# Unity官网多人联网游戏入门教程译文（Multiplayer Networking）

###### 1. 简介

多人联网的工作比较琐碎、复杂，世界各地各种各样的计算机上不同的工程实例之间的同步与通信会出现许多问题。

在Unity内置的多人联网功能（Multiplayer Networking）与高级应用程序接口（High Level API, HLAPI）的帮助下，创建多人工程会变得更容易。

这个简单的多人联网案例展示了从头开始多人联网工程的创建，所用的资源与脚本也比较简单，希望能够简要介绍Multiplayer Networking与High Level API。

这个文档按步骤创建了一个多人联网工程，其基础思想适用于许多其他的更复杂的多人游戏。工程完成的效果是，两个玩家通过各自的实例，分别控制他们的游戏物体，授权的服务器控制与同步动作。玩家们可以互相射击，打败其他玩家与静止的敌人；如果自己被打败则自动复位。

这个实例教学针对中级用户，最好读过Multiplayer Networking的相关手册，特别是关于 NetworkingOverview、The HighLevel API、NetworkSystem Concepts。

首先：

在Unity3D中创建一个新的3D工程。 保存默认场景命名为“Main”。

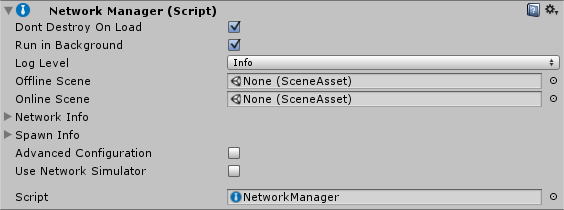
###### 2. Network Manager（网络管理器）

在这一节，我们会创建一个新的Network Manager，掌控这个多玩家工程的状态，包括游戏状态控制、派生管理、场景管理、配对以及信息调试。对于高级用户，可以从NetworkManager中产生一个类来定制化这个组件，这一节暂不讨论。

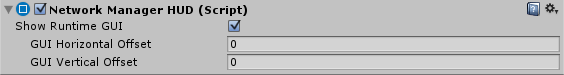
为了创建一个Network Manager，我们创建一个新的游戏物体，添加NetworkManager与NetworkManagerHUD组件。步骤如下：

  1.创建一个空游戏物体。NetworkManager”。添加组件NetworkManager＼NetworkManagerHUD。

NetworkManager组件管理游戏的网络状态。​

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DIjlfPT95)

NetworkManagerHUD连同NetworkManager一起工作，在运行时提供了一个更简单的用户界面来控制游戏的网络状态。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DIo0o6Jc1)

运行时，NetworkManagerHUD如下图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DIppu0Iac)

###### 3. 创建玩家预制件

默认状态下，NetworkManager组件会为每一个连接到网络的玩家创建游戏物体，通过克隆玩家与之间然后将克隆体派生到游戏中。网络派生与玩家游戏物体在客户端上的同步将会在后续讨论。

此案例中，玩家游戏物体是一个简单的胶囊体，配置一个“护目镜”，以标明胶囊体的朝向。最终的玩家游戏物体如下图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DIPV7iE48)

创建玩家游戏物体：

1. 创建一个新的胶囊体。＂Player”.

2. 选中Player。添加一个小立方体为其子物体。＂Visor”。

3. 创建一个新的材质，命名为“Black”。将颜色改成黑色。

4. 设置Visor的材质为Black材质。

5. 选中Player。 添加组件Network NetworkIdentity。用于辨别物体是否在网络状态下，并且通知网络系统。

6. 勾选NetworkIdentity组件中的LocalPlayer Authority。

[IMG_257](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DIXKgYZ85)

让客户端能够控制这个游戏物体的动作。

用Player创建玩家预制件：

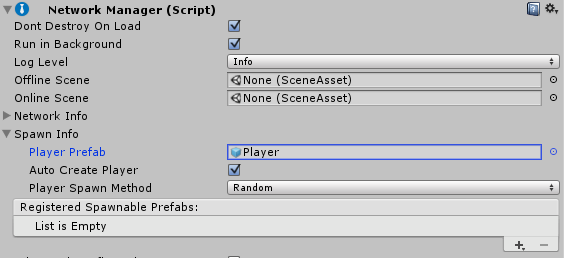
  7. 将Player拖入工程窗口， 在场景中删除Player，保存场景。

###### 4. 注册玩家预制件

创建玩家预制件之后，必须在网络系统中完成注册工作，Network Manager会用预制件往场景中派生玩家游戏物体。

  1. 在Hierarchy窗口选中NetworkManager游戏物体。

2. 将Player预制件拖入PlayerPrefab。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJ1Cdhi64)

NetworkManager组件用于管理联网游戏物体的派生，包括玩家物体。大部分游戏都有独立的预制件来代表所有玩家，NetworkManager为每一个玩家预制件都安排了位置。每个玩家客户端加入主机的时候都会创建新的游戏物体来代表玩家。

###### 5.玩家移动

为了实现玩家移动的功能，我们采用一个新的脚本叫做“PlayerController”。首先，这个脚本不用到联网的代码，所以只能在单机状态下运行。

   1. 在Player预制件上添加“PlayerController”脚本。

using UnityEngine;

public class PlayerController : MonoBehaviour{

void Update() {

var x = Input.GetAxis("Horizontal") \* Time.deltaTime \* 150.0f;

var z = Input.GetAxis("Vertical") \* Time.deltaTime \* 3.0f;

transform.Rotate(0, x, 0);

transform.Translate(0, 0, z);

}

}

   2. 保存脚本。 返回Unity，保存场景。

###### 6.在线测试玩家移动

测试主机上玩家的移动

到目前为止，Player能够在客户端移动，但是还没有联网。测试：

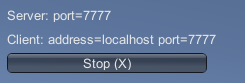
1. 进入运行模式。

NetworkManagerHUD的默认界面为：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJhtGar77)

2. 点击LAN Host按钮，使其成为Host。

NetworkManager会在场景中创建一个新的游戏物体，NetworkManagerHUD会改变，显示服务器正在工作。此时玩家端以Host的方式运行，意味着服务器与客户端在同一进程。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJjWoKz27)

