虚幻中的智能指针

虚幻引擎交互开发工程师

01

指针

前言

C++中，往往令人头痛的是指针的管理问题！在对象动态构建时，我们需要将对象指针进行存储，一旦忘记释放，那么将会导致不可预估的错误。在C++中排查指针导致的内存泄漏问题实在令人头痛！在虚幻中，为了解决此类问题，加入了智能指针（共享指针，共享引用，弱指针），当我们使用动态方式构建对象时，再也不需要担心内存释放问题！指针的释放规则由引擎制定，包括释放时机！

02

自定义类

前言

在构建自定义类时，我们经常遇到一种情况，当类中持有U类对象指针时，我们希望阻止垃圾回收器对对象释放。但是自定义类中又无法使用UPROPERTY宏，那么我们可以采取将类继承自FGCObject，并重写父类函数AddReferencedObjects。将需要阻止释放的指针加入到操作队列，以防止对象被垃圾回收器回收！

注意：当构建类被释放时（需要我们保证），并且调用其析构函数（**析构函数需要重写父类析构函数**），对象将自动清除其所添加的所有引用。

语法结构

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GCObject.h"

/**
 *
 */
class UECPP_API FClass : public FGCObject
{
public:
    FClass();
    virtual ~FClass() override;


    virtual void AddReferencedObjects(FReferenceCollector& Collector) override;

protected:
    class UObject* p_Uobject;
};
```

```
void FPClass::AddReferencedObjects(FReferenceCollector& Collector)
{
    Collector.AddReferencedObject(p_Object);//将指针在垃圾回收器中标记为强引用，无法被回收
}
```

03

智能指针

前言

虚幻中存在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（非侵入式），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。

注意：智能指针只能使用于自定义类，U类禁止使用

共享指针和共享引用优点

优点	描述
简洁的语法	您可以像操作常规的C++指针那样来复制、解引用及比较共享指针。
防止内存泄露	当没有共享引用时资源自动销毁。
弱引用	弱指针允许您安全地检查一个对象是否已经被销毁。
线程安全	包含了可以通过多个线程安全地进行访问的"线程安全"版本。
普遍性	您几乎可以创建到 任何 类型的对象的共享指针。
运行时安全	共享引用永远不会为null，且总是可以进行解引用。
不会产生引用循环	使用弱引用来断开引用循环。
表明用途	您可以轻松地区分对象 拥有者 和 观察者 。
性能	共享指针的性能消耗最小。 所有操作所占时间都是固定的。
强大的功能	支持针'const'、前置声明的不完全类型、类型转换等。
内存	所占内存大小是C++指针在64-位系统中所占内存的二倍 (外加了一个共享的16字节的引用控制器。)

三个指针

共享指针	描述
共享指针 (TSharedPtr)	引用计数的非侵入式的权威智能指针。
共享引用 (TSharedRef)	不能设置为null值的、引用计数的、非侵入式权威智能指针。
弱指针 (TWeakPtr)	引用计数的、非侵入式弱指针引用。

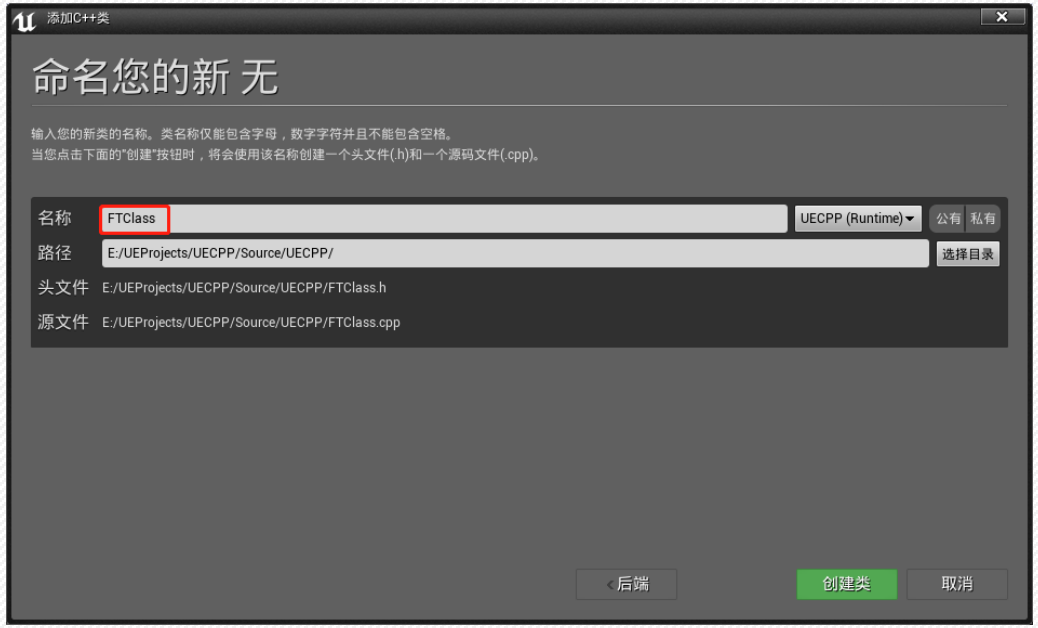
04

共享指针

前言

共享指针是虚幻中最常用的智能指针，在操作上可以帮助我们构建托管内存指针！共享指针本身非侵入式的，这使得指针的使用与操作和普通指针一致！共享指针支持主动指向空，并且共享指针是线程安全的，节省内存，性能高效

注意：构建自定义类时，需要使用F开头



基本操作语法

声明和初始化

