UnLua-UE4下的Lua脚本插件

罗谦

腾讯互娱研发效能部

大纲

- 概述
- 特性
- 引擎和Lua绑定
- Lua访问引擎
- 引擎访问Lua
- 覆写是怎样炼成的
- 优化
- 智能语法提示
- 调试

概述

- 脚本插件
- 它不是蓝图的替代者，而是一种补充
  - 没有Asset的预览
  - 不支持nativization
  - 无法保持内容的引用
  - …
- 但是, 它为用Lua写游戏逻辑提供了支持

特性

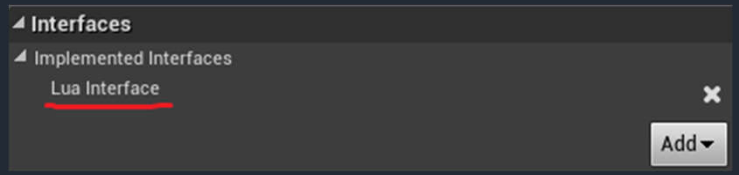
- 无需胶水代码访问**UCLASS**, **UPROPETY**, **UFUNCTION**, **USTRUCT**, **UENUM**
- 无辅助代码覆写(Override) **'BlueprintEvent'**
- 无辅助代码覆写(Override) **AnimNotify**
- 无辅助代码覆写(Override) **RepNotify**
- 无辅助代码覆写(Override) **Input Event**
- 简单完备的静态导出方案
- 高度优化的**UFUNCTION**调用
- 高度优化的容器(**TArray**, **TSet**, **TMap**)访问
- 高度优化的结构体访问
- 支持**UFUNCTION**(带**'BlueprintCallable'**或**'Exec'**标签)默认参数
- 支持自定义的碰撞检测相关枚举
- 支持编辑器内**Server/Client**模拟
- 支持Lua协程中执行**Latent**函数，同步写法完成异步逻辑
- 支持根据**Blueprint**类型自动生成Lua模板代码

引擎和Lua绑定

- 统一且简单的静态绑定方式
  - 仅一个Interface和一个纯虚函数
  - 同时支持蓝图类和C++类

```
UCLASS(config=Game)
class ATestProjectCharacter : public ACharacter, public ILuaInterface
```

```
FString ATestProjectCharacter::GetModuleName_Implementation() const
{
    return TEXT("TestProjectCharacter");
}
```



引擎和Lua绑定

- 动态绑定
  - 动态创建的Object
  - 动态生成的Actor

```lua
function BP_PlayerCharacter_C:SpawnWeapon()
    local WeaponClass = UClass.Load("/Game/Core/Blueprints/Weapon/BP_DefaultWeapon.BP_DefaultWeapon")
    local NewWeapon = GWorld:SpawnActor(WeaponClass, self:GetTransform(), ESpawnActorCollisionHandlingMethod.AlwaysSpawn, self, self, "Weapon.BP_DefaultWeapon_C")
    return NewWeapon
end
```

```lua
local MyObjectClass = UClass.Load("/Game/Core/Blueprints/BP_MyObject.BP_MyObject")
local MyObject = NewObject(MyObjectClass, self, nil, "BP_MyObject_C")
```

Lua访问引擎

- 无巨量的胶水代码!
- 支持反射体系内的所有数据
- 手动导出一些最基础的函数
  - UObject.Load, UObject.GetName, UObject.GetClass, etc.
  - UClass.Load, UClass.IsChildOf
  - UWorld.SpawnActor
- 手动导出数学库
  - FVector, FVector2D, FVector4, FQuat, FRotator, FTransform, etc.

Lua访问引擎

- 访问UClass

```
local Widget = UWidgetBlueprintLibrary.Create(self, UClass.Load("/Game/Core/UI/UMG_Main"))
```