3. 按住WASD或者上下左右箭头移动玩家物体。

4. 点击Stop按钮可以结束host。 退出Unity的运行模式。

测试客户端玩家的移动

测试客户端玩家移动需要游戏的两个实例同时运行，其中一个是Host。其中一个实例可以在编辑器运行，另一个可以在游戏创建的项目中运行，所以我们先要基于现有工程创建项目。

将Main场景添加到 build settings窗口。

6. 保存工程。

  7. 选择standalone application，创建单机项目并运行。  创建的单机项目运行时，勾选windowed，并选择一个相对较低的分辨率使得项目能够在编辑器旁边运行。 点击Host按钮，使其成为主机。

  8. 测试场景：按住WASD或者上下左右箭头移动玩家物体。

  9. 返回Unity。

  10. 进入Unity的运行模式。游戏在Unity的窗口中会显示NetworkManagerHUD的界面。

  11. 点击Unity场景中的LANClient按钮，作为客户端来连接主机。场景中会出现两个游戏物体，一个是主机上的本地玩家，另一个是连接上主机的客户端上的远程玩家。

测试客户端玩家游戏物体的移动：

  12. 按住WASD或者上下左右箭头移动玩家物体。注意两个游戏物体都用的是同样的输入方法。

  13. 转换到单机版玩家。

注意主机上的玩家游戏物体位置与客户端上面的并不太同步。

这是因为PlayerController 脚本并没有联网，两个玩家物体有着相同脚本，在每一个游戏实例中都处理相同的输入指令。主机与客户端都互相知晓对方的存在，但是玩家物体并没有与主机通信， NetworkManager没有追踪它们的位置，所以没有被同步。

  14. 关闭单机版游戏。 返回Unity，退出运行模式。

###### 7.将玩家移动联网

为了使网络知晓玩家移动，同时确保本地玩家只能控制本地的玩家游戏物体，我们需要更新PlayerController 脚本。

首先要用到命名空间UnityEngine.Networking，并且使PlayerController 脚本在NetworkBehaviour下产生，而不是 MonoBehaviour。

  1. 打开PlayerController 脚本。添加命名空间UnityEngine.Networking。

  2. 将 MonoBehaviour改为NetworkBehaviour。

  3. 在Update函数中添加isLocalPlayer的判断，这样的话就只有本地玩家处理本地输入。

最后的脚本：

using UnityEngine;

using UnityEngine.Networking;

public class PlayerController : NetworkBehaviour{

void Update() {

if (!isLocalPlayer) return;//这样的话就只有本地玩家处理本地输入。

var x = Input.GetAxis("Horizontal") \* Time.deltaTime \* 150.0f;

var z = Input.GetAxis("Vertical") \* Time.deltaTime \* 3.0f;

transform.Rotate(0, x, 0);

transform.Translate(0, 0, z);

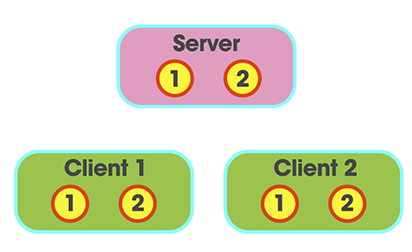
}

}

命名空间UnityEngine.Networking包含了我们需要的网络代码。 NetworkBehaviour是基于 MonoBehaviour的一个特殊的类，游戏物体上附属的联网脚本必须从NetworkBehaviour继承。

注意这个判断：!isLocalPlayer，LocalPlayer是NetworkBehaviour的一部分，所有源自NetworkBehaviour的代码会理解LocalPlayer的概念。

在一个网络状态下的工程里，服务器与所有的客户端都同时在相同的游戏物体上执行相同脚本的相同代码。这意味着，在有1个服务器与2个客户端的情况下，则可能有6个玩家游戏物体。因为有2个玩家，则服务器上有2个玩家游戏物体，而每个服务器上也各有2个玩家游戏物体，总共6个。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJBSGDAfe)

所有的游戏物体都由同样的预制件派生，并且分享PlayerController脚本。NetworkBehavior下的脚本能够自动理解玩家“拥有”什么样的游戏物体。LocalPlayer是指本地客户端所拥有的游戏物体，这样的所有权是由NetworkManager分配，当一个客户端连接到服务器时，本地客户端创造出来的实例会被标记为LocalPlayer，而其他的实例则不会。

通过isLocalPlayer的布尔型判断，脚本能够选择执行或者忽视某段代码，比如在我们的例子里有：

if (!isLocalPlayer) return;

这样的话只有本地玩家执行移动代码。在1个服务器与2个客户端的情况下，某个客户端游戏物体上的代码会被另一个客户端或者服务器直接返回，不会执行。

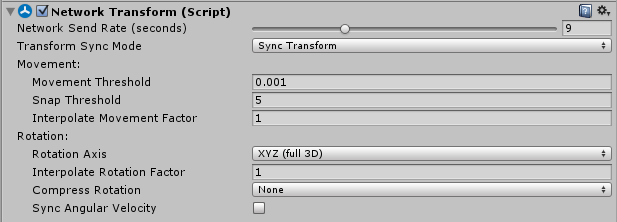
如果我们此时运行游戏，那么所有的玩家游戏物体状态不会同步，本地输入只会作用到本地游戏物体上，而物体的位置状态还没有通过网络实时更新。

为了使玩家游戏物体状态同步，我们需要给游戏物体预制件添加NetworkTransform组件：

  4. 保存脚本，返回Unity。

  5. 在project窗口选中玩家预制件。  添加组件 NetworkTransform。

  6. 保存工程。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJJLsKG36)

