1.3 创建和加入Session

1.3.1 创建多人游戏菜单界面

（1）修改WB_MainMenu界面

在开始之前，我们先美化一下我们之前的游戏菜单界面。我们添加一个Border在Canvas Panel下面。接着修改这个Border的定位器拖拽到合适的位置，然后设置Slot/OffsetLeft，OffsetTop，OffsetRight，OffsetBottom都为0，那么这时，Border边框就会定位到定位器上，如图1.3.1.1所示。接下来，我们修改Border的Appearance/Brush Color的透明度A为0.5。

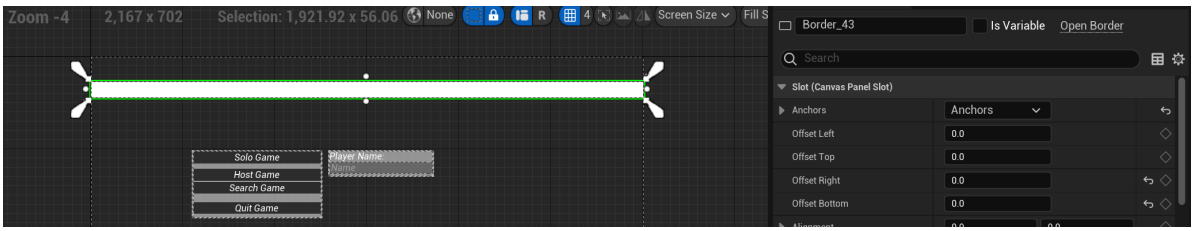

图1.3.1.1 设置Border定位器

然后，我们拖拽一个Text到Border层级下，修改Content/Text为“Main Menu”。然后复制我们SoloGame的字体设置，粘贴在这个Text中。然后修改一下Appearance/ Font/ Size为50。然后我们可以调整一下Border的大小使得匹配文字的大小。我们修改Text的Slot/Horizontal Alignment和Vertical Alignment为居中。最后的界面如图1.3.1.2所示。然后，我们保存关闭这个蓝图。

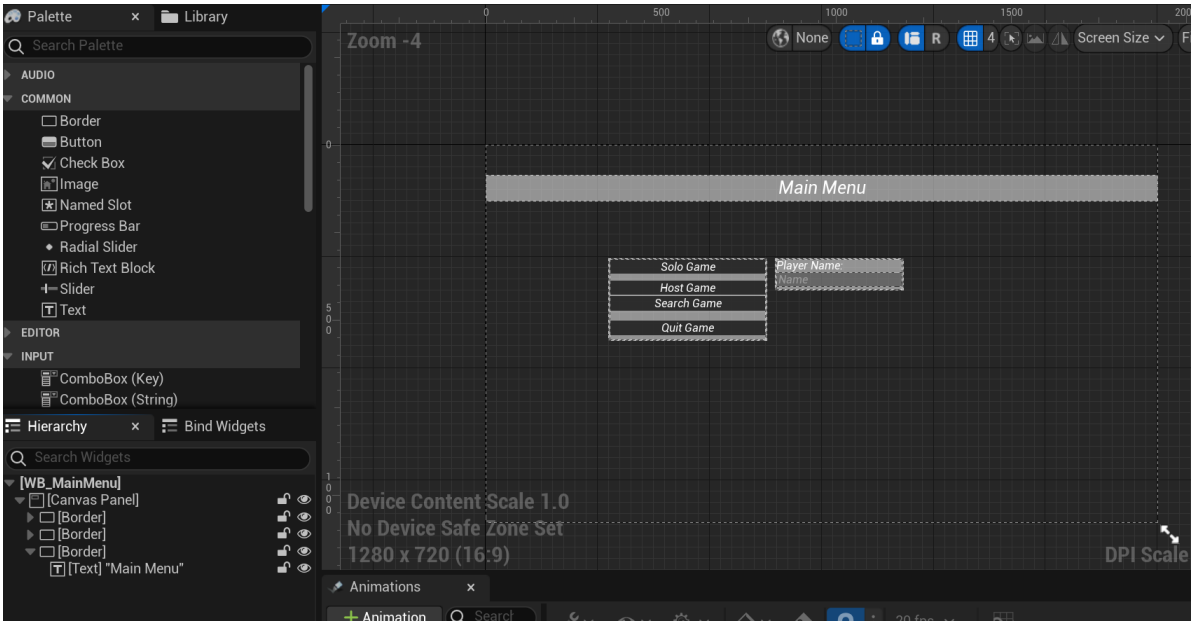

图1.3.1.2 修改后的开始菜单

在Content Browser界面将这个蓝图复制粘贴一份出来命名为WB_MpMenu，意味着Multiplayer多人菜单。

（2）修改WB_MpMenu界面

接着，我们打开WB_MpMenu蓝图，修改我们刚才设置的Text文本从“Main Menu”改成“Multiplayer”。然后我们删除用于输入Player Name的一整个Border组件。因为在这个界面我们并不需要输入用户名。接着，我们移动一下我们左侧的按钮菜单的顺序，如图1.3.1.3所示。

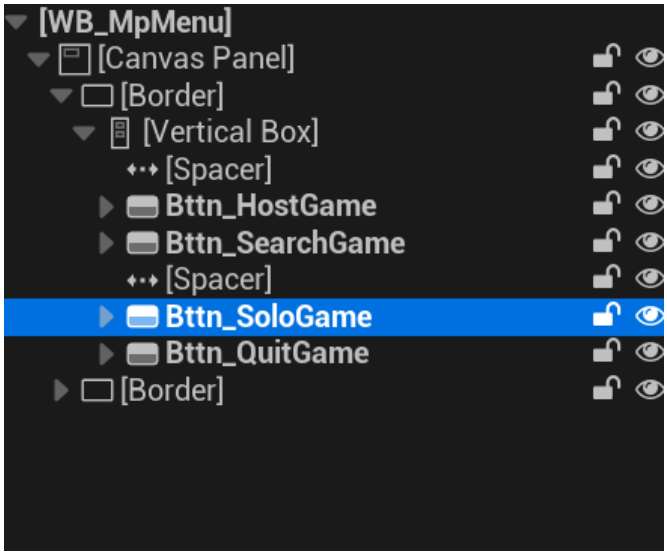

图1.3.1.3 修改按钮顺序

修改Bttn_SoloGame的按钮名称为Bttn_ConnectionType，按钮的Text文本改成Connection。

我们拖拽一个Border到Vertical Box层级下，并添加一个Text组件进去。我们修改Border的Appearance/ BrushColor为（0，0，0，0.3），同时我们复制上面的字体粘贴到这个Text中。接着将这个Text进行居中设置。修改Content/ Text文本内容为“LAN”表示局域网。我们将来会修改这个文本的内容，因此我们修改这个文本的变量名称为Text_Connecttype并且勾选Is Variable。

我们将这个表示LAN的Border移动到Connection的按钮下方。将原本在SearchGame下方的Spacer移动到LAN的Border组件下方。将最上方的Spacer移动到SearchGame下。

