Unity 网络多玩家游戏开发教程(上册)

（内部资料）

非卖品

大学霸
www.daxueba.net

前言

多玩家通过联网的方式共同进行的游戏被称为网络游戏。网络游戏由于具备更强的娱乐性和对战性成为游戏的主流。在这个互联互通的时代，多人联网玩同一个游戏，已经成为大众的习惯。Unity 作为强大的游戏开发平台，为网络游戏的的开发提供大量的组件和 API。同时，也涌现了大量的第三方的插件。

网络游戏涉及网络构建、通信方式、数据带宽、数据逻辑同步等多方面问题。本教程针对这些热点问题，重点讲解 Unity 游戏开发中的常见技术、插件等。内容包括：

- Unity 自带网络组件 Network View（上册）
- 第三方组件网络功能插件 PUN（上册）
- Yahoo 专向技术 Play.IO（上册）
- 第三方 SDK PubNub（下册）
- 实体差值和预测（下册）
- 击中检测（下册）

1.学习所需的系统和软件

- 安装 Windows 7 操作系统
- 安装 Unity 4.x

2.学习建议

大家学习之前，可以致信到 xxxxxxxxxx，获取相关的资料和软件。如果大家在学习过程遇到问题，也可以将问题发送到该邮箱。我们尽可能给大家解决。

3.特别声明

因为篇幅有限，书中在讲解代码时很难细致到每行代码，如有疏漏可能会造成读者制作效果与书中所讲不一致的情况，本书附带源代码，此时应该以代码为准。

大学霸

目　录

第 1 章　Unity 自带网络功能——实例：乒乓球游戏

Unity 拥有大量的第三方插件，专门提供了对网络功能的支持。但是，大部分开发者第一次接触到的还是 Unity 自带的网络功能，也就是大家经常说到的 Unity Networking API。这些 API 是借助于组件 NetworkView 发挥作用的，而它可以简化开发者大量的网络功能编码任务。

1.1　实现机制

Unity 提供了名为 NetworkView 的组件，拥有此组件的对象才可以向网络传输、或者接收数据。本节会详解介绍此组件的各属性，及其向网络传输、接收数据的方法。图 1-1 所示，是本节要介绍的内容的示意图。

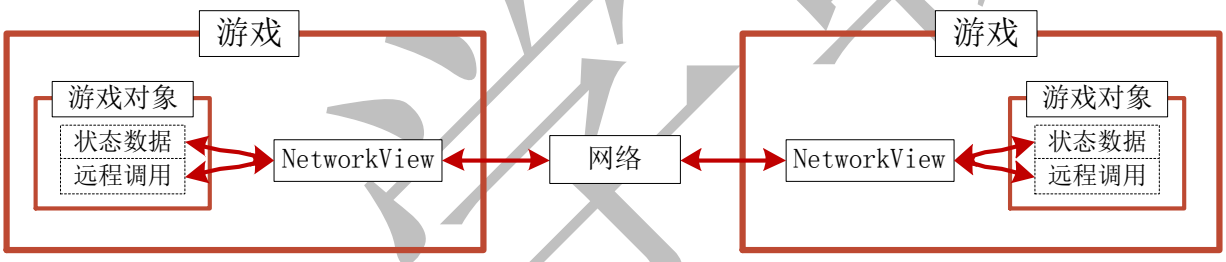

图 1-1　NetworkView 组件的用途

1.1.1　NetworkView 组件

在 Unity 中，NetworkView 组件用于处理游戏在网络上的数据传输，通常负责具体的游戏对象的状态数据串行化（state serialization）。例如，将游戏对象上 Transform、Rigidbody 和 Animation 组件的数据串行化到网络上。

为游戏对象添加 NetworkView 组件的方法是，单击 Component|Miscellaneous|Network View 命令，此组件如图 1-2 所示。

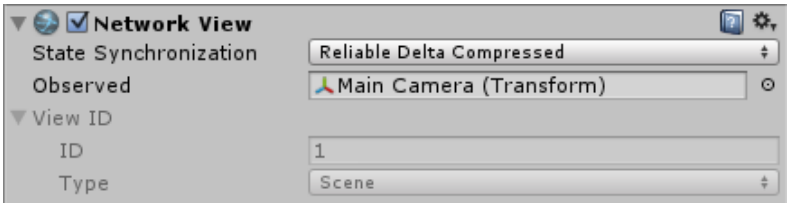

图 1-2　NetworkView 组件

提示：此组件被添加到了游戏场景中默认创建的对象 Main Camera 上。

❑ State Synchronization：表示要同步（串行化）的状态信息；

提示：此属性有 3 个可选项。Off 表示不需要同步任何状态；Reliable Delta Compressed 表示使用可信任的数据差异压缩传输方式，且只传输变化了的数据，若没有变化，那么就不会传输任何数据；Unreliable 表示要使用不信任的数据传输方式，每次都传输全部的数据，即时数据没有发生变化（需要占用更大的带宽）。

❑ Observed：表示要同步的信息的类型；

提示：示例中要同步的信息的类型是 Transform 组件上的数据。

❑ ID：唯一的标识特定对象上的 Network view 组件；

提示：多人网络游戏中，同一对象的此 ID 值是一致的。

1.1.2 自定义串行化数据

上一节讲解了使用 NetworkView 组件指定串行化数据的方法。例如，数据类型是 Transform 组件上的数据，就要为 NetworkView 组件上的 Observed 属性指定为特定对象上的 Transform 组件。而本节就来说明在脚本中自定义串行化数据的方法，数据的类型同样是 Transform 组件上的数据。

在 Project 视图里，新建 C#脚本，命名为 ExampleUnityNetworkSerializePosition，并添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ExampleUnityNetworkSerializePosition : MonoBehaviour
05  {
06      public void OnSerializeNetworkView( BitStream stream, NetworkMessageInfo info )
07      {
08          //判断当前是否打算向网络上写入数据
09          if( stream.isWriting )
10          {
11              //写入对象的位置信息
12              Vector3 position = transform.position;
13              stream.Serialize( ref position );
14          }
15          //当前正在从网络上读取数据
16          else
17          {
18              //将读取到的数据存储到 Vector3 类型的变量中
19              Vector3 position = Vector3.zero;
20              stream.Serialize( ref position );
21              //实时修改对象的位置
22              transform.position = position;
23          }
24      }
25  }
```

❑ 脚本 06 行，OnSerializeNetworkView()方法会被 Unity 主动调用，用于串行化自定义的数据；

❑ BitStream 类表示串行化的数据，它有两个变量 IsReading 和 IsWriting，分别用于判断当前处于读取，还是写入状态；

- BitStream 类依据当前的读取，或者写入状态，调用自己的 Serialize()方法从流中读取数据，或者写入数据到流中；

1.1.3 使用远程过程调用

远程过程调用（RPC，Remote Procedure Calls）的作用是，为了达到多人游戏的同步效果，调用远程机器上，特定对象上的方法。例如，玩家自己的角色在当前的机器上发射了子弹，那么处于同一游戏中的其它玩家，也应该在它们的机器上，看到那个角色发射了子弹。本节就来说明在脚本中使用远程过程调用的方法。

在 Project 视图里，新建 C#脚本，命名为 ExampleUnityNetworkCallRPC，并添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ExampleUnityNetworkCallRPC : MonoBehaviour
05  {
06      void Update()
07      {
08          //如果此 NetworkView 组件不属于此对象，就不做任何反应
09          if( !networkView.isMine )
10              return;
11          //当按下空格键的时候，调用所有同一游戏中，同一对象上的 testRPC 方法
12          if( Input.GetKeyDown( KeyCode.Space ) )
13              networkView.RPC( "testRPC", RPCMode.All );
14      }
15      [RPC]
16      void testRPC( NetworkMessageInfo info )
17      {
18          //记录调用此方法的电脑所在的 IP 地址
19          Debug.Log( "Test RPC called from " + info.sender.ipAddress );
20      }
21  }
```

- 远程过程调用就是使用 networkView.RPC()方法，第一个参数表示要远程调用的方法名，第二个参数表示接收远程调用的主机；
- 脚本 15 行的属性标记是必不可少的；

提示：networkView.RPC()方法的第二个参数，有 5 个可选项，如图 1-3 所示，分别表示只发送给服务器、除发送者以外的所有主机、除发送者以外的所有主机并添加到缓存、所有主机，以及所有主机并添加到缓存。

图 1-3　networkView.RPC()方法第二个参数的可选项

1.2　默认服务器机制

游戏本身需要在网络上创建一个服务，然后其它游戏才能连接到这个服务，进而实现在同一游戏场景中一同玩耍的效果，如图 1-4 所示。

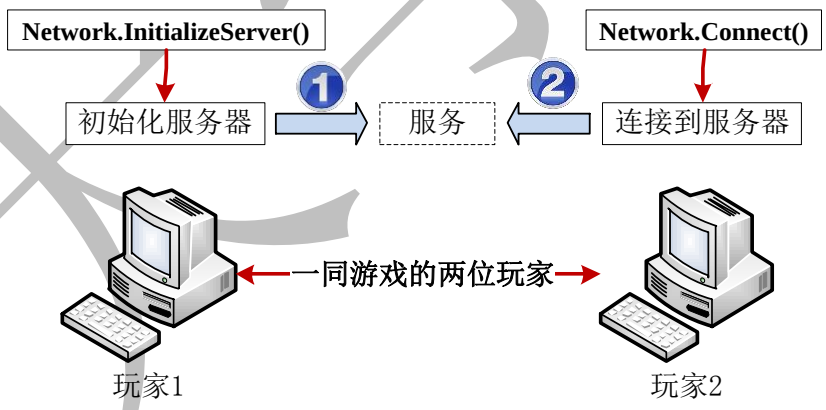

图 1-4　默认服务器机制

1.2.1　初始化服务器

对于游戏本身而言，无论是玩家打算创建一个新的游戏也好，还是打算加入其中一个游戏也罢，游戏本身都必须先在本机上初始化服务器。

在 Project 视图里，新建 C#脚本，命名为 ExampleUnityNetworkInitializeServer，并添加下面的代码：

```
01    using UnityEngine;
```

```
02  using System.Collections;
03
04  public class ExampleUnityNetworkInitializeServer : MonoBehaviour
05  {
06      void OnGUI()
07      {
08          //在游戏视图上绘制一个按钮，当此按钮被单击的时候，开始启动服务器
09          if( GUILayout.Button( "Launch Server" ) )
10          {
11              LaunchServer();
12          }
13      }
14      //启动服务器
15      void LaunchServer()
16      {
17          //接受来自 8 个主机的连接，监听的端口号是 25005，支持 NAT 穿透
18          Network.InitializeServer( 8, 25005, true );
19      }
20      // Network.InitializeServer()调用完毕后，主动调用此方法
21      void OnServerInitialized()
22      {
23          Debug.Log( "Server initialized" );
24      }
25  }
```

- 脚本 18 行，方法 Network.InitializeServer()用于初始化服务器，需要传入 3 个参数：第一个参数表示允许连接到此服务器上的主机的数目；第二个参数表示要监听的端口号；第三个参数表示是否支持 NAT 穿透；
- 脚本 21 行，方法 OnServerInitialized()会在 Network.InitializeServer()调用完毕后，被主动调用；

将脚本 ExampleUnityNetworkInitializeServer 赋予游戏场景的 Main Camera 对象，然后运行游戏，效果如图 1-5 所示。当单击了游戏视图上的 Launch Server 按钮以后，完成服务器的初始化之后就会在 Console 视图上显示“Server initialized”信息。

