UE4 工程规范

专业术语：

Levels/Maps(关卡/地图)

Map(地图)这个词通常也会被称为 Level(关卡)，两者的含义是等同的，在这里可以查看这个词的发展经历。

Cases(大小写)

对于字母大小写的规范有数种，以下是几种常见的规范。

・PascalCase

每个单词的首字母都是大写，单词之间没有其他字符，例如：DesertEagle, StyleGuide, ASeriesOfWords。

・camelCase

第一个单词的首字母小写，后面的单词的首字母大写，例如：desertEagle, styleGuide, aSeriesOfWords。

・Snake_case

单词之间用下划线链接，单词的首字母可以大写也可以小写，例如：desert_Eagle, Style_Guide, a_Series_of_Words。

Variables / Properties(变量/属性)

'变量'和'属性'两个词在很多情况下是可以互相通用的。但如果他们同时出现在一个环境时，含义有一些不同：

Property (属性)

'属性'通常定义在一个类的内部。例如，如果一个类 BP_Barrel 有一个内部成员 bExploded，那么 bExploded 可以是做是 BP_Barrel 的一个属性。

当'属性'用在类的内部时，通常用来获取已经定义好的数据。

Variable (变量)

'变量'通常用在给函数传递参数，或者用在函数内的局部变量。

当'变量'用在类的内部时，通常是用来定义什么或者用来保存某些数据的。

1. 资源命名约定

对于资源的命名的规范应该像法律一样被遵守。一个项目如果有好的命名规范，那么在资源管理、查找、解析、维护时，都会有极大的便利性。

大多数资源的命名都应该有前缀，前缀一般是资源类型的缩写，然后使用下划线和资源名链接。

1.1 基本命名规则 Prefix_BaseAssetName_Variant_Suffix

时刻记住这个命名规范 Prefix_BaseAssetName_Variant_Suffix，只要遵守它，大部分情况下都可以让命名规范。下面是详细的解释。

Prefix(前缀)和 Suffix(后缀)由资源类型确定，请参照下面的资源类型表。

所有资源都应该有一个 BaseAssetName(基本资源名)。所谓基本资源名表明该资源在逻辑关系上属于那种资源，任何属于该逻辑组的资源都应该遵守同样的命名规范。

基本资源名应该使用简短而便于识别的词汇，例如，如果你有一个角色名字叫做 Bob，那么所有和 Bob 相关的资源的 BaseAssetName 都应该叫做 Bob。

Varient(扩展名)用来保证资源的唯一性，同样，扩展名也应该是简短而容易理解的短词，以说明该资源在所属的资源逻辑组中的子集。例如，如果 Bob 有多套皮肤，那么这些皮肤资源都应该使用 Bob 作为基本资源名同时包含扩展名，例如'Evil'类型的皮肤资源，名字应该是 Bob_Evil，而 Retro 类型的皮肤应该是用 Bob_Retro。

一般来说，如果仅仅是为了保证资源的唯一性，Varient 可以使用从 01 开始的两位数字来表示。例如，如果你要制作一堆环境中使用的石头资源，那么他们应该命名为 Rock_01, Rock_02, Rock_03 等等。除非特殊需要，不要让数字超过三位数，如果你真的需要超过 100 个的资源序列，那么你应该考虑使用多个基础资源名。

基于你所制作的资源扩展属性，你可以把多个扩展名串联起来。例如，如果你在制作一套地板所使用的资源，那么你的资源除了使用 Flooring 作为基本名，扩展名可以使用多个，例如 Flooring_Marble_01, Flooring_Maple_01, Flooring_Tile_Squares_01。

1.1 e1 Bob

资源类型	资源名
Skeletal Mesh	SK_Bob
Material	M_Bob
Texture (Diffuse/Albedo)	T_Bob_D
Texture (Normal)	T_Bob_N
Texture (Evil Diffuse)	T_Bob_Evil_D

1.2 e2 Rocks

资源类型	资源名
Static Mesh (01)	S_Rock_01
Static Mesh (02)	S_Rock_02
Static Mesh (03)	S_Rock_03
Material	M_Rock
Material Instance (Snow)	MI_Rock_Snow

1.2 资源类型表

当给一个资源命名的时候，请参照以下表格来决定在资本资源名前后所使用的前缀和后缀。

1.2.1 通用类型

资源类型	前缀	后缀	备注
Level / Map			所有地图应该放在 Maps 目录下
Level (Persistent)		_P	
Level (Audio)		_Audio	
Level (Lighting)		_Lighting	
Level (Geometry)		_Geo	
Level (Gameplay)		_Gameplay	
Blueprint	BP_		

Material	M_		
Static Mesh	S_ or SM_		建议使用 S_.
Skeletal Mesh	SK_		
Texture	T_	_?	参照纹理
Particle System	PS_		
Widget Blueprint	WBP_ or WB_		建议使用 WBP_.