NetworkTransform组件负责同步游戏物体的transform性质，将本地玩家游戏物体的transform性质传送到服务器与其他客户端上。

###### 8.测试多玩家移动

为了测试多玩家移动，我们需要有两个游戏实例同时运行，Unity编辑器中的实例已经被更新了，我们需要更新单机版实例。

1.选择standalone application，创建单机项目并运行。

2.游戏在窗口中会显示NetworkManagerHUD的界面。

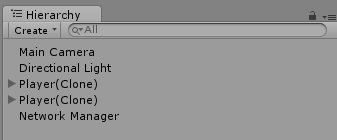
3.点击LAN Host按钮，使此实例成为主机。

我们能看到场景中游戏运行。

4.返回Unity，进入运行模式。

5.点击Unity场景中的LANClient按钮，作为客户端来连接主机。

场景中会出现两个游戏物体，一个是主机上的本地玩家，另一个是连接上主机的客户端上的远程玩家。只是目前它们处于同一位置，所以看起来像是一个。在Hierarchy窗口中，我们可以明显看到两个玩家克隆体。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJLiLH0a6)

测试场景中客户端玩家物体的移动：

   6.按住WASD或者上下左右箭头移动客户端玩家物体。

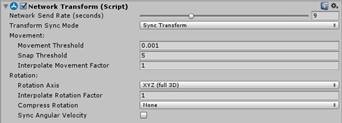
注意游戏物体现在是独立移动的。

    7.返回单机项目界面。按住WASD或者上下左右箭头测试主机上玩家物体的移动。

两个游戏物体能够独立移动，并且整个游戏状态在服务器与客户端上同步了。

在两个实例上运行的时候，本地客户端上的远程玩家游戏物体的移动不是很平滑顺畅，有一顿一顿的感觉，基于网络的应用都会有一定程度的延迟，服务器与客户端之间数据传输速度受限。

有许多方法可以优化延迟与数据传输。比如，NetworkTransform组件有Network Send Rate设置NetworkTransform发送同步数据的频率。服务器与客户端之间网络的刷新频率对多人游戏效果有很大影响。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJMQ3c67a)

更重要的是，有许多方法能够帮助网络应用的中的同步性差异，无论是因为sheer latency或者synchronizational updates。这些方法包括内插、外插，以及许多其他滤波或者预测的方式，此节暂不讨论。

在状态同步与游戏效果之间要寻求一个平衡，所有的联网游戏也并不是完全同步的。

8.关闭单机版。  返回Unity，退出运行模式。

###### 9.标识本地玩家

目前为止，玩家游戏物体区别开了，但是玩家还不能辨认它们，所以我们将本地的玩家游戏物体标识为蓝色。

1.打开PlayerController脚本编辑。  增加一个OnStartLocalPlayer 函数来改变玩家物体的颜色。

最终代码：

using UnityEngine;

using UnityEngine.Networking;

public class PlayerController : NetworkBehaviour{

void Update() {

if (!isLocalPlayer) return;

var x = Input.GetAxis("Horizontal") \* Time.deltaTime \* 150.0f;

var z = Input.GetAxis("Vertical") \* Time.deltaTime \* 3.0f;

transform.Rotate(0, x, 0);

transform.Translate(0, 0, z);

}

public override void OnStartLocalPlayer(){

GetComponent<MeshRenderer>().material.color = Color.blue;//改变玩家物体的颜色

}

}

这个函数只会在本地玩家物体上才会执行，所以每个玩家看见自己的游戏物体是蓝色的，OnStartLocalPlayer是理想的初始化地方，可以配置摄像机与输入方式。

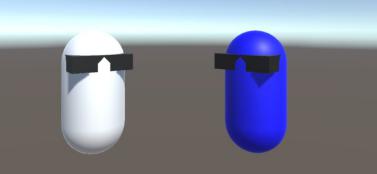
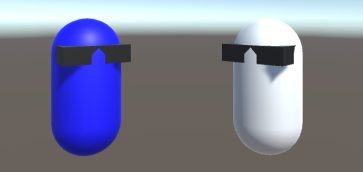
    2.保存脚本，返回Unity。

    3.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

    4.返回Unity，进入运行模式。  点击Unity场景中的LANClient按钮，作为客户端来连接主机。

本地玩家控制的游戏物体为蓝色，非本地玩家游戏物体为白色。

  5.关闭单机版。返回Unity，退出运行模式。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DK1WxM165)[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DJZHotD25)

###### 10.射击（单个玩家）

多人游戏中射击是一种常见功能，子弹在所有客户端都能正常工作。

这一节会增加射击功能，但只针对单个玩家，下一节会针对网络。

    1.创建球体，重命名为“Bullet”。设置scale为(0.2, 0.2,0.2)。添加Rigidbody，设置Use Gravity为false。

    2.将Bullet物体拖入工程，做成预制件。在场景中删除Bullet物体。   保存场景。

PlayerController脚本需要更新，运用 Bullet进行射击动作。 打开PlayerController。添加Bullet的public游戏物体变量.添加产生子弹位置的public变量。在Update函数中添加输入操作。添加Fire函数发射子弹。

最终代码如下：

using UnityEngine;

using UnityEngine.Networking;

public class PlayerController : NetworkBehaviour{

public GameObject bulletPrefab;

public Transform bulletSpawn;

void Update() {

if (!isLocalPlayer) return;

var x = Input.GetAxis("Horizontal") \* Time.deltaTime \* 150.0f;

var z = Input.GetAxis("Vertical") \* Time.deltaTime \* 3.0f;

transform.Rotate(0, x, 0);

transform.Translate(0, 0, z);

if (Input.GetKeyDown(KeyCode.Space)) Fire();

}

void Fire() {

var bullet = (GameObject)Instantiate(bulletPrefab, bulletSpawn.position, bulletSpawn.rotation);

bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward \* 6;

Destroy(bullet, 2.0f);

}

public override void OnStartLocalPlayer() {

GetComponent<MeshRenderer>().material.color = Color.blue;

}

}

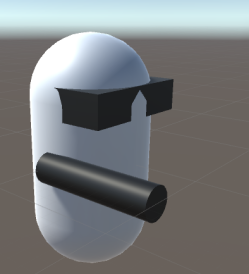
    3.保存脚本，返回Unity。

    4.将Player预制件拖入Hierarchy视图，  选中Player物体。

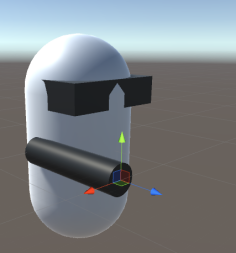
    5.创建一个圆柱体为Player的子物体，并命名为“Gun”。移除CapsuleCollider 组件。