然后，我们修改Bttn_QuitGame的变量名为Bttn_MainMenu，并将文本内容更改为MainMenu。

最后我们得到的层级如下图1.3.1.4所示。

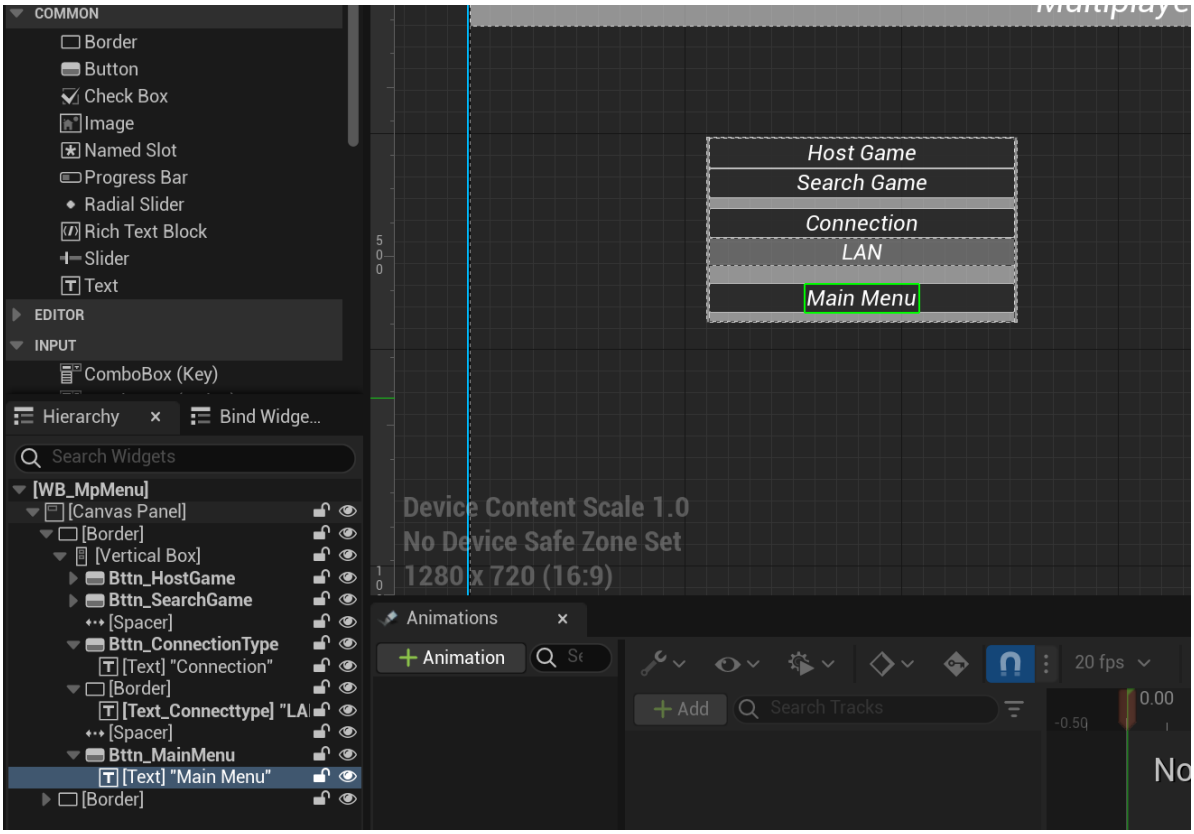

图1.3.1.4 修改WB_MpMenu

我们回到WB_MainMenu主菜单中，删除Search Game按钮。因为我们在WB_MpMenu菜单中已经创建了这个按钮。然后我们修改HostGame的文本内容为Mutiplayer，修改按钮的变量名为Bttn_MPGame。接着调整一下边框到合适大小。得到如图1.3.1.5的层次和界面。

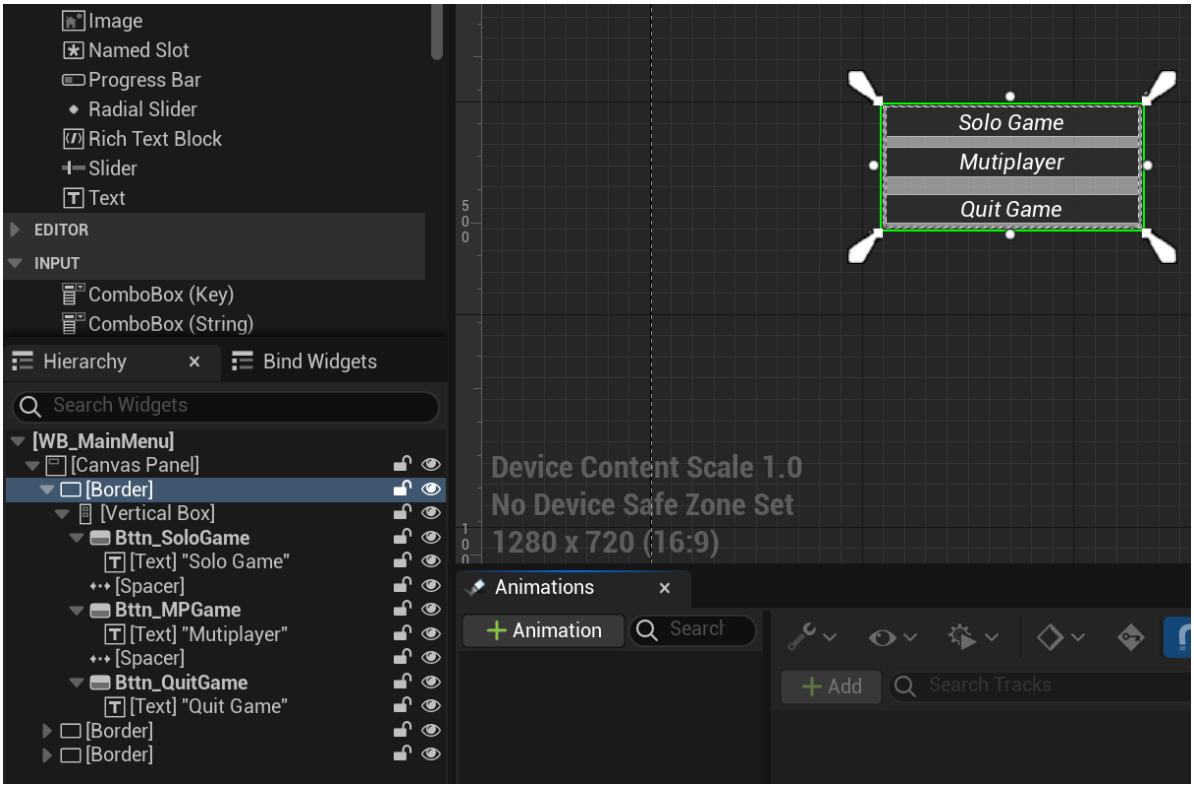

图1.3.1.5 修改WB_MainMenu

此时，我们就完成了WB_MpMenu的绘制。

1.3.2 编写切换菜单的代码

（1）创建MPUserWidget类

在编辑器中创建一个继承于UserWidget的C++类，命名为MPUserWidget，然后我们编写我们的TutorialGameInstance类。添加如下的声明。

```
UPROPERTY(EditAnywhere, Category = "UI")
TSubclassOf<UMPUserWidget> MPUserWidgetClass;

UPROPERTY()
UMPUserWidget* MPMenu;
```

