UE4编码规范

Epic有着一些简单的代码编写标准和规则。本文档并非用于讨论或进行编辑，而是反映Epic前代码标准的状态。

代码规则对程序员来说至关重要，原因有下：

- 软件生命周期中80%的时间皆需要维护。
- 原开发者几乎不会对软件进行终身维护。
- 代码规则优化了软件的可读性，工程师能够更加快速地完全熟悉新代码。
- 如需向模组社区开发者公开源代码，则源代码需要易于理解。
- 交叉编译器兼容性实际上需要此类规则。

以下的代码标准以C++为中心，但无论使用何种语言，建议均按此标准执行。在适用情况下，相关章节可为指定语言提供同等规则或例外。

类排列

通常类均是由读取者进行排列，而非写入者。多数读取者会使用类的公共接口，因此首先需要对此进行声明，之后再声明类的私有实现。

版权声明

由Epic提供用于分配的源文件（.h、.cpp、.xaml、etc.）必须在文件的首行包含版权声明。版权声明的格式必须严格按照以下形式编写：

```
// Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
```

若此行缺失或格式错误，CIS将生成错误或失败提示。

命名规则

- 命名（如类型或变量）中的每个单词需大写首字母，单词间通常无下划线。例如：Health和UPrimitiveComponent，而不是lastMouseCoordinates或delta_coordinates。
- 类型名前缀需使用额外的大写字母，以将其和变量名进行区分。例如：FSkin为类型名，而Skin则是FSkin的实例。
    - 模板类的前缀为T，如TArray。
    - 继承自UObject的类前缀为U。
    - 继承自AActor的类前缀为U。
    - 继承自SWidget的类前缀为S。
    - 抽象界面类的前缀为I。

    - 列举的前缀为E。
    - 布尔变量必须以b为前缀（如“PendingDestruction”，获得“bHasFadedIn”）。
    - 其他多数类均以F为前缀，而部分子系统则以其他字母为前缀。
    - ypedefs应以任何与其类型相符的字母为前缀：若为结构体的Typedefs，则使用F；若为UObject的Typedefs，则使用U，以此类推。
        - 特别模板实例化的Typedef不再是模板，并应加上相应前缀，例如：

```
typedef TArray<FMyType> FArrayOfMyTypes;
```

    - C#中省略前缀。 多数情况下，UnrealHeaderTool需要正确的前缀，因此添加前缀至关重要。
- 类型和变量的命名为名词。
- 方法名是动词，以描述方法的效果或未被方法影响的返回值。

变量、方法和类的命名应清楚、明了且进行描述。命名的范围越大，一个良好的描述性命名就越重要。避免过度缩写。

所有变量应逐个声明，以便对变量的含义提供注释。其同样被JavaDocs格式需要。变量前可使用多行或单行注释，空白行为分组变量可选使用。

所有返回布尔的函数应发起true/false的询问，如IsVisible()或ShouldClearBuffer()。

程序（无返回值的函数）应在Object后使用强变化动词。一个例外是若方法的Object是其所在的Object；此时需以上下文来理解Object。避免以“Handle”和“Process”为开头；此类动词会引起歧义。

若函数参数通过引用传递，同时该值会写入函数，建议以“Out”做为函数参数命名的前缀（非必需）。此操作将明确表明传入该参数的数值将被函数替换。

若In或Out参数同样为布尔，以b作为In/Out的前缀，如“bOutResult”。

返回值的函数应描述返回的值；命名应说明函数将返回的值。此规则对布尔函数极为重要。请参考以下两个范例方法：

```
bool CheckTea(FTea Tea) {...} // true的意义是什么？
bool IsTeaFresh(FTea Tea) {...} // 命名明确说明茶是新鲜的
```

范例

```
float TeaWeight;

int32 TeaCount;

bool bDoesTeaStink;

FName TeaName;

FString TeaFriendlyName;

UClass* TeaClass;
```

```
USoundCue* TeaSound;

UTexture* TeaTexture;
```

基础C++类型的可移植别名

- bool代表布尔值（不会假定布尔尺寸）。布尔不会进行编译。
- TCHAR代表字符（不会假定TCHAR尺寸)。
- uint8代表无符号字节（1字节）。
- int8代表带符号的字节（1字节）。
- uint16代表无符号"短"字符（2字节）。
- int16代表带符号的"短"字符（2字节）。
- uint32代表无符号的整数（4字节）。
- int32代表带符号的整数（4字节）。
- uint64代表无符号"四字"（8字节）。
- int64代表带符号的"四字"（8字节）。
- float代表单精确浮点（4字节）。
- double代表双精确浮点（8字节）。
- PTRINT单表可能含有指针的整数（不会假定PTRINT尺寸）。