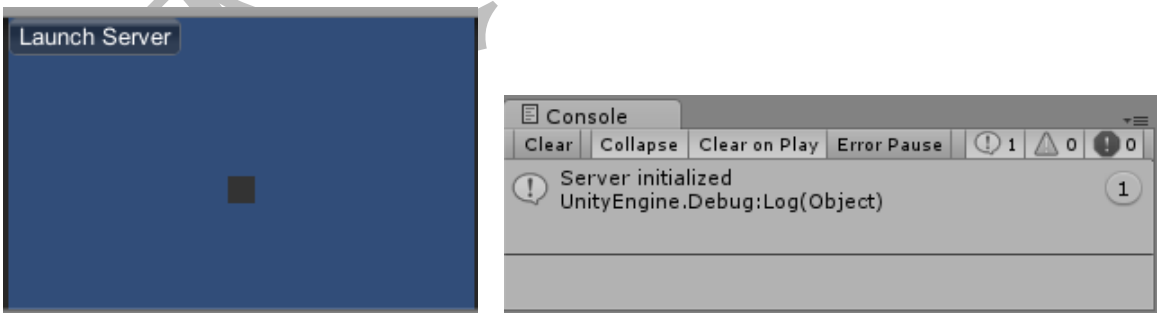

图 1-5　初始化服务器效果展示

1.2.2　连接到服务器

上一节使用脚本初始化了一个服务器，而本节会编写一个脚本，用于连接上一节创建的那个服务器，具体的操作步骤如下：

（1）在 Project 视图里，新建 C#脚本，命名为 ExampleUnityNetworkingConnectToServer，并添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ExampleUnityNetworkingConnectToServer : MonoBehaviour
05  {
06      private string ip = "";
07      private string port = "";
08      private string password = "";
09      void OnGUI()
10      {
11          //在游戏视图上绘制 IP 地址输入区域
12          GUILayout.Label( "IP Address" );
13          ip = GUILayout.TextField( ip, GUILayout.Width( 200f ) );
14          //在游戏视图上绘制端口号输入区域
15          GUILayout.Label( "Port" );
16          port = GUILayout.TextField( port, GUILayout.Width( 50f ) );
17          //在游戏视图上绘制密码输入区域
18          GUILayout.Label( "Password (optional)" );
19          password = GUILayout.PasswordField( password, '*', GUILayout.Width( 200f ) );
20          //当按钮被点击后，开始连接指定的服务器
21          if( GUILayout.Button( "Connect" ) )
22          {
23              int portNum = 25005;
24              if( !int.TryParse( port, out portNum ) )
25              {
26                  Debug.LogWarning( "Given port is not a number" );
27              }
28              //使用获取到的 IP 和端口号直接连接服务器
29              else
30              {
31                  Network.Connect( ip, portNum, password );
32              }
33          }
34      }
35      //连接到服务器以后，被 Unity 主动调用
36      void OnConnectedToServer()
37      {
38          Debug.Log( "Connected to server!" );
39      }
40      //连接服务器操作失败后，被 Unity 主动调用
41      void OnFailedToConnect( NetworkConnectionError error )
42      {
43          Debug.Log( "Failed to connect to server: " + error.ToString() );
44      }
45  }
```

- 脚本 06~08 行，分别声明了用于表示 IP 地址、端口号和密码的变量；
- 脚本 09~34 行，方法 OnGUI()用于在界面上绘制对应的 UI 控件，作为玩家输入指定信息的接口。
- 脚本 31 行，方法 Network.Connect()用于使用获取到的 IP 和端口号直接连接服务器；

（2）上一节，我们为游戏场景中的 Main Camera 对象，赋予了

ExampleUnityNetworkInitializeServer 脚本，作用是初始化服务器。现将其单独编译成可执行文件，方法是单击 File|Build Settings…命令，在打开的 Build Settings 对话框中，添加当前的游戏场景到 Scenes In Build 列表区域，再单击 Build And Run 即可，如图 1-6 所示。

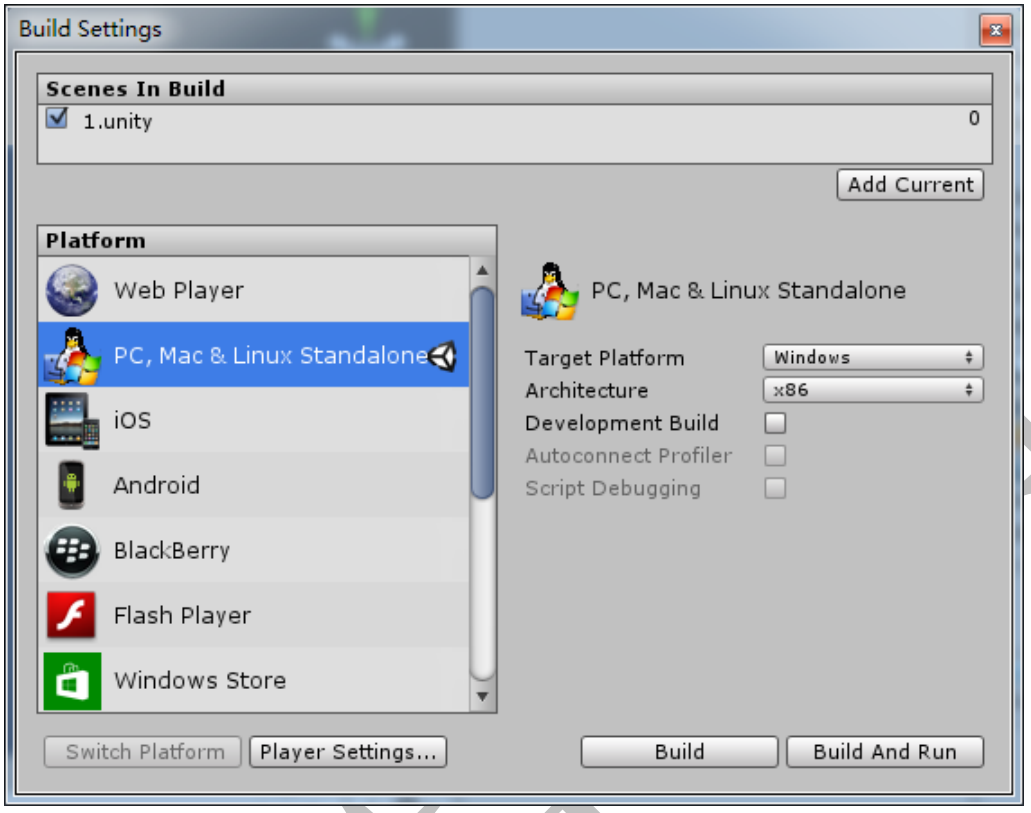

图 1-6　Build Settings 对话框

（3）第（2）步的操作，初始化并启动了一个服务器，它监听的端口号是 25005，没有任何密码。回到 Unity 编辑器中，选中 Main Camera 对象，移除其上的 ExampleUnityNetworkInitializeServer(Script) 组件，然后赋予本节编写的脚本 ExampleUnityNetworkingConnectToServer，并运行游戏，在游戏视图上输入 IP 地址和，端口号以后，就可以连接到那个服务器了，效果如图 1-7 所示。

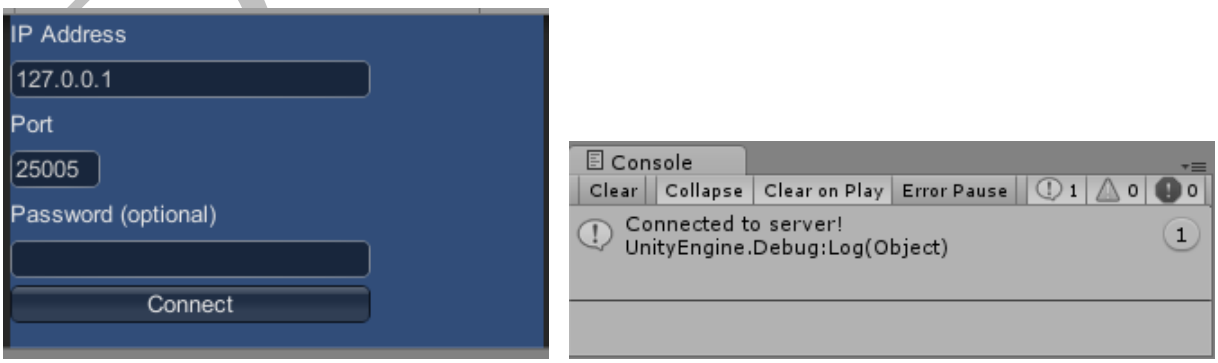

图 1-7　连接到服务器的效果展示

1.3 自定义服务器机制

本章前面两节，涉及服务器的操作，都是使用了 Unity 默认指定的主服务器（Master Server）和通讯服务主体（Facilitator）来完成的。但在实际的开发中，建议读者使用自己的主服务器和通讯服务主体。本节就来说明，设置自己的主服务器的方法，如图 1-8 所示。

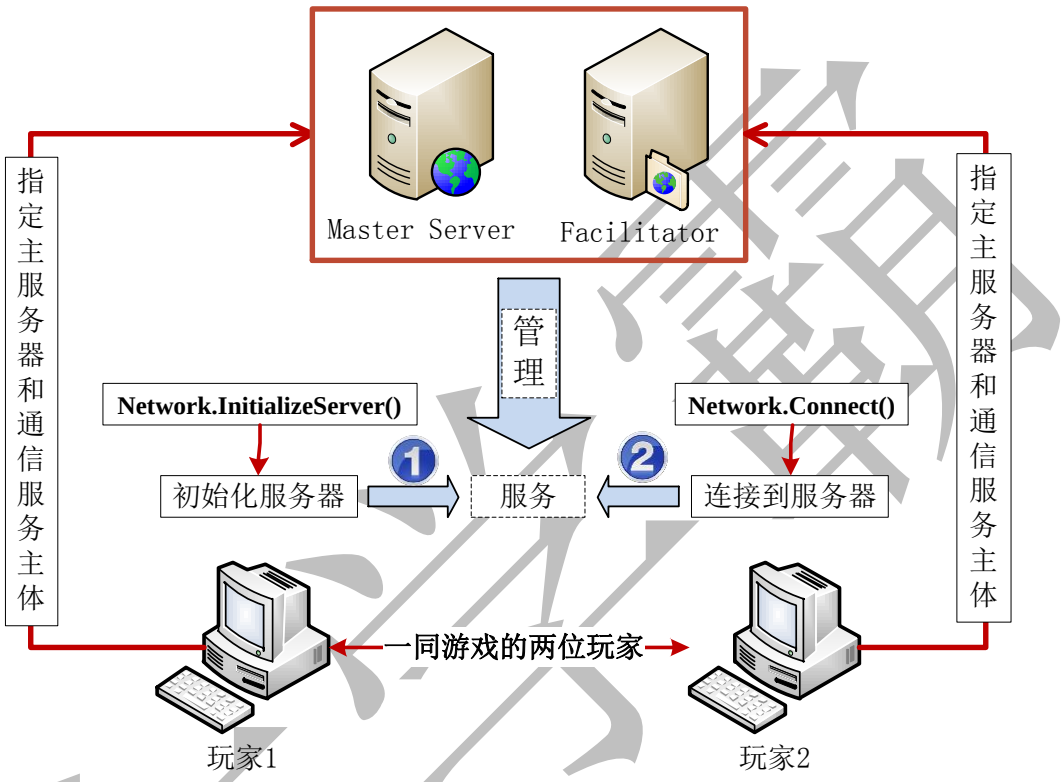

图 1-8 自定义服务器机制

1.3.1 设置主服务器

Master Server 的作用是显示当前所有被初始化了的服务器，而 Facilitator 的作用是辅助各客户端连接彼此，使用的技术是 NAT 穿透（NAT punch-through）。在了解了这些以后，本小节接下来就要介绍设置主服务的方法了，如下：

（1）输入网址：http://unity3d.com/master-server/，进入如图 1-9 所示的网页，下载 Unity 提供的 Master Server 和 Facilitator 组件。

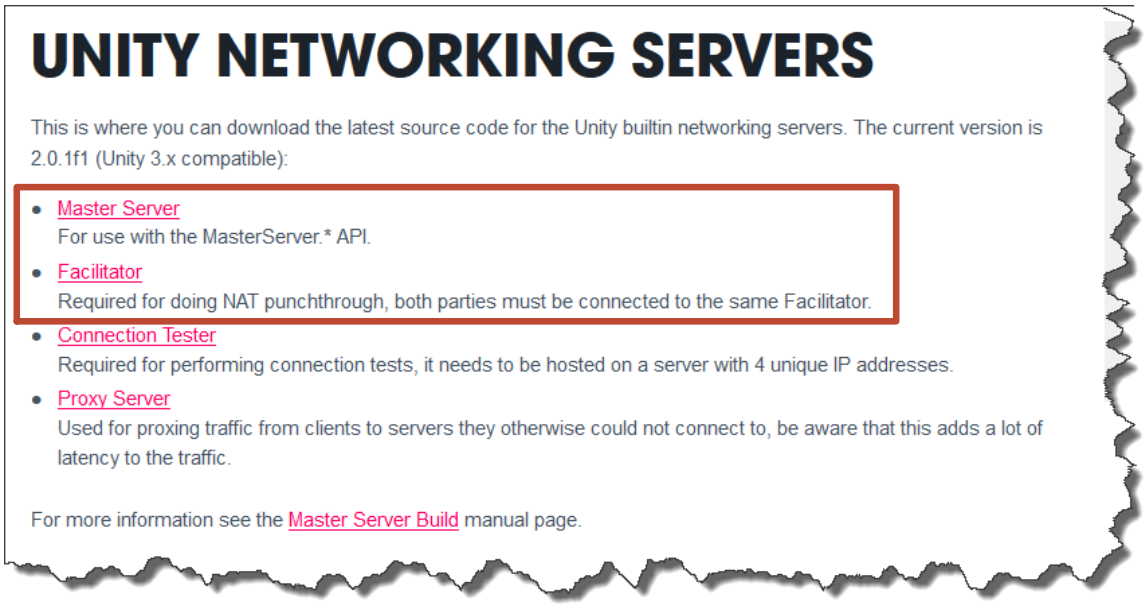

图 1-9　Unity Networking Servers 页面

（2）下载到的两个压缩文件如图 1-10 所示。压缩文件内部是未编译的源码，包含了 3 种类型的项目文件夹，如图 1-11 所示。读者应该依据当前所做开发时使用的操作系统环境，决定要编译哪一个项目文件。

提示：详情请参看网址：http://docs.unity3d.com/Manual/net-MasterServerBuild.html。

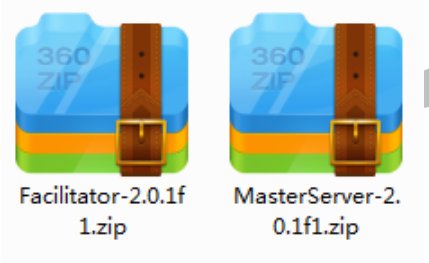

图 1-10　下载到的压缩文件

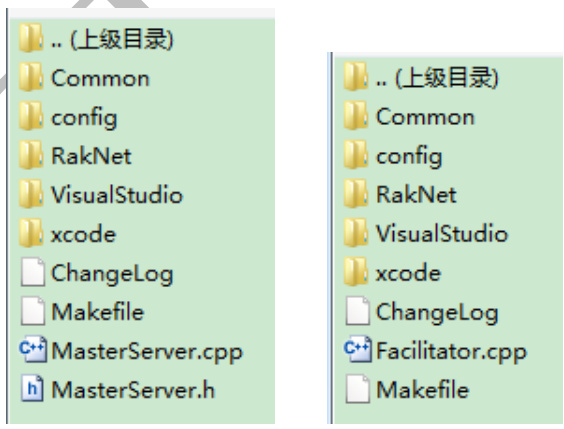

图 1-11　压缩文件中的 3 种项目文件夹

（3）本书写作时，使用的操作系统平台是 Windows，因此需要编译的是 Visual Studio 文件夹里的项目文件，使用的编译器是 Microsoft Visual C++ 2010 Express，此编译器是个免费的版本，下载网址是：http://www.visualstudio.com/downloads/download-visual-studio-vs，页面如图 1-12 所示。

图 1-12　Microsoft Visual C++ 2010 Express 下载页面

（4）使用 Release，而非 Debug 模式编译项目文件，最后得到的两个 EXE 文件如图 1-13 所示。它们就是我们自己的主服务器和通讯服务主体。

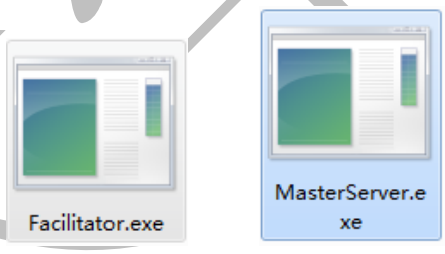

图 1-13　编译后得到的 EXE 文件

（5）双击打开这两个可执行文件，如图 1-14 所示。可知默认情况下，MasterServer 使用的端口是 23466，而 Facilitator 使用的端口是 50005。

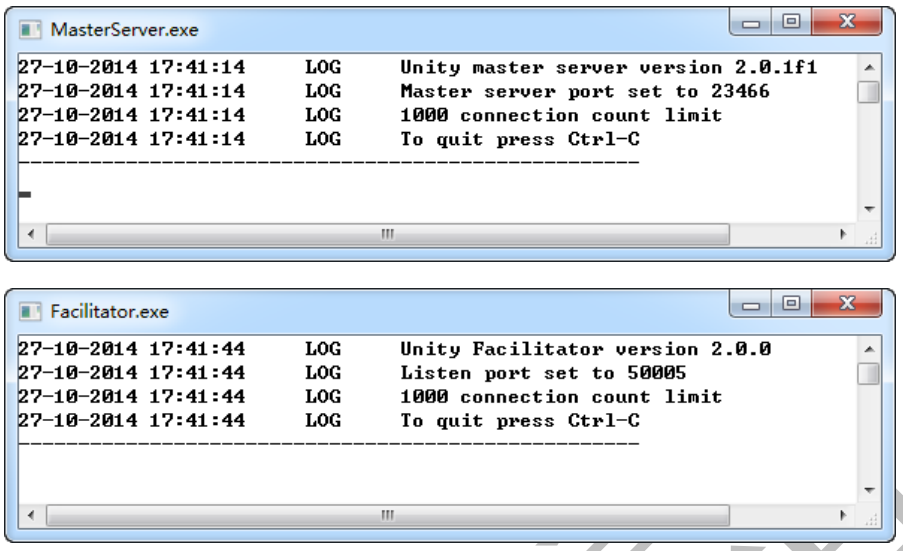

图 1-14　主服务器和通讯服务主体

1.3.2　连接到自定义的主服务器

本节将会编写脚本，使得游戏程序顺利连接到上一节下载到的自定义的主服务器，而非 Unity 默认指定的主服务器上。

（1）在当前的电脑上运行上一节编译得到的两个可执行文件 MasterServer.exe 和 Facilitator.exe。因为是在当前的电脑上运行的，所以它们的 IP 就是 127.0.0.1。

提示：如果 MasterServer.exe 和 Facilitator.exe 是运行在其它的电脑上的，就需要知道那台电脑的 IP 地址。

（2）在 Project 视图里，新建 C# 脚本，命名为 ExampleUnityNetworkingConnectToMasterServer，并添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ExampleUnityNetworkingConnectToMasterServer : MonoBehaviour
05  {
06      //假定主服务器和通讯服务主体都运行在本机上
07      public string MasterServerIP = "127.0.0.1";
08      void Awake()
09      {
10          //指定主服务器的 IP 地址和端口号
11          MasterServer.ipAddress = MasterServerIP;
12          MasterServer.port = 23466;
13          //指定通讯服务主体的 IP 地址和端口号
14          Network.natFacilitatorIP = MasterServerIP;
15          Network.natFacilitatorPort = 50005;
16      }
17  }
```

