

Project TA(TargetActor)	
엔진 구조 개발	황정인
서버 클라이언트 통신 개발(RPC, IdlCompiler)	
파일 입출력, 파싱(Xml, Binary) 개발	
각종 컨텐츠 개발	
게임관련 데이터(GameData, PathPoint, SpawnData) // 기획	
<목차>	
1. 구조와 개념	
- 프로젝트 구조	
- App	
- ActorManager	
- ActorSystemManager	
- SpawnDataManager	
- GameDataManager	
- 기본 정보 설명	
- Common, Client, Server의 접두사의 의미	
- initialize, open, close 함수의 의미	
- 공유 자원과 SRW Lock	
- ScopedLock의 필요성	
- ECS(Entity Component System) 구조	
- Actor, ActorComponent, ActorSystem	
- ActorPool과 ActorComponentPool	
- onToPool, onFromPool, onActive 함수의 의미	
- ThreadLoadTaskManager	
- 게임관련 데이터의 병렬적 로드	
- ActorEventTimer	
- 이벤트 큐 관리, 액터 관련 이벤트를 처리	
2. 서버 클라이언트 통신	
- IOCP와 Overlapped I/O	

- IOCP
- Overlapped I/O
- IOCP와 Overlapped I/O의 조합

- RPC 통신
  - RPC통신의 필요성과 원리
  - IdlCompiler
  - 간단한 원격함수 호출 예시

### 3. 게임관련 데이터의 세이브와 로드

- 데이터 형식
  - Xml
  - Binary
  - **MemoryBuffer**와 **Serializer**
- **GameData**
  - Excel에서 에디팅
  - 로드 과정
- **UE4**에서 에디팅
  - 데이터들을 배치할 수 있는 레벨과 레벨태그
  - 에디터에서 Play 없이 클릭 한번으로 각 데이터를 추출
- **PathPoint**
  - PathPoint 데이터가 필요한 이유
  - UE4에서 에디팅
  - 로드 과정
  - 동작 원리
- **SpawnData**
  - UE4에서 에디팅
  - 로드 과정
  - 동작 원리
- **Recast/Detour Navigation**

- UE4의 RecastNavMesh
- UE4에서 리소스 로드
  - 리소스의 동적 비동기 로드
  - 동적 비동기 로드조건
  - 데이터 검증을 위한 방법

## 4. 컨텐츠

- TA게임엔진과 UE4의 통신
  - TAGameEvent
- AI
  - BehaviorTree
    - AiBTNode
    - AiBTNodeComposite
    - AiBTNodeCondition
    - AiBTNodeExecution
- Move
  - Sector이동과 통보
  - Sector이동 로직 의사코드
- Item
  - Item과 ItemSet
- UI
  - TACHunkUserWidget
- Interaction
  - InteractionActor
  - InteractionButton과 InteractionObject
  - Interaction과 Camera전환

## 5. 기타

- 실수를 줄이기 위한 방법
  - TA\_ASSERT\_DEV, TA\_LOG\_DEV, TA\_COMPILE\_DEV
- 빠른 개발을 위한 방법
  - 컴파일타임을 줄이기 위한 방법
    - Pimpl 패턴과 전방선언
    - 매크로 스위치문 활용
    - 템플릿 적극 활용
- 유용한 기능들
  - Enum <=> String 변환
  - ToStringCast와 FromStringCast