若代码中的整数宽度非重要，可使用C++ +的int和无符号int类型（不同的平台上的大小不同）。明确尺寸的类型必须以序列化或复制的格式进行使用。

注解

注解即沟通，而沟通极为重要。请牢记以下几点有关注解的信息（来自Kernighan & Pike所著一书 *《编程实践（The Practice of Programming）》*）：

指南

- 编写自描述代码：

```
// 差：
t = s + l - b;

// 优：
TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

- 编写有用注解：

```
// 差：
// increment Leaves
++Leaves;

// 优：
// we know there is another tea leaf
++Leaves;
```

- 不对较差代码进行注解——而重新编写：

```
// 差：
// total number of leaves is sum of
// small and large leaves less the
// number of leaves that are both
t = s + l - b;

// 优：
TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

- 不要编写悖论代码：

```
// 差：
// never increment Leaves!
++Leaves;

// 优：
// we know there is another tea leaf
++Leaves;
```

常量正确性

常量即是文档也是编译器指令，因此应保证所有代码正确。

其中包括：若函数不修改参数，常量指针或引用将传递函数参数；若方法不修改对象，将方法标记为常量；若循环不修改容器，则在容器上使用常量迭代。

```
// SomeMutatingOperation不会修改InArray，但可能会修改OutResult
void SomeMutatingOperation(FThing& OutResult, const TArray<int32>& InArray);

void FThing::SomeNonMutatingOperation() const
{
    // 若此代码在FThing上被调用，其不会修改FThing
}

    TArray<FString> StringArray;
    for (const FString& : StringArray)
    {
        // 此循环的主体不会修改StringArray
    }
```

基于值的函数参数和本地值通用倾向常量。读取者可了解函数主体不会修改变量，使之更为易懂。因此操作影响JavaDoc进程，进行此操作时须确保声明与定义相匹配：

```
    void AddSomeThings(const int32 Count);

    void AddSomeThings(const int32 Count)
    {
        const int32 CountPlusOne = Count + 1;

        // 函数主体不会改变Count或CountPlusOne
    }
```

此操作有一列外：按值传递参数最终将被移入容器（参加"移动语意"）；此情况非常罕见。

```
    void FBlah::SetMemberArray(TArray<FString> InNewArray)
    {
        MemberArray = MoveTemp(InNewArray);
    }
```

创建指针的常量时，将常量关键字放在末尾（而非指针所指之处）。无法"重新指定"引用，因此无法以相同方式创建常量：

```
    // 非常量对象的常量指针--无法重新指定指针，但仍可修改T
    T* const Ptr = ...;

    // 非法
    T& const Ref = ...;
```

绝不要在返回类型上使用常量，此操作将会禁止复杂类型的移动语意，并会对内置类型发出编译警告。此规则仅适用于返回类型自身，而非指针目标类型或返回的引用：

```
    // 差 - 返回常量数组
    const TArray<FString> GetSomeArray();

    // 优 - 将引用返回至常量数组
    const TArray<FString>& GetSomeArray();

    // 优 - 将指针返回至常量数组
    const TArray<FString>* GetSomeArray();

    // 差 - 将常量指针返回至常量数组
    const TArray<FString>* const GetSomeArray();
```

范例格式

通常适用基于JavaDoc的系统从代码和版本文档中自动提取注释，因此需要遵守几项特别的注释格式规则。

以下例子展示的是类、状态、方法和变量注释的格式。记住：注释应扩大代码。代码记载实现，而注释记载目的。修改代码的部分目的后应及时更新注释。

注意：支持两种不同的参数注释格式，具体为Steep和Sweeten方法。Steep使用的是传统@参数格式，如Sweeten范例所示，对简单函数而言Steep方式可更加清晰地将参数文档整合到函数的描述性注释中。