- 脚本 10~15 行，明确的指定了自定义主服务的 IP 地址和端口号。如此一来，再连接服务器的时候，就不会在使用 Unity 默认指定的服务器了。

（3）在本章“初始化服务器”一节，曾编写过一个脚本

ExampleUnityNetworkInitializeServer ，现将其与脚本 ExampleUnityNetworkingConnectToMasterServer，一同赋予游戏场景中的 Main Camera 对象，然后运行游戏，效果如图 1-15 所示。说明新的连接已经建立，来自于当前主机的 25005 端口。

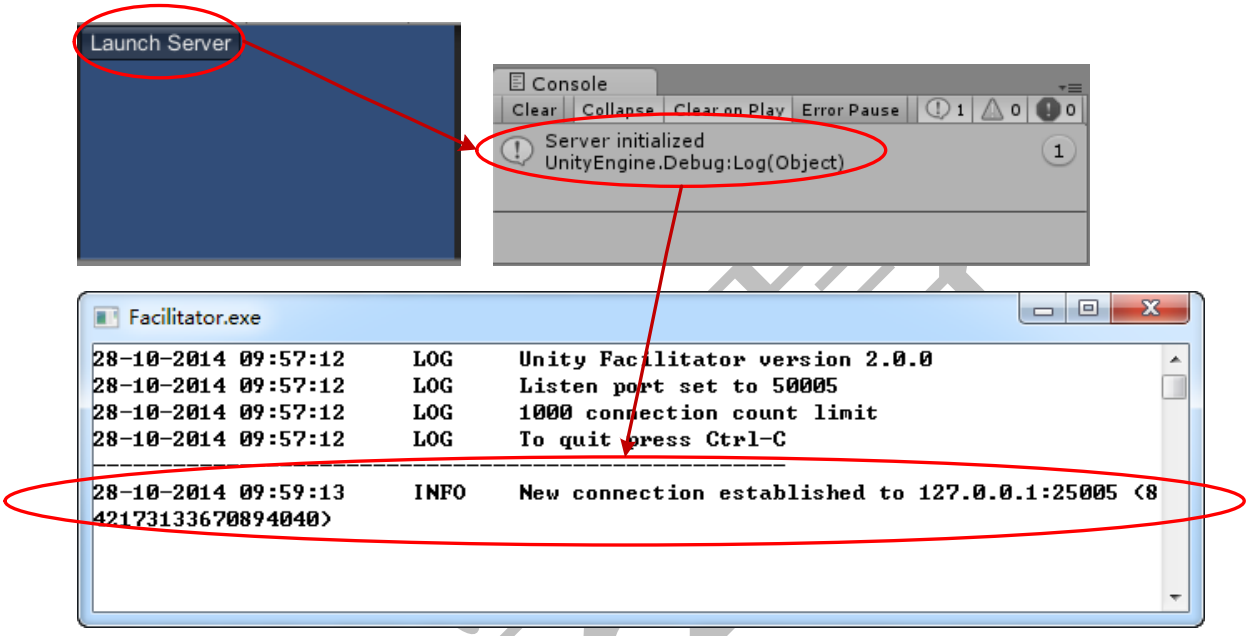

图 1-15 连接到自定义的主服务器

1.4 注册“服务”

联网玩同一游戏的玩家不只一位，所以在网络上也不会只存在一个服务。作为一个想要加入游戏的玩家而言，他需要从众多的服务中选择一个，然后再加入其中。不同的服务在网络上是通过“服务名”来标识的，而服务名是需要注册的。本节就来说明在网络上注册“服务名”，然后让其它玩家通过这个名称加入游戏的方法，如图 1-16 所示。

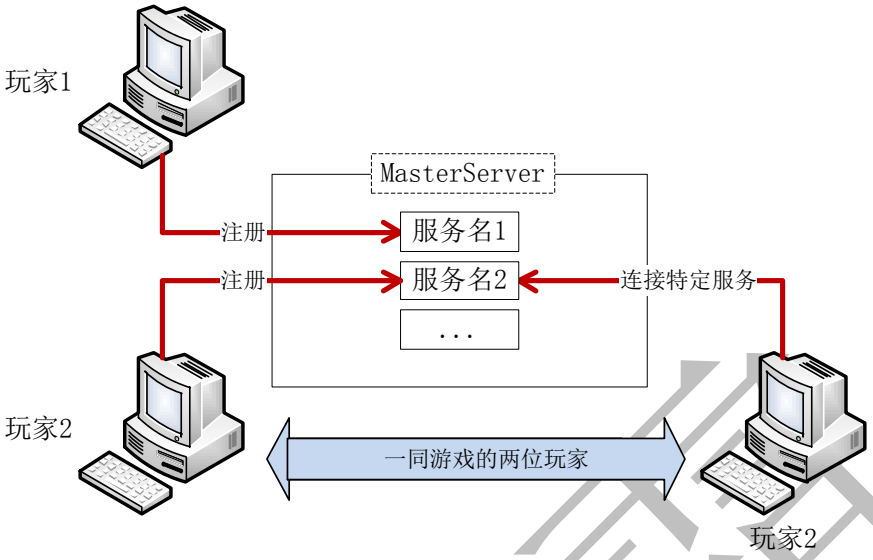

图 1-16　注册“服务”

1.4.1　在主服务器上注册一个服务

MasterServer 上会显示所有的服务列表，但前提是游戏本身需要去主动注册。例如，上一节只是连接到了自定义的服务器，而没有注册，因此 MasterServer 上不会有任何输出。本节将会说明在主服务器上注册一个服务的方法。

提示：只有一个玩家在 MasterServer 上注册了一个具体的服务以后，其它玩家才能加入这个服务，并一起游戏！

（1）打开本章“初始化服务器”一节所创建的脚本 ExampleUnityNetworkInitializeServer，然后添加用于注册的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ExampleUnityNetworkInitializeServer : MonoBehaviour
05  {
06      void OnGUI()
07      {
08          if( GUILayout.Button( "Launch Server" ) )
09          {
10              LaunchServer();
11          }
12      }
13      //启动服务器
14      void LaunchServer()
15      {
16          //接受来自 8 个主机的连接，监听的端口号是 25005，支持 NAT 穿透
17          Network.InitializeServer( 8, 25005, true );
18          MasterServer.RegisterHost("GameTypeName1",          "gameName2",
"comment2");
19      }
20      // Network.InitializeServer()调用完毕后，主动调用此方法
21      void OnServerInitialized()
```

```
22        {
23              Debug.Log( "Server initialized" );
24        }
25  }
```

脚本 18 行的代码，就实现了“注册”的功能。调用的方法是 MasterServer.RegisterHost()，需要 3 各参数：

- 第一个参数用于定义“服务类型名”；
- 第二个参数用于定义“游戏名”；
- 第三个参数用于定义“附加显示的信息”；

（2）将脚本 ExampleUnityNetworkInitializeServer 与 ExampleUnityNetworkingConnectToMasterServer，一同赋予游戏场景中的 Main Camera 对象，然后运行游戏，效果如图 1-17 所示。

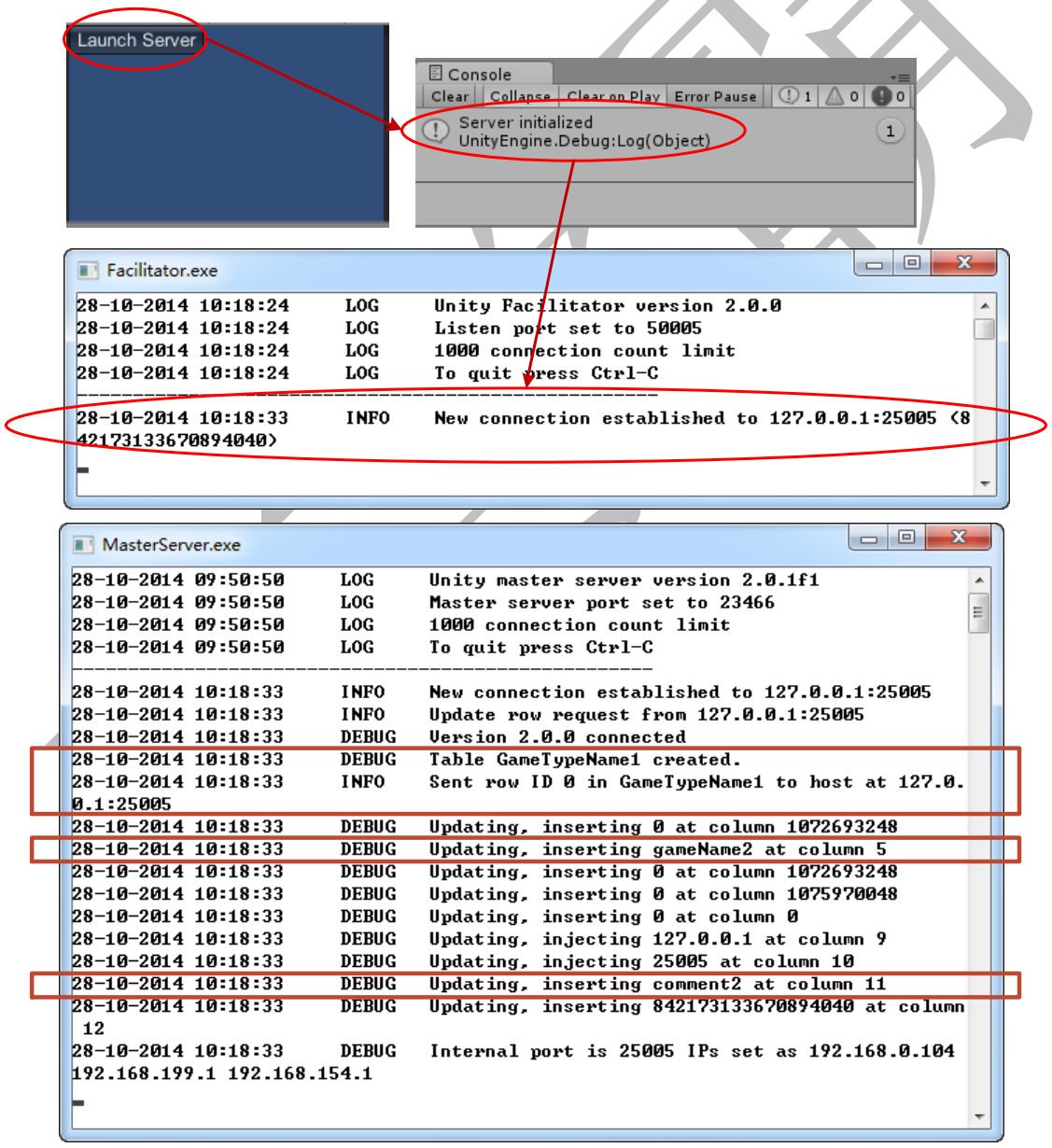

图 1-17　在 MasterServer 上注册一个服务器的效果

1.4.2　在游戏视图上浏览特定服务

游戏可以在游戏视图上浏览 MasterServer 上的特定服务，然后选择一个服务加入其中。这就是本节要实现的效果。

（1）在 Project 视图里，新建 C#脚本，命名为 ExampleUnityNetworkingBrowseServers，并添加下面的代码：

```
using UnityEngine;
using System.Collections;

public class ExampleUnityNetworkingBrowseServers : MonoBehaviour
{
    //表示当前是否正在加载 MasterServer 上的特定服务
    private bool loading = false;
    //滚动条的初始位置
    private Vector2 scrollPos = Vector2.zero;
    void Start()
    {
        //自定义的用于请求 MasterServer 上特定服务的方法
        refreshHostList();
    }
    void OnGUI()
    {
        if( GUILayout.Button( "Refresh" ) )
        {
            refreshHostList();
        }
        if( loading )
        {
            GUILayout.Label( "Loading..." );
        }
        else
        {
            scrollPos = GUILayout.BeginScrollView( scrollPos,
                    GUILayout.Width( 200f ), GUILayout.Height( 200f ) );
            //存储所有服务名的数组
            HostData[] hosts = MasterServer.PollHostList();
            for( int i = 0; i < hosts.Length; i++ )
            {
                if( GUILayout.Button( hosts[i].gameName,
GUILayout.ExpandWidth( true ) ) )
                {
                    Network.Connect( hosts[i] );
                }
            }
            if( hosts.Length == 0 )
            {
                GUILayout.Label( "No servers running" );
            }
            GUILayout.EndScrollView();
        }
    }
```

```
45      void refreshHostList()
46      {
47          loading = true;
48          //清除上次从 MasterServer.RequestHostList()中接收的服务列表
49          MasterServer.ClearHostList();
50          //向 MasterServer 请求当前的服务列表
51          MasterServer.RequestHostList( "GameTypeName1" );
52      }
53      //此方法用于记录来自 MasterServer 的事件
54      void OnMasterServerEvent( MasterServerEvent msevent )
55      {
56          //此事件表明接收到了来自 MasterServer 的服务列表
57          if( msevent == MasterServerEvent.HostListReceived )
58          {
59              loading = false;
60          }
61      }
62  }
```

- 脚本 30 行，调用 MasterServer.PollHostList()方法，访问接收到的服务列表信息；
- 脚本 31~37 行，将服务信息按列，以按钮的形式显示于游戏视图中，当单击视图上的对应按钮时，会加入到这个服务中；
- 脚本 45~52 行，自定义了方法 refreshHostList()，用于更新游戏视图上的服务列表。步骤是，首先清除上次从 MasterServer 接收到的服务列表信息，然后再向 MasterServer 请求新的服务列表信息；

提示：请求服务列表，使用的方法是 MasterServer.RequestHostList()，需要传入“游戏类型名”作为参数，此处是“GameTypeName1”。

- 脚本 54~61 行，是 Unity 会主动调用的方法 OnMasterServerEvent()，它用于记录来自 MasterServer 的各种事件，所以也包括“已接收到服务列表”的事件；

（2）下面就来验证上一步骤所创建脚本实现的功能，即能否让玩家在游戏视图上浏览到当前 MasterServer 上的服务列表。为此需要首先运用上一节所学到的知识，在主服务器上注册两个服务。

（3）注册第一个服务的脚本 ExampleUnityNetworkInitializeServer 的代码如下，端口是 25005，服务类型名是 GameTypeName1，但游戏名为 gameName1。然后将此游戏项目编译成一个可执行文件，并运行此可执行文件。

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ExampleUnityNetworkInitializeServer : MonoBehaviour
05  {
06      …                                   //省略
07      //启动服务器
08      void LaunchServer()
09      {
10          //接受来自 8 个主机的连接，监听的端口号是 25005，支持 NAT 穿透
11          Network.InitializeServer( 8, 25005, true );
12          MasterServer.RegisterHost("GameTypeName1",          "gameName1",
"comment1");
13      }
14      …                                   //省略
15  }
```

