虚幻4网络整理

虚幻网络

虚幻引擎4使用标准的Listen-Server体系结构。这意味着，服务器是权威的，所有数据必须先从客户端发送到服务器。然后服务器验证数据并根据您的代码作出反应。

一个小例子：

当您移动您的角色，作为客户端，在多人游戏中，您实际上并没有移动您的角色自己，但告诉服务器你想要移动它。

然后，服务器会为所有其他人（包括您）更新角色的位置。

注意：为了防止本地客户端的“滞后”感觉，另外还要让这个玩家直接在本地控制他们的角色，尽管服务器仍然可以覆盖角色的位置当客户端开始作弊！

这意味着，客户将（几乎）不会直接与其他客户端有交互。

另一个例子：

当向另一个客户端发送聊天消息时，您实际上将其发送到服务器，然后服务器将其传递给您想要访问的客户端。这也可能是一个队伍，公会，组等。

重要

决不信任客户端！ 相信客户端意味着你就不检测客户端执行的相关操作。 这就允许作弊！

举个简单的射击游戏例子

确保在服务器上检测，如果客户端实际上有弹药才允许再次射击，而不是直接射击！

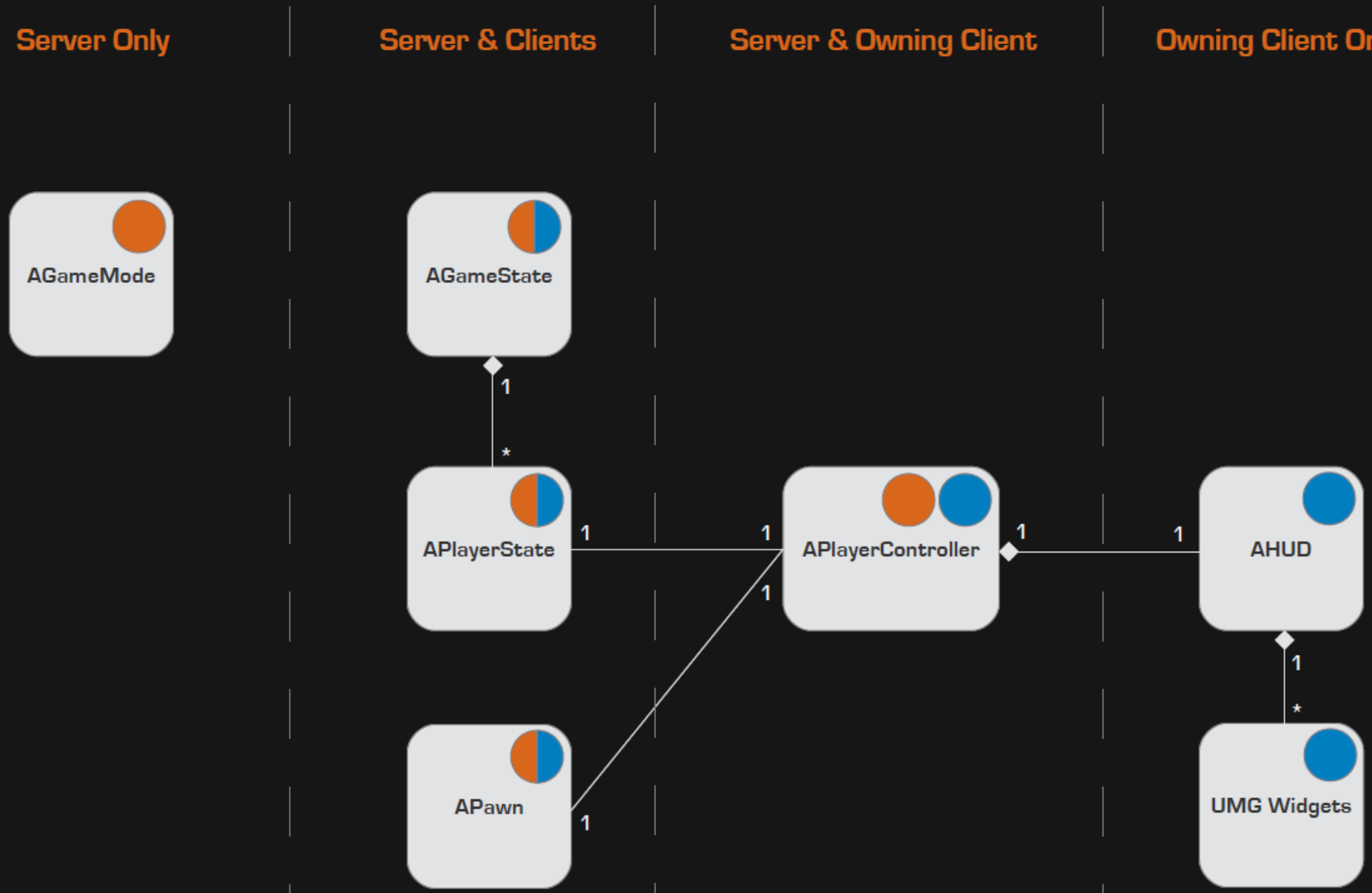
框架与网络

我们可以将U4的服务器-客户端架构分为4个部分：

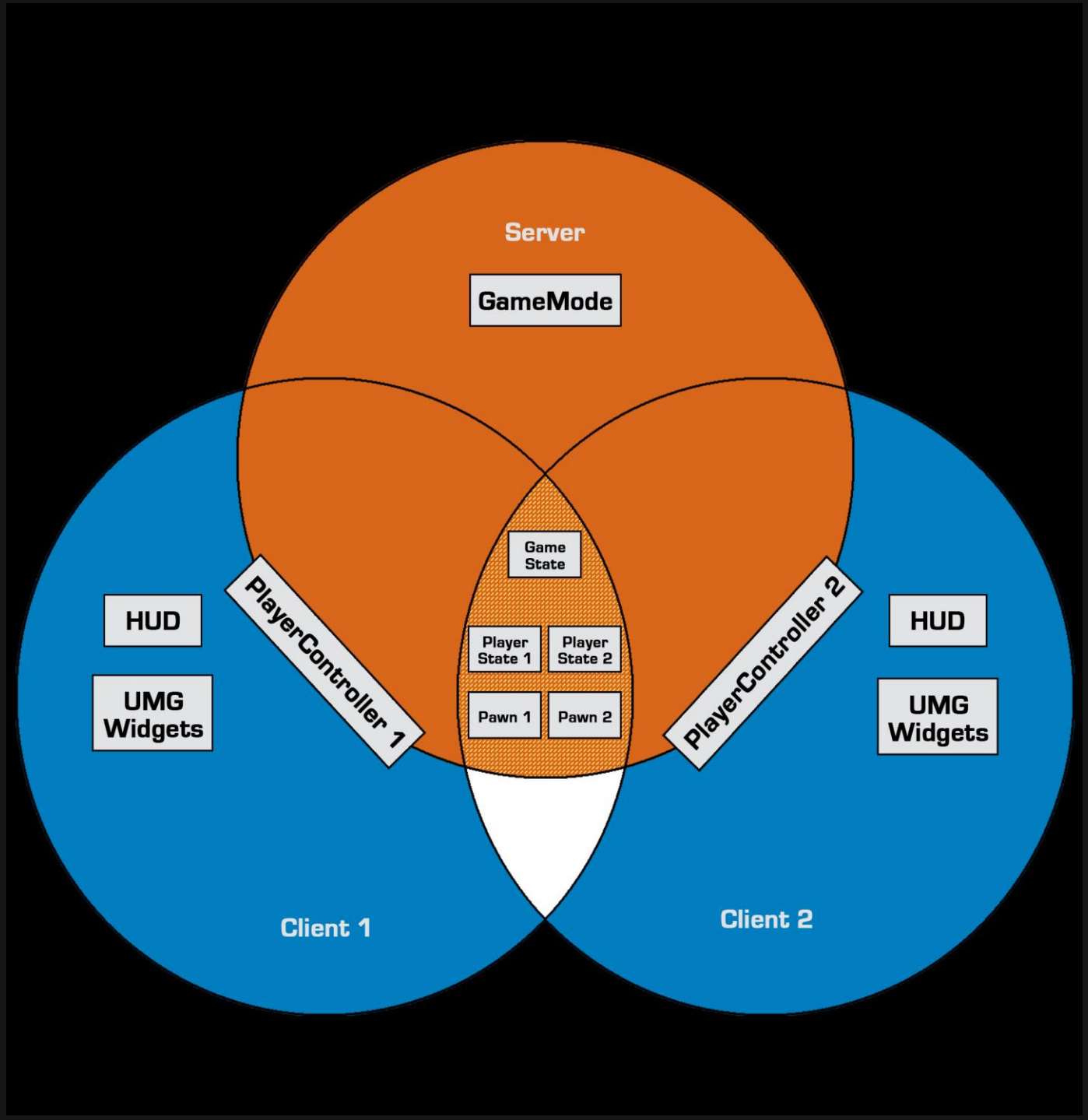
- Server Only - 这些对象仅存在于服务器上
- Server & Clients - 这些对象存在于服务器和所有客户端上
- Server & Owning Client - 这些对象仅存在于服务器和自身客户端上
- Owning Client Only - 这些对象仅存在于自己客户端

“Owning Client”是玩家/客户端，它拥有自身Actor的。
就像你拥有你自己的电脑一样。
所有权对于“RPC”来说变得很重要。

下表显示了一些常见的类
它们存在与哪一部分。



这是在网络框架中最重要的一些类的布局



左边的图片显示
（服务器与两个客户端）
这些类的对象将是
通过网络分发框架：

客户端1和客户端2之间的交点里没有对象
因为他们实际上并不共享客户端之间的信息

Game Mode

注意：使用4.14，GameMode类分为GameModeBase和GameMode。
GameModeBase具有较少的功能，因为一些游戏可能不需要旧的GameMode类的完整功能列表。
类AGameMode用于定义游戏的规则。 这包括
使用的类，如APawn，APlayerController，APlayerState等。
它仅在服务器上可用。 客户端没有GameMode的对象
并且在尝试检索它时只会得到一个nullptr。
举例：
GameMode可能通过常用模式了解
作为死亡匹配，团队死亡或抢夺旗帜。
这意味着，GameMode可以定义如下内容：

