1. 引入Object基类的好处和缺点
    - 好处
    1. 有了统一基类Object之后，可以用一个Object指针追踪到所有派生对象。
    2. 通用的属性和接口。因为有继承机制，如果某个功能所有对象都需要，就可以在Object内加上。
    3. 便于GC。
    4. 统一的序列化模型。有一个基类的话序列化比较统一。
    5. 反射。如果没有一个统一的Object，你很难为各种对象实现GetType接口。（此处gettype怎么实现
    - 缺点
    1. 接口和属性太多然后并不是所有对象都全用得上。
    2. 多重继承。需要限制。避免菱形继承。
2. c++实现反射

    ue实现反射和qt中的类似，用宏做标记，然后用UHT分析生成generated.h/.cpp文件后再一起编译。

- 生成，收集，注册，链接

3. 代码生成

    虚幻头文件分析工具（UHT）是支持UObject系统的自定义解析和代码生成工具。

    UCLASS生成代码

```
#define BODY_MACRO_COMBINE_INNER(A,B,C,D) A##B##C##D
#define BODY_MACRO_COMBINE(A,B,C,D) BODY_MACRO_COMBINE_INNER(A,B,C,D)
#define GENERATED_BODY(...) BODY_MACRO_COMBINE(CURRENT_FILE_ID,_,__LINE__,_GENERATED_BODY)
```

GENERATED_BODY最终只是生成另一个宏的名称，此处的__LINE__是标准宏，CURRENT_FILE_ID定义在生成的generated.h里面。

`#define CURRENT_FILE_ID Hello_Source_Hello_MyClass_h`

而generated.h头文件中除了CURRENT_FILE_ID定义之外，就是两个GENERATED_BODY定义（字符连接之后的宏名），两个宏是构造函数是否自定义实现而产生的差别，如果MyClass类需要UMyClass(const FObjectInitializer& ObjectInitializer)的构造函数自定义实现，则需要用GENERATED_UCLASS_BODY宏来让最终生成的宏指向Hello_Source_Hello_MyClass_h_11_GENERATED_BODY_LEGACY（MyClass.generated.h的66行），其最终展开的内容会多一个构造函数的内容实现。

内部是四个宏定义：

```
#define Hello_Source_Hello_MyClass_h_11_GENERATED_BODY_LEGACY \ //两个重要的定义
PRAGMA_DISABLE_DEPRECATION_WARNINGS \
public: \
    Hello_Source_Hello_MyClass_h_11_PRIVATE_PROPERTY_OFFSET \
    Hello_Source_Hello_MyClass_h_11_RPC_WRAPPERS \
    Hello_Source_Hello_MyClass_h_11_INCLASS \
    Hello_Source_Hello_MyClass_h_11_STANDARD_CONSTRUCTORS \
public: \
PRAGMA_ENABLE_DEPRECATION_WARNINGS
```

- 倒数第一个宏

`Hello_Source_Hello_MyClass_h_11_ENHANCED_CONSTRUCTORS`

```
#define Hello_Source_Hello_MyClass_h_11_ENHANCED_CONSTRUCTORS \
    /** Standard constructor, called after all reflected properties have been initialized */ \
    NO_API UMyClass(const FObjectInitializer& ObjectInitializer = FObjectInitializer::Get()) : Super(ObjectInitializer) { }; \   //默认的构造函数实现
private: \  //禁止掉C++11的移动和拷贝构造
    /** Private move- and copy-constructors, should never be used */ \
    NO_API UMyClass(UMyClass&&); \
    NO_API UMyClass(const UMyClass&); \
public: \
    DECLARE_VTABLE_PTR_HELPER_CTOR(NO_API, UMyClass); \     //因为WITH_HOT_RELOAD_CTORS关闭，展开是空宏
    DEFINE_VTABLE_PTR_HELPER_CTOR_CALLER(UMyClass); \   //同理，空宏
    DEFINE_DEFAULT_OBJECT_INITIALIZER_CONSTRUCTOR_CALL(UMyClass)
```