编译保存之后，设置WB_MpMenu的父类为MPUserWidget，接着在TutorialGameInstance编辑器面板中进行WB_MpMenu的绑定。

（2）从WB_MainMenu切换到WB_MpMenu

首先，在TutorialGameInstance的C++中声明一个函数UI_ShowMpMenu

```
UFUNCTION()
void UI_ShowMpMenu();
```

在cpp中创建对应的定义。

```
#include "MPUserWidget.h"
void UTutorialGameInstance::UI_ShowMpMenu()
{
    if (!IsValid(MPMenu)) {
        MPMenu = CreateWidget<UMPUserWidget>(GetWorld(), MPUserWidgetClass);
    }
    MPMenu->AddToViewport(0);
}
```

这样我们就能够切换到MPMenu界面了。接下来，我们在MainMenuUserWidget的头文件中绑定我们的Bttn_MPGame按键。我们首先对按键和响应函数进行声明。

```
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UButton* Bttn_MPGame;

UFUNCTION()
void OnBttn_MPGameClick();
```

然后在cpp中创建对应的定义和将按钮绑定到函数上。

```
void UMainMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_QuitGame) {...}
    if (Bttn_SoloGame) {...}
    if (Input_PlayerName) {...}
    if (Bttn_MPGame) {
        Bttn_MPGame->OnClicked.AddDynamic(this,
&UMainMenuUserWidget::OnBttn_MPGameClick);
    }
}
void UMainMenuUserWidget::OnBttn_MPGameClick()
{
    MyGameInstance->UI_ShowMpMenu();
    RemoveFromParent();
}
```

我们点击按钮的时候，会调用我们的UI_ShowMpMenu函数。并且还会调用RemoveFromParent函数，将当前的 Widget 从其父 Widget 或用户界面层级中移除，但不会销毁该 Widget 对象。它可以在运行时动态调整用户界面。

（3）从WB_MpMenu切换到WB_MainMenu

首先，我们仿照MainMenuUserWidget这个C++类在MPUserWidget的C++类中创建基本的声明和定义。

MPUserwidget.h

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "MPUserWidget.generated.h"
```

```
class UButton;
class UTutorialGameInstance;
/**
 *
 */
UCLASS()
class UETUTORIAL_API UMPUserWidget : public UUserWidget
{
    GENERATED_BODY()
public:
    virtual void NativeOnInitialized() override;
protected:
    virtual void NativeConstruct() override;
    UTutorialGameInstance* MyGameInstance;
};
```

MPUserwidget.cpp

```
void UMPUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
}
void UMPUserWidget::NativeConstruct()
{
    Super::NativeConstruct();
}
```

接下来，我们绑定蓝图中的按钮Bttn_MainMenu和响应按钮点击的函数。

```
public:
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_MainMenu;

    UFUNCTION()
    void OnBttn_MainMenuClick();
```

在cpp中创建它的定义。

```
void UMPUserWidget::OnBttn_MainMenuClick()
{
    MyGameInstance->UI_ShowMainMenu();
    RemoveFromParent();
}
void UMPUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_MainMenu) {
        Bttn_MainMenu->OnClicked.AddDynamic(this,
&UMPUserWidget::OnBttn_MainMenuClick);
    }
}
```

我们将这个OnBttn_MainMenuClick函数绑定到Bttn_MainMenu按钮上。

此时，我们点击保存编译运行，就可以切换两个界面了。

1.3.3 界面文本的改变

首先，我们在TutorialGameInstance中创建一个变量bIsLanConnection用于标识是否处于网络连接状态。

```
UPROPERTY(EditAnywhere)
bool bIsLanConnection = false;
```

我们创建一个函数用于改变当前状态。

```
UFUNCTION()
bool ChangeConnectionType();
```

在cpp中我们创建对应的定义。

```
bool UTutorialGameInstance::ChangeConnectionType()
{
    bIsLanConnection = !bIsLanConnection;
    return bIsLanConnection;
}
```

接着，我们创建一个函数用于修改WB_MpMenu界面的文本, 并且对文本Text_Connecttype进行声明。

```
UFUNCTION()
void SetConnectionType_Text(bool IsLanConnenction);

UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UTextBlock* Text_Connecttype;
```

我们在cpp中创建对应的定义。

```
#include "Components/TextBlock.h"
void UMPUserWidget::SetConnectionType_Text(bool IsLanConnenction)
{
    if (IsLanConnenction) {
        Text_Connecttype->SetText(FText::FromString(TEXT("LAN")));
    }
    else {
        Text_Connecttype->SetText(FText::FromString(TEXT("Online")));
    }
}
```

这样子，我们就能修改我们的文本内容了。

我们在每次载入页面的时候进行判断，判断此时是否处于连接状态。我们修改我们的NativeConstruct函数。

```
void UMPUserWidget::NativeConstruct()
{
    Super::NativeConstruct();
    UE_LOG(LogTemp, Warning, TEXT("NativeConstruct called"));
    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
    SetConnectionType_Text(MyGameInstance->bIsLanConnection);
}
```

我们获取MyGameInstance->bIsLanConnection传入我们更改文本的函数即可。

1.3.4 创建会话

（1）准备工作

我们先确保所有内容已经保存，并且UE编辑器正在运行中，接着我们在VS2022中修改以下文件信息。

我们需要引入一些模块，在*.Build.cs的文件中，在PublicDependencyModuleNames后添加"OnlineSubsystem", "OnlineSubsystemUtils", "OnlineSubsystemNull"，如同下面的代码。

```
        PublicDependencyModuleNames.AddRange(new string[] { "Core",
"CoreUObject", "Engine", "InputCore", "EnhancedInput", "Niagara", "UMG",
"SlateCore", "OnlineSubsystem", "OnlineSubsystemUtils", "OnlineSubsystemNull"
});
```

接下来，我们需要在DefaultEngine.ini （这个文件在Config中）的末尾加入下面两行代码。

```
[OnlineSubsystem]
DefaultPlatformService=Null
```

我们接下来在UE编辑器中，点击上方菜单栏的Tools/Refresh Visual Studio Project，刷新一下项目，然后关闭编辑器并重新开始调试即可。

（2）创建OnCreateSession多播委托

我们在TutorialGameInstance中创建一个多播委托DECLARE_MULTICAST_DELEGATE(FCreateMPSessionEventDelegate)并声明事件和触发这个事件的函数。

```
#include "OnlineSubsystem.h"
#include "Interfaces/OnlineSessionInterface.h"
#include "OnlineSessionSettings.h"
#include "Online.h"
DECLARE_MULTICAST_DELEGATE(FCreateMPSessionEventDelegate);
...
FCreateMPSessionEventDelegate OnCreateSession;
void TriggerSessionCreation();
```

函数的实现如下：

```
void UTutorialGameInstance::TriggerSessionCreation()
{
    OnCreateSession.Broadcast();
}
```

创建一个函数绑定到这个委托上。

```
UFUNCTION()
void OnCreateMPSession();
```

.c 文件中进行定义，并在Init函数中进行绑定。

```
void UTutorialGameInstance::Init()
{
    Super::Init();