    6.设置“Gun”的Position为(0.5, 0.0,0.5)，Rotation为(90.0, 0.0, 0.0)，Scale 为(0.25, 0.5, 0.25)。材质为“Black”。

Player如图所示：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKaAMWM7c)

    7.选中Player物体。 创建子物体“Bullet Spawn”。Position改为(0.5, 0.0,1.0)。Bullet Spawn的位置应该在Gun的末端。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKh8y8X83)

   8.用修改后的Player预制件替代project窗口中的原Player预制件. 删除场景中的Player，保存场景。

    9.选中Player预制件。  将Bullet预制件拖入PlayerController的BulletPrefab的区域中。   将Player的子物体BulletSpawn拖入PlayerController的BulletSpawn区域内。保存工程。

    10.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

    11.移动游戏物体，按空格键。返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

按空格键发现并没有子弹发射，因为子弹与发射代码并不是网络状态。

    12.关闭单机版。返回Unity，退出运行模式。

###### 11.多人射击

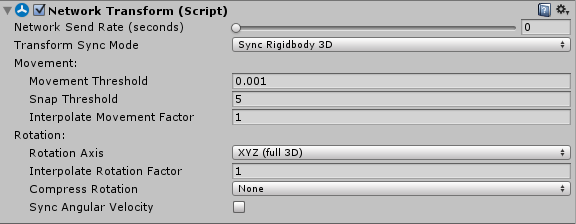
这一节给子弹添加网络功能，Bullet预制件与射击代码均需更新。

首先，Bullet预制件需用NetworkIdentity在网络中标识，以及NetworkTransform来同步位置状态，预制件也需要用NetworkManager 注册。

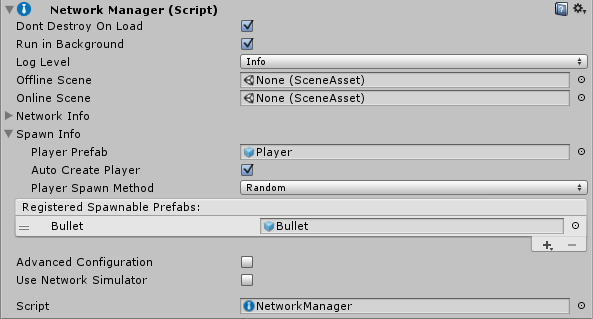
    1.在Project窗口选中Bullet预制件。

    2.添加组件：NetworkIdentity。NetworkTransform。设置Network Send Rate 为0。

子弹发射之后方向与速度都不会改变，所以不需要发送位置状态更新信息，每一个客户端都能计算出子弹的位置，设置Network Send Rate 为0使得位置信息不在网络中同步，减少运算量。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKmP0aw54)

    3.选中NetworkManager 。打开Spawn Info 折页。将Bullet预制件添加到Registered Spawnable Prefabs中去

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKo7J4W2e)

    5.打开PlayerController脚本编辑。

我们在之前使玩家物体移动的工作中讲到用Multiplayer Networking HLAPI搭建的工程结构，其原理就是使得服务器与客户端所有同样的游戏物体同时执行相同的代码。有一个方法解决服务器与客户端共享脚本的方法就是添加isLocalPlayer判断来区分哪些物体是可以执行脚本的。

另一个解决方法是运用[Command] 属性。运用[Command] 属性意味着以下函数被客户端调用，却在服务器上执行，函数参数自动传送到服务器。Command只能从本地玩家物体发送。启用一个网络命令的时候，函数前面必须加上“Cmd”。

    6.在Fire函数前加上[Command] 属性，使其成为网络中指令。函数名称加上前缀“Cmd”变成“CmdFire”。

7.Update中的函数同样改名为“CmdFire”。

下一个概念是Unity网络功能下的Network Spawning，在HLAPI中“Spawn”的概念不仅仅是简单的初始化。它意味着在服务器以及所有与之相连的客户端上创建游戏物体，然后这个游戏物体被Spawning系统管理，服务器上物体状态的更新信息会被传送到所有客户端，如果物体在服务器上被销毁了，那么在客户端上也将会被销毁。派生的游戏物体会被添加到服务器管理的联网游戏物体中去，这意味着如果有新的客户端加入，那么物体也会在新的客户端上派生。

 8.添加NetworkServer.Spawn函数来派生子弹。

最终脚本：

using UnityEngine;

using UnityEngine.Networking;

public class PlayerController : NetworkBehaviour{

public GameObject bulletPrefab;

public Transform bulletSpawn;

void Update(){

if (!isLocalPlayer) return;

var x = Input.GetAxis("Horizontal") \* Time.deltaTime \* 150.0f;

var z = Input.GetAxis("Vertical") \* Time.deltaTime \* 3.0f;

transform.Rotate(0, x, 0);

transform.Translate(0, 0, z);

if (Input.GetKeyDown(KeyCode.Space)) CmdFire();

}

[Command]//6.在Fire函数前加上[Command] 属性，使其成为网络中指令。

void CmdFire() {

var bullet = (GameObject)Instantiate(bulletPrefab, bulletSpawn.position, bulletSpawn.rotation);

bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward \* 6;

NetworkServer.Spawn(bullet);//8.添加NetworkServer.Spawn函数来派生子弹。

Destroy(bullet, 2.0f);

}

public override void OnStartLocalPlayer() {

GetComponent<MeshRenderer>().material.color = Color.blue;

