

FUObjectItem

FUObjectArray

```
struct FUObjectItem
{
    // Pointer to the allocated object
    class UObjectBase* Object;
    // Internal flags
    int32 Flags;
    // UObject Owner Cluster Index
    int32 ClusterRootIndex;
    // Weak Object Pointer Serial number associated with the object
    int32 SerialNumber;
};

// 获取UObject对象对应的FUObjectItem
FUObjectItem* ObjItem = GUObjectArray.IndexToObject(Obj->GetUniqueID());
```

GC Mark

IsUnreable可用于判断UObject是否可达，但该方法调用必须在GC标记阶段完成后才有效，即使一个对象时不可达的，但是在没有执行GC标记前仍然返回true（可达）。因为GC Mark阶段会给对象添加EInternalObjectFlags::Unreachable标记，该标记用于判断对象是否可达。

GC Mark是在TaskGraph线程执行的操作，游戏线程处于等待状态

GC Sweep

主要有三个阶段

1. ConditionalBeginDestroy

在增量GC中，会严格执行内存加载存储顺序，来保持内存排列没有发生变化，否则可能造成在GC清理时候造成内存错乱等问题。首先会增量得把不可达对象全部执行BeginDestroy，并修改UObject Flag标记为RF_BeginDestroyed，来防止BeginDestroy函数执行多次。这个过程阶段执行时间会有限制，**如果超过了这个限制时间还没有把全部UObject遍历完，将下次Tick时候执行（分帧执行，防止卡顿）**，并退出后续的增量GC操作，直到把全部的UObject遍历完全后开始下一步的ConditionalFinishDestroy操作。

2. ConditionalFinishDestroy

ConditionalFinishDestroy遍历执行所有已经执行了BeginDestroy的不可达对象，目的准备执行FinishDestroy并修改UObject Flag标记为RF_FinishDestroyed。当然这这个过程也是会有时间限制的，如果超过了限制时间，也是会等到下次Tick继续增量执行。

这个过程会有两个阶段，第一个阶段遍历所有的UObject(已经执行了BeginDestroy的不可达UObject) 将已经处于ReadyForFinishDestroy状态的UObject直接执行FinishDestroy，其余还没有进入准备状态的Object可能某个图形资源在等待渲染资源的完成，为了不想等待渲染线程的过程走完造成GC堵塞，将其存入到一个Pending列表中。执行IsReadyForFinishDestroy方法后可通知渲

