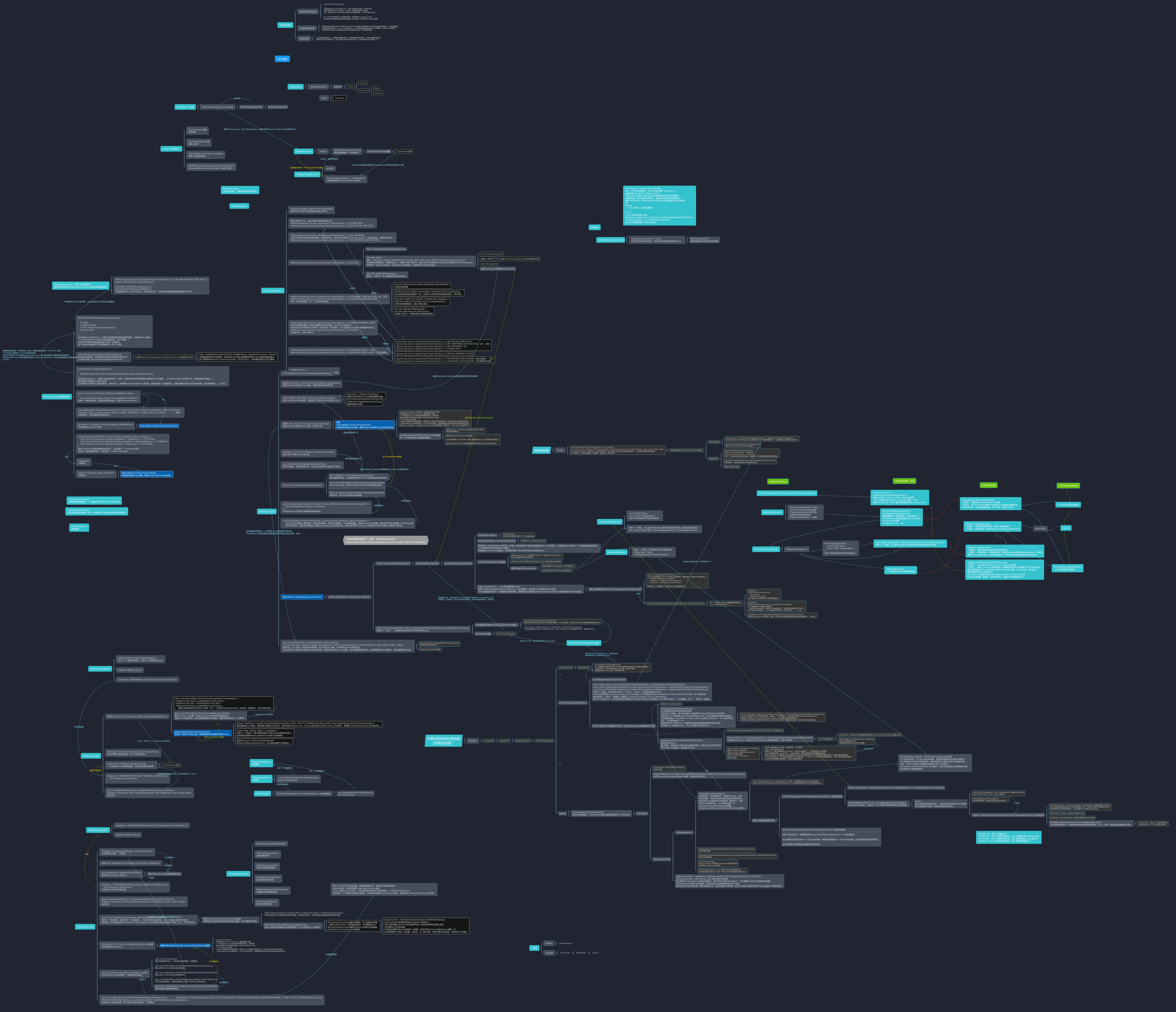

- 写文章的思路
  - 类型系统代码的生成
    - 从GENERATED_BODY()宏出发
    - 大致地讲讲generated.h和gen.cpp。给那一堆宏划分出区块，各自的作用。讲的时候注重共性与个性的区分，比如多个构造函数就是每个都有的。此外，像FObjectInitializer这些与主题关系不是很紧密的，可以作为额外话来讲。
    - 在一个个空文件的基础上加上函数和变量，开始讲解UE4Codegen_Private。然后说明对于函数和变量的信息是用哪些代码来生成的，同时讲解MetaData元数据。
  - 生成的信息的收集
    - 将前面生成的信息进行收集，讲解FCompiledInDefer类是如何把所有的UClass类中的信息都收集到一个全局变量里的。讲的时候要注重其和Z_Construct_XXX的关系——这个构造函数是被收集到那个全局变量里，在注册时才被调用的。也已被要讲到利用static的特性实现在引擎循环开始之前，将信息都收集好。
  - 信息的消费
    - 从引擎的注册流程讲起，在哪里会用到哪些信息。讲的时候就要按照流程来，也要从数据的角度来讲。最后等引擎自己消费讲完后，可以讲讲我们用户如何进行消费（这个我还需要写点代码实验一下）

UE4反射

- UObjectBase { UObjectBaseUtility { UObject { UField { UEnum / UStruct { UClass / UFunction
- FField { FProperty

- NewObject<>函数 { StaticConstructObject_Internal方法 { StaticAllocateObject方法 { AllocateUObject方法
  - 分配内存
  - 调用ClassConstructor，传入FObjectInitializer，创建对象在StaticAllocateObject方法分配的内存上。

- UClass（还没写完）
  - ClassConstructor变量 类构造器。
  - ClassDefaultObject变量 类默认对象。
  - TMap<FName, UFunction*> FuncMap 存储了类函数的映射。
  - FGCReferenceTokenStream ReferenceTokenStream 在AssembleReferenceTokenStream( )函数中完成。

