PuerTs框架简单梳理

希望可以理清下面几个问题：

- Ts和C++如何交互
- PuerTs如何做热更

1. 虚拟机启动时机

插件内部使用V8作为JS虚拟机的运行环境，虚拟机的启动流程从文档中已经写出：

```
1         JsEnv = MakeShared<puerts::FJsEnv>();
2         TArray<TPair<FString, UObject*>> Arguments;
3         Arguments.Add(TPair<FString, UObject*>(TEXT("GameInstance"), this)); //
4         JsEnv->Start("QuickStart", Arguments);
```

首先，PuerTs支持起多个虚拟机，虚拟机的启动是在构造函数里完成的：

```
    // 初始化Isolate和DefaultContext
    v8::V8::SetSnapshotDataBlob(SnapshotBlob.get());

    CreateParams.array_buffer_allocator = v8::ArrayBuffer::Allocator::NewDefaultAllocator();
#if WITH_QUICKJS
    MainIsolate = InExternalRuntime ? v8::Isolate::New(InExternalRuntime) : v8::Isolate::New(CreateParams);
#else  #if WITH_QUICKJS
    check(!InExternalRuntime && !InExternalContext);
    MainIsolate = v8::Isolate::New(CreateParams);
#endif  #if WITH_QUICKJS #else
    auto Isolate = MainIsolate;
#ifdef THREAD_SAFE
    v8::Locker Locker(Isolate);
    UserObjectRetainer.Isolate = Isolate;
    SysObjectRetainer.Isolate = Isolate;
#endif  #ifdef THREAD_SAFE
    Isolate->SetData(0, static_cast<IObjectMapper*>(this));    //直接传this会有问题，强转后地址会变
```

一个v8::Isolate代表了一个V8的完整的实例，拥有独立的栈堆。所以多个V8实例之间是相互隔离的。

在构造完一个新的Isolate之后，会初始化一个v8::Context，并注册一系列全局对象和全局方法供v8调用，从而实现ts通过v8访问到C++的对象，并调用到C++的方法。

- 启动一个v8上下文

```
388         v8::Local<v8::Context> Context = node::NewContext(Isolate);
389
390     #endif  #if !defined(WITH_NODEJS) #else
391
392         DefaultContext.Reset(Isolate, Context);
393
394         v8::Context::Scope ContextScope(Context);
395
```

- 往上下文中注册了许多全局方法

```cpp
v8::Local<v8::Object> Global = Context->Global();

v8::Local<v8::Object> PuertsObj = v8::Object::New(Isolate);
Global->Set(Context, key: FV8Utils::InternalString(Isolate, String:"puerts"), value: PuertsObj).Check();

auto This:Local<External> = v8::External::New(Isolate, value:this);

Global
    ->Set(Context, key: FV8Utils::ToV8String(Isolate, String:"__tgjsEvalScript"),
        value: v8::FunctionTemplate::New(
            Isolate,
            callback: [](const v8::FunctionCallbackInfo<v8::Value>& Info) ->void
            {
                auto Self = static_cast<FJsEnvImpl*>((v8::Local<v8::External>::Cast(that:Info.Data()))->Value());
                Self->EvalScript(Info);
            },
            This) // Local<FunctionTemplate>
            ->GetFunction(Context) // MaybeLocal<Function>
            .ToLocalChecked()) // Maybe<bool>
    .Check();

Global
    ->Set(Context, key: FV8Utils::ToV8String(Isolate, String:"__tgjsLog"),
        value: v8::FunctionTemplate::New(
            Isolate,
            callback: [](const v8::FunctionCallbackInfo<v8::Value>& Info) ->void
            {
                auto Self = static_cast<FJsEnvImpl*>((v8::Local<v8::External>::Cast(that:Info.Data()))->Value());
```