1.2.2 动作

资源类型	前缀	后缀	备注
Aim Offset	AO_		
Aim Offset 1D	AO_		
Animation Blueprint	ABP_		
Animation Composite	AC_		
Animation Montage	AM_		
Animation Sequence	A_ or AS_		建议使用 A_.
Blend Space	BS_		
Blend Space 1D	BS_		
Level Sequence	LS_		
Morph Target	MT_		
Paper Flipbook	PFB_		
Rig	Rig_		
Skeletal Mesh	SK_		
Skeleton	SKEL_		

1.2.3 AI

资源类型	前缀	后缀	备注
AI Controller	AIC_		
Behavior Tree	BT_		
Blackboard	BB_		
Decorator	BTDecorator_		
Service	BTService_		
Task	BTTask_		

1.2.4 蓝图

资源类型	前缀	后缀	备注
Blueprint	BP_		
Blueprint Function Library	BPFL_		
Blueprint Interface	BPI_		
Blueprint Macro Library	BPML_		可能的话尽量不要使用蓝图宏
Enumeration	E		没有下划线
Structure	F or S		没有下划线
Widget Blueprint	WBP_ or WB_		建议使用 WBP_.

1.2.5 材质

资源类型	前缀	后缀	备注
Material	M_		
Material (Post Process)	PP_		
Material Function	MF_		
Material Instance	MI_		
Material Parameter Collection	MPC_		
Subsurface Profile	SP_ or SSP_		建议使用 SP_.
Physical Materials	PM_		

1.2.6 纹理

资源类型	前缀	后缀	备注
Texture	T_		
Texture (Diffuse/Albedo/Base Color)	T_	_D	
Texture (Normal)	T_	_N	
Texture (Roughness)	T_	_R	
Texture (Alpha/Opacity)	T_	_A	
Texture (Ambient Occlusion)	T_	_O or _AO	建议使用 _O.
Texture (Bump)	T_	_B	
Texture (Emissive)	T_	_E	
Texture (Mask)	T_	_M	
Texture (Specular)	T_	_S	
Texture (Packed)	T_	_*	参见下面的纹理打包备注.
Texture Cube	TC_		
Media Texture	MT_		
Render Target	RT_ or RTT_		建议使用 RT_.
Cube Render Target	RTC_		
Texture Light Profile	TLP		

1.2.6.1 纹理合并

把多张纹理存于一个纹理文件中是很常见的方法，比如通常可以把自发光(Emissive)，粗糙度(Roughness)，环境光(Ambient Occlusion)以 RGB 通道的形式保存在纹理中，然后在文件的后缀中，注明这些信息，例如_EGO。

一般来说，在纹理的 Diffuse 信息中附带 Alpha/Opacity 信息是很常见的，这时在 _D 后缀中可以加入 A 也可以不用加。

不推荐同时把 RGBA 四个通道的信息保存在一张纹理中，这是由于带有 Alpha 通道的纹理要比不带的占用更多的资源，除非 Alpha 信息是以蒙版(Mask)的形式保存在 Diffuse/Albedo 通道中。

1.2.7 杂项

资源类型	前缀	后缀	备注
Animated Vector Field	VFA_		
Camera Anim	CA_		
Color Curve	Curve_	_Color	
Curve Table	Curve_	_Table	
Data Asset	*_		前缀取决于何种类型资源
Data Table	DT_		
Float Curve	Curve_	_Float	
Foliage Type	FT_		
Force Feedback Effect	FFE_		
Landscape Grass Type	LG_		
Landscape Layer	LL_		
Matinee Data	Matinee_		
Media Player	MP_		
Object Library	OL_		
Redirector			(暂时空缺)
Sprite Sheet	SS_		
Static Vector Field	VF_		
Touch Interface Setup	TI_		
Vector Curve	Curve_	_Vector	

1.2.8 2D

资源类型	前缀	后缀	备注
Paper Flipbook	PFB_		
Sprite	SPR_		
Sprite Atlas Group	SPRG_		
Tile Map	TM_		
Tile Set	TS_		

1.2.9 物理

资源类型	前缀	后缀	备注
Physical Material	PM_		
Physical Asset	PHYS_		
Destructible Mesh	DM_		

1.2.10 声音

资源类型	前缀	后缀	备注
Dialogue Voice	DV_		
Dialogue Wave	DW_		
Media Sound Wave	MSW_		
Reverb Effect	Reverb_		
Sound Attenuation	ATT_		

Sound Class			没有前缀和后缀，这些资源应该放在 SoundClasses 目录
Sound Concurrency		_SC	在 SoundClass 之后命名
Sound Cue	A_	_Cue	
Sound Mix	Mix_		
Sound Wave	A_		

1.2.11 界面

资源类型	前缀	后缀	备注
Font	Font_		
Slate Brush	Brush_		
Slate Widget Style	Style_		
Widget Blueprint	WBP_ or WB_		建议使用 WBP_.