方法注释仅能在其公开声明的地方使用一次。方法注释仅包含与方法调用者有关的信息，包括任何可能与调用者有关的方法覆盖信息。与调用者无关的方法实现和覆盖应在方法实现中进行注释。

```
    class IDrinkable
    {
    public:
        /**
         * 玩家喝下此物时调用。
         * @参数 OutFocusMultiplier - 返回时将包含应用至喝下者聚焦的乘数。
         * @参数 OutThirstQuenchingFraction - 返回时将包含喝下者渴度的冷却小数（0-1）。
         */
        virtual void Drink(float& OutFocusMultiplier, float& OutThirstQuenchingFraction) =
0;
    };

    class FTea : public IDrinkable
    {
    public:
        /**
         * 根据茶叶浸泡时的水的体积和温度，计算出茶叶的差量食味值。
         * @参数 VolumeOfWater - 用于冲泡的水量（以毫升计量）
         * @参数 TemperatureOfWater - 水温（以开氏度计量）
         * @参数 OutNewPotency - 开始浸泡后茶的效力（0.97到1.04）
         * @返回茶每分钟的强度变化（以茶食味值TTU计量）
         */
        float Steep(
            const float VolumeOfWater,
            const float TemperatureOfWater,
            float& OutNewPotency
            );

        void Sweeten(const float EquivalentGramsOfSucrose);

        float GetPrice() const
        {
            return Price;
        }

        virtual void Drink(float& OutFocusMultiplier, float& OutThirstQuenchingFraction)
override;

    private:

        float Price;

        float Sweetness;
    };
```

```
    float FTea::Steep(const float VolumeOfWater, const float TemperatureOfWater, float&
OutNewPotency)
    {
        ...
    }

    void FTea::Sweeten(const float EquivalentGramsOfSucrose)
    {
        ...
    }

    void FTea::Drink(float& OutFocusMultiplier, float& OutThirstQuenchingFraction)
    {
        ...
    }
```

类注释包括了什么？

- 此类解决的问题的描述。创建此类的原因。

方法注释的所有部分的意义为何？

- 首先是函数目的：记载 *该函数解决的问题。*如上所述，注释记载 *目的*而代码记载 *实现。*
- 然后是参数注释：所有参数注释应包括计量单位、预期数值范围、“无可能”数值，以及状态/错误代码的意义。
- 最后是返回注释：其记载预期返回值的方式与记载输出变量相同。

C++11和现代语言语法

虚幻引擎的设计使起可带规模移植到众多C++编译器中，因此需谨慎使用与可能支持的编译器兼容的功能。有些功能十分有用，因此大家习惯将其包裹在宏中进行大范围使用。但建议在可能支持的编译器更新至最新标准前不要进行类似操作。

目前所使用的C++11部分语言功能在现代编译器上普遍有着很好的支持，例如基于范围、移动语意和匿名函数。某些情况下，可将此类功能包裹到预处理条件语句中进行使用（例如容器中的优右值引用）。但某些语言功能会导致新平台无法消化语法，因此需要避免使用此类功能。

除非在以下指定为可支持的现代C++编译器功能，否则在指定编译器的语言功能被包裹到预处理宏或条件语句并节制使用前，应避免使用指定编译器的语言功能。

static_asset

需要编译时断言时，此关键字有效。

override与final

推荐使用此类关键字。文中可能遗漏了此内容，但在之后会进行补充。

nullptr

所用情况下均使用nullptr，而非C-style NULL宏。

此情况有一例外：C++/CX版本（如Xbox One）中的nullptr实际上为已管理的空引用类型。其主要与原声C++中的nullptr兼容（与其类型相同的和部分模板实例化情景除外），因此应使用TYPE_OF_NULLPTR宏进行兼容，而非更为常用的decltype（nullptr）。

“auto”关键字

不应在C++代码中使用自动模式，但以下范例除外。必须时刻清楚正在初始化的类型。其意味着读取者必须明确可见此类型。此规则同样适用于C#中“var”关键字的使用。

什么时候可以使用自动模式？

- 需要将匿名函数与变量绑定时，可使用自动模式。因为代码中无法表达匿名函数类型。
- 仅迭代器类型十分冗长且会损坏可读性时，适用于迭代函数。
- 无法清楚识别表达式的模板代码中适用。此为高阶情况。

类型应对读取代码者清晰可见，这一点至关重要。部分IDE能够推断类型，但此操作需要代码处于编译状态。其同样不会协助用户进行合并/对比；在隔离中查看GitHub等单个源文件时，则可协助合并/对比。