在 `JsEnv->Start` 中，可以看到启动虚拟机的流程是往v8的上下文中放入一些全局参数，最后启动一个入口脚本。

```cpp
void FJsEnvImpl::Start(const FString& ModuleName, const TArray<TPair<FString, UObject*>>& Arguments)
{
#ifdef SINGLE_THREAD_VERIFY
    ensureMsgf(BoundThreadId == FPlatformTLS::GetCurrentThreadId(), TEXT("Access by illegal thread!"));
#endif  #ifdef SINGLE_THREAD_VERIFY
    if (Started)
    {
        Logger->Error( Message:"Started yet!");
        return;
    }

    auto Isolate = MainIsolate;
#ifdef THREAD_SAFE
    v8::Locker Locker(Isolate);
#endif  #ifdef THREAD_SAFE
    v8::Isolate::Scope IsolateScope(Isolate);
    v8::HandleScope HandleScope(Isolate);
    auto Context:Local<Context> = v8::Local<v8::Context>::New(Isolate, DefaultContext);
    v8::Context::Scope ContextScope(Context);
```

```cpp
    for (int i = 0; i < Arguments.Num(); i++)
    {
        auto Object = Arguments[i].Value;
        v8::Local<v8::Value> Args[2] = {
            FV8Utils::ToV8String(Isolate, Arguments[i].Key), FindOrAdd(Isolate, [&]Context, Object->GetClass(), Ob
        auto Result:MaybeLocal<Value> = ArgvAdd->Call(Context, Argv, argc:2, Args);
    }

    ExecuteModule(ModuleName,
        Preprocessor: [](const FString& Script, const FString& Path) ->FString
        {
            auto PathInJs:FString = Path.Replace( From:TEXT("\\"), To:TEXT("\\\\"));
            auto DirInJs:FString = FPaths::GetPath(Path).Replace( From:TEXT("\\"), To:TEXT("\\\\"));
            return FString::Printf( Fmt:TEXT("(function() { var __filename = '%s', __dirname = '%s', exports ={}, m
                                    "exports, filename : __filename }; (function (exports, require, console,
                                    "%s\n})(exports, puerts.genRequire('%s'), puerts.console);})()"),
                *PathInJs, *DirInJs, *Script, *DirInJs);
        });
    Started = true;
}
```

2. 对象持有

首先Ts侧的方法NewObject：

```
let Widget = UE.NewObject(UE.Button.StaticClass(), this.WidgetTree.RootWidget) as UE.Button;
```

查阅C++侧对应方法：

```
Global
    ->Set(Context, key: FV8Utils::ToV8String(Isolate, String:"__tgjsNewObject"),
        value: v8::FunctionTemplate::New(
            Isolate,
            callback: [](const v8::FunctionCallbackInfo<v8::Value>& Info)->void
            {
                auto Self = static_cast<FJsEnvImpl*>((v8::Local<v8::External>::Cast(that:Info.Data()))->Value());
                Self->NewObjectByClass(Info);
            },
            This)// Local<FunctionTemplate>
            ->GetFunction(Context)// MaybeLocal<Function>
            .ToLocalChecked())// Maybe<bool>
    .Check();
```

可以看到，实现其实放在`NewObjectByClass`之中，深入进去可以看到内部的实现其实是先根据第一个参数加载了一个UClass*，然后构造一个UObject之后将其放入虚拟机的上下文中：

```
    UClass* Class = Cast<UClass>(Src:FV8Utils::GetUObject([&]Context, Info[0]));   Class: 0x0000014b90e39300 (Name=DroneMovementComponent)

    if (Class)
    {
        if (Info.Length() > 1)
        {
            Outer = FV8Utils::GetUObject([&]Context, Info[1]);
        }
        if (Info.Length() > 2)
        {
            Name = FName(*FV8Utils::ToFString(Isolate, Info[2]));
        }
        if (Info.Length() > 3)
        {
            ObjectFlags = (EObjectFlags) (Info[3]->Int32Value(Context).ToChecked());
        }
        UObject* Object = NewObject<UObject>(Outer, Class, Name, ObjectFlags);   Object: UObject *   Outer: 0x0000014d4fd84b80 (Name=TS_DroneCine

        auto Result:Local<Value> = FV8Utils::IsolateData<IObjectMapper>(Isolate)->FindOrAdd(Isolate, [&]Context, Object->GetClass(), Object);   Re
        Info.GetReturnValue().Set(Result);   Result: {val_=0x0000014babe36ab0 {}}   Info: {implicit_arg_=0x0000001d9c1727b8 {1491629930625}, val
    }
```

构造UObject的函数放在`FindOrAdd`中，在这个函数中会首先遍历UE对象在虚拟机中的指针是否被缓存住，如果缓存住说明虚拟机内部已经存在这个对象了，直接返回虚拟机中的指针即可；如果没有被C++缓存，则在虚拟机中构造一个新的对象。

```
auto PersistentValuePtr:Global<Value>* = ObjectMap.Find(UEObject);
if (!PersistentValuePtr)    // create and link
{
    auto PersistentValuePtr2:Global<Value>* = GeneratedObjectMap.Find(UEObject);
    if (PersistentValuePtr2)    // TODO: 后续尝试改为新建一个对象，这个对象持有UObject的引用，并且把调用转发到Iter2->second
    {
        return v8::Local<v8::Value>::New(Isolate, *PersistentValuePtr2);
    }
    auto BindTo:Local<External> = v8::External::New(Context->GetIsolate(), UEObject);
    v8::Handle<v8::Value> Args[] = {BindTo};
    return GetJsClass(Class, Context)->NewInstance(Context, argc:1, Args).ToLocalChecked();
}
else
```

