**<Project TargetActor>**

// 코드가 계속 늘어나고 구조가 조금씩 변하기 때문에 핵심 로직만 작성하고 자세한 건 코드를 봐야한다.

**\*\* 프로젝트 구조**

\* 싱글 프로세스 멀티스레드 구조

로딩 없는 오픈 월드 형식의 MMORPG 게임을 만들 예정이다.

채널, 월드이동 로딩이 존재하지 않는다.

\* ECS 구조(Entity Component System 구조)

보통 많이 사용하는 액터가 컴포넌트를 들고 있는 형태가 아닌, 컴포넌트와 액터를 분리해서 사용한다.

그리고 이 컴포넌트들은 시스템이 모아서 처리한다.

이 방식을 사용하게 되면, 메모리 지역성 특징 때문에 캐시 미스 확률이 떨어져서 성능이 올라가고, 각 시스템은 공유자원에 대해서 최대한 독립적으로 동작하도록 방향을 잡는다. (물론 어쩔 수 없이 공유자원에 대한 동기화가 필요한 경우도 생긴다.)

그리고 데이터와 기능을 분리하여 코드 가독성을 높인다.

\* 공통된 코드 관리 구조

서버와 클라이언트간 공유해야 할 코드가 무조건 존재한다. 공유 코드는 Common…으로 만들고 서버 클라에서 해당 코드를 상속받아 Server…, Client… 형식으로 작성한다.

\* 네트워크 통신 구조

IOCP와 Overlapped I/O를 이용한 통신 구조를 가지고 있고, 각 액터의 키를 컴플리션 키로 등록하고, 해당 패킷을 처리한다. 여러 워커스레드에서 IOCP를 대기하고, 처리한다.

RPC통신을 한다. IDL 파일을 컴파일하여 생성되는 함수로 서로 통신하게 된다. 생성되는 함수만 구현하면 된다.

**\*\* 패턴 / 구현기법**

\* 컴파일 의존성을 최소화

프로젝트가 커지다 보면 하나의 해더파일만 수정해도 다른 해더파일들이 다시 컴파일 되는 현상이 나타난다. 그래서 컴파일 타임을 최소화시키기 위해서 특정 기능을 가진 객체(ex) NetworkObject 등)를 가질 때 포인터로 선언하고 전방선언을 해서 의존성을 최소화한다. 함수에서도 마찬가지로 레퍼런스형이나 포인터형을 받아서 전방선언한다. 이 방식말고 가상함수로 컴파일 의존성을 낮추는 방식도 존재하긴 하는데, 전자가 좀더 직관적이라고 생각해서 해당 방식을 사용한다.

\* 풀 사용

실시간으로 생성 삭제를 하게 되면, 일단 지속적으로 비용이 발생하게 된다. 또한 여러 곳에서 참조하고 있다면 크래시 날 확률이 높다. 크래시는 shared\_ptr같은 자료구조를 사용해서 막으면 되긴 하지만, 락걸고 사용해야 하기 때문에 성능이 너무 떨어지며, 락을 안 걸면 멀티스레드 환경에서 안전하지 않다.

풀을 사용해서 삭제된 메모리에 접근할 확률을 없애고, 대신 유효한지 플래그를 두어서 해당 플래그를 읽고 사용하도록 하였다.

풀에서 빠져나온 객체를 따로 관리하는 unordered\_set은 만들지 않았다. 만들게 되면 여러곳(거리가 멀든 가깝든)에서 액터를 검색할 때 모두 락을 걸고 들어와서 찾기 때문이다. 대신 Sector, ViewList개념을 만들어서 특정지역 액터들을 관리한다.

\* 상호배제

일단 성능이 가장 괜찮았던 것은 SRWLock(Slim Reader Writer Lock) 이였다. 기본적으로도 성능이 좋지만, 이를 사용하면 읽기에 대해서 락을 걸 수 있기 때문에, 여러 곳에서 조회를 빠르게 해야 하는 경우(ex) Sector에서 검색) 성능이 좋다.

또 해당 락을 사용할 때 lock / unlock를 쌍으로 안 지키거나 여러 개의 락을 걸어야 하는 상황에서 코드실수로 데드락이 발생할 수 있기 때문에, ScopedLock이라는 클래스를 만들어서 해당 스코프안에서만 락을 얻고, 스코프를 빠져나가면 락을 반납하는 기능을 만들어서 사용한다.

