一，GC的定义

GC即为Garbage Collection

二，为什么要执行GC

程序运行过程中会申请大量的内存空间，如果不及时释放会导致内存使用殆尽

三，ue4如何执行GC

ue4采用标记-清扫方式

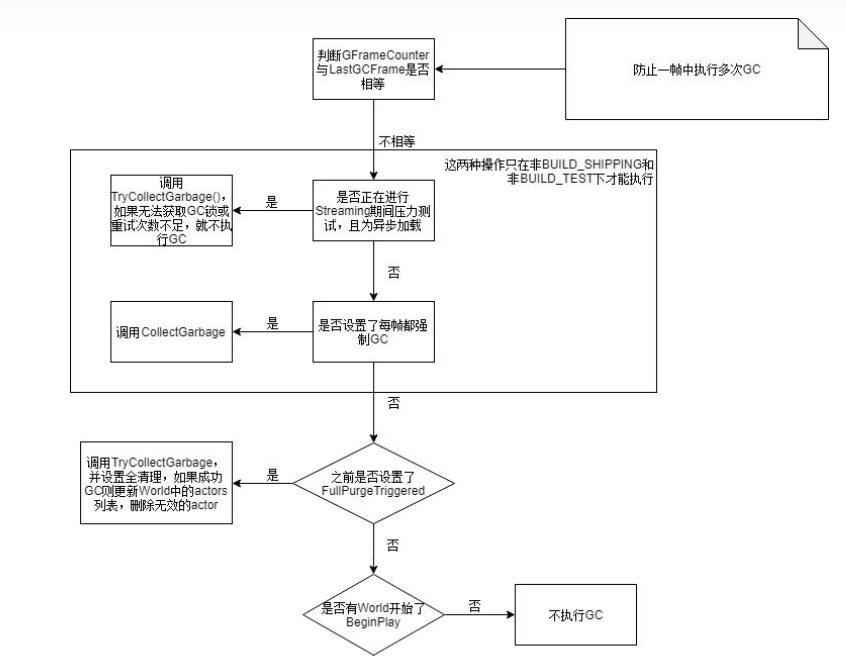
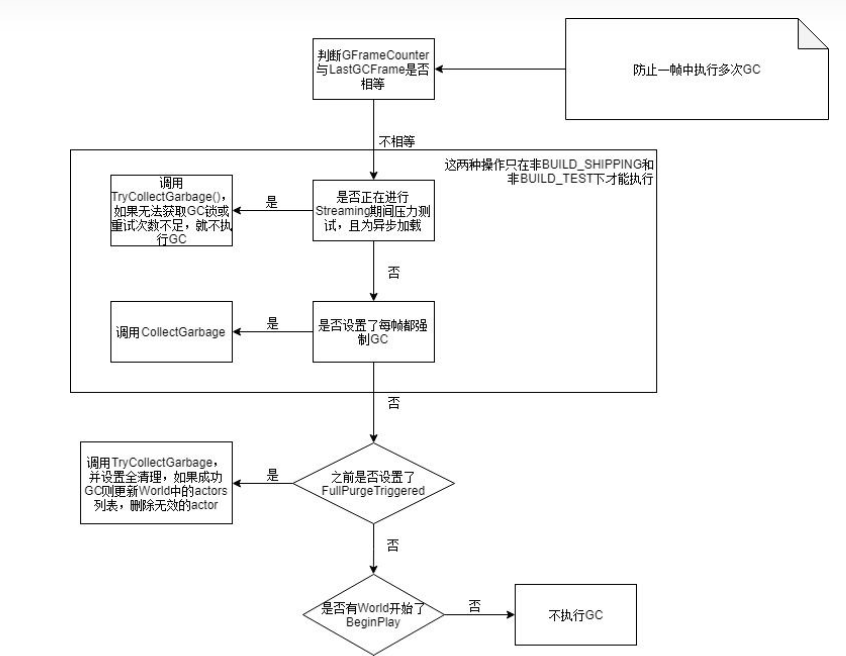
GC分为主动引发和被动引发两种方式