    OnStartSoloGameEventTriggered.AddDynamic(this,
&UTutorialGameInstance::StartSoloGameEvent);
    ChangePlayerNameEvent.AddDynamic(this,
&UTutorialGameInstance::OnPlayerNameChanged);
    OnCreateSession.AddUObject(this ,&UTutorialGameInstance::OnCreateMPSession);
}
void UTutorialGameInstance::OnCreateMPSession()
{
    IsSoloGame = false;
    StartLocalSessionCreation(MaxNumPublicConnections);//在这里
MaxNumPublicConnections参数是一个头文件中定义的变量，我们后续会定义它。
StartLocalSessionCreation这个函数我们也会进行声明并定义。
}
```

在这里，我们在Init中对OnCreateSession进行绑定时，由于是一个非动态的多播委托，所以我们使用AddUObject进行绑定。然后，我们在.h头文件中我们可以添加MaxNumPublicConnections和StartLocalSessionCreation的声明。

```
UPROPERTY(EditAnywhere)
int32 MaxNumPublicConnections = 4;

void StartLocalSessionCreation(int32 NumPublicConnections);
```

在.c文件中进行定义。简单的调用一个CreateLocalSession函数即可，后续我们会完成CreateLocalSession函数的实现。

```
void UTutorialGameInstance::StartLocalSessionCreation(int32
NumPublicConnections)
{
    CreateLocalSession(NumPublicConnections);  // 创建会话
}
```

（3）调用UE的Online系统实现创建会话

我们需要先在TutorialGameInstance.h声明几个变量用于存储我们的会话信息和创建会话。

```
TSharedPtr<FOnlineSessionSearch> SessionSearch;
IOnlineSessionPtr SessionInterface;
```

第一个变量用于存储我们后续搜索会话得到的信息，第二个变量用于保存我们创建会话使用的接口。在Init中定义这个接口。

```
#include "OnlineSubsystem.h"
```

```
#include "Interfaces/OnlineSessionInterface.h"
#include "OnlineSessionSettings.h"
void UTutorialGameInstance::Init()
{
    Super::Init();
    OnStartSoloGameEventTriggered.AddDynamic(this,
&UTutorialGameInstance::StartSoloGameEvent);
    ChangePlayerNameEvent.AddDynamic(this,
&UTutorialGameInstance::OnPlayerNameChanged);
    OnCreateSession.AddUObject(this ,&UTutorialGameInstance::OnCreateMPSession);

    if (IOnlineSubsystem* Subsystem = IOnlineSubsystem::Get()) {
        SessionInterface = Subsystem->GetSessionInterface();
    }
}
```

在TutorialGameInstance.h中声明一个CreateLocalSession函数，用于实现我们的创建Session过程。

```
void CreateLocalSession(int32 NumPublicConnections);
```

接下来，我们在TutorialGameInstance.c 中完成它的定义。

```
void UTutorialGameInstance::CreateLocalSession(int32 NumPublicConnections)
{
    if (SessionInterface.IsValid()) {
        FName SessionName (*PlayerProfileinfo.PlayerName.ToString());

        // 配置会话设置
        FOnlineSessionSettings SessionSettings;
        SessionSettings.bIsLANMatch = bIsLanConnection;                          //
使用局域网
        SessionSettings.bShouldAdvertise = true;                    // 广播会话信息
        SessionSettings.NumPublicConnections = NumPublicConnections; // 最大玩家数
        SessionSettings.bAllowJoinInProgress = true;                // 允许加入进行中
的会话
        SessionSettings.bUsesPresence = true;                      // 不使用在线状态
        SessionSettings.bIsDedicated = false; // 设置为非专用服务器

        SessionSettings.Set(FName(TEXT("SERVER_NAME")),
PlayerProfileinfo.PlayerName.ToString(),
EOnlineDataAdvertisementType::ViaOnlineService);

        // 创建会话
        bool bResult = SessionInterface->CreateSession(0, FName("Create
Session"), SessionSettings);
        if (bResult) {
            GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString::Printf(TEXT("Session creation started successfully.")));
        }
    }
}
```

我们先判断这个SessionInterface接口的可用性，然后创建一个FName类型的变量，用于存储当前角色的名称。接着创建一个会话设置，进行配置。最后的SessionSettings.Set设置的是一个自定义的键值对，然后EOnlineDataAdvertisementType::ViaOnlineService的作用是在线服务公开这个数据信息。接着，我们SessionInterface->CreateSession创建这个会话，第一个参数一般用作0，标识本地玩家控制器，第二个参数用于标识Session的名字，第三个参数是我们刚才设置的SessionSettings，存储了我们创建会话的信息。

（4）绑定创建完毕的回调函数。

我们在TutorialGameInstance.h声明一个OnCreateSessionComplete函数用于回调。

```
void OnCreateSessionComplete(FName SessionName, bool bWasSuccessful);
```

并且我们需要在Init中对其进行绑定。

```
void UTutorialGameInstance::Init()
{
    Super::Init();

    OnStartSoloGameEventTriggered.AddDynamic(this,
&UTutorialGameInstance::StartSoloGameEvent);
    ChangePlayerNameEvent.AddDynamic(this,
&UTutorialGameInstance::OnPlayerNameChanged);
    OnCreateSession.AddUObject(this ,&UTutorialGameInstance::OnCreateMPSession);

    if (IOnlineSubsystem* Subsystem = IOnlineSubsystem::Get()) {
        SessionInterface = Subsystem->GetSessionInterface();
        if (SessionInterface.IsValid())
        {
            // 绑定 OnCreateSessionComplete 委托
            SessionInterface->OnCreateSessionCompleteDelegates.AddUObject(this,
&UTutorialGameInstance::OnCreateSessionComplete);
        }
    }
}
```

并且定义我们刚绑定的OnCreateSessionComplete函数。

```
void UTutorialGameInstance::OnCreateSessionComplete(FName SessionName, bool
bWasSuccessful)
{
    if (bWasSuccessful)
    {
        GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString::Printf(TEXT("Session %s created successfully!"),
*SessionName.ToString()));
        GetWorld()->ServerTravel("LobbyMenu_Map?listen");
    }
}
```

当我们创建Session结束的时候，我们就会进入这段函数，首先我们判断是否创建成功。如果成功，我们就会进入GetWorld()->ServerTravel创建一个LobbyMenu_Map地图，其中?listen表示以这个方式打开。

在这个时候，我们只需要调用TriggerSessionCreation()这个函数就可以创建一个会话了。

1.3.5 加入会话

首先，我们在TutorialGameInstance.h中创建一个带有一个参数的多播委托，用于加入会话。

```
DECLARE_MULTICAST_DELEGATE_OneParam(FJoinMPSessionEventDelegate,FOnlineSessionSe
archResult&);
```

接着，我们对委托事件进行声明。并且声明它的触发函数，我们加入Session的函数和加入Session后调用的函数。

```
FJoinMPSessionEventDelegate OnJoinSession;//委托声明
void TriggerSessionJoin(FOnlineSessionSearchResult& SessionResult);//触发事件
void OnJoinMPSession(FOnlineSessionSearchResult& SessionResult);//加入Session的函
数
virtual void OnJoinSessionComplete(FName SessionName,
EOnJoinSessionCompleteResult::Type Result);//Join成功后调用的函数
```

接下来我们在TutorialGameInstance.cpp中进行具体的实现。首先我们需要在Init中，为事件绑定对应的函数。