\* Worker Thread

네트워크 통신관련 작업과 여러가지 작업을 같이 처리한다. 네트워크 작업은 알아서 스레드가 깨어나서 처리되고, 다른 작업들(Timer관련(일정 주기마다 AI수행 / 일정주기마다 몬스터 스폰 등 ), 특정이벤트(AIEvent 등))은 PostQueuedCompletionStatus를 이용하여 스레드를 깨워서 처리한다.

이렇게 하는 이유는 스레드를 여러 개로 안나누고 하나의 스레드에서 모두 처리하게 하기 위해서이다. 그래야 필요한 곳에 스레드 자원을 더 나눠줄 수 있다.

\* NonBlocking 자료구조

여러 스레드에서 한번에 접근해도 안전한 자료구조를 만들어서 락으로 인한 성능하락을 막아준다. CompareAndSet기법을 사용해서 구현된다.

간단한거 구현하기도 조금 복잡하고 여러가지 이슈가 생길 수 있다. 그래서 현실적으로 모든 곳에 적용할 수 없고 간단한 곳에 적용해보려 한다. (예를 들어 액터A의 ViewList 락걸고 접근 안하고, 논블로킹 메시지큐를 만들어서 그 큐를 액터A가 처리하게 한다.)

구현은 했지만 완전하지 않아서 아직 사용 못한다.

**\*\* Xml Data / Excel**

\* Excel : Xml Exporter

엑셀에서 Visual Basic을 사용하여 간단하게 해당 표를 Xml 형식으로 내보내는 기능을 만들었다.

\* Xml Parser

**\*\* AI**

\* Behavior Tree 언리얼 엔진과 비슷하게 구현해서 사용

**\*\* Unreal Engine 정보 (Unreal Engine Documents 참고한 내용)**

언리얼 기법/////////////////////////////////////////////////////////////////////////////////////////////////////

\* Reflection

- 정의

이미 로딩이 완료된 클래스에서 또 다른 클래스를 동적으로 로딩하여 클래스 내부 멤버를 사용할 수 있게 해준다. 런타임에서 동적으로 특정 클래스의 정보의 객체화를 통해 분석 및 추출해낼 수 있는 프로그래밍 기법이다.

- 세부

실행 시간에 다른 클래스를 동적으로 로딩하여 접근할 때, 클래스와 그 멤버에 관련된 정보를 얻어야 할 때 등 사용한다. 없더라도 구현할 수 있지만 사용하면 좀 더 유연하게 구현할 수 있다.

언리얼에도 해당 기능을 사용하고 있으며, Garbage Collection, Blueprint / C++ 통신, Serialization, Network Replication등에 사용된다고 나와있다.

\* 레퍼런스 자동 업데이트

AActor 또는 UActorComponent 가 소멸되거나 다른 식으로 플레이에서 제거되면, 리플렉션 시스템에 보이고 있는 해당 모든 레퍼런스는 자동으로 null 로 초기화 된다.

다른 곳에서AActor 와 UActorComponent 포인터를 소멸시키는 경우 null 이 된다. 허상 참조할 가능성이 없어지기 때문에 안정성이 보장된다.

단, 이 기능은 UPROPERTY로 마킹되어 있거나 언리얼 컨테이너 클래스에 저장된 UActorComponent또는 AActor레퍼런스에만 적용된다. Raw 포인터에 저장된 오브젝트 레퍼런스는 엔진에서 알지 못한다. 그렇기 때문에 UObject\* 변수가 UPROPERTY가 아닌 경우에는 TWeakObjectPtr 사용하면 된다.

// TWeakObjectPtr를 사용할 때 한 개의 스레드에서만 소멸되고 참조하고 하면 상관없을텐데 만약 여러개의 스레드에서 접근해서 소멸시키면 문제가 될 수 있다. 그럴경우 TSharedObjectPtr을 사용해서 가비지컬렉션에 의해 삭제되는 것을 막아야 할듯하다.

// 게임 루프 자체는 싱글 스레드로 돌아가는 것 같다.

\* Garbage Collection

- 정의

게임엔진에서 더 이상 다른 객체에 의해 참조되지 않는 객체를 자동으로 삭제하게 하는 프로그래밍 기법이다.