- FObjectInitializer { Get方法 { FUObjectThreadContext::Get() 返回当前线程的一个单例实例。 { TopInitializerChecked函数 { TopInitializer函数
  - Get方法，返回单例实例。
  - TopInitializer函数返回线程的FObjectInitializer堆栈中的顶部的那个元素。
  - 当前线程会保存一个FObjectInitializer的堆栈。
- FUObjectThreadContext
  - Get方法
  - TArray<FObjectInitializer*> InitializerStack; 当前线程使用的FObjectInitializer的堆栈
- FRestoreClassInfo 记住类的信息，以便在调用后可以恢复它
- FObjectInitializer

MyClass.generated.h

- PRAGMA_DISABLE_DEPRECATION_WARNINGS 按照平台不同进行不同的被弃用警告 的声明。
- 暂时不知道干什么，也找不到定义在哪里的两个宏。#define CppLearning_Source_CppLearning_Private_MyClass_h_15_SPARSE_DATA #define CppLearning_Source_CppLearning_Private_MyClass_h_15_INCLASS_NO_PURE_DECLS
- #define CppLearning_Source_CppLearning_Private_MyClass_h_15_RPC_WRAPPERS 这两个宏声明了供蓝图调用的函数。此时还是空宏，读者可在左侧的MyClass.generated.h（添加函数版）中看到它的作用。#define CppLearning_Source_CppLearning_Private_MyClass_h_15_RPC_WRAPPERS_NO_PURE_DECL
- #define CppLearning_Source_CppLearning_Private_MyClass_h_15_INCLASS \
  - static void StaticRegisterNativesUMyClass();
  - DECLARE_CLASS 参数：(UMyClass, UObject, COMPILED_IN_FLAGS(0), CASTCLASS_None, TEXT("/Script/CppLearning"), NO_API) 声明类的时候要做的一些重复性的工作。重载new和=操作符，将传入的Class和基类Class用typedef重命名为ThisClass和Super。还声明了一些StaticGet方法，比如StaticClass获得类，还有获得Package和Flag的。
    - StaticClassCastFlags方法
    - 重载new操作符 { 调用StaticAllocateObject方法来创建新的对象
    - StaticPackage方法
    - 调用了gen.cpp中声明的StaticClass方法
  - DECLARE_SERIALIZER(UMyClass) 重载了<<操作符，传入的是序列化相关的内容。
- #define CppLearning_Source_CppLearning_Private_MyClass_h_15_STANDARD_CONSTRUCTORS（旧）【或】#define CppLearning_Source_CppLearning_Private_MyClass_h_15_ENHANCED_CONSTRUCTORS 声明一系列构造函数，与一个包装好的构造器。
  - NO_API UMyClass(const FObjectInitializer& ObjectInitializer); 标准的构造函数
  - DEFINE_DEFAULT_OBJECT_INITIALIZER_CONSTRUCTOR_CALL(UMyClass) 将上面的标准构造函数套了个壳，以便UClass保存其构造函数的指针，用于反射。
  - DECLARE_VTABLE_PTR_HELPER_CTOR(NO_API, UMyClass); \ DEFINE_VTABLE_PTR_HELPER_CTOR_CALLER(UMyClass); \ 引擎内部热重载相关，建议不要去深究。
  - NO_API UMyClass(UMyClass&&); \ NO_API UMyClass(const UMyClass&); \ 禁用掉了原生C++提供的移动与复制构造函数
- #define CppLearning_Source_CppLearning_Private_MyClass_h_15_PRIVATE_PROPERTY_OFFSET 类的私有属性的偏移。类首先需要有私有成员变量，这个宏才不会是空宏。类的成员在内存中的地址与类的入口地址是有一点距离的，这个宏就是为了指出这个偏移量而存在的。#define CppLearning_Source_CppLearning_Private_MyClass_h_12_PROLOG 不清楚作用，也找不到定义。
- #define CppLearning_Source_CppLearning_Private_MyClass_h_15_GENERATED_BODY \ 【或】#define CppLearning_Source_CppLearning_Private_MyClass_h_15_GENERATED_BODY_LEGACY（历史遗留版）
  - 调用宏
  - CppLearning_Source_CppLearning_Private_MyClass_h_15_INCLASS_NO_PURE_DECLS CppLearning_Source_CppLearning_Private_MyClass_h_15_RPC_WRAPPERS_NO_PURE_DECLS（新）【或】CppLearning_Source_CppLearning_Private_MyClass_h_15_RPC_WRAPPERS（旧）CppLearning_Source_CppLearning_Private_MyClass_h_15_SPARSE_DATA
  - CppLearning_Source_CppLearning_Private_MyClass_h_15_PRIVATE_PROPERTY_OFFSET
  - CppLearning_Source_CppLearning_Private_MyClass_h_15_ENHANCED_CONSTRUCTORS（新的加强版）【或】CppLearning_Source_CppLearning_Private_MyClass_h_15_STANDARD_CONSTRUCTORS（历史遗留的标准版）
- CURRENT_FILE_ID
- 调用UMyClass::StaticClass()来获得内存中静态的对象

MyClass.generated.h（声明了函数的版本）

建议先把右边的MyClass,MyEnum,MyStruct看完后再看这部分

- #define CppLearning_Source_CppLearning_Private_MyClass_h_15_RPC_WRAPPERS_NO_PURE_DECLS \ virtual void NativeFunc_Implementation(); \ \ DECLARE_FUNCTION(execNativeFunc); \ DECLARE_FUNCTION(execCallableFunc) 前面提到的RPC_WRAPPERS宏，此时发挥作用了。将这些供蓝图调用的函数信息保存下来。

h中使用DECLARE宏声明，cpp中使用DEFINE宏定义函数体。

MyClass.gen.cpp(声明函数版)

需要提前注意的是，不管是Native与否，函数后面都会生成一个UFunction对象。只不过Native函数的UFunction在绑定的时候会去它所属于的UClass里的NativeFunctionLookupTable通过函数名字查找真正的函数指针，而非Native的UFunction会把函数指针指向UObject::ProcessInternal，用来处理蓝图虚拟机调用的情况(by大钊)