如果是构造一个新的对象，那么新的对象被ts层持有后，应当也加入ObjectMap中，这里好像没有看到加入的时机？

3. Ts侧的脚本绑定

Ts侧如何和UE侧交互？在阅读相关源码后，希望可以解答以下问题：

- Ts如何通过 `UE.Gameplaystatics.xxx` 等类似的方法，调用到UE侧的静态方法
- Ts如何访问到对象身上的属性、方法
- 继承引擎类功能是如何实现的
- Mixin是如何实现的

LoadUEType函数

首先关注到的是UE侧的 `LoadUEType` 函数，当TS侧调用UE.xxx时，或者调用StaticClass()时都会调用到UE侧的这个函数。

在 `uelazyload.js` 中，可以看到存在一个缓存表，将所有加载完毕的UE类型存下来。如果所需的类型不在缓存表中，则调用这个函数去获取：

```
var global = global || (function () { return this; }());
(function (global) {
    "use strict";

    let loadUEType = global.__tgjsLoadUEType;
    global.__tgjsLoadUEType = undefined;

    let loadCDataType = global.__tgjsLoadCDataType;
    global.__tgjsLoadCDataType = undefined;

    let cache = Object.create(null);

    let UE = new Proxy(cache, {
        get: function(classWrapers, name) {
            if (!(name in classWrapers)) {
                classWrapers[name] = loadUEType(name);
            }
            return classWrapers[name];
        }
    });
```

简单推断：这个函数的功能是将UE侧的一个类型Wrap到Js侧，访问绝大部分UE类型时，都需要这个函数的协助。

测试

下面是一段测试用的Ts代码，首先通过继承引擎类方式，在Ts侧定义了一个Gamemode，并重写了它的RecieveBeginPlay方法：

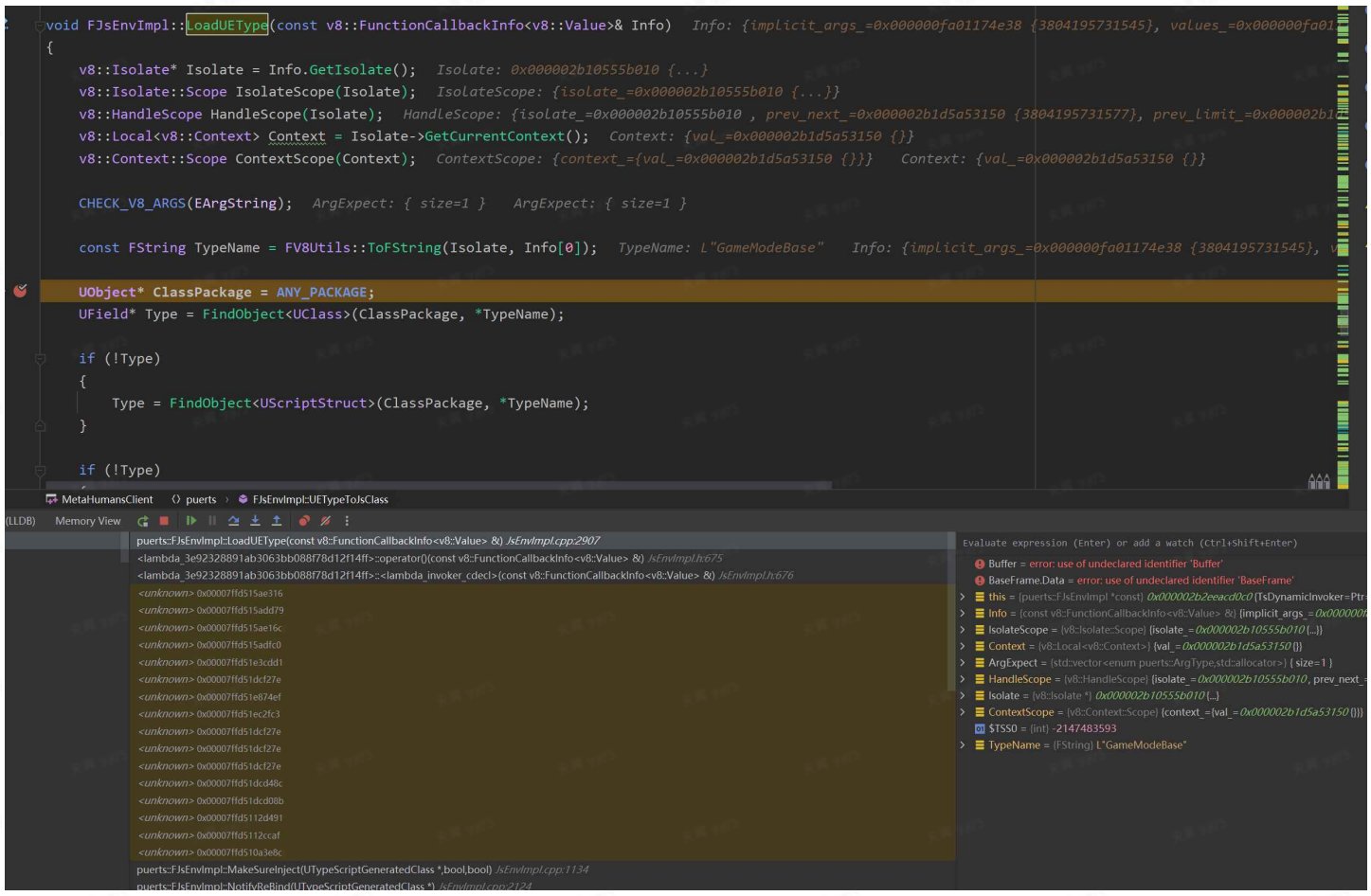
```
class TestGamemode extends UE.GameModeBase {
    ReceiveBeginPlay(): void {
        let actor = UE.GameplayStatics.GetActorOfClass(this.GetWorld(), UE.CppAc
        let guid = actor.ActorGuid;
        actor.PrintText();
        console.error("Get actor guid: " + guid);
```

```
    }
};

export default TestGamemode;
```