1.2.12 特效

资源类型	前缀	后缀	备注
Particle System	PS_		
Material (Post Process)	PP_		

2. 目录结构

对资源目录的规范管理和资源文件同等重要，都应该像法律一样被严格遵守。不规范的目录结构会导致严重的混乱。

有多种不同管理 UE4 资源目录的方法，在本套规范中，我们尽量利用了 UE4 的资源浏览器的过滤和搜索功能来查找资源，而不是按照资源类型来划分目录结构。

如果你正确遵守了前面使用前缀的资源命名规范，那么就没有必要按照资源类型创建类似于 Meshes, Textures, 和 Materials 这样的目录结构，因为你可以在过滤器中通过前缀来找到特定类型的资源。

2 e1 目录结构示例

```
|-- Content
    |-- GenericShooter
        |-- Art
            |-- Industrial
                |-- Ambient
                |-- Machinery
                |-- Pipes
            |-- Nature
                |-- Ambient
                |-- Foliage
                |-- Rocks
                |-- Trees
            |-- Office
```

```
|-- Characters
    |-- Bob
    |-- Common
        |-- Animations
        |-- Audio
    |-- Jack
    |-- Steve
    |-- Zoe
|-- Core
    |-- Characters
    |-- Engine
    |-- GameModes
    |-- Interactables
    |-- Pickups
    |-- Weapons
|-- Effects
    |-- Electrical
    |-- Fire
    |-- Weather
|-- Maps
    |-- Campaign1
    |-- Campaign2
|-- MaterialLibrary
    |-- Debug
    |-- Metal
    |-- Paint
    |-- Utility
    |-- Weathering
|-- Placeables
    |-- Pickups
|-- Weapons
    |-- Common
    |-- Pistols
        |-- DesertEagle
        |-- RocketPistol
    |-- Rifles
```

使用这种目录结构的原因列在下面。

2.1 文件夹命名

关于文件夹的命名，有一些通用的规范。

2.1.1 使用 PascalCase 大小写规范

文件夹的命名需要遵守 PascalCase 规范，也就是所有单词的首字母大写，并且中间没有任何连接符。例如 DesertEagle, RocketPistol, 和 ASeriesOfWords。

2.1.2 不要使用空格

作为对 2.1.1 的补充，绝对不要在目录名中使用空格。空格会导致引擎以及其他命令行工具出现错误，同样，也不要把你的工程放在包含有空格的目录下面，应该放在类似于 D:\Project 这样的目录里，而不是 C:\Users\My Name\My Documents\Unreal Projects 这样的目录。

2.1.3 不要使用其他 Unicode 语言字符或奇怪的符号

如果你游戏中的角色的名字叫做'Zoë'，那么文件夹要命名为 Zoe。在目录名中使用这样的字符的后果甚至比使用空格还严重，因为某些引擎工具在设计时压根就没有考虑这种情况。

顺便说一句，如果你的工程碰到了类似于这篇帖子中的情况，并且当前使用的系统用户名中包含有 Unicode 字符(比如 Zoë)，那么只要把工程从 My Documents 目录移到类似于 D:\Project 这种简单的目录里就可以解决了。

记住永远在目录名中只使用 a–z, A–Z, 和 0–9 这些安全的字符，如果你使用了类似于_, @, –, *, #这样的字符，难免会碰到一些操作系统、源码管理工具或者一些弱智的工具让你吃个大亏。

2.2 使用一个顶级目录来保存所有工程资源

所有的工程资源都应该保存在一个以工程名命名的目录中。例如你有一个工程叫做'Generic Shooter'，那么所有该工程的资源都应该保存在 Content/GenericShooter 目录中。

开发者目录 Developers 不用受此限制，因为开发者资源是跨工程使用的，参照下面的开发者目录中的详细说明。

使用顶级目录的原因有很多。

2.2.1 避免全局资源

通常在代码规范中会警告你不要使用全局变量以避免污染全局命名空间。基于同样的道理，不存在于工程目录中的资源对于资源管理会造成不必要的干扰。

每个属于项目资源都应该有它存在的目的。如果仅仅是为了测试或者体验而使用的资源，那么这些资源应该放在开发者目录中。

2.2.2 减少资源迁移时的冲突

当一个团队有多个项目时，从一个项目中把资源拷贝到另一个项目会非常频繁，这时最方便的方法就是使用引擎的资源浏览器提供的 Migrate 功能，因为这个功能会把资源的依赖项一起拷贝到目标项目中。

这些依赖项经常造成麻烦。如果两个工程没有项目顶级目录，那么这些依赖项很容易就会被拷贝过来的同名资源覆盖掉，从而造成意外的更改。

这也是为什么 EPIC 会强制要求商城中出售的资源要遵守同样的规定的原因。

执行完 Migrate 资源拷贝后，安全的资源合并方法是使用资源浏览器中的'替换引用'(Replace References)工具，把不属于工程目录中的资源引用替换掉。一旦资源资源完成完整的合并流程，工程目录中不应该存在另一个工程的顶级目录。这种方法可以 100%保证资源合并的安全性。