- DEFINE_FUNCTION(UMyClass::execNativeFunc) { P_FINISH; P_NATIVE_BEGIN; P_THIS->NativeFunc_Implementation(); P_NATIVE_END; } 另外还有execCallbleFunc，这里针对的是蓝图可能没实现的函数。(也就是Native函数)（ImplementableFunc因为必然有蓝图实现，所以不需要。）因此在这里需要为蓝图调用的版本声明一份函数体。专门生成exec前缀是为了供蓝图虚拟机（VM）调用。
- void UMyClass::StaticRegisterNativesUMyClass() 原本是没有实现的，现在我们得知它要负责函数注册的工作。在这里只注册了execCallableFunc和execNativeFunc
  - 调用FNativeFunctionRegistrar::RegisterFunctions对函数进行注册
    - Class->AddNativeFunction(UTF8_TO_TCHAR(InArray->NameUTF8), InArray->Pointer); 注册函数指针和名字的映射。把相应的Class内定义的函数添加到UClass内部的函数表里去。那个函数表是NativeFunctionLookupTable，定义在Class.h，感兴趣的读者可以自行翻阅。
- void UMyClass::ImplementableFunc() { ProcessEvent(FindFunctionChecked(NAME_UMyClass_ImplementableFunc),NULL); } 另外还有NativeFunc，这里针对的是要求有C++实现，但是可能没有实现的函数(也就是非Native函数)。（CallableFunc是一定实现了的，否则编译都无法通过。）因此需要在这里提供一份默认实现。这也就是UHT帮我们生成的函数体。当我们在C++里调用ImplementableFunc的时候，其实会触发一次函数查找，如果在蓝图中有定义该名字的函数，则会得到调用。（by大钊）
- struct Z_Construct_UFunction_UMyClass_CallableFunc_Statics { static const UE4CodeGen_Private::FFunctionParams FuncParams; }声明一个静态的结构，存放对应函数的信息。类型为FFuntionParams。
- const UE4CodeGen_Private::FFunctionParams Z_Construct_UFunction_UMyClass_CallableFunc_Statics::FuncParams = { (UObject*(*)())Z_Construct_UClass_UMyClass, nullptr, "CallableFunc", nullptr, nullptr, 0, nullptr, 0, ...省略}; 对其初始化，将该函数的必要信息填入。
- 传入
- UFunction* Z_Construct_UFunction_UMyClass_CallableFunc() 构造函数的UFunction*对象 { UE4CodeGen_Private::ConstructUFunction
- const FClassFunctionLinkInfo Z_Construct_UClass_UMyClass_Statics::FuncInfo[] = { { &Z_Construct_UFunction_UMyClass_CallableFunc, "CallableFunc" }, // 3778107657 { &Z_Construct_UFunction_UMyClass_ImplementableFunc, "ImplementableFunc" }, // 165807643 { &Z_Construct_UFunction_UMyClass_NativeFunc, "NativeFunc" }, // 1037346021 } 现在ClassParams也要存储函数的信息了，因此增加了一个FuncInfo字段。类似地，要存储属性信息，也会增加一个PropPointers字段。
- ClassParams 不再赘述。
- UClass* Z_Construct_UClass_UMyClass(); 不再赘述。 { UE4CodeGen_Private::ConstructUClass 来构造并注册UClass对象。请与ConstructUFunction区别。

- FClassFunctionLinkInfo 类里的函数链接信息，一个函数名字对应一个UFunction对象
- FCppClassTypeInfoStatic 类在Cpp里的类型信息，用一个结构是为了将来也许还会添加别的字段
- FFunctionParams 函数参数

MyClass.gen.cpp

- void EmptyLinkFunctionForGeneratedCodeMyClass() {} 位置
- 项目名_API UClass* Z_Construct_UClass_UMyClass_NoRegister(); 构造UMyClass对应的UClass对象，但是没有后续的注册过程
- COREUOBJECT_API UClass* Z_Construct_UClass_UObject(); 引用CoreUObject里的函数，主要是为了得到UObject对应的UClass
  - OuterClass = UObject::StaticClass(); 调用UObject的StaticClass获取其静态对象。
  - UObjectForceRegistration(OuterClass); 强制对其进行注册。
- 项目名_API UClass* Z_Construct_UClass_UMyClass(); 构造UMyClass对应的UClass对象，并进行注册。
  - 调用 UE4CodeGen_Private::ConstructUClass 来构造并注册UClass对象，确保UObject本身的UClass已经注册生成
    - ConstructUProperties（下面UStruct的构造中也有这个函数。）内部转发给ConstructFProperties，源码原理比较简单。一个400行的switch-case语句来判断属性的类型，然后根据类型不同调用不同的属性构造函数（比如FFieldPathProperty，FTextProperty之类的）。ReadMore变量针对的是容器类型。如果有进一步构造子属性的需求的话，会将ReadMore的值设定的更大（比如Set的ReadMore被设置为1）在switch语句结束后，函数的最后部分用循环将更多的需求递归构造完。NewProp->ArrayDim = PropBase->ArrayDim;//设定属性维度，单属性为1，int32 prop[10]这样的为10
    - CreateLinkAndAddChildFunctionsToMap函数 用一个for循环将传入的函数组遍历
      - 使用Functions->CreateFuncPtr()创建出UFunction对象，并将对象串成链表。
      - 调用AddFunctionToFunctionMap函数 UClass里保存有一个FuncMap，存储了函数对象(UFunction)与函数名字的映射。AddFunctionToFuntionMap函数直接调用FuncMap.Add进行添加。
  - 转发给GetPrivateStaticClass方法
  - 指针数组存储了它
  - 传入ClassParams参数。
- UPackage* Z_Construct_UPackage__Script_CppLearning(); 构造 项目 本身的UPackage对象
- void UMyClass::StaticRegisterNativesUMyClass() 静态注册函数，然而其暂时为空。在类中加入函数之后会被UHT实现。
- struct Z_Construct_UClass_UMyClass_Statics
  - static UObject* (* const DependentSingletons[])(); 静态函数指针数组，这里面要保存该UClass所依赖的对象的构造函数。
  - static const UE4CodeGen_Private::FClassParams ClassParams; FClassParams是一个保存了该UClass类下许多必要信息的结构。
  - static const FCppClassTypeInfoStatic StaticCppClassTypeInfo; 用来标志，该UClass类是否为抽象类的。
  - 被作为初始化FClassParams结构类的ClassParams变量的参数之一
- const FCppClassTypeInfoStatic Z_Construct_UClass_UMyClass_Statics::StaticCppClassTypeInfo = { TCppClassTypeTraits<UMyClass>::IsAbstract, }; 对标志该UClass类是否为抽象类的变量初始化
  - 对其初始化
- const UE4CodeGen_Private::FClassParams Z_Construct_UClass_UMyClass_Statics::ClassParams = { //在这个结构中要填入属性数组，依赖对象函数组，类的未注册函数，类中函数的数量，属性(Properties)的数量，被实现了的接口的数量，EClassFlags等 //因为有好多行，建议感兴趣的自己去看generated.cpp中这块的代码，看的时候对照FClassParams结构的定义即可知道每个参数是干啥用的。
- 将注册函数传递给它。在引擎进入main线程后开始注册。FClassParams会把收集来的类的信息在注册时交给内存统一保存。