（4）注册第二个服务的脚本 ExampleUnityNetworkInitializeServer 的代码如下，端口是 25006，服务类型名是 GameTypeName1，但游戏名为 gameName2。然后将此游戏项目编译成一个可执行文件，并运行此可执行文件。

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ExampleUnityNetworkInitializeServer : MonoBehaviour
05  {
06      …                                   //省略
07      //启动服务器
08      void LaunchServer()
09      {
10          //接受来自 8 个主机的连接，监听的端口号是 25005，支持 NAT 穿透
11          Network.InitializeServer( 8, 25006, true );
12          MasterServer.RegisterHost("GameTypeName1",          "gameName2",
"comment2");
13      }
14      …                                   //省略
15  }
```

（5）此时进入到游戏项目，为游戏场景中的 Main Camera 对象赋予 ExampleUnityNetworkingConnectToMasterServer 和 ExampleUnityNetworkingBrowseServers 脚本，然后运行游戏，效果如图 1-18 所示。

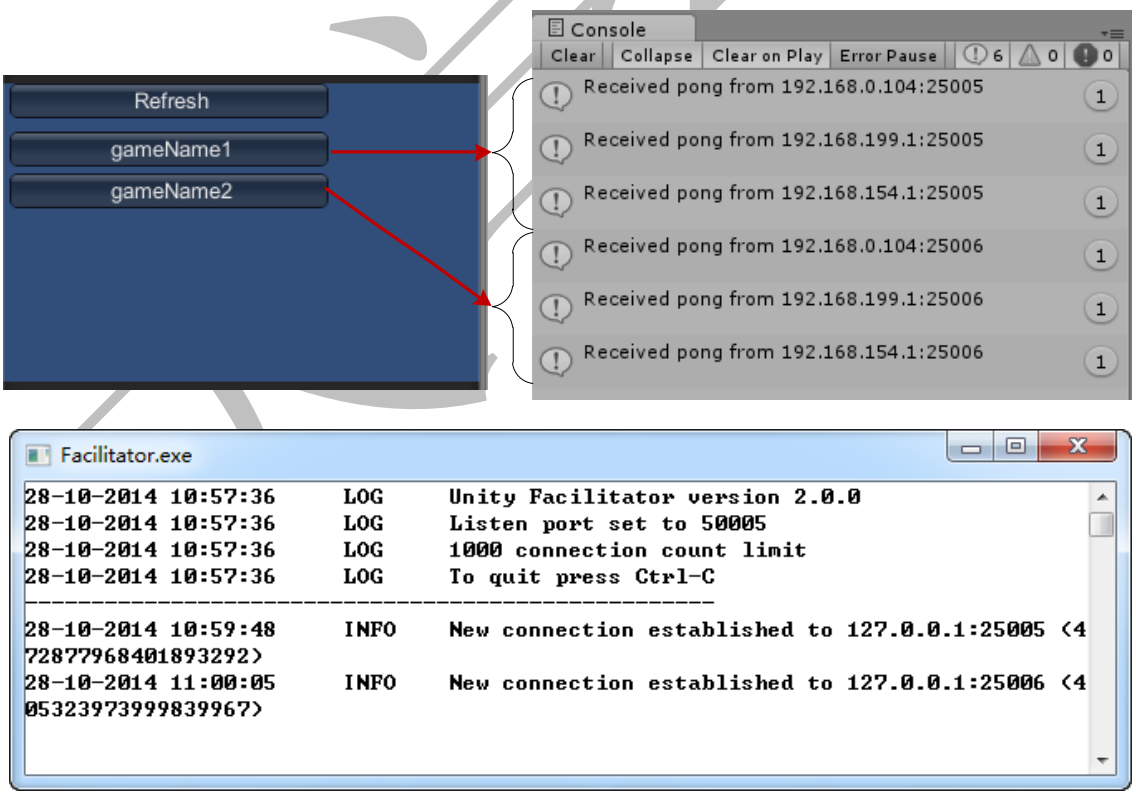

图 1-18　在游戏视图上浏览特定服务的效果

- 从 Facilitator.exe 上可以看到，一共有两个游戏程序与服务器建立了联系，分别来自端口号 25005 和 25006；

- 游戏视图上也显示了 3 个按钮，第一个名为 Refresh 的按钮，用于获取 MasterServer 上的服务列表；因为 MasterServer 上现在有两个服务，因此就在视图上添加了后面的两个按钮：gameName1 和 gameName2；
- 点击对应的服务名按钮，即可加入到指定的服务中去，特定信息也会输出到 Console 视图上；
- MaterServer.exe 上也显示出了许多信息，但是内容太多，这里就不截图展示了；

1.5 实例：乒乓球游戏

本节会编写一个乒乓球游戏，但是只能在单机上玩。下一节才会介绍运用本章所学的知识，令其成为网络上的多人游戏的方法。本节会专注于游戏场景的搭建，以及游戏逻辑的实现。

1.5.1 搭建游戏场景

本小节会详细说明搭建游戏场景的各步骤，具体的内容如下：

（1）为游戏场景添加一个 Cube 对象，命名为 Paddle_P1，作为 Player1 的球拍，设置下列属性：

- Tag：Player；
- Position：(-17,0,0)；
- Rotation：(0,0,0)；
- Scale：(1,1,4)；
- 复选 Box Collider 组件下的 Is Trigger 属性；

（2）选中 Paddle_P1 对象，然后按下快捷键 Ctrl+D，得到前者的拷贝，命名为 Paddle_P2，作为 Player2 的球拍，并设置下列属性：

- Position：(17,0,0)；

（3）再添加一个 Cube 对象，命名为 Goal_P1，作为 Player1 球拍后的界限，设置下列属性：

- Tag：Goal；
- Position：(-20,0,0)；
- Rotation：(0,0,0)；
- Scale：(1,1,24)；
- 复选 Box Collider 组件下的 Is Trigger 属性；

（4）选中 Goal_P1 对象，然后按下快捷键 Ctrl+D，得到前者的拷贝，命名为 Goal_P2，作为 Player2 球拍后的界限，设置下列属性：

- Position：(20,0,0)；

（5）再添加一个 Cube 对象，命名为 Boundary_Up，作为球场的上边界，设置下列属性：

- Tag：Boundary；
- Position：(0,0,11.5)；
- Rotation：(0,90,0)；
- Scale：(1,1,40)；
- 复选 Box Collider 组件下的 Is Trigger 属性；

（6）选中 Boundary_Up 对象，然后按下快捷键 Ctrl+D，得到前者的拷贝，命名为 Boundary_Down，作为球场的下边界，设置下列属性：

- Position：(0,0,-11.5)；

（7）选中场景中的 Main Camera 对象，设置下列属性：

- Position：(0,10,0)；
- Rotation：(90,0,0)；
- Scale：(1,1,1)；
- Camera 组件下 Projection 属性：Orthographic；
- Camera 组件下 Size 属性：15；

（8）为游戏场景添加一个 Directional light 对象，设置其下列属性：

- Position：(0,100,0)；
- Rotation：(90,0,0)；
- Scale：(1,1,1)；

（9）为游戏场景添加一个 Cube 对象，命名为 Ball，作为乒乓球，设置器下列属性：

- Position：(0,0,0)；
- Rotation：(0,0,0)；
- Scale：(1,1,1)；
- 添加 Rigidbody 组件，取消 Use Gravity 属性的复选，复选 Is Kinematic 属性，如图 1-19 所示。

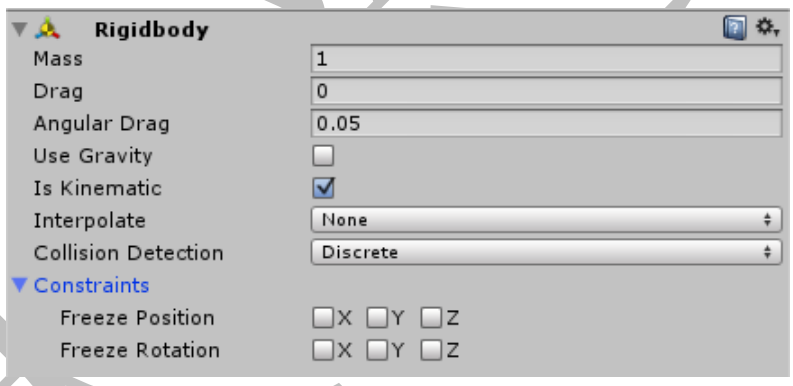

图 1-19　Rigidbody 组件属性设置

（10）完成以上的所有设置以后，游戏场景就搭建完毕了，效果如图 1-20 所示。

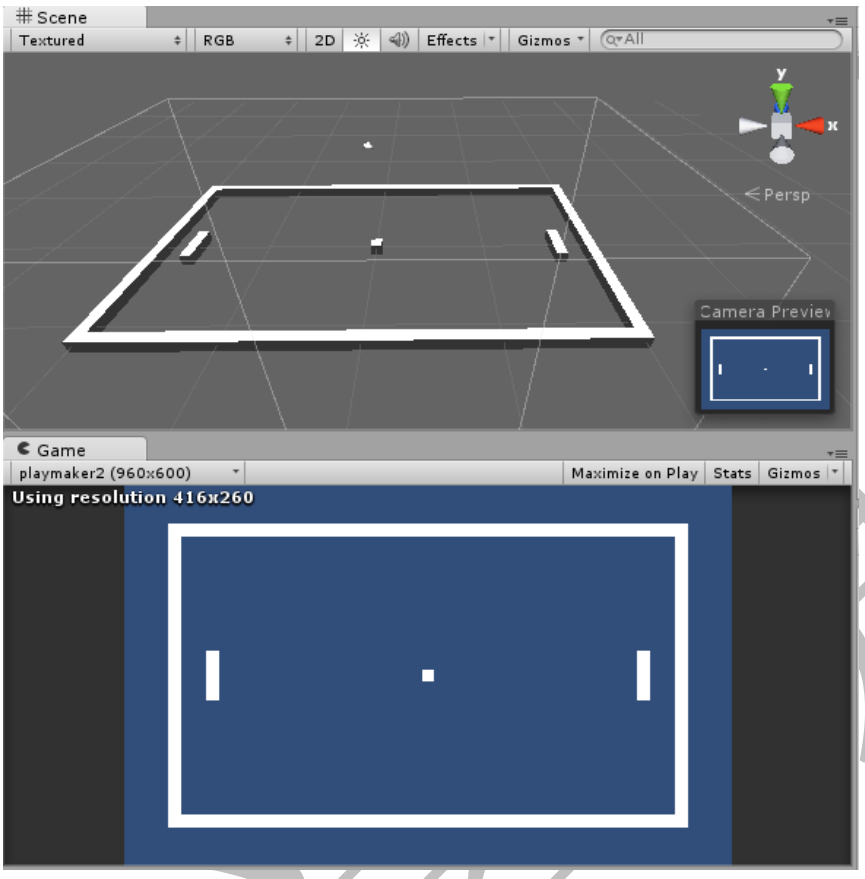

图 1-20　Scene 和 Game 视图的效果

1.5.2　游戏的功能逻辑

本节要制作的乒乓球游戏有以下的功能逻辑：

- “乒乓球”可以与游戏场景中的所有 Cube 对象发生碰撞；
- 与“球拍”碰撞的时候，乒乓球在 X 轴上的方向会反转；
- 与“球场上下边界”碰撞的时候，乒乓球在 Z 轴上的方向会反转；
- 与“球场界限”碰撞的时候，对方得分 1；
- 得分满 10 分，一方获胜；

1.5.3　编写实现游戏逻辑的脚本

本小节会专注于实现乒乓球游戏的功能逻辑，也就是脚本代码的编写。

1.乒乓球的游戏逻辑

乒乓球起先位于游戏视图的中心，游戏开始后，会以一定的速度朝着一个方向移动，并与游戏场景中的所有其它 Cube 对象发生碰撞。在 Project 视图里，新建一个 C#脚本，命名为 Ball，添加实现这一系列功能的代码，如下：

```
01  using UnityEngine;
02  using System.Collections;
03
```

```
04  public class Ball : MonoBehaviour
05  {
06      //乒乓球的起始速度
07      public float StartSpeed = 5f;
08      //乒乓球的最大速度
09      public float MaxSpeed = 20f;
10      //每次碰撞发生后，速度的增量
11      public float SpeedIncrease = 0.25f;
12      //乒乓球当前的速度
13      private float currentSpeed;
14      //乒乓球当前的运动方向
15      private Vector2 currentDir;
16      //是否重新发球
17      private bool resetting = false;
18      void Start()
19      {
20          //设置起始的球速
21          currentSpeed = StartSpeed;
22          //设置起始球的移动方向
23          currentDir = Random.insideUnitCircle.normalized;
24      }
25      void Update()
26      {
27          //处于重新发球状态时，球体固定不动
28          if( resetting )
29              return;
30          //朝指定方向移动球体
31          Vector2 moveDir = currentDir * currentSpeed * Time.deltaTime;
32          transform.Translate( new Vector3( moveDir.x, 0f, moveDir.y ) );
33      }
34      void OnTriggerEnter( Collider other )
35      {
36          //与球场上下边界发生碰撞
37          if( other.tag == "Boundary" )
38          {
39              //反转三维坐标中 Z 轴的值，二维坐标中 Y 轴的值
40              currentDir.y *= -1;
41          }
42          //与球拍发生碰撞
43          else if( other.tag == "Player" )
44          {
45              //反转三维坐标中 X 轴的值，二维坐标中 X 轴的值
46              currentDir.x *= -1;
47          }
48          //与球场界限发生碰撞
49          else if( other.tag == "Goal" )
50          {
51              // 重新发球
52              StartCoroutine( resetBall() );
53              //加分
54              other.SendMessage(                                    "GetPoint",
SendMessageOptions.DontRequireReceiver );
55          }
```