2.2.2 e1 举例：基础材质的麻烦

举个例子，你在一个工程中创建了一个基础材质，然后你把这个材质迁移到了另一个工程中。如果你的资源结构中没有顶级目录的设计，那么这个基础材质可能放在 Content/MaterialLibrary/M_Master 这样的目录中，如果目标工程原本没有这个材质，那么很幸运暂时不会有麻烦。

随着两个工程的推进，有可能这个基础材质因工程的需求不同而发生了不同的修改。

问题出现在，其中一个项目的美术制作了一个非常不错的模型资源，另一个项目的美术想拿过来用。而这个资源使用了 Content/MaterialLibrary/M_Master 这个材质，那么当迁移这个模型时，Content/MaterialLibrary/M_Master 这个资源就会出现冲突。

这种冲突难以解决也难以预测，迁移资源的人可能压根就不熟悉工程所依赖的材质是同一个人开发的，也不清楚所依赖的资源已经发生了冲突，迁移资源必须同时拷贝资源依赖项，所以 Content/MaterialLibrary/M_Master 就被莫名其妙覆盖了。

和这种情况类似，任何资源的依赖项的不兼容都会让资源在迁移中被破坏掉，如果没有资源顶级目录，资源迁移就会变成一场非常让人恶心的任务。

2.2.3 范例，模板以及商场中的资源都是安全没有风险的

正如上面 2.2.2 所讲，如果一个团队想把官方范例、模板以及商城中购买的资源放到自己的工程中，那么这些资源都是可以保证不会干扰现有工程的，除非你购买的资源工程和你的工程同名。

当然也不能完全信任商城上的资源能够完全遵守顶级目录规则。的确有一些商城资源，尽管大部分资源放在了顶级目录下面，但仍然留下了部分资源污染了 Content 目录

如果坚持这个原则 2.2，最糟糕的情况就是购买了两个不同的商场资源，结果发现他们使用了相同的 EPIC 的示例资源。但只要你坚持把自己的资源放在自己的工程目录中，并且把使用的 EPIC 示例资源也放在自己的目录中，那么自己工程也不会受到影响。

2.2.4 容易维护 DLC、子工程、以及补丁包

如果你的工程打算开发 DLC 或者子工程，那么这些子工程所需要的资源应该迁移出来放在另一个顶级目录中，这样的结构使得编译这些版本时可以区别对待子工程中的资源。子工程中的资源的迁入和迁出代价也更小。如果你想在子项目中修改一些原有工程中的资源，那么可以把这些资源迁移到子工程目录中，这样不会破坏原有工程。

2.3 用来做临时测试的开发者目录

在一个项目的开发期间，团队成员经常会有一个'沙箱'目录用来做测试而不会影响到工程本身。因为工作是连续的，所以即使这些'沙箱'目录也需要上传到源码服务器上保存。但并不是所有团队成员都需要这种开发者目录的，但使用开发者目录的成员来说，一旦这些目录是在服务器上管理的，总会需要一些麻烦事。

首先团队成员极容易使用这些尚未准备好的资源，这些资源一旦被删除就会引发问题。例如一个做模型的美术正在调整一个模型资源，这时一个做场景编辑的美术如果在场景中使用了这个模型，那么很可能会导致莫名其妙的问题，进而造成大量的工作浪费。

但如果这些模型放在开发者目录中，那么场景美术人员就没有任何理由使用这些资源。资源浏览器的缺省设置会自动过滤掉这个目录，从而保证正常情况下不可能出现被误用的情况。

一旦这些资源真正准备好，那么美术人员应该把它们移到正式的工程目录中并修复引用关系，这实际上是让资源从实验阶段'推进'到了生产阶段。

2.4 所有的地图文件应该保存在一个名为'Maps'的目录中

地图文件非常特殊，几乎所有工程都有自己的一套关于地图的命名规则，尤其是使用了 sub-levels 或者 streaming levels 技术时。但不管你如何组织自己的命名规则，都应该把所有地图保存在/Content/Project/Maps。

记住，尽量使用不浪费大家的时间的方法去解释你的地图如何打开。比如通过子目录的方法去组织地图资源，例如建立 Maps/Campaign1/ 或 Maps/Arenas，但最重要的是一定要都放在/Content/Project/Maps。

这也有助于产品的打版本工作，如果工程里的地图保存的到处都是，版本工程师还要到处去找，就让人很恼火了，而把地图放在一个地方，做版本时就很难漏掉某个地图，对于烘培光照贴图或者质量检查都有利。

2.5 使用 Core 目录存储系统蓝图资源以及其他系统资源

使用/Content/Project/Core这个目录用来保存一个工程中最为核心的资源。例如，非常基础的 GameMode, Character, PlayerController, GameState, PlayerState，以及如此相关的一些资源也应该放在这里。