此宏主要做的事是禁止移动和拷贝构造，并用`DEFINE_DEFAULT_OBJECT_INITIALIZER_CONSTRUCTOR_CALL`声明了一个构造函数包装器。

```
#define DEFINE_DEFAULT_OBJECT_INITIALIZER_CONSTRUCTOR_CALL(TClass) \
    static void __DefaultConstructor(const FObjectInitializer& X) { new((EInternal*)X.GetObj())TClass(X); }
```

原因是反射根据名字创造对象需要调用构造函数，需要函数指针，但是函数指针不能指向构造函数，因此这里包装一下。

因此可以如下方保存。

```
class COREUOBJECT_API UClass : public UStruct
...
{
    ...
    typedef void (*ClassConstructorType) (const FObjectInitializer&);
    ClassConstructorType ClassConstructor;
    ...
}
```

- 倒数第二个宏

```
#define Hello_Source_Hello_MyClass_h_11_INCLASS \
    private: \
    static void StaticRegisterNativesUMyClass(); \  //定义在cpp中，目前都是空实现
    friend HELLO_API class UClass* Z_Construct_UClass_UMyClass(); \ //一个构造该类UClass对象的辅助函数
    public: \
    DECLARE_CLASS(UMyClass, UObject, COMPILED_IN_FLAGS(0), 0, TEXT("/Script/Hello"), NO_API) \   //声明该类的一些通用基本函数
    DECLARE_SERIALIZER(UMyClass) \  //声明序列化函数
    /** Indicates whether the class is compiled into the engine */ \
    enum {IsIntrinsic=COMPILED_IN_INTRINSIC};   //这个标记指定了该类是C++Native类，不能动态再改变，跟蓝图里构造的动态类进行区分。
```

其中生命的`DECLARE_CLASS`声明较为重要，而DECLARE_CLASS内主要定义了类的自身的一些信息如父类为Super，自身为ThisClass，并提供获取信息的内敛函数。如下：

```
#define DECLARE_CLASS( TClass, TSuperClass, TStaticFlags, TStaticCastFlags, TPackage, TRequiredAPI  ) \
private: \
    TClass& operator=(TClass&&);   \
    TClass& operator=(const TClass&);   \
    TRequiredAPI static UClass* GetPrivateStaticClass(const TCHAR* Package); \
public: \
    /** Bitwise union of #EClassFlags pertaining to this class.*/ \
    enum {StaticClassFlags=TStaticFlags}; \
    /** Typedef for the base class ({{ typedef-type }}) */ \
    typedef TSuperClass Super;\
    /** Typedef for {{ typedef-type }}. */ \
    typedef TClass ThisClass;\
    /** Returns a UClass object representing this class at runtime */ \
    inline static UClass* StaticClass() \
    { \
        return GetPrivateStaticClass(TPackage); \
    } \
    /** Returns the StaticClassFlags for this class */ \
    inline static EClassCastFlags StaticClassCastFlags() \
    { \
        return TStaticCastFlags; \
    } \
    DEPRECATED(4.7, "operator new has been deprecated for UObjects - please use NewObject or NewNamedObject instead") \
    inline void* operator new( const size_t InSize, UObject* InOuter=(UObject*)GetTransientPackage(), FName InName=NAME_None, EObjectFlags InSetFlags=RF_NoFlags ) \
    { \
        return StaticAllocateObject( StaticClass(), InOuter, InName, InSetFlags ); \
    } \
    /** For internal use only; use StaticConstructObject() to create new objects. */ \
    inline void* operator new(const size_t InSize, EInternal InInternalOnly, UObject* InOuter = (UObject*)GetTransientPackage(), FName InName = NAME_None, EObjectFlags InSetFlags = RF_NoFlags) \
    { \
        return StaticAllocateObject(StaticClass(), InOuter, InName, InSetFlags); \
} \
    /** For internal use only; use StaticConstructObject() to create new objects. */ \
    inline void* operator new( const size_t InSize, EInternal* InMem ) \
    { \
        return (void*)InMem; \
    }
```