```
void UTutorialGameInstance::Init()
{
    Super::Init();

    OnStartSoloGameEventTriggered.AddDynamic(this,
&UTutorialGameInstance::StartSoloGameEvent);
    ChangePlayerNameEvent.AddDynamic(this,
&UTutorialGameInstance::OnPlayerNameChanged);
    OnCreateSession.AddUObject(this ,&UTutorialGameInstance::OnCreateMPSession);
    OnJoinSession.AddUObject(this, &UTutorialGameInstance::OnJoinMPSession);

    if (IOnlineSubsystem* Subsystem = IOnlineSubsystem::Get()) {
        SessionInterface = Subsystem->GetSessionInterface();
        if (SessionInterface.IsValid())
        {
            // 绑定OnCreateSessionComplete函数到 OnCreateSessionCompleteDelegates
委托
            SessionInterface->OnCreateSessionCompleteDelegates.AddUObject(this,
&UTutorialGameInstance::OnCreateSessionComplete);
            //绑定OnJoinSessionComplete函数到OnJoinSessionCompleteDelegates委托
            SessionInterface->OnJoinSessionCompleteDelegates.AddUObject(this,
&UTutorialGameInstance::OnJoinSessionComplete);
        }
    }
}
```

然后，我们编写TriggerSessionJoin的逻辑。这里就是进行事件的触发。

```
void UTutorialGameInstance::TriggerSessionJoin(FOnlineSessionSearchResult&
SessionResult)
{
    OnJoinSession.Broadcast(SessionResult);
}
```

在OnJoinMPSession函数中，我们会加入会话，会话的信息从传入的参数SessionResult获得。

```
void UTutorialGameInstance::OnJoinMPSession(FOnlineSessionSearchResult&
SessionResult)
{
    IsSoloGame = false;
    SessionInterface->JoinSession(0, FName("Create Session"), SessionResult);

    FString ServerName;
    SessionResult.Session.SessionSettings.Get(FName(TEXT("SERVER_NAME")),
ServerName);
    GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString::Printf(TEXT("JoinSession successfully! %s"), *ServerName));
}
```

这个函数中，我们会将IsSoloGame设置为false。然后调用SessionInterface->JoinSession来加入Session，第一个参数0表示当前玩家。第二个参数表示标识会话的名称，这个名称应该与我们在CreateSession中的第二个参数相同。第三个参数传入我们的搜索结果SessionResult，这里面包含了许多Session的信息，包括我们之前的SessionSettings。

接着，我们应该处理加入Session后的函数OnJoinSessionComplete。

```
void UTutorialGameInstance::OnJoinSessionComplete(FName SessionName,
EOnJoinSessionCompleteResult::Type Result)
{
    GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString::Printf(TEXT("In OnJoinSessionComplete!")));
    if (APlayerController* PController =
UGameplayStatics::GetPlayerController(GetWorld(), 0)) {
        FString JoinAddress = "";
        SessionInterface->GetResolvedConnectString(SessionName, JoinAddress);
        if (JoinAddress != "") {
            PController->ClientTravel(JoinAddress,
ETravelType::TRAVEL_Absolute);
        }
    }
}
```

在这个函数中，我们会创建进行地图的载入。第二个参数EOnJoinSessionCompleteResult::Type Result用于表示加入会话的状态，枚举值包括：

- Success：成功加入会话。
- SessionIsFull：会话已满。
- SessionDoesNotExist：会话不存在。
- CouldNotRetrieveAddress：无法解析会话地址。
- AlreadyInSession：已经处于会话当中。

我们通过SessionInterface->GetResolvedConnectString会话接口获取目标会话的连接地址，并存放到JoinAddress当中。如果获取失败，那么JoinAddress保持为空。在检验非空之后，PController->ClientTravel将客户端跳转到目标服务器。

1.3.6 搜索结果列表

首先，在WB_MpMenu中修改按钮组件的上一层级的Border名字为Border_MenuButton设置为IsVariable。设置Bttn_SearchGame的Text组件的文本内容为Text_SearchGameButton并设置为IsVariable。拖拽一个Border放在Canvas下命名为Border_ServerList并勾选IsVariable，修改一下定位的位置，透明度设置为0.5。

接下来，我们往Border中拖拽入一个Scroll Box命名为ScrollBox_ServerList并勾选IsVariable，用于充当服务器列表。

我们接下来创建一个UserWidget蓝图组件命名为WB_ServerRow，这个组件将用来填充我们的ScrollBox_ServerList。在这个画板中，我们拖拽入一个Canvas Panel组件。

接下来，往CanvasPanel中拖拽入一个Button命名为Bttn_JoinGame设置为IsVariable，设置Button的Appearance/Style下的每一种状态的Tint都为（0，0，0，0.7）。

往Button中拖拽一个HorizontalBox水平框进来，并且往HorizontalBox中拖拽一个Text进来。我们复制在主菜单中的字体过来，粘贴在这个Text的属性上，设置Size为20。设置这个Text的属性名为Text_ServerName,设置内容为ServerName。并将这个Text设置为IsVariable。设置Text的Slot/Size为Fill，然后对应的HorizontalAlignment为左，VerticalAlignment为中。设置Appearance/Min Desired Width为175。

我们将这个Text复制并粘贴出来，放到HorizontalBox中。将复制出来的名称改为Text_NumPlayer，设置内容为0/0，修改Slot/HorizontalAlignment为中，VerticalAlignment为中，设置Appearance/Min Desired Width为120，设置Appearance/ Justification为中。

我们将刚才的Text_NumPlayer复制并粘贴出来放到HorizontalBox中，名称改为Text_Ping，设置内容为0ms，修改Slot/ HorizontalAlignment为右，设置Appearance/Min Desired Width为175。

然后我们选择HorizontalBox水平框，我们在Slot/HorizontalAlignment设置为第四个Fill Horizontally，VerticalAlignment也一样设置。

我们接下来为创建一个继承于UserWidget的C++类，命名为ServerRowUserWidget，并将它作为WB_ServerRow的父类。

这一切做完之后，我们就获得了一个用于存放搜索结果的界面，如图1.3.6.1所示。

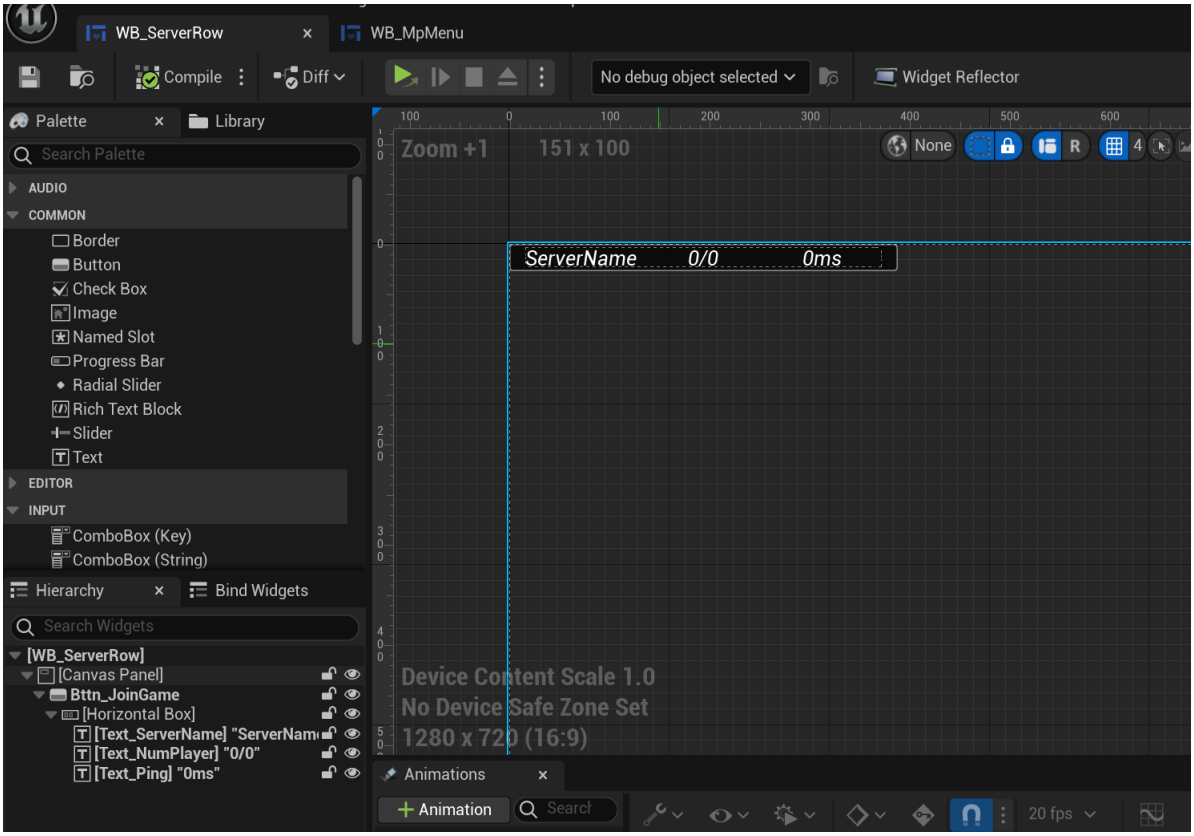

图1.3.6.1 存放搜索结果的组件

我们在ServerRowUserWidge.h中对我们的文本组件和按钮进行声明。