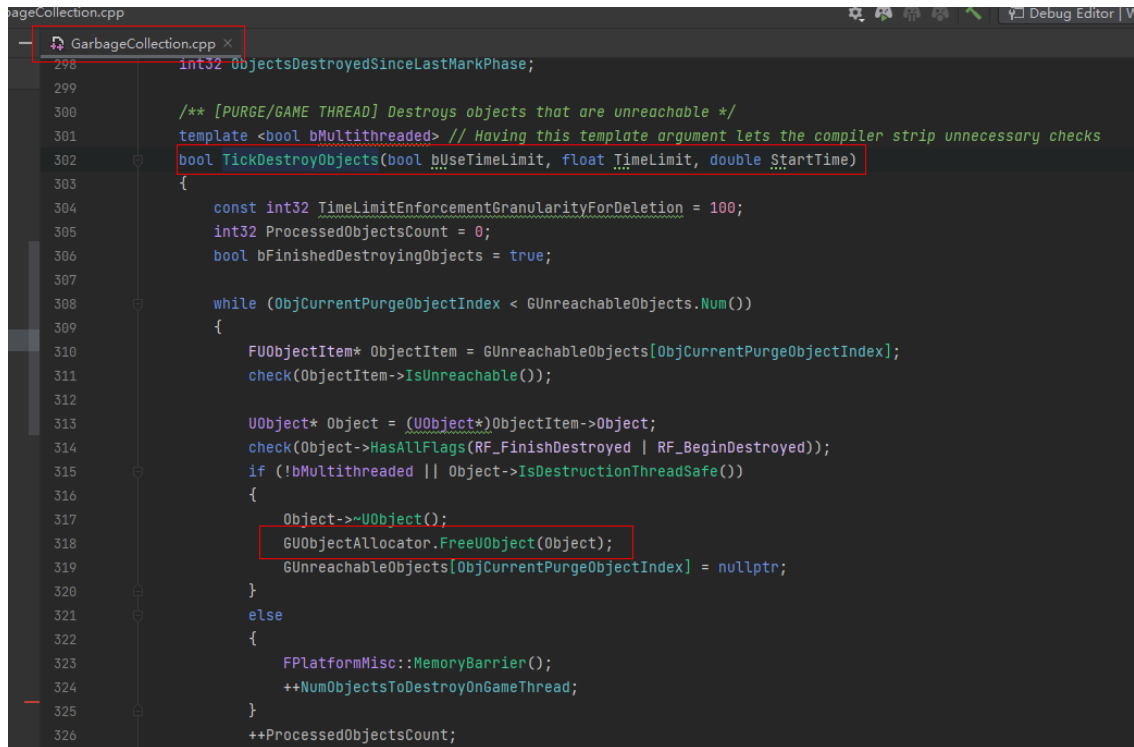
染线程释放这个资源对象。

第二阶段就是在第一阶段基础上，获取到剩余所有UObject的Pending列表并遍历，再次执行第一阶段的过程，直到Pending列表中的所有UObject对象都执行了FinishDestroy方法才会进入到第三个DestructUObject过程。如果这一次Tick有的UObject没有进入ReadyForFinishDestroy状态的话，将再下一次Tick继续尝试判断UObject资源是否从渲染线程退出并是否处于ReadyForFinishDestroy状态。

3. DestructUObject

TickDestroyObjects->GUObjectAllocator.FreeUObject

最终才是UObject对象的完全析构，第一和第二个过程都是在HashTable将UObject移除，并做一些Destroy准备工作，在这个第三过程才是从内存中完全析构销毁。将全部的对象都销毁之后将变量初始化，准备下一次的GC操作。



执行GC操作的函数

以阻塞的方式尝试进行一次GC Mark

GEngine->PerformGarbageCollectionAndCleanupActors();

TryCollectGarbage(GARBAGE_COLLECTION_KEEPFLAGS, false); // ① 会先检查在其他线程中是否有UObject操作 ② 连续尝试没成功的次数 > GNumRetriesBeforeForcingGC时 注：UE4.25中GNumRetriesBeforeForcingGC配置为10

GEngine->ForceGarbageCollection(false); // 下一帧才以阻塞的方式尝试进行一次GC Mark

以阻塞的方式进行一次GC Mark

CollectGarbage(RF_NoFlags, false);

CollectGarbage(GARBAGE_COLLECTION_KEEPFLAGS, false);

如果连续2次调用GC Mark，在第2次GC Mark之前，会先阻塞执行一次全量的GC Sweep

UObjectBase.cpp 4 Obj.cpp 4 GarbageCollection.cpp 4 MyTest1Character.cpp UObjectGlobals.h MyTest1Character.h WindowsPlatform.h

→ FAsyncPurge.TickDestroyObjects.while.if class FAsyncPurge : public FRunnable

```
294     const int32 TimeLimitEnforcementGranularityForDeletion = 100;
295     int32 ProcessedObjectsCount = 0;
296     bool bFinishedDestroyingObjects = true;
297
298     while (ObjCurrentPurgeObjectIndex < GUnreachableObjects.Num())
299     {
300         FObjectItem* ObjectItem = GUnreachableObjects[ObjCurrentPurgeObjectIndex];
301         check(ObjectItem->IsUnreachable());
302
303         UObject* Object = (UObject*)ObjectItem->Object;
304         check(Object->HasAllFlags(RF_FinishDestroyed | RF_BeginDestroyed));
305         if (!Thread || Object->IsDestructionThreadSafe())
306         {
307             Object->~UObject();
308             GUObjectAllocator.FreeUObject(Object);
309         }
310     }
```

