Demo8-NetWork网络连接、发送、接收数据

2018年3月27日 17:10

关于NetWork这一块，我折腾了两个多星期了，因为每天只有1个多小时的时候，思路很不连续，一度想放弃。还好坚持下来了，其实NetWork这块的使用不难，只是我自己掉进坑里了。

没错，我要来和大家分享我的坑，以及演示NetWork从连接到发送数据，再到接收数据的处理。

1.大致流程

网络组件的使用大致使用流程如下：
a.创建频道（NetworkChannel）
b.绑定频道辅助器（NetworkChannelHelper）
c.注册频道的消息监听类（PacketHandler），即服务端发送某类消息过来时，客户端有对应的类进行接收处理
d.频道连接服务端
e.使用频道辅助器（NetworkChannelHelper）序列化消息对象，通过频道发送消息
f.服务端接收消息并回应客户端
f.消息监听类（PacketHandler）处理服务端回应的消息

在使用上，大致就是这么个逻辑，真的很简单，我也不知道自己是怎么掉坑里的。
由于本Demo的代码较多，不适合把每个细节都贴出来讲解，请大家结合本篇文章查看源码（Demo8）

2.频道（NetworkChannel）

频道是和服务端通信的基础，我们可以创建很多个频道，如聊天频道、战斗频道，不同的频道连接不同服务器，适用于有多个逻辑服务器的游戏。

创建频道的同时，我们需要把频道辅助器也一起创建，如：

```
private GameFramework.Network.INetworkChannel m_Channel;
private NetworkChannelHelper m_NetworkChannelHelper;

// 获取框架网络组件
NetworkComponent Network
    = UnityGameFramework.Runtime.GameEntry.GetComponent<NetworkComponent> ();

// 创建频道
m_NetworkChannelHelper = new NetworkChannelHelper ();
m_Channel = Network.CreateNetworkChannel ("testName", m_NetworkChannelHelper);
```

使用Network组件的CreateNetworkChannel函数即可创建频道，第一个参数是频道名称，第二个参数就是频道辅助器。

具体代码看Demo8_ProcedureLaunch的OnEnter函数。

3.连接服务器

连接服务器很简单，创建好频道后，调用Connet函数即可：

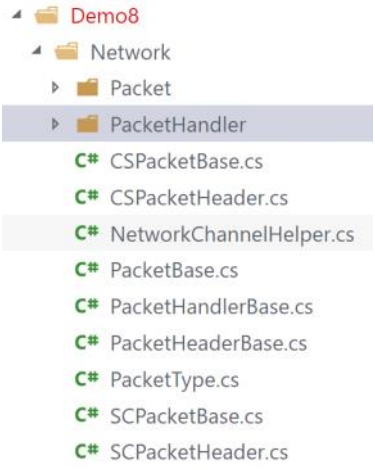
```
// 连接服务器
m_Channel.Connect (IPAddress.Parse ("127.0.0.1"), 8098);
```

至于服务端，木头已经给大家写（抄）了一个了（Demo8_SocketServer），同样是在Demo8_ProcedureLaunch的OnEnter函数可以看到启动服务端的代码。
至于服务端代码的细节，大家可以忽略。

4.频道辅助器（NetworkChannelHelper）

其实要理解Network的使用方式，关键就是理解这个辅助器了，木头也就是在这里掉进坑里的。
频道辅助器是可以自定义的，只要实现了INetworkChannelHelper接口即可。

当然，为了学习使用方式，我当然是去官方的StarForce项目里找例子了，于是我把StarForce里的NetworkChannelHelper拷过来用了，包括其它的一些必备类，有这些：



其它的文件先不管，NetworkChannelHelper有三个关键的函数：

```
/// <summary>
/// 序列化消息包。
/// </summary>
/// <typeparam name="T">消息包类型。</typeparam>
/// <param name="packet">要序列化的消息包。</param>
/// <returns>序列化后的消息包字节流。</returns>
1 reference
public byte[] Serialize<T> (T packet) where T : Packet {…
}

/// <summary>
/// 反序列消息包头。
/// </summary>
/// <param name="source">要反序列化的来源流。</param>
/// <param name="customErrorData">用户自定义错误数据。</param>
/// <returns></returns>
1 reference
public IPacketHeader DeserializePacketHeader (Stream source, out object customErrorData) {…
}

/// <summary>
/// 反序列化消息包。
/// </summary>
/// <param name="packetHeader">消息包头。</param>
/// <param name="source">要反序列化的来源流。</param>
/// <param name="customErrorData">用户自定义错误数据。</param>
/// <returns>反序列化后的消息包。</returns>
1 reference
public Packet DeserializePacket (IPacketHeader packetHeader, Stream source, out object customErrorData) {…
}
```