## 6. 학습자료

- 문자열 관련 정리
- UE4

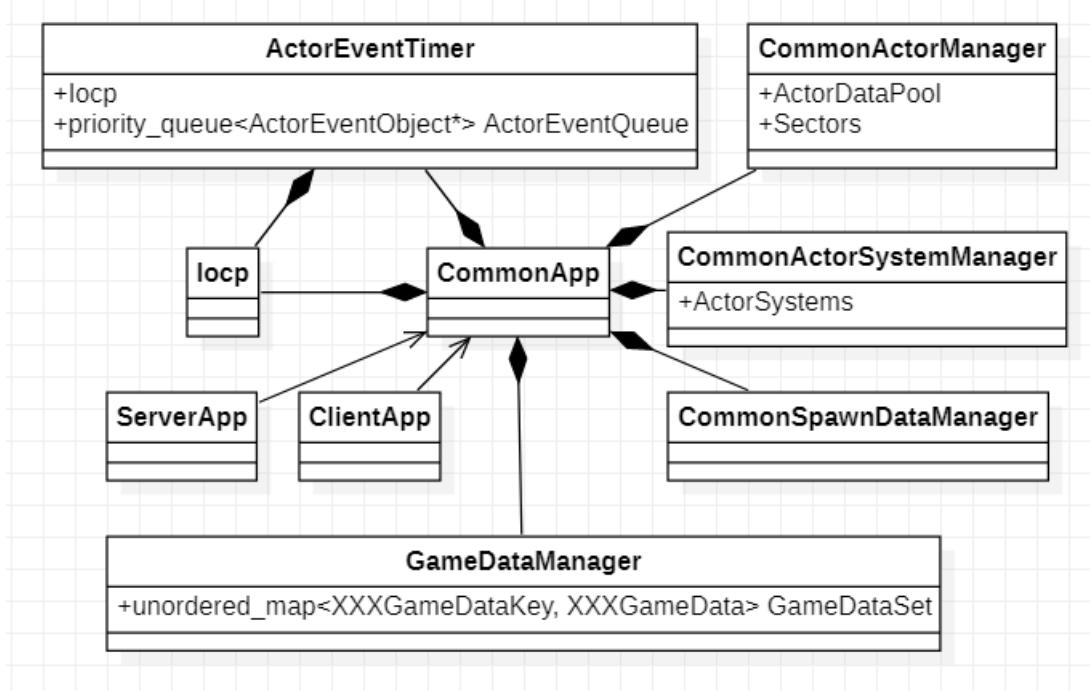
### 1. 구조와 개념->프로젝트 구조

#### - App

엔진에서 가장 최상위 클래스 역할을 한다. 여기서 모든 풀, 각종 데이터 등 할당, 초기화가 일어난다. 또 프로세스가 끝나면 모든 할당했던 메모리들을 해제시킨다.

간단히 각 매니저들에 대해서 소개하면 다음과 같다.

- ActorEventTimer : 각종 이벤트를 관리한다.
- CommonActorManager : 액터풀과 컴포넌트풀, 섹터들을 관리한다.
- CommonActorSystemManager : 액터시스템들을 관리한다.
- CommonSpawnDataManager : 스폰관련 정보를 관리한다.
- GameDataManager : 게임데이터 관련 정보를 관리한다.
- IOCP : Overlapped I/O와 함께 통신, 이벤트 처리에 사용된다.



## - ActorManager

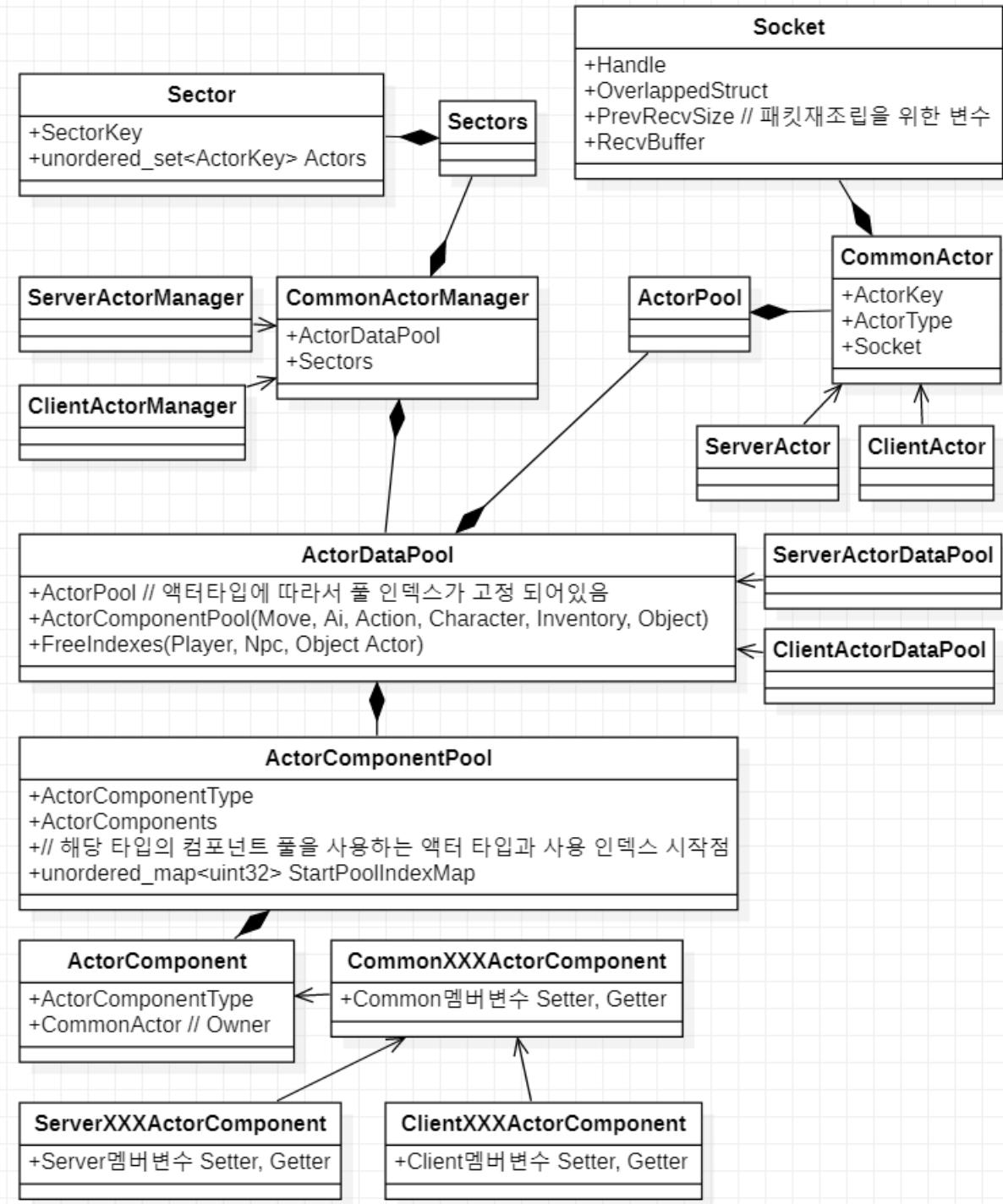
(ActorPool, ActorComponentPool 등 자세한 설명은 다음 ECS구조 설명할 때 같이 하겠다.)