Lua访问引擎

- 访问UStruct

```
local Position = FVector()
```

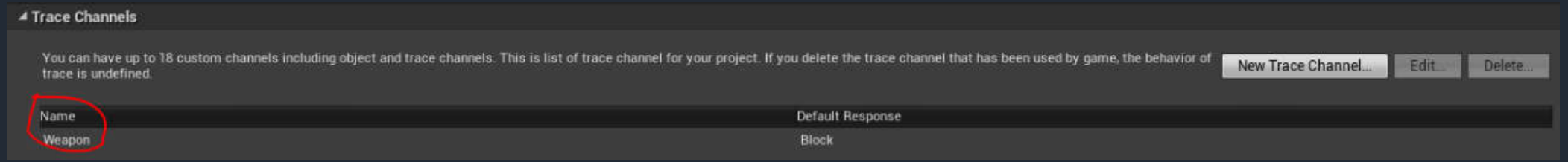
Lua访问引擎

- 访问UEnum

```
Weapon:K2_AttachToComponent(Point, nil, EAttachmentRule.SnapToTarget, EAttachmentRule.SnapToTarget, EAttachmentRule.SnapToTarget)
```

Lua访问引擎

- 访问自定义的碰撞检测相关UEnum




```
local bHit = UKismetSystemLibrary.LineTraceSingle(self, Start, End, ETraceTypeQuery.Weapon, false, nil, EDrawDebugTrace.None, HitResult, true)
```

Lua访问引擎

- 访问UProperty

```
local Position = FVector()
Position.X = 256.0
```

Lua访问引擎

- 访问Delegate/Multi-cast Delegate/Sparse Delegate

```
FloatTrack.InterpFunc:Bind(self, BP_PlayerCharacter_C.OnZoomInOutUpdate)
FloatTrack.InterpFunc:Unbind()
FloatTrack.InterpFunc:Execute(0.5)
```

```
self.ExitButton.OnClicked:Add(self, UMG_Main_C.OnClicked_ExitButton)
self.ExitButton.OnClicked:Remove(self, UMG_Main_C.OnClicked_ExitButton)
self.ExitButton.OnClicked:Clear()
self.ExitButton.OnClicked:Broadcast()
```

Lua访问引擎

- 访问UFunction

```
Widget:PlayAnimation(Anim, 0.0, 1, EUMGSequencePlayMode::Forward, 1.0)
```

```
Widget:PlayAnimation(Anim)
```

Lua访问引擎

- UFunction非常量引用参数处理(原子类型)

```
UFUNCTION()
void GetPlayerBaseInfo(int32 &Level, float &Health, FString &Name) const;
```

```
local Level, Health, Name = self:GetPlayerBaseInfo(9, 99, "Marcus")
```

```
local Level, Health, Name = self:GetPlayerBaseInfo()
```

Lua访问引擎

- UFunction非常量引用参数处理(非原子类型)

```
UFUNCTION()
void GetHitResult(FHitResult &HitResult) const;
```

```
local HitResult = FHitResult()
self:GetHitResult(HitResult)
```

```
local HitResult = self:GetHitResult()
```

Lua访问引擎

- UFunction返回值参数处理(原子类型)

```
UFUNCTION()
float GetMeleeDamage() const;
```

```
local MeleeDamage = self:GetMeleeDamage()
```

Lua访问引擎

- UFunction返回值参数处理(非原子类型)

```
UFUNCTION()
FVector GetCurrentLocation() const;
```

```
local Location = self:GetCurrentLocation()
```

```
local Location = FVector()
self:GetCurrentLocation(Location)
```

```
local Location = FVector()
local LocationCopy = self:GetCurrentLocation(Location)
```

Lua访问引擎

- 访问Latent Function

```
UFUNCTION(BlueprintCallable, Category="Utilities|FlowControl", meta=(Latent, WorldContext="WorldContextObject"
static void Delay(UObject* WorldContextObject, float Duration, struct FLatentActionInfo LatentInfo );
```

```
coroutine.resume(coroutine.create(function(GameMode, Duration) UKismetSystemLibrary.Delay(GameMode, Duration) end), self, 5.0)
```

Lua访问引擎

- 访问基础容器 (TArray, TMap, TSet)

```
local Indices = TArray(0)
Indices:Add(1)
Indices:Add(3)
Indices:Remove(0)
local NbIndices = Indices:Length()
```

```
local Vertices = TArray(FVector)
local Actors = TArray(AActor)
```

