UE4C++RPC联网基础文档(持续更新)

简介 :

RPC , 全称 : Remote procedure calls 远程过程调用

相关文档 :

<http://api.unrealengine.com/CHN/Gameplay/Networking/index.html> 官方基础

<http://api.unrealengine.com/CHN/Gameplay/HowTo/Networking/ReplicateActor/index.html> 官方蓝图案例

## 1, Actor复制

<1> bNetLoadOnClient = bool

这个变量是给一开始就放置在场景中的对象使用的

如果bNetLoadOnClient 设置为true, 当客户端连接上服务端时, 客户端也会存在这个对象

如果bNetLoadOnClient 设置为false, 当客户端连接上服务端时, 客户端不会存在这个对象

SetReplicates无论是否为true都不会影响这个变量的作用

<1> SetReplicates(bool)

调用 SetReplicates(true) 设置Actor可以复制

调用 SetReplicates(false) 设置Actor不可以复制

当在服务端Spawn可复制的Actor时, 客户端会生成

当在客户端Spawn可复制Actor时, 客户端不会生成

## 2, 状态判断

(1) AActor的HasAuthority(), 返回true是说明Actor是该端创建的角色

在关卡蓝图或者是GameMode以及默认放在场景中的Actor等对象使用这个函数可以用来判断是否是服务端, 因为关卡蓝图和GameMode与默认放在场景中的对象可以看做是由服务端生成的

(2) Actor的角色判断

AActor 中都有个 ENetRole Role 变量是用来识别角色的 Actor 的身份的。ENetRole 的几个值：

ROLE\_None：

ROLE\_SimulatedProxy：这个actor是其他客户端在本机客户端的一个模拟代理

ROLE\_AutonomousProxy：这个actor是本机客户端的自己控制的角色

ROLE\_Authority：这个actor是服务器上的actor

示例代码 :

void AMyNetCharacter::BeginPlay()

{

Super::BeginPlay();

switch (Role)

{

case ROLE\_Authority: Flag = "ROLE\_Authority"; break;

case ROLE\_SimulatedProxy: Flag = "ROLE\_SimulatedProxy"; break;

case ROLE\_AutonomousProxy: Flag = "ROLE\_AutonomousProxy"; break;

default: Flag = "ROLE\_None";

}

}

(3) 是否是服务端判断, 不推荐使用HasAuthority()来判断

推荐使用GetWorld()->IsServer()或者GetNetMode()判断

(4) 端的判断

使用GetNetMode()函数可以获取端的属性ENetMode, 分类如下

NM\_Standalone 单独端, 单机游戏

NM\_DedicatedServer 专用服务器

NM\_ListenServer 监听服务器

NM\_Client 客户端

NM\_MAX 官方没有解释

参考文档 : <https://docs.unrealengine.com/en-us/Gameplay/Networking/Overview>

## 3, 联网变量

联网变量只有在服务端修改才会更新到服务端和所有客户端, 在客户端修改只会更新所在客户端, 对服务端和其他客户端没有影响

(1) 首先用UPROPERTY(Replicated, ......)定义变量, 如 :

UPROPERTY(Replicated)

FString Inventory;

(2) 必须在cpp文件重写GetLifetimeReplicatedProps方法, 如 :

void ARPCProjectCharacter::GetLifetimeReplicatedProps(TArray< FLifetimeProperty > & OutLifetimeProps) const

{

Super::GetLifetimeReplicatedProps(OutLifetimeProps);

......

}

注意, 重写这个方法不需要在.h文件进行声明

(3) 添加头文件 #include "UnrealNetwork.h"

在上面的重写方法里用宏注册变量到联网系统

DOREPLIFETIME(ARPCProjectCharacter, Inventory); // 所有端的对象都更新

DOREPLIFETIME\_CONDITION(ARPCProjectCharacter, Inventory, COND\_OwnerOnly); // 该属性仅发送至 actor 的所有者

DOREPLIFETIME\_CONDITION(ARPCProjectCharacter, Inventory, COND\_SkipOwner); // 该属性将发送至除所有者之外的每个连接

DOREPLIFETIME\_CONDITION(ARPCProjectCharacter, Inventory, COND\_Custom); // 该属性没有特定条件，但需要通过 SetCustomIsActiveOverride 得到开启/关闭能力

更多条件参考官方文档 :

<http://api.unrealengine.com/CHN/Gameplay/Networking/Actors/Properties/Conditions/index.html>