本思维导图来源知乎，作者：DarkFlameMaster
https://www.zhihu.com/people/xian-sui-bian-qi-ge-ming-hao-liao/posts

- IMPLEMENT_CLASS(UMyClass, 879100156) { #define IMPLEMENT_CLASS(TClass, TClassCrc)
  - UClass* TClass::GetPrivateStaticClass({ GetPrivateStaticClass方法 { GetPrivateStaticClassBody方法
    - FindObject<UClass> { StaticFindObject 从内存中寻找是否已经存在了一个待生成的对象
    - GUObjectAllocator . AllocateUObject方法 { 利用FMemory::Malloc分配一块内存
    - 然后使用 ::new(UMyClass)来生成一个对象。此时会将套了壳的构造函数指针传入以供初始化，也就是说UClass保存了一个构造自身函数的指针。::new是最外层命名空间的new运算符 这里是placement new的语法，将对象生成在了AllocateUObject分配的内存上。
    - InitializePrivateStaticClass函数
      - 调用InSuperClassFn()，设定该类的SuperStruct（即Super::StaticClass()。）换句话说就是声明该类的基类是谁。
      - InWithinClassFn()（即WithinClass::StaticClass()）设定该类的Outer类类型
      - 调用UObjectBase::Register()
        - 将自己添加到FPendingRegistrantInfo的映射中。
        - 将自己添加到FPendingRegistrant的链表中。
    - RegisterNativeFunc() （以下两行解释来自大钊）就是上文的StaticRegisterNativesUMyClass，在此刻调用，用来像UClass里添加Native函数。Native函数指的是在C++有函数体实现的函数，而蓝图中的函数和BlueprintImplementableEvent的函数就不是Native函数。
      - 实质上调用的是UMyClass::StaticRegisterNativesUMyClass()
        - static const FNameNativePtrPair Funcs[] = { //exec开头的都是在.generated.h里定义的蓝图用的，暂时不管它，理解为可以调用就行了。 { "AddHP", &UMyClass::execAddHP }, { "CallableFunc", &UMyClass::execCallableFunc }, { "NativeFunc", &UMyClass::execNativeFunc }, }; 这里声明了一个函数数组，把这个类的Native函数都装进去。
        - 传递进去
        - FNativeFunctionRegistrar::RegisterFunctions(Class, Funcs, ARRAY_COUNT(Funcs)); { 用一个for循环，将Native函数的数据添加进去。Class->AddNativeFunction
          - 定义在Class.h struct FNativeFunctionLookup(FName Name; FNativeFuncPtr Pointer;)这个便是UClass中存储Native函数的数据结构。
          - 定义在Class.h TArray<FNativeFunctionLookup> NativeFunctionLookupTable; 这个便是存储Native函数信息的数组。"这里不用TMap而用TArray是因为一般来说我们的一个类里写的函数数量并不会太多，对于元素比较少的情况下，TArray的线性查找也很快，而且还省内存。"——大钊
          - new(NativeFunctionLookupTable) FNativeFunctionLookup(InName,InPointer); 同样用了placement new的语法，新建一个存储Native函数信息的结构并将其保存到数组中。一气呵成。
  - 回顾前面的内容，我们知道StaticClass方法真正实现是在GetPrivateStaticClassBody方法。在那里面，会分配内存，真正在内存中创建出对象，然后将其添加到映射中，等待注册。
  - static TClassCompiledInDefer<TClass> AutoInitialize##TClass(TEXT(#TClass), sizeof(TClass), TClassCrc); \ 注意这个"static"，这就意味着它会发生在引擎的主循环之前。
    - 在构造函数中调用UClassCompiledInDefer函数。
      - GetDeferredClassRegistration().Add(ClassInfo) 将该UClass类的FFieldCompiledInInfo信息存储到一个TArray单例中，在引擎CoreUObject模块加载的时候会使用上。
      - TMap<FName, FFieldCompiledInInfo*>& DeferMap =GetDeferRegisterClassMap(); 这个映射看起来和上面的TArray单例大同小异，实际上它只在HotReload热重载时才用上，因此我就不关心。
    - 定义Register函数 { return TClass::StaticClass();
    - 调用Register方法，其实就是调用了StaticClass方法。
    - GetDeferredClassRegistration函数
- static FCompiledInDefer Z_CompiledInDefer_UClass_UMyClass (Z_Construct_UClass_UMyClass, &UMyClass::StaticClass, TEXT("/Script/CppLearning"), TEXT("UMyClass"), false, nullptr, nullptr, nullptr); 延迟注册，注入信息，在启动的时候调用。由于这是static变量，它的初始化在main函数之前。延迟注册是为了避免在引擎启动时注册过多东西，使得迟迟才能进入main函数，用户观感就是进程卡死。因此要延迟到main函数中，用多线程等技术注册。
  - GetConvertedDynamicPackageNameToTypeName().Add 将其添加到待注册的列表中。
  - UObjectCompiledInDefer函数