Lua访问引擎

- 静态导出反射体系外的数据

```
struct Vec3
{
        Vec3() : x(0), y(0), z(0) {}
        Vec3(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}

        void Set(const Vec3 &V) { *this = V; }
        Vec3& Get() { return *this; }
        void Get(Vec3 &V) const { V = *this; }

        bool operator==(const Vec3 &V) const { return x == V.x && y == V.y && z == V.z; }

        static Vec3 Cross(const Vec3 &A, const Vec3 &B) { return Vec3(A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y - A.y * B.x); }
        static Vec3 Multiply(const Vec3 &A, float B) { return Vec3(A.x * B, A.y * B, A.z * B); }
        static Vec3 Multiply(const Vec3 &A, const Vec3 &B) { return Vec3(A.x * B.x, A.y * B.y, A.z * B.z); }

        float x, y, z;
};
```

```
BEGIN_EXPORT_CLASS(Vec3, float, float, float)
        ADD_PROPERTY(x)
        ADD_PROPERTY(y)
        ADD_PROPERTY(z)
        ADD_FUNCTION(Set)
        ADD_NAMED_FUNCTION("Equals", operator==)
        ADD_FUNCTION_EX("Get", Vec3&, Get)
        ADD_CONST_FUNCTION_EX("GetCopy", void, Get, Vec3&)
        ADD_STATIC_FUNCTION(Cross)
        ADD_STATIC_FUNCTION_EX("MulScalar", Vec3, Multiply, const Vec3&, float)
        ADD_STATIC_FUNCTION_EX("MulVec", Vec3, Multiply, const Vec3&, const Vec3&)
END_EXPORT_CLASS()
IMPLEMENT_EXPORTED_CLASS(Vec3)
```

引擎访问Lua

- 覆写(Override)所有 ‘BlueprintEvent’
  - 用‘BlueprintImplementableEvent’标记的UFunction
  - 用‘BlueprintNativeEvent’标记的UFunction
  - 所有蓝图中定义的events/functions

引擎访问Lua

- 覆写‘BlueprintEvent’
  - 覆写不带返回值的‘BlueprintEvent’

```
UFUNCTION(BlueprintImplementableEvent, meta=(DisplayName = "BeginPlay"))
void ReceiveBeginPlay();
```

```
function BP_PlayerController_C:ReceiveBeginPlay()
  print("ReceiveBeginPlay in Lua!")
end
```

  - 覆写带返回值的‘BlueprintEvent’

```
UFUNCTION(BlueprintImplementableEvent, Category="Character")
bool GetCharacterInfo(int32 &HP, FVector &Position, FString &Name);
```

```
function BP_PlayerCharacter_C:GetCharacterInfo(HP, Position, Name)
        Position.X = 128.0
        Position.Y = 128.0
        Position.Z = 0.0
        return 99, nil, "Marcus", true
end
```

引擎访问Lua

- 覆写AnimNotify


```
function ABP_PlayerCharacter_C:AnimNotify_NotifyPhysics()
        UBPI_Interfaces_C.ChangeToRagdoll(self.Pawn)
end
```

引擎访问Lua

- 覆写RepNotify

```
UFUNCTION()
virtual void OnRep_Health();

UPROPERTY(ReplicatedUsing= OnRep_Health)
int32 Health;
```

```
function BP_PlayerCharacter_C:OnRep_Health(...)
        print("call OnRep_Health in Lua")
end
```

引擎访问Lua

- 覆写Input Events
  - Axis Inputs

```
function BP_PlayerController_C:Turn(AxisValue)
        self:AddYawInput(AxisValue)
end
```

  - Action Inputs

```
function BP_PlayerController_C:Aim_Pressed()
        UBPI_Interfaces_C.UpdateAiming(self.Pawn, true)
end

function BP_PlayerController_C:Aim_Released()
        UBPI_Interfaces_C.UpdateAiming(self.Pawn, false)
end
```

引擎访问Lua

- 覆写Input Events
  - Key Inputs

```
function BP_PlayerController_C:P_Pressed()
        print("P_Pressed")
end

function BP_PlayerController_C:P_Released()
        print("P_Released")
end
```