110 % No issues found

Call Stack

| Name |
|---|
| UE4Editor-MyTest1-Win64-Debug.dllIFMyTest1Listener::NotifyUObjectDeleted(const UObjectBase * Object=0x000002c612aa6b00, int Index=17912) Line 25 |
| UE4Editor-CoreUObject-Win64-Debug.dllIFUObjectArray::FreeUObjectIndex(UObjectBase * Object=0x000002c612aa6b00) Line 280 |
| UE4Editor-CoreUObject-Win64-Debug.dllIUObjectBase::~UObjectBase() Line 131 |
| UE4Editor-CoreUObject-Win64-Debug.dllIUObject::~~UObject() |
| UE4Editor-Engine-Win64-Debug.dllIAActor::~~AActor() |
| UE4Editor-Engine-Win64-Debug.dllIAActor::vector deleting destructor'(unsigned int) |
| UE4Editor-CoreUObject-Win64-Debug.dllIFAsyncPurge::TickDestroyObjects(bool bUseTimeLimit=false, float TimeLimit=0.002000000009, double StartTime=17390284.614233900) Line 308 |
| UE4Editor-CoreUObject-Win64-Debug.dllIFAsyncPurge::TickPurge(bool bUseTimeLimit=false, float TimeLimit=0.002000000009, double StartTime=17390284.614233900) Line 429 |
| UE4Editor-CoreUObject-Win64-Debug.dllIncrementalDestroyGarbage(bool bUseTimeLimit=false, float TimeLimit=0.002000000009) Line 1691 |
| UE4Editor-CoreUObject-Win64-Debug.dllIncrementalPurgeGarbage(bool bUseTimeLimit=false, float TimeLimit=0.002000000009) Line 1421 |
| UE4Editor-CoreUObject-Win64-Debug.dllCollectGarbageInternal(EObjectFlags KeepFlags=RF_NoFlags, bool bPerformFullPurge=false) Line 1902 |
| UE4Editor-CoreUObject-Win64-Debug.dllCollectGarbage(EObjectFlags KeepFlags=RF_NoFlags, bool bPerformFullPurge=false) Line 2067 |

限制时间来分帧进行一次GC Sweep

```
IncrementalPurgeGarbage(true); // 以缺省0.002的时间进行一次GC Sweep
```

```
IncrementalPurgeGarbage(true, 0.1); // 以0.1的时间进行一次GC Sweep
```

引擎在每帧Tick中都在通过限制时间来分帧异步进行GC Sweep

The image shows a Visual Studio Code editor window with a C++ source file named 'UnrealEngine.cpp'. A breakpoint is set at line 1388, which is highlighted in yellow. The code is part of a garbage collection system, specifically the 'ConditionalCollectGarbage' function. The breakpoint is hit, and the 'Call Stack' is visible on the left, showing the sequence of function calls leading to the current line.

Source File: UnrealEngine.cpp

```

1375         bShouldDelayGarbageCollect = false;
1376     }
1377     // Perform incremental purge update if it's pending or in progress.
1378     else if (!IsIncrementalPurgePending())
1379     {
1380         // Purge reference to pending kill objects every now and so often.
1381         && (TimeSinceLastPendingKillPurge > TimeBetweenPurgingPendingKillObjects)
1382     {
1383         SCOPE_CYCLE_COUNTER(STAT_GCMarkTime);
1384         PerformGarbageCollectionAndCleanupActors();
1385     }
1386     else
1387     {
1388         SCOPE_CYCLE_COUNTER(STAT_GCSweepTime);
1389         IncrementalPurgeGarbage(true); // ≤ 7ms elapsed
1390     }