- 获取到之前收集的信息，现在要消费它们了！

信息的收集

- UClass
  - static FCompiledInDefer Z_CompiledInDefer_UClass_UMyClass (Z_Construct_UClass_UMyClass, &UMyClass::StaticClass, TEXT("/Script/CppLearning"), TEXT("UMyClass"), false, nullptr, nullptr, nullptr); FCompiledInDefer是一个静态结构，static的特性决定了它在引擎开始之前就会被初始化——也就是说调用它的构造函数。传入的参数有：类型的构造函数，获取函数，包的路径，类的名字等。
    - 构造函数调用了UObjectCompiledInDefer函数。
      - 如果不是动态类
        - TArray<UClass *(*)()>& DeferredCompiledInRegistration = GetDeferredCompiledInRegistration(); GetDeferredCompiledInRegistration函数会返回一个待注册的函数列表——也是静态的，惰性初始化且单例。
        - DeferredCompiledInRegistration.Add(InRegister); 将Z_Construct_UClass_UMyClass函数传入。
      - 如果是动态类
        - FDynamicClassStaticData ClassFunctions; ClassFunctions.ZConstructFn = InRegister; ClassFunctions.StaticClassFn = InStaticClass; 声明了一个装有动态类的信息的变量，也就是保存了构造函数和静态获取类的函数。
        - GetDynamicClassMap().Add(FName(DynamicPathName), ClassFunctions); 将信息装到一个装有所有的动态/本地化类的映射中。
        - 将类名从路径名中删去

反射实战

- XXXProperty->ContainerPtrToValuePtr 传入一个UObject的地址，这个UObject便是"Container"，返回值是ContainerPtr+Offset_Internal。Offset_Internal指出了该Property在类中相对起始地址的偏移量。类描述的是一系列对象的共同特征，也就是说类的实例也都是这样。因此ContainerPtr+Offset_Internal=该Property类变量在内存中的地址。例：class A{ X var;//声明一个X类的变量var } A *A1; A *A2;//声明A的两个实例。FProperty* property=A1->GetClass()->FindPropertyByName("SomeName"); X *result=property->ContainerPtrToValuePtr(A1); 那么result的值就是A1中的var的地址。
- SetPropertyValue (A,Value) { *GetPropertyValuePtr(A) = Value; 用*解引用传回的A的地址，将A所在的地址的值设置为Value。 { GetPropertyValuePtr 将属性值的地址转换为适当的类型

- FPendingRegistrantInfo
  - const TCHAR* Name; const TCHAR* PackageName 记录了待注册类的名字和所在的包名。
  - 保存了一个映射，可以通过UObjectBase指针来找到注册信息(上面说的类名和包名。) static TMap<UObjectBase*, FPendingRegistrantInfo> PendingRegistrantInfo;
- FPendingRegistrant { 保存了一个链表，单纯地用于指出注册的顺序。UObjectBase* Object; FPendingRegistrant* NextAutoRegister

类型信息的代码生成 / 类型信息的收集、存储 / 类型信息的消费 / 引擎启动时的注册流程

- static FCompiledInDefer Z_CompiledInDefer_UClass_UMyClass → TArray<UClass *(*)()> GetDeferredCompiledInRegistration(); 存储Class的Z_Construct_UClass_UMyClass函数。（Enum则是存储EMyEnum_StaticEnum函数，Struct则是StaticStruct()，但它们最后也都是转发给Z_Construct_xxx）
- IMPLEMENT_CLASS { TClassCompiledInDefer<TClass> TClassCompiledInDefer其实是FFieldCompiledInInfo的子类。也就是说它也保存着类的Register,ClassPackage，size等。
- TArray<FFieldCompiledInInfo*> GetDeferredClassRegistration() 模块加载的类，延迟到我们一次性注册它们 内含有Register()函数(转发给StaticClass方法) ClassPackage()函数 Class的大小：Size。 等。
- GetPrivateStaticClassBody { UObjectBase::Register() → FPendingRegistrantInfo(const TCHAR* Name; const TCHAR* PackageName) 记录了待注册类的名字和所在的包名。
- static TMap<UObjectBase*, FPendingRegistrantInfo> PendingRegistrantInfo; 保存了一个映射，可以通过UObjectBase指针来找到上面说的类名和包名。
- FPendingRegistrant 一个单纯地记录了注册顺序的链表
- UClassRegisterAllCompiledInClasses 【消费】：消费类的UClass的StaticClass函数。【任务】：把CoreUObject里面定义的类的UClass都给先构建出来。注册并未完成，因为只是刚刚构建，有了对象，还得将它注册。
- UObjectProcessRegistrants 【消费】：获取待注册项的列表，得知了注册的顺序 【任务】：遍历该列表，调用UObjectForceRegistration来干活
- UObjectForceRegistration 【消费】：消费之前保存的类的名字和包的名字。【任务】：处理注册项，为其创建名字，所属的Package(调用DeferredRegister，里面再调用了CreatePackage)，设定它的类型等。并将待注册的对象添加到全局的列表中。
- UObjectLoadAllCompiledInDefaultProperties 【消费】：UClass的Z_Construct_UClass_UMyClass函数。【任务】：利用Z_Construct函数构建出（准确的说是获取之前构建好了的)Class对象后，通过前面设置的Package来将其分类为CoreUObject包的，Engine包的，其它包的。然后按照顺序为它们创建CDO。而Z_Construct函数会调用UE4Codegen_Private::ConstructXXX来为UClass设置函数，属性等，实现了的接口，配置文件名等最后的工作。
- CoreUObject模块加载时
- AppInit
- InitUObject
- ProcessNewlyLoadedUObjects 一个独自完整的注册函数。