  - Touch/AxisKey/VectorAxis/Gesture Inputs

覆写是怎样炼成的

- Thunk函数的“颜”

```
class COREUOBJECT_API UFunction : public UStruct
{

private:
    /** C++ function this is bound to */
    FNativeFuncPtr Func;

public:
    /* ... */
    FORCEINLINE FNativeFuncPtr GetNativeFunc() const { ... }

    /**
     * Sets the native func pointer.
     *
     * @param InFunc - The new function pointer.
     */
    FORCEINLINE void SetNativeFunc(FNativeFuncPtr InFunc)
    {
        Func = InFunc;
    }
```

```
typedef void (*FNativeFuncPtr)(UObject* Context, FFrame& TheStack, RESULT_DECL);
```

```
typedef void (UObject::*Native)(FFrame& TheStack, RESULT_DECL);
```

覆写是怎样炼成的

- Thunk函数的调用

```
void UObject::ProcessEvent( UFunction* Function, void* Parms )
{
#if 1 Active Preprocessor Block
#endif
    // Scope required for scoped script stats.
    {
#if 1 Active Preprocessor Block
#endif
        // Call native function or UObject::ProcessInternal.
        const bool bHasReturnParam = Function->ReturnValueOffset != MAX_uint16;
        uint8* ReturnValueAddress = bHasReturnParam ? ((uint8*)Parms + Function->ReturnValueOffset) : nullptr;
        Function->Invoke(this, NewStack, ReturnValueAddress);
```

```
void UFunction::Invoke(UObject* Obj, FFrame& Stack, RESULT_DECL)
{
    checkSlow(Func);

    UClass* OuterClass = (UClass*)GetOuter();
    if (OuterClass->IsChildOf(UInterface::StaticClass())) { ... }

    TGuardValue<UFunction*> NativeFuncGuard(Stack.CurrentNativeFunction, this);
    return (*Func)(Obj, Stack, RESULT_PARAM);  ≤ 1ms elapsed
}
```

覆写是怎样炼成的

- Thunk函数替换

```
class COREUOBJECT_API UFunction : public UStruct
{

private:
    /** C++ function this is bound to */
    FNativeFuncPtr Func;

public:
    /* ... */
    FORCEINLINE FNativeFuncPtr GetNativeFunc() const { ... }

    /**
     * Sets the native func pointer.
     *
     * @param InFunc - The new function pointer.
     */
    FORCEINLINE void SetNativeFunc(FNativeFuncPtr InFunc)
    {
        Func = InFunc;
    }
```

覆写是怎样炼成的

- Thunk函数替换不适合所有场景

```
void UObject::CallFunction( FFrame& Stack, RESULT_DECL, UFunction* Function )
{
#if PER_FUNCTION_SCRIPT_STATS Active Preprocessor Block
#endif // PER_FUNCTION_SCRIPT_STATS

#if STATS || ENABLE_STATNAMEDEVENTS Active Preprocessor Block
#endif

    checkSlow(Function);

    if (Function->FunctionFlags & FUNC_Native) { ... }
    else
    {
        ProcessScriptFunction(this, Function, Stack, RESULT_PARAM, ProcessInternal);
    }
}
```

覆写是怎样炼成的

- 新Opcode注册

```
enum
{
    EX_CallLua = EX_Max - 1
};
```

```
extern uint8 GRegisterNative(int32 NativeBytecodeIndex, const FNativeFuncPtr& Func);
static FNativeFunctionRegistrar CallLuaRegistrar(UObject::StaticClass(), "execCallLua", (FNativeFuncPtr)&FLuaInvoker::execCallLua);
static uint8 CallLuaBytecode = GRegisterNative(EX_CallLua, (FNativeFuncPtr)&FLuaInvoker::execCallLua);
```

覆写是怎样炼成的

- Opcode注入

```
Function->Script.Add(EX_CallLua);
int32 Index = Function->Script.AddZeroed(sizeof(Userdata));
FMemory::Memcpy(Function->Script.GetData() + Index, &Userdata, sizeof(Userdata));
Function->Script.Add(EX_Return);
Function->Script.Add(EX_Nothing);
```

覆写是怎样炼成的

- 返回值处理

```
struct FFrame : public FOutputDevice
{
public:
    // Variables.
    UFunction* Node;
    UObject* Object;
    uint8* Code;
    uint8* Locals;

    UProperty* MostRecentProperty;
    uint8* MostRecentPropertyAddress;

    /** The execution flow stack for compiled Kismet code */
    FlowStackType FlowStack;

    /** Previous frame on the stack */
    FFrame* PreviousFrame;

    /** contains information on any out parameters */
    FOutParmRec* OutParms;
```

```
#define DEFINE_FUNCTION(func) void func( UObject* Context, FFrame& Stack, RESULT_DECL )
```

```
#define RESULT_PARAM Z_Param__Result
#define RESULT_DECL void*const RESULT_PARAM
```