- 세부

메모리관리를 위해서 사용되는 기법 중 하나다. 언리얼에서도 이 방법을 사용하고 있다.

언리얼에서 레퍼런스 그래프를 만들어 어느 오브젝트가 사용중인지 판단한다. 이 그래프의 루트에는 루트세트라는 지정된 오브젝트 세트가 있고 추가할 수 있다. 가비지 컬렉션이 발생하면, 엔진은 루트세트부터 시작해서 UObject 레퍼런스 트리를 검색해서 참조된 오브젝트를 전부 추적한다. 참조되지 않은 오브젝트는 더 이상 필요치 않다고 판단하고 삭제한다.

삭제를 막기 위해서는 해당 오브젝트에 UPROPERTY 레퍼런스를 유지하거나, 그에 대한 포인터를 TArray또는 다른 언리얼 컨테이너 클래스에 저장해야 한다. 이렇게 하면 일방적으로 삭제되고 허상포인터로 남지않고 생명주기를 같이할 수 있다. 루트세트에 링크되는 오브젝트나 액터자체에 레퍼런싱되는 컴포넌트는 예외다.  Destroy 함수를 호출해서 소멸 마킹할 수 있다.

다음과 같은 상황에서 TWeakObjectPtr을 사용해야해야 할 것 같다. 예를 들어 내가 UPROPERTY 레퍼런스로 targetActor로 가지고 있으면 targetActor를 null로 초기화하지 않으면 해당 targetActor가 소멸 마킹되어도(Destroy함수가 불려도) 가비지 컬렉터에서 삭제못한다는 뜻이 된다. 이럴때 TWeakObjectPtr로 처리하면 될 듯 하다. 간단히 말하면 소유권이 필요한 상황이 아닐 때 사용하면 된다.

참고로 TWeakObjectPtr : UObjects사용 / TWeakPtr : 나머지사용 이다. Shared도 마찬가지

참고 : 멀티스레드에서 돌아가는지 확인해보려고 했는데 콜스택이 다음과 같이 찍히는 걸로봐서 싱글스레드(게임스레드 , 메인스레드)에서 돌아간다.

> UE4Editor-CoreUObject.dll!TryCollectGarbage(EObjectFlags KeepFlags, bool bPerformFullPurge) Line 2073 C++

UE4Editor-Engine.dll!UEngine::PerformGarbageCollectionAndCleanupActors() Line 1409 C++

UE4Editor-Engine.dll!UEngine::ConditionalCollectGarbage() Line 1383 C++

UE4Editor-Engine.dll!UWorld::Tick(ELevelTick TickType, float DeltaSeconds) Line 1758 C++

UE4Editor-UnrealEd.dll!UEditorEngine::Tick(float DeltaSeconds, bool bIdleMode) Line 1515 C++

UE4Editor-UnrealEd.dll!UUnrealEdEngine::Tick(float DeltaSeconds, bool bIdleMode) Line 414 C++

UE4Editor.exe!FEngineLoop::Tick() Line 4850 C++

[Inline Frame] UE4Editor.exe!EngineTick() Line 62 C++

UE4Editor.exe!GuardedMain(const wchar\_t \* CmdLine) Line 169 C++

UE4Editor.exe!GuardedMainWrapper(const wchar\_t \* CmdLine) Line 137 C++

UE4Editor.exe!WinMain(HINSTANCE\_\_ \* hInInstance, HINSTANCE\_\_ \* hPrevInstance, char \* \_\_formal, int nCmdShow) Line 268 C++

\* Network Replication

UObject 시스템에는 네트워크 통신과 멀티플레이어 게임을 위한 함수 세트가 있다.

UProperty에는 태그를 붙여서 네트워크 플레이 도중 데이터의 리플리케이트 여부를 엔진에게 알릴 수 있다. 간단히 말하면 서버에서 변수변경 => 모든 클라이언트에서 전송 , 클라이언트에서는 변수 변할 때 콜백으로 받아서 처리한다.

UFunction 역시 태그를 붙여 원격 머신에서 실행시킬 수 있다. 예를 들어 “server”함수는 클라이언트 머신에서 호출 시 실제로 서버 머신에서 해당 액터의 서버 버전이 실행되도록 한다. 반대로 “client”함수는 서버에서 호출 될 수 있지만, 해당 액터의 소유중은 클라인트 버전이 실행된다. (RPC)

// 현재 개발중인 프로젝트에는 자체 RPC가 사용되고 있어서 따로 사용하진 않을 것이다.