Serialize函数：用于序列化消息，这里用的是Protobuf。
DeserializePacketHeader函数：用于反序列化消息头。
DeserializePacket函数：用于反序列化消息内容（不含消息头）。

不知道为什么，如果直接用StarForce这个项目的NetworkChannelHelper，不做改动的话，无论如何都无法用好这几个函数，反序列化总是出错。

木头本人没用过Protobuf，所以也是一边研究一边了解这个东西。
同时，一开始也不是很了解E大（框架作者）的思路，我硬是想要在不改动这个helper类的前提下，把demo跑通。

我失败了无数次，在无数次的失败过程中，稍微熟悉了Protobuf，以及理解了（也可能是误解？）E大的思路。

改，必须得改。
再此，我也求助各位，如果我的改动是错误的，或者思路错了，希望大家能给我指点一二，非常感谢。

5.消息包（Packet）

在序列化消息之前，必须得有消息包。

Packet类是框架的消息包抽象类，我们的消息包类都要继承它。
而在StarForce的项目了，将消息类型区分为了服务端消息和客户端消息：

```
namespace StarForce
{
    12 references
    public enum PacketType : byte
    {
        /// <summary>
        /// 未定义。
        /// </summary>
        1 reference
        Undefined = 0,

        /// <summary>
        /// 客户端发往服务器的包。
        /// </summary>
        3 references
        ClientToServer,

        /// <summary>
        /// 服务器发往客户端的包。
        /// </summary>
        2 references
        ServerToClient,
    }
}
```

这个倒不是必须有的，但我依旧按照框架作者的思路来研究。

因此，对于服务端要解析的消息包和客户端要解析的消息包就要区分开来，所以有了CSPacketBase和SCPacketBase类。

分别表示从客户端发到服务端的消息、从服务端发到客户端的消息：

```
public abstract class CSPacketBase : PacketBase
{
    1 reference
    public override PacketType PacketType
    {
        get
        {
            return PacketType.ClientToServer;
        }
    }
}
```

```
public abstract class SCPacketBase : PacketBase
{
    1 reference
    public override PacketType PacketType
    {
        get
        {
            return PacketType.ServerToClient;
        }
    }
}
```

6.消息头（PacketHeader）

刚刚说的只是消息的内容，按照框架作者的思路，我们还需要有一个消息头。

消息头的作用是告诉消息接收方，当前的消息是什么类型的消息，消息长度是多少。

而我就是在这里陷入了无数次的失败循环，在后面会说到。

既然消息包是区分类型的，那消息头自然也要区分，所有有SCPacketHeader和CSPacketHeader，如：

```
[Serializable, ProtoContract(Name = @"SCPacketHeader")]
7 references | 钟迪龙, 20 days ago | 1 author (钟迪龙)
public sealed class SCPacketHeader : PacketHeaderBase
{
    /* 注意,ProtoMember是木头加上的,以便可以使用protobuf序列化 */
    [ProtoMember (1)]
    6 references
    public override int Id
    {
        get;
        set;
    }

    /* 注意,ProtoMember是木头加上的,以便可以使用protobuf序列化 */
    [ProtoMember (2)]
    6 references
    public override int PacketLength
    {
        get;
        set;
    }

    1 reference
    public override PacketType PacketType
    {
        get
        {
            return PacketType.ServerToClient;
        }
    }
}
```

注意上面代码里的注释，这是我掉入的第一个坑。
不知道是不是我误解了框架作者的思路，总之，如果我不在消息头的Id和PacketLength对象里加入ProtoMember特性的话，是无法将这两个字段进行序列化的。
包括这个类头上的ProtoContract特性也是我加的，否则序列化时会报错（UnExpectedType）。

大家也看到了，消息头主要有一个Id字段和PacketLength字段。
Id字段用来指定该消息是什么类型，它对应的是消息包（Packet）的Id字段。
PacketLength用来指定消息包的长度，以便解析。