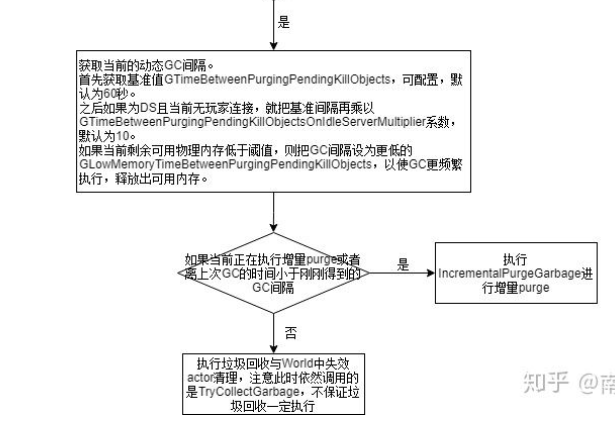
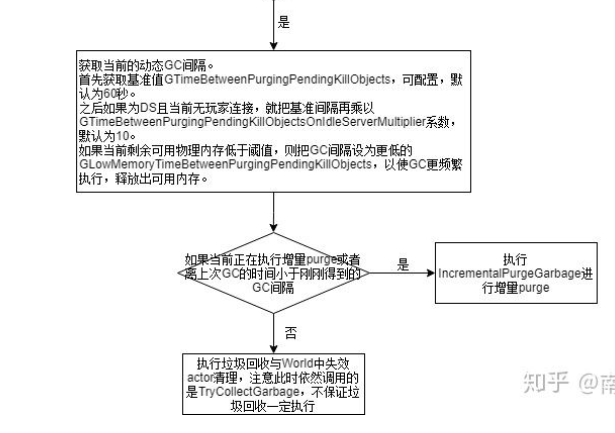
但是引擎大部分时间会进行被动引发

被动引发：

如何触发GC：

World::Tick()->UEngine::ConditionalCollectGarbage()

Streaming期间压力测试是啥？



调用bool TryCollectGarbage(*EObjectFlags* KeepFlags，bool bPerformFullPurge = true)，如果调用成功则删除无效的actors (void CleanupActors()  目的将关卡中Actors中为空指针的元素清理出去，但是不执行shrink)

具体的GC流程：

void CollectGarbage(EObjectFlags Keepflags,bool bPerformPurge)

void *CollectGarbage*(*EObjectFlags* KeepFlags, bool bPerformFullPurge)

{

       // No other thread may be performing UObject operations while we're running

*AcquireGCLock*();

       // Perform actual garbage collection

       CollectGarbageInternal(KeepFlags, bPerformFullPurge);

       // Other threads are free to use UObjects

*ReleaseGCLock*();

