# P1 UE4 的入口Main函数

1. 在WinMain(在LaunchWindows.cpp) 入口函数里面调用GuardedMain(在Launch.cpp)函数，下面主要介绍这个函数里面的东西
2. 用Delegate的方式进行了预初始化：  
   FCoreDelegates::GetPreMainInitDelegate().Broadcast();
3. 定义一个结构体用于出作用域后自动析构，调用EngineExit()
4. 进行引擎预初始化：  
   int32 ErrorLevel = EnginePreInit( CmdLine );
5. 判断是否为EDITOR，如是则调用EditorInit(*GEngineLoop*)，否则调EngineInit()，其实两者内部都会调*GEngineLoop*.Init()，只是EditorInit还初始化了编辑器的一些东西
6. 随后进入循环，调用EngineTick()
7. 当检测到IsEngineExitRequested()时，退出循环，最后调用EditorExit()，且出作用域自动调EngineExit()

# P2 UE4的引擎循环FEngineLoop::Tick()函数

FEngineLoop::Tick()(在LaunchEngineLoop.cpp)函数是GuardedMain(在Launch.cpp)函数里面的EngineTick()调用的，相当于每帧都会执行的一些操作。主要实现了以下操作：

1. 内存跟踪、LLM和一些Ensure和Check以及Log等工作
2. FCoreDelegates进行开始这一帧的广播：  
   FCoreDelegates::OnBeginFrame.Broadcast();
3. 当考虑最大tick率的时候更新FApp的当前时间和时间差：  
   GEngine->UpdateTimeAndHandleMaxTickRate();
4. 这是一个奇怪的函数，没看懂为啥：  
   GEngine->SetGameLatencyMarkerStart(CurrentFrameCounter);
5. 对每个WorldContext的Scene的所有primitive的状态进行更新：  
   Scene->UpdateAllPrimitiveSceneInfos(RHICmdList);
6. 开始RHI渲染线程，此函数根据不同平台调用不同的虚函数：  
   BeginFrameRenderThread(RHICmdList, CurrentFrameCounter);
7. 开始新的一帧，在内部把InternalFrameIndex++，有一个循环，把每一个PrimitiveComponent的VelocityData进行更新，并把bNeedsUniformBufferUpdate设为true，要在每个片元移动后重新创建UniformBuffer：  
   Scene->StartFrame();
8. 忽略动态分辨率在一帧开始时的操作（仅Engine）：  
   GEngine->EmitDynamicResolutionEvent(EDynamicResolutionStateEvent::BeginFrame);
9. 更新性能监控、内存跟踪、STAT等
10. 计算FPS：当前FPS与上一个FPS进行加权：  
    GAverageMS = GAverageMS \* 0.75f + FrameTime \* 0.25f;
11. 在渲染线程处理一些逐帧任务：  
    FlushPendingDeleteRHIResources\_RenderThread();
12. 根据情况判断是否停止tick以及停止渲染，其值存给bIdleMode  
    bIdleMode = ShouldUseIdleMode();
13. 允许在每次运动控制器输入之前设置一次world-to-meters  
    extern ENGINE\_API float GNewWorldToMetersScale;
14. 处理一些平台文件的操作：  
    FPlatformFileManager::Get().TickActivePlatformFile();
15. 轮询一些游戏控制器或者面板输入：  
    SlateApp.PollGameDeviceState();
16. Media做一些引擎之前的工作：  
    MediaModule->TickPreEngine();
17. 主要的游戏和引擎Tick函数，以后会细致分析这个函数：  
    GEngine->Tick(FApp::GetDeltaTime(), bIdleMode);  
    GEngine是UEngine\*类型，主要有两个派生类：UEditorEngine和UGameEngine，根据多态的规则来确定调哪个Tick函数，两者Tick函数相似，最终都会调用Viewport->Draw()来绘制
18. 处理与movie的冲突，因为刚好在GEngine->Tick里面有可能会有用户代码，里面可能会涉及到movie的操作，所以紧随其后，等待movie操作完毕，主要根据判断是否有PreLoadScreen来看是先加载movie还是PreLoadScreen
19. 处理异步ShaderCompiling：  
    GShaderCompilingManager->ProcessAsyncResults(true, false);
20. 处理异步DistanceFieldAsyncQueue：  
    GDistanceFieldAsyncQueue->ProcessAsyncTasks();
21. Media做一些Slate之前的工作：  
    MediaModule->TickPreSlate();
22. 做一些Slate的输入的一些操作，主要是在网络同步之前做的：  
    ProcessLocalPlayerSlateOperations();
23. 用UDemoNetDriver构建虚拟网络驱动，用于记录或者回放。
24. 处理automation controller（仅Editor）和worker（仅Engine）  
    FModuleManager::GetModuleChecked<IAutomationControllerModule>(AutomationController).Tick();  
    FModuleManager::GetModuleChecked<IAutomationWorkerModule>(AutomationWorkerModuleName).Tick();
25. 处理RHI的操作，根据不同的平台调不同的虚函数：  
    RHITick( FApp::GetDeltaTime() ); // Update RHI.
26. 累加全局的帧计数，忽略前面的几帧不计入TotalTickTime：  
    GFrameCounter++;
27. 找到下一帧需要被清理的对象：  
    PendingCleanupObjects = GetPendingCleanupObjects();
28. 进行帧同步，即游戏线程和渲染线程的同步：  
    FrameEndSync.Sync( CVarAllowOneFrameThreadLag->GetValueOnGameThread() != 0 );
29. 处理一些关于core ticker, threads & deferred commands的操作，并且删除上一帧的删除对象PreviousPendingCleanupObjects
30. Media的一些后处理工作：  
    MediaModule->TickPostRender();
31. FCoreDelegates进行结束这一帧的广播：  
    FCoreDelegates::OnEndFrame.Broadcast();
32. 忽略动态分辨率在一帧结束时的操作（仅Engine）：  
    GEngine->EmitDynamicResolutionEvent(EDynamicResolutionStateEvent::EndFrame);
33. 结束RHI渲染线程：  
    EndFrameRenderThread(RHICmdList, CurrentFrameCounter);
34. 这是一个奇怪的函数，没看懂为啥：  
    GEngine->SetGameLatencyMarkerEnd(CurrentFrameCounter);
35. 最后设置一些STAT的参数，如CPU utilization、UObject count