来总结一下消息头和消息包：
发送消息时，会将消息头和消息包一起序列化后进行发送，而消息头在序列化前会将Id赋值为消息包的Id，将PacketLength赋值为消息包序列化后的长度。

接收消息时，先解析消息头，根据消息头的Id定位消息包类型，然后根据消息头的PacketLength解析消息包内容。

7.NetworkChannelHelper的序列化操作

接下来就是木头遇到的第二个坑——Serialize函数。

一开始的Serialize函数是长这样的：

```
/// <summary>
/// 序列化消息包。
/// </summary>
/// <typeparam name="T">消息包类型。</typeparam>
/// <param name="packet">要序列化的消息包。</param>
/// <returns>序列化后的消息包字节流。</returns>
1 reference
public byte[] Serialize<T>(T packet) where T : Packet
{
    PacketBase packetImpl = packet as PacketBase;
    if (packetImpl == null)
    {
        Log.Warning("Packet is invalid.");
        return null;
    }

    if (packetImpl.PacketType != PacketType.ClientToServer)
    {
        Log.Warning("Send packet invalid.");
        return null;
    }

    // 恐怖的 GCAlloc，这里是例子，不做优化
    using (MemoryStream memoryStream = new MemoryStream())
    {
        CSPacketHeader packetHeader = ReferencePool.Acquire<CSPacketHeader>();
        Serializer.Serialize(memoryStream, packetHeader);
        Serializer.SerializeWithLengthPrefix(memoryStream, packet, PrefixStyle.Fixed32);
        ReferencePool.Release(packetHeader);

        return memoryStream.ToArray();
    }
}
```

CSPacketHeader是客户端发送给服务端的消息头，这里仅仅是创建了一个消息头，然后序列化，接着就把消息包也序列化了。我没有看到任何设置消息头Id和packetLength的操作，我以为有什么厉害的地方自动做了这些事情。

但是，这里的Serializer.Serialize(memoryStream, packetHeader)执行是报错的，这个刚刚有提过，因为CSPacketHeader没有指定ProtoContract特性，这样是无法用Protobuf序列化的，所以我给CSPacketHeader加上了这个特性。我不知道我的做法是否有误，**如果有错误的地方，希望大家能指正**。

加了特性之后是能顺利序列化了，但是，消息头依然没有进行任何赋值操作。
并且，这样序列化的消息头，在解析时也会报错。

最终，我自行给消息头赋值，并且改用SerializeWithLengthPrefix进行序列化，整个流程才得以跑通：

```
// 恐怖的 GCAlloc，这里是例子，不做优化(这句注释是框架作者写的,我本人并不懂GC什么的)
using (MemoryStream memoryStream = new MemoryStream ()) {
    /* 以下内容为本头本人做的改动,不知道是否有错误的地方(虽然它运行起来是正确的),希望大家能帮忙指正 */
    // 因为头部消息有8字节长度，所以先跳过8字节
    memoryStream.Position = 8;
    Serializer.SerializeWithLengthPrefix (memoryStream, packet, PrefixStyle.Fixed32);

    // 头部消息
    CSPacketHeader packetHeader = ReferencePool.Acquire<CSPacketHeader> ();
    packetHeader.Id = packet.Id;
    packetHeader.PacketLength = (int) memoryStream.Length - 8; // 消息内容长度需要减去头部消息长度
```

```
// 恐怖的 GCAlloc，这里是例子，不做优化(这句注释是框架作者写的,我本人并不懂GC什么的)
using (MemoryStream memoryStream = new MemoryStream ()) {
    /* 以下内容为木头本人做的改动,不知道是否有错误的地方(虽然它运行起来是正确的),希望大家能帮忙指正 */
    // 因为头部消息有8字节长度，所以先跳过8字节
    memoryStream.Position = 8;
    Serializer.SerializeWithLengthPrefix (memoryStream, packet, PrefixStyle.Fixed32);

    // 头部消息
    CSPacketHeader packetHeader = ReferencePool.Acquire<CSPacketHeader> ();
    packetHeader.Id = packet.Id;
    packetHeader.PacketLength = (int) memoryStream.Length - 8; // 消息内容长度需要减去头部消息长度

    memoryStream.Position = 0;
    Serializer.SerializeWithLengthPrefix (memoryStream, packetHeader, PrefixStyle.Fixed32);

    ReferencePool.Release (packetHeader);

    return memoryStream.ToArray ();
}
```