}

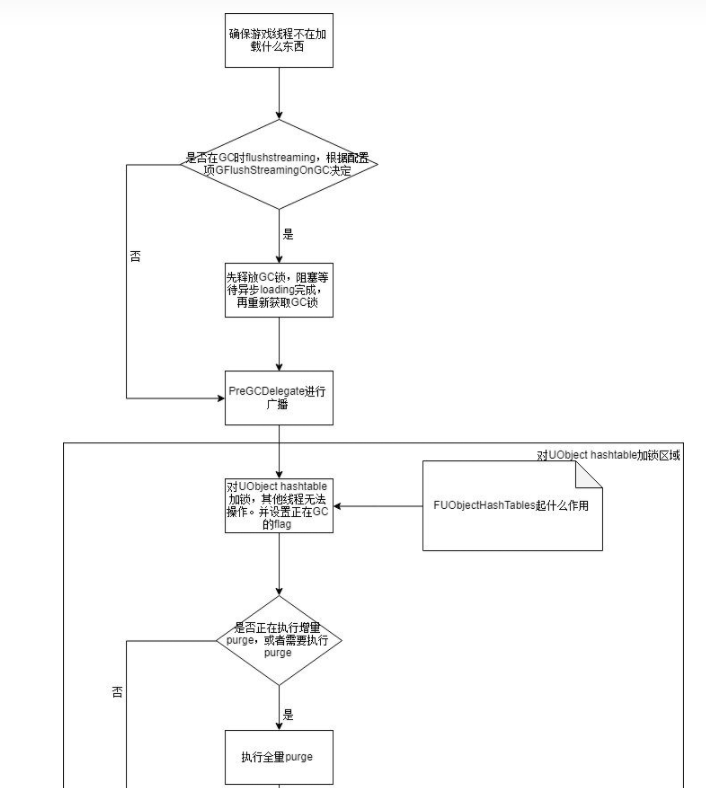
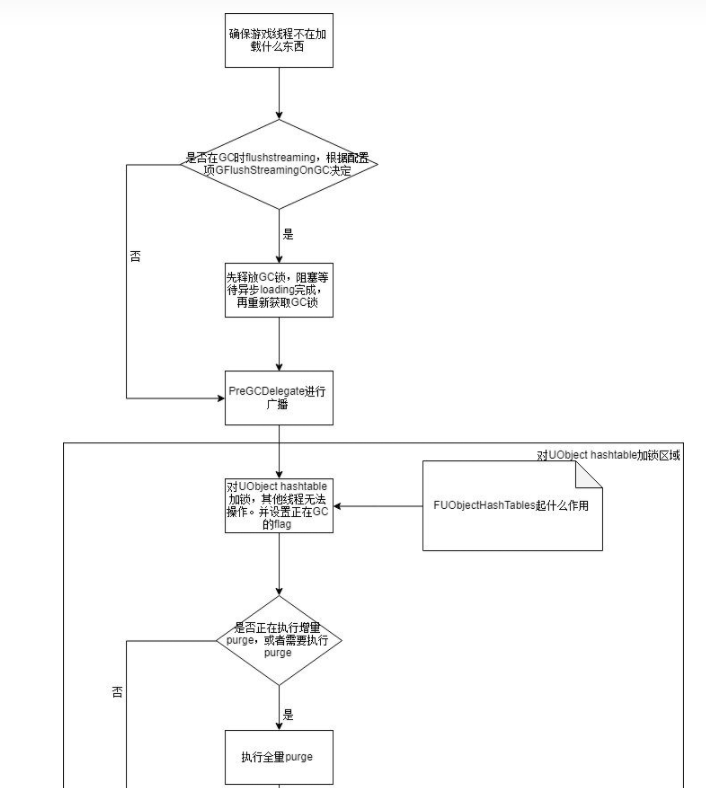
GC锁主要是为了防止其他线程对UObject做相关操作，会与GC冲突，这主要用于保护异步加载过程。

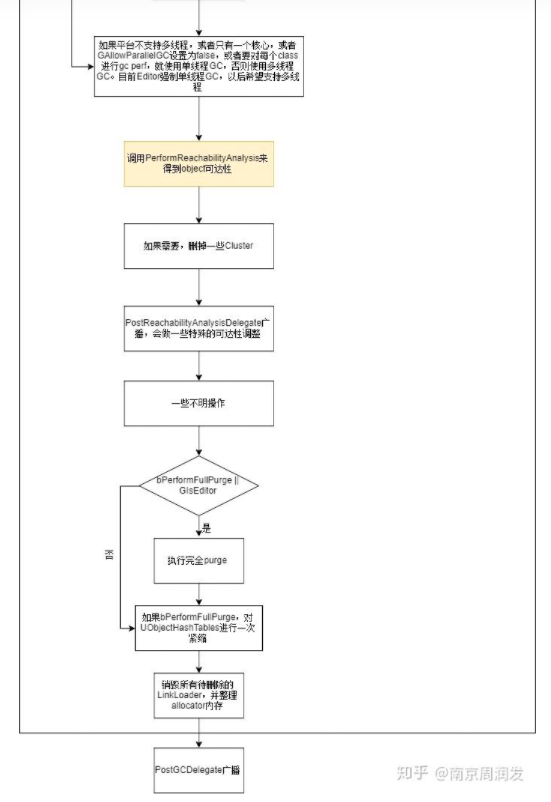
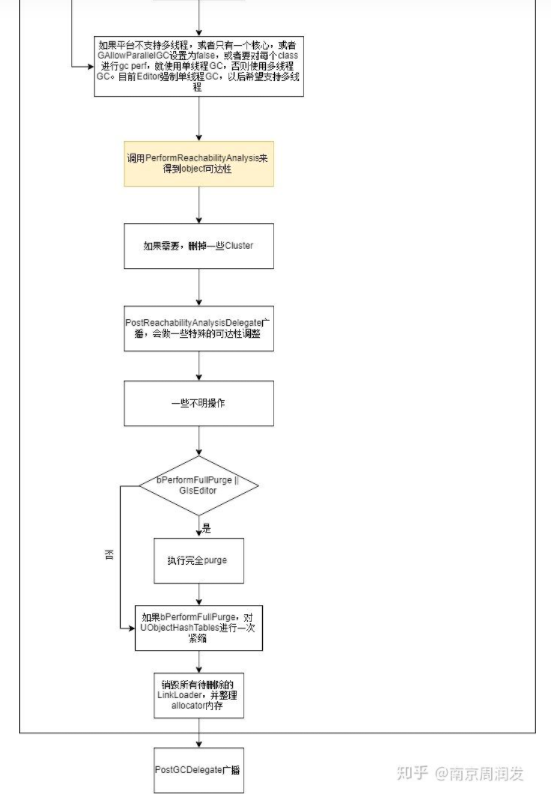
//删除所有未被引用的对象，保存那些有这些标记的对象