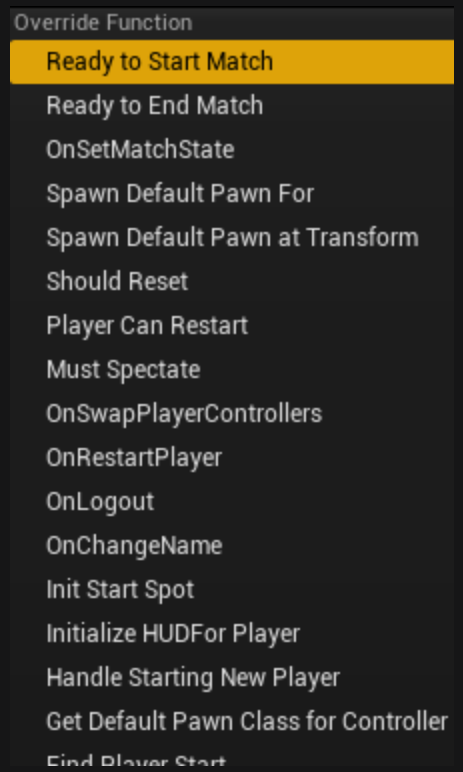
- 我们有团队，还是每个人都有自己的分数？
- 获奖条件是什么？ 某人/一个团队需要多少次杀死？
- 如何获得积分？ 杀人？ 偷旗
- 你的角色能使用什么？ 允许什么武器？ 只有手枪？

示例和用法

在多人游戏中，GameMode还有一些有趣的功能，
可以帮助我们管理玩家或一般的匹配流程

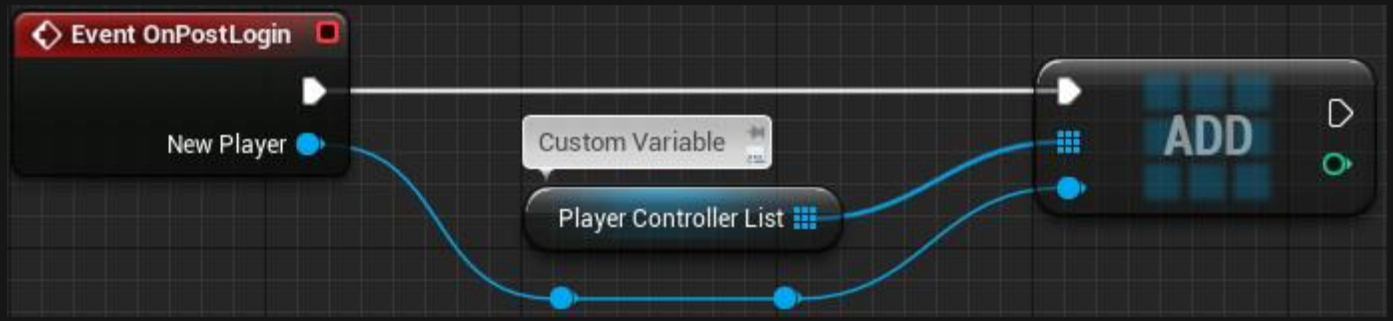
蓝图

在Blueprint版本里我们可以“Override Function”部分：
您可以为这些功能实现自己的逻辑，以适应特定的规则



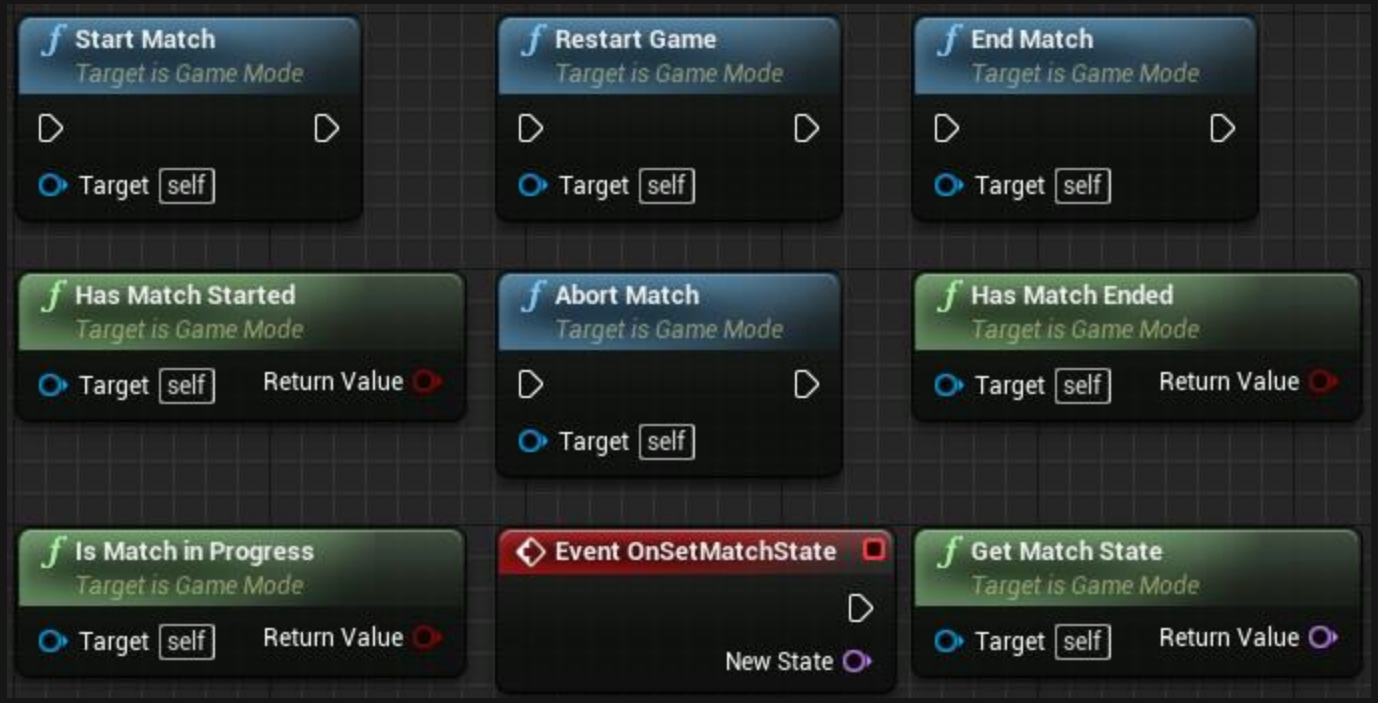
你的游戏 这包括改变GameMode产生的方式
DefaultPawn或您想要决定的游戏是否准备开始：
例如，检查所有玩家是否已加入并准备好：

但也有事件可以用来对某些事情做出反应，
这是通过匹配发生的。
我经常使用的一个很好的例子是“Event OnPostLogin”。
每当一个新玩家加入游戏时，都会被触发。
稍后您将了解有关连接过程的更多信息，但现在我们将继续进行。
该事件会传递一个有效的PlayerController。
这可以用来与这个玩家交互，为他产生一个新的Pawn
或者将他的PlayerController保存在数组中以供以后使用。
但是PC中的角色Pawn是为空的，还没有被创建



如前所述，您可以使用GameMode来管理一般游戏流程
你可以覆盖相关的函数来做你想要做的事（例如“Ready to start Match”）。

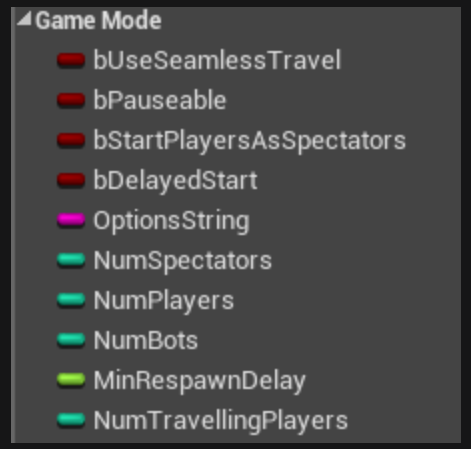
这些函数和事件可用于控制当前的游戏状态。 当“Ready to ..”
功能返回TRUE时，大部分将被自动调用，但您也可以手动调用它们。

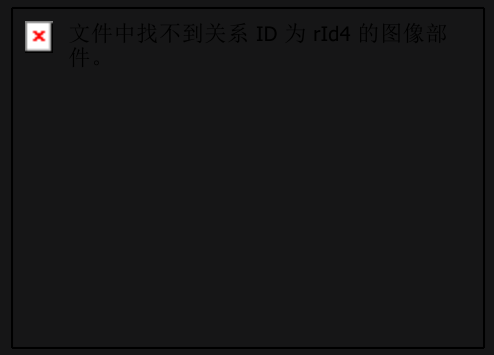


'New State'是一个'FName'类型。 你现在可以问：“为什么不在GameState Class中处理？”
GameMode功能实际上与GameState一起工作。 而管理这些并非是客户端的事，因为
GameMode只存在于服务器上！