# P3 UE4的UEditorEngine::Tick和UGameEngine::Tick函数

这两个函数主要是在LaunchEngineLoop.cpp的FEngineLoop::Tick()的GEngine->Tick通过多态调用了，还有其他的重载如：UUnrealEdEngine::Tick和UDisplayClusterEditorEngine::Tick以及UDisplayClusterGameEngine::Tick。此处主要分析UGameEngine::Tick并以此为基础来看与其他重载的不同之处。  
UGameEngine::Tick主要有以下实现：

1. 检查输入DeltaSeconds的有效性
2. 清除已经关闭的Viewports（三种情况下跳过：commandlet，dedicated server，headless）：  
   CleanupGameViewport();
3. 如果当前没有Viewports则Exit()（headless下跳过）
4. 根据DeltaSeconds来设置是否降低细节，其内部主要是计算了当前的framerate来与阈值比较：  
   GameViewport->SetDropDetail(DeltaSeconds);
5. 处理一些子系统，如：ProcessAsyncLoading，*EngineAnalytics和StudioAnalytics*
6. 保存当前对于Gworld的WorldList中的元素(WorldContext)为OriginalGWorldContext
7. 进入WorldList的循环，对每个元素执行以下操作：
8. 将元素存到GWorld：  
   GWorld = Context.World();
9. 处理travel以及一些NetNames（关于漫游和网络同步的一些操作）：  
   TickWorldTravel(Context, DeltaSeconds);
10. 对World实际进行Tick，实际上是对Levels进行tick，这个函数之后会详细分析  
    Context.World()->Tick( LEVELTICK\_All, DeltaSeconds );
11. 更新SkyCapture和ReflectionCapture：  
    USkyLightComponent::UpdateSkyCaptureContents(Context.World());  
    UReflectionCaptureComponent::UpdateReflectionCaptureContents(Context.World());
12. 加载一次Player，在加载一次世界时仅仅运行一次：  
    GamePlayer->Exec( GamePlayer->GetWorld(), \*(FString("CAUSEEVENT ") + InitialExec), \*GLog );
13. 更新过渡的方式：（判断现在处于什么过渡方式：connecting，pause等，并决定是否更改）  
    UpdateTransitionType(Context.World());
14. 根据需求来阻止异步加载：  
    BlockTillLevelStreamingCompleted(Context.World());
15. 更新LevelStreaming：  
    Context.World()->UpdateLevelStreaming();
16. 做一些切换地图的操作：  
    ConditionalCommitMapChange(Context);
17. 对tickable的对象进行处理：  
    FTickableGameObject::TickObjects(nullptr, LEVELTICK\_All, false, DeltaSeconds);
18. 将上面存储的OriginalGWorldContext又存到GWorld：  
    GWorld = GetWorldContextFromHandleChecked(OriginalGWorldContext).World();
19. 对Meida进行处理：  
    MediaModule->*TickPostEngine*();
20. 对Viewport进行Tick：（里面用delegate的方式进行广播）：  
    GameViewport->Tick(DeltaSeconds);
21. 隐藏闪屏并打开游戏窗口
22. Render所有的viewport并且做一些渲染后的操作：（此函数后续会进一步分析）：  
    RedrawViewports();（内部实际调用：GameViewport->Viewport->Draw）  
    GetRendererModule().PostRenderAllViewports();
23. 对资源流做一些处理：  
    IStreamingManager::Get().Tick( DeltaSeconds );
24. 更新音频：  
    GameAudioDeviceManager->UpdateActiveAudioDevices(bIsAnyNonPreviewWorldUnpaused);
25. 给渲染线程发送指令：  
    （仅Editor）处理一些Editor后面的操作和资产注册

UEditorEngine::Tick与UGameEngine::Tick的不同之处（简称Editor和Game）：

1. Editor把GameViewport->Tick，Viewport->Draw(即Render)，GameViewport->SetDropDetail放在了WorldList的循环里面，而Game在外面；
2. Editor在没有viewport的时候调用的是EndPlayMap()，只是清理资源关闭了窗口，并没有完全退出主程序（编辑器还在），而Game则是直接退出主程；
3. Editor还有一个FEditorViewportClient::Tick，这个语句在一个Viewport的循环里面，对每一个视口主要做一个相机变换的更新。Game关于这个的实现暂时没找到
4. Editor还考虑了窗口被遮挡时的绘制，Game没有
5. 此外Editor还多了很多Game没有的操作，此处不赘述