# 虚幻引擎指针

**原则：只要存在有效的引用，则对象不会被释放。如果对象是场景对象，调用Destory则强制释放指针对象，并更新到所有引用。**

* 所有的U类成员指针都应当加入UPROPERTY标记，这样可以防止操作指针时出现野指针。有效提升了指针对象的引用操作
* Tarry数组，如果存数的是U类指针，也应该加上UPROPERTY标记宏，这样的好处是当指针对象在引擎被释放时，数组中的元素也将变为空（如果不加U，则不会为空。变为空但并不移除出数组，需要手动移除）
* 函数传递U类指针时，如果接收指针的对象使用UPROPERTY标记，则也会进行计数器更新。如原有对象被释放，被赋值的指针也会跟随更新被释放

# 智能指针

UE为了解决动态内存泄漏问题，并且增加C++的移植括拓展性，设计了一套非常强大的动态内存管理机制，而这套机制中根本在于智能指针，并且UE的智能指针速度相比STL更快，速度和普通C++指针速度一样。

智能指针本质的目的是将释放内存工作进行托管。当两个智能指针指向同一个空间，一个设置为空，另一个不会跟随为空，智能指针设置为空并不是释放内存空间，只是在减少空间引用。

这与UPROPERTY标记不同，UPROPERTY标记的u类指针，当对象调用destroy后，指针在轮巡时会主动被置为null，而智能指针的目的是内存托管，所以，当内存被释放时，智能指针不会被隐式置为null

对于指针，如果选择了使用智能指针进行内存托管，就不要再显示调用delete关键字进行内存释放，这将导致内存管理器出现释放内存错误。

智能指针分为

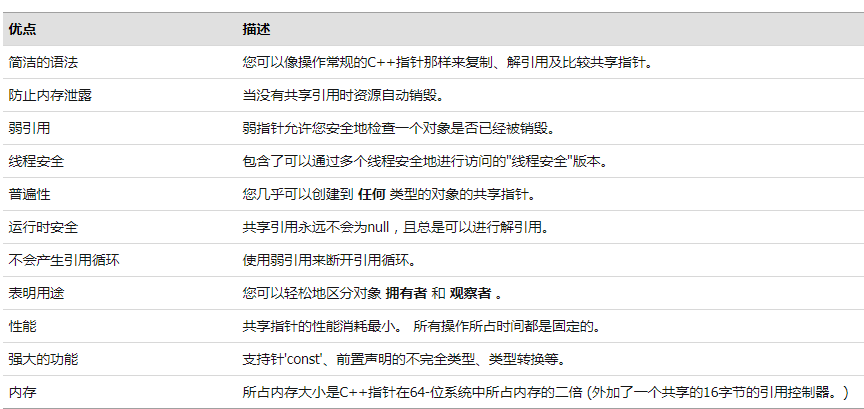


实现基本原理

一块内存，如果存在有效引用（可直接到达内存的操作方式），则我们可以认为当前内存是有效并且合理的！但是当一块内存不存在引用，则我们可以视为此块内存为被弃用无效的，则可以回收重复利用，这就是内存垃圾回收机制的基本原理

**所有的UBO类均不能使用UE智能指针**

### 优点



### 常用函数

MakeShareable()

用于通过C++指针初始化共享指针(启用隐式转换) 。

StaticCastSharedRef()//编译阶段

静态类型转换效用函数，一般用于向下类型转换为派生类型。

ConstCastSharedRef()

将一个'const(常量的)'引用转换为'mutable（可变的）'智能引用。

DynamicCastSharedRef()

动态类型转换效用函数，一般用于向下类型转换为派生类型。

StaticCastSharedPtr()

静态类型转换效用函数，一般用于向下类型转换为派生类型。

ConstCastSharedPtr()

将一个'const(常量的)'引用转换为'mutable（可变的）'智能指针。

DynamicCastSharedPtr()