当然GameMode也有重要的变量

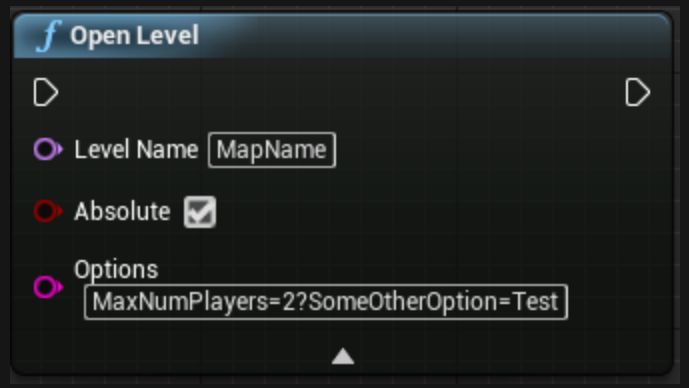
这是已经继承的变量的列表。

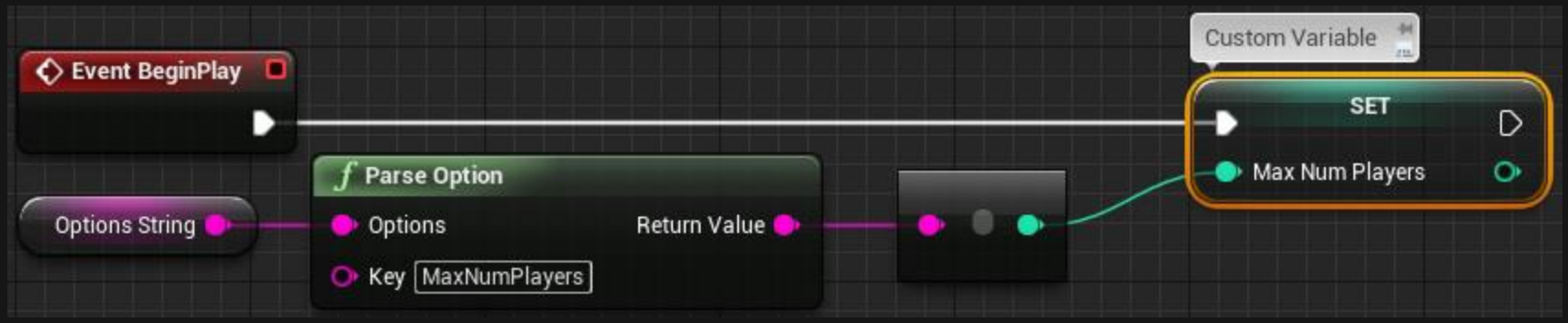




其中一些可以通过设置
GameMode蓝图的ClassDefaults：
他们中的大多数都是字面意思，比如“Use Seamless Travel”，
如果勾选了就会打开无缝切换流程。
或者“bDelayedStart”，如果bDelayed Start已设置，
请等待手动匹配开始

这里有一个更重要的变量“OptionsString”。其中“？”分开，
您可以通过“OpenLevel”函数或使用控制台命令“ServerTravel”来传递。
您可以使用“Parse Option”来提取传递的内容，例如“MaxNumPlayers”：





UE4 C++

之前蓝图相关的代码示例

由于“ReadyToStartMatch”是“BlueprintNativeEvent”，实际的C++实现函数称为“ReadyToStartMatch_Implementation”。然后我们要覆盖它：

```
/* Header file of our GameMode Child Class inside of the Class declaration */
// Maximum Number of Players needed/allowed during this Match
int32 MaxNumPlayers;

// Override Implementation of ReadyToStartMatch
virtual bool ReadyToStartMatch_Implementation() override;
```

```
/* CPP file of our GameMode Child Class */
bool ATestGameMode::ReadyToStartMatch_Implementation() {
    Super::ReadyToStartMatch();

    return MaxNumPlayers == NumPlayers;
}
```

UE4 C++

'OnPostLogin'是虚函数，在C++中是'PostLogin'。

```
/* Header file of our GameMode Child Class inside of the Class declaration */
// List of PlayerControllers
TArray<class APlayerController*> PlayerControllerList;

// Overriding the PostLogin function
virtual void PostLogin(APlayerController* NewPlayer) override;
```

```
/* CPP file of our GameMode Child Class */
void ATestGameMode::PostLogin(APlayerController* NewPlayer) {
    Super::PostLogin(NewPlayer);

    PlayerControllerList.Add(NewPlayer);
}
```

UE4 C++

当然，所有Match-handling函数都可以被覆盖和更改，
所以我不会在这里列出。详细的请查阅下面链接
https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/GameFramework/AGameMode/index.html
GameMode的最后一个C++示例将是“Options String”：

```
/* Header file of our GameMode Child Class inside of the Class declaration */
// Maximum Number of Players needed/allowed during this Match
int32 MaxNumPlayers;

// Override BeginPlay, since we need that to recreate the BP version
virtual void BeginPlay() override;
```

```
/* CPP file of our GameMode Child Class */
void ATestGameMode::BeginPlay() {
    Super::BeginPlay();

    // 'FCString::Atoi' converts 'FString' to 'int32' and we use the static 'ParseOption' function of the
    // 'UGameplayStatics' Class to get the correct Key from the 'OptionsString'
    MaxNumPlayers = FCString::Atoi( *(UGameplayStatics::ParseOption(OptionsString, “MaxNumPlayers”)) );
}
```

Game State

注意：使用4.14，GameState类分成了GameStateBase和GameState。
GameStateBase具有较少的功能，
因为一些游戏可能不需要旧的GameState类的完整功能列表。
在服务器和客户端之间AGameState类可能是最重要的共享信息。
GameState用于跟踪当前游戏状态。
这包括多人游戏里重要的玩家列表（APlayerState）。
GameState被复制到所有客户端。 所以每个人都可以访问它。 这使得
GameState成为多人游戏中最为核心的类之一。
GameMode可以告诉GameState需要多少次杀戮才能获胜
跟踪每个玩家和团队目前击杀数量！
您在这里存储的信息完全取决于您。
它可以是一组数组或一组自定义结构，用于跟踪相关数据。

示例和用法

在多人游戏中，GameState类用于跟踪游戏的当前状态，
其中还包括玩家及其玩家状态。 GameMode确保了
GameState的Match State功能被调用，
GameState为您提供了在客户端上使用它的机会。
GameState允许我们创建我们自己的逻辑，
主要是将信息传递给客户端。

蓝图

我们可以从父类GameState中获取一些我们可以使用的变量。
PlayerArray *， 以及MatchState和ElapsedTime被replicated，
所以客户端也可以访问它们。
这不等于AuthorityGameMode。 只有服务器可以访问它
因为GameMode只存在于服务器上。
PlayerArray不会直接replicated，但是每个PlayerState都是在他们构造函数
时自己添加到PlayerArray中。 GameState也会在创建时收集一次。

PlayerArray在C++中描述，它被“复制”。
PlayerState类中：

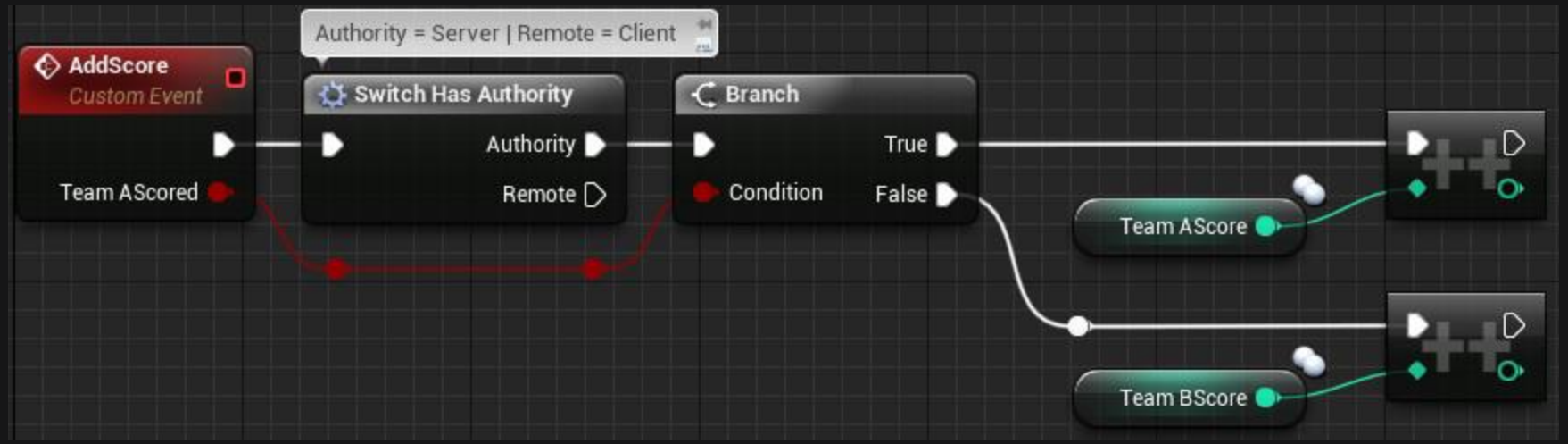
```
void APlayerState::PostInitializeComponents() {
    […]
    UWorld* World = GetWorld();
    // Register this PlayerState with the Game's ReplicationInfo
    if(World->GameState != NULL) {
        World->GameState->AddPlayerState(this);
    }
    […]
}
```

在GameState中：

```
void AGameState::PostInitializeComponents() {
    […]
    for(TActorIterator<APlayerState> It(World); It; ++It) {
        AddPlayerState(*It);
    }
}
void AGameState::AddPlayerState(APlayerState* PlayerState) {
    if(!PlayerState->bIsInactive) {
        PlayerArray.AddUnique(PlayerState);
    }
}
```

所有这些都发生在Server和GameState的客户端实例上！

我可以提供给你的一个小例子是跟踪两队“A”和“B”的得分。
假设我们有一个自定义事件，当团队得分时，它被调用。
它传递一个布尔值，所以我们知道哪个Team得分。 稍后在“Replication”部分，只有服务器可以复制变量，所以我们确保只有他可以调用此事件。
它来自另一个类（例如一个杀死某人的武器）
并且这应该总是发生在服务器上，所以我们不需要RPC。



由于这些变量和GameState被replicated，您可以使用这两个变量，并将它们放在您想要的任何其他类中，例如，将它们显示在Scoreboard Widget。

UE4 C++

为了重现这个小例子，我们需要更多的代码，但是尽管功能本身，
只需要在Class中设置复制所需的代码。

```
/* Header file of our GameState Class inside of the Class declaration */
// You need this include to get the Replication working. Good place for it would be your Projects Header!
#include "UnrealNetwork.h"

// Replicated Specifier used to mark this variable to replicate
UPROPERTY(Replicated)
    int32 TeamAScore;
UPROPERTY(Replicated)
    int32 TeamBScore;

// Function to increase the Score of a Team
void AddScore(bool TeamAScored);
```