```

Call Stack

- UE4Editor-Engine-Win64-Debug.dll!UnrealEngine::ConditionalCollectGarbage() Line 1388
- UE4Editor-Engine-Win64-Debug.dll!UWorld::Tick(ELevelTick TickType=LEVELTICK_Alt, float DeltaSeconds=0.400000006) Line 1758
- UE4Editor-Engine-Win64-Debug.dll!UGameEngine::Tick(float DeltaSeconds=0.720838070, bool bIdleMode=false) Line 1706
- UE4Editor-Win64-Debug.exe!EngineLoop::Tick() Line 4850
- UE4Editor-Win64-Debug.exe!EngineTick() Line 63
- UE4Editor-Win64-Debug.exe!GuardedMain(const wchar_t* CmdLine=0x0000183419bb280) Line 172
- UE4Editor-Win64-Debug.exe!WinMain(HINSTANCE_ * hInst=0x00007f687b10000, HINSTANCE_ * hPrevInst=0x0000000000000000, char* __format=0x00001833e664486, int nCmdShow=10) Line 257
- [Inline Frame] UE4Editor-Win64-Debug.exe!invoke_main() Line 102
- UE4Editor-Win64-Debug.exe!_scrt_common_main_seh() Line 288
- kernel32.dll!BaseThreadInitThunk()
- ntdll.dll!RtlUserThreadStart()

阻塞的方式进行一次GC Sweep

```
IncrementalPurgeGarbage(false); // 以阻塞的方式进行一次GC Sweep
```

以阻塞的方式尝试进行一次全量的GC（包括Mark和Sweep阶段）

```
TryCollectGarbage(GARBAGE_COLLECTION_KEEPFLAGS, true);
```

```
GEngine->Exec(nullptr, TEXT("obj trygc"));
```

```
GEngine->ForceGarbageCollection(true); // 下一帧才以阻塞的方式尝试进行一次全量的GC
```

以阻塞的方式进行一次全量的GC（包括Mark和Sweep阶段）

```
CollectGarbage(RF_NoFlags);
```

```
CollectGarbage(GARBAGE_COLLECTION_KEEPFLAGS);
```

```
CollectGarbage(GARBAGE_COLLECTION_KEEPFLAGS, true);
```

```
GEngine->Exec(nullptr, TEXT("obj gc"));
```

GC相关的代理

```
static FSimpleMulticastDelegate& GetPreGarbageCollectDelegate(); // GC Mark或全量GC执行之前的代理通知
```

```
static FSimpleMulticastDelegate& GetPostGarbageCollect(); // GC Mark或全量GC完成之后的代理通知
```

```
static FSimpleMulticastDelegate PreGarbageCollectConditionalBeginDestroy; // GC Sweep ConditionalBeginDestroy之前的代理通知
```

```
static FSimpleMulticastDelegate PostGarbageCollectConditionalBeginDestroy; // GC Sweep ConditionalBeginDestroy完成之后的代理通知
```

```
static FSimpleMulticastDelegate PostReachabilityAnalysis; // GC Mark可达性分析之后的代理通知
```

GC相关的状态API

```
bool IsGarbageCollectingOnGameThread() // GC是否在游戏线程上
```

```
bool IsInGarbageCollectorThread() // 是否在GC线程上
```

```
bool IsGarbageCollecting() // 是否正在执行GC逻辑
```

```
bool IsGarbageCollectionWaiting() // GC是否在等待运行
```

GC锁

使得在垃圾回收时，其他线程的任何UObject操作都不会工作，避免出现一边回收一边操作导致的问题

```
FGCCSyncObject::Get().TryGCLock(); // 尝试获取GC锁
```

```
AcquireGCLock(); // 获取GC锁
```

```
ReleaseGCLock(); // 释放GC锁
```

bool IsGarbageCollectionLocked() // GC锁是否已经被获取了

```
{  
    FGCScopeGuard GCGuard; // 进入作用域获取GC锁，离开自动释放GC锁 非GameThread有效  
    Package = new FAsyncPackage(*this, *InRequest, EDLBootNotificationManager);  
}
```

引擎中的GC逻辑

在Tick中调用GC逻辑

具体实现在：void UEngine::ConditionalCollectGarbage()函数中

在LoadMap中以阻塞的方式进行一次全量的GC

具体实现在：void UEngine::TrimMemory()函数中

GC相关的设置