```
56              //增加速度
57              currentSpeed += SpeedIncrease;
58              //检测乒乓球的速度，是否在指定的范围之内
59              currentSpeed = Mathf.Clamp( currentSpeed, StartSpeed, MaxSpeed );
60          }
61          IEnumerator resetBall()
62          {
63              //重新设置乒乓球的位置、速度，以及移动方向
64              resetting = true;
65              transform.position = Vector3.zero;
66              currentDir = Vector3.zero;
67              currentSpeed = 0f;
68              //等待 3 秒后，开始新一轮的比赛
69              yield return new WaitForSeconds( 3f );
70              Start();
71              resetting = false;
72          }
73      }
```

需要说明的是：

- 乒乓球的起始位置、速度是一定的，但是方向是随机的，即脚本 18~24 行代码实现的功能；
- 在重新发球的的时候，需要等待 3 秒；
- 在每次碰撞发生以后，乒乓球的速度都会增加一点点；

将此脚本赋予游戏场景中的乒乓球对象 Ball，如图 1-21 所示，在 Inspector 视图上，可以设置乒乓球的起始速度、最大速度，以及速度增量。然后就可以运行游戏了，效果如图 1-22 所示。

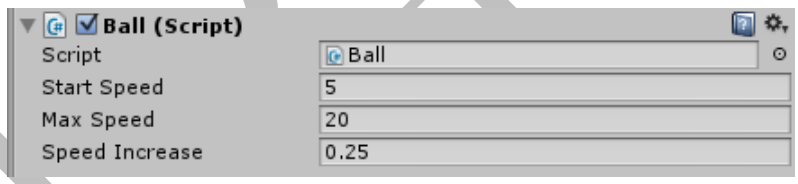

图 1-21　Ball(Script)组件属性设置

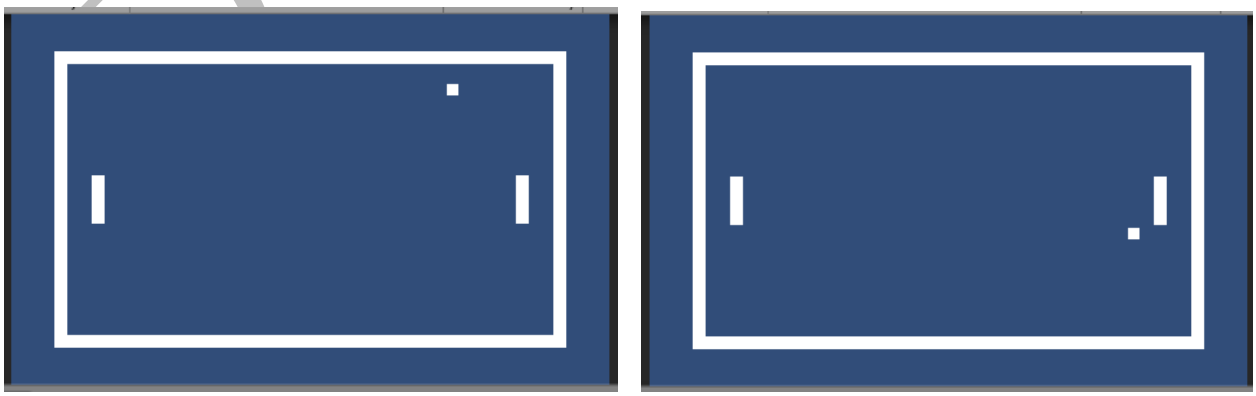

图 1-22　乒乓球的游戏逻辑

2.球拍的游戏逻辑

球拍只被允许上下移动，速度是恒定的，范围也是固定的。当玩家按下键盘上的上下方向键的时候，就是移动球拍的时机。在 Project 视图里，新建一个 C#脚本，命名为 Paddle，添加实现这一系列功能的代码，如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class Paddle : MonoBehaviour
05  {
06      //球拍的移动速度
07      public float MoveSpeed = 10f;
08      //球拍上下移动的范围
09      public float MoveRange = 10f;
10      //球拍是否接受玩家的控制
11      public bool AcceptsInput = true;
12      void Update()
13      {
14          //球拍不接受玩家的控制，则中止
15          if( !AcceptsInput )
16              return;
17          //获取玩家的输入
18          float input = Input.GetAxis( "Vertical" );
19          //移动球拍
20          Vector3 pos = transform.position;
21          pos.z += input * MoveSpeed * Time.deltaTime;
22          //限制球拍的移动范围
23          pos.z = Mathf.Clamp( pos.z, -MoveRange, MoveRange );
24          //修改球拍的位置
25          transform.position = pos;
26      }
27  }
```

需要说明的是：

- 球拍可以拒绝玩家的控制；
- 获取玩家在键盘上的输入，使用的是 Unity 自定义的“输入轴”，即脚本 18 行的代码。

提示：查看 Unity 定义的“输入轴”的方法是，单击 Edit|Project Settings|Input 命令，即可打开名为 InputManager 的对话框，在此对话框中不光可以查看“输入轴”，还可以修改，如图 1-23 所示。

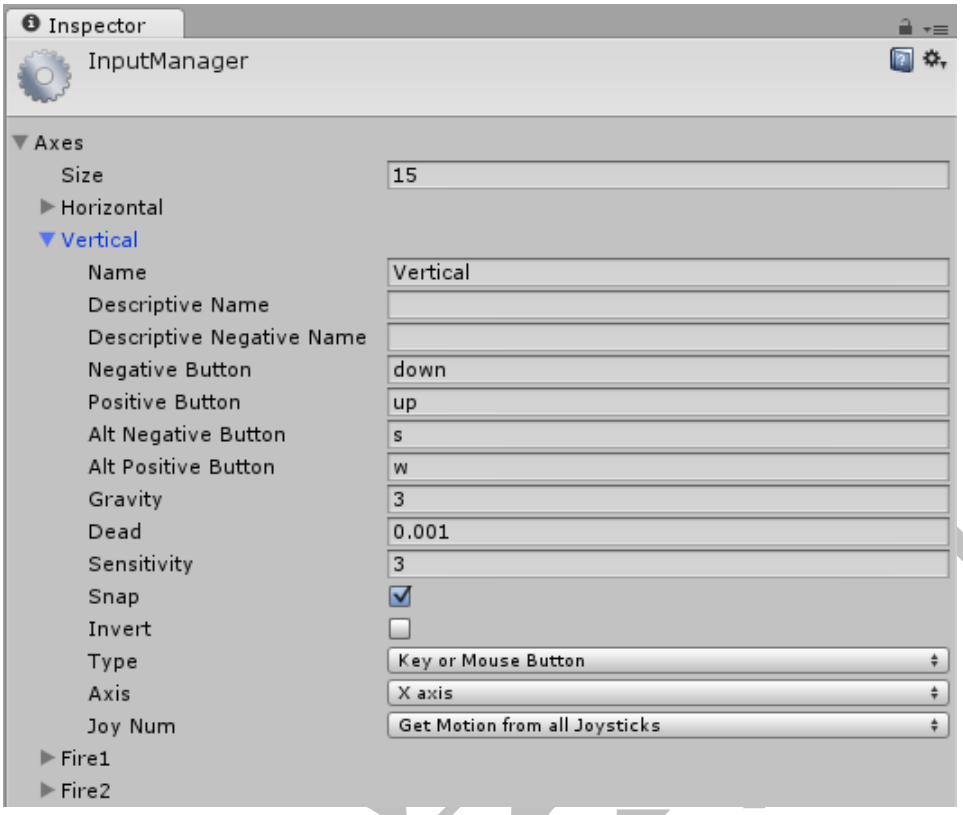

图 1-23　InputManager 对话框

将此脚本赋予游戏场景中的 Paddle_P1 和 Paddle_P2 对象，如图 1-24 所示。在 Inspector 视图上，可以设置球拍的移动速度、移动范围和是否接受玩家的控制。然后就可以运行游戏了，效果如图 1-25 所示。

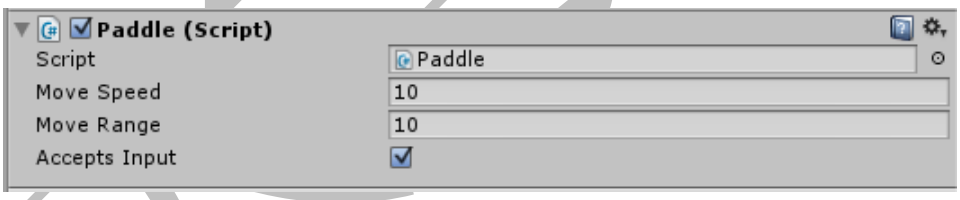

图 1-24　Paddle(Script)组件属性设置

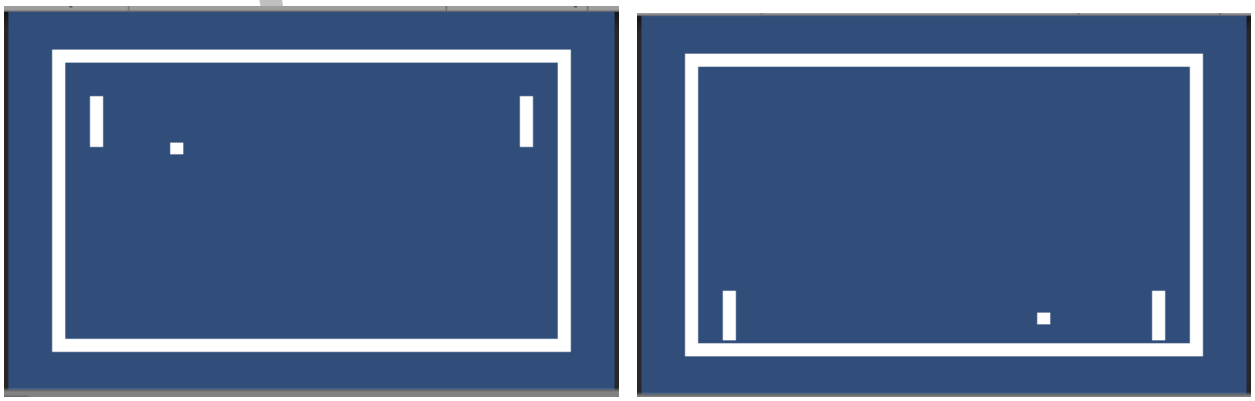

图 1-25　球拍的游戏逻辑

3.分数的游戏逻辑

分数的游戏逻辑涉及内容较多，因此下面决定按步骤依次说明。

（1）添加一个空游戏对象到游戏场景，命名为 scorekeeper，然后在 Project 视图里，新建一个 C#脚本，命名为 Scorekeeper，专门用于记录两位玩家各自的比分。为此脚本添加下面的代码：

```
using UnityEngine;
using System.Collections;

public class Scorekeeper : MonoBehaviour
{
    //表示获胜的分数
    public int ScoreLimit = 10;
    //Player1 的分数
    private int p1score = 0;
    // Player2 的分数
    private int p2score = 0;
    //加分操作
    public void AddScore( int player )
    {
        //为 Player 1 加 1 分
        if( player == 1 )
        {
            p1score++;
        }
        //为 Player 2 加 1 分
        else if( player == 2 )
        {
            p2score++;
        }
        //检查是否有玩家达到了胜利的分数
        if( p1score >= ScoreLimit || p2score >= ScoreLimit )
        {
            // player 1 分数大于 player 2
            if( p1score > p2score )
                Debug.Log( "Player 1 wins" );
            // player 2  分数大于  player 1
            if( p2score > p1score )
                Debug.Log( "Player 2 wins" );
            //分数相同
            else
                Debug.Log( "Players are tied" );
            //开始新一轮的比赛
            p1score = 0;
            p2score = 0;
        }
    }
}
```