若正以合适方式使用自动模式，请牢记使用正确常量，以及与类型明明相同的&或*。在自动模式的协助下，此操作将使推断类型强制为所需的任何事物。

基于范围

可更易理解代码，也更易进行维护。使用老旧TMap迭代器迁移代码时，请注意老旧的Key()和Value()函数之前是迭代器类型的方法，而现在只是下方键值TPair的键和值域：

```
    TMap<FString, int32> MyMap;

    // 旧格式
    for (auto It = MyMap.CreateIterator(); It; ++It)
    {
        UE_LOG(LogCategory, Log, TEXT("Key: %s, Value: %d"), It.Key(), *It.Value());
    }

    // 新格式
    for (TPair<FString, int32>& Kvp : MyMap)
    {
        UE_LOG(LogCategory, Log, TEXT("Key: %s, Value: %d"), Kvp.Key, *Kvp.Value);
    }
```

部分独立迭代器类型拥有替代范围：

```
    // 旧格式
    for (TFieldIterator<UProperty> PropertyIt(InStruct, EFieldIteratorFlags::IncludeSuper);
PropertyIt; ++PropertyIt)
    {
        UProperty* Property = *PropertyIt;
        UE_LOG(LogCategory, Log, TEXT("Property name: %s"), *Property->GetName());
    }

    // 新格式
    for (UProperty* Property : TFieldRange<UProperty>(InStruct,
EFieldIteratorFlags::IncludeSuper))
    {
        UE_LOG(LogCategory, Log, TEXT("Property name: %s"), *Property->GetName());
    }
```

匿名函数

现在可在所有编译器上使用匿名函数，但使用时需考虑几项事务。最佳的匿名函数长度不应超过两条语句，作为较大表达式或语句的一部分时尤为如此。例如作为泛型算法中的谓词时：

```
    // 查找名字含有“Hello”一次的事物
    Thing* HelloThing = ArrayOfThings.FindByPredicate([](const Thing& Th){ return
Th.GetName().Contains(TEXT("Hello")); });

    // 以命名倒序排列阵列
    AnotherArray.Sort([](const Thing& Lhs, const Thing& Rhs){ return Lhs.GetName() >
Rhs.GetName(); });
```

请注意：常用的状态性匿名函数无法指定至函数指针。

记载非浅显匿名函数时应使用普通函数相同的记载方法。建议将其分为数行，以便添加注释。

对于大型匿名函数和延迟执行来说，倾向使用显式捕捉而非自动捕捉（[&]和[=]）。使用错误的捕捉语意意外捕捉到变量可能会造成负面结果，这种情况在维护代码时更易发生：

- 若在捕捉的变量情景外执行匿名函数，指针的通过引用捕捉和通过值捕捉会意外造成悬空引用：

```
void Func()
{
    int32 Value = GetSomeValue();

    // 诸多代码

    AsyncTask([&]()
    {
        // 此处数值无效
        for (int Index = 0; Index != Value; ++Index)
        {
            // ...
        }
```

```
    });
}
```

- 通过值捕捉含有不必要的复制，因此其可能会影响性能：

```
void Func()
{
    int32 ValueToFind = GetValueToFind();

    // 匿名函数意外被[=]捕捉（本应只捕捉ValueToFind），因此其会复制一个ArrayOfThings的副本
    FThing* Found = ArrayOfThings.FindByPredicate(
        [=](const FThing& Thing)
        {
            return Thing.Value == ValueToFind && Thing.Index < ArrayOfThings.Num();
        }
    );
}
```

- 意外捕捉的UObject指针在垃圾回收器中不可见：

```
void Func(AActor* MyActor)
{
    // 捕捉MyActor指针不会禁止对象被收集
    AsyncTask([=]()
    {
        MyActor->DoSlowThing();
    });
}
```

- 若引用任何成员变量，即使使用[=]，自动捕捉也会固定隐式捕捉“此”。[=]可使未含有自身副本的匿名函数的表达式产生副本：

```
void FStruct::Func()
{
    int32 Local = 5;
    Member = 5;

    auto Lambda = [=]()
    {
        UE_LOG(LogTest, Log, TEXT("Local: %d, Member: %d"), Local, Member);
    };

    Local = 100;
    Member = 100;

    Lambda(); // Logs "Local: 5, Member: 100"
}
```

大型匿名函数或返回另一函数调用的结果时，倾向显式返回类型。此类操作与“自动”关键字相同：

```
// 此处无返回类型，返回类型不清楚
auto Lambda = []() -> FMyType
{
    return SomeFunc();
};
```