其中StaticClass()函数内部调用的GetPrivateStaticClass函数实现在对应的.cpp文件中。

```
#define IMPLEMENT_CLASS(TClass, TClassCrc) \
```

```
    static TClassCompiledInDefer<TClass> AutoInitialize##TClass(TEXT(#TClass), sizeof(TClass), TClassCrc); \ //延迟注册
    UClass* TClass::GetPrivateStaticClass(const TCHAR* Package) \ //.h里声明的实现，StaticClas()内部就是调用该函数
    { \
        static UClass* PrivateStaticClass = NULL; \ //又一次static lazy
        if (!PrivateStaticClass) \
        { \
            /* this could be handled with templates, but we want it external to avoid code bloat */ \
            GetPrivateStaticClassBody( \    //该函数就是真正创建UClass*,以后
                Package, \  //Package名字
                (TCHAR*)TEXT(#TClass) + 1 + ((StaticClassFlags & CLASS_Deprecated) ? 11 : 0), \//类名，+1去掉U、A、F前缀，+11去掉_Deprecated前缀
                PrivateStaticClass, \   //输出引用
                StaticRegisterNatives##TClass, \
                sizeof(TClass), \
                TClass::StaticClassFlags, \
                TClass::StaticClassCastFlags(), \
                TClass::StaticConfigName(), \
                (UClass::ClassConstructorType)InternalConstructor<TClass>, \
                (UClass::ClassVTableHelperCtorCallerType)InternalVTableHelperCtorCaller<TClass>, \
                &TClass::AddReferencedObjects, \
                &TClass::Super::StaticClass, \
                &TClass::WithinClass::StaticClass \
            ); \
        } \
        return PrivateStaticClass; \
    }
```

就是转调用，信息传给GetPrivateStaticClassBody函数。而.generated.cpp实现了很多构造函数，都是.h内声明的。GetPrivateStaticClassBody的实现如下：

可以看到这个最终的函数做了几件事：

1. 为传入的PrivateStaticClass分配内存
2. 在分配的内存上用placement new方法调用构造函数
3. 初始化UCLass对象
4. 注册本地函数，这就是注册部分的。

```
void GetPrivateStaticClassBody(
    const TCHAR* PackageName,
    const TCHAR* Name,
    UClass*& ReturnClass,
    void(*RegisterNativeFunc)(),
    uint32 InSize,
    EClassFlags InClassFlags,
    EClassCastFlags InClassCastFlags,
    const TCHAR* InConfigName,
    UClass::ClassConstructorType InClassConstructor,
    UClass::ClassVTableHelperCtorCallerType InClassVTableHelperCtorCaller,
```

```
    UClass::ClassAddReferencedObjectsType InClassAddReferencedObjects,
    UClass::StaticClassFunctionType InSuperClassFn,
    UClass::StaticClassFunctionType InWithinClassFn,
    bool bIsDynamic /*= false*/
    )
{
    ReturnClass = (UClass*)GUObjectAllocator.AllocateUObject(sizeof(UClass), alignof(UClass), true);//分配内存
    ReturnClass = ::new (ReturnClass)UClass //用placement new在内存上手动调用构造函数
    (

 EC_StaticConstructor,Name,InSize,InClassFlags,InClassCastFlags,InConfigName,
    EObjectFlags(RF_Public | RF_Standalone | RF_Transient | RF_MarkAsNative | RF_MarkAsRootSet),

 InClassConstructor,InClassVTableHelperCtorCaller,InClassAddReferencedObjects
    );

 InitializePrivateStaticClass(InSuperClassFn(),ReturnClass,InWithinClassFn(),PackageName,Name);//初始化UClass*对象
    RegisterNativeFunc();//注册Native函数到UClass中去
}
```