(4) RepNotify 和 ReplicatedUsing

参考文档 :

<https://www.cnblogs.com/timy/p/9890010.html>

<https://blog.csdn.net/maxiaosheng521/article/details/80770197>

功能是在可复制的Actor下定义某个值, 这个值绑定某个方法, 当在服务端这个值修改时, 绑定的方法会在客户端执行, 如果想要在服务端也执行绑定的方法, 需要再调用一次, 当在客户端修改这个值或者执行这个值绑定的方法时, 只会对所在的客户端有效果, 对服务端和其他客户端没有效果

如 :

.h文件

UPROPERTY(ReplicatedUsing = OnRep\_变量名)

bool 变量名;

UFUNCTION()

void OnRep\_变量名();

.cpp文件

//修改变量值, 修改变量名的话客户端会执行OnRep\_变量名()方法, 但不会执行服务端的方法

变量名 = true;

//执行变量方法, 会在服务端执行OnRep\_变量名()方法, 但是不会在客户端执行

OnRep\_变量名();

//所以如果要在服务端和客户端都执行, 必须两个语句一起写

## 4, 联网方法

参考文档 <https://blog.csdn.net/l346242498/article/details/75669807>

(1) Reliable 和 Unreliable

Reliable 一定会执行

Unreliable 在网络不好的时候可能会丢弃

(2) Client , Server, NetMulticast

Client : 如果在服务器运行, 会在拥有该Actor的客户端上调用; 如果在客户端调用,只会在当前客户端上执行

Server : 客户端调用时, 在服务器运行, 通常用于客户端给服务器传递数据, 服务端调用只在服务端运行

NetMulticast：在服务端调用, 会广播到所有客户端

(3) WithValidation 进行安全检查, 如果是使用Server函数, 一定要在UFUNCTION内添加这个声明, 并且在.h文件声明 void 方法名\_Implementation(var) 与 bool 方法名\_Validate(var) 方法并且在.cpp文件实现, 而方法名本身的方法不同实现, 其中\_Implementation实现逻辑, \_Validate内实现安全检验, 防止作弊, 但是有些类型不能作为参数, 比如FString, 编译会报错

案例 : <https://blog.csdn.net/zzk1995/article/details/50266729>

示例 :

.h文件

UFUNCTION(Server, Reliable, WithValidation)

void ServerMove(FVector Velocity, bool bSweep);

virtual bool ServerMove\_Validate(FVector Velocity, bool bSweep);

virtual void ServerMove\_Implementation(FVector Velocity, bool bSweep);

.cpp文件

bool ASWeapon::ServerHandleFiring\_Validate()

{

return true;//默认返回为true

}

void ASWeapon::ServerHandleFiring\_Implementation()

{

//具体的函数逻辑

}

## 5, UE监听服务器联网游戏流程

(1) GameMode

<1>GameMode只在服务端存在, 可以用来管理服务端对象和客户端对象

<2>创建项目时, 会自动生成 项目名GameMode 类, 如果没有在编辑器的WorldSetting中指定GameMode, 运行游戏后会自动使用这一个GameMode来作为游戏的GameMode

<3>GameMode的DefaultPawnClass需要自己设置, 如果不设置的话会默认给设置为DefaultPawn类, 如果要使用自己定义的角色类,需要把DefaultPawnClass设置为NULL, 然后在PostLogin以及Logout等函数进行角色的生成

<4>重写GameMode下的PostLogin函数, 在每次有新的端链接上服务端时(服务端创建时自己也会调用), 会调用这个额函数并且传入对应端的控制器PlayerController, 可以再这个方法里生成自定义的角色并且添加到对应控制器的Process让对应端进行控制, 这样子的话角色就都是在服务端进行生成的, 在服务端调用角色的HasAuthority()会返回true, 在客户端调用角色的HasAuthority()会返回false. 如果不重写这个PostLogin方法, 而是在WorldSetting下设定DefaultPawnClass为自定义的角色类, 那么客户端的角色依然是由服务端端生成的, 那么客户端调用角色的HasAuthority()会返回false, 在服务端调用角色的HasAuthority()会返回true;(推荐自己重写PostLogin进行处理)