这个目录非常明显的告诉其他团队成员:"不要碰我！"。非引擎程序员很少有理由去碰这个目录，如果工程目录结构合理，那么游戏设计师只需要使用子类提供的功能就可以工作，负责场景编辑的员工只需要使用专用的的蓝图就可以，而不用碰到这些基础类。

例如，如果项目需要设计一种可以放置在场景中并且可以被捡起的物体，那么应该首先设计一个具有被捡起功能的基类放在 Core/Pickups 目录中，而各种具体的可以被捡起的物体，如药瓶、子弹这样的物体，应该放在/Content/Project/Placeables/Pickups/这样的目录中。游戏设计师可以在这些目录中定义和设计这些物体，所以他们不应该去碰 Core/Pickups 目录下的代码，要不然可能无意中破坏工程中的其他功能

2.6 不要创建名为 Assets 或者 AssetTypes 的目录

2.6.1 创建一个名为 Assets 的目录是多余的。

因为本来所有目录就是用来保存资源的。

2.6.2 创建名为 Meshes、 Textures 或者 Materials 的目录是多余的。

资源的文件名本身已经提供了资源类型信息，所以在目录名中再提供资源类型信息就是多余了，而且使用资源浏览器的过滤功能，可以非常便利的提供相同的功能。

比如想查看 Environment/Rocks/目录下所有的静态 Mesh 资源？只要打开静态 Mesh 过滤器就可以了，如果所有资源的文件名已经正确命名，这些文件还会按照文件名和前缀正确排序，如果想查看所有静态 Mesh 和带有骨骼的 Mesh 资源，只要打开这两个过滤器就可以了，这种方法要比通过打开关闭文件夹省事多了。

这种方法也能够节省路径长度，因为用前缀 S_只有两个字符，要比使用 Meshes/七个字符短多了。

这么做其实也能防止有人把 Mesh 或者纹理放在 Materials 目录这种愚蠢行为。

2.7 超大资源要有自己的目录结构

这节可以视为针对 2.6 的补充。

有一些资源类型通常文件数量巨大，而且每个作用都不同。典型的例子是动画资源和声音资源。如果你发现有 15 个以上的资源属于同一个逻辑类型，那么它们应该被放

在一起。

举例来说，角色共用的动画资源应该放在 Characters/Common/Animations 目录中，并且其中应该还有诸如 Locomotion 或者 Cinematic 的子目录。

这并不适用与纹理和材质。比如 Rocks 目录通常会包含数量非常多的纹理，但每个纹理都都是属于特定的石头的，它们应该被正确命名就足够了。即使这些纹理属于材质库。

2.8 材质库 MaterialLibrary

如果你的工程中使用了任何基础材质、分层材质，或者任何被重复使用而不属于特定模型的材质和纹理，这些资源应该放在材质库目录 Content/Project/MaterialLibrary。

这样可以很容易管理这些'全局'材质。

这也使得'只是用材质实例'这个原则得以执行的比较容易。如果所有的美术人员都只是用材质实例，那么所有的原始材质都应该保存在这个目录中。你可以通过搜索所有不在 MaterialLibrary 中的基础材质来验证这一点。

MaterialLibrary 这个目录并不是仅能保存材质资源，一些共用的工具纹理、材质函数以及其他具有类似属性的资源都应该放在这个目录或子目录中。例如，噪声纹理应该保存在 MaterialLibrary/Utility 目录中。

任何用来测试或调试的材质应该保存在 MaterialLibrary/Debug 中，这样当工程正式发布时，可以很容易把这些材质从删除，因为这些材质如果不删除，可能在最终产品中非常扎眼。

3. 蓝图

这一章会专注于蓝图和蓝图的实现。如果可能的话，本规则和 Epic 官方提供的标准一致。

3.1 编译

需要保证所有蓝图在编译时 0 警告和 0 错误。你应该尽快修复所有警告和异常，以免它们造成可怕的麻烦。

绝对不要提交那些断开的蓝图，如果你需要通过源码服务器保存，那么必须暂时搁置它们。

断开的蓝图有巨大的破坏力，而且会在蓝图之外展现威力，比如造成引用失效，未定义的行为，烘培失败，或者频繁的重新编译。一个断开的蓝图可能会毁掉整个项目。

3.2 变量

变量(variable)和属性(property)这两个词经常是可以互换的。

3.2.1 命名规范

3.2.1.1 使用名词

所有非布尔类型的变量必须使用简短、清晰并且意义明确的名词作为变量名。

3.2.1.2 PascalCase

所有非布尔类型的变量的大小写需要遵守 PascalCase 规则。

3.2.1.2 e1

- Score
- Kills
- TargetPlayer
- Range
- CrosshairColor
- AbilityID

3.2.1.3 布尔变量需要前缀 b

所有布尔类型变量需要遵守 PascalCase 规则，但前面需要增加小写的 b 做前缀。

例如：用 bDead 和 bEvil，不要 使用 Dead 和 Evil.