}

}

    9.保存脚本，返回Unity。

    10.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

    11.移动游戏物体。

    12.返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

14.按空格键则所有的客户端上都会有子弹，每个玩家都能相互射击。现在子弹在另一个玩家身上反弹，并没有其他明显效果。

    15.关闭单机版。返回Unity，退出运行模式。

###### 12.玩家生命值（单机）

多人网络中同步玩家状态是一个重要的概念。

比如，子弹击中玩家会对玩家的生命状态产生影响，目前的生命值就是一个状态，需要进行网络同步。

子弹碰撞

第一步就是添加子弹的碰撞逻辑，比如，子弹碰撞到其他碰撞体的时候销毁子弹。

1.在工程窗口选中Bullet预制件，添加脚本命名为“Bullet”。添加碰撞逻辑。

using UnityEngine;

public class Bullet : MonoBehaviour{

void OnCollisionEnter() {

Destroy(gameObject);

}

}

    2.保存脚本，返回Unity。

    3.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

    4.移动游戏物体。

    5.返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

当Bullet碰撞到游戏物体的时候会自动销毁，因为Bullet由NetworkManager管理，如果服务器上的Bullet销毁，则所有客户端里面对应的Bullet也会销毁。

    7.关闭单机版。返回Unity，退出运行模式。

玩家生命值

我们需要一个新的脚本来记录与显示玩家的实时生命值。

1.在Player预制件添加脚本，并命名为“Health”。

      2.添加一个字段保存玩家最大生命值的常数，初始化为100。

    3.添加一个字段保存玩家实时生命值，初始化为最大值。

    4.子弹命中玩家时减少生命值。

最终代码：

using UnityEngine;

public class Health : MonoBehaviour{

public const int maxHealth = 100;//2.添加一个字段保存玩家最大生命值的常数，初始化为100。

public int currentHealth = maxHealth;//3.添加一个字段保存玩家实时生命值，初始化为最大值。

public void TakeDamage(int amount) {//4.子弹命中玩家时减少生命值

currentHealth -= amount;

if (currentHealth <= 0) {

currentHealth = 0;

Debug.Log("Dead!");

}

}

}

保存脚本。Bullet脚本需要更新，在子弹击中玩家的时候调用Health脚本中的TakeDamage函数。

    5.打开Bullet脚本编辑。碰撞处理中调用Health脚本中TakeDamage函数。

最终的Bullet脚本：

using UnityEngine;

using System.Collections;

public class Bullet : MonoBehaviour{

void OnCollisionEnter(Collision collision){

var hit = collision.gameObject;

var health = hit.GetComponent<Health>();

if (health != null) health.TakeDamage(10);

Destroy(gameObject);

}

}

    6.保存Bullet脚本。

子弹会对玩家造成伤害，但是这种伤害并不可见，我们需要为每个玩家创建一个血槽。

创建简易血槽

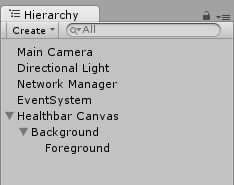
    1.在场景中创建新的Image。

    2.Canvas重命名“HealthbarCanvas”。Image重命名为“Background”。设置 Width 值100，Height值10。

    3.将Image组件中的Source Image设置为InputFieldBackground，Color设置为Red。

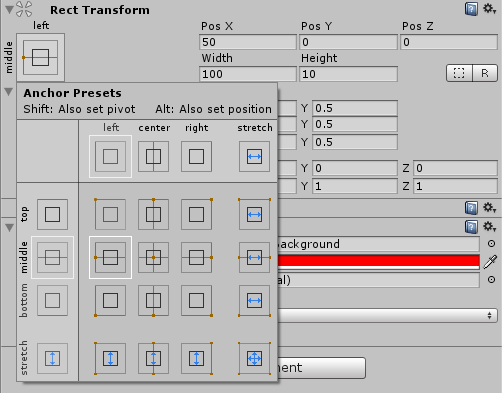
    4.复制Background，将复制后的物体重命名为“Foreground”。使Foreground成为Background的子物体。

游戏物体的关系如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKAwv1u18)

    5.选中“Foreground”，将Image组件中的Color设置为Green。

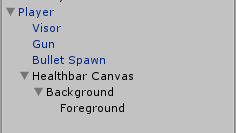
    6.打开AnchorPresets窗口，设置Pivot和Position为Middle Left。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKBMM4Nb9)

    7.将project窗口的Player预制件拖入Hierarchy。

    8.将Canvas的Render Mode改为World Space。

    9.将Healthbar Canvas变成Player的子物体。如图

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKGJ6xB73)

    10.选中Healthbar Canvas。重置RectTransform。scale为 (0.01, 0.01, 0.01)。Position为 (0.0, 1.5, 0.0)。

    11.选中Player，修改Player预制件。保存场景。

为了将Healthbar Canvas融合到目前的health与damage系统，我们需要在Health脚本中调用Healthbar，用Foreground Image来显示当前生命值。

   12.添加字段存储Healthbar的Foreground的RectTranform，因为我们要通过改变它的宽来显示生命值。

注意我们需要以二维变量的形式改变RectTransform的size Delta的宽。

最终代码如下：

using UnityEngine;

public class Health : MonoBehaviour{

public const int maxHealth = 100;

public int currentHealth = maxHealth;

public RectTransform healthBar;

public void TakeDamage(int amount) {

currentHealth -= amount;

if (currentHealth <= 0) {

currentHealth = 0;

Debug.Log("Dead!");

}

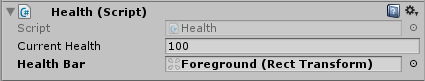
healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);//20.更新生命值显示，

}

}

    13.保存脚本，返回Unity。

    14.选中Hierarchy窗口的Player，将Health组件中的Healthbar设置为Foreground游戏物体。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKMhuoece)

    15.选中Player，应用于Player预制件（注意同时改变NetworkManager中消失的Player与Bullet等再补上）。

    16.删除场景中的Player，保存场景。

最后，我们需要使Healthbar始终面向摄像机。

    17.在Project中选中Player预制件上的HealthBarCanvas。添加脚本“Billboard”。增加使Healthbar面向主摄像机的代码。

最终脚本如下：

using UnityEngine;

public class Billboard : MonoBehaviour{

void Update() {

transform.LookAt(Camera.main.transform);

