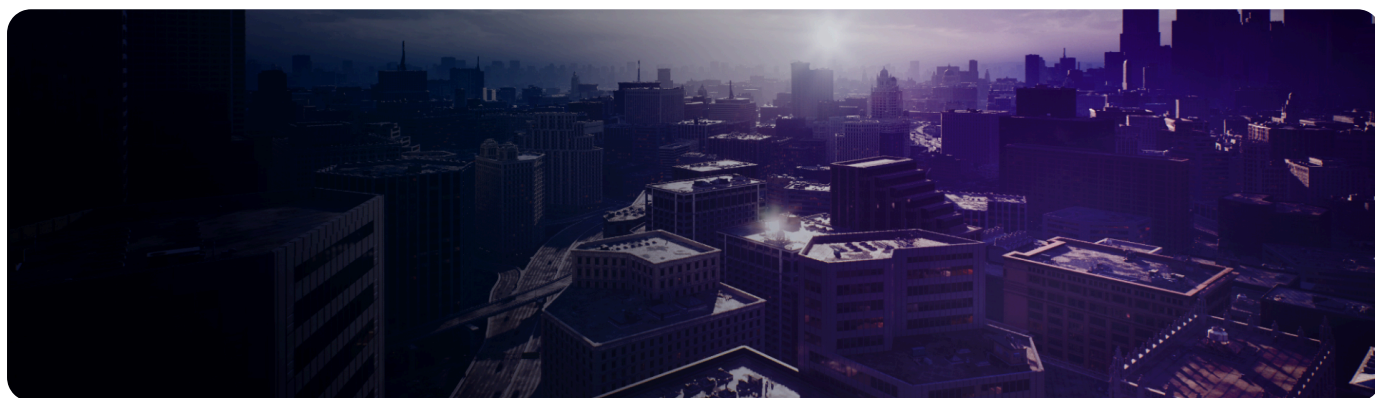


# 代码规范

通过遵守既定标准和最佳实践，编写可维护的代码。



在Epic内部，我们会遵循一些基本的代码标准和规范。本文并非旨在探讨当前的某项工作，而是介绍Epic目前采用的代码规范。下文阐述了我们会严格遵守的一些代码规范。

代码规范对程序员十分重要，原因有几点：

- 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。
- 代码规范可提高软件可读性，让工程师更加快速透彻地理解新代码。
- 如决定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。

## 类组织

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：

```
1 // Copyright Epic Games, Inc. All Rights Reserved.  
2
```

 复制完整片段

若此行缺失或格式错误，CIS将生成错误或失败提示。

## 命名规范

- 所有代码和注释都应采用美式标准英语的拼写和语法。
- 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`，而非`lastMouseCoordinates`或`delta_coordinates`。
- 类型名前缀需使用额外的大写字母，用于区分其和变量命名。例如：`FSkin`为类型名，而`Skin`则是`FSkin`的实例。
  - 模板类的前缀为T。
  - 继承自`UObject`的类前缀为U。
  - 继承自`AActor`的类前缀为A。
  - 继承自`SWidget`的类前缀为S。
  - 抽象界面类的前缀为I。
  - Epic提供的概念类型的类（用作`TModels`的第一个参数），其前缀为C。
  - 枚举的前缀为E。
  - 布尔变量必须以b为前缀（例如`bPendingDestruction`或`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。
  - Typedefs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：

```
1 typedef TArray<FMytype> FArrayOfMyTypes;  
2
```

 复制完整片段

- 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。
- 类型模板参数和基于这些模板参数的嵌套类型别名不受上述前缀规则的约束，因为类型的类别是未知的。
  - 一般在描述性术语后添加一个Type后缀。
  - 通过使用In前缀，将模板参数与别名区分开来：

```

1  template <typename InElementType>
2  class TContainer
3  {
4  public:
5  using ElementType = InElementType;
6  };
7

```

 复制完整片段

- 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。
- 宏应该全部大写，用下划线隔开，并以 `UE_` 作为前缀（参见命名空间）。

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：

```

1  // True的意义是什么？

```

```
2 bool CheckTea(FTea Tea);
3
4 // 命名明确说明茶是新鲜的
5 bool IsTeaFresh(FTea Tea);
6
```

 复制完整片段

## 示例

```
1 float TeaWeight;
2 int32 TeaCount;
3 bool bDoesTeaStink;
4 FName TeaName;
5 FString TeaFriendlyName;
6 UClass* TeaClass;
7 USoundCue* TeaSound;
8 UTexture* TeaTexture;
9
```