void CollectGarbageInternal(*EObjectFlags* KeepFlags, bool bPerformFullPurge)

{

}





**标记流程**

使用FRealTimeGC的PerformReachabilityAnalysis方法进行uobject可达性分析。

第一步，我们可以向ObjectsToSerialize添加FGCObject::GGCObjectReferencer

后者是一个静态的UGCObjectReferencer，添加后可用于在非UObject对象上调用AddReferencedObjects方法，可以用处理非Object对象的GC（需要继承自FGCObject）

第二步，调用MarkObjectsAsUnreachable（核心标记函数）把所有不带KeepFlags和EInternalObjectFlags::GarbageCollectionKeepFlags标记的对象标记为不可达（ObjectItem->*SetFlags*(*EInternalObjectFlags*::*Unreachable*)）

且对象必须处于PendingKill的状态下才能标记为不可达。

ue4 使用簇(cluster)来提高效率

第三步，调用PerformReachabilityAnalysisOnObjects来判断uobject可达性

搜索一遍所有的UObjects，找到不可达对象，对于不可达对象，如果不在Cluster中，就直接加入到GUnreachableObjects数组中，如果是ClusterRoot，则需要对它所包含的所有Objects进行分析，如果Object没有来自Cluster外部的引用，则也为不可达，需要加入到GUnreachableObjects数组中。

**清理流程**

1，调用UnhashUnreachableObjects方法，对在GUnreachableObjects数组中所有的不可达且未设置RF\_BeginDestroyed的UObject调用ConditionalBeginDestroy方法，可以让UObject进行一些异步的清理操作

2，IncrementalPurgeGarbage()执行清扫工作，执行清扫工作，遍历要清理的对象，调用这些对象的ConditionalFinishDestroy方法

3，执行真正的清理过程，遍历待清理对象，执行UObject的析构函数，并释放内存空间。检查是否超过了TimeLimit，超过的化下一帧继续清理。

**自动更新引用**

Actor或UActorComponent被销毁或者从运行中删除时，对反射系统可见的对它的所有引用（ UProperty`指针和虚幻引擎容器类中存储的指针，如`TArray`）都将自动清空。

 指向UObject的指针会在一个对象销毁时，自动把指向这个对象的UProperty指针更新为NULL（前提是这个指针需要被UPROPERTY宏标记，如果不标记的话，会造成悬空指针）

**非UObject引用的UObject指针**

因为本身是被非UObject引用，所以就不会在循环中被自动加入到全局的GUObjectArray列表中，除非我们在FGCObject类的AddReferencedObject函数中把它手动地加进去（然后这个方法会在循环中被自动地调用）

**一些示例**

1，在一个函数中，NewObject<T>，如果没有正确被其他对象持有（如AddToRoot等），会在下一次GC中被释放掉（是因为什么原因被释放掉的？）

2，在一个函数中，NewObject<T>，如果没有被带有UPROPERTY()标签的UObject\*指针持有，在下一次GC中也会被释放掉