而InitializePrivateStaticClass内容如下：

```
COREUOBJECT_API void InitializePrivateStaticClass(
    class UClass* TClass_Super_StaticClass,
    class UClass* TClass_PrivateStaticClass,
    class UClass* TClass_WithinClass_StaticClass,
    const TCHAR* PackageName,
    const TCHAR* Name
    )
{
    //...
    if (TClass_Super_StaticClass != TClass_PrivateStaticClass)
    {
        TClass_PrivateStaticClass->SetSuperStruct(TClass_Super_StaticClass);    //设定类之间的SuperStruct
    }
    else
    {
        TClass_PrivateStaticClass->SetSuperStruct(NULL);    //UObject无基类
    }
    TClass_PrivateStaticClass->ClassWithin = TClass_WithinClass_StaticClass;    //设定Outer类类型
    //...
    TClass_PrivateStaticClass->Register(PackageName, Name); //转到UObjectBase::Register()
    //...
}
```

所做的事主要是最后一步的转发，UObjectBase::Register()，此函数对每个UClass*开始了注册。

```
struct FPendingRegistrantInfo
{
    const TCHAR*    Name;   //对象名字
    const TCHAR*    PackageName;    //所属包的名字
    static TMap<UObjectBase*, FPendingRegistrantInfo>& GetMap()
    {   //用对象指针做Key，这样才可以通过对象地址获得其名字信息，这个时候UClass对象本身其实还没有名字，要等之后的注册才能设置进去
        static TMap<UObjectBase*, FPendingRegistrantInfo> PendingRegistrantInfo;
        return PendingRegistrantInfo;
    }
};
//...
struct FPendingRegistrant
{
    UObjectBase*    Object; //对象指针，用该值去PendingRegistrants里查找名字。
    FPendingRegistrant* NextAutoRegister;   //链表下一个节点
};
static FPendingRegistrant* GFirstPendingRegistrant = NULL;  //全局链表头
static FPendingRegistrant* GLastPendingRegistrant = NULL;   //全局链表尾
//...
void UObjectBase::Register(const TCHAR* PackageName,const TCHAR* InName)
{
    //添加到全局单件Map里，用对象指针做Key，Value是对象的名字和所属包的名字。
    TMap<UObjectBase*, FPendingRegistrantInfo>& PendingRegistrants = FPendingRegistrantInfo::GetMap();
    PendingRegistrants.Add(this, FPendingRegistrantInfo(InName, PackageName));
    //添加到全局链表里，每个链表节点带着一个本对象指针，简单的链表添加操作。
    FPendingRegistrant* PendingRegistration = new FPendingRegistrant(this);
    if(GLastPendingRegistrant)
    {
        GLastPendingRegistrant->NextAutoRegister = PendingRegistration;
    }
    else
    {
        check(!GFirstPendingRegistrant);
        GFirstPendingRegistrant = PendingRegistration;
    }
    GLastPendingRegistrant = PendingRegistration;
}
```

记录的数据结构为一个Map加上一个链表，Map保证快速查找，链表是顺序结构用来辅助，因为有些情况需要遵循添加的顺序。

这里只是简单记录一下信息，并没有做实际的操作。因为此时还在static阶段，UObject对象分配索引什么的还没初始化好，无法进行实际操作。

如果添加函数：

```
    UFUNCTION(BlueprintCallable, Category = "Hello")
    void CallableFunc();    //C++实现，蓝图调用
```

generated.h文件中：

```
    DECLARE_FUNCTION(execCallableFunc) \    //声明供蓝图调用的函数
    { \
        P_FINISH; \
        P_NATIVE_BEGIN; \
        this->CallableFunc(); \
        P_NATIVE_END; \
    }
```

其中添加exec前缀供蓝图调用。

```
void execCallableFunc( FFrame& Stack, void*const Z_Param__Result )  //蓝图虚拟机的使用的函数接口
{
    Stack.Code += !!Stack.Code; /* increment the code ptr unless it is null */
    {
        FBlueprintEventTimer::FScopedNativeTimer ScopedNativeCallTimer; //蓝图的计时统计
        this->CallableFunc(); //调用我们自己的实现
    }
}
```