需要说明的是：

- 一共可以有两位玩家同时游戏；
- 分数达到胜利标准的玩家获胜；
- 起始的分数是 0，赢一局加 1 分；

将此脚本赋予游戏场景中的 scorekeeper 对象。如图 1-26 所示，在 Inspector 视图上，可以设置玩家胜利的分数限制。

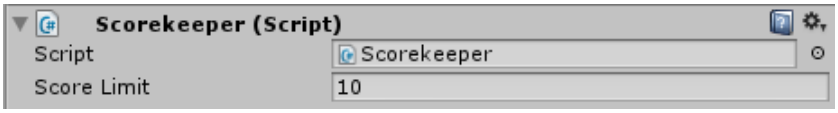

图 1-26　Scorekeeper(Script)组件属性设置

（2）乒乓球与球拍后的界限碰撞以后，是触发玩家分数改变（加 1 分）的契机。在 Project 视图里新建一个 C#脚本，命名为 Goal，用于指定该给哪个玩家加分，代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class Goal : MonoBehaviour
05  {
06      //在 Inspector 视图上，指定为哪位玩家加分，即 1 还是 2
07      public int Player;
08      //表示 scorekeeper 对象上的 Scorekeeper(Script)组件
09      public Scorekeeper scorekeeper;
10      public void GetPoint()
11      {
12          //当乒乓球与球拍后的界限碰撞以后，触发分数的增加操作
13          scorekeeper.AddScore( Player );
14      }
15  }
```

将此脚本赋予游戏场景中的 Goal_P1 和 Goal_P2 对象，然后在 Inpector 视图里分别设置各自的 Goal(Script)组件的属性，如图 1-27 所示。即碰到了 Player1 后的界限，就为 Player2 加分，反之，为 Player1 加分。

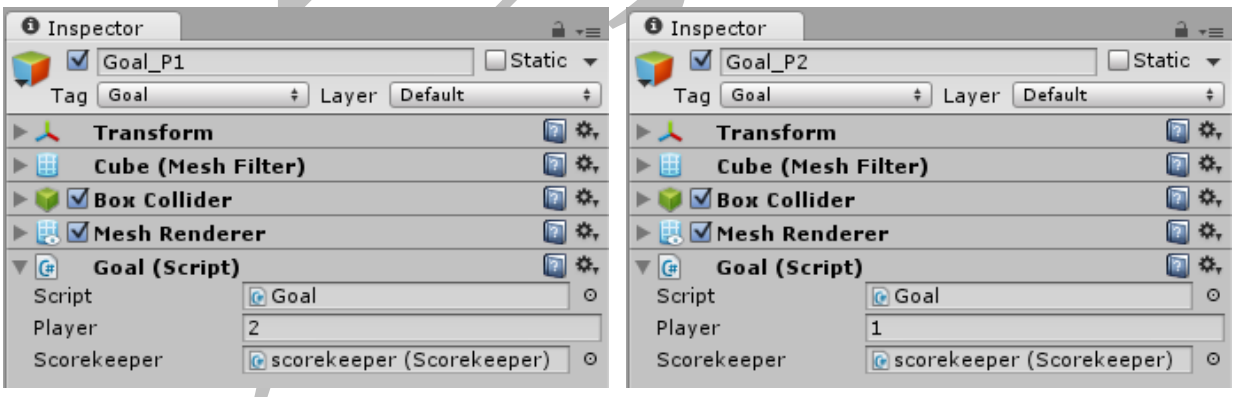

图 1-27　Goal(Script)组件属性设置

（3）分数应该实时的显示在游戏视图上，玩家才能时刻知道此时自己的得分。下面就实现让分数信息显示于游戏视图上的功能。为游戏场景添加 3D Text 对象，命名为 p1Score，用于显示 Player1 的分数，设置其下列属性，如图 1-28 所示。

- Position：(-16,0,9)；
- Rotation：(90,0,0)；
- Scale：(1,1,1)；
- Text：0；

- Character Size：2；

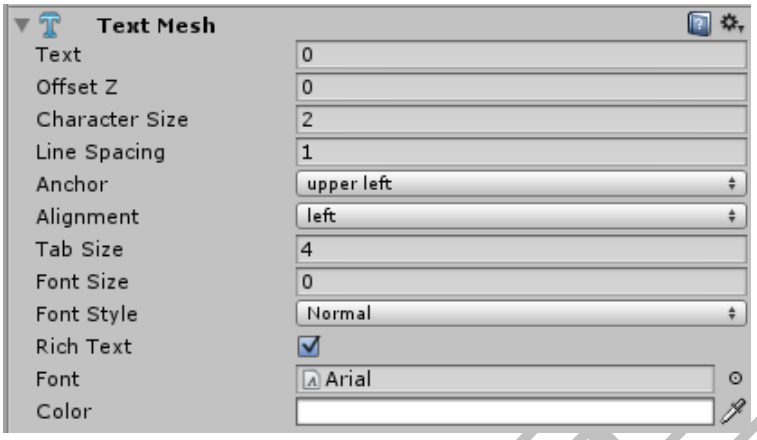

图 1-28　Text Mesh 组件属性设置

（4）拷贝 p1Score，命名为 p2Score，用于显示 Player2 的分数，设置其下列属性：

- Position：(14,0,9)；

完成以上设置以后，游戏视图的效果如图 1-29 所示。可知显示玩家分数信息的对象，分别位于游戏视图的左上角和右上角。

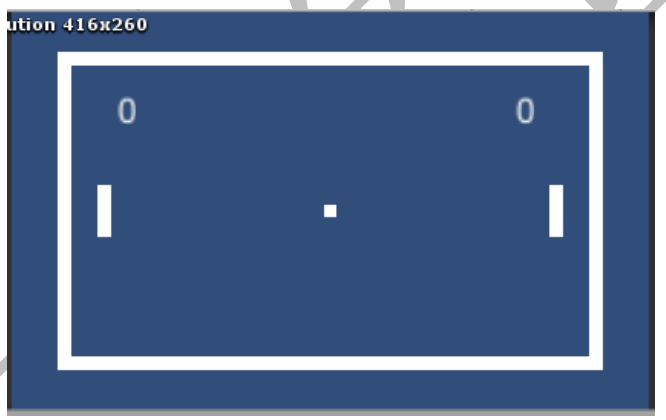

图 1-29　游戏视图中分数的显示位置

（5）打开脚本 Scorekeeper，添加用于修改显示的分数的代码，添加代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class Scorekeeper : MonoBehaviour
05  {
06      //表示获胜的分数
07      public int ScoreLimit = 10;
08      //Player1 的分数
09      private int p1score = 0;
10      // Player2 的分数
11      private int p2score = 0;
12      //表示 p1Score 的 Text Mesh 组件
13      public TextMesh Player1ScoreDisplay;
14      //表示 p2Score 的 Text Mesh 组件
15      public TextMesh Player2ScoreDisplay;
16      //加分操作
```

```
17      public void AddScore( int player )
18      {
19          //为 Player 1 加 1 分
20          if( player == 1 )
21          {
22              p1score++;
23          }
24          //为 Player 2 加 1 分
25          else if( player == 2 )
26          {
27              p2score++;
28          }
29          //检查是否有玩家达到了胜利的分数
30          if( p1score >= ScoreLimit || p2score >= ScoreLimit )
31          {
32              // player 1 分数大于 player 2
33              if( p1score > p2score )
34                  Debug.Log( "Player 1 wins" );
35              // player 2  分数大于  player 1
36              if( p2score > p1score )
37                  Debug.Log( "Player 2 wins" );
38              //分数相同
39              else
40                  Debug.Log( "Players are tied" );
41              //开始新一轮的比赛
42              p1score = 0;
43              p2score = 0;
44          }
45          //修改 p1Score 显示的文本内容
46          Player1ScoreDisplay.text = p1score.ToString();
47          //修改 p2Score 显示的文本内容
48          Player2ScoreDisplay.text = p2score.ToString();
49      }
50  }
```

（6）运行游戏，效果如图 1-30 所示。可知此时 Player1 的分数是 3，而 Player2 的分数是 1。

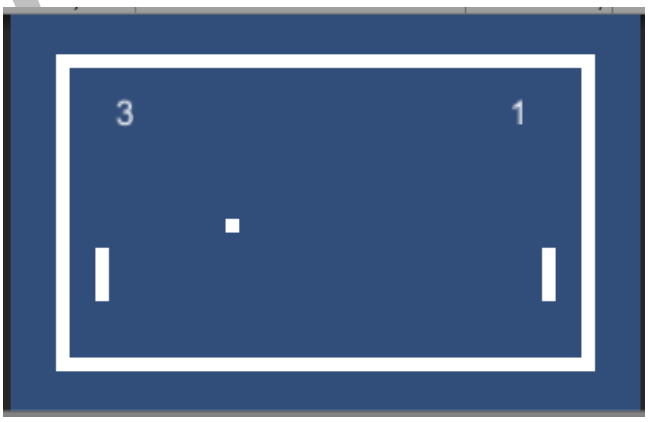

图 1-30　分数的游戏逻辑

1.6　为游戏实例添加网络对战功能

上一节已经完成了一个游戏实例，即乒乓球游戏。本节会在此实例的基础上，添加本章所讲解的涉及网络的功能，使得此实例拥有网络对战的能力，即对战的是两个实实在在的玩家。

1.6.1　初始化服务器

当游戏初次运行的时候，应该首先检查当前是否建立了网络连接，没有的话，就需要完成“初始化服务器”的操作，即建立与主服务器的连接。具体的实现步骤如下：

（1）在 Project 视图里，新建一个 C#脚本，命名为 RequireNetwork，为此脚本添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class RequireNetwork : MonoBehaviour
05  {
06      void Awake()
07      {
08          if( Network.peerType == NetworkPeerType.Disconnected )
09              Network.InitializeServer( 1, 25005, true );
10      }
11  }
```

- 脚本 08 行，通过查看 Network.peerType 变量，得知当前是否建立的网络连接；
- 脚本 09 行，使用 Network.InitializeServer()方法，初始化了服务器，允许 1 人接入此网络，使用的端口号是 25005；

（2）将脚本 RequireNetwork 与 ExampleUnityNetworkingConnectToMasterServer，一同赋予游戏场景中的 Main Camera 对象，如图 1-31 所示，即游戏实例会与我们自定义的服务器建立网络连接。

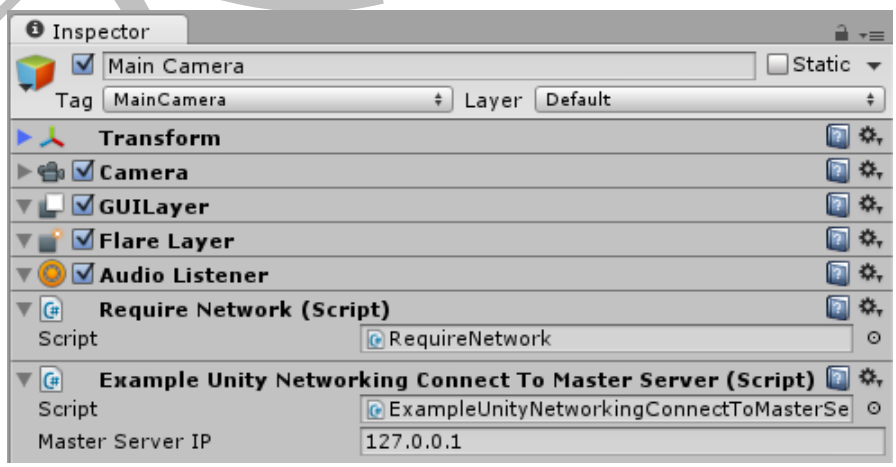

图 1-31　赋予 Main Camera 对象的两个脚本组件

（3）运行游戏，在 Facilitator.exe 里可以看到新的网络连接会被建立。退出游戏以后，此连接就中断了，如图 1-32 所示。

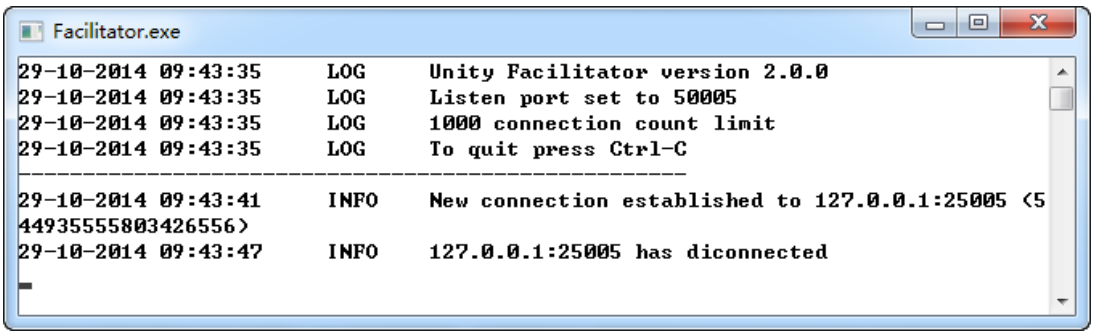

图 1-32　网络连接的建立与断开

1.6.2　串行化球拍的移动状态

联机对战的两个玩家，可以使用键盘上的方向键移动自己的球拍，同时这一效果也应该实时的同步到另一玩家的游戏视图上。同步的过程就是向网络串行化球拍移动状态的过程。具体的实现步骤如下：

（1）打开脚本 Paddle，添加实现串行化球拍移动状态功能的代码，如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class Paddle : MonoBehaviour
05  {
06      //球拍的移动速度
07      public float MoveSpeed = 10f;
08      //球拍上下移动的范围
09      public float MoveRange = 10f;
10      //球拍是否接受玩家的控制
11      public bool AcceptsInput = true;
12      //从网络上读取到的球拍位置状态信息
13      private Vector3 readNetworkPos;
14      void Start()
15      {
16          //若此球拍不是被当前玩家所控制的，就拒绝对其输入做出反应
17          AcceptsInput = NetworkView.isMine;
18      }
19      void Update()
20      {
21          //如果此球拍不是被当前玩家所控制的，就接收来自网络的位置信息
22          if( !AcceptsInput )
23          {
24              transform.position =
25                  Vector3.Lerp( transform.position, readNetworkPos, 10f *
Time.deltaTime );
26              return;
27          }
28          //获取玩家的输入
29          float input = Input.GetAxis( "Vertical" );
30          //移动球拍
31          Vector3 pos = transform.position;
32          pos.z += input * MoveSpeed * Time.deltaTime;
```

```
33              //限制球拍的移动范围
34              pos.z = Mathf.Clamp( pos.z, -MoveRange, MoveRange );
35              //修改球拍的位置
36              transform.position = pos;
37          }
38          void OnSerializeNetworkView( BitStream stream )
39          {
40              //将球拍的位置信息，写入网络，即串行化到网络
41              if( stream.isWriting )
42              {
43                  Vector3 pos = transform.position;
44                  stream.Serialize( ref pos );
45              }
46              //从网络上读取球拍的位置信息
47              else
48              {
49                  Vector3 pos = Vector3.zero;
50                  stream.Serialize( ref pos );
51                  readNetworkPos = pos;
52              }
53          }
54  }
```

对于此脚本，需要说明的是：

- 对于游戏本身而言，两个球拍中，一个是被当前玩家所控制的，另一个则是被网络上对战的另一方所控制的；
- 对于被当前玩家控制的球拍而言，玩家可以令其上下移动，而此球拍的位置信息会被实时的传输到网络上；
- 对于被网络上对战的另一方所控制的球拍而言，它会依据从网络上接收到的信息，实时改变自身的位置；

（2）在 Hierarchy 视图里，选中 Paddle_P1 对象，为其添加 Network View 组件，如图 1-33 所示。

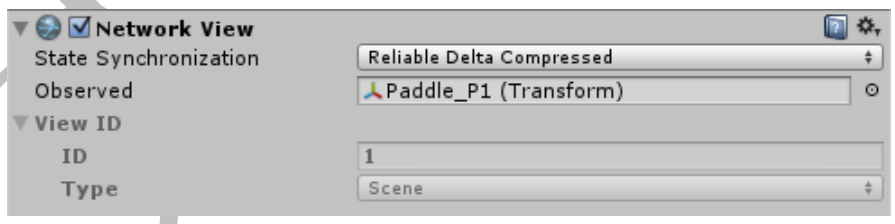

图 1-33　Network View 组件的属性及其设置

（3）将 Paddle_P1 拖动到 Project 视图里，来创建此对象的预置资源。新建两个空对象，命名为 P1_Pos 和 P2_Pos，依次设置其位置属性如下，也就是与 Paddle_P1 和 Paddle_P2 对象所在的位置一致。

- Position：(-17,0,0)；
- Position：(17,0,0)；

（4）删除游戏场景中的 Paddle_P1 和 Paddle_P2 对象。球拍不应该总是显示在游戏视图中，而是应该在有玩家加入游戏的时候才显示。

1.6.3 指定球拍出现的时机

本小节决定让游戏场景中的 scorekeeper 对象，控制游戏视图中球拍的出现与否，即：

- 对于开始了一个新游戏的玩家而言，为他实例化 Paddle_P1 球拍；
- 对于加入游戏的玩家而言，为他实例化 Paddle_P2 球拍；

（1）打开脚本 Scorekeeper，添加实现对应功能的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class Scorekeeper : MonoBehaviour
05  {
06      //表示获胜的分数
07      public int ScoreLimit = 10;
08      //表示 Paddle_P1 出现的位置
09      public Transform SpawnP1;
10      //表示 Paddle_P2 出现的位置
11      public Transform SpawnP2;
12      //表示要实例化的球拍
13      public GameObject paddlePrefab;
14      //Player1 的分数
15      private int p1score = 0;
16      // Player2 的分数
17      private int p2score = 0;
18      //表示 p1Score 的 Text Mesh 组件
19      public TextMesh Player1ScoreDisplay;
20      //表示 p2Score 的 Text Mesh 组件
21      public TextMesh Player2ScoreDisplay;
22      void Start()
23      {
24          if( Network.isServer )
25          {
26              //对于开始了一个新游戏的玩家而言，为他实例化 Paddle_P1 球拍
27              Network.Instantiate( paddlePrefab, SpawnP1.position,
Quaternion.identity, 0 );
28          }
29      }
30      //有玩家与此游戏建立了连接以后，Unity 会主动调用此函数
31      void OnPlayerConnected( NetworkPlayer player )
32      {
33          networkView.RPC( "net_DoSpawn", player, SpawnP2.position );
34      }
35      [RPC]
36      void net_DoSpawn( Vector3 position )
37      {
38          //对于加入游戏的玩家而言，为他实例化 Paddle_P2 球拍
39          Network.Instantiate( paddlePrefab, position, Quaternion.identity, 0 );
40      }
41      //加分操作
42      public void AddScore( int player )
43      {
44          //为 Player 1 加 1 分
```