再一次求助，我对Protobuf的研究非常的浅（时间精力关系），如果这里也有错误的地方，希望大家能打我脸。

头部消息的8字节是怎么来的呢?
因为CSPackertHeader的Id和PacketLength两个字段（int类型）是参与了序列化的，两个int类型是8个字节。

由于消息头要记录消息内容的长度，所以，我的做法是先把消息内容序列化，然后再获取它的长度。
在序列化消息内容前，先让Position跳过8，给后面的消息头预留位置，这样消息头后续就能插到流的最前面。

另外，NetworkChannelHelper的PacketHeaderLength需要返回8（因为消息头的长度是8）。
在客户端接收到消息的时候，会根据这里设置长度来获取消息头。

```
/// <summary>
/// 获取消息包头长度。
/// </summary>
3 references
public int PacketHeaderLength {
    get {
        return 8;
    }
}
```

8.NetworkChannelHelper的反序列化

由于消息头的序列化方式改了，所以NetworkChannelHelper反序列化消息头的操作也要改动：

```
/// <summary>
/// 反序列消息包头。
/// </summary>
/// <param name="source">要反序列化的来源流。</param>
/// <param name="customErrorData">用户自定义错误数据。</param>
/// <returns></returns>
1 reference
public IPacketHeader DeserializePacketHeader (Stream source, out object customErrorData) {
    // 注意：此函数并不在主线程调用！
    customErrorData = null;

    return Serializer.DeserializeWithLengthPrefix<SCPacketHeader> (source, PrefixStyle.Fixed32);
    // return (IPacketHeader)RuntimeTypeModel.Default.Deserialize(source, ReferencePool.Acquire<SCF
}
```

这里的改动如果有错误的话，也是希望大家能指出的。

客户端收到消息时，会先调研DeserializePacketHeader函数解析消息头。
随后再调研DeserializePacket函数解析消息内容，在这一步，消息头的Id就很有用了，我们要根据Id来区分这个消息的类型，然后再做这个类型的反序列化。

至于Id是怎么区分出消息类型的，这个就要看NetworkChannelHelper的Initialize函数了，具体大家看代码，这个函数主要做了以下的事情：
a.将继承了SCPacketBase的类型的类Id和类型保存到一个Dictionary里，以便在DeserializePacket函数里可以根据消息头Id获取消息类型。至于为什么只保存SCPacketBase，因为我们客户端只需要接收来自服务端的消息（SCPacketBase就是来自服务端的消息包类）
b.注册继承了PacketHandlerBase的类，也就是很前面提到的消息监听类
c.订阅一些事件

9.消息监听类（PacketHandler）

最后但可能是最重要的一个，就是消息监听类，能发送消息，能接收消息，最终要做的当然就是对消息的处理。
消息监听类就是用来处理不同消息的类，因此，我们可能会有很多很多这样的类。

上一步提到，在NetworkChannelHelper的Initialize函数里会注册消息监听类——也就是那些继承了PacketHandlerBase的类。
比如我们的Demo8_HelloPacketHandler：

```
using GameFramework;
using GameFramework.Network;
using StarForce;
public class Demo8_HelloPacketHandler : PacketHandlerBase {
    public override int Id {
        get {
            return 10;
        }
    }

    public override void Handle (object sender, Packet packet) {
        SCHello packetImpl = (SCHello) packet;
        Log.Info ("Demo8_HelloPacketHandler 收到消息: '{0}'.", packetImpl.Name);
    }
}
```

这里的Id也是有讲究，一定要和消息类的Id对应，比如我们这个Handler是用来处理Hello消息的，所以Id必须和SCHello类的Id一致：

```
public class SCHello : SCPacketBase {
    public override int Id {
        get {
            return 10;
        }
    }

    [ProtoMember (1)]
    public string Name { get; set; }
    public override void Clear () {
    }
}
```

只有Id一致才能正确处理对应的消息。

10.结束

在Network组件上，我大致就是遇到了这些问题，其余的地方，大家看代码就可以了（其实也没有多少代码了）。

服务端代码大家不用管，它在接收到客户端消息后，会回发一条SCHello的消息过来。

大家拉了代码后，运行Demo8的Demo8_Launch场景，若能看到下面的日志，就代表成功了：