 复制完整片段

## 包容性选词

当你编写虚幻引擎代码库时，用语应始终尊重他人，表现出包容性和专业水准。

为内容命名时，这一原则均适用，如类、函数、数据结构、类型、变量、文件和文件夹、插件，等等。为UI、错误消息和通知编写面向用户的文本片段时，该原则适用。在注释和变更列表说明等中编写 关于 (*About*) 代码的内容时也适用。

下述小节提供了一些指导建议，可帮助你在选择措辞和名称时保存足够的敬意，并符合各种情景和受众，同时能更有效地传达信息。

## 种族、民族和宗教包容性

- 禁止使用会强化刻板印象的暗喻或明喻。

这包括将黑白对立的措辞，例如 *blacklist* / *whitelist* 。

- 禁止在措辞中提及历史创伤或亲身体验的歧视。

这包括 *slave* 、 *master* 和 *nuke* 。

## 性别包容性

- 使用代词 *they* 、 *them* 和 *their* 来指称假定当事人，即使是单数也不例外。
- 指称非人事物时，总是使用 *it* 和 *its* 。例如：模块、插件、函数、客户端、服务器或其他软件或硬件组件。

不要给无性别的事物指定性别。

- 禁止使用诸如 *guys* 之类呈现性别的集体名词。
- 避免诸如 *poor\_man* 之类包含专断性别的口语短语。

## 俚语

- 请记住，你的措辞会向全球受众呈现，他们熟悉的成语可能不同，对同一件事物有不同的态度，对同一个文化的理解也可能不同。
- 避免俚语和俗语，即使你认为很滑稽或无伤大雅。这些用语对于母语非英语的人来说可能很难理解，也不太好翻译。
- 禁止使用脏话。

## 义项太多的词语

- 对于许多术语，我们只取用了其技术含义，但这些词在非技术语境还有其他用法。例如：*abort* 、 *execute* 或 *native* 。在使用此类词语时，务必精确表达意思，并检查相应上下文。

## 词语表

以下列表标出了一些术语，我们过去在虚幻代码库中使用过这些术语，但我们认为现在应该用更好的词加以替换。

## 黑名单

替代词： *deny list* 、 *block list* 、 *exclude list* 、 *avoid list* 、 *unapproved list* 、 *forbidden list* 、 *permission list*

## Whitelist


替代词: *allow list*、*include list*、*trust list*、*safe list*、*prefer list*、*approved list*、*permission list*

## Master

替代词: *primary*、*source*、*controller*、*template*、*reference*、*main*、*leader*、*original*、*base*

## Slave

替代词: *secondary*、*replica*、*agent*、*follower*、*worker*、*cluster node*、*locked*、*linked*、*synchronized*

 我们正积极致力于使我们的代码符合上述原则。

## 可移植的C++代码

- `bool` 代表布尔值（不会假定布尔尺寸）。`BOOL` 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` 代表带符号字节（1字节）。
- `uint16` 代表无符号"短"字符（2字节）。
- `int16` 代表带符号"短"字符（2字节）。
- `uint32` 代表无符号整数（4字节）。
- `int32` 代表带符号整数（4字节）。
- `uint64` 代表无符号"四字"（8字节）。
- `int64` 代表带符号"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` 代表可能含有指针的整数（不会假定PTRINT尺寸）。

在某些情况下，整型长度的影响并不重要，这时你可以使用C++中的 `int` 类型和无符号 `int` 类型；这些类型的长度会随平台而变化，但至少能保证32位长度。在涉及序列化或复制类型的格式时，仍然必须使用长度明确的类型。

## 标准库的使用

过去，UE避免直接使用C和C++标准库。造成这种情况的原因有很多种，其中包括：用我们自己的实现替代低效缓慢的实现、允许对内存分配进行额外控制、在可用的新功能广泛出现前先自己“丰衣足食”、进行合适但非标准化的行为更改、在代码库中使用一致的语法、避免采用与UE习惯用法不相容的结构。但是，近年来，随着标准库愈发稳定和成熟，它包含了许多我们自己不想用抽象层封装或重新实现的功能。

假如希望使用之前从未用过的全新标准库组件，则应将这类情况提交给代码规范小组进行评估。这还有助于我们把组件添加到白名单后，保持白名单为最新状态。

当可以在标准库和我们自己的库之间选择时，请选择能提供更优结果的方案，但请记住，一致性也非常重要。如果某个长期使用的UE实现无法再满足需求，我们就可以选择弃用它，并改用标准库。

