsyncResult result){  try{  Socket s = result.AsyncState as Socket;  //这个返回值是你收到了多少个字节  int num =s.EndRecive(result);  //进行消息处理 假设  Encoding.UTF8.GetString(resultBytes, 0, num);    s.BeginReceive(resultBytes, 0,resultBytes.Length, SocketFlags.None, RecieveCallBack, s);  }  catch(SocketException e){  print(“接受消息出问题” + e.SocketErrorCode + e.Message );  }  } |

**发送消息**

BeginSend

EndSend

|  |
| --- |
| byte[] bytes = Encoding.UTF8.GetBytes(“1231313134”);  socketTcp.BeginSend(bytes, 0, bytes.Length, SocketFlags.None, (result) =>{  try{  socketTcp.EndSend(result); //返回值是int，代表发送了多少个字节。  }  catch(SocketException e){  print(“发送错误” + e.SocketErrorCode + e.Message);  }  }, socketTcp) |

* **回调函数概念**：等特定条件完成或满足，再执行的函数。
* **IAsyncResult result作为形参的作用**：是异步操作的核心状态与结果数据，让回调函数能知道***这个异步操作是谁发起的***，带了什么上下文。

下面的案例中，标黄色的部分是回调函数的主体，result是异步操作的回执单

|  |
| --- |
| socketTcp.BeginReceive(resultBytes, 0, resultBytes.Length,SocketFlags.None, (result)=>{  try{  //这个返回值是你收到了多少个字节  int num = socketTcp.EndRecive(result);  //进行消息处理 假设  Encoding.UTF8.GetString(resultBytes, 0, num);  }  catch(SocketException e){  print(“接受消息出问题” + e.SocketErrorCode + e.Message );  }  }, socketTcp); |

异步通信常用方式2

Socket Tcp通信中的异步方法2(Async 结尾方法)

关键变量类型

SocketAsyncEventArgs

它会作为Async异步方法的传入值

我们需要通过它进行一些关键参数的赋值

**服务器端**

AcceptAsync

|  |
| --- |
| **//SocketAsyncEventArgs 本质是一个委托**  SocketAsyncEventArgs e =new SocketAsyncEventArgs();  **//传入的socket就是当前连接的socket(socketTcp)**  **//传入的args 就是e**  e.Completed += (socket, args) =>{  if(args.SocketError == SocketError.Success){  **//获取连入的客户端socket**  Socket clientSocket = args.AcceptSocket;  **//继续监听下一个连入**  (socket as Socket).AcceptAsync(args);  }else  {  print(“连入客户端失败” + args.SocketError);  }  };  socketTcp.AcceptAsync(e); |

**客户端**

ConnectAsync

和客户端连接很像

|  |
| --- |
| SocketAsyncEventArgs e2 = new SocketAsyncEventArgs();  e2.Completed += (socket, args) =>{  if(args.SocketError == SocketError.Success){  **//连接成功**  }  else  {  **//连接失败**  print(args.SocketEvent);  }  };  socketTcp.ConnectAsync(e2); |

**服务端和客户端**

**发送消息**

SendAsync

|  |
| --- |
| SocketAsyncEventArgs e3 = new SocketAsyncEventArgs();  **//发送字节消息**  byte[] bytes2 = Encoding.UTF8.GetBytes(“我是消息”);  e3.SetBuffet(bytes2, 0, bytes2.Length);  **//添加 监听完成事件**  e3.Completed += (socket, args) =>{  if(args.SocketError == SocketError.Success){  print(“发送成功”);  }  else{    }  }  socketTcp.SendAsync(e3); |

**接收消息**

ReceiveAsync

|  |
| --- |
| SocketAsyncEventArgs e4 = new SocketAsyncEventArgs();  **//设置接收数据的容器，偏移位置，接收数据大小**  e4.SetBuffet(new byte[1024 \* 1024], 0, 1024\*1024);  e4.Completed +=(socket, args) =>{  if(args.SocketError == SocketError.Success){  **//收取存储在容器当中的字节**  **//Buffer是容器**  **//BytesTransferred 是收取了多少个字节**  Encoding.UTF8.GetString(args.Buffer, 0, args.BytesTransferred);  **//假设我们还要接收数据**  **//这次无需设置容器是谁，会接着上一个容器继续存储。在哪个位置，接收长度。**  args.SetBuffet(0, args.Buffer.Length);  **//继续接收下一条消息**  (socket as Socket).RecieveAsync(args);  }  else{    }  };  socketTcp.ReceiveAsync(e4); |

总结

C#中网络通信 异步方法 主要提供了两种方案

1. **Begin开头的API**

内部开多线程，通过回调形式返回结果，需要和End相关方法 配合使用。

1. **Async结尾的API**

内部开多线程，通过回调形式返回结果，依赖SocketAsyncEventArgs对象。***更加方便***。