在.generated.cpp文件中

```
UFunction* Z_Construct_UFunction_UMyClass_CallableFunc()
    {
        UObject* Outer=Z_Construct_UClass_UMyClass();
        static UFunction* ReturnFunction = NULL;
        if (!ReturnFunction)
        {
            ReturnFunction = new(EC_InternalUseOnlyConstructor, Outer, TEXT("CallableFunc"), RF_Public|RF_Transient|RF_MarkAsNative) UFunction(FObjectInitializer(), NULL, 0x04020401, 65535); //FUNC_BlueprintCallable|FUNC_Public|FUNC_Native|FUNC_Final
            ReturnFunction->Bind();
            ReturnFunction->StaticLink();
#if WITH_METADATA
            UMetaData* MetaData = ReturnFunction->GetOutermost()->GetMetaData();
            MetaData->SetValue(ReturnFunction, TEXT("Category"), TEXT("Hello"));
            MetaData->SetValue(ReturnFunction, TEXT("ModuleRelativePath"), TEXT("MyClass.h"));
#endif
        }
        return ReturnFunction;
    }
```

蓝图也基本是转调用。

- 生成之后就需要信息的收集。

  最主要的使用的c++的静态注册模式来进行信息的收集。避免了手动添加。由于static在main函数之前初始化。

  ue里有两种格式采用这种注册模式：

```cpp
template <typename TClass>
struct TClassCompiledInDefer : public FFieldCompiledInInfo
{
    TClassCompiledInDefer(const TCHAR* InName, SIZE_T InClassSize, uint32 InCrc)
    : FFieldCompiledInInfo(InClassSize, InCrc)
    {
        UClassCompiledInDefer(this, InName, InClassSize, InCrc);//保存this来调用Register方法。
    }
    virtual UClass* Register() const override
    {
        return TClass::StaticClass();
    }
};

static TClassCompiledInDefer<TClass> AutoInitialize##TClass(TEXT(#TClass), sizeof(TClass), TClassCrc);
```

```cpp
struct FCompiledInDefer
{
    FCompiledInDefer(class UClass *(*InRegister)(), class UClass *(*InStaticClass)(), const TCHAR* Name, bool bDynamic, const TCHAR* DynamicPackageName = nullptr, const TCHAR* DynamicPathName = nullptr, void (*InInitSearchableValues)(TMap<FName, FName>&) = nullptr)
    {
        if (bDynamic)
        {

GetConvertedDynamicPackageNameToTypeName().Add(FName(DynamicPackageName), FName(Name));
        }
        UObjectCompiledInDefer(InRegister, InStaticClass, Name, bDynamic, DynamicPathName, InInitSearchableValues);
    }
};
static FCompiledInDefer Z_CompiledInDefer_UClass_UMyClass(Z_Construct_UClass_UMyClass, &UMyClass::StaticClass, TEXT("UMyClass"), false, nullptr, nullptr, nullptr);
```

static对象在每个生成的cpp文件中用以收集信息，将信息归拢到一起，这是在程序一开始的时候就开始去做的。

关于初始化的顺序的正确性：

无明确规定，由编译器确定。

解决方法：

1. 设计时不相互依赖。
2. 触发一个强制引用保证前置对象已完成。（引用完整性，参照完整性）

可以看到最终都是将信息添加到一个静态Array里面。

```
void UClassCompiledInDefer(FFieldCompiledInInfo* ClassInfo, const TCHAR* Name, SIZE_T ClassSize, uint32 Crc)
{
    //...
    // We will either create a new class or update the static class pointer of the existing one
    GetDeferredClassRegistration().Add(ClassInfo);  //static TArray<FFieldCompiledInInfo*> DeferredClassRegistration;
}
void UObjectCompiledInDefer(UClass *(*InRegister)(), UClass *(*InStaticClass)(), const TCHAR* Name, bool bDynamic, const TCHAR* DynamicPathName, void (*InInitSearchableValues)(TMap<FName, FName>&))
{
    //...
    GetDeferredCompiledInRegistration().Add(InRegister);    //static TArray<class UClass *(*)()> DeferredCompiledInRegistration;
}
```