UE4 C++

我们将在Replication部分中阅读有关此函数的更多信息!

```cpp
/* CPP file of our GameState Class */
// This function is required through the Replicated specifier in the UPROPERTY Macro and is declared by it
void ATestGameState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {

    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME(ATestGameState, TeamAScore);
    DOREPLIFETIME(ATestGameState, TeamBScore);
}
```

```cpp
/* CPP file of our GameState Class */
void ATestGameState::AddScore(bool TeamAScored) {

    if(TeamAScored)
          TeamAScore++;
    else
          TeamBScore++;
}
```

Player State

对于玩家来说APlayerState是非常重要的类。
玩家相关的最新数据保存在这个类里。 每个玩家都有自己的PlayerState。 PlayerState也被复制(replicated)给所有人，可使用检索和显示其他客户端上的数据。 我们可以在GameState类中使用PlayerArray来访问当前游戏中所有的PlayerState。

您可能想要存储的一些数据：

- 玩家名字 - 当前连线的玩家名称
- 得分 - 连接的玩家的当前得分
- Ping - 连线玩家的当前Ping
- 公会ID - 公会的ID，玩家可能在这个工会里
- 或其他玩家可能需要了解的其他复制信息

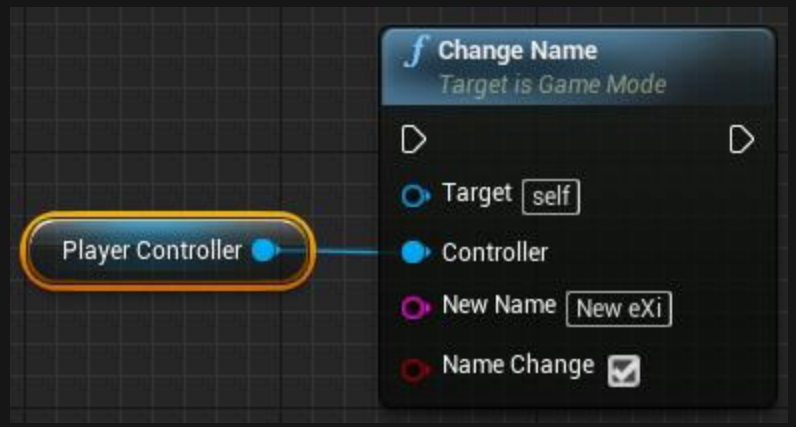
示例和用法

在多人游戏中，PlayerState是保存有关在线玩家的状态信息。
这包括Name，Score，Ping和您的自定义变量。
由于该PlayerState类被复制(replicated)，所以它可以轻松地用于检索客户端在其他客户端上的数据。

蓝图

可悲的是，当我写这篇文章时，主要的重要功能不会暴露在Blueprint中。
所以我只会在PlayerState示例的C++部分中覆盖它们。 但我们可以看看
在一些变量：

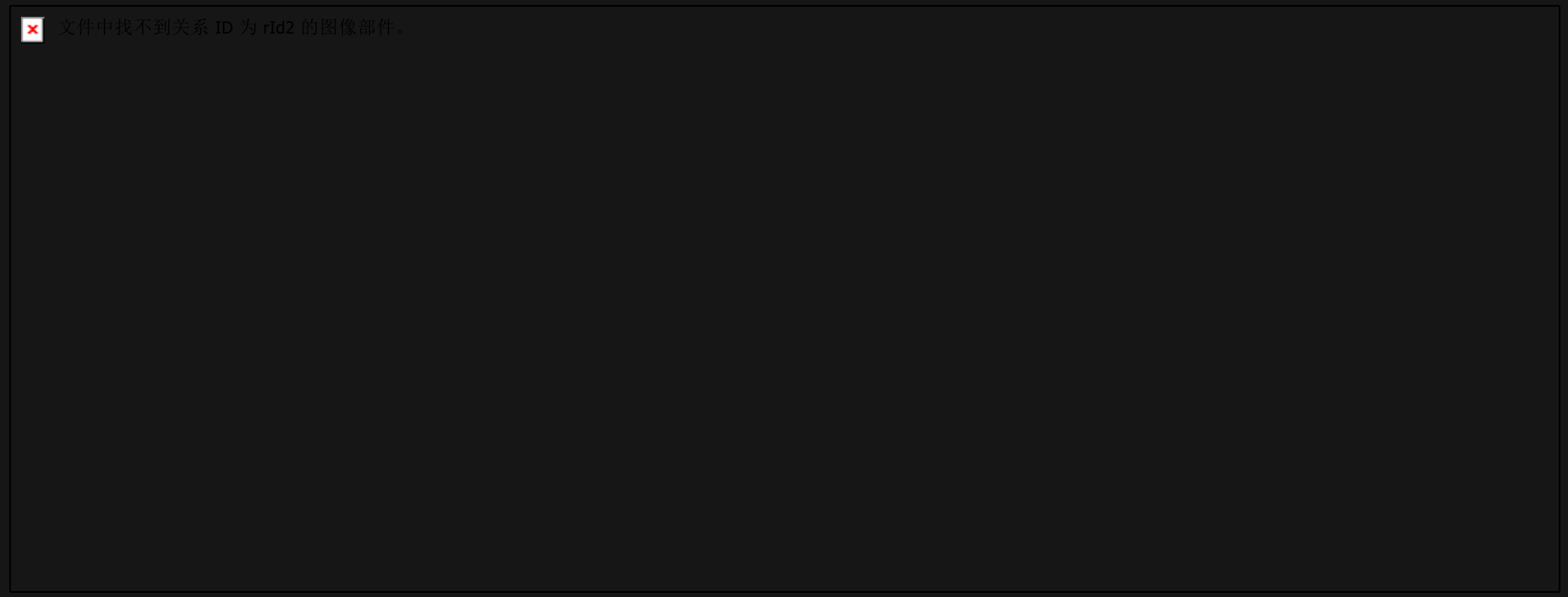
这些变量都被复制(replicated)，因此它们在所有客户端上保持同步。
可悲的是，他们不容易在Blueprints中设定，但是你可以创建属于你的版本。



关于PlayerName的一个例子
变量可以通过调用“ChangeName”来设置，
一个GameMode函数，
并将玩家的PlayerController传递给它。

PlayerState也用于确保在场景切换
或意外连接问题期间数据能保持持续。
PlayerState有两个专用于处理玩家重新连接的功能
以及与服务器一起切换玩家到一张新的地图。

PlayerState负责将已经拥有的信息复制到新的PlayerState中。
这可以通过“场景切换”创建，或者是因为玩家正在重新连接。



UE4 C++

当然也存在于C ++中。 这里是

```
/* Header file of our PlayerState Child Class inside of the Class declaration */

// Used to copy properties from the current PlayerState to the passed one
virtual void CopyProperties(class APlayerState* PlayerState);


// Used to override the current PlayerState with the properties of the passed one

virtual void OverrideWith(class APlayerState* PlayerState);
```

这些功能可以实现您自己的C ++ PlayerState子类中进行管理
您添加自定义PlayerState的数据。 确保在末尾添加“override”说明符，并调用
“Super ::”使得父类的方法执行（如果要的话）。

实现可能类似于：

文件中找不到关系 ID 为 rId2 的图像部件。

Pawn

APawn类是Player实际控制的“Actor”。 大部分的时间是人型角色，但它也可以是一只猫，飞机，船等。
玩家一次只能拥有一个Pawn，
但在Pawn之间你可以轻松来回切换。

Pawn主要复制给所有客户端

Pawn的子类ACharacter经常被使用，因为它已经有了MovementComponent，其处理复制位置，旋转等与玩家角色。
注意：不是客户端正在移动角色。 服务器正在从客户端获取移动输入，然后移动并复制(replicating)角色！

示例和用法

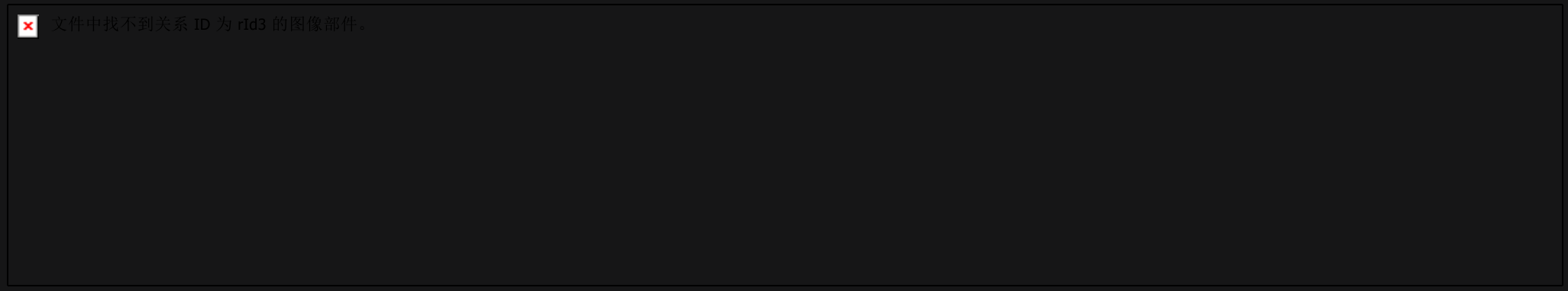
在多人游戏中，我们主要使用Pawn的复制部分来显示Character和
与他人分享一些信息。 我们举个角色身上的“Health”例子。
但是我们不仅复制“Health”使其可见，而且我们也复制它，
以便服务器拥有权限，客户端无法作弊。

蓝图

尽管有“standard”可覆盖的函数，Pawn有两个
对拥有和不拥有的事件。 当他没有拥有情况下我们可以
用来隐藏Pawn：

注意：由于在服务器上发生了这些事件，
所以只有当Pawn并且需要MulticastRPCFunction来更改可见性时，
才会在服务器版本上调用这些事件，就像上面的截图！