<5>重写GameMode下的Logout函数, 可以处理客户端登出时的逻辑

<6>GameMode提供RestartPlayer函数允许寻找可用的PlayerStart点并且重新生成角色, 其最主要的目的就是寻找生成点, 可以在PostLogin中通过NewPlayer->GetPawn()是否存在, 如果存在的话, 可以销毁, 重写RestartPlayer函数, 销毁角色后调用RestartPlayer函数, 把NewPlayer(PlayerController)参数传入进行自定义的重新生成对象, 类似函数还有RestartPlayerAtPlayerStart以及RestartPlayerAtTransform, 如果自己书写生成点逻辑的话可以不用调用这个函数

(2) 使用exe后缀输入和open IP地址联网

//这里只讨论NM\_Standalone, NM\_ListenServer以及NM\_Client的情况

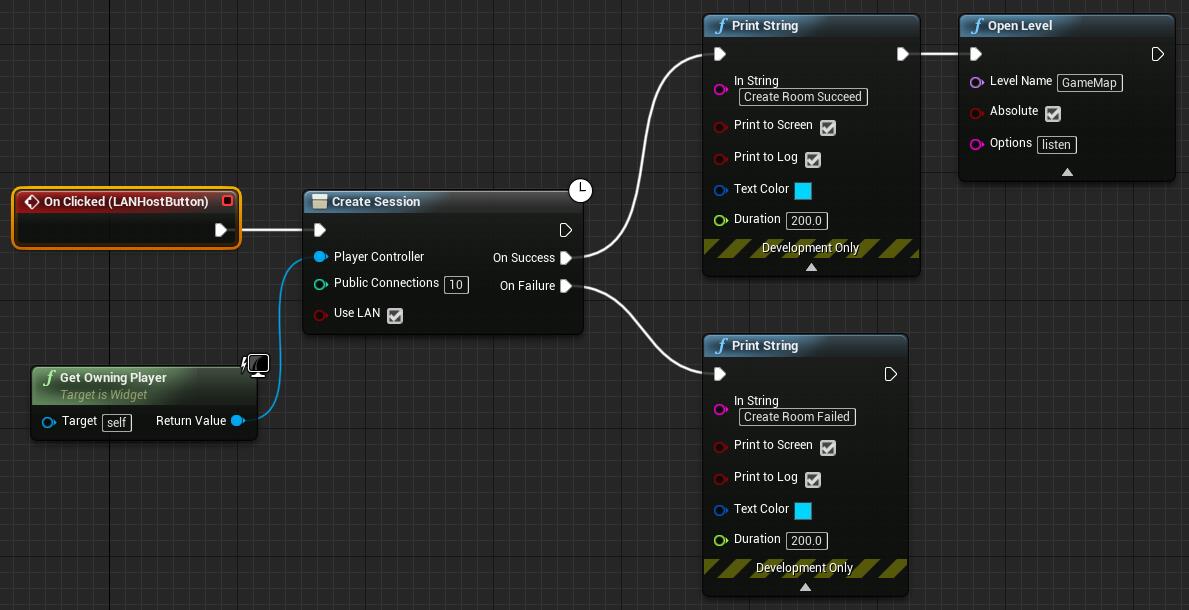
<1> NM\_Standalone --> 打包出exe后, 不创建快捷方式直接运行exe, 执行GetNetMode()返回NM\_Standalone类型, 在这个状态下打印GetWorld()->IsServer()以及HasAuthority()都是true, 如果在存在监听服务器的情况下, 运行命令行 open 127.0.0.1 , 该单独端就会链接上监听服务器, 执行GetNetMode()返回NM\_Client类型, 打印GetWorld()->IsServer()以及HasAuthority()都是false, 如果再次运行命令行 open 127.0.0.1 , 该端会先断开服务器, 然后再链接一次