优化

- 结构体访问优化
  - 结构体创建

```
local Vec = FVector(1.0, 2.0, 3.0)
```

  - 直观实现（伪代码）

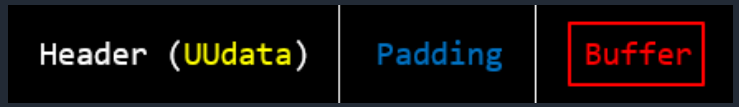
```
FVector *Vec = new FVector(1.0f, 2.0f, 3.0f);
void **Userdata = (void**)lua_newuserdata(L, sizeof(void*));
*Userdata = Vec;
```

  - UnLua实现（伪代码）

```
uint8 PaddingSize = CalcUserdataPadding(alignof(FVector));
void *Userdata = lua_newuserdata(L, sizeof(FVector) + PaddingSize);
FVector *Vec = new((uint8*)Userdata + PaddingSize) FVector(1.0f, 2.0f, 3.0f);
```

优化

- 结构体访问优化
  - 创建时节省一次内存分配
  - GC时节省一次内存释放
  - 缓存友好
  - 内存布局



优化

- UFunction调用优化
  - 持久化参数缓存

```
    // create persistent parameter buffer. memory for speed
#if ENABLE_PERSISTENT_PARAM_BUFFER
    Buffer = nullptr;
    if (InFunction->ParmsSize > 0)
    {
        Buffer = FMemory::Malloc(InFunction->ParmsSize, 16);
#if STATS  Active Preprocessor Block
#endif
    }
#endif
```

优化

- UFunction调用优化
  - 为Native Local函数返回值参数预分配缓存

```
        // pre-create OutParmRec for 'out' property
        FOutParmRec *Out = (FOutParmRec*)FMemory::Malloc(sizeof(FOutParmRec), alignof(FOutParmRec));
#if STATS  Active Preprocessor Block
#endif
        Out->PropAddr = Property->ContainerPtrToValuePtr<uint8>(Buffer);
        Out->Property = Property;
        if (CurrentOutParmRec)
        {
            CurrentOutParmRec->NextOutParm = Out;
            CurrentOutParmRec = Out;
        }
        else
        {
            OutParmRec = Out;
            CurrentOutParmRec = Out;
        }
```

  - 为Native Local函数提供快速调用路径

```
if (FinalFunction->HasAnyFunctionFlags(FUNC_Native))
{
    uint8* ReturnValueAddress = FinalFunction->ReturnValueOffset != MAX_uint16 ? (uint8*)Params + FinalFunction->ReturnValueOffset : nullptr;
    FFrame NewStack(Object, FinalFunction, Params, nullptr, Function->Children);
    NewStack.OutParms = OutParmRec;
    FinalFunction->Invoke(Object, NewStack, ReturnValueAddress);
}
```

优化

- 传参优化
  - UFunction带常量引用参数

```
UFUNCTION(BlueprintCallable)
void UpdatePositions(const TArray<FVector> &NewPositions);
```

  - 直观实现（伪代码）

```
void *Dest = Property->ContainerPtrToValuePtr(ParamsBuffer);
void *Src = GetParamPtr(L, ParamIndex);
ParamProperty->CopySingleValue(Dest, Src);
Object->ProcessEvent(Function, ParamsBuffer);
```

  - UnLua实现（伪代码）

```
void *Dest = Property->ContainerPtrToValuePtr(ParamsBuffer);
void *Src = GetParamPtr(L, ParamIndex);
FMemory::Memcpy(Dest, Src, ParamSize);     // ParamSize == sizeof(TArray)
Object->ProcessEvent(Function, ParamsBuffer);
```

优化

- 传参优化
  - 浅拷贝
  - 正确性保证
  - 对于复杂数据结构（例如含有大量元素的容器），浅拷贝的性能优势巨大

优化

- 非常量引用参数优化
  - UFunction带非常量引用参数

```
UFUNCTION(BlueprintCallable)
void GetPositions(TArray<FVector> &OutPositions) const;
```