시행착오 ///////////////////////////////////////////////////////////////////////////////////////////////////////

\* 서버로부터 Spawn / Unspawn 지시 받았을 때

여러 스레드에서 한번에 스폰 언스폰하는 것은 thread safe하지 않아서 불가능하다. 언리얼 게임스레드에서 처리해줘야한다. 다른 스레드에서 게임인스턴스에 이벤트를 넘겨주면 인스턴스가 틱마다 이벤트 큐에서 확인해서 빼서 처리해야할 것 같다. 큐는 언리얼 내부 락프리 자료구조이기 떄문에 락은 따로 걸지않는다.

\* ActorSpawn / NewObject / CreateDefaultSubobject 와 생성자 BeginPlay

- ActorSpawn / NewObject

: 생성자에서 사용하면 안된다. 정확하게 디버깅은 안해봤지만 생성자가 호출되는 시점은 최초 UClass객체를 만들고 CDO를 만드는 시점이기 때문에, 월드라든지 조건이 충족 안되어 있어서 그럴 확률이 높다. 사용해야한다면 BeginPlay에 사용하자.

- CreateDefaultSubobject

: 생성자에서 사용해서 컴포넌트를 생성해주면된다.

액터 / 컴포넌트 정보//////////////////////////////////////////////////////////////////////////////////////////

\* Pawn vs Character

기본적으로 필요한 것

=> 충돌 : 캡슐 컴포넌트 / 시각 : 스켈레탈 메시 컴포넌트 / 움직임 : 무브먼트 컴포넌트

카메라 : 카메라 컴포넌트, 스프링암 컴포넌트

Character는 캡슐 / 스켈레탈 메시 / 캐릭터 무브먼트 컴포넌트 가지고 있고 Get~함수로 접근가능

\* Pawn <=> Controller

플레이어가 로그인 완료 => PostLogin 이벤트 함수 호출 => 플레이어가 조종할 폰을 생성하고 빙의

폰과 플레이어 컨트롤러가 생성되는 시점은 각 액터의 PostInitializeComponents함수로 파악 가능

빙의하는 시점은 컨트롤러의 Possess / 폰의 PossessedBy로 파악 가능

\* Pawn vs Controller

Controller는 플레이어의 의지를 전달한다. 게임의 물리적인 요소를 고려하지않은 플레이어 의지와 관련된 정보를 관리한다. 예를 들어 플레이어의 의지를 나타내는 Control Rotation이라는 속성을 제공

Pawn은 Controller와 반대로 게임의 물리적인 상황을 관리한다. 예를 들어 속도를 관리한다. PawnMovementComponent는 그냥 Pawn에게 다시 전달해준다.

결론은 회전은 Controller를 통해서(AddControllerXXXInput) 일반적인 움직임은 Pawn를 통해서(AddMovementInput) 한다.

\* Pawn, Controller 기타 정보

- Pawn

bUseControllerRotationYaw : 컨트롤 회전의 Yaw회전 값과 폰의 Yaw회전값은 기본적으로 연동 되어있는데, 이 플래그를 false로 주면 컨트롤러에서 폰으로 반영되지 않는다.

GetControlRotation() : 컨트롤 회전을 컨트롤러로부터 받아온다.

- Controller

ControlRotation : 컨트롤 회전값으로 (0,0,0)이면 액터가 바라보는 방향이 X축방향(1,0,0)이다.

\* SpringArmComponent

bInheritXXX : 플래그 켜면 부모의 로테이션을 받아온다. // Roll Pitch Yaw

bUsePawnControlRotation : 사용시 자체회전 불가, 폰의 컨트롤 로테이션을 받아온다.