引擎启动时的注册流程（信息的消费）

- WinMain { GuardedMain { EnginePreInit { GEngineLoop.PreInit { PreInitPreStartupScreen
  - LoadCoreModules { StartupModule { UClassRegisterAllCompiledInClasses 这一步骤的目的主要是为了把CoreUObject里面定义的类的UClass都给先构建出来。注册并未完成，因为只是刚刚构建，有了对象，还得将它注册。构建用的是StaticClass方法，在关联线中有写。
  - 空
  - ProcessNewlyLoadedUObjects
    - UClassRegisterAllCompiledInClasses
    - const TArray<UClass* (*)()>& DeferredCompiledInRegistration = GetDeferredCompiledInRegistration(); const TArray<FPendingStructRegistrant>& DeferredCompiledInStructRegistration = GetDeferredCompiledInStructRegistration(); const TArray<FPendingEnumRegistrant>& DeferredCompiledInEnumRegistration = GetDeferredCompiledInEnumRegistration(); 声明了三个数组，分别对应UObject，UStruct，UEnum，它们的结构都大同小异。UObject保存的是Z_Construct_UClass_UMyClass函数。这个函数在转发给UE4CodeGen_Private::ConstructUClass时，传入的参数里包含有获取"所在包"的函数（也就是Z_Construct_UPackage__Script_CppLearning）而UStruct和UEnum，它们的列表中的元素是FPendingXXXRegistrant的结构，这个结构中包含了一个注册函数，和一个"所在包"的变量。
    - While(上面的三个列表都还不为空，FPendingRegistrant的链表也不为空)
      - UObjectProcessRegistrants();
      - UObjectLoadAllCompiledInStructs(); 为代码里的枚举和结构构造类型对象 代码没什么好看的，用for循环遍历上面说的FPendingXXXRegistrant的列表 然后对每一个元素调用CreatePackage和RegisterFn，为它们创建包和调用注册函数。注册函数就是gen.cpp中的Z_Construct_UEnum_项目名_EMyEnum，Struct也是同理。另外，大钊特地提到了一点："顺序总是先enum再struct。其原因其实是因为更基础的类型总是先构造。代码里enum不能嵌套struct，但struct里可以包含enum。"
        - TArray<FPendingEnumRegistrant> PendingEnumRegistrants = MoveTemp(GetDeferredCompiledInEnumRegistration());
        - static FName LongEnginePackageName(TEXT("/Script/Engine")); //引擎包的名字
      - UObjectLoadAllCompiledInDefaultProperties(); 继续注册UObject。我们知道，代码里面UClass里可以包含结构和枚举。因此对Class的注册放在Struct和Enum的后面，这也是UE的讲究。
        - TArray<UClass* (*)()> PendingRegistrants = MoveTemp(GetDeferredCompiledInRegistration()); 也是用MoveTemp，这和对UEnum和UStruct里做的事类似，此处不再赘述。
          - 用一个For循环遍历它
            - UClass* Class = Registrant();//调用生成代码里的Z_Construct_UClass_UMyClass创建UClass
            - 用if else判断Class->GetOutermost()->GetFName() 判断Class所属的包是什么。//按照所属于的Package分到3个数组里
        - TArray<UClass*> NewClassesInCoreUObject; 装属于CoreUObject包的对象。TArray<UClass*> NewClassesInEngine; 装属于Engine包的对象。TArray<UClass*> NewClasses; 属于其它包的。
        - 分别用for循环遍历这三个数组，对其中的每一个元素调用 Class->GetDefaultObject(); 创建出CDO。循环的顺序是先CoreUObject，再Engine,最后其它。以下是翻译自大钊的解释："这三个数组的顺序是：CoreUObject，Engine和其他。按照此顺序构造的原因是根据依赖关系。构造CDO的过程，有可能触发uasset的加载和UObject构造函数的调用，所以就可能在内部触发其他Package里对象的加载和构造。CoreUObject最底层（它不会引用其他的Package里的对象），Engine次之（它有可能引用底层的对象），其他（就不确定会引用哪个了）。所以依照此顺序能避免依赖倒置，从而减少重复调用查找。"
  - 空
  - AppInit { FCoreDelegates::OnInit.Broadcast(); 在前文注册的委托，在CoreUOject模块加载的时候指向了InitUObject { InitUObject
    - FCoreDelegates::OnExit.AddStatic(StaticExit); 注册退出事件
    - FModuleManager::Get().OnProcessLoadedObjectsCallback().AddStatic(ProcessNewlyLoadedUObjects); 注册ProcessNewlyLoadedUObjects函数，后面会多次调用它。
    - StaticUObjectInit()
      - UObjectBaseInit();
        - UObjectProcessRegistrants(); 处理注册项，为其创建名字，所属的Package，设定它的类型等。并将待注册的对象添加到全局的列表中。然而它的CDO仍然未被创建，拥有名字，只是代表它可以被查找到。CDO的创建，在ProcessNewlyLoadedUObjects函数的UObjectLoadAllCompiledInDefaultProperties函数中。
          - TArray<FPendingRegistrant> PendingRegistrants;//声明一个暂存要被注册的UObject指针的列表。DequeuePendingAutoRegistrants(PendingRegistrants); //从链表中提取注册项到上面的暂存列表中。
          - 通过for循环遍历暂时列表。
            - UObjectForceRegistration(PendingRegistrant.Object); //真正的注册
              - TMap<UObjectBase*, FPendingRegistrantInfo>& PendingRegistrants = FPendingRegistrantInfo::GetMap();
              - FPendingRegistrantInfo* Info = PendingRegistrants.Find(Object); 用UObjectBase指针，来查找一下这个映射中是否保存着它的注册信息。
                - If(Info) 因为之前可能已经注册过了，注册过的对象会被从Map中移除 所以可能是找不到的。如果没找到就无事发生。
                  - const TCHAR* PackageName = Info->PackageName;//对象所在的Package const TCHAR* Name = Info->Name; //对象名
                  - PendingRegistrants.Remove(Object); 将它从映射中删除（因为我们现在已经将它注册过了）
                  - 传入参数
                  - Object->DeferredRegister(UClass::StaticClass(),PackageName,Name);//延迟注册
                    - UPackage* Package = CreatePackage(nullptr, PackageName); //创建对象属于的Package
                    - OuterPrivate = Package; //设定Outer到该Package
                    - ClassPrivate = UClassStaticClass; //设定该对象属于的UClass*类型
                    - AddObject(FName(InName), EInternalObjectFlags::None); //注册该对象的名字，将该对象添加到全局的对象列表中。这个"添加"其实就是注册的真正动作。 { NamePrivate = InName; //设定对象的名字
          - DequeuePendingAutoRegistrants(PendingRegistrants); //继续尝试提取 在每一项注册之后，都要重复调用DequeuePendingAutoRegistrants一下来继续提取。这么做是因为在真正注册一个UObject的时候，里面有可能触发另一个Module的加载，从而导致有新的注册项进来。所以就需要不断的提取注册直到把所有处理完
        - GUObjectAllocator.AllocatePermanentObjectPool(SizeOfPermanentObjectPool); 初始化对象分配器
        - GUObjectArray.AllocateObjectPool(MaxUObjects, MaxObjectsNotConsideredByGC, bPreAllocateUObjectArray); 初始化对象管理数组
        - void InitAsyncThread(); InitAsyncThread(); //初始化Package(uasset)的异步加载线 用来后续Package(uasset)的加载
        - Internal::GObjInitialized = true; //指定UObject系统初始化完毕 这样在后续就可以用bool UObjectInitialized()来判断对象系统是否可用。
      - GObjTransientPkg = NewObject<UPackage>(nullptr, TEXT("/Engine/Transient"), RF_Transient); GObjTransientPkg->AddToRoot(); //这个临时包总不会释放 我们可以发现在UObjectBaseInit初始化结束后，就已经可以开始NewObject了，标志着整个UObject系统的成功创建 GObjTransientPkg是个全局变量，所有没有Outer的对象都会放在这个包里。我们在NewObject的时候，如果不提供Outer，则会返回这个临时包，符合了UObject对象必须在UPackage里的一贯基本原则。