}

}

测试场景：

    18.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。  
移动游戏物体。

    19.返回Unity，进入运行模式。

    20.点击Unity场景中的LANClient按钮，作为客户端来连接主机。

目前，所有服务器与客户端上的玩家生命值全部应用。但一个玩家射击另一个的时候，子弹与Health脚本独自在服务器与客户端上运行，没有同步。子弹通过Network Manager当做Spawned物体管理，碰到碰撞体时自动消失。因为子弹是同步的，而玩家当前生命值没有同步，所以导致玩家当前生命值在服务器与客户端上都不一样。

    29.关闭单机版。返回Unity，退出运行模式。

###### 13.网络化生命值

玩家当前生命值应该应用在服务器上，然后再客户端同步，这叫做Server Authority。

为了让我们的生命值与伤害系统在网络状态与Server Authority下工作，需要用到State Synchronization和一个特殊变量叫SyncVars。网络同步变量SyncVars，需要用到[SyncVar]属性。

    1.打开Health脚本编辑。使脚本在NetworkBehaviour中进行。

    2.使currentHealth增加[SyncVar]。

  3.增加“isServer”的判断来接受TakeDamage函数，这样的话damage只能对服务器进行。

最终脚本：

using UnityEngine;

using UnityEngine.Networking;

public class Health : NetworkBehaviour{

public const int maxHealth = 100;

[SyncVar]//让我们的生命值与伤害系统在网络状态与Server Authority下工作

public int currentHealth = maxHealth;

public RectTransform healthBar;

public void TakeDamage(int amount){

if (!isServer) return;//这样的话damage只能对服务器进行

currentHealth -= amount;

if (currentHealth <= 0) {

currentHealth = 0;

Debug.Log("Dead!");

}

healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);

}

}

    4.保存脚本，返回Unity。

    5.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

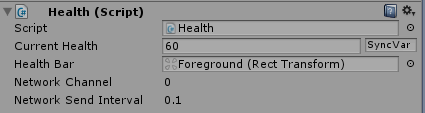
   移动游戏物体。

6.返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

目前，所有服务器与客户端上的玩家生命值全部应用。但一个玩家射击另一个的时候，子弹与Health脚本独自在服务器与客户端上运行，没有同步。子弹通过Network Manager当做Spawned物体管理，碰到碰撞体时自动消失。因为子弹是同步的，而玩家当前生命值没有同步，所以导致玩家当前生命值在服务器与客户端上都不一样。

    7.关闭单机版。返回Unity，退出运行模式。

现在玩家生命值能在服务器上响应并且在所有客户端上同步。可以在检视面板中看到currentHealth的值在网络状态下同步。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DKX8bke82)

但是Healthbar只在服务器上工作，没有在其他客户端上同步。因为我们没有在网络上同步Foreground的RectTransform，设置Healthbar的Size Delta的代码只在服务器上运行，所以Healthbar在客户端上没有改变。

我们需要同步Foreground的RectTransform。

    8.关闭单机项目，返回Unity，退出运行模式。

状态同步用到的另一个工具：SyncVar hook，SyncVar hook会通过一个函数联系SyncVar。这些函数在SyncVar的值改变时在服务器与客户端上调用。

  9.打开Health脚本。将改变Healthbar的代码挪入函数OnChangeHealth。

值得注意的是这个函数必须有一个和[SyncVar]属性中的变量类型一样的变量，这个例子里面是int currentHealth，SyncVar的值会被传送到hook函数中去。

    10.为SyncVar属性中设置currentHealthhook。

最终代码：

using UnityEngine;

using UnityEngine.Networking;

public class Health : NetworkBehaviour{

public const int maxHealth = 100;

[SyncVar(hook = "OnChangeHealth")]//为SyncVar属性中设置currentHealthhook。

public int currentHealth = maxHealth;

public RectTransform healthBar;

public void TakeDamage(int amount) {

if (!isServer) return;

currentHealth -= amount;

if (currentHealth <= 0) {

currentHealth = 0;

Debug.Log("Dead!");

}

}

void OnChangeHealth(int health) {

healthBar.sizeDelta = new Vector2(health, healthBar.sizeDelta.y);

}

}

现在如果currentHealth的值改变了，那么OnChangedHealth会被服务器调用，所有的客户端会更新Healthbar。

    11.保存脚本，返回Unity。

    12.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

   移动游戏物体。

    13.返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

如果玩家之间相互射击，所有的Healthbar会显示玩家的当前生命值，并在网络同步。

    14.关闭单机版。返回Unity，退出运行模式。

###### 14.死亡与复活

除了控制台里的提示之外，游戏没有在玩家生命值降为零的时候做出其他反应。现在我们需要让玩家生命值为零的时候迅速移动到初始位置复活。

我们要介绍另一种状态同步的方法：[ClientRpc]属性。

ClientRpc可以被任何服务器上派生的有NetworkIdentity的游戏物体调用。虽然这个函数被服务器调用，但是在所有的客户端上执行。ClientRpc与一般命令恰恰相反，一般的命令在客户端上调用，在服务器上执行。

要调用ClientRpc，用到[ClientRpc]属性并在函数名称前加“Rpc”前缀，这个函数在被服务器调用之后就会在客户端上运行，所有变量会被传送到客户端。我们需要在Health脚本中创建一个Respawn函数并且在TakeDamage中调用（当玩家生命值为零时）。

  1.打开Health脚本编辑。创建一个带着[ClientRpc]属性与Rpc前缀的函数Respawn。

在TakeDamage中如果玩家的currentHealth到了零，我们需要使玩家复活。

    2.改变TakeDamage中的代码，重置currentHealth到最大。

    3.将Debug.Log替代成调用RpcRespawn函数。

最终代码：

using UnityEngine;

using UnityEngine.Networking;

public class Health : NetworkBehaviour{

public const int maxHealth = 100;

[SyncVar(hook = "OnChangeHealth")]

public int currentHealth = maxHealth;

public RectTransform healthBar;

public void TakeDamage(int amount) {

if (!isServer) return;

currentHealth -= amount;

if (currentHealth <= 0) {

currentHealth = maxHealth;//2.改变TakeDamage中的代码，重置currentHealth到最大。

RpcRespawn();//3.将Debug.Log替代成调用RpcRespawn函数。

}

}

void OnChangeHealth(int currentHealth){

healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);