```
public:
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UTextBlock* Text_ServerName;

    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UTextBlock* Text_NumPlayer;

    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UTextBlock* Text_Ping;

    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_JoinGame;
```

然后，我们在ServerRowUserWidge.h中创建一个FOnlineSessionSearchResult*类型的变量Result。这个变量会用于存储当前这个组件保存的Result信息。

```
FOnlineSessionSearchResult* Result;
```

我们声明一个WB_MpMenu变量，我们会通过这个变量来修改上一级的按钮显示文字之类的。

```
UPROPERTY()
UMPUserWidget* WB_MpMenu;
```

接着，我们在ServerRowUserWidge.h创建一个委托事件DECLARE_MULTICAST_DELEGATE_OneParam (FUpdateSeverRowInfoEventDelegate, FOnlineSessionSearchResult&)用于更新我们的列表。并声明其触发函数。

```
DECLARE_MULTICAST_DELEGATE_OneParam(FUpdateSeverRowInfoEventDelegate,
FOnlineSessionSearchResult&);//带有一个参数的多播委托

FUpdateSeverRowInfoEventDelegate UpdateSeverRowInfoEvent;//声明这个委托事件
void TriggerUpdateServerRowInfo(FOnlineSessionSearchResult& SessionResult);//触发
这俄格事件

private:
void UpdateServerRow_Info(FOnlineSessionSearchResult& SessionResult);//绑定在这个事
件上的函数
```

我们需要在ServerRowUserWidge.h声明一个初始化函数NativeOnInitialized来初始化这个组件。

```
virtual void NativeOnInitialized() override;
```

接着，我们在ServerRowUserWidge.cpp中创建这部分的定义。

```
void UServerRowUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    UpdateSeverRowInfoEvent.AddUObject(this,
&UServerRowUserWidget::UpdateServerRow_Info);
}
```

在这里面，我们对委托事件，绑定了响应函数。接下来，我们实现这个响应函数和触发函数。

```
void UServerRowUserWidget::TriggerUpdateServerRowInfo(FOnlineSessionSearchResult&
SessionResult)
{
    UpdateSeverRowInfoEvent.Broadcast(SessionResult);
}

void UServerRowUserWidget::UpdateServerRow_Info(FOnlineSessionSearchResult&
SessionResult)
{
    Result = &SessionResult;
    if (!Result) {
        UE_LOG(LogTemp, Error, TEXT("UpdateServerRow_Info is failed"));
        return;
    }
    if (Text_ServerName) {
        FString ServerName;
        if (Result->Session.SessionSettings.Get(FName(TEXT("SERVER_NAME")),
ServerName)) {
            Text_ServerName->SetText(FText::FromString(ServerName));
            GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString::Printf(TEXT("ServerName %s "), *ServerName));
        }
        else {
            UE_LOG(LogTemp, Warning, TEXT("Server Name not found"));
            Text_ServerName->SetText(FText::FromString(TEXT("Server Name not
found")));
            GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green,
*FString::Printf(TEXT("Server Name not found")));
        }
```

```
    }

    if (Text_NumPlayer) {
        int32 MaxPlayers  = Result-
>Session.SessionSettings.NumPublicConnections;
        int32 OpenSlots = Result->Session.NumOpenPublicConnections;
        int32 CurrentPlayers = MaxPlayers - OpenSlots;
        FString PlayerCountString = FString::Printf(TEXT("%d/%d"),
CurrentPlayers + 1, MaxPlayers);
        FText PlayerCountText = FText::FromString(PlayerCountString);
        Text_NumPlayer->SetText(PlayerCountText);
    }

    if (Text_Ping) {
        Text_Ping->SetText(FText::FromString(FString::Printf(TEXT("%d ms"),
Result->PingInMs)));
    }
}
```

在UpdateServerRow_Info中，我们会先判断传入的Result为非空。然后我们会修改Text_ServerName，Text_NumPlayer和Text_Ping的内容。具体的内容我们可以通过Session.SessionSettings.Get(FName(TEXT("SERVER_NAME")), ServerName)将我们获得之前存放的键值对信息进行获取。Session.SessionSettings.NumPublicConnections获取我们的最大玩家数。

接下来，我们会对为这个组件的按钮点击进行函数的绑定和实现。在ServerRowUserWidget.h声明JoinGame函数。

```
UFUNCTION()
void JoinGame();
```

在ServerRowUserWidget.cpp中实现它。不过，我们先在NativeOnInitialized中进行点击事件的绑定。

```
void UServerRowUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    UpdateSeverRowInfoEvent.AddUObject(this,
&UServerRowUserWidget::UpdateServerRow_Info);
    if (Bttn_JoinGame) {
        Bttn_JoinGame->OnClicked.AddDynamic(this,
&UServerRowUserWidget::JoinGame);
    }
}
```

然后，我们实现JoinGame函数，这个函数主要是调用GameInstance的触发事件函数。

```
void UServerRowUserWidget::JoinGame()
{
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());