在Beginplay中，获取场景中类型为 `CppActor` 的Actor，访问了它身上的属性guid，调用了它身上的方法 `PrintText()`

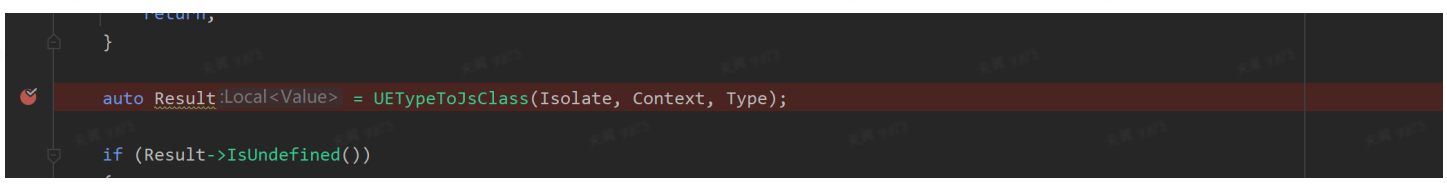
我们跟一下LoadUEType函数的断点：



其中这个函数被调用了三次，对应的类型名分别是： `GameModeBase` 、 `GameplayStatics` 和 `CppActor`

后两次应该都是Ts侧代码发起的调用，GameModeBase的类型加载应该是继承引擎类功能发起的，后面可以跟一下。

我们跟一下第二次，即加载GameplayStatics类型的时候，发生了什么。这个函数主要调用了 `UETypeToJsClass` ：



而 `UETypeToJsClass` 主要是将UE的类型作为Js对象返回，主要调用了里面的 `GetJsClass` 函数

```
v8::Local<v8::Function> FJsEnvImpl::GetJsClass(UStruct* InStruct, v8::Local<v8::
{
    bool Existed;
    auto Ret = GetTemplateOfClass(InStruct, Existed)->GetFunction(Context).ToLoc

    if (UNLIKELY(!Existed))    // first create
    {
        auto Class = Cast<UClass>(InStruct);
```

```
#if !defined(ENGINE_INDEPENDENT_JSENV)
        if (Class && !Class->IsNative() && !InStruct->IsA<UTypeScriptGeneratedCl
        {
            auto SuperClass = Cast<UTypeScriptGeneratedClass>(Class->GetSuperCla
            if (SuperClass)
            {
                MakeSureInject(SuperClass, false, false);
                v8::Local<v8::Value> VProto;
                if (Ret->Get(Context, FV8Utils::ToV8String(MainIsolate, "prototy
                {
                    v8::Local<v8::Object> Proto = VProto.As<v8::Object>();
                    __USE(Proto->SetPrototype(Context, BindInfoMap[SuperClass].P
                }
            }
        }
#endif
    }

    return Ret;
}
```