}

[ClientRpc]//创建一个带着[ClientRpc]属性与Rpc前缀的函数Respawn。

void RpcRespawn(){

if (isLocalPlayer)

            transform.position = Vector3.zero;

}

}

在我们的案例中客户端可以控制本地玩家物体的位置，因为玩家物体在客户端上有本地授权。

如果在玩家生命值降为零的时候，服务器设置玩家物体到原始位置，那么客户端会覆盖服务器，因为客户端上的本地授权。

所以我们用ClientRpc函数来指导对应的Client移动玩家物体，然后因为NetworkTransform，物体的位置在其他客户端同步。

    4.保存脚本，返回Unity。

    5.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

6.移动游戏物体。

    7.返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

如果一方玩家生命值降为零，那么会重新出现在初始位置且生命值恢复。

    8.关闭单机版。返回Unity，退出运行模式。

###### 15.非玩家游戏物体

这一节我们讨论非玩家游戏物体比如说敌人。

客户端连接服务器时，玩家物体被派生，敌人物体需要由服务器控制。

这一节我们会增加一个EnemySpawner创建非玩家游戏物体，使其能够被玩家击中并且杀死。

    1.创建新游戏物体”Enemy Spawner”。添加组件NetworkIdentity. 设置ServerOnly使其为true。

将Server Only设置为true，可阻止Enemy Spawner被发送到客户端上去。

    2.选中EnemySpawner，创建新脚本“EnemySpawner”。

using UnityEngine;

using UnityEngine.Networking;

public class EnemySpawner : NetworkBehaviour{

public GameObject enemyPrefab;

public int numberOfEnemies;

public override void OnStartServer() {

for (int i = 0; i < numberOfEnemies; i++) {

var spawnPosition = new Vector3(Random.Range(-8.0f, 8.0f), 0.0f, Random.Range(-8.0f, 8.0f));

var spawnRotation = Quaternion.Euler(0.0f, Random.Range(0, 180), 0.0f);

var enemy = (GameObject)Instantiate(enemyPrefab, spawnPosition, spawnRotation);

NetworkServer.Spawn(enemy);

}

}

}

这段程序里面有一些值得注意的：

1.需要命名空间UnityEngine.Networking。

2.类要在NetworkBehaviour中进行。

3.要用到一个虚拟函数OnStartServer。

4.服务器开始运行时，一定数目的敌人会在随机位置随机方向，然后用NetworkServer.Spawn(enemy)派生。OnStartServer 与 OnStartLocalPlayer类似，OnStartServer 在服务器开始监听网络之际工作。

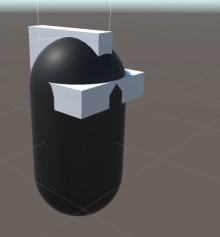
    3.保存脚本，返回Unity。

既然Enemy Spawner已经被创建，我们需要被派生的游戏物体。

为了简便，我们用player预制件作为enemy的基础，enemy同样需要NetworkIdentity和NetworkTransform组件，以及Health脚本与Healthbar。

4.将player改为enemy。重命名“Enemy”。删除Gun\BulletSpawn。移除PlayerController。将Enemy创建为新预制件。

    5.将Black Material应用于 Enemy。设置Visor材质为Default-Material。为Enemy创建一个cube子物体“Mohawk”。Position为(0.0. 0.55, -0.15),Scale为(0.2, 1.0, 1.0),移除BoxCollider组件。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DLb2Be078)

    6.将这些改变应用于Enemy预制件中。删除场景中的Enemy预制件。

    7.保存场景。

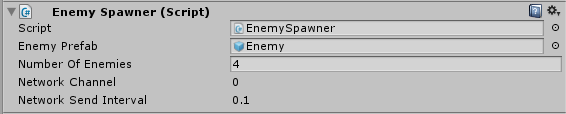
我们需要注册Enemy预制件，并且将Enemy预制件添加到Enemy Spawner里。

    8.保存场景。

    9.选中NetworkManager 。打开Spawn Info折页。在Registered Spawnable Prefabs列表中加入Enemy预制件。

    10.选中Hierarchy窗口中的Enemy Spawner 。将Project窗口中的Enemy预制件拖入EnemySpawner 空的区域。

    11.设置Number Of Enemies为4。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DLcA7G7cf)

   12.保存工程。

因为Enemy预制件有Health脚本，Bullet脚本应该探测到enemy的Health脚本，所以这里不需要改动。

    13.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。这个时候，第一个player与4个enemy被创建。

    14.移动游戏物体。  返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

第二个玩家被创建。射击enemy会让它们损失生命值，但是当它们的生命值为0的时候不会重生，因为所有enemy都不符合RpcRespawn中的isLocalPlayer。它们的生命值会被设置为最大，由服务器控制。

    15.关闭单机版。返回Unity，退出运行模式。

###### 16.消灭Enemy

Enemy的生命值为0的时候应该被销毁。

我们需要调整Health脚本来区分player与enemy，最简单的方式就是增加一个DestroyOn Death的判断。

    1.打开Health脚本编辑。为Destroy OnDeath的判断增加一个公共字段。

    2.在TakeDamage中，在生命值为零重生之前增加Destroy On Death的判断。

最终代码：

using UnityEngine;

using UnityEngine.Networking;

public class Health : NetworkBehaviour{

public const int maxHealth = 100;

public bool destroyOnDeath;//1.为Destroy OnDeath的判断增加一个公共字段。

[SyncVar(hook = "OnChangeHealth")]

public int currentHealth = maxHealth;

public RectTransform healthBar;

public void TakeDamage(int amount) {

if (!isServer) return;

currentHealth -= amount;

if (currentHealth <= 0) {

if (destroyOnDeath) {//2.生命值为零重生之前增加Destroy On Death的判断。

Destroy(gameObject);

}else{

currentHealth = maxHealth;

//called on the Server, will be invoked on the Clients

RpcRespawn();