浅显匿名函数接受自动捕捉和隐式返回类型。例如，排序调用中的语意明显且为显式，将使其过度冗长——请自行判断。

强类型化列举

对于普通列举和UENUM，列举类应固定作为旧格式命名空间列举的替换使用。例如：

```
// 旧列举
UENUM()
namespace EThing
{
    enum Type
    {
        Thing1,
        Thing2
    };
}

// 新列举
UENUM()
enum class EThing : uint8
{
    Thing1,
    Thing2
};
```

若UPROPERTY基于uinit8，则同样支持UPROPERTY——此操作替换旧的 `TEnumAsByte<>` 解决方案：

```
// 旧属性
UPROPERTY()
TEnumAsByte<EThing::Type> MyProperty;

// 新属性
UPROPERTY()
EThing MyProperty;
```

作为标签使用的列举类可利用ENUM_CLASS_FLAGS(EnumType)宏来自动定义所有按位运算符：

```
enum class EFlags
{
    None  = 0x00,
    Flag1 = 0x01,
    Flag2 = 0x02,
    Flag3 = 0x04
};

ENUM_CLASS_FLAGS(EFlags)
```

有一例外："真实"情景中标签的使用——此为语言的局限。相反，所有标签列举应含有名为"空"的列举器，并将其设为0以进行对比：

```
// 旧
if (Flags & EFlags::Flag1)

// 新
if ((Flags & EFlags::Flag1) != EFlags::None)
```

移动语意

TArray、TMap、TSet、FString等所有主要容器类型含有移动构造函数与移动赋值运算符。通过值传递/返回这些类型时，通常自动使用这些容器，但使用MoveTemp可对其进行显式调用，其等同于UE4的std::move。

通过值返回容器或字符串适用于表达式，而无需临时副本的通常开销。通过值和MoveTemp的使用之规则仍在制定，但已可在基础代码的部分已优化区域中被找到。

默认成员初始器

默认成员初始器可用于定义类自身中默认类：

```
UCLASS()
class UTeaOptions : public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY()
    int32 MaximumNumberOfCupsPerDay = 10;

    UPROPERTY()
    float CupWidth = 11.5f;

    UPROPERTY()
    FString TeaType = TEXT("Earl Grey");

    UPROPERTY()
    EDrinkingStyle DrinkingStyle = EDrinkingStyle::PinkyExtended;
```

```
    };
```

按此编写的代码有以下优势：

- 其不需要在多个构造函数之间重复初始器
- 不能混合初始化顺序和声明顺序。
- 成员类型、属性标签和默认值均在一处，便于阅读及维护。

其也存在以下几类劣势：

- 需要重新编译依赖文件才能修改默认设置。
- 引擎的补丁中无法修改标头，此格式将限制修复的类型。
- 无法以此方式初始化所有对象，例如基类、UObject子对象、前置声明类型的指针、构造函数参数的推断值、多步骤初始化成员等。
- 标头中含有一些初始器，其余则在.cpp文件的构造函数中，可能会影响可读性和可维护性。

请谨慎考虑是否使用。根据经验而言，默认成员初始器更适用于游戏代码而非引擎代码。同样也可考虑使用默认值的配置文件。

第三方代码

修改引擎中使用的库的代码时，请务必以//@UE4注释标记变更，以及修改理由。此操作能够将变更以更简便的方式合并到库中，使注册者可轻松查找到做出的变更。

引擎中的第三方代码需标有注释，该注释应可被轻易查找。例如：

```
    // @third party code - BEGIN PhysX
    #include <PhysX.h>
    // @third party code - END PhysX

    // @third party code - BEGIN MSDN SetThreadName
    // [http://msdn.microsoft.com/en-us/library/xcb2z8hs.aspx]
    // 用于在调试器中设置线程命名
    ...
    //@third party code - END MSDN SetThreadName
```

代码格式

大括号{ }

大括号格式必须一致。在Epic的传统做法中，大括号固定被放在新行。请保持这种格式。

固定在单语句块中使用大括号，例如：

```
if (bThing)
{
    return;
}
```

If - Else

if-else语句中的每个执行块都应该使用大括号。此举是为防止编辑时出错——未使用大括号时，可能会意外地将另一行加入if块中。这会造成if表达式无法控制该行，使之成为较差代码；更糟糕的情况是条件编译的项目会导致if/else语句中断。所以请务必使用大括号。

```
if (HaveUnrealLicense)
{
    InsertYourGameHere();
}
else
{
    CallMarkRein();
}
```