```
45              if( player == 1 )
46              {
47                  p1score++;
48              }
49              //为 Player 2 加 1 分
50              else if( player == 2 )
51              {
52                  p2score++;
53              }
54              //检查是否有玩家达到了胜利的分数
55              if( p1score >= ScoreLimit || p2score >= ScoreLimit )
56              {
57                  // player 1 分数大于 player 2
58                  if( p1score > p2score )
59                      Debug.Log( "Player 1 wins" );
60                  // player 2  分数大于  player 1
61                  if( p2score > p1score )
62                      Debug.Log( "Player 2 wins" );
63                  //分数相同
64                  else
65                      Debug.Log( "Players are tied" );
66                  //开始新一轮的比赛
67                  p1score = 0;
68                  p2score = 0;
69              }
70              //修改 p1Score 显示的文本内容
71              Player1ScoreDisplay.text = p1score.ToString();
72              //修改 p2Score 显示的文本内容
73              Player2ScoreDisplay.text = p2score.ToString();
74          }
75      }
```

新添加的代码，都使用详细的注释说明了，这里就不重复说明了。

（2）在 Hierarchy 视图里，选中 scorekeeper 对象，然后在 Inspector 视图里设置其上 Scorekeeper(Script)组件的属性，如图 1-34 所示。即 Paddle_P1 球拍会在 P1_Pos 对象的位置上出现，而 Paddle_P2 球拍会在 P2_Pos 对象的位置上出现。

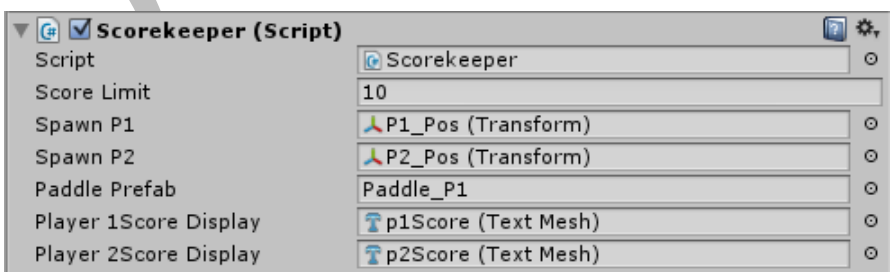

图 1-34　Scorekeeper(Script)组件属性设置

（3）运行游戏，如图 1-35 所示，是游戏运行前后的 Game 视图效果比较，即 Paddle_P1 球拍是在游戏开始后才出现的。

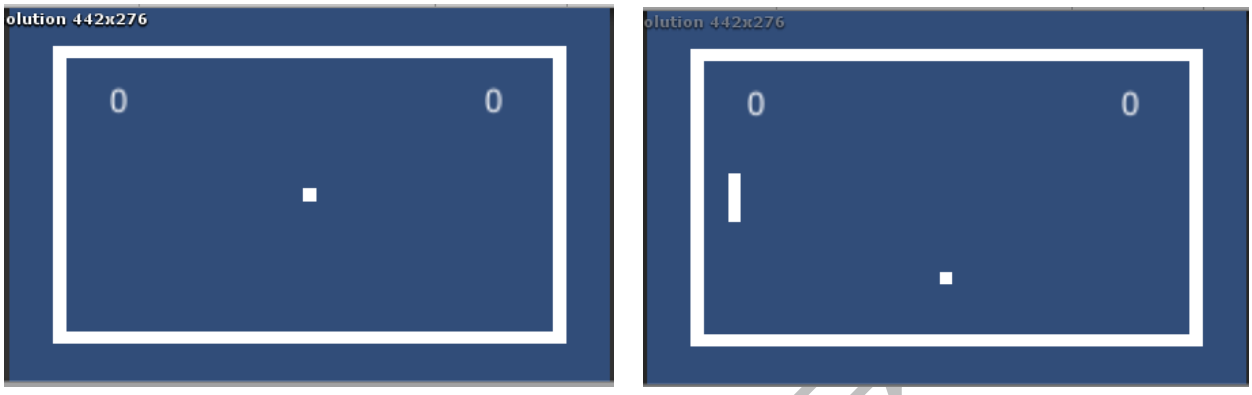

图 1-35　指定球拍出现的时机

1.6.4　串行化乒乓球的移动状态

乒乓球的移动情况，应该以创建游戏的玩家的游戏为主，而这个位置信息应该同步给加入游戏的玩家，即需要将乒乓球的位置状态数据串行化到网络上。除此以外，当游戏中只有一个玩家的时候，乒乓球应该固定在游戏视图的中心位置上。实现以上这些功能的步骤如下：

（1）打开脚本 Ball，并添加实现对应功能的代码，如下：

```
using UnityEngine;
using System.Collections;

public class Ball : MonoBehaviour
{
    //乒乓球的起始速度
    public float StartSpeed = 5f;
    //乒乓球的最大速度
    public float MaxSpeed = 20f;
    //每次碰撞发生后，速度的增量
    public float SpeedIncrease = 0.25f;
    //乒乓球当前的速度
    private float currentSpeed;
    //乒乓球当前的运动方向
    private Vector2 currentDir;
    //是否重新发球
    private bool resetting = false;
    void Start()
    {
        //设置起始的球速
        currentSpeed = StartSpeed;
        //设置起始球的移动方向
        currentDir = Random.insideUnitCircle.normalized;
    }
    void Update()
    {
        //处于重新发球状态时，球体固定不动
        if( resetting )
            return;
```

```
30              //游戏中没有网络接入，即当前只有一个玩家的时候，不改变乒乓球的位置
31              if( Network.connections.Length == 0 )
32                  return;
33              //朝指定方向移动球体
34              Vector2 moveDir = currentDir * currentSpeed * Time.deltaTime;
35              transform.Translate( new Vector3( moveDir.x, 0f, moveDir.y ) );
36          }
37          void OnTriggerEnter( Collider other )
38          {
39              //与球场上下边界发生碰撞
40              if( other.tag == "Boundary" )
41              {
42                  //反转三维坐标中 Z 轴的值，二维坐标中 Y 轴的值
43                  currentDir.y *= -1;
44              }
45              //与球拍发生碰撞
46              else if( other.tag == "Player" )
47              {
48                  //反转三维坐标中 X 轴的值，二维坐标中 X 轴的值
49                  currentDir.x *= -1;
50              }
51              //与球场界限发生碰撞
52              else if( other.tag == "Goal" )
53              {
54                  //  重新发球
55                  StartCoroutine( resetBall() );
56                  //加分
57                  other.SendMessage(                                          "GetPoint",
SendMessageOptions.DontRequireReceiver );
58              }
59              //增加速度
60              currentSpeed += SpeedIncrease;
61              //检测乒乓球的速度，是否在指定的范围之内
62              currentSpeed = Mathf.Clamp( currentSpeed, StartSpeed, MaxSpeed );
63          }
64          IEnumerator resetBall()
65          {
66              //重新设置乒乓球的位置、速度，以及移动方向
67              resetting = true;
68              transform.position = Vector3.zero;
69              currentDir = Vector3.zero;
70              currentSpeed = 0f;
71              //等待 3 秒后，开始新一轮的比赛
72              yield return new WaitForSeconds( 3f );
73              Start();
74              resetting = false;
75          }
76          void OnSerializeNetworkView( BitStream stream )
77          {
78              //将乒乓球的位置、速度、移动方向数据串行化到网络上
79              if( stream.isWriting )
80              {
81                  Vector3 pos = transform.position;
```

```
82                  Vector3 dir = currentDir;
83                  float speed = currentSpeed;
84                  stream.Serialize( ref pos );
85                  stream.Serialize( ref dir );
86                  stream.Serialize( ref speed );
87              }
88              //从网络上读取乒乓球的位置、速度、移动方向数据
89              else
90              {
91                  Vector3 pos = Vector3.zero;
92                  Vector3 dir = Vector3.zero;
93                  float speed = 0f;
94                  stream.Serialize( ref pos );
95                  stream.Serialize( ref dir );
96                  stream.Serialize( ref speed );
97                  transform.position = pos;
98                  currentDir = dir;
99                  currentSpeed = speed;
100             }
101         }
102 }
```

新添加的代码，都使用详细的注释说明了，这里就不重复说明了。

（2）赋予游戏场景中的 Ball 对象 Network View 组件，如图 1-36 所示。

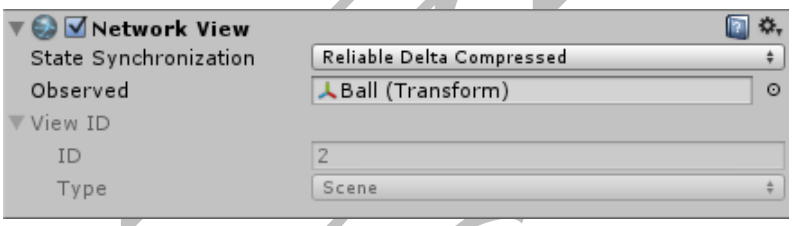

图 1-36　Network View 组件及其属性

（3）运行游戏的时候，乒乓球就会固定在视图的中心一动不动，因为此时游戏里只有一个玩家而已，如图 1-37 所示。

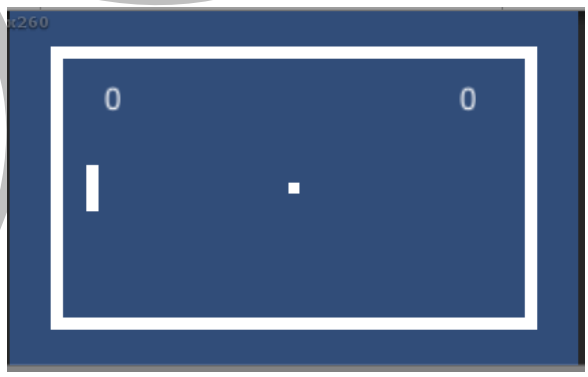

图 1-37　只有一个玩家的时候，位于游戏视图中心的乒乓球，一动不动

1.6.5　游戏分数的网络化

游戏的分数有自己的增加逻辑，即乒乓球进入对方球拍后的界限处，并与界限发生碰撞的时候，一方得 1 分。分数的增加，以及输赢的逻辑，是通过调用 Scorekeeper 脚本里定义的 AddScore()方法来实现的，因此要将游戏分数网络化，就必须使用本章学习的知识，远程调用 Scorekeeper 脚本里定义的 AddScore()方法才行。

打开脚本 Scorekeeper，添加实现对应功能的代码，如下：

```
using UnityEngine;
using System.Collections;

public class Scorekeeper : MonoBehaviour
{
    //表示获胜的分数
    public int ScoreLimit = 10;
    //表示 Paddle_P1 出现的位置
    public Transform SpawnP1;
    //表示 Paddle_P2 出现的位置
    public Transform SpawnP2;
    //表示要实例化的球拍
    public GameObject paddlePrefab;
    //Player1 的分数
    private int p1score = 0;
    // Player2 的分数
    private int p2score = 0;
    //表示 p1Score 的 Text Mesh 组件
    public TextMesh Player1ScoreDisplay;
    //表示 p2Score 的 Text Mesh 组件
    public TextMesh Player2ScoreDisplay;
    void Start()
    {
        if( Network.isServer )
        {
            //对于开始了一个新游戏的玩家而言，为他实例化 Paddle_P1 球拍
            Network.Instantiate( paddlePrefab, SpawnP1.position, Quaternion.identity,
0 );
            //只有一个玩家的时候，就让 Player2 的分数显示“Waitting…”
            Player2ScoreDisplay.text = "Waiting...";
        }
    }
    //有玩家与此游戏建立了连接以后，Unity 会主动调用此函数
    void OnPlayerConnected( NetworkPlayer player )
    {
        networkView.RPC( "net_DoSpawn", player, SpawnP2.position );
        //另一玩家加入游戏的时候，就让 Player2 的分数显示“0”
        Player2ScoreDisplay.text = "0";
    }
    [RPC]
    void net_DoSpawn( Vector3 position )
    {
        //对于加入游戏的玩家而言，为他实例化 Paddle_P2 球拍
        Network.Instantiate( paddlePrefab, position, Quaternion.identity, 0 );
```

```
44          }
45          //游戏的过程中，Player2 离开了游戏
46          void OnPlayerDisconnected( NetworkPlayer player )
47          {
48              //重置 Player1 和 Player2 的分数为 0
49              p1score = 0;
50              p2score = 0;
51              //修改 Player1 和 Player2 的分数显示，即 Player1 的显示“0”，Player2 的显示“Waitting…”
52              Player1ScoreDisplay.text = p1score.ToString();
53              Player2ScoreDisplay.text = "Waiting...";
54          }
55          //断开与服务器的连接以后，进入游戏的主菜单
56          void OnDisconnectedFromServer( NetworkDisconnection cause )
57          {
58              Application.LoadLevel( "Menu" );
59          }
60          //加分操作
61          public void AddScore( int player )
62          {
63              networkView.RPC( "net_AddScore", RPCMode.All, player );
64          }
65          [RPC]
66          public void net_AddScore( int player )
67          {
68              //为 Player 1 加 1 分
69              if( player == 1 )
70              {
71                  p1score++;
72              }
73              //为 Player 2 加 1 分
74              else if( player == 2 )
75              {
76                  p2score++;
77              }
78              //检查是否有玩家达到了胜利的分数
79              if( p1score >= ScoreLimit || p2score >= ScoreLimit )
80              {
81                  // player 1 分数大于 player 2
82                  if( p1score > p2score )
83                      Debug.Log( "Player 1 wins" );
84                  // player 2  分数大于  player 1
85                  if( p2score > p1score )
86                      Debug.Log( "Player 2 wins" );
87                  //分数相同
88                  else
89                      Debug.Log( "Players are tied" );
90                  //开始新一轮的比赛
91                  p1score = 0;
92                  p2score = 0;
93              }
94              //修改 p1Score 显示的文本内容
95              Player1ScoreDisplay.text = p1score.ToString();
```