- NamePrivate：定义了对象的名字 OuterPrivate：定义了对象的从属关系，这个对象属于哪个包(UPackage) ClassPrivate：定义了对象的类型关系，这个对象的类型(UClass)是什么 SuperStruct：定义了类型的继承关系，这个对象的基类是什么
- 在UObjectBase.cpp中有一个名为FPendingRegistrant的结构 这个结构本质是一个UObjectBase的链表，保存那些要被自动注册的对象指针。这个链表指出了被注册对象的顺序。那看关联线可以看到它被写入信息的位置。同时，在文件中还将这个链表的头尾指针声明为了全局的。在DequeuePendingAutoRegistrants函数中，就可以直接通过全局的链表头指针，来对链表进行读取和删除操作。

MyEnum.generated.h

- #define FOREACH_ENUM_EMYENUM(op) 定义了一个遍历枚举的宏，只是为了方便使用(by大钊)
- #define CURRENT_FILE_ID
- template<> CPPLEARNING_API UEnum* StaticEnum<EMyEnum>

对它的实现

MyEnum.gen.cpp

- 项目名_API UEnum* Z_Construct_UEnum_CppLearning_EMyEnum();
  - static const UE4CodeGen_Private::FEnumeratorParam Enumerators[] = { { "EMyEnum::MY_Dance", (int64)EMyEnum::MY_Dance }, { "EMyEnum::MY_Rain", (int64)EMyEnum::MY_Rain }, { "EMyEnum::MY_Song", (int64)EMyEnum::MY_Song }, }; 将我们在枚举类型中定义的几个枚举，写入一个名为FEnumeratorParam（枚举对）的结构中，过会儿用来注册。
  - static const UE4CodeGen_Private::FEnumParams EnumParams 类似于UClass，这也是一个保存了UEnum中种种必要信息的结构。包括Enum本身的名字，展示给编辑器的名字的函数，里面有哪些枚举对，Flag等等。
    - 传给FEnumParams结构
  - UE4CodeGen_Private::ConstructUEnum 同样地，类似于UClass上面，这里也是用它来构造并注册UEnum。
    - UEnum* NewEnum = new (EC_InternalUseOnlyConstructor, Outer, UTF8_TO_TCHAR(Params.NameUTF8), Params.ObjectFlags) UEnum(FObjectInitializer()); 直接创建该Enum对象。源码里用了重载new的方式，注意不是placement new。这个new的方式定义在DECLARE_CLASS宏中。是调用了StaticAllocateObject来分配内存的。
    - TArray<TPair<FName, int64>> EnumNames; 声明了一个枚举对，用for循环来将传入的Params中的枚举对填入。然后再通过调用NewEnum的SetEnum来存入枚举对数组。
    - 调用NewEnum->SetEnumDisplayNameFn 将Params中的DispalyNameFunc，名字展示函数作为参数传入。
    - 传给ConstructUEnum函数
  - 传入后，被以Result = (*FnRegister()) 的形式调用。
- UPackage* Z_Construct_UPackage__Script_CppLearning(); 生成项目其本身对应的包，和UClass前面类似。
- static UEnum* EMyEnum_StaticEnum()方法 一个返回该UEnum类型的函数。会在延迟注册时被调用。 { GetStaticEnum函数
  - 在静态初始化的时候会调用它，然后再嵌套调用Z_Construct
- template<> CPPLEARNING_API UEnum* StaticEnum<EMyEnum>(){ return EMyEnum_StaticEnum(); }
  - 调用它来实现
- static FCompiledInDeferEnum Z_CompiledInDeferEnum_UEnum_EMyEnum (EMyEnum_StaticEnum, TEXT("/Script/CppLearning"), TEXT("EMyEnum"), false, nullptr, nullptr); 延迟注册。