应该避免在同一API中混合使用UE规范和标准库规范。

`<atomic>`：应在新代码中使用，在迁移旧代码时也应该使用。原子性（Atomic）将在所有受支持平台上高效推广。我们自己的 `TAtomic` 仅实现了部分功能，对我们而言，对它继续进行维护和改善已没有太大意义。

`<type_traits>`：应在旧版UE特性（trait）和标准特性重叠的地方使用。特性（trait）通常作为编译器内在函数（compiler intrinsic）实现，从而确保正确性，且编译器可以了解标准特性并选择更快的编译路径，而不必将其视为纯C++。值得关注的是，我们的特性通常具有大写的 `Value` "static"关键字或 `Type` "typedef"关键字，而标准特性则使用 `value` 和 `type`。这是一项重要区别，因为组合特性需要特定语法，如 `std::conjunction`。我们添加的新特性应使用小写的 `value` 或 `type`，以便支持组合，而现有特性应更新，以便支持任一情况。

`<initializer_list>`：必须用于支持初始化器（initializer）语法（用括号括起来）。这种情况下，语言和标准库会产生重叠，如果你想支持这种情况，则没有其他方案。

`<regex>`：可以直接使用，但其使用应封装在仅和编辑器有关的代码中。我们没有计划实现自己的正则表达式（regex）解决方案。

`<limits>`： `std::numeric_limits` 可以完整使用。

`<cmath>`：这个头文件中只有浮点比较函数可以使用。

`<cstring>`：如果能产生明显的性能改进，建议使用 `memcpy()` 和 `memset()` 而不是 `FMemory::Memcpy` 和 `FMemory::Memset`。

除了Interop作代码外，应避免使用标准容器和字符串。

## 注释

注释即沟通，而沟通极为重要。请牢记以下几点有关注释的信息（摘取自Kernighan & Pike的著作《*The Practice of Programming*》）：

## 指南

- 编写含义清晰的代码：

```
1 // 错误示范:
2 t = s + l - b;
3
4 // 正确示范:
5 TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

 复制完整片段

- 编写有用的注释：

```
1 // 错误示范:
2 // 增加叶子
3 ++Leaves;
4
5 // 正确示范:
6 //我们知道还有另一种茶叶
7 ++Leaves;
```

 复制完整片段

- 不要对低质量代码进行注释——重新编写这些代码：

```
1 // 错误示范:
2 // 大叶子和小叶子之和
3 // 减去兼具两者的叶子数量
4 // 之后得出的总数
```



```
5 t = s + l - b;  
6  
7 // 正确示范:  
8 TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

 复制完整片段

- 不要让代码与注释自相矛盾:

```
1 // 错误示范:  
2 // 永远不要增加叶子!  
3 ++Leaves;  
4  
5 // 正确示范:  
6 // 我们知道还有另一种茶叶  
7 ++Leaves;
```

 复制完整片段

## 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的常量正确。

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。

范例：

```
1 void SomeMutatingOperation(FThing& OutResult, const TArray<Int32>&  
   InArray)  
2 {  
3 // 此处不会修改InArray, 但可能会修改OutResult  
4 }  
5  
6 void FThing::SomeNonMutatingOperation() const  
7 {  
8 // 若此代码在FThing上被调用, 其不会修改FThing  
9 }  
10  
11 TArray<FString> StringArray;
```

```
12 for (const FString& :StringArray)
13 {
14 // 此循环的主体不会修改StringArray
15 }
16
```

 复制完整片段

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。操作影响JavaDoc进程，因此进行此操作时须确保声明与定义相匹配：

范例：

```
1 void AddSomeThings(const int32 Count);
2
3 void AddSomeThings(const int32 Count)
4 {
5 const int32 CountPlusOne = Count + 1;
6 // 函数主体不会改变Count或CountPlusOne
7 }
8
```

 复制完整片段

一个例外是按值传递参数，这种参数最终会被移动到容器中（见"移动语义"），不过这种参数应该很少见。

范例：

```
1 void FBlah::SetMemberArray(TArray<FString> InNewArray)
2 {
3 MemberArray = MoveTemp(InNewArray);
4 }
5
```

 复制完整片段

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：

范例：

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

```
3
4 // 非法
5 T& const Ref = ...;
6
```

 复制完整片段

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义，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。

范例：

```
1 // 差 - 返回常量数组
2 const TArray<FString> GetSomeArray();
3
4 // 优 - 返回常量数组的引用
5 const TArray<FString>& GetSomeArray();
6
7 // 优 - 返回指向常量数组的指针
8 const TArray<FString>* GetSomeArray();
9
10 // 差 - 返回指向常量数组的常量指针
11 const TArray<FString>* const GetSomeArray();
12
```