\* CharacterMovementComponent

bOrientRotationToMovement : 캐릭터가 움직이는 방향으로 자동으로 회전시켜준다. 회전이 먼저 되고 거기에 따라서 캐릭터를 움직일 때 캐릭터 회전 부드럽게 하기 위해서 사용

bUseControllerDesiredRotation : 컨트롤러의 로테이션 값이 바뀌면 캐릭터를 부드럽게 회전시켜서 이동한다. 회전을 하면서 캐릭터가 부드럽게 이동하기 위해서 사용

RotationRate : 회전속도를 사용하여 부드럽게 회전

JumpZVelocity : 점프 속도

IsFlying() / IsMovingOnGround() / IsFalling() / IsSwimming() / IsCrouching() : 현재상태 판단

\* UPROPERTY

- 아키타입

컨텐츠 브라우저 상에 표시되는 블루프린트 클래스

ArcheType = Blueprint asset == Prefab (Unity3D)

- 인스턴스

아키타입을 통해 생성된 객체

- Visible ~ (AnyWhere, DefaultsOnly, InstanceOnly)

편집없이 보여주기만 가능

- Edit ~ (AnyWhere, DefaultsOnly, InstanceOnly)

편집가능

- (Visible, Edit) ~ AnyWhere

아키타입 또는 인스턴스 모두 적용

- (Visible, Edit) ~ DefaultsOnly

아키타입만 적용

블루프린트 편집화면에서만 보여진다.

- (Visible, Edit) ~ InstanceOnly

인스턴스만 적용

인스턴스의 속성을 보여주는 에디터 뷰포트에서만 보여진다

- BlueprintReadOnly

블루프린트상에서 읽기만 가능

- BlueprintReadWrite

블루프린트상에서 읽기/쓰기 가능

- meta(XXX = value, …)

meta(Allow(Private or Proctected)Access = true) : private 나 protected변수를 외부에 노출

\* Pawn vs Controller

Controller는 플레이어의 의지를 전달한다. 게임의 물리적인 요소를 고려하지않은 플레이어 의지와 관련된 정보를 관리한다. 예를 들어 플레이어의 의지를 나타내는 Control Rotation이라는 속성을 제공

\* Collision

- Collision Enabled

Query : 두 물체의 충돌 영역이 서로 겹치는지 테스트하는 설정, 충돌 영역의 겹침을 감치하는 것을 오버랩이라고 부른다. 충돌영역이 겹치면 해당 컴포넌트의 BeginOverlap이벤트가 발생한다.

Physics : 물리 시뮬레이션 관련

Query and Physics : 둘 다 적용

- Collision Channel과 Response

콜리전 채널이라는 개념이 있어서 해당 채널과 어떻게 반응할지 정해줄 수 있다.

Object Channel : 콜리전 영역에 지정하는 콜리전 채널

Trace Channel : 어떤 행동에 설정하는 콜리전 채널

Response의 종류는 다음과 같다.

Ignore : 아무 충돌이 일어나지 않는다.

Overlap : 물체가 뚫고 지나갈 수 있지만, 이벤트를 발생시킨다.

Block : 물체가 뚫고 갈 수 없다.

- Collision Event

Overlap 이벤트 : 겹침 반응에서 이벤트가 발생한다. // 블록인 경우에도 Overlap이벤트도 발생가능

Hit 이벤트 : 블록 반응에서 이벤트가 발생한다.

Query : 두 물체의 충돌 영역이 서로 겹치는지 테스트하는 설정, 충돌 영역의 겹침을 감치하는 것을 오버랩이라고 부른다. 충돌영역이 겹치면 해당 컴포넌트의 BeginOverlap이벤트가 발생한다.

Physics : 물리 시뮬레이션 관련

Query and Physics : 둘 다 적용

\* UMG

- 언리얼 모션 그래픽 UI 디자이너

- Widget Blueprint

UMG.UserWidget을 상속받은 정의클래스를 상속받은 블루프린트

보통 캔버스에 여러가지 UI를 추가하여 사용할 수 있다.

절대적으로 UI를 만들때는 캔버스에 보통 추가해서 만들고, 상대적인 UI를 만들때는 캔버스를 삭제한다.

예를들어 HUDWidget C++클래스를 만들었다고 가정하면, HUDWidget을 상속받은 HUDWidgetBlueprint가 존재하고 HUDWidget의 코드를 구현할 때 HUDWidgetBlueprint에 있는 UI들을 이름으로 찾아서 컨트롤 하면 된다.

다만 사용하는 코드에서 HUDWidget객체를 생성할 때 HUDWidgetBlueprint를 통해서 생성해야한다.

**\*\* Unreal Engine Base // 이득우님 블로그 참고 // 공부목적**

\* 재생성

언리얼 빌드 툴에 의해서 환경(윈도우, 맥 등)에 따라서 적절하게 다시 생성되기 때문에 Config, Content , Plugins, Source 폴더와 uproject 파일을 제외한 것들은 지워도 된다.