UE4 的蓝图编辑器在显示变量名称时，会自动把前缀 b 去掉

3.2.1.4 布尔类型变量命名规则

3.2.1.4.1 一般的独立信息

布尔类型变量如果用来表示一般的信息，名字应该使用描述性的单词，不要包含具有提问含义的词汇，比如 Is，这个词是保留单词。

例如：使用 bDead and bHostile，不要使用 bIsDead and bIsHostile。

也不要使用类似于 bRunning 这样的动词，动词会让含义变得复杂。

3.2.1.4.2 复杂状态

不要使用布尔变量保存复杂的，或者需要依赖其他属性的状态信息，这会让状态变得复杂和难以理解，如果需要尽量使用枚举来代替。

例如：当定义一个武器时，不要使用 bReloading 和 bEquipping 这样的变量，因为一把武器不可能即在 reloading 状态又在 equipping 状态，所以应该使用定义一个叫做 EWeaponState 的枚举，然后用一个枚举变量 WeaponState 来代替，这也使得以后增加新的状态更加容易。

例如：不要使用 bRunning 这样的变量，因为你以后有可能还会增加 bWalking 或者 bSprinting，这也应该使用一个枚举来非常清晰的定义这样的状态。

3.2.1.5 考虑上下文

蓝图中的变量命名时需要考虑上下文环境，避免重复不必要的定义。

3.2.1.5 e1

假设有一个蓝图名为 BP_PlayerCharacter。

不好的命名：

- PlayerScore
- PlayerKills
- MyTargetPlayer
- MyCharacterName
- CharacterSkills
- ChosenCharacterSkin

这些变量的命名都很臃肿。因为它们都是属于一个角色蓝图 BP_PlayerCharacter 的，没必要在变量中再重复这一点。

好的命名：

- Score
- Kills
- TargetPlayer
- Name
- Skills
- Skin

3.2.1.6 不要在变量中包含原生变量类型名

所谓原生变量是指那些最简单的保存数据的变量类型，比如布尔类型、整数、浮点数以及枚举。

String 和 vectors 在蓝图中也属于原生变量类型，但严格来讲它们其实不是。

由三个浮点数组成的 vector 经常被视为一个整体数据类型，比如旋转向量。

文本类型变量(Text)不属于原生类型，因为它们内部还包含有国际化信息。原生类型的字符串变量类型是 String，而不是 Text。

原生类型的变量名中不应该包含变量类型名。

例如：使用 Score, Kills, 以及 Description，不要使用 ScoreFloat, FloatKills, 或者 DescriptionString。

但也有例外情况，当变量的含义包含了"多少个"这样的信息，并且仅用一个名字无法清晰的表达出这个含义时。

例如：游戏中一个围墙生成器，需要有一个变量保存在 X 轴上的生成数量，那么需要使用 NumPosts 或者 PostsCount 这样的变量，因为仅仅使用 Posts 可能被误解为某个保存 Post 的数组。

3.2.1.7 非原生类型的变量，需要包含变量类型名

非原生类型的变量是指那些通过数据结构保存一批原生类型的复杂变量类型，比如 Structs、Classes、Interface，还有一些有类似行为的原生变量比如 Text 和 Name 也属于此列。

如果仅仅是原生变量组成的数组，那么这个数组仍然属于原生类型。

这些变量的名字应该包含数据类型名，但同时要考虑不要重复上下文。

如果一个类中包拥有一个复杂变量的实例，比如一个 BP_PlayerCharacter 中有另一个变量 BP_Hat，那么这个变量的名字就不需要包含变量类型了。

例如：使用 Hat、Flag 以及 Ability，不要使用 MyHat、MyFlag 和 PlayerAbility。

但是，如果一个类并不拥有这个属性，那么就需要在这个属性的名字中包含有类型的名字了。

例如：一个蓝图类 BP_Turret 用来顶一个炮塔，它拥有瞄准 BP_PlayerCharacter 作为目标的能力，那么它内部会保存一个变量作为目标，名字应该是 TargetPlayer，这个名字非常清楚的指明了这个变量的数据类型是什么。

3.2.1.8 数组

数组的命名规则通常和所包含的元素的规则一样，但注意要用复数。

例如：用 Targets、Hats 以及 EnemyPlayers，不要使用 TargetList、HatArray 或者 EnemyPlayerArray。

3.2.2 可编辑变量

所有可以安全的更改数据内容的变量都需要被标记为 Editable。

相反，所有不能更改或者不能暴露给设计师的变量都不能表上可编辑标志，除非因为引擎的原因，这些变量需要被标为 Expose On Spawn。

总之不要轻易使用 Editable 标记。

3.2.2.1 Tooltips

对于所有标记为 Editable 的变量，包括被标记为 Expose On Spawn 的变量，都应该在其 Tooltip 内填写关于如何改变变量值，以及会产生何种效果的说明。