 复制完整片段

## 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**、**方法** 和 **变量** 注释的格式。记住：注释应扩大代码。代码说明实现，而注释说明意图。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 `@param` 多行格式，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和值文档整合到函数的描述性注释中。如 `@see` 或 `@return` 等特殊注释标签仅用于在主要注释后另起新行。

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的相关细节应在方法实现

中进行注释。

```
1  /** 可饮用物的接口。*/
2  class IDrinkable
3  {
4  public:
5  /**
6   * 玩家喝下此物时调用。
7   * @param OutFocusMultiplier - 返回时将包含应用至喝下者聚焦的乘数。
8   * @param OutThirstQuenchingFraction - 返回时将包含喝下者渴度的冷却小数 (0-1) 。
9   * @warning 仅在饮料准备好后调用此。
10  */
11  virtual void Drink(float& OutFocusMultiplier, float&
    OutThirstQuenchingFraction) = 0;
12  };
13
14  /** 单杯茶。*/
15  class FTea : public IDrinkable
16  {
17  public:
18  /**
19   * 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。
20   * @param VolumeOfWater - 用于冲泡的水量 (以毫升计)
21   * @param TemperatureOfWater - 水温 (以开氏度计)
22   * @param OutNewPotency - 开始浸泡后茶的效力 (0.97到1.04)
23   * @return 茶每分钟的强度变化 (以茶食味值TTU计)
24   */
25  float Steep(
26  const float VolumeOfWater,
27  const float TemperatureOfWater,
28  float& OutNewPotency
29  );
30
31  /** 向茶添加甜料, 以产生相同甜度的蔗糖克数计量。*/
32  void Sweeten(const float EquivalentGramsOfSucrose);
33
34  /** 日本销售的茶量 (以日元计) 。*/
35  float GetPrice() const
36  {
37  return Price;
38  }
39
40  virtual void Drink(float& OutFocusMultiplier, float&
    OutThirstQuenchingFraction) override;
```

```

41
42 private:
43 /** 价格 (日元) */
44 float Price;
45
46 /** 当前甜度, 以同等蔗糖克数计量 */
47 float Sweetness;
48 };
49
50 float FTea::Steep(const float VolumeOfWater, const float
    TemperatureOfWater, float& OutNewPotency)
51 {
52 ...
53 }
54
55 void FTea::Sweeten(const float EquivalentGramsOfSucrose)
56 {
57 ...
58 }
59
60 void FTea::Drink(float& OutFocusMultiplier, float&
    OutThirstQuenchingFraction)
61 {
62 ...
63 }
64

```

 复制完整片段

## 类注释包括什么？

- 此类解决的问题的描述。
- 创建此类的原因。

## 多行法注释的所有部分的意义为何？

1. **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。
2. **参数注释**：所有参数注释应包括：
  - 计量单位，
  - 预期值范围，
  - "无可能"值
  - 以及状态/错误代码的含义。
3. **返回注释**：其记载预期返回值的方式与记载输出变量相同。为了避免冗余，如函数的唯一目的返回此值，且已函数目的中进行记载，则不应使用显式@return注释。

4. **其他信息**：可选择使用 `@warning`、`@note`、`@see` 和 `@deprecated` 记载额外相关信息。此类注释应在其他注释后单列一行声明。

## 现代C++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。

编译虚幻引擎所需的最低语言版本是C++17，而且我们使用了许多现代语言特性，这些特性在现代编译器中得到了良好的支持。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（例如容器中的右值引用）。不过，出于移植或其他目的，有时我们会避免使用某些语言特性。

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，不应使用指定编译器的语言功能。

## 静态\_断言

需要编译时断言时，此关键字有效。

## 覆盖与完成

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 `__cplusplus < 201103L ? TYPE_OF_NULLPTR : decltype(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 `auto` 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。

C++17的结构化绑定功能也不应该使用，因为它本质上是 `auto` 。

什么时候可以使用 `auto` ？

- 需要将匿名函数与变量绑定时。因为代码中无法表达匿名函数类型。
- 仅迭代器类型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，或在隔离中查看GitHub等单个源文件时的用户

若正以合适方式使用 `自动` 模式，请牢记使用正确常量，以及与类型命名相同的`&`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

利用此可更易理解代码，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：

范例：

```
1 TMap<FString, int32> MyMap;
2
3 // 旧样式
4 for (auto It = MyMap.CreateIterator(); It; ++It)
5 {
6     UE_LOG(LogCategory, Log, TEXT("Key:%s, Value:%d"), It.Key(), *It.Value());
7 }
8
9 // 新样式
10 for (TPair<FString, int32>& Kvp :MyMap)
11 {
12     UE_LOG(LogCategory, Log, TEXT("Key:%s, Value:%d"), *Kvp.Key, Kvp.Value);
13 }
14
```