\* Unreal Build Tool

언리얼 빌트 툴이 실행되면 프로젝트의 Source폴더를 찾고 그 폴더아래에 빌드 할 대상 정보를 지정한 Target.cs파일을 조사하면서 하위에 모듈이 있는지 검사한다. 모듈은 항상 폴더 단위로 나뉘어 있어야 하며, 언리얼 엔진이 지정한 특정한 룰을 따라야 한다.

Source폴더의 조사가 끝나면 Intermediate폴더 안의 ProjectFiles폴더에 자신이 조사한 언리얼 소스 구조를 정리해 프로젝트 파일을 생성함과 동시에 언리얼 엔진 설치 폴더에 언리얼 엔진 설치 폴더로부터 소스코드에 대한 프로젝트 파일도 복사해온다.(엔진 모듈관련)

최종적으로 이 둘을 합친 솔루션 파일을 생성해준다.

\* 모듈 / 빌드

- Module

언리얼 엔진모듈과 사용자 게임모듈로 구성되어 있다.

게임제작에 사용되는 로직을 담은 기본 모듈을 주 게임 모듈이라고 한다. 처음 프로젝트 만들고 클래스 추가하면 생성된다. 우리가 클래스를 추가하면 프로젝트명으로 자동으로 사용자 게임 모듈이 생성된다.

- Module 만들기

<필요한것>

1. 모듈의 이름과 동일한 폴더

2. 모듈의 빌드 규칙 파일 : 모듈이름.Build.cs

3. 프리컴파일드 헤더와 소스파일 : 모듈이름.h , 모듈이름.cpp

<과정>

1. 필요한것들 모두 생성 // 모듈이름.cpp에서 IMPLEMENT\_MODULE사용하는 것만 주 게임모듈과 다름

2. 프로젝트명.Target.cs, 프로젝트명Editor.Target.cs에 해당 모듈 추가해야 dll 생성한다.

3. 기본적으로 Public(외부 모듈에 공개할 파일)폴더에 해더, Private(내부 모듈에서 사용할 파일)폴더에 소스를 분리시켜준다.

PublicDependencyModuleNames : 헤더파일이 참고할 모듈

PrivateDependencyModuleNames : 소스코드에서만 참고할 모듈추가하면 생성된다

- Solution Configurations

Development Editor / Development 거의 둘 중 하나 사용한다. Editor붙은 버전은 디버깅이 가능한 개발버전으로 컴파일해서 에디터용 dll 생성, Editor없는 설정은 게임 패키징용 exe파일 생성

- Target.cs

빌드 구성을 지정하는데 사용되는 설정 파일이다. 에디터용은 Type : Editor / 게임용은 Type : Game

Type : Server / Program 도 존재 // 엔진에서 보조로 사용할 콘솔 프로그램 제작 / 서버 빌드 제작 사용

- 컴파일 과정

언리얼 엔진에서는 바로 컴파일을 진행하는게 아니라 먼저 오브젝트 선언된 것들을 찾아서 언리얼 헤더 툴(UHT)에 의해 파싱하고 부가적인 메타 정보를 담은 소스코드를 프로젝트 intermediate폴더에 생성해 준다. 이 작업이 끝나면 본격적으로 컴파일한다.

\* Unreal Object / UClass

- Unreal Object

U~로 시작하며 NewObject<클래스>(); 형식으로 시작한다.

1. CDO(Class Default Object) : 객체의 초기 값을 자체적으로 관리합니다.

2. Reflection : 객체 정보를 런타임에서 실시간 조회가 가능합니다.

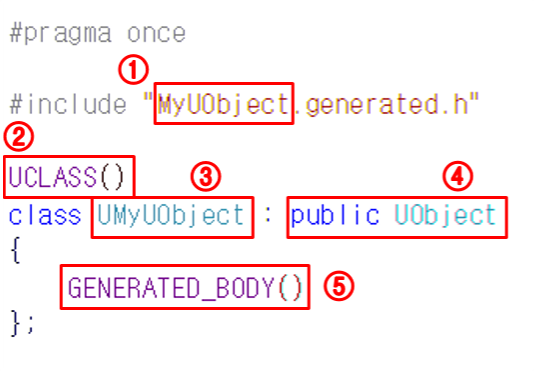
3. GC(Garbage Collection) : 참조되지 않는 객체를 메모리에서 자동 해제할 수 있습니다.