在这里又调用了`GetTemplateOfClass`，这个函数将UE的类型`InStruct`作为虚拟机中的`v8::FunctionTemplate`返回。根据注释的介绍，`FunctionTemplate`的作用主要是将C++中的函数包裹，用于在Js侧调用。所以LoadUEType最终会将UE的类型信息包装成一个FunctionTemplate返回给Js。

跟进去看一下这个函数做了什么：

首先它做了一个缓存，已经生成过的类型可以直接返回，如果该类型是第一次生成，那么主要会做三件事情:

1. 初始化一个该类型的`StructWrapper`，作用是收集该类型身上的反射信息，这一步也会做缓存。
2. 调用`StructWrapper`的`ToFunctionTemplate`方法，将类型的反射信息封装为`FunctionTemplate`。其中最关键的方法，可以仔细阅读。其中反射信息中的属性都会被封装为`Getter`和`Setter`，方法也会被封装到FunctionTemplate中。Js侧拿到封装好的信息后，就可以访问UE侧的对象了。对对象属性的访问会转发到`Getter`和`Setter`上去，而对方法的访问会转发到`FFunctionTranslator::Call(const v8::FunctionCallbackInfo<v8::Value>& Info)`这个方法上去。
3. 如果该类型存在父类型，则递归调用`GetTemplateOfClass`方法，得到其父类型的`FunctionTemplate`，将其作为该类型的原型对象

总结

LoadUEType函数将UE侧的类型加载到JS侧以供调用，具体实现是通过收集类型上的反射信息，收集类型及其父类型身上的方法，封装成一个FunctionTemplate返回。

TypescriptGeneratedClass与DynamicInvoker

继承引擎类功能实现Ts侧按照特定写法继承一个UE类，会在UE侧生成一个对应的蓝图资产。需要搞明白这两件事：

1. 如何生成蓝图资产
2. 蓝图资产如何实现动态绑定Ts代码
3. 如何实际调用到Js方法

如何生成蓝图资产

自动生成蓝图的原理是：监听Js文件变化，将继承UE类的Js类封装为蓝图类并序列化为资产文件。

1. **监听Js文件变化**

在Plugins/Puerts/PuertsEditor/Private/PuertsEditorModule中可以看到，UE编辑器会在Engine Init的时候启动一个特殊的虚拟机，这个虚拟机专门用来监听Js文件变化并生成对应资产。

2. **生成TypescriptGeneratedClass**

当监听到文件变化后，PuerTs通过PEBlueprintAsset来创建对应的蓝图。首先调用 `LoadOrCreate` 生成Js类对应的TypescriptGeneratedClass，然后调用 `AddMemberVariable` 、 `AddFunction` 等方法添加资产中在蓝图编辑器上可见的属性和方法。注意此时应该并没有完成蓝图上的方法和Ts脚本方法的绑定，只是让方法能在编辑器中看到。

TypescriptGeneratedClass是继承于BlueprintGeneratedClass的，涉及到蓝图的底层架构，具体可以参考这篇文章，讲的比较细致：

https://neil3d.github.io/unreal/bp_in_depth.html#%E7%82%B9%E5%87%BBcompile%E6%8C%89%E9%92%AE%E7%B…
neil3d.github.io

这一部分和蓝图编译生成BlueprintGeneratedClass很类似，只是追加的属性和方法都是Ts侧Wrap过来的。之后也会调用蓝图的编译接口。

3. **序列化为资产**

最后，通过调用 `PEBlueprintAsset::Save()` 会将TypescriptGeneratedClass保存为资产。这其中包括蓝图编译、序列化为uaaset等操作。此外，还会将所有需要被绑定的函数名称记录在TypescriptGeneratedClass中，以便后面完成动态绑定。