 复制完整片段

部分独立迭代器类型拥有替代范围。

范例：

```

1 // 旧样式
2 for (TFieldIterator<UPROPERTY> PropertyIt(InStruct,
    EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
3 {
4 UPROPERTY* Property = *PropertyIt;
5 UE_LOG(LogCategory, Log, TEXT("Property name:%s"), *Property->GetName());
6 }
7
8 // 新样式
9 for (UPROPERTY* Property :TFieldRange<UPROPERTY>(InStruct,
    EFieldIteratorFlags::IncludeSuper))
10 {
11 UE_LOG(LogCategory, Log, TEXT("Property name: %s"), *Property->GetName());
12 }
13

```

 复制完整片段

## 匿名函数

可以自由使用匿名函数。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。

范例：

```

1 // 查找名字含有"Hello"一次的事物
2 Thing* HelloThing = ArrayOfThings.FindByPredicate([](const Thing& Th){
    return Th.GetName().Contains(TEXT("Hello")); });
3
4 // 以命名倒序排列阵列
5 Algo::Sort(ArrayOfThings, [](const Thing& Lhs, const Thing& Rh){ return
    Lhs.GetName() > Rh.GetName(); });
6

```

 复制完整片段

注意：常用的状态性匿名函数无法指定至函数指针。

非浅显匿名函数应使用正则函数相同的方式进行记载。建议将其分为数行，以便添加注释。

应使用显式捕捉，而非自动捕捉（`[&]` 和 `[=]`）。对于可读性、可维护性和性能而言，特别是使用大量匿名函数和延迟执行时，此操作非常重要。其可声明作者的目的，以便在检查代码时找



出错误。错误的捕捉会造成负面结果，这种情况在随时间维护代码时更易发生。

- 若匿名函数的执行被延迟，指针（包括 `此` 指针）的通过引用捕捉和通过值捕捉会意外造成悬空引用
- 若创建非延迟匿名函数非必需副本，通过值捕捉将影响性能。
- 意外捕捉的UObject指针对垃圾回收器是不可见的。如引用成员变量，即使 `[=]` 可使匿名函数产生包含自身信息副本的错觉，自动捕捉也将隐式捕捉 `此`。

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

```
1 // 此处无返回类型，返回类型不清楚
2 auto Lambda = []() -> FMyType
3 {
4     return SomeFunc();
5 }
6
```

 复制完整片段

浅显非延迟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。

可使用C++14中的捕捉初始器功能：

```
1 TSharedPtr<FThing> ThingPtr = MakeUnique<FThing>();
2 AsyncTask([UniquePtr = MoveTemp(UniquePtr)]())
3 {
4     // 此处使用UniquePtr
5     });
6
```

 复制完整片段

## 强类型化枚举

对于普通枚举和 `UENUM`，枚举类应固定作为旧格式命名空间枚举的替换使用。例如：

```
1 // 旧枚举
2 UENUM()
3 namespace EThing
4 {
```

```

5 enum Type
6 {
7   Thing1,
8   Thing2
9 };
10 }
11
12 // 新枚举
13 UENUM()
14 enum class EThing : uint8
15 {
16   Thing1,
17   Thing2
18 }
19

```

 复制完整片段

此操作同时支持 `UPROPERTY`，作为替换旧的 `TEnumAsByte<>` 解决方案：枚举属性可为任何大小，而非仅为字节大小：

```

1 // 旧属性
2 UPROPERTY()
3 TEnumAsByte<EThing::Type> MyProperty;
4
5 // 新属性
6 UPROPERTY()
7 EThing MyProperty;
8

```

 复制完整片段

但公开到蓝图的枚举依旧须基于 `uint8`。

用作标志的枚举类可以利用 `ENUM_CLASS_FLAGS(EnumType)` 宏自动定义所有按位运算符：

```

1 enum class EFlags
2 {
3   None = 0x00,
4   Flag1 = 0x01,
5   Flag2 = 0x02,
6   Flag3 = 0x04
7 };
8

```

```
9  ENUM_CLASS_FLAGS(EFlags)
```

```
10
```

 复制完整片段

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

```
1 // 旧
2 if (Flags & EFlags::Flag1)
3
4 // 新
5 if ((Flags & EFlags::Flag1) != EFlags::None)
6
```