4. Serialization : 객체와 속성 정보를 통으로 안전하게 보관하고 로딩합니다.

5. Delegate : 함수를 묶어서 효과적으로 관리하고 호출할 수 있습니다.

6. Replication : 네트워크 상에서 객체간에 동기화를 시킬 수 있습니다.

7. Editor Integration : 언리얼 에디터 인터페이스를 통해 값을 편집할 수 있습니다.



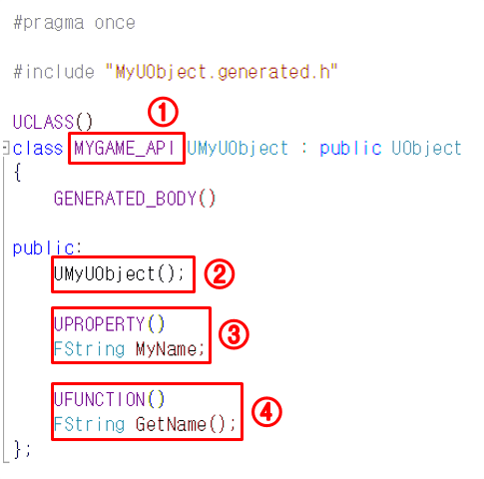
1. 클래스이름.generated.h를 반드시 가장 마지막에 include 시켜주어야 합니다.

2. 클래스 선언 전에 UCLASS매크로를 사용해야 합니다.

3. 언리얼 오브젝트의 접두사는 U, A 그리고 S가 있습니다. 액터 기반이라면 A를 액터 기반이 아니라면 모두 U를 써주되 UI를 담당하는 슬레이트만 S를 사용하면 됩니다.

4. UObject 클래스는 언리얼 오브젝트 최상단에 위치한 기본 클래스입니다. 이로 상속받은 클래스는 모두 언리얼 오브젝트가 됩니다.

5. 클래스 선언 내부에 GENERATED\_BODY() 매크로도 선언해줍니다.



1. 모듈이름\_API 매크로는 상황에 따라 부가 설정을 해주는 매크로입니다. 예를 들어 윈도우 에디터에서 사용할 수 있게 dll로 빌드할 때에는 이 매크로는 MS의 DLL문법인 \_\_declspec(dllexport)구문으로 자동 치환됩니다. 이렇게 선언해야 다른 모듈에서 현재 모듈 내 클래스에 접근할 수 있게 됩니다. 게임 빌드시에는 굳이 외부에 공개할 필요 없이 모든 모듈이 하나의 exe에 통합되므로 이 때는 빈 문자열로 치환됩니다.

2. 언리얼 오브젝트의 생성자는 오브젝트의 기본 값을 지정하는데 사용됩니다. 차후에 설명할 클래스 기본 객체인 CDO를 생성할 때 이 생성자 코드가 한번 실행됩니다.

3. 멤버 변수 위에 UPROPERTY 매크로를 얹어주면 이 변수는 앞으로 언리얼 엔진의 관리를 받게 됩니다. 향후 언리얼 오브젝트가 소멸되더라도 언리얼 엔진의 관리를 받아 해당 멤버 변수의 메모리도 같이 자동으로 소멸되며, 메모리 사용량도 체크할 수 있습니다.

4. 멤버 함수 위에 UFUNCTION 매크로를 얹어주면 블루프린트와 연동되게 할 수 있으며 딜리게이트나 리플리케이션과 같은 함수를 사용할 수 있어서 멤버 함수의 활용폭이 늘어납니다.

- UClass / CDO

언리얼 오브젝트의 정보를 담은 메타파일 : UClass

UClass는 언리얼 오브젝트에 대한 클래스 계층 구조 정보, 멤버변수, 함수에 대한 정보를 모두 기록하고 있다가 최종적으로 UClass 인스턴스를 만들고 이 인스턴스로 기본객체(CDO)를 같이 생성한다.

언리얼 오브젝트의 생성자는 인스턴스를 초기화해 CDO를 제작하기 위한 목적으로 사용한다. 이 생성자는 초기화에만 사용 게임플레이에서는 사용할 일 없다.