4. **总结**

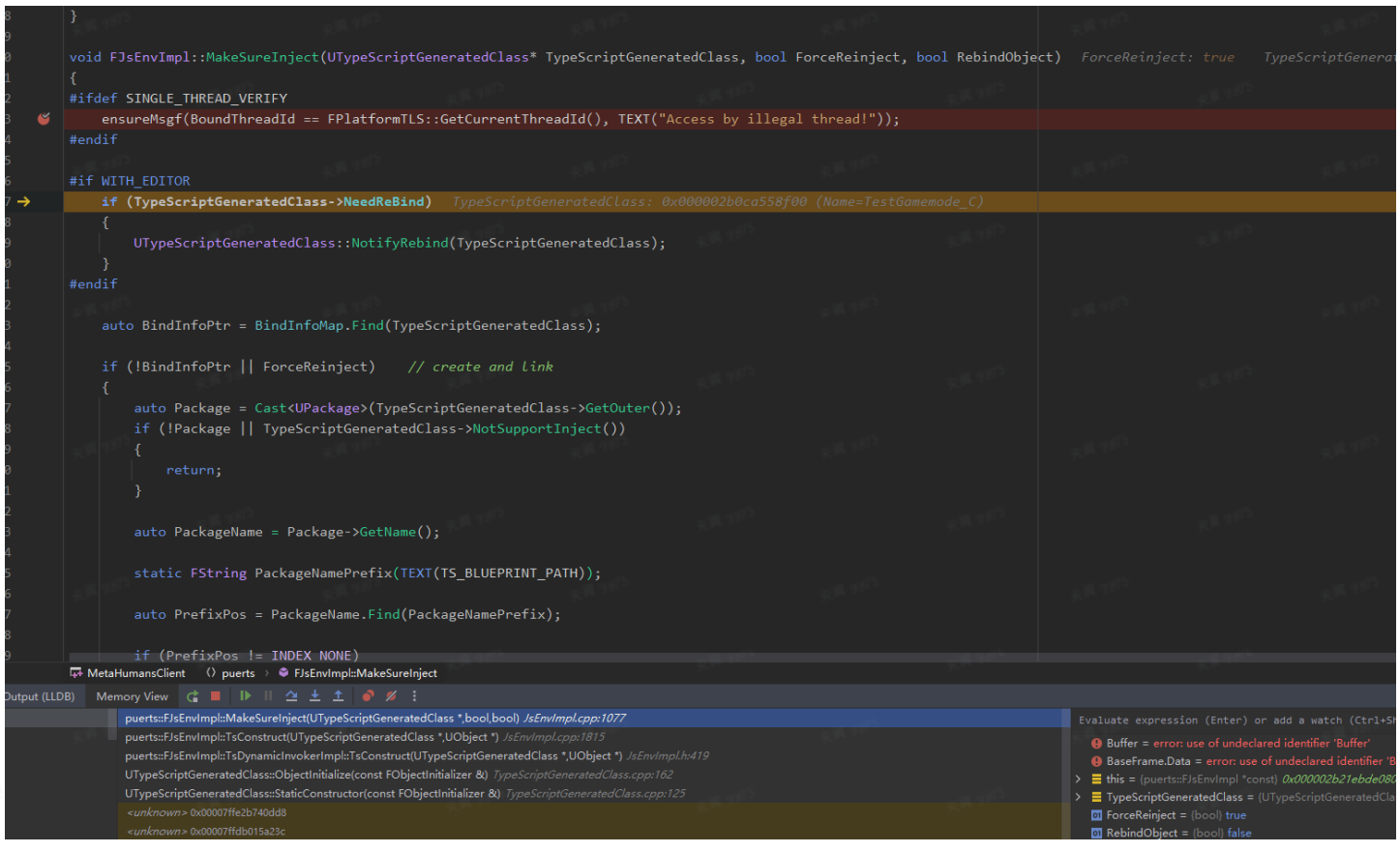
PuerTs通过监听Js文件变化来自动生成蓝图资产。生成的蓝图类型为TypescriptGeneratedClass，它是继承于普通的蓝图类BlueprintGeneratedClass的，其中附带了Ts侧的类型信息。

如何动态绑定Ts脚本

我们使用一个例子来测试，这是Ts类，它继承自UE类的 `GamemodeBase` ，并重写了 `RecieveBeginPlay` 方法。

1. **TypescriptGeneratedClass初始化**

`UTypeScriptGeneratedClass::StaticConstructor` 会对所有生成类蓝图进行初始化，可以看一下调用栈：



可以看到走到了JsEnvImpl.cpp的`MakeSureInject`函数，这个函数是绑定Ts脚本的关键。MakeSureInject主要做了这几件事情：

1. 通过调用`UTypeScriptGeneratedClass::NotifyRebind`方法，找到继承链上所有需要绑定的类，递归调用`MakeSureInject`完成注入。
2. 通过蓝图存放的位置，找到对应Js文件存放的位置，生成一份绑定信息`BindInfo`，记载了该生成类对应的Js模块路径
3. 遍历所有生成类上需要绑定的方法UFunction，通过v8虚拟机访问到Js侧具体的方法实现，缓存到TsFunctionMap中（UFunction -> v8::function）