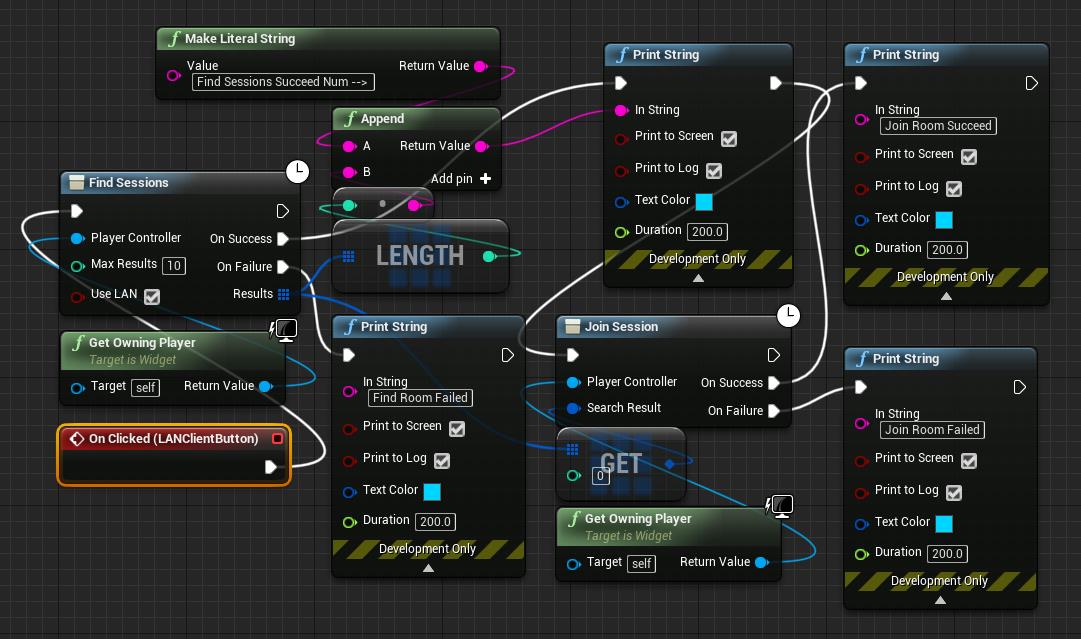
<2> NM\_ListenServer --> 打包出来的exe生成快捷方式, 并且在快捷方式的属性下在exe结尾添加 ?listen , 如 : RPCProject.exe ?listen, 运行该快捷方式就会运行监听服务器端, 执行GetNetMode()返回NM\_ListenServer类型, 打印GetWorld()->IsServer()以及HasAuthority()都是true, 如果运行命令行 open 127.0.0.1, 该监听服务器端就会变成单独端NM\_Standalone

<3> NM\_Client --> 打包出来的exe生成快捷方式, 并且在快捷方式的属性下在exe结尾添加 127.0.0.1 -game , 如 : RPCProject.exe 127.0.0.1 -game, 运行该快捷方式, 如果存在监听端, 就会链接上监听服务器, 成为客户端, 执行GetNetMode()返回NM\_Client类型, 打印GetWorld()->IsServer()以及HasAuthority()都是false

(3) 创建寻找加入会话蓝图模式(这里只实现局域网)

创建UMG添加两个按钮, 一个叫LANHostButton, 给服务端创建会话并且加入场景, 一个叫LANClientButton, 给客户端寻找以及加入会话, 分别添加按键事件





<1> 服务端点击LANHostButton后, 会执行CreateSession方法, 创建一个会话, 填入的PlayerController是本地的PlayerController, Public Connections必须不为0, UseLAN为true, 这样子填写客户端使用FindSessions时才能够成功寻找到Session并且Session不为0, 创建Session成功后要进行场景转换, 调用OpenLevel函数, Absolute设置为true, 必须在Options填入listen才能开启监听服务器, 不添加的话客户端就没办法进入服务器

<2> 服务端点击LANClientButton后, 会先执行FindSessions方法, 来寻找存在的会话, 填入的PlayerController是本地的PlayerController, 填入的MaxResults必须不为0, UseLAN为true, 这样子填写客户端使用FindSessions时才能够成功寻找到Session并且Session不为0, 寻找Session成功并且数量不为0后, 执行JoinSession函数来加入会话, 这里直接使用找到的会话数组里的第一个结果来链接Session, 填入的PlayerController是本地的PlayerController, 如果加入会话成功, 客户端会直接跳转到服务端此时已经进入的地图

(4) 创建寻找加入会话C++模式(这里只实现局域网)

C++联网步骤和蓝图基本相同, 但是中间多了一个StartSession方法需要调用, 主要可以参考

UCreateSessionCallbackProxy

UStartSessionCallbackProxy

UFindSessionsCallbackProxy

UJoinSessionCallbackProxy

UDestroySessionCallbackProxy

这些类的实现, 具体实现参考项目里的 URPCInstance类

UE4官方推荐将联网的逻辑放在GameInstance下处理, GameInstance在整个游戏所有关卡中都存在, 用来传递关卡数据, 保存数据方面起作用, 联网数据放在GameInstance下方便在任何关卡去操作联网