 复制完整片段

## 移动语意 (Move Semantics)

`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。通过值传递和 `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：

```
1 UCLASS()
2 class UTeaOptions : public UObject
3 {
4     GENERATED_BODY()
5
6     public:
7     UPROPERTY()
8     int32 MaximumNumberOfCupsPerDay = 10;
9
10    UPROPERTY()
11    float CupWidth = 11.5f;
```

```
12
13 UPROPERTY()
14 FString TeaType = TEXT("Earl Grey");
15
16 UPROPERTY()
17 EDrinkingStyle DrinkingStyle = EDrinkingStyle::PinkyExtended;
18 };
19
```

 复制完整片段

按此编写的代码有以下优势：

- 无需跨多个构造函数重复初始器。
- 不会打乱初始化顺序和声明顺序。
- 成员类型、属性标志和默认值均在一处，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头文件，此格式将限制修复的类型。
- 部分对象无法以此方式格式化，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。
- 如将部分初始器放入头文件，而其余在.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：

```
1 // @third party code - BEGIN PhysX
2 #include <physx.h>
3 // @third party code - END PhysX
4 // @third party code - BEGIN MSDN SetThreadName
5 // [http://msdn.microsoft.com/en-us/library/xc2z8hs.aspx]
6 // 用于在调试器中设置线程命名
7 ...
```

```
8 //@third party code - END MSDN SetThreadName
9
```

 复制完整片段

# 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。请遵循此格式。

固定在单语句块中使用大括号。例如：

```
1 if (bThing)
2 {
3 return;
4 }
5
```

 复制完整片段

## If - Else

if-else语句中的所有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。多余行不受if表达式控制，会成为较差代码。条件编译的项目导致if/else语句中断时，也会造成不良结果。因此务必使用大括号。

```
1 if (bHaveUnrealLicense)
2 {
3 InsertYourGameHere();
4 }
5 else
6 {
7 CallMarkRein();
8 }
9
```

 复制完整片段

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：

```
1  if (TannicAcid < 10)
2  {
3      UE_LOG(LogCategory, Log, TEXT("Low Acid"));
4  }
5  else if (TannicAcid < 100)
6  {
7      UE_LOG(LogCategory, Log, TEXT("Medium Acid"));
8  }
9  else
10 {
11     UE_LOG(LogCategory, Log, TEXT("High Acid"));
12 }
13
```

 复制完整片段

## 制表符和缩进

以下为代码缩进的标准。

- 通过执行块缩进代码。
- 在行的起始使用制表符，而非空格将制表符设为4字符。有时则需要使用空格，以便忽略制表符的空格数保持代码对齐。例如：以无制表符字符对齐代码。
- 若在C#中编写代码，请同样使用制表符，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。各条件应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。

```
1  switch (condition)
2  {
3      case 1:
```

```
4 ...
5 // 落入
6
7 case 2:
8 ...
9 break;
10
11 case 3:
12 ...
13 return;
14
15 case 4:
16 case 5:
17 ...
18 break;
19
20 default:
21 break;
22 }
23
```

 复制完整片段

## 命名空间

可在合适之处使用命名空间组织类、函数和变量，使用时需要遵循以下规则。

- 大多数UE代码目前尚未包裹在整体命名空间中。。在整体作用域中避免碰撞，特别是在使用或包括第三方代码时。
- UnrealHeaderTool不支持命名空间，所以命名空间不应在定义 `UCLASSES`、`USTRUCTS` 等时使用。
- 假如新的API不属于 `UCLASSES`、`USTRUCTS` 等，则至少应该位于 `UE::` 命名空间中，并且最好位于嵌套的命名空间中，例如 `UE::Audio::`。假如某个命名空间保管着实现细节，并且该实现细节不属于面向公众的API，则应位于 `Private` 命名空间中，例如 `UE::Audio::Private::`。
- `使用` 声明：
  - 请勿将 `使用` 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。
  - 若将 `使用` 声明放入命名空间，其会带入同一翻译单元中的其他命名空间。只需保持一致即可。

- 只有遵守以上规则，才能在头文件中使用 `using` 声明。
- 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 `using` 仅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。
- 宏无法存在于命名空间中，但应使用 `UE_` 作为前缀，例如 `UE_LOG`。

## 物理依赖性

- 文件名应尽量不添加前缀。例如使用 `Scene.cpp`，而非 `UnScene.cpp`。此操作可通过减少用于标识文件的字母数，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。
- 设置指令后所有头文件应防止使用含有 `#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`。

```
1 #pragma once
2 //<file contents>
```