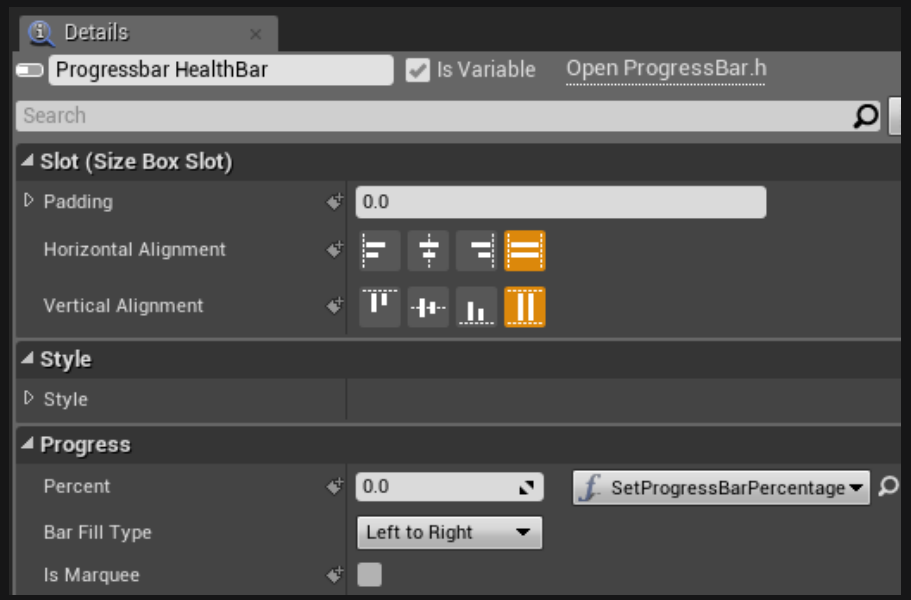
这里我们以“Health”为例子。 下图
将显示如何使用“EventAnyDamage”和复制的“Health”变量
降低玩家的生命。 这发生在服务器而不是客户端上!

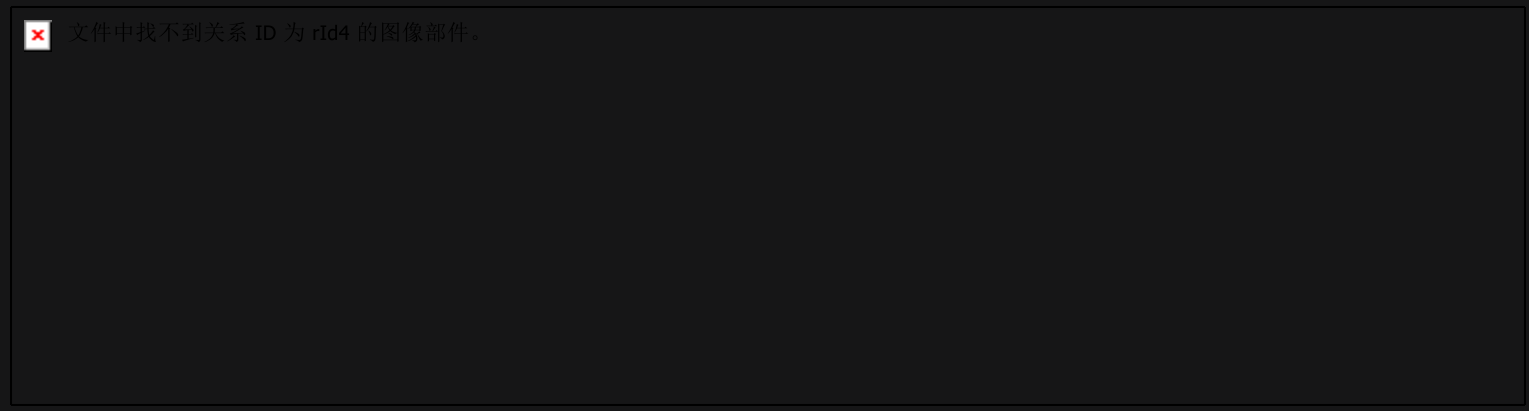


如果在服务器调用，那么Pawn应该是复制的，
“DestroyActor”节点也将销毁Actor的客户端版本。
在客户端节点上，我们可以使用复制的“Health”变量
来绘制每个人头上的血条。

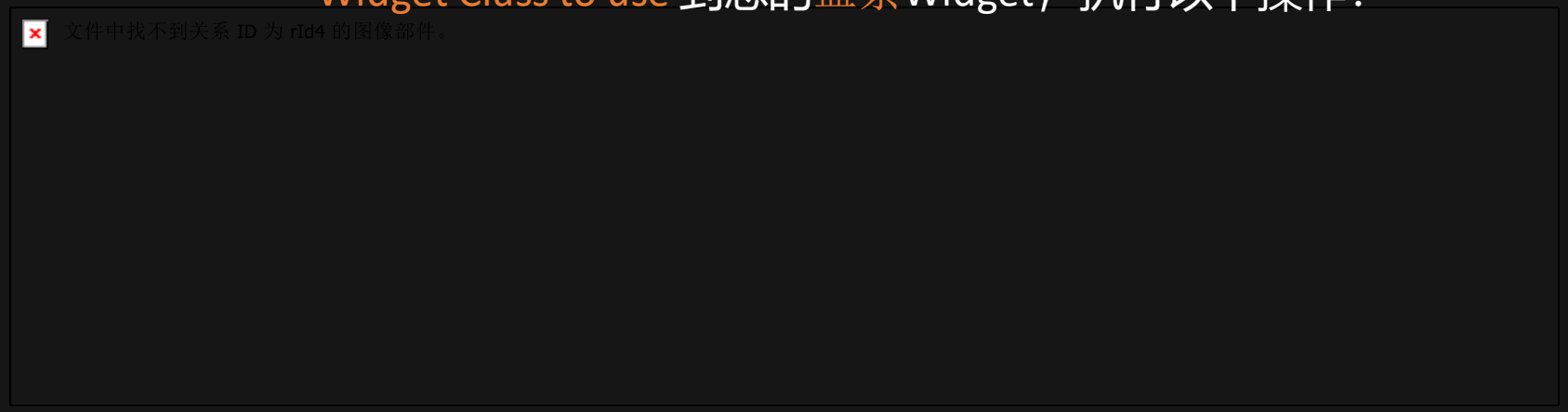
你可以简单的创建一个Widget 里面有Progressbar和“Pawn”的引用。

假设我们在“TestPawn”类上有一个“Health”和“MaxHealth”变量，全部设置为复制。
现在在Widget中创建“TestPawn”引用变量之后，以及
ProgressBar，我们可以将该栏的百分比绑定到以下函数：

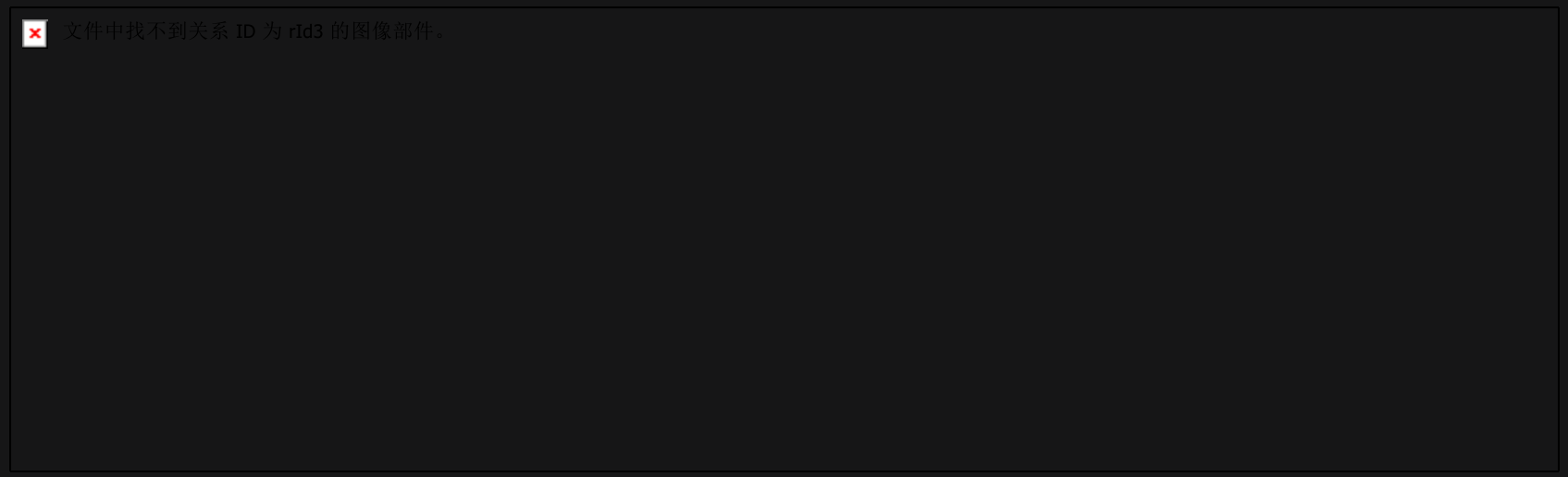




现在设置WidgetComponent（这个Class是实验性的）之后，我们可以在BeginPlay设置
'Widget Class to use'到您的血条Widget，执行以下操作：



在服务器和所有客户端Pawn实例上调用“BeginPlay”。
所以每个实例都可以设置自己拥有的Widget和Pawn Reference。
而且由于“Pawn”和“Health”变量是复制的，所以我们可以正确无误的显示在Pawn的头上。



如果复制过程在这一点上还不清楚，只要继续阅读，
在后面的内容会有所解释。

UE4 C++

我们在这里将专注于“Possess Events”和“Damage Event”。
在C ++中，两个Possess事件被称为：

文件中找不到关系 ID 为 rId2 的图像部件。

'UnPossessed'事件不通过旧的PlayerController。

当前无法显示此图像。

UE4 C++

我们还需要一个MulticastRPCFunction。 稍后您会在RPC章节中学习到
:

```
/* Header file of our Pawn Child Class, inside of the Class declaration */
UFUNCTION(NetMulticast, unreliable)
    void Multicast_HideMesh();
```

```
/* CPP file of our Pawn Child Class */
void ATestPawn::UnPossessed() {
    Super::UnPossessed();

    Multicast_HideMesh();
}

// You will read later about RPC's and why that '_Implementation' is a thing
void ATestPawn::Multicast_HideMesh_Implementation() {
    SkeletalMesh->SetVisibility(false);
}
```