  - 和C++类似的Lua调用方式

```
local Positions = TArray(FVector)
self:GetPositions(Positions)
```

  - 两次浅拷贝（传参和值返回各一次）

优化

- 返回值参数优化
  - UFunction返回非原子类型

```
UFUNCTION(BlueprintCallable)
const FVector& GetCOM() const;
```

  - 直观实现
    - 新创建Userdata并将其压入Lua栈顶
  - UnLua实现
    - 先创建Userdata并将其作为参数传入函数
    - 利用了传参优化
    - 多次调用（例如循环）情况下，避免了大量的Userdata创建和GC，性能优势明显

智能语法提示

- 符号信息（反射体系内）
  - 导出模块UnLuaIntelliSense
  - 和UHT一同工作
  - 符号信息位于ProjectDir/Plugins/UnLua/Intermediate/IntelliSense

```lua
---@class APointLight : ALight
---@field public PointLightComponent UPointLightComponent
local APointLight = {}

---@param NewLightFalloffExponent number
function APointLight:SetLightFalloffExponent(NewLightFalloffExponent) end

---BEGIN DEPRECATED (use component functions now in level script)
---@param NewRadius number
function APointLight:SetRadius(NewRadius) end

return APointLight
```

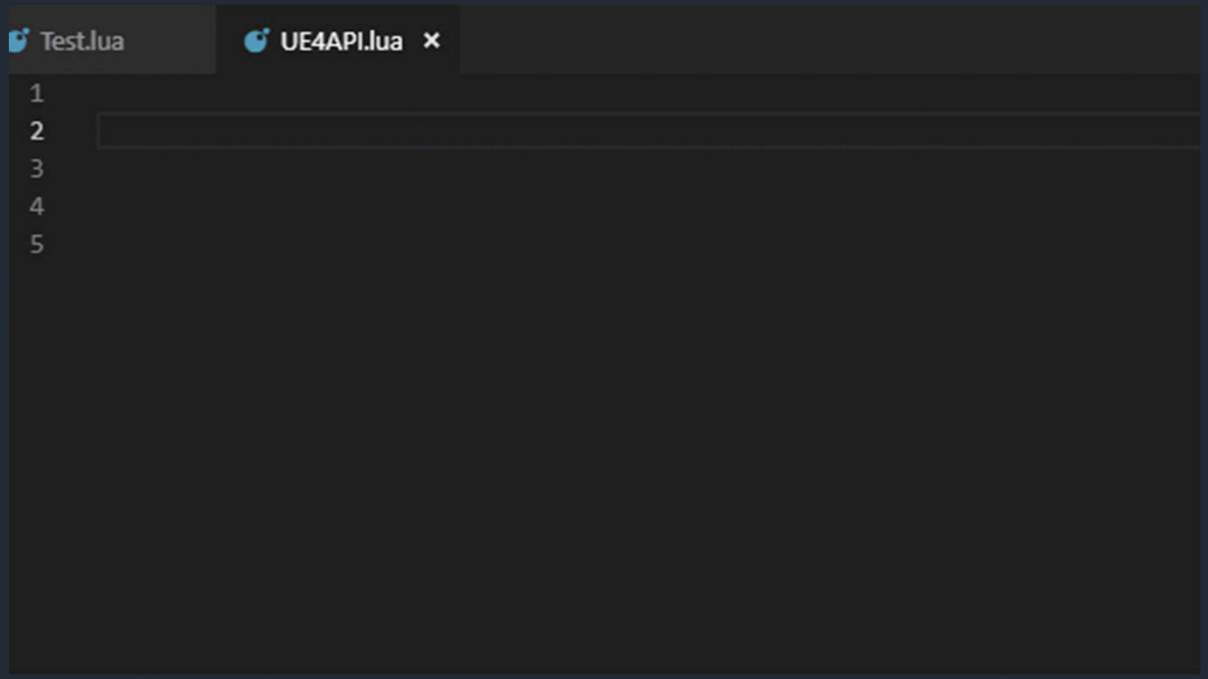
智能语法提示

- 符号信息（反射体系外）
  - UnLuaIntelliSenseCommandlet
  - 符号信息位于ProjectDir/Plugins/UnLua/Intermediate/IntelliSense/StaticallyExports

```lua
---@class Vec3
---@field public x number
---@field public y number
---@field public z number
local Vec3 = {}

---@param P0 number @[out]
---@param P1 number @[out]
---@param P2 number @[out]
function Vec3:GetXYZ(P0, P1, P2) end

---@param P0 number
---@param P1 number
---@param P2 number
function Vec3:SetXYZ(P0, P1, P2) end

---@param P0 Vec3
---@return boolean
function Vec3:Equals(P0) end

---@return string
function Vec3:ToString() end

return Vec3
```

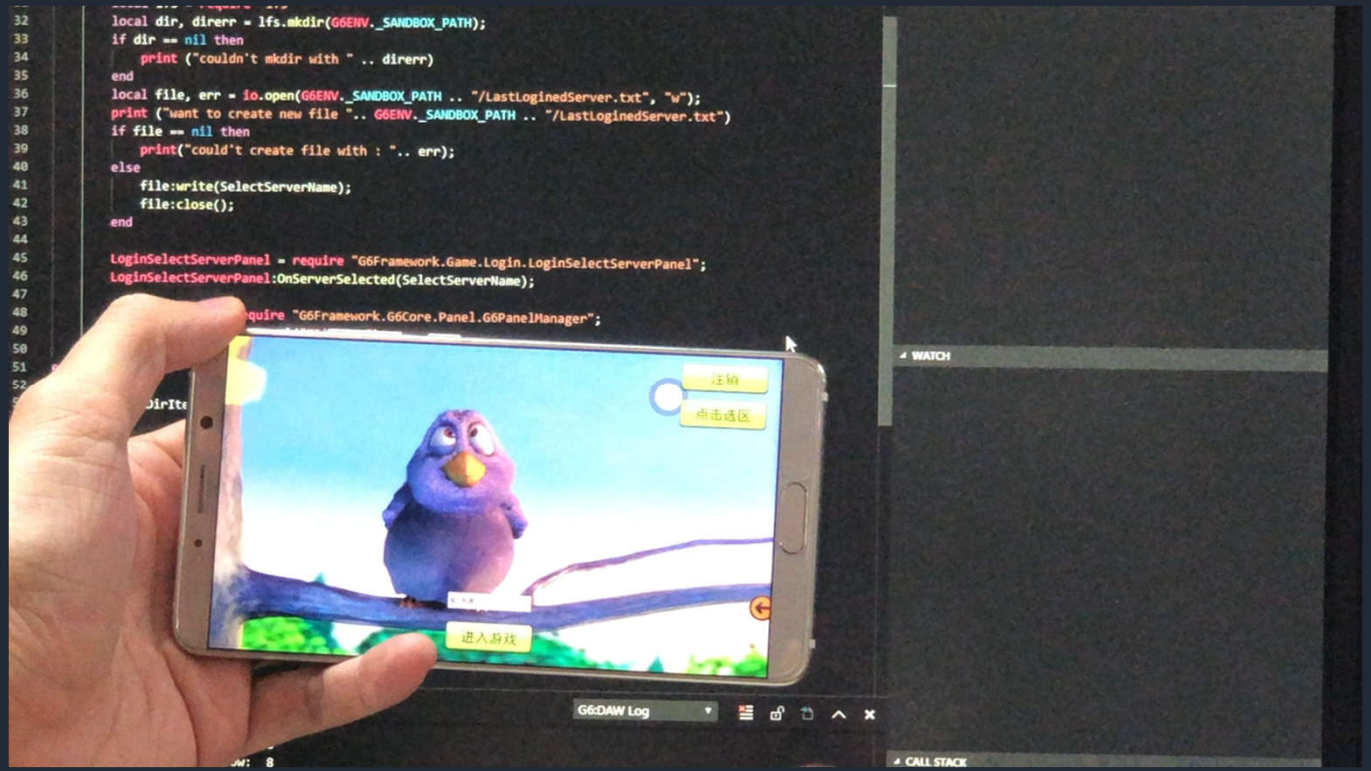
智能语法提示

- IDE



调试

- Debug Any Where



谢谢!

https://github.com/Tencent/UnLua