 复制完整片段

- 通常需要最小化物理耦合。特别是，避免包含来自其他头文件的标准库头文件。
- 若可使用前置声明，而非头文件，请使用前置声明。
- 包含时尽量细粒化。例如，勿包含Core.h，而在核心中包含需要定义的特定头文件。
- 直接包含全部所需头文件，以便进行细粒化包含。
- 请勿依赖被包含的其他头文件间接包含的头文件。
- 请勿依赖利用其他头文件进行包含。应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头文件中。剩余的定义需在私有目录中。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，同时不会将头文件文件和源文件进行区分。
- 不必再为了预编译头文件生成（precompiled header generation）相关的头文件设置而担心。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。从而导致编译时间大大增长。
- 内联函数会强制在不使用其的文件中强行编译，因此勿使用过多内联函数。内联函数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，否则应固定讲类成员声明为私有。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。
- 尽量将方法拆分为子方法。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。此类操作更加容易。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出错。应修复编译器警告的内容。如无法修复，使用 `#pragma` 压制警告，此为最后补救办法。
- 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。
- 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移出，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對象使用静态。
- 使用中间变量来简化复杂表达式。若含有复杂表达式，将其拆分为指定至中间变量的子表达式将更易理解（该子表达式的命名描述了其在父表达式中的意义）。例如：

```
1 if ((Blah->BlahP->WindowExists->Etc && Stuff) &&
```

```
2 !(bPlayerExists && bGameStarted && bPlayerStillHasPawn &&
3 IsTuesday()))))
4 {
5 DoSomething();
6 }
7
```

 复制完整片段

应该替换成

```
1 const bool bIsLegalWindow = Blah->BlahP->WindowExists->Etc && Stuff;
2 const bool bIsPlayerDead = bPlayerExists && bGameStarted &&
  bPlayerStillHasPawn && IsTuesday();
3 if (bIsLegalWindow && !bIsPlayerDead)
4 {
5 DoSomething();
6 }
```

 复制完整片段

- 指针与引用应仅含一个空格，该空格位于指针/引用右侧。使用 **在文件中查找** 可方便快速地找到特定类型的所有指针和引用。
- 使用：

```
1 FShaderType* Ptr
2
```

 复制完整片段

*Not these:*

```
1 FShaderType *Ptr
2 FShaderType * Ptr
```

 复制完整片段

- 不允许隐藏变量。C++可在外部作用域隐藏变量，但此操作会导致语意不清。例如，该成员函数中含有三个可用 **计算** 变量：

```
1 class FSomeClass
2 {
3 public:
```

```

4 void Func(const int32 Count)
5 {
6     for (int32 Count = 0; Count != 10; ++Count)
7     {
8         // 使用计算
9     }
10 }
11
12 private:
13     int32 Count;
14 }

```

 复制完整片段

- 避免在函数调用中使用匿名文字。建议使用描述其含义的命名常量：

```

1 // 旧样式
2 Trigger(TEXT("Soldier"), 5, true);
3
4 // 新样式
5 const FName ObjectName = TEXT("Soldier");
6 const float CooldownInSeconds = 5;
7 const bool bVulnerableDuringCooldown = true;
8 Trigger(ObjectName, CooldownInSeconds, bVulnerableDuringCooldown);
9

```

 复制完整片段

由于无需查找函数声明即可理解目的，因此此操作可协助普通读者快速理解。

- 请避免在头文件中定义无关重要的静态变量，因为这会导致在每一个包含该头文件的转译单元中编译出一个实例：

```

1 // SomeModule.h
2 static const FString GUsefulNamedString = TEXT("String");
3

```

 复制完整片段

应该替换为

```

1 // SomeModule.h
2 extern SOMEMODULE_API const FString GUsefulNamedString;
3
4 // SomeModule.cpp

```

```
5 const FString GUsefulNamedString = TEXT("String");
```

 复制完整片段

## API设计指导方针

- 应避免使用 **布尔** 函数参数，切勿用于传递到函数的标签。其也拥有与前文提到匿名文字问题，但API利用更多行为扩展时，此类问题将成倍增加。相反，应优先使用枚举（参见[强类型化枚举](#)章节中将枚举用作标签的建议）：

```
1 // 旧样式
2 FCup* MakeCupOfTea(FTea* Tea, bool bAddSugar = false, bool bAddMilk =
   false, bool bAddHoney = false, bool bAddLemon = false);
3 FCup* Cup = MakeCupOfTea(Tea, false, true, true);
4
5 // 新样式
6 enum class ETeaFlags
7 {
8     None,
9     Milk = 0x01,
10    Sugar = 0x02,
11    Honey = 0x04,
12    Lemon = 0x08
13 };
14 ENUM_CLASS_FLAGS(ETeaFlags)
15
16 FCup* MakeCupOfTea(FTea* Tea, ETeaFlags Flags = ETeaFlags::None);
17 FCup* Cup = MakeCupOfTea(Tea, ETeaFlags::Milk | ETeaFlags::Honey);
18
```