若多向if语句的缩进量与首条if语句的缩进量相同，则应互相缩进。此操作可提高结构体的可读性：

```
if (TannicAcid < 10)
{
    UE_LOG(LogCategory, Log, TEXT("Low Acid"));
}
else if (TannicAcid < 100)
{
    UE_LOG(LogCategory, Log, TEXT("Medium Acid"));
}
else
{
    UE_LOG(LogCategory, Log, TEXT("High Acid"));
}
```

制表符

- 通过执行块缩进代码。
- 在行的起始使用制表符（tab），而非空格。将制表符（tab）设为4字符。注意：有时需要使用空格，以便忽略制表符（tab）的空格数保持代码对齐。例如：以无制表符（tab）字符对齐代码。
- 若在C#中编写代码，请同样使用制表符（tab）字符，而非空格。因为程序员时常在C#和C++间切换，且更倾向使用制表符（tab）的统一设置。Visual Studio默认在C#文件中使用空格，因此在编写虚幻引擎代码时请变更此设置。

切换语句

除空白条件外（拥有相同代码的多个选择），切换条件语句应显式标注条件将会落入另一条件。每个条件包含中断或落入注释。其他代码控制传输命令（返回、继续等）同样适用。

固定设有默认条件，其中包含有中断——以防在默认后添加新的条件。

```
    switch (condition)
    {
        case 1:
            ...
            // falls through
        case 2:
            ...
            break;
        case 3:
            ...
            return;
        case 4:
        case 5:
            ...
            break;
        default:
            break;
    }
```

命名空间

可在合适之处使用命名空间组织类、函数和变量，需要遵循以下规则。

- 虚幻代码目前尚未包裹在整体命名空间中。需要在整体作用域中注意碰撞，特别是在连入第三方代码时。
- 请勿将“适用”声明加入整体作用域中，也不能放入.cpp文件中（其会导致“统一”构建系统出错。）
- 可将“使用”放入其他命名空间或函数中体中。
- 注意：若将“使用”放入命名空间，其会将其他命名空间带入同一翻译单元中。只需保持一致即可。
- 若遵守以上规则，可仅在标头文件中使用“使用”。
- 注意：前置声明需要在各自命名空间中进行声明，否则将造成链接错误。
- 若在命名空间中声明过多类/类型，将导致在其他整体作用域中使用这些类型时出现困难。（例如出现在类声明时，函数签名需要使用显式命名空间。）
- 可使用“使用”为命名空间中的变量在作用域中命名别称（例如（使用Foo::FBar）。但在虚幻代码中并不常用。
- UnrealHeaderTool不支持命名空间。定义UCLASS、USTRUCT等时，请勿使用。

物理依赖性

- 若可能，文件名不应添加前缀。例如应为Scene.cpp，而非UnScene.cpp。此操作可通过减少消除文件歧义所需字母数，实现在解决方案中使用工作区Whiz或Visual Assist的打开文件等工具。
- 设置指令后所有标头应防止使用含有#pragma once等多种格式。注意：需要使用的编译器现在均支持使用#pragma once。

```
#pragma once

<file contents>
```

- 通常需要最小化物理耦合。
- 若可使用前置声明，而非标头，请使用前置声明。
- 尽可能地进行细粒化包含；勿包含Core.h，在核心中包含需要定义的特定标头。
- 直接包含全部所需标头，以便进行细粒化包含。
- 请勿依赖被包含的其他标头间接包含的标头。
- 请勿依赖利用其他标头进行包含；包含所需的全部对象。
- 模块含有私有和公开源目录。其他模块所需定义必须在公开目录的标头中，而剩余的需在私有目录中。注意：在较老的虚幻模块中，此类目录可能被称为Src和Inc，但此类目录用于区分私有和公开代码的方式则相同，同时不会将标头文件和源文件进行区分。
- 可设定用于生成预编译标头的标头。使用UnrealBuildTool的效率更高。
- 将大型函数拆分为逻辑子函数。编译器优化一方面是消除常用的子表达式，而函数越大，编译器进行辨识的工作量就越大，导致编译时间大大增长。
- 内联函数会强制在不需要它们的文件中重新编译，请谨慎使用。内联仅可在浅显访问器中和分析显示有益时使用。
- 使用FORCEINLINE时，需更加谨慎。所有代码和本地变量将扩展至调用函数中，将导致与大型函数相同的编译时间问题。