对于UCLASS的收集：

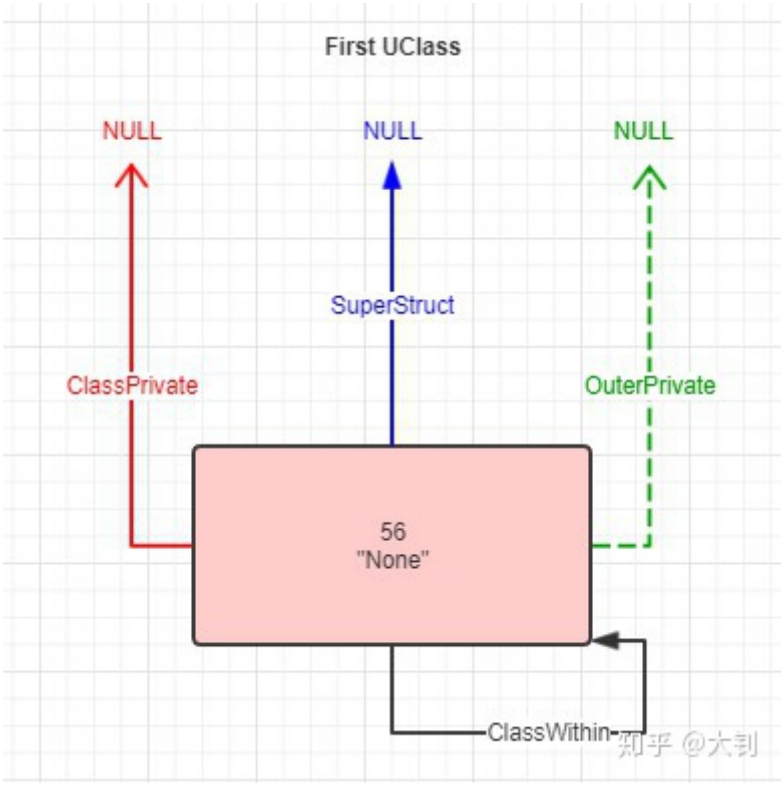
```
static TClassCompiledInDefer<UMyClass> AutoInitializeUMyClass(TEXT("UMyClass"), sizeof(UMyClass), 899540749);
//……
static FCompiledInDefer Z_CompiledInDefer_UClass_UMyClass(Z_Construct_UClass_UMyClass, &UMyClass::StaticClass, TEXT("UMyClass"), false, nullptr, nullptr, nullptr);
```

两种都用了，可以简单理解为前者是内存分配，后者继续进行一些构造和属性等注册。

延迟注册是为了用户的体验。

UEnum就只用了一个，比较简单。在static阶段会向内存注册一个构造UEnum*的函数指针用于回调：

Struct，Function所用的模式一样。



然后之后就进入main函数，主要分析的是coreUObject模块的加载，主要是一个个转发调用。

画一个流程图：

八九十

绑定链接：

将函数指针绑定到正确地址。

1. 内存构造。刚创建出来一块白纸一般的内存，简单的调用了UClass的构造函数。UE里一个对象的构造，构造函数的调用只是个起点而已。
2. 注册。给自己一个名字，把自己登记在对象系统中。这步是通过DeferredRegister而不是Register完成的。
3. 对象构造。往对象里填充属性、函数、接口和元数据的信息。这步我们应该记得是在gen.cpp里的那些函数完成的。
4. 绑定链接。属性函数都有了，把它们整理整理，尽量优化一下存储结构，为以后的使用提供更高性能和便利。
5. CDO创建。既然类型已经有了，那就万事俱备只差国家包分配一个类默认对象了。每个UClass都有一个CDO（Class Default Object），有了CDO，相当于有了一个存档备份和参照，其他对象就心不慌。
6. 引用记号流构建。一个Class是怎么样有可能引用其他别的对象的，这棵引用树怎么样构建的高效，也是GC中一个非常重要的话题。有了引用记号流，就可以对一个对象高效的分析它引用了其他多少对象。