3.2.2.2 滑动条(Slider)以及取值范围

对于可编辑的变量，如果不适合直接输入具体数值，那么应该通过一个滑动条(Slider)并且加上取值范围来让设计师输入。

举例：一个产生围墙的蓝图，拥有一个 PostsCount 的变量，那么-1 显然适合不合理的输入，所以需要设上取值范围注明 0 是最小值。

如果在构造脚本中需要一个可编辑变量，那么一定要首先定义一个合理的取值范围，要不然可能会有人设上一个非常大的值造成编辑器崩溃。

一个变量的取值范围只有当明确知道其范围时才需要定义，因为滑块的取值范围的确能够阻止用户输入危险数值，但用户仍然能够通过手动输入的方式输入一个超出滑块范围的值给变量，如果变量的取值范围未定义，那么这个值就会变得'很危险'但还是在合理的。

3.2.3 分类

如果一个类的变量很少，那么没有必要使用分类。

如果一个类的变量规模达到中等(5-10)，那么所有可编辑的变量应该自己的分类，而不应该放在缺省分类中，通常叫做 Config。

如果类中的变量的数量非常大，那么所有可编辑的变量都应该放在 Config 分类的子分类下，所有不可编辑的变量应该根据它们的用途建立相关分类保存。

通过在分类名中添加字符|，你可以直接建立子分类，比如 Config | Animations

举例：一个武器的类中的变量分类目录大致如下：

```
|-- Config
    |-- Animations
    |-- Effects
    |-- Audio
    |-- Recoil
    |-- Timings
|-- Animations
|-- State
|-- Visuals
```

3.2.4 变量的访问权限

在 C++中，变量的访问类型由类成员的属性决定，Public 类型的表示其他类都可以访问，Protetced 类型的成员表示子类可以访问，Private 类型变量表示只有类内部函数可以访问此变量。

蓝图并没有类似的权限访问设计。

就是视可编辑类型的变量作为 Public 类型变量，视不可编辑的变量作为 Protected 类型变量。

3.2.4.1 私有变量

尽量不要把变量声明为 private 类型，除非变量一开始就打算永远被类内部访问，并且类本身也没打算被继承。尽量用 protected，private 类型用在当你有非常清楚的理由要去限制子类的能力。

3.2.5 高级显示

如果一个变量可以被编辑，但通常不会有人碰到，那么就把它标记为高级显示 Advanced Display。这些变量在蓝图中会缺省隐藏，除非点击节点上的高级显示箭头。

有意思的是，Advanced Display 这个选项本身，在编辑器的变量属性中也是一个高级显示类型的。

3.2.6 Transient 变量

所有不能编辑并且初始值为 0 或者 null 的变量应该被标记为 Transient。

Transient 类型的变量是指那些不需要被序列化（保存或者加载），并且初始值为 0 或者 null 的变量。一般用在引用其他对象，它们的值只有在运行时才知道。

这种属性的变量会被强制初始化为 0 或者 null，并且不允许编辑器序列化它，以加快蓝图的加载时间。

3.2.7 SaveGame 变量

只有从 SaveGame 继承的子类中的成员变量才能够使用 SaveGame 属性，并且确保该变量应该被保存时才把这个属性设置上。

绝对不要将 SaveGame 和 Transient 同时使用，这是明显不合理的。

3.2.8 Config 变量

不要使用 Config Variable 这个标记，这会让设计师在控制蓝图行为上更加困难。这个标记一般用在 C++中，用来标记那些极少被改变的变量，你可以认为它们是那些被标上 Advanced Advanced Display 的变量。

3.3 函数、事件以及事件派发器

这一节用来解释应该如何管理函数、事件以及事件派发器。除非特殊说明，所有适用于函数的规则，同样适用于事件。

3.3.1 函数命名

对于函数、事件以及事件派发器的命名极其重要，仅仅从一个名字本身，就有很多条件要考虑，比如说：

- 是纯虚函数吗?
- 是状态查询函数吗?
- 是事件相应函数吗?
- 是远程调用函数吗?
- 函数的目的是什么？

如果命名得当，这些问题甚至更多问题的答案会在名字中体现出来。

3.3.1.1 所有函数的命名都应该是动词

所有函数和事件执行者都是需要做一些动作，可能是去获取信息，也可能是数据计算，或者搞点什么事情。因此，所有函数都应该用动词开始，并且用一般现代时态，并且有上下文来表明它们究竟在做什么。