封装

使用保护关键字强制进行封装。除非其是类的公开/保护接口的一部分，否则应固定奖类成员声明为私有。请谨慎使用，但缺少存取器时，在不破坏插件和现有项目的情况下，重构插件将非常困难。

若特定域仅能通过派生类使用，将其设置为私有并提供受保护的存取器。

若类并非用于派生，则使用完成。

一般格式问题

- 最小化依赖性距离。代码基于有特定值的变量时，在使用该变量时设定其的值。在执行块顶部初始化变量，且不将其用于一百行代码，将可能导致在未意识到依赖性的情况下意外地更改值。将其设在下行中，可明确变量初始化的原因及其工作区域。
- 若可能，将方法拆分为子方法。人类更善于首先纵观全局，再深入查看感兴趣的细节，而不会以细节入手，最后重构全局。相较于包含子方法全部代码的同等方法，若以此方法，理解调用多个命名优良的子方法序列的简易方法会更为容易。 在函数声明或函数调用站中，请勿在函数命名和设参数列表为优先的空括号间添加空格。
- 修复编译器警告。出现编译器警告消息意味着某些项目有错。修复编译器警告的内容。若无法修复，使用#pragma压制警告；这是最后的补救办法。
- 在文件末尾留下空白行。所有.cpp和.h文件应包含空白行，以便和gcc兼容。

- 勿使浮点隐式转换为int32。此操作十分缓慢，且无法再所有编译器中进行编译。请固定使用appTrunc()函数转换为int32。此操作可在保证交叉编译器兼容性的情况下，同时生成更快代码。
- 接口类（前缀为"I"）应固定为抽象，且必须未含有成员变量。接口可包含非纯虚幻方法，只要方法已实现内联，即可包括非虚幻或静态方法。
- 调试代码需为有用并经过优化，或为已迁入。与其他代码相互混杂的调试代码将导致读取其他代码时出现困难。
- 在字符串文字周围固定使用TEXT()宏。若未使用，在文字中构建FStrings的代码将导致不理想的字符转换过程。
- 避免循环相同的多余运算。将常用子表达式从循环中提升出来，以避免冗余计算。在某些情况下，使用静态来避免函数调用间的整体多余运算。例如：在字符串文字中构建FName。
- 注意热重载。最小化依赖性来减少迭代时间。无使用可能会在重载时发生改变的函数内联或模板。仅对在重载时保持不变的对象使用静态。
- 使用中间变量来简化复杂表达式。若含有复杂表达式，将其拆分为分配至中间变量的子表达式将更易理解（该子表达式的的命名描述了其在父表达式中的意义）。例如：

```
if ((Blah->BlahP->WindowExists->Etc && Stuff) &&
    !(bPlayerExists && bGameStarted && bPlayerStillHasPawn &&
    IsTuesday()))
{
    DoSomething();
}
```

应替换为

```
const bool bIsLegalWindow = Blah->BlahP->WindowExists->Etc && Stuff;
const bool bIsPlayerDead = bPlayerExists && bGameStarted && bPlayerStillHasPawn &&
IsTuesday();
if (bIsLegalWindow && !bIsPlayerDead)
{
    DoSomething();
}
```

- 声明覆盖方法时，使用虚幻和覆盖关键字。 若在派生类（此类在父类中覆盖虚幻函数）中声明虚幻函数，必须同时使用虚幻和覆盖关键字。例如：

```
class A
{
public:
    virtual void F() {}
};
class B : public A
{
public:
    virtual void F() override;
};
```

注意：由于最近新添了覆盖关键字，许多现有代码并不遵循此规则。需在方便时将覆盖关键字添加至该代码。

- 指针与引用应仅含一个空格，该空格位于指针/引用右侧。在文件中可更易查找指向特定类型的所有指针或引用。

  使用：

  ```
  FShaderType* Type
  ```

  不使用：

  ```
  FShaderType *Type
  FShaderType * Type
  ```

- 不允许遮蔽变量。C++可在外部作用域遮蔽变量，会导致语意不清。举例而言，该成语函数中含有三个可用“计算”变量：

  ```
  class FSomeClass
  {
  public:
      void Func(const int32 Count)
      {
          for (int32 Count = 0; Count != 10; ++Count)
          {
              // Use Count
          }
      }

  private:
      int32 Count;
  };
  ```