```
96              //修改 p2Score 显示的文本内容
97              Player2ScoreDisplay.text = p2score.ToString();
98          }
99      }
```

对于此脚本，需要说明的是：

- 脚本 29、37 行，实现了一个较人性化的效果，即当 Player2 未加入时，显示等待状态；
- 对脚本中 AddScore()方法的内容做了调整，令其远程调用 net_AddScore()方法，实现游戏分数的修改；

运行游戏，效果如图 1-38 所示，即 Player2 未加入游戏，分数处显示当前处于等待状态。

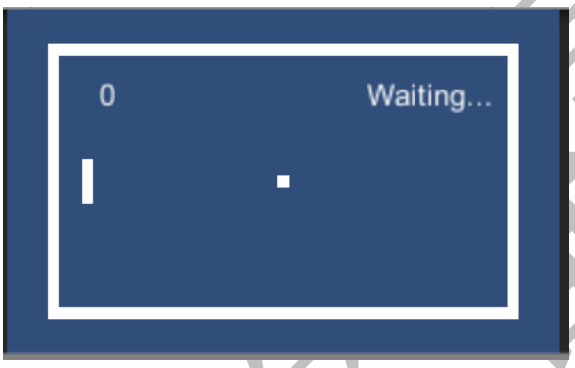

图 1-38　游戏分数的网络化

1.6.6　加入游戏

游戏里面应该有两个游戏场景，首次出现的是“菜单”场景，在此场景上玩家可以决定当前是要创建一个游戏，还是要加入一个游戏。本小节的任务就是添加这样一个场景，并实现对应的功能，具体的操作步骤如下：

（1）将游戏项目中一直在被操作的场景命名为 Game，此场景是玩家游戏时的场景。然后新建一个场景，命名为 Menu。进入 Menu 场景。

（2）在 Project 视图里，新建一个 C#脚本，命名为 ConnectToGame，为此脚本添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class ConnectToGame : MonoBehaviour
05  {
06      //表示要加入的游戏的主机的 IP 地址
07      private string ip = "";
08      //表示要加入的游戏的主机的端口号
09      private int port = 25005;
10      void OnGUI()
11      {
12          //在游戏视图上绘制 IP 输入区域
13          GUILayout.Label( "IP Address" );
14          ip = GUILayout.TextField( ip, GUILayout.Width( 200f ) );
15          //在游戏视图上绘制端口输入区域
16          GUILayout.Label( "Port" );
```

```
17              string port_str = GUILayout.TextField( port.ToString(), GUILayout.Width( 100f ) );
18              int port_num = port;
19              if( int.TryParse( port_str, out port_num ) )
20                  port = port_num;
21              //依据 IP 地址和端口号，加入到游戏中
22              if( GUILayout.Button( "Connect", GUILayout.Width( 100f ) ) )
23              {
24                  Network.Connect( ip, port );
25              }
26              //依据端口号创建一个游戏，并等待其它玩家的加入
27              if( GUILayout.Button( "Host", GUILayout.Width( 100f ) ) )
28              {
29                  Network.InitializeServer( 1, port, true );
30              }
31          }
32          void OnConnectedToServer()
33          {
34              Debug.Log( "Connected to server" );
35              //成功加入到游戏中以后，切换到 Game 场景
36              NetworkLevelLoader.Instance.LoadLevel( "Game" );
37          }
38          void OnServerInitialized()
39          {
40              Debug.Log( "Server initialized" );
41              //成功创建游戏以后，切换到 Game 场景
42              NetworkLevelLoader.Instance.LoadLevel( "Game" );
43          }
44      }
```

此脚本会在游戏视图上，绘制出如图 1-39 所示的窗口界面。

图 1-39　脚本所绘制的窗口界面

对于此脚本需要说明的是：

- 如果是要加入一个游戏，就需要知道游戏所在主机的 IP 地址和端口号，然后调用 Network.Connect()方法加入游戏即可，即脚本 24 行的代码；
- 如果是要创建一个游戏，则只需要端口号即可，然后调用 Network.InitializeServer()方法即可，即脚本 29 行的代码；
- 脚本 36、42 行，用来载入场景所使用的方法 NetworkLevelLoader.Instance.LoadLevel()，是我们自定义的方法，下面就会来说明；

（3）在 Project 视图里，新建一个 C#脚本，命名为 NetworkLevelLoader，为此脚本添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class NetworkLevelLoader : MonoBehaviour
05  {
06      public static NetworkLevelLoader Instance
07      {
08          get
09          {
10              if( instance == null )
11              {
12                  GameObject go = new GameObject( "_networkLevelLoader" );
13                  //避免此对象在 Hierarchy 视图里显示，从而造成混乱
14                  go.hideFlags = HideFlags.HideInHierarchy;
15                  instance = go.AddComponent<NetworkLevelLoader>();
16                  //直到新的场景载入以后，才销毁此对象
17                  GameObject.DontDestroyOnLoad( go );
18              }
19              return instance;
20          }
21      }
22      private static NetworkLevelLoader instance;
23      public void LoadLevel( string levelName, int prefix = 0 )
24      {
25          StopAllCoroutines();
26          StartCoroutine( doLoadLevel( levelName, prefix ) );
27      }
28      IEnumerator doLoadLevel( string name, int prefix )
29      {
30          Network.SetSendingEnabled( 0, false );
31          Network.isMessageQueueRunning = false;
32          Network.SetLevelPrefix( prefix );
33          Application.LoadLevel( name );
34          yield return null;
35          yield return null;
36          Network.isMessageQueueRunning = true;
37          Network.SetSendingEnabled( 0, true );
38      }
39  }
```

对于网络游戏，载入游戏场景的正确流程是：首先禁用网络队列（network queue），然后加载游戏场景，等待 2 帧，最后再启用网络队列，即上面脚本 23~38 行所实现的功能。

（4）将脚本 ConnectToGame 和 ExampleUnityNetworkingConnectToMasterServer，赋予游戏场景 Menu 中的 Main Camera 对象，加入游戏的功能就完成了。

1.6.7　网络对战功能演示

要演示网络对战的功能，就需要有两个游戏程序被运行才行，其中一个负责创建游戏，而另一个负责加入游戏。具体的操作方法如下：

（1）在 Unity 中单击 File|Build Settings…命令，打开 Build Settings 对话框，将游戏项目中的两个游戏场景拖动到对话框的 Scenes In Build 列表框中，如图 1-40 所示。最后，单

击 Build 按钮，即可编译出一个当前游戏项目的可执行文件。

提示：排在前面的游戏场景，在游戏运行时会被首先载入。

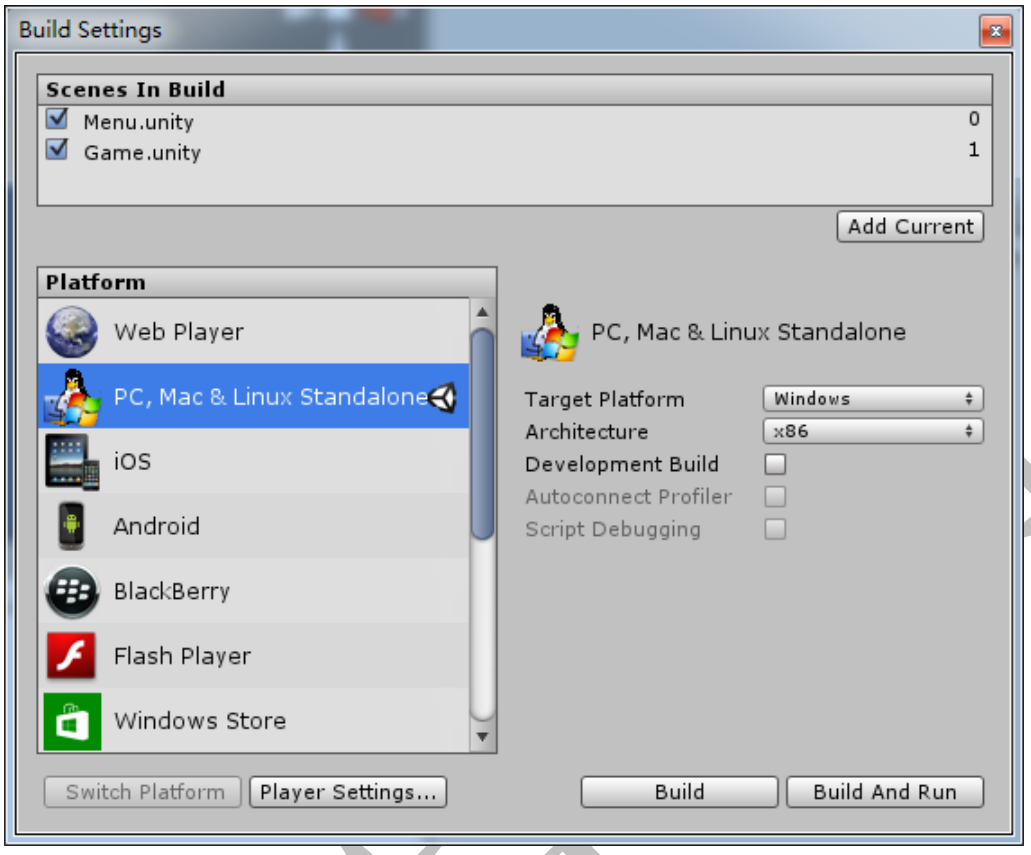

图 1-40　Build Settings 对话框

（2）先打开 MasterServer.exe 和 Facilitator.exe。然后再运行编译好的游戏可执行文件。游戏首先进入的是 Menu 场景，如图 1-41 所示，端口号默认设置了 25005，单击 Host 按钮，即可完成游戏的创建。

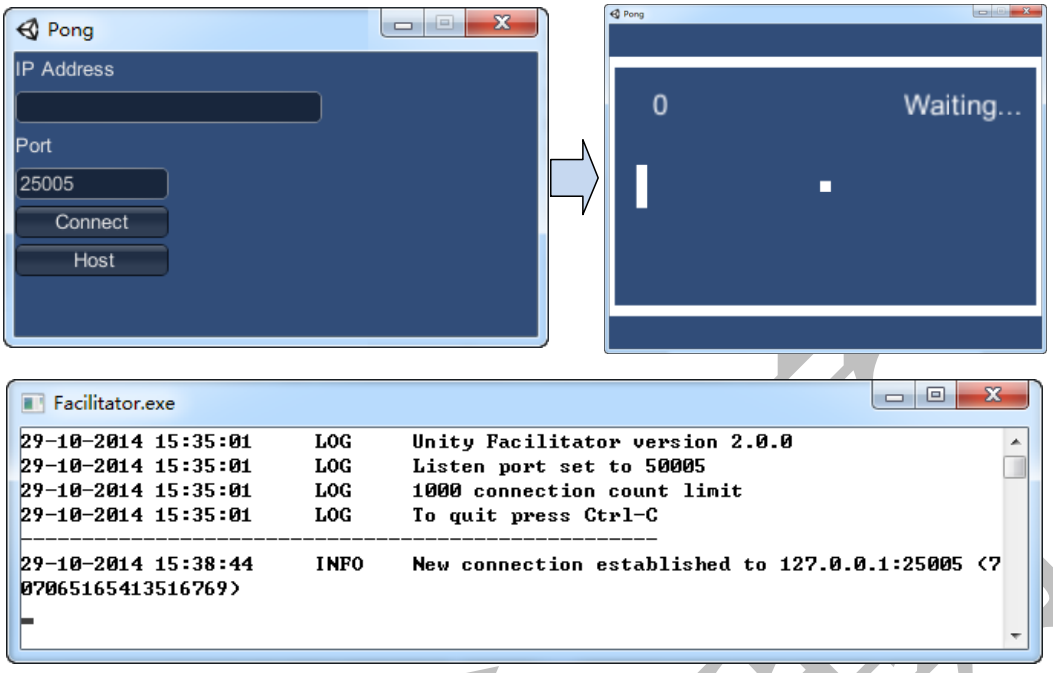

图 1-41　创建游戏

（3）继续运行一个编译好的的游戏可执行文件，即再运行一个游戏实例。同样进入的是 Menu 场景，IP 地址填写 127.0.0.1，端口号不变还是 25005，然后单击 Connect 按钮，就可以加入到上一个游戏实例创建的游戏了，如图 1-42 所示。

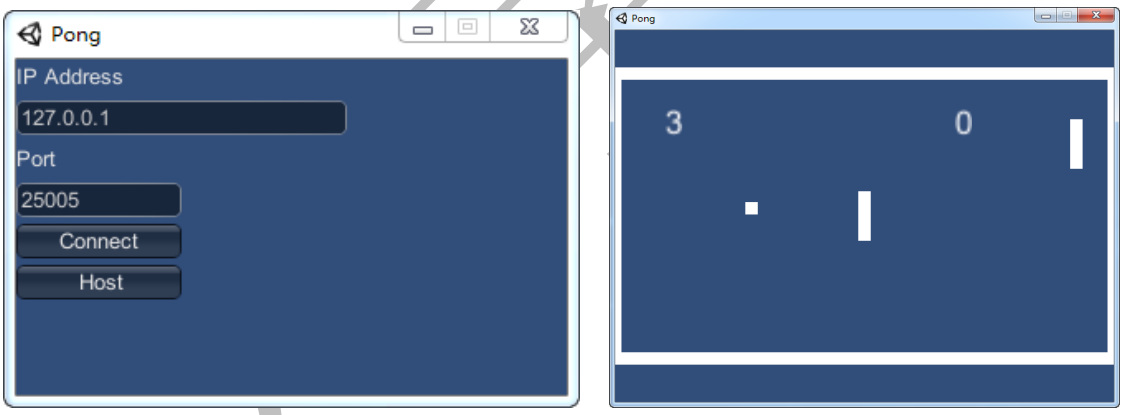

图 1-42　网络对战功能演示

提示：因为这两个实例同时运行在一台电脑上，所以一次只能激活一个窗口。处于非活动状态的游戏窗口画面就会处于静止状态，而活动窗口中一个球拍会在游戏视图的中央显示，这是因为另一个玩家没有任何输入，默认就令其处于（0,0,0）的位置上。

1.7　Unity 自带网络功能——模型示意图

如图 1-43 所示，是本章所讲解的网络模型。本章所涉及的全部网络相关内容，就是围绕此示意图依次进行说明的。

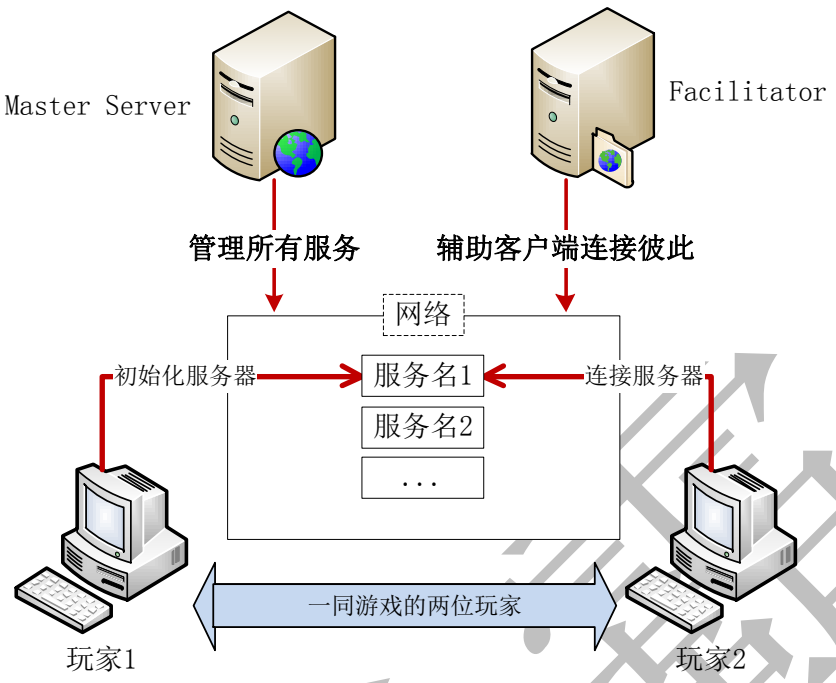

图 1-43　网络模型示意图