- FMetaDataPairParam 元数据对
- FEnumeratorParam 枚举项 { const FMetaDataPairParam* MetaDataArray; int32 NumMetaData;
  - 包含一个元数据数组
  - 包含一个元数据对
  - 包含枚举项数组
- FEnumParams { const FEnumeratorParam* EnumeratorParams; //枚举项数组 { const FMetaDataPairParam* MetaDataArray; int32 NumMetaData;

MyStruct.generated.h

- template<> CPPLEARNING_API UScriptStruct* StaticStruct<struct FMyStruct>();
- #define CURRENT_FILE_ID

实现

MyStruct.gen.cpp

- UPackage* Z_Construct_UPackage__Script_CppLearning(); 生成项目对应的包，不再赘述。
  - 作为参数传入
- 项目名_API UScriptStruct* Z_Construct_UScriptStruct_FMyStruct();
  - 作为参数传入
- class UScriptStruct* FMyStruct::StaticStruct() 静态获取UScriptStruct的方法。 { 用GetStaticStruct方法获取结构的单例
  - 转发给
- template<> CPPLEARNING_API UScriptStruct* StaticStruct<FMyStruct>(){ return FMyStruct::StaticStruct(); }对前面.h文件中的方法的实现
- static FCompiledInDeferStruct Z_CompiledInDeferStruct_UScriptStruct_FMyStruct (FMyStruct::StaticStruct, TEXT("/Script/CppLearning"), TEXT("MyStruct"), false, nullptr, nullptr); 延迟注册
- static struct FScriptStruct_CppLearning_StaticRegisterNativesFMyStruct 声明了一个静态的结构，里面只有一个构造函数——利用了静态函数的特性，在main函数之前进行初始化。因此程序一启动就会调用UScriptStruct::DeferCppStructOps向程序注册该结构的CPP信息（大小，内存对齐等）
  - 调用UScriptStruct::DeferCppStructOps函数 主要用来 动态获取结构体的构造和析构函数。用于在程序中使用。
    - TMap<FName,UScriptStruct::ICppStructOps*>& DeferredStructOps = GetDeferredCppStructOps(); 用于保存成方法以便可以通过动态的方式构造，析构等本机结构，以便与静态构造函数有关的问题
    - DeferredStructOps.Add(Target,InCppStructOps) 在这个结构名和对象的Map映射里登记 "Struct相应的C++操作类"
      - 在PrepareCppStructOps函数中被调用。这个函数的注释是："查找CppStructOps，如果我们没有它，并设置属性大小"而PrepareCppStructOps函数又会在Struct的默认构造函数UScriptStruct::UScriptStruct中被调用。
        - CppStructOps = GetDeferredCppStructOps().FindRef(GetFName()); 这个GetFName获得的自然是UScriptStruct的名字了。接下来会根据CppStructOps的各种字段（比如有没有序列化器之类的）来设置StructFlags的值。同时也会调用FMemory相关的一些函数，和自己的Constructor和Destruct函数，对自身的属性尺寸进行一些设置。说实话，这一部分内容，即使去看它们的实现，笔者依然不太清楚。
- UScriptStruct* Z_Construct_UScriptStruct_FMyStruct()函数 构造关联的UScriptStruct
  - 调用UE4CodeGen_Private::ConstructUScriptStruct()函数
    - ConstructUProperties 内部转发给ConstructFProperties，源码原理比较简单。一个400行的switch-case语句来判断属性的类型，然后根据类型不同调用不同的属性构造函数（比如FFieldPathProperty，FTextProperty之类的）。ReadMore变量针对的是容器类型。如果有进一步构造子属性的需求的话，会将ReadMore的值设定的更大（比如Set的ReadMore被设置为1）在switch语句结束后，函数的最后部分用循环将更多的需求递归构造完。
- struct Z_Construct_UScriptStruct_FMyStruct_Statics{ } 存储了该UStruct的信息的一个静态的结构变量。
  - static void* NewStructOps(); 模板来管理动态访问c++结构体的构造和销毁（注释翻译）
  - static const UE4CodeGen_Private::FFloatPropertyParams NewProp_Score; 我定义的float Score;在后文会对其初始化。
  - static const UE4CodeGen_Private::FUnsizedIntPropertyParams NewProp_SSSS; 我定义的 int SSSS;在后文会对其初始化。
  - static const UE4CodeGen_Private::FPropertyParamsBase* const PropPointers[]; 在后文会对其初始化。使用的是初始化了的SSSS和Score对其初始化。
  - static const UE4CodeGen_Private::FStructParams ReturnStructParams; 用于构造与注册的参数结构。
  - 作为参数传入
  - 对其初始化
- static const UE4CodeGen_Private::FFloatPropertyParams NewProp_Score = { UE4CodeGen_Private::EPropertyClass::Float, "Score", RF_Public|RF_Transient|RF_MarkAsNative, 0x0010000000000004, 1, nullptr, STRUCT_OFFSET(FMyStruct, Score), METADATA_PARAMS(NewProp_Score_MetaData, ARRAY_COUNT(NewProp_Score_MetaData)) }; 这就是Score的属性信息。其它的成员变量也是类似，不再赘述。

- FPropertyParamsBase
  - FPropertyParamsBaseWithOffset
  - FGenericPropertyParams 通用的属性参数
  - FObjectPropertyParams 对象引用类型属性参数
  - FDelegatePropertyParams 委托类型的属性参数
  - FMulticastDelegatePropertyParams 多播委托类型的属性参数
  - FEnumPropertyParams 枚举类型属性参数
- 被打上UPROPERTY宏的变量，根据其类型的不同，来使用不同的属性参数。比如float类型，使用的就是FFloatPropertyParams类型。在UStruct的gen.cpp文件中，会生成其结构中的各个成员的信息——FXXXPropertyParams。然后收集一个个属性的信息整合成数组，合并到结构参数(FStructParams)里去，最后传给ConstructUScriptStruct来构造。
- 生成属性信息。

- 元数据
  - FField() { GetMapForObject
  - Package { ConstructXXX { AddMetaData { SetValue