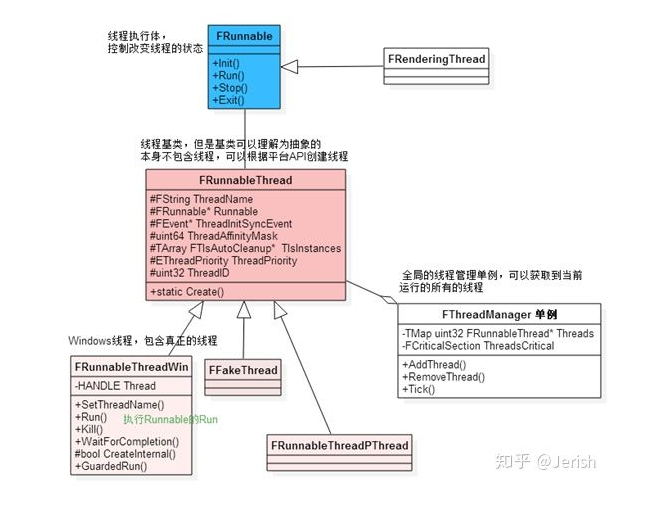
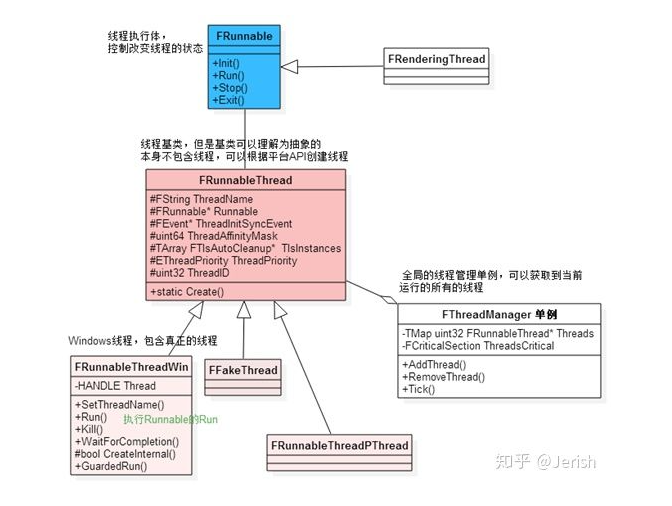
一、UE4中的多线程

ue4自己实现了一套多线程机制，用法上像Java。

二、如何使用UE4的多线程

①可以继承FRunnable接口创建单个线程

FRunnable只有5,6个函数接口，为“线程执行体”。



FRunnableThread是真正的线程，不同平台的线程都继承自他，如FRunnableThreadWin，里面会调用Windows平台的创建线程的API接口。

创建：

在构造函数中Runnable继承类的构造函数中创建

FSimpleRunnable(const *FString*& ThreadName)

{

        // 获取 FEvent 指针

        ThreadEvent = *FPlatformProcess*::*GetSynchEventFromPool*();

        // 创建线程实例

        m\_ThreadName = ThreadName;

        hreadIns = *FRunnableThread*::*Create*(this, \*m\_ThreadName, 0, *TPri\_Normal*);

        m\_ThreadID = ThreadIns->*GetThreadID*();

*UE\_LOG*(LogTemp, *Warning*, *TEXT*("Thread Start! ThreadID = %d"), m\_ThreadID);

}

②可以直接创建AsyncTask来调用线程池里面空闲的线程

1.AsyncTask是什么？

AsyncTask系统是一套基于线程池的异步任务处理系统，AsyncTask是放到线程池里面的异步任务（ 调用函数StartSynchronousTask默认使用UE的全局线程池，使用他不需要自己单开线程处理）

template<typename TTask>

class FAsyncTask:private IQueuedWork

FQueuedThreadPool:UE的线程池，和一般的线程池实现类似，线程池维护了多个线程FQueuedThread和多个队列QueuedWork，线程池在引擎PreInit的时候执行初始化操作