动态类型转换效用函数，一般用于向下类型转换为派生类型。

### 共享指针TSharedPtr

非侵入式操作，不关心模版数据类型，支持强引用也支持弱引用。具有防止内存泄漏，防止存在未初始化的内存

额外功能

* 共享所有权 - 引用计数
* 自动失效 - 可以安全地引用易发生变化的对象
* 弱引用 - 允许通过弱指针 避免产生引用循环。
* 针对程序员应用目的进行了改进 - 区分了拥有者和观察者，提供了不能为null的引用 ( 共享引用 )

##### 声明和初始化

TSharedPtr<T> SPtr;//空白的共享指针

TSharedPtr<T> SPtr(new T());//构建共享指针

TSharedPtr<float> ptr = MakeShareable(new float(1.0));

##### 解引用和访问

m\_fb->PrintLog();

m\_fb.Get()->PrintLog();

(\*m\_fb).PrintLog();

三种均可

##### 比较

if (m\_fb == p1)

{

}

##### 判断安全性是否为空

if (m\_fb)

{

}

if (m\_fb.IsValid())

{

}

if (m\_fb.Get() != NULL)

{

}

三种均可

##### 释放

两种均可

m\_fb = NULL;

m\_fb.Reset();

### 共享引用TSharedRef

共享引用禁止为空，表明了共享引用创建后必须给予有效初始化，可以使得代码更加安全简洁，保证了对象访问的安全性。建议使用共享引用。无法主动释放共享引用，可以跟随对象释放减少引用计数器

共享引用的安全性体现在如果使用共享引用构建的对象，无法将对象空间设置为空，当然可以借助指向其他共享引用来减少引用计数，来释放空间

共享引用本质，无法主动减少引用计数器，只能通过被动方法，例如生命周期终结，共享引用易主

##### 声明和初始化

共享引用创建不进行初始化将导致编辑器崩溃



错误的范例



##### 解引用和操作



##### 和共享指针的转换

共享引用支持隐式转换为共享指针，由于共享引用是安全的，所以转换是隐式转换



从共享指针转换到共享引用是不安全的，所以需要调用TS函数



### 弱指针

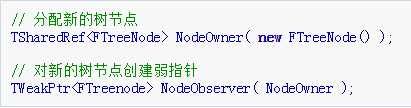
不会阻止对象的销毁，如果引用对象被销毁，则弱指针也将自动清空。一般弱指针的操作意图是保存了一个到达目标对象的指针，担不会控制该对象的生命周期，若指针不会增加引用计数，可以用来断开引用循环问题

##### 声明和初始化

1. 直接构建空的弱指针

TWeakPtr<FString> pWeak; //构建空的弱指针

1. 依托共享指针或是共享引用或是其他弱指针进行创建



弱指针可以随意复制，使用完毕后可以直接指向NULL

##### 解引用和访问

弱指针无法直接对对象成员进行直接访问，需要先提升弱指针到共享指针，通过调用Pin函数

// Get access to the node through the weak pointer.

TSharedPtr<TFreeNode> LockedObserver( NodeObserver.Pin() );

// Check that the shared reference was successfully created from the weak reference.

If( LockedObserver.IsValid() )

{

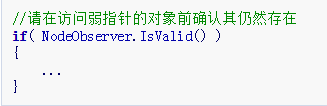
// Object still exists, so it can be accessed.

LockedObserver->ListChildren();

}

Pin的意图是锁定对象，阻止对象在调用时被销毁

##### 判断弱指针的有效性



弱指针的调用都应该先确认是否存在有效的引用

##### 转换

通过Pin函数可以把弱指针转换到共享指针。

### 引用循环

类自身存在自身引用，或是AB类互存相互引用，当对象释放时计数器释放后，但是还存在其他有效的计数器引用导致内存无法释放，造成内存泄漏

自身引用自身：当有效指针释放时，由于自身内部还存在一个引用指针，导致引用计数器没有置零无法释放

AB互相引用：A释放时B内存在A引用导致无法直接释放，B释放时A也存在B的有效引用也无法释放，导致内存泄漏