OnRep 这样的相应函数，事件具柄和事件派发器的命名不遵守这个规则。

好的例子：

- Fire – 如果类是一个角色或者武器，那么这是一个好命名，如果是木桶，玻璃，那这个函数就会让人困惑了。
- Jump – 如果类是一个角色，那么这是个好名字，如果不是，那么需要一些上下文来解释这个函数的含义。
- Explode
- ReceiveMessage
- SortPlayerArray
- GetArmOffset
- GetCoordinates
- UpdateTransforms
- EnableBigHeadMode
- IsEnemy – "Is" 是个动词。

不好的例子：

- Dead – 是已经死了？还是死的动作?
- Rock
- ProcessData – 无意义，这个名字等于没说。
- PlayerState – 不能用名词。
- Color – 如果是动词，那么缺少上下文，如果是名词，也不行。

3.3.1.2 属性的状态变化响应函数应该命名为 OnRep_Variable

所有用来响应状态变化的函数应该用 OnRep_Variable 的形式，这是由蓝图编辑器强制规定的，如果你在 C++中写 OnRep 函数，应该同样遵守这个规则。

3.3.1.3 返回布尔变量的信息查询函数应该是问询函数

如果一个函数不改变类的状态，并且只是返回信息、状态或者计算返回给调用者 yes/no，这应该是一个问询函数。同样遵守动词规则。

非常重要的是，应该假定这样的函数其实就是执行某种动作，并且返回动作是否执行成功。

好的例子：

- IsDead
- IsOnFire
- IsAlive
- IsSpeaking
- IsHavingAnExistentialCrisis
- IsVisible
- HasWeapon – "Has" 是动词。

・WasCharging – "Was" 是动词"be"的过去式 用 "was"表示查询以前的状态。
・CanReload – "Can"是动词。
不好的例子：
・Fire – 是查询正在开火？还是查询能不能开火？
・OnFire – 有可能和事件派发器函数混淆。
・Dead – 是查询已经死亡？还是查询会不会死亡？
・Visibility – 是查询可见状态？还是设置可见状态？

3.3.1.4 事件的响应函数和派发函数都应该以 On 开头

事件的响应函数和派发函数都应该以 On 开头，然后遵守动词规则，如果是过去式，那么动词应该移到最后以方便阅读。

在遵守动词规则的时候，需要优先考虑英语中的固定句式。

有一些系统用 Handle 来表示事件响应，但在'Unreal'用的是 On 而不是 Handle。

好的例子：
・OnDeath – 游戏中非常常见。
・OnPickup
・OnReceiveMessage
・OnMessageRecieved
・OnTargetChanged
・OnClick
・OnLeave
不好的例子：
・OnData
・OnTarget
・HandleMessage
・HandleDeath

3.3.1.5 远程调用函数应该用目标作为前缀

任何时候创建 RPC 函数，都应该把目标作为前缀放在前面，例如 Server、 Client 或者 Multicast，没有例外。

前缀之后的部分，遵守上面的其他规则。

好的例子：
・ServerFireWeapon
・ClientNotifyDeath
・MulticastSpawnTracerEffect
不好的例子：
・FireWeapon – 没有使用目标前缀。
・ServerClientBroadcast – 混淆。
・AllNotifyDeath – 用 Multicast，不要用 All。
・ClientWeapon – 没有用动词，让人困惑。

3.3.2 所有函数都应该有返回节点

所有函数都应该有返回节点，没有例外。

返回节点明确标注了蓝图到此执行完毕。蓝图中的结构有可能有并行结构 Sequence、

循环结构 ForLoopWithBreak 或者逆向的回流节点组成，明确结束节点使蓝图易于阅读维护和调试。

如果启用了返回节点，当你的蓝图中有分支没有正常返回，或者流程有问题，蓝图的编译器会提出警告。

比如说，有程序员在并行序列中添加了一个新的分支，或者在循环体外添加逻辑但没有考虑到循环中的意外返回，那么这些情况都会造成蓝图的执行序列出现意外。蓝图编译器会立即给这些情况提出警告。

3.4 蓝图图形

本节包含了关于蓝图图形的内容。

3.4.1 不要画“意面”

蓝图中所有连线都应该有清晰的开始点和结束点。你的蓝图不应该让阅读者在一堆乱糟糟的线中翻来翻去。以下内容是帮助你避免“意大利面”样式的蓝图产生。

3.4.2 保持连线对齐，而不是节点

不要试图让节点对齐，对齐的应该是连线。你无法控制一个节点的大小和上面连接点的位置，但你能通过控制节点的位置来控制连线。笔直的连线让整个蓝图清晰美观，歪歪扭扭的连线会让蓝图丑陋不堪。你可以通过蓝图编辑器提供的功能直接让连线变直，方法是选择好节点，用快捷键 Q。

好的例子：所有上面的节点的执行线都保持为直线。

不好的例子：某些节点的执行线歪了。

可接受的例子：有些节点无论你怎么用对齐工具都无法对齐，这种情况下，就尽量缩短它们之间连线的长度。

3.4.3 白色的可执行线优先级最高

如果发现白色执行线和其他数据线无法同时对齐，白色执行线的优先级更高。