UE4 C++

而且我们也想在C ++中重新创建血值示例。 一如既往，如果你不明白此时复制的步骤，别担心，当前章节将向您解释。
如果它们在复制方面似乎很复杂，那么现在就跳过这些例子。
“TakeDamage”函数相当于“EventAnyDamage”节点。 造成损害，你通常会在你要损害的Actor上调用“TakeDamage”，如果该Actor实现这个功能，它会对它做出反应，类似于这个例子。

```
/* Header file of our Pawn Child Class, inside of the Class declaration */

// Replicated Health Variable
UPROPERTY(Replicated)
    int32 Health;


// Overriding the Damage Event
virtual float TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,
                         AController* EventInstigator, AActor* DamageCauser) override;
```

UE4 C++

文件中找不到关系 ID 为 rId2 的图像部件。

Player Controller

APlayerController类可能是我们遇到的最有趣和最复杂的类。
它并且是许多客户端的核心，
因为这是客户端实际拥有的第一个类。
PlayerController可以看作是玩家的“输入”。它是玩家与服务器的链接。这意味着，每个客户端都有一个PlayerController。
客户端的PlayerController只存在于他自己和服务器上，而其他的客户端不了解其他PlayerController。

每个客户只知道自己的！

其结果是，该服务器具有所有Client PlayerController！
所有实际的输入（按钮按下，鼠标移动，控制器轴等）需要放在PlayerController中。
设置角色特定输入是一个很好的做法（汽车的控制方式与人类不同）
进入你的角色/Pawn类，并输入应该适用于所有角色的输入，
直到PlayerController对应的角色对象无效时！

这里有个重点：

首先输入始终通过PlayerController传递。 如果PlayerController
不使用它，它将在可能使用相同的其他类中处理
输入。 当然，输入可以随时停用。

此外，一件重要的事情是：
如何获取正确的PlayerController？

蓝图里获取节点“GetPlayerController（0）”或代码行
“UGameplayStatics :: GetPlayerController（GetWorld（），0）;”
在服务器和客户端上它的工作方式有所不同。

- 在Listen-Server上调用它将返回Listen-Server的PlayerController
- 在客户端调用它将返回客户端的PlayerController
- 在专用服务器上调用它将返回第一个客户端的PlayerController

其他数字超过'0'将不会返回其他客户端。 这个索引是
意在用于本地玩家（分屏情况下），我们将不会覆盖。

示例和用法

尽管PlayerController是网络里最重要的类之一，
所以我们将创建一个小例子，只是为了弄清楚为什么需要它。
在关于所有权的章节中，您将阅读到为什么
PlayerController对RPC很重要。以下示例将显示您
如何使用PlayerController来增加一个复制的变量
在GameState中按一个Widget按钮。

为什么我们需要PlayerController？

那么，我不想再次写下RPC和所有权章节，所以只是一个简短的解释：
Widgets只存在于客户端/侦听服务器上，它们是由客户端拥有的，
服务器RPC没有运行的实例。

它根本没有复制！

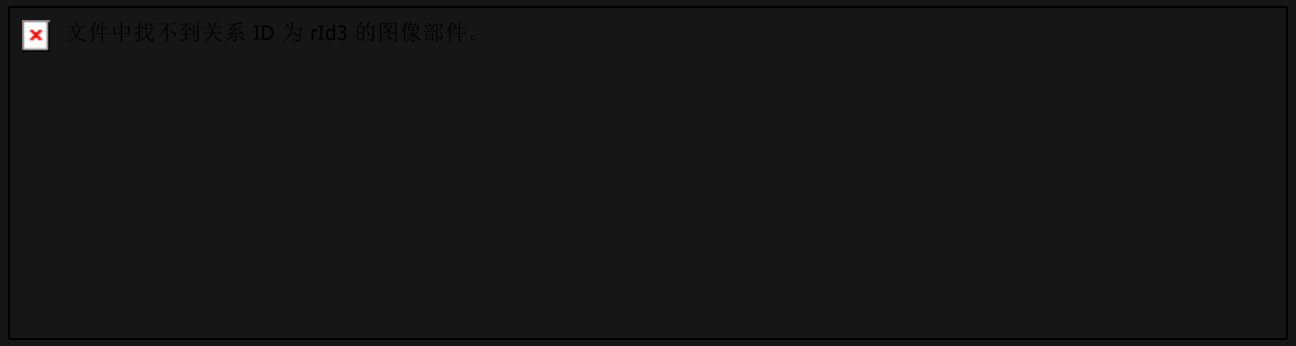
这意味着我们需要一种方法让Button触发到服务器，让其变量增加。

为什么不直接在GameState上调用RPC？因为它由服务器拥有。
服务器RPC需要客户端作为拥有者！

蓝图

首先我们需要一个简单的Widget可以按一下Button。 这是一个网络纲要，
所以请自己学习Widget。
下面我以反向顺序发布蓝图，你可以看到它在哪里结束以及具体是什么事件
实际上调用了上一张图像的事件。 所以从我们的目标开始，GameState。
这是个很普通的变量，增加一个复制的整数变量：

此事件将在我们在PlayerController中的ServerRPC内的服务器端被调用：



最后，我们有一个Button，它被按下并调用ServerRPC：

所以当我们点击（客户端）按钮时，我们在PlayerController中使用ServerRPC
去服务器端（可能是因为PlayerController由客户端拥有！）
然后调用GameState的“IncreaseVariable”事件来增加复制的整数。
这个整数，由于它被服务器复制和设置，现在将更新所有的实例
GameState和客户端也可以看到更新！

UE4 C++

对于本示例的C ++版本，我将使用PlayerController的BeginPlay替换Widget。
这没有什么意义，但在C ++中实现Widgets需要
更多的代码，我不想在这里做，因为这要写的太多！

```
/* Header file of our PlayerController Child Class, inside of the Class declaration */
// Server RPC. You will read more about this in the RPC Chapter
UFUNCTION(Server, unreliable, WithValidation)
    void Server_IncreaseVariable();

// Also overriding the BeginPlay function for this example
virtual void BeginPlay() override;
```

```
/* Header file of our GameState Child Class, inside of the Class declaration */
// Replicated Integer Variable
UPROPERTY(Replicated)
    int32 OurVariable;

public:
    // Function to Increment the Variable
    void IncreaseVariable();
```

```cpp
/* CPP file of our PlayerController Child Class */
// Otherwise we can't access the GameState functions
#include "TestGameState.h"

// You will read later about RPC's and why that '_Validate' is a thing
bool ATestPlayerController::Server_IncreaseVariable_Validate() {
    return true;
}

// You will read later about RPC's and why that '_Implementation' is a thing
void ATestPlayerController::Server_IncreaseVariable_Implementation() {
    ATestGameState* GameState = Cast<ATestGameState>(UGameplayStatics::GetGameState(GetWorld()));

    GameState->IncreaseVariable();
}

void ATestPlayerController::BeginPlay() {
    Super::BeginPlay();

    // Make sure only the Client Version of this PlayerController calls the ServerRPC
    if(Role < ROLE_Authority)
          Server_IncreaseVariable();
}
```

UE4 C++

```cpp
/* CPP file of our GameState Child Class */
// This function is required through the Replicated specifier in the UPROPERTY Macro and is declared by it
void ATestGameState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    // This actually takes care of replicating the Variable
    DOREPLIFETIME(ATestGameState, OurVariable);
}

void ATestGameState::IncreaseVariable() {
    OurVariable++;
}
```

这是一些代码。 如果你不明白使用某些功能和命名，别担心。 即将到来的章节将帮助您了解为什么这样做。

HUD

AHUD类是仅在每个客户端上可用的类
可以通过PlayerController访问。 它将被自动产生。

在UMG发布之前，早已使用了HUD类
在客户端的视口中绘制文本，纹理等。

到目前为止，Widgets大部分时间都取代了HUD类。

您仍然可以使用HUD类进行调试，也可能有一个独立的区域来
管理创建，显示，隐藏和销毁的控件。

由于HUD没有直接链接到网络，示例只会显示单机的东西。
这就是为什么我会跳过它们。

Widgets

Widgets用于Epic Games的新UI系统，称为Unreal Motion Graphics。
它们继承自Slate，它是一种用于在C ++中创建UI的语言
用于虚幻引擎4编辑器本身。
Widgets只能在本地在客户端（Listen-Server）上可用。
它们不被复制，你总是需要一个分离的，复制的类
通过例如按钮按压来执行复制的动作。
要了解有关UMG和Widget的更多信息，
https://docs.unrealengine.com/latest/INT/API/Runtime/UMG/index.html。
我们已经在APawn示例中使用了Widget的一个小例子。
所以在这里我会跳过他们。

Dedicated vs Listen Server

Dedicated服务器

Dedicated服务器是一个不需要客户端的独立服务器。
它与游戏客户端分离，主要用于
有一个服务器运行，玩家可以随时加入/离开。
可以为Windows和Linux编译专用服务器，可以运行
在虚拟服务器上，玩家可以通过固定的IP地址连接到虚拟服务器。
Dedicated服务器没有可视化部分，所以他们不需要一个UI，
也没有一个PlayerController。
他们也没有在游戏中代表他们的角色或类似的东西。

在国内网络游戏基本都是走这一类，有专门的服务器程序开发这块！

Listen-Server

Listen-Server是一个客户端的服务器。
这意味着，服务器始终至少有一个客户端连接。
该客户端称为Listen-Server，如果他断开连接，服务器将关闭。
由于也是客户端，Listen-Server需要UI并具有PlayerController，
代表客户部分。 在侦听服务器上获取“PlayerController（0）”，
将返回该客户端的PlayerController。
由于Listen-Server在客户端本身运行，其他人需要连接的IP
是客户之一。 与专用服务器相比，这往往来了
互联网用户没有静态IP的问题。
但是使用OnlineSubsystem，可以解决IP问题的变化。

