UE4线程

Runnable线程

创建并执行线程

1. 创建类FTestTask并继承FRunnable，FRunnable提供了Init、Run、Stop和Exit方法。

```
class CORE_API FRunnable
{
public:

    /* Initializes the runnable object. ... */
    virtual bool Init(){...}

    /* Runs the runnable object. ... */
    virtual uint32 Run() = 0;

    /* Stops the runnable object. ... */
    virtual void Stop() { }

    /* Exits the runnable object. ... */
    virtual void Exit() { }

    /* Gets single thread interface pointer used for ticking this runnable w
    virtual class FSingleThreadRunnable* GetSingleThreadInterface( ){...}

    /** Virtual destructor */
    virtual ~FRunnable() { }
};
```

2. 实现方法

```
//当Run返回时，线程执行完成
virtual uint32 Run()
{
    //DO_TASK: while(1) {}
    return 0;
}
```

3. 在FTestTask中创建一个FRunnableThread线程

```
//声明线程
FRunnableThread* Task_Thread;

//创建线程
//1. 线程实例（this） 2. 线程名 3. 堆栈大小（0表示使用当前堆栈大小） 4. 优先级
void CreateThread()
{
    Task_Thread = FRunnableThread::Create(this, TEXT("HelloWorld"),  0,
TPri_Normal);
}
```

4. 创建并执行线程

```
FTestTask* NewTask = new FTestTask();
NewTask->CreateThread();
```

线程切换

```
virtual uint32 Run()
{
    while(true)
    {
        //Do work
        //切换线程
        FGraphEventRef Task = FFunctionGraphTask::CreationAndDispatchWhenReady(
            [&](){
                ThreadTaskDelegate.ExecuteIfBound();            //要在游戏线程执行的
方法
            }, TStatId(), Task, ENamedThreads::GameThread);
        //等待在游戏线程执行的任务完成
        FTaskGraphInterface::Get().WaitUntilTaskCompletes(Task);
    }
    return 0;
}
```

GraphTask线程

创建和销毁线程需要消耗一定的资源，所以还可以通过GraphTask利用闲置线程。

实现代码：

1. 创建FTestTask类，并实现相关接口

```
class FTestTask
{
public:

    FTestTask(float _f) : m_f(_f)
    {

    }

    static ESubsequentsMode::Type GetSubsequentsMode()
    {
        return ESubsequentsMode::TrackSubsequents;
    }

    FORCEINLINE TStatId GetStatId()
    {
        RETURN_QUICK_DECLARE_CYCLE_STAT(FTestTask,
STATGROUP_TaskGraphTasks);
    }
```

```
    //执行的任务
    void DoTask(ENamedThreads::Type CurrentThread, FGraphEventRef
Subsequents)
    {
        //Do work
    }

    //返回要使用的线程
    static ENamedThreads::Type GetDesiredThread()
    {
        return ENamedThreads::AnyThread;        //使用任意闲置线程
    }

private:
    float m_f;
};
```

2. 创建并执行线程

```
//ConstructAndDispatchWhenReady为延迟构造，可以传指针，不能传引用
TGraphTask<FTestTask>::CreateTask(NULL,
ENamedThreads::GameThread).ConstructAndDispatchWhenReady(4.5f);
```

https://neil3d.github.io/unreal/mcpp-fork-join.html

AsyncTask线程

1. 创建FTestTask类

```
class FTestTask : public FNonAbandonableTask
{
    friend class FAsyncTask<FTestTask>;

    int32 InstanceInt;

    FTestTask(int32 _InstanceInt) : InstanceInt(_InstanceInt)
    {

    }

    void DoWork()
    {
        //Do work
    }

    FORCEINLINE TStatId GetStatId() const
    {
        RETURN_QUICK_DECLARE_CYCLE_STAT(FTestTask,
STATGROUP_ThreadPoolAsyncTasks);
    }
};
```

2. 创建并执行线程

```
//创建线程任务
FAsyncTask<FTestTask>* TestTask = new FAsyncTask<FTestTask>(5);
//执行线程
TestTask->StartBackgroundTask();          //通过线程池调用个线程后台执行任务
TestTask->StartSynchronousTask();         //使用游戏线程同步执行任务
//判断任务是否完成
TestTask->IsDone();
//确保任务完成
TestTask->EnsureCompletion();
delete TestTask;
```

三种线程的区别

Runnable

可用复杂计算，例如网络访问、数据库访问等

GraphTask

调用闲置线程，支持任务顺序，即按顺序执行异步任务，适用于简单计算，但会占用游戏时间

AsyncTask

引擎启动后会在线程池创建10个线程，AsyncTask可以从线程池调用线程，可执行复杂计算

在异步线程不要执行SpawnActor, NewObject, Destroy UObject/Actor, DrawDebugline等方法，因为这些方法只能在主线程执行，并且通过断言而防止在其它线程执行。

线程安全

防止发生多线程对同一资源的访问冲突->互斥锁、条件锁、自旋锁、读写锁、递归锁、超时锁->频繁加锁可能会导致死锁->UE4解决方案：ESPMode::ThreadSafe、原子读取

指定线程中执行任务

该方法可以同步回游戏线程，也可以从线程池调用任意闲置线程，缺点是无法传参

该方法实际是通过调用TaskGraph启动线程的接口来调用线程操作的。

```
//在游戏线程执行任务
AsyncTask(ENamedThreads::GameThread, []()
{
    //TODO
});
```