至此就基本完成了Js脚本的绑定。

如何调用到Js方法

在TypescriptGeneratedClass中，使用了宏DEFINE_FUNCTION，将所有对蓝图方法的调用转发到了JS上。

DEFINE_FUNCTION作用是定义蓝图字节码的C++实现的，这里应该做了一些什么操作，所有对该蓝图的调用都会转发到execCallJS上面：

```
DEFINE_FUNCTION(UTypeScriptGeneratedClass::execCallJS)
{
    UFunction* Func = Stack.CurrentNativeFunction ? Stack.CurrentNativeFunction
    check(Func);
    // UE_LOG(LogTemp, Warning, TEXT("overrided function called, %s(%p)"), *Func

    UTypeScriptGeneratedClass* Class = Cast<UTypeScriptGeneratedClass>(Func->Get
    if (Class)
    {
        Class->ProcessPendingConstructJob();
        auto PinedDynamicInvoker = Class->DynamicInvoker.Pin();
        if (PinedDynamicInvoker)
        {
            PinedDynamicInvoker->InvokeTsMethod(Context, Func, Stack, RESULT_PAR
        }
        else
```

```
        {
            UE_LOG(Puerts, Error, TEXT("call %s::%s fail!, DynamicInvoker invali
        }
    }
    else
    {
        UE_LOG(Puerts, Error, TEXT("calling a not ts class method %s::%s"), *Fun
    }
}
```

在execCallJs中，主要是通过该生成类对应的`DynamicInvoker`去调用一个Ts方法。由于DynamicInvoker是和虚拟机一一绑定的，所以一个生成类只能将它的Js方法全部绑定到一个虚拟机上去。

这之后会调用到`FJsEnvImpl::InvokeTsMethod`来调用到具体的Js方法上去，首先通过TsFunctionMap找到该UFunction对应的JsFunction，再通过虚拟机调用

```
void FJsEnvImpl::InvokeTsMethod(UObject* ContextObject, UFunction* Function, FFr
{
#ifdef SINGLE_THREAD_VERIFY
    ensureMsgf(BoundThreadId == FPlatformTLS::GetCurrentThreadId(), TEXT("Access
#endif
#ifdef THREAD_SAFE
    v8::Locker Locker(MainIsolate);
#endif
    auto FuncInfo = TsFunctionMap.Find(Function);
    if (!FuncInfo)
    {
        Logger->Error(FString::Printf(TEXT("call %s::%s of %p fail: can not find
            *ContextObject->GetClass()->GetName(), *Function->GetName(), Context
        SkipFunction(Stack, RESULT_PARAM, Function);
        return;
    }
    else
    {
        auto Isolate = MainIsolate;
        v8::Isolate::Scope IsolateScope(Isolate);
        v8::HandleScope HandleScope(Isolate);
        auto Context = DefaultContext.Get(Isolate);
        v8::Context::Scope ContextScope(Context);

        v8::Local<v8::Value> ThisObj = v8::Undefined(Isolate);

        if (!Function->HasAnyFunctionFlags(FUNC_Static))
        {
            const auto PersistentValuePtr = GeneratedObjectMap.Find(ContextObjec
            if (!PersistentValuePtr)
            {
                Logger->Error(FString::Printf(TEXT("call %s::%s of %p fail: can
                    *ContextObject->GetClass()->GetName(), *Function->GetName(),
                SkipFunction(Stack, RESULT_PARAM, Function);
                return;
            }
            ThisObj = PersistentValuePtr->Get(Isolate);
        }

        v8::TryCatch TryCatch(Isolate);
```

```
41
42          FuncInfo->FunctionTranslator->CallJs(
43              Isolate, Context, FuncInfo->JsFunction.Get(Isolate), ThisObj, Contex
44
45          if (TryCatch.HasCaught())
46          {
47              Logger->Error(FString::Printf(TEXT("call %s::%s of %p fail: %s"), *C
48                  *Function->GetName(), ContextObject, *FV8Utils::TryCatchToString
49          }
50      }
51  }
```