OnlineSubsystem它定义的接口提供了一套抽象并干净的通用网络接口，
可用于多个在线平台。这里 平台 的意思是指诸如 Steam，Xbox Live，
Facebook 等。这个系统的主要目的是为了较好的移植性。

Replication

什么是“Replication”？

复制是服务器将信息/数据传递给客户端的行为。
这可以仅限于特定的实体和组。
蓝图主要根据受影响的AActor的设置执行复制。

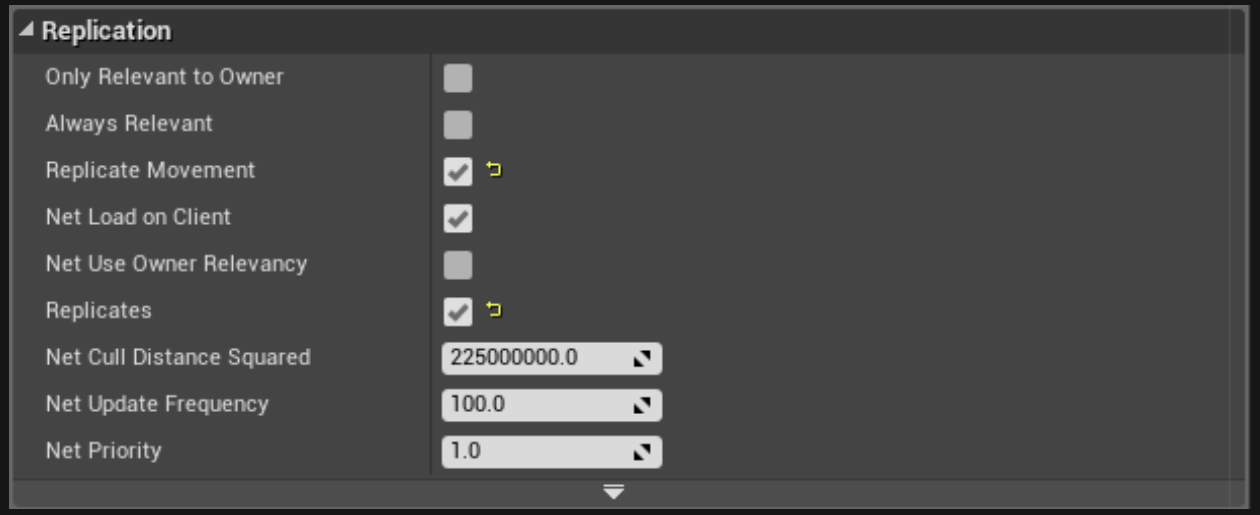
能够复制属性的第一个类是Actor。
所有之前提到的Classes从某个角度继承自Actor，
使他们能够根据需要复制属性。

虽然并不是全部都这样做。

例如，GameMode根本不会复制，只存在于服务器上。

怎么使用“Replication”？

复制可以在默认构造函数中激活



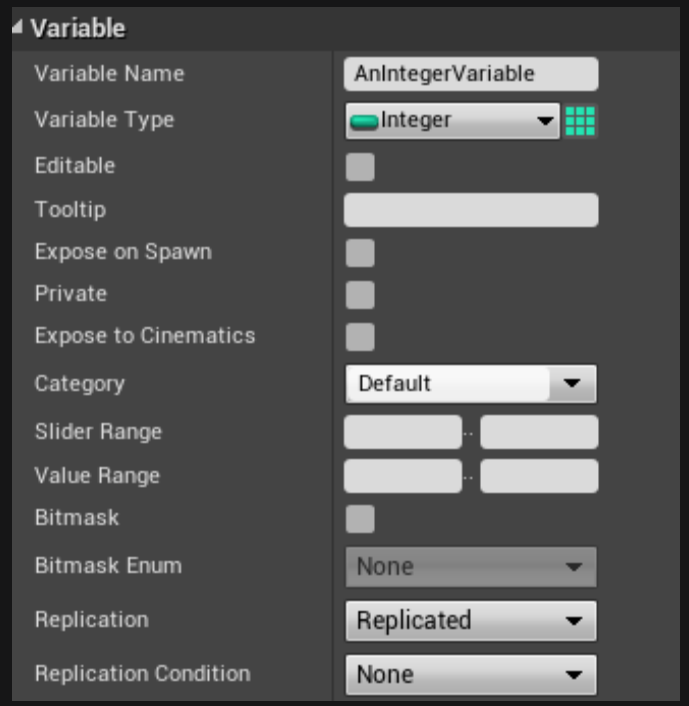

```
ATestCharacter::ATestCharacter() {
    // Network game, so let's setup Replication
    bReplicates = true;
    bReplicateMovement = true;
}
```

'bReplicates'勾选为TRUE的Actor, 只有当服务器创建该Actor时才会给所有客户端生成并且复制该Actor。

如果一个客户端创建了Actor，那么只有这个客户端就存在这个Actor。

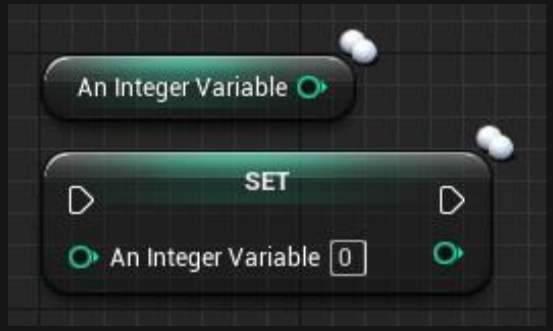
Replicating属性

当开启复制时，我们可以复制变量。 有多种方法可以做到这一点。 我们将从最基本的一个开始：



将“Replication”下拉菜单设置为“Replicated”将确保这一点变量被复制到该Actor的所有复制实例。 当然这样仅适用于设置为复制的Actor。

使用4.14，现在可以在某些条件下复制变量，即使在蓝图。 要了解有关条件的更多信息，请进一步滚动。



复制变量标有2个白色圆圈。

在C ++中复制变量在开始时稍微增加一些工作。
但是它也允许我们指定哪一个变量可以被复制。

```
/* Header file inside of the Classes declaration */
// Create replicated health variable
UPROPERTY(Replicated)
float Health;
```

.cpp文件将重写“GetLifetimeReplicatedProps”函数。
该功能有UE4宏提供cho，因此我们不需要关心那个。 在这里，您实际上定义了复制变量的规则。

```
void ATestPlayerCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    // Here we list the variables we want to replicate + a condition if wanted
    DOREPLIFETIME(ATestPlayerCharacter, Health);
}
```

您还可以在此处进行条件复制：

```
// Replicates the Variable only to the Owner of this Object/Class
DOREPLIFETIME_CONDITION(ATestPlayerCharacter, Health, COND_OwnerOnly);
```

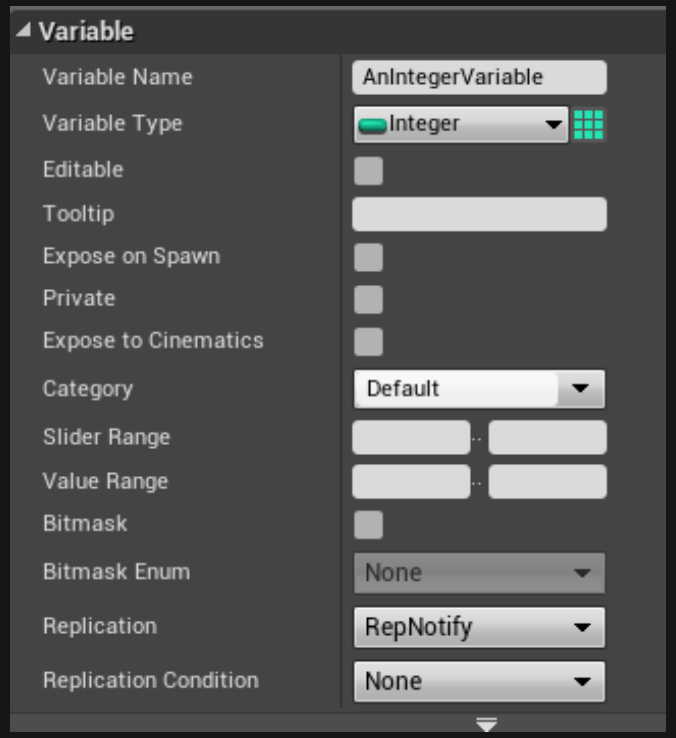
条件	说明
COND_InitialOnly	该属性仅在初始数据组尝试发送
COND_OwnerOnly	该属性仅发送至 actor 的所有者
COND_SkipOwner	该属性将发送至除所有者之外的每个连接
COND_SimulatedOnly	该属性仅发送至模拟 actor
COND_AutonomousOnly	该属性仅发送给自主 actor
COND_SimulatedOrPhysics	该属性将发送至模拟或 bRepPhysics actor
COND_InitialOrOwner	该属性将发送初始数据包，或者发送至 actor 所有者
COND_Custom	该属性没有特定条件，但需要通过 SetCustomIsActiveOverride 得到开启/关闭能力

重要的是，您了解整个复制过程仅适用于服务器到客户端，
而不是其他方式。

稍后我们将学习如何让服务器复制某些东西，
客户端要与他人共享（例如：PlayerName）。

在复制里有个不同版本是所谓的“RepNotify”版本。
在接收更新到值时将会在所有实例上调用的函数。

这样，您可以调用在复制值后需要调用的逻辑。



在蓝图中，在复制下拉菜单中选择
“RepNotify”后，将自动创建此功能：

C ++版本需要更多，但工作原理相同：

```cpp
/* Header file inside of the Classes declaration */
// Create RepNotify Health variable
UPROPERTY(ReplicatedUsing=OnRep_Health)
float Health;

// Create OnRep Function | UFUNCTION() Macro is important! | Doesn't need to be virtual though
UFUNCTION()
virtual void OnRep_Health();
```

```cpp
/* CPP file of the Class */
void ATestCharacter::OnRep_Health() {
    if(Health < 0.0f)
        PlayDeathAnimation();
}
```