    MyGameInstance->TriggerSessionJoin(*Result);

    if (WB_MpMenu) {
        WB_MpMenu->Border_MenuButton->SetIsEnabled(false);
```

```
        WB_MpMenu->Border_ServerList->SetIsEnabled(false);
    }
    else {
        GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Green, TEXT("JoinGame
is failed, WB_MpMenu is nullptr"));
        return;
    }
}
```

触发事件之后，我们会将整个菜单页面变为不可点击，防止用户点击后退出界面，这个时候整个界面会进行等待状态。

1.3.7 绑定按钮和搜索会话

（1）为Bttn_HostGame按钮绑定函数

在MPUserWidget.cpp中我们为Bttn_HostGame添加一个点击事件后的响应函数OnBttn_HostGameClick。

```
void UMPUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();

    if (Bttn_MainMenu) {...}

    if (Bttn_HostGame) {
        Bttn_HostGame->OnClicked.AddDynamic(this,
&UMPUserWidget::OnBttn_HostGameClick);
    }

    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
}
```

这个OnBttn_HostGameClick函数是用于调用GameInstance中的触发函数的，在MPUserWidget.h中我们进行如下的声明。

```
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UButton* Bttn_HostGame;

UFUNCTION()
void OnBttn_HostGameClick();
```

点击HostGame后，我们进行触发创建会话事件。

```
void UMPUserWidget::OnBttn_HostGameClick()
{
    MyGameInstance->TriggerSessionCreation();
}
```

（2）实现Bttn_SearchGame

我们需要先声明Border_ServerList，这个组件将用于展示我们的搜索结果。同时也声明我们当时在蓝图中要声明的变量，在MPUserWidget.h中添加以下声明。

```
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UBorder* Border_ServerList;
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UTextBlock* Text_SearchGameButton;
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UScrollBox* ScrollBox_ServerList;
```

接着，在MPUserWidget.h为我们的Bttn_SearchGame添加点击函数。

```
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UButton* Bttn_SearchGame;

UFUNCTION()
void OnBttn_SearchGameClick();
```

我们还需要绑定查找完毕后的响应函数。OnFindSessionsComplete。

```
void OnFindSessionsComplete(bool bWassuccessful);
```

在cpp中这样实现。修改NativeOnInitialized初始化函数。

```
void UMPUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();

    if (Bttn_MainMenu) {...}
    if (Bttn_HostGame) {...}
    if (Bttn_SearchGame) {
        Bttn_SearchGame->OnClicked.AddDynamic(this,
&UMPUserWidget::OnBttn_SearchGameClick);
    }
    if (Border_ServerList) {
        Border_ServerList->SetVisibility(ESlateVisibility::Hidden);
    }
    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
    if (MyGameInstance->SessionInterface.IsValid()) {
        MyGameInstance->SessionInterface-
>OnFindSessionsCompleteDelegates.AddUObject(this,
&UMPUserWidget::OnFindSessionsComplete);
    }
}
```

我们绑定OnBttn_SearchGameClick函数到OnClicked事件上，当事件触发就会调用这个OnBttn_SearchGameClick函数。我们将Border_ServerList设置为隐藏状态，当我们后续点击搜索的时候才会显示这个组件。接着我们对MyGameInstance->SessionInterface绑定搜索完毕后的OnFindSessionsComplete函数。

进行如下的OnBttn_SearchGameClick点击实现。

```
void UMPUserWidget::OnBttn_SearchGameClick()
{
    if (Border_ServerList&& ScrollBox_ServerList) {

        Border_ServerList->SetVisibility(ESlateVisibility::Visible);
```

```
        ScrollBox_ServerList->ClearChildren();

        ClearAllSeverinfo_WB();//这个函数我们后续会进行定义，用于清空
ServerRowUserWidget视图
        Text_SearchGameButton->SetText(FText::FromString(TEXT("Searching...")));
        Border_MenuButton->SetIsEnabled(false);

        MyGameInstance->SessionSearch = MakeShareable(new
FOnlineSessionSearch());
        MyGameInstance->SessionSearch->bIsLanQuery = MyGameInstance-
>bIsLanConnection;
        MyGameInstance->SessionSearch->MaxSearchResults = 10;
        MyGameInstance->SessionSearch->QuerySettings.Set("SEARCH_PRESENCE",
true, EOnlineComparisonOp::Equals);

        MyGameInstance->SessionInterface->FindSessions(0, MyGameInstance-
>SessionSearch.ToSharedRef());

    }
}
```

我们会先将Border_ServerList设置为可见状态，清空这个框的所有子元素（假如之前点击过Search那么这里面会有之前的元素，需要删去），然后我们用MakeShareable创建一个会话搜索对象。然后bIsLanQuery设置当前网络的搜索范围（局域网或在线模式）。设置搜索返回的最大结果数量。搜索条件过滤器QuerySettings.Set，第一个参数SEARCH_PRESENCE是一个预定义的关键字，用于筛选支持在线状态的会话，第二个参数表示指定与关键字对应的值，这里SEARCH_PRESENCE的值就是true，第三个参数Equals表示比较运算符，用于判断关键字的值是否匹配搜索条件。

我们需要一个数组来存放我们的搜索结果列表。在MPUserWidget.h中进行如下的声明

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Widgets")
TArray<UServerRowUserWidget*> AllServerinfo_WB;
UFUNCTION()
void ClearAllSeverinfo_WB();
```

然后，我们实现我们的ClearAllSeverinfo_WB函数。

```
void UMPUserWidget::ClearAllSeverinfo_WB()
{
    //清空AllServerinfo_WB
    for (UServerRowUserWidget* Widget : AllServerinfo_WB)
    {
        if (Widget)
        {
            Widget->RemoveFromParent();  // 如果它们已添加到视图中，先移除它们
            Widget->ConditionalBeginDestroy();  // 销毁该 Widget（可选，取决于你是否
需要清理内存）
        }
    }
    AllServerinfo_WB.Empty();
}
```

我们需要考虑到，当用户点击进入这个多人游戏菜单界面的时候，我们应该NativeConstruct进行一些设置。

```
void UMPUserWidget::NativeConstruct()
{
    Super::NativeConstruct();
    SetConnectionType_Text(MyGameInstance->bIsLanConnection);

    ScrollBox_ServerList->ClearChildren();
    ClearAllSeverinfo_WB();
    Text_SearchGameButton->SetText(FText::FromString(TEXT("Searching Game")));
    Border_MenuButton->SetIsEnabled(true);
    Border_ServerList->SetIsEnabled(true);
    Border_ServerList->SetVisibility(ESlateVisibility::Hidden);
}
```

我们会先将ScrollBox_ServerList的子元素设置为空，并且清楚视图元素和搜索结果列表，修改页面为可点击，并且隐藏Border_ServerList这个元素。