}

}

}

void OnChangeHealth(int currentHealth){

healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);

}

[ClientRpc]

void RpcRespawn() {

if (isLocalPlayer) transform.position = Vector3.zero;

}

}

3.保存脚本, 返回Unity。

    4.选中project窗口中的enemy预制件。设置Destroy OnDeath为true。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002PIeJVzy74DLiGiPw72)

    5.保存工程，测试运行。

    6.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

这个时候，第一个player与4个enemy被创建。

    7.移动游戏物体。

    8.返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

9.射击enemy使其生命值为0，则enemy会被销毁，而玩家则会被重生。

    10.关闭单机版。返回Unity，退出运行模式。

###### 17.派生与复位

设置玩家的初始化位置

目前玩家都是在相同的初始位置被创建。游戏一开始，除非我们先挪动一个玩家，不然它们会处在同样的位置不容易辨识。所以理想状态下，玩家应该在不同的位置被派生。

NetworkStartPosition组件可以用来实现这项功能，因为它内置特性处理派生位置。为了创建独特的派生位置，我们需要创建两个新的游戏物体，每个都有NetworkStartPosition组件。

    1.创建一个新的空物体“Spawn Position 1”。添加组件NetworkStartPosition。Position为(3, 0, 0)。

    2.复制SpawnPosition 1，重命名为“Spawn Position 2”。设置Position为(-3, 0, 0)。

    3.选中Hierarchy窗口中的Network Manager。打开Spawn Info折页。将Player Spawn Method改为 Round Robin。

因为Spawn Position有NetworkStartPosition组件，所以NetworkManager会自动找到它。

然后NetworkManager会用这些物体的Transform Position以及NetworkStartPosition来派生新的游戏物体。

有两个玩家派生的方法：Random和Round Robin。

Random会用 NetworkSpawnPositions中的一个随机派生点，Round Robin会循环使用派生点。

测试：

    1.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

这个时候，第一个player与4个enemy被创建。

    2.移动游戏物体。

3.返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

现在游戏开始时会在不同的地点创建玩家物体。

    4.关闭单机版。返回Unity，退出运行模式。

设置复活位置

案例的最后一步是用Spawn Position物体上的NetworkStartPosition组件创建一个简易系统，来创建玩家再生的序列位置。这个步骤并不是必须的，只是为了让实例更加完整。

值得注意的是存储再生位置的更简单的方法是在Start函数中存储本地玩家的位置，这样的话Network Manager上的Round Robin Player Spawn Method的start position在某种性质上属于玩家。

我们需要创建一个序列，找到有NetworkStartPosition附属的所有游戏物体，将它们加入到序列中并用它们的Transform Position作为一个再生点。这和Network Manager中设置初始位置很类似，但是是以一种更简单的方式，我们没有用Round Robin，只是用了Random。

   1.打开Health脚本。添加NetworkStartPosition类型新的数组，来存储派生点。

   2.添加Start函数。在Start函数中增加游戏物体是否LocalPlayer的判断。

   3.添加逻辑找到有NetworkStartPosition组件的所有实例，并且将它们存储到 Spawn Point序列里面。

  4.在RpcRespawn函数isLocalPlayer的判断里，更新代码，用spawn points序列中的一个随机位置。

最终代码：

using UnityEngine;

using UnityEngine.Networking;

public class Health : NetworkBehaviour{

public const int maxHealth = 100;

public bool destroyOnDeath;

[SyncVar(hook = "OnChangeHealth")]

public int currentHealth = maxHealth;

public RectTransform healthBar;

private NetworkStartPosition[] spawnPoints;

void Start() {

if (isLocalPlayer)

spawnPoints = FindObjectsOfType(typeof(NetworkStartPosition)) as NetworkStartPosition[];

}

public void TakeDamage(int amount) {

if (!isServer) return;

currentHealth -= amount;

if (currentHealth <= 0){

if (destroyOnDeath){

Destroy(gameObject);

} else{

currentHealth = maxHealth;

RpcRespawn();

}

}

}

void OnChangeHealth(int currentHealth){

healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);

}

[ClientRpc]

void RpcRespawn(){

if (isLocalPlayer) {

Vector3 spawnPoint = Vector3.zero;

if (spawnPoints != null && spawnPoints.Length > 0)

spawnPoint = spawnPoints[Random.Range(0, spawnPoints.Length)].transform.position;

transform.position = spawnPoint;

}

}

}

6.保存脚本，返回Unity。

    7.选择standalone application，创建单机项目并运行。点击LAN Host按钮，使单机项目上的实例成为主机。

这个时候，第一个player与4个enemy被创建。

    8.移动游戏物体。 返回Unity，进入运行模式。点击Unity场景中的LANClient按钮，作为客户端来连接主机。

现在游戏开始时会在不同的地点创建玩家物体，玩家生命值降为0时，会随机在已有的一个派生位置上复活。

    9.关闭单机版。返回Unity，退出运行模式。

###### 18.简要总结

在这个例子中，我们囊括了一个简单的联网多人游戏所需要的大部分概念和组件。

我们用Multiplayer Networking High LevelAPI (HLAPI)创建工程的基本结构。使用HLAPI时，服务器与所有客户端都在同样的游戏物体上同时执行相同的代码。我们讨论了如何用HLAPI为Server、LocalPlayer和Client添加判断。

我们也讨论了HLAPI的RPC，包括了Commands和ClientRpc。我们探索了如何用 Commands工作，Command在客户端调用，在服务器上执行。ClientRpc在服务器调用，在客户端执行。

我们也讨论了SyncVars和SyncVar hooks。[SyncVar]属性里可以保存变量，当变量值随着SyncVar hook改变时调用函数。

我们讨论了如何用NetworkIdentity和NetworkTransform组件控制联网物体的Transform的同步。

案例也包括了编辑器里许多联网状态下的内置组件，包括NetworkManager、NetworkManagerHUD和the NetworkStartPosition。

我们希望这个案例能够帮助开始理解如何创建一个网络工程，并且迅速熟悉一些重要概念。