总结

继承引擎类功能的实现主要是以下几点：

1. 监听Js文件变化来生成TypescriptGeneratedClass类，它存储了所有Ts类上的相关信息
2. 在TypescriptGeneratedClass的初始化阶段，调用MakeSureInject来实现脚本绑定
3. 通过DEFINE_FUNCTION实现一个调用Js的统一接口execCallJs，通过TsFunctionMap找到对应的Js函数完成调用

Mixin原理

PuerTs对Mixin的介绍是这样的：

> 把一个ts类（假设是类A）mixin到一个蓝图类（类B）的能力：
>
> - 如果A和B都有同样的函数，A的逻辑会替换B的
> - 一些事件（比如，ReceiveBeginPlay），如果A有，继续B没有，也会被回调
> - 可新增方法或字段

做一个简单的测试：

```
testActor.PrintText();

// 执行mixin
const MixinBpCls = blueprint.tojs<typeof UE.Game.Playground.TestActor.TestActor_C>(bpCls);
const MixinedBpCls = blueprint.mixin(MixinBpCls, MixinTestActor);

testActor.PrintText();
```

将一个Ts类Mixin到TestActor这个蓝图类中，尝试替换掉PrintText方法：

```
LogOnline: OSS: Creating online subsystem instance for: :Context_39
LogBlueprintUserMessages: [TestActor_4] HelloHelloHelloHello
Puerts: Error: (0x000002B1C9D39DB0) WAAAAAAAAAAAAA
PIE: Server logged in
```

发现确实成功被替换了。查看Mixin在C++侧的逻辑，位于 `FJsEnvImpl::Mixin` 中，简单看一下。

Mixin如何替换蓝图方法的

在 `FJsEnvImpl::Mixin` 中，PuerTs遍历Js类的方法，并寻找蓝图上是否存在该方法。如果找到，则调用 `UJSGeneratedClass::Mixin` 完成Js方法对蓝图类方法的替换工作。

```
1  auto Keys = MixinMethods->GetOwnPropertyNames(Context).ToLocalChecked();
```

```
TArray<FName> ReplaceMethodNames;
for (decltype(Keys->Length()) i = 0; i < Keys->Length(); ++i)
{
    auto Key = Keys->Get(Context, i).ToLocalChecked();
    auto MethodName = FV8Utils::ToFName(Isolate, Key);
    auto Function = To->FindFunctionByName(MethodName);
    if (Function)
    {
        auto JsFunc = MixinMethods->Get(Context, Key).ToLocalChecked();
        auto MixinedFunc = UJSGeneratedClass::Mixin(Isolate, New, Function, Mixi
        MixinFunctionMap.Emplace(
            MixinedFunc, v8::UniquePersistent<v8::Function>(Isolate, v8::Local<v
        ReplaceMethodNames.Add(MethodName);
    }
}
```

跟进去看一下 `UJSGeneratedClass::Mixin` ,大概是把原来蓝图类中的UFunction重定向到Ts侧来。

```
UFunction* UJSGeneratedClass::Mixin(v8::Isolate* Isolate, UClass* Class, UFuncti
    TSharedPtr<puerts::IDynamicInvoker, ESPMode::ThreadSafe> DynamicInvoker, boo
{
    // ......

    Function->DynamicInvoker = DynamicInvoker;
    Function->FunctionTranslator = std::make_unique<puerts::FFunctionTranslator>
    Function->TakeJsObjectRef = TakeJsObjectRef;

    Function->Next = Class->Children;
    Class->Children = Function;
    Class->AddFunctionToFunctionMap(Function, Function->GetFName());

    Function->FunctionFlags |= FUNC_Native;     //让UE不走解析
    Function->SetNativeFunc(&UJSGeneratedFunction::execCallMixin);
    Function->Bind();
    Function->StaticLink(true);
    Function->ClearInternalFlags(EInternalObjectFlags::Native);

    if (Existed)
    {
        Function->Original = Super;
        Function->OriginalFunc = Super->GetNativeFunc();
        Function->OriginalFunctionFlags = Super->FunctionFlags;
        Super->FunctionFlags |= FUNC_Native;     //让UE不走解析
        Super->SetNativeFunc(&UJSGeneratedFunction::execCallMixin);
        Class->AddNativeFunction(*Super->GetName(), &UJSGeneratedFunction::execC
        UJSGeneratedFunction::SetJSGeneratedFunctionToScript(Super, Function);
    }
    return Function;
}
```

Mixin是如何添加蓝图方法的

跟了一下还没有看到，后面补充吧