在线程池中所有的线程都是FQueuedThread类型（继承自FRunnable的线程）

通过FQueuedThreadPool的Create函数创建了多个FQueuedThread，然后每个FQueuedThread会执行Run函数

我们看到，当DoWorkEvent执行Wait的时候，如果该线程的Event处于无信号状态（默认刚创建是无信号的），那么wait会等待10毫秒并返回false，线程处于While无限循环中。如果线程池添加了任务（AddQueuedWork）并执行了DoWorkEvent的Trigger函数，那么Event就会被设置为有信号，Wait函数就会返回true，随后线程跳出循环进而处理任务。

实现方案：

class FSimpleAsyncTask:public FNonAbandonableTask

{

    friend class FAutoDeleteAsyncTask<FSimpleAsyncTask>;

    public:

    void DoWork();

    private:

    FORCEINLINE TStatId GetStatId() const

    {

        RETURN\_QUICK\_DECLARE\_CYCLE\_STAT(MyAsyncTask, STATGROUP\_ThreadPoolAsyncTasks);

    }

}

MyTestActor.cpp

FAutoDeleteAsyncTask<FSimpleAsyncTask>\* Task= new FAutoDeleteAsyncTask<FSimpleAsyncTask>();

Task->StartBackgroundTask();

③可以通过TaskGraph系统来异步完成一些自定义任务。

Task Graph 系统是UE4一套抽象的异步任务处理系统，可以创建多个多线程任务，指定各个任务之间的依赖关系，按照该关系来依次处理任务。

在引擎初始化FTaskGraphImplementation的时候，我们就会默认构建24个FWorkerThread工作线程（这里支持最大的线程数量也就是24），其中里面有5个是默认带名字的线程，StatThread、RHIThread、AudioThread、GameThread、ActualRenderingThread，还有前面提到的N个非指定名称的任意线程，这个N由CPU核数决定。对于带有名字的线程，他不需要创建新的Runnable线程，因为他们会在其他的时机创建，如StatThread以及RenderingThread会在FEngineLoop.PreInit里创建。而那N个非指定名称的任意线程，则需要在一开始就手动创建Runnable线程，同时设置其优先级比前面线程的优先级要低。到这里，我们应该可以理解，有名字的线程专门要做他名字对应的事情，非指定名称的任意线程则可以用来处理其他的工作，我们在CreateTask创建任务时会通过自己写好的函数决定当前任务应该在哪个线程执行。

实现方案：

class TestGraphTask

{

public:

TestGraphTask()

{

}

static const TCHAR\*GetTaskName()

{

    return TEXT("FMyTestTask");

}

FORCEINLINE static TStatId GetStatId()

{

    RETURN\_QUICK\_DECLARE\_CYCLE\_STAT(FMyTestTask, STATGROUP\_TaskGraphTasks);

}

*/\*\* return the thread for this task \*\*/*

static ENamedThreads::Type GetDesiredThread()

{

    return ENamedThreads::AnyThread;

}

static ESubsequentsMode::Type GetSubsequentsMode()

{

    return ESubsequentsMode::TrackSubsequents;

}

void DoTask(ENamedThreads::Type CurrentThread, const FGraphEventRef& MyCompletionGraphEvent)

{

    int Count = 0;

    while (true)

    {

        FWindowsPlatformProcess::Sleep(0.5);

        GEngine->AddOnScreenDebugMessage(-1,10.0f,FColor::Green,FString::FromInt(i));

        i++;

    }

}

private:

int32 i=0;

};

TestActor.cpp

TGraphTask<TestGraphTask>::CreateTask(NULL, ENamedThreads::GameThread).ConstructAndDispatchWhenReady();

FTaskGraphInterface::Get().WaitUntilTaskCompletes(Join, ENamedThreads::GameThread\_Local);

CreateTask的第一个参数就是该任务依赖事件数组（这里为NULL），如果传入一个事件数组的话，那么当前任务就会通过SetupPrereqs函数设置这些依赖事件，并且在所有依赖事件都触发后再将该任务放到任务队列里面分配线程执行。

ConstructuctAndDispatchWhenReady()可以向GraphTask传递参数

另外，不要在非GameThread线程内执行下面几个操作：

不要 Spawn / Modify/ delete UObjects or AActors

不要使用定时器 TimerManager

不要使用任何绘制接口，例如 DrawDebugLine

三、关于GameThread和RenderingThread的同步问题

渲染线程也需要等到GPU完成