 复制完整片段

此类形式可防止意外置换标签，避免意外在指针和整数参数中转换，而无需重复多余默认值，从而变得更加高效。将 **布尔** 传递到类似setter（例如 void FWidget::SetEnabled(bool bEnabled)）的函数完整状态时，可使用 **布尔** 作为参数。如此设置发生变化，则使用重构。

- 避免过长的函数参数列表。如函数使用多个参数，则选择传递专属结构体：

```
1 // 旧样式
2 TUniquePtr<FCup[]> MakeTeaForParty(const FTeaFlags* TeaPreferences,
   uint32 NumCupsToMake, FKettle* Kettle, ETeaType TeaType =
   ETeaType::EnglishBreakfast, float BrewingTimeInSeconds = 120.0f);
```

```

3
4 // 新样式
5 struct FTeaPartyParams
6 {
7     const FTeaFlags* TeaPreferences = nullptr;
8     uint32 NumCupsToMake = 0;
9     FKettle* Kettle = nullptr;
10    ETeaType TeaType = ETeaType::EnglishBreakfast;
11    float BrewingTimeInSeconds = 120.0f;
12 };
13 TUniquePtr<FCup[]> MakeTeaForParty(const FTeaPartyParams& Params);

```

 复制完整片段

- 避免使用 `布尔` 和 `Fstring` 重载函数，此操作可能导致意外行为：

```

1 void Func(const FString& String);
2 void Func(bool bBool);
3
4 Func(TEXT("String")); // 调用布尔重载!

```

 复制完整片段

- 接口类（前缀为"I"）固定为抽象，切不可拥有成员变量。只要接口为内联实现，其可包含非纯虚拟方法，甚至非虚拟或静态方法。
- 声明覆盖方法时应使用 `虚拟` 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 `虚拟` 和 `覆盖` 关键字。例如：

```

1 class A
2 {
3 public:
4     virtual void F() {}
5 };
6
7 class B : public A
8 {
9 public:
10    virtual void F() override;
11 }
12

```

 复制完整片段



由于最近新添 `覆盖` 关键字，许多现有代码并未遵守此规则。需在方便时将 `覆盖` 关键字添加至该代码。

## 平台特定代码

应固定抽取平台特定代码，并在正确命名的子目录中平台特定源文件内实现，例如：

```
1 Engine/Platforms/[PLATFORM]/Source/Runtime/Core/Private/[PLATFORM]PlatformM
2
```

复制完整片段

通常应避免在名为[PLATFORM]的目录外，将 `PLATFORM_[PLATFORM]` 的使用（例如 `PLATFORM_XBOXONE`）添加到代码。

相反，应扩展硬件抽象层，以添加静态函数，例如在FPlatformMisc中：

```
1 FORCEINLINE static int32 GetMaxPathLength()
2 {
3     return 128;
4 }
5
```

复制完整片段

之后，平台可覆盖此函数，返回平台特定常量值，或使用平台API决定结果。

如强制内联该函数，其将具有和使用定义时同样的性能特征。

必须需使用定义时，应新建描述可应用到函数的特定属性的#define，例如

`PLATFORM_USE_PTHREADS`。在Platform.h中设置默认值，并覆盖平台（平台特定Platform.h文件中需要）的默认值。

例如，在Platform.h中有：

```
1 #ifndef PLATFORM_USE_PTHREADS
2 #define PLATFORM_USE_PTHREADS 1
3 #endif
```

 复制完整片段

Windows/WindowsPlatform.h有：

```
1 #define PLATFORM_USE_PTHREADS 0
2
```

 复制完整片段

跨平台代码可直接使用该定义，无需知晓平台。

```
1 #if PLATFORM_USE_PTHREADS
2 #include "HAL/PThreadRunnableThread.h"
3 #endif
4
```

 复制完整片段

理由：将引擎的平台特定细节集中化，平台特定源文件中可完全包含此类细节。此操作可便于维护多个平台的引擎，还可将代码移植到新平台，而无需在基本代码中查找平台特定定义。

PS4、XboxOne和Nintendo Switch等NDA平台要求将平台代码保存在平台特定文件夹中。

无论 `[PLATFORM]` 子目录是否显示，应确保编译并运行代码。换言之，跨平台代码不可依赖平台特定代码。