接下来，我们应该实现我们搜索到结果之后应该做什么，实现在我们之前的已经声明过的OnFindSessionsComplete函数中。

```
void UMPUserWidget::OnFindSessionsComplete(bool bWassuccessful)
{
    Text_SearchGameButton->SetText(FText::FromString(TEXT("Search Game")));
    Border_MenuButton->SetIsEnabled(true);
    if (bWassuccessful) {
        if (!MyGameInstance->SessionSearch.IsValid()) {
            UE_LOG(LogTemp, Error, TEXT("Search is nullptr"));
            return;
        }

        if (!MyGameInstance->SessionSearch->SearchResults.Num()) {
            UE_LOG(LogTemp, Error, TEXT("FindSessions found 0 results"));
        }
        else {
            if (!MyGameInstance->ServerRowUserWidgetClass)
            {
                UE_LOG(LogTemp, Error, TEXT("ServerRowUserWidgetClass is not
set"));
                return;
            }
            for (FOnlineSessionSearchResult& Result : MyGameInstance-
>SessionSearch->SearchResults) {
                UServerRowUserWidget* ServerRow =
CreateWidget<UServerRowUserWidget>(GetWorld(), MyGameInstance-
>ServerRowUserWidgetClass);
                ServerRow->WB_MpMenu = this;
                ServerRow->TriggerUpdateServerRowInfo(Result);
                AllServerinfo_WB.Add(ServerRow);
                ScrollBox_ServerList->AddChild(ServerRow);
            }
        }
    }
}
```

我们会将Border_MenuButton设置为可点击状态，如果搜索成功并且搜索结果大于0，那么我们会遍历搜索结果的列表，为每一个Result创建一个UServerRowUserWidget视图，然后设置视图的更新和添加到当前列表中。

1.3.8 总结

至此，我们完成了Session的创建，搜索和加入的功能。我们在UE编辑器中，我们可以这样运行，并进行查找，如下图1.3.8.1所示。

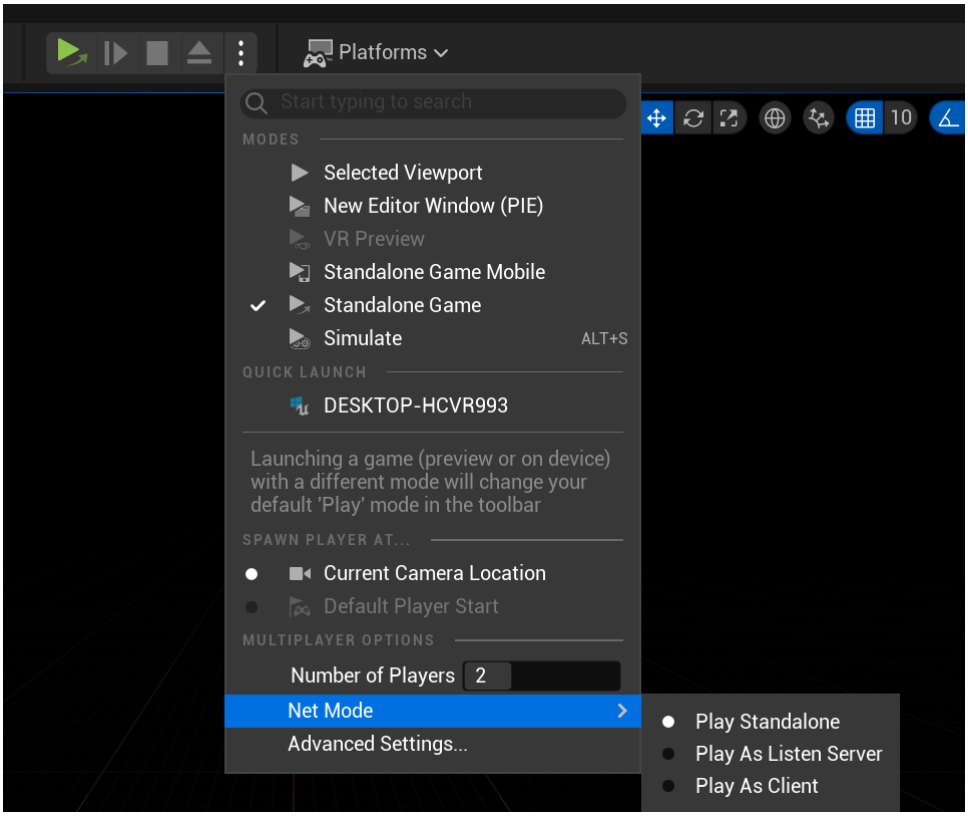

图1.3.8.1 运行游戏

我们运行两个进程后就会如图1.3.8.2所示，成功搜索到会话。

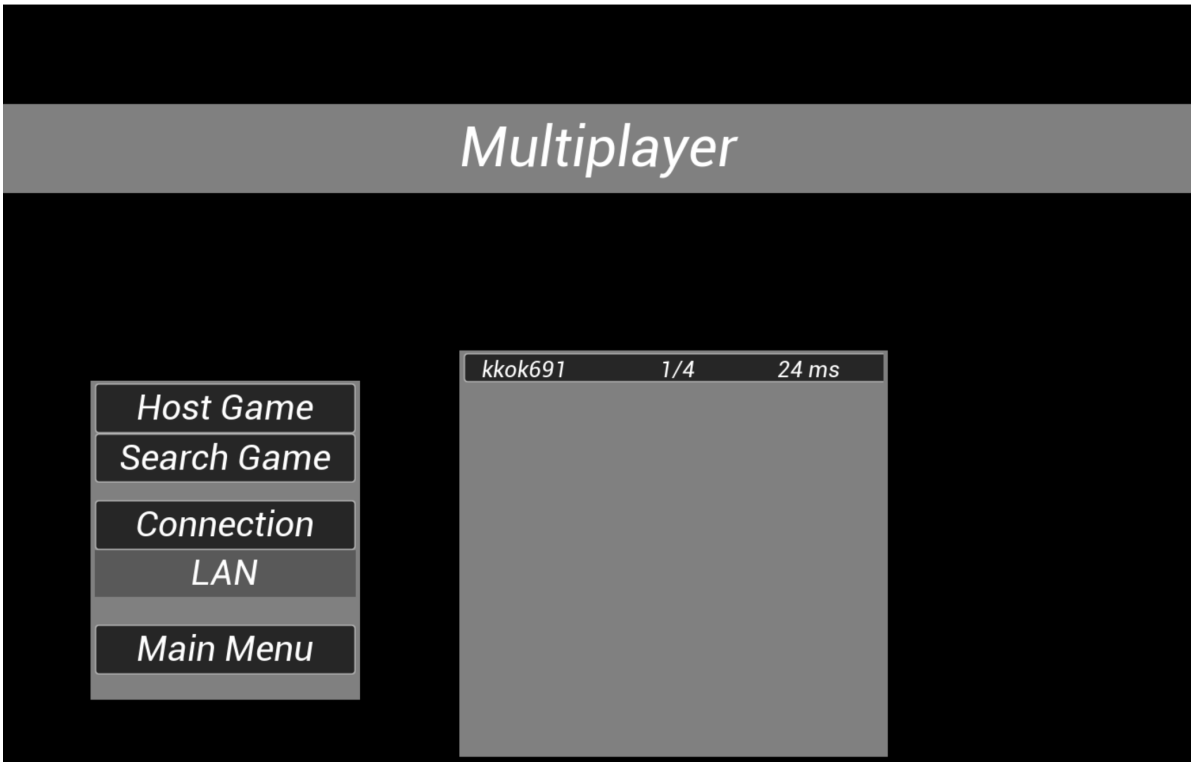

图1.3.8.2 搜索会话