```
TSharedPtr<FNClass> pN;//构建一个共享指针，但并没有维护任何内存
TSharedPtr<FNClass> pN(new FNClass());//构建一个共享指针，并维护了一块内存
TSharedPtr<FNClass> pN = MakeShareable(new FNClass());//构建一个共享指针，并维护了一块内存
```

MakeShareable函数是用来构建共享指针的快捷方式

解引用和操作（以下三个均可使用）

```
pN->CallFun();//CallFun是成员函数
pN.Get()->CallFun();
(*pN).CallFun();
```

基本操作语法

比较

```
TSharedPtr<FNClass> pN1;
TSharedPtr<FNClass> pN = MakeShareable(new FNClass());

if (pN1 == pN)//比较两个智能指针管理的内存是否是一份
{
}
```

判断是否有效

```
TSharedPtr<FNClass> pN = MakeShareable(new FNClass());

if (pN.IsValid())//判断是否为空 注意操作的函数是共享指针的成员函数
{
}
if (pN.Get() != nullptr)//注意操作的函数是共享指针的成员函数
{
}
```

基本操作语法

释放（两种方式均可）

```
TSharedPtr<FNClass> pN = MakeShareable(new FNClass());

pN.Reset();
pN = nullptr;
```

获取引用计数器

```
TSharedPtr<FNClass> pN = MakeShareable(new FNClass());

pN.GetSharedReferenceCount();//获得当前地址被引用个数
```

05

共享引用

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。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。如果想释放内存，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

操作语法

声明和初始化

```
TSharedRef<FNClass> pN;//错误 执行将导致崩溃
TSharedRef<FNClass> pN(new FNClass());//正确
```

解引用操作

```
TSharedRef<FNClass> rF(new FNClass());

rF->CallFun();
(*rF).CallFun();
rF.Get().CallFun();//共享引用中的Get返回的是引用对象数据，可以防止删除数据
```

和共享指针转换

```
//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
TSharedPtr<FNClass> pSN = pN;
//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
TSharedRef<FNClass> pM = pSN.ToSharedRef();
```

06

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，弱指针不会增加引用计数，可以用来断开**引用循环**问题。

无论谁销毁了对象，只要其对象被销毁，弱指针都将自动清空，这使你能够安全缓存指向可变对象的指针。这也意味着，弱指针可能会意外清空，并且，你可以使用弱指针断开引用循环。

当不再存在对对象的共享引用时，弱指针的对象将被销毁。

弱指针有助于表明意图。当你在某个类中看到一个弱指针时，你就会明白该类仅缓存指向对象的指针，它并不控制它的生命周期。

操作语法

声明和初始化

```
TWeakPtr<FNClass> pWN;//构建空的弱指针

TSharedPtr<FNClass> pN;
TWeakPtr<FNClass> pWN1(pN);//借助共享指针构建

TSharedRef<FNClass> pRN;
TWeakPtr<FNClass> pWN2(pRN);//借助共享引用构建
```

解引用操作

```
//弱指针无法直接调用（弱指针调用会出现对象为空，因为弱指针不会阻止对象释放）
//弱指针如果需要操作，需要转换为共享指针
TSharedPtr<FNClass> pN(pWN.Pin());
if (pN.IsValid())//检查是否转换成功
{
    //使用共享指针的操作方式
}
```

Pin函数会阻止对象被销毁

操作语法

检查是否有效

```
//检查弱指针指向的对象空间是否存在
if (pWN.IsValid())
{
}
```

释放操作

```
//主动释放，但是并不会影响引用计数
pWN = nullptr;
```

07

总结

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。

智能指针强调的是当前内存的使用者存在多少，当不存在时，进行回收！

注意：智能指针构建的均是对象数据类型

感谢观看

虚 幻 四 高 级 程 序 开 发