使用'ReplicatedUsing = FUNCTIONAME'，我们指定当变量成功复制时应该调用什么函数。
这个函数需要有'UNFUNCTION（）'宏，即使它是空的！

Remote Procedure Calls

复制的其他方法被称为“RPC”。
“Remote Procedure Call”(远程过程调用)的简写。

他们习惯于在另一个实例上调用某些东西。 就和电视遥控器与电视机相同。

UE4使用它来调用从客户端到服务器，服务器到客户机或服务器到特定组的功能。

这些RPC不能有返回值！ 要返回的东西，你需要使用第二个RPC。

但这只能在某些规则下工作。
它们列在本表中，可以在官方文档中找到：

- Run on Server - 是要在Actor的服务器实例上执行
- Run on owning Client - 是要在Actor的客户端所有者
- NetMulticast - 是要在该Actor的所有实例上执行

要求和注意事项

1.他们必须来自Actor
2.必须复制Actor
3.如果RPC从服务器被调用以在客户端上执行，
只有实际拥有Actor的客户端才能执行该功能
4.如果从客户端调用RPC在服务器上执行，
客户端必须拥有要调用RPC的Actor
5.多播 RPC是一个例外：
◦如果从服务器调用它们，服务器将在本地执行，
并在所有当前连接的客户端上执行它们
◦如果它们是从客户端调用的，它们只能在本地执行，
并且不会在服务器上执行
◦现在，我们有一个简单的多播事件调节机制：
▪给定的多播功能不会复制两次以上
Actor的网络更新时间。 长期来看，我们期待着改善。

从服务器调用的 RPC

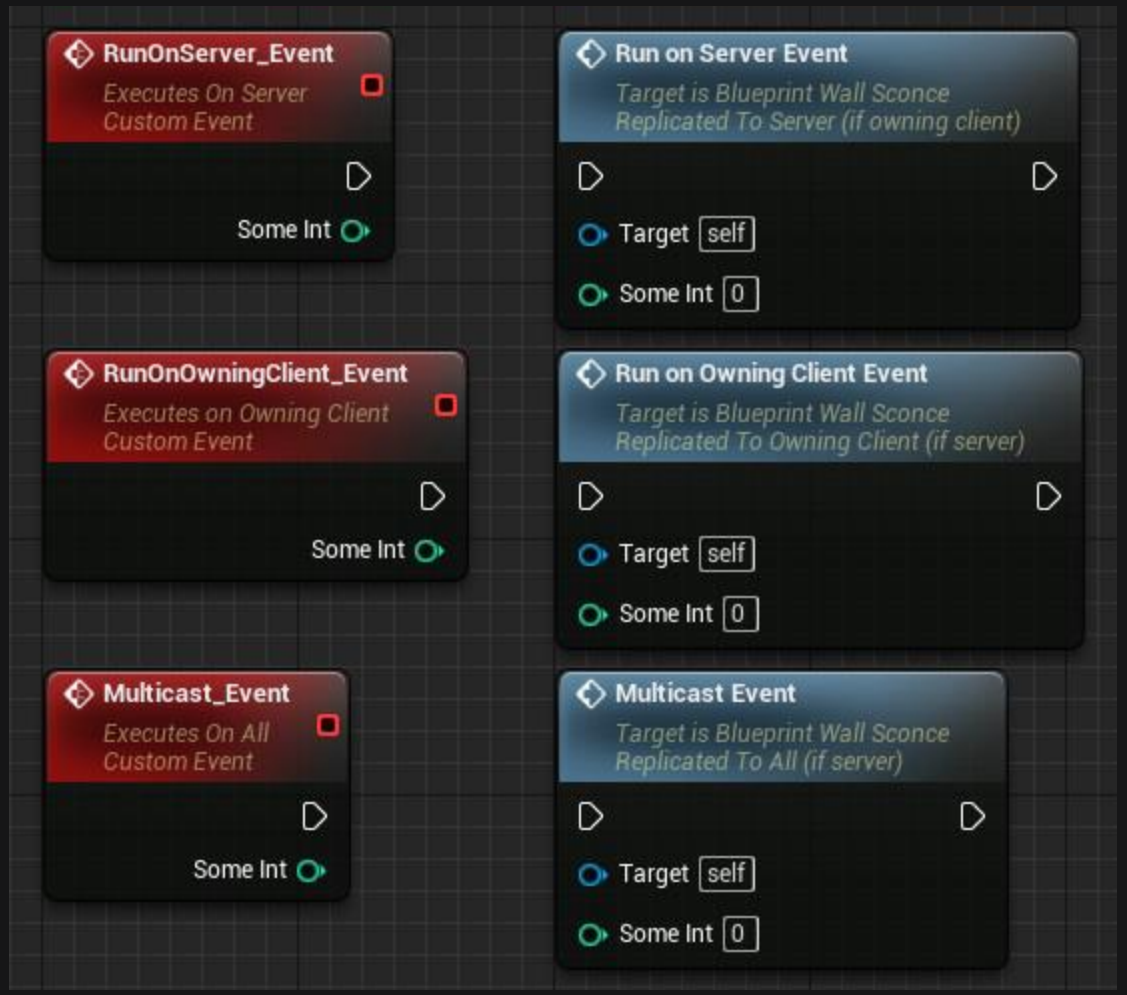
Actor 所有权	未复制	NetMulticast	Server	Client
Client-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在 actor 的所属客户端上运行
Server-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行
Unowned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行

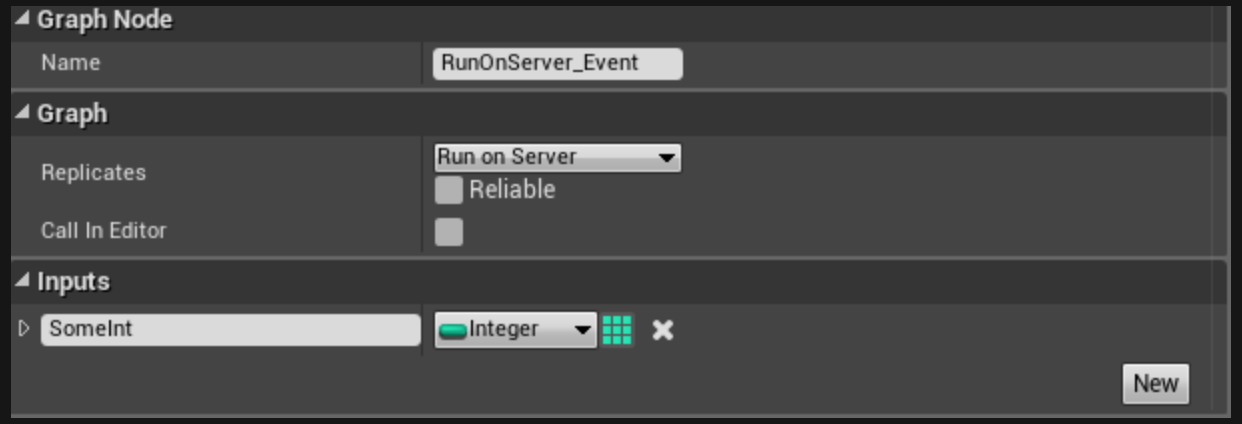
从客户端调用的 RPC

Actor 所有权	未复制	NetMulticast	Server	Client
Owned by invoking client	在执行调用的客户端上运行	在执行调用的客户端上运行	在服务器上运行	在执行调用的客户端上运行
Owned by a different client	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行
Server-owned actor	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行
Unowned actor	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行

RPCs in Blueprints

Blueprints中的RPC是通过创建CustomEvents并将其设置为Replicate创建的。
RPC不能有返回值！ 所以函数不能用来创建它们。





“Reliable”复选框可用于将RPC标记为“important”，
确保执行99.99%，而不会丢失
由于连接问题等等。
注意：不要将每个RPC标记为Reliable！

RPCs in C++

要使用C++中的整个网络内容，您需要在项目头文件中包含“UnrealNetwork.h”！ C++中的RPC比较容易创建，我们只需要将说明符添加到UFUNCTION（）宏。

```
// This is a Server RPC, marked as unreliable and WithValidation (is needed!)
UFUNCTION(Server, unreliable, WithValidation)
void Server_PlaceBomb();
```

CPP文件将实现不同的函数。 这个需要'_Implementation'作为后缀。

```
// This is the actual implementation (Not Server_PlaceBomb). But when calling it, we use "Server_PlaceBomb"
void ATestPlayerCharacter::Server_PlaceBomb_Implementation() {
    // BOOM!
}
```

CPP文件还需要一个带有“_Validate”作为后缀的版本。 后来更多关心这个。

```
bool ATestPlayerCharacter::Server_PlaceBomb_Validate() {
    return true;
}
```

另外两种类型的RPC是这样创建的：

客户端RPC需要标记为“reliable”或“unreliable”！

```
UFUNCTION(Client, unreliable)
void ClientRPCFunction();
```

多播RPC，也需要标记为“reliable”或“unreliable”！

```
UFUNCTION(NetMulticast, unreliable)
void MulticastRPCFunction();
```

当然，我们也可以在RPC中添加'reliable'关键字，使其可靠。

```
UFUNCTION(Client, reliable)
void ReliableClientRPCFunction();
```

Validation（C++）

Validation的想法是，如果RPC的验证功能检测到任何参数不正确，
它可以通知系统断开RPC调用的客户端/服务器。
对于每个ServerRPCFunction，现在需要验证。 'WithValidation'关键字
在UFUNCTION宏中用于此。

```
UFUNCTION(Server, unreliable, WithValidation)
void SomeRPCFunction(int32 AddHealth);
```

以下是可以使用“_Validate”函数的示例：

```
bool ATestPlayerCharacter::SomeRPCFunction_Validate(int32 AddHealth) {
    if(AddHealth > MAX_ADD_HEALTH) {
          return false;                    // This will disconnect the caller!
    }
    return true;                           // This will allow the RPC to be called!
}
```