액터와 액터컴포넌트, 섹터들을 관리하는 역할을 한다. ActorDataPool와, Sectors를 멤버로 가지고 있으며, ActorDataPool안에는 ActorPool과 ActorComponentPool로 나뉜다. 이 풀에게 요청에 의해서 풀에서 객체를 꺼내서 빌려쓰고, 다 쓰면 반납하는 식으로 사용한다.

ActorPool에 객체로 존재하는 Actor들은 기본적으로 ActorKey와 ActorType, 그리고 선택적으로 Socket을 가지고 있다. (Player액터인 경우)

ActorComponentPool의 경우 ActorComponent를 가지고 있는데, 이 ActorComponent는 기능이 없고, Get, Set만 가능한 가변적인 데이터라고 보면 된다.

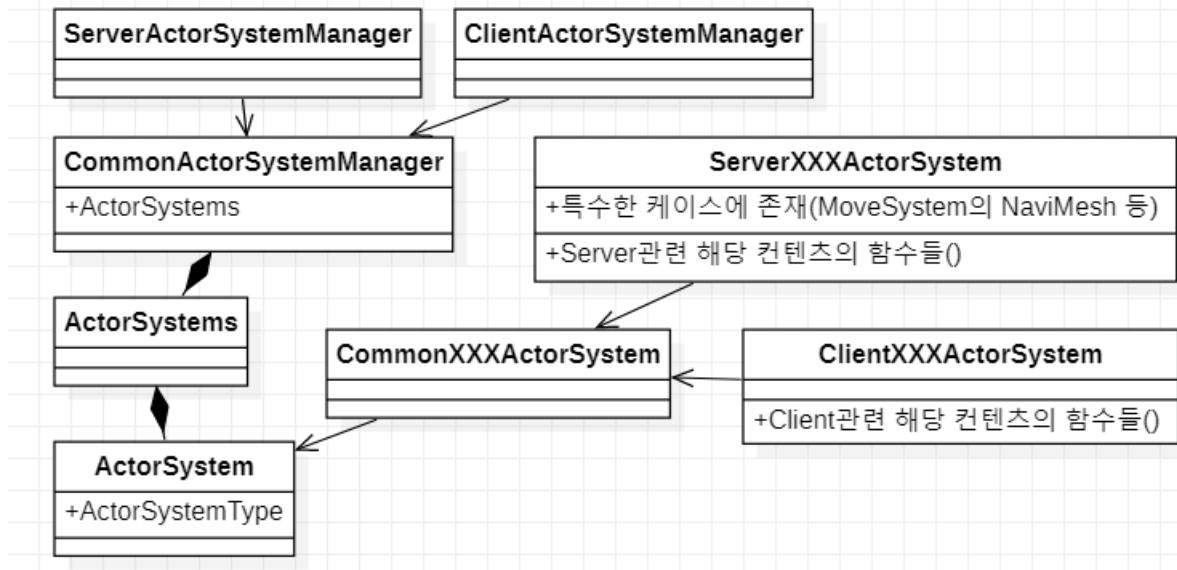
Sectors에는 액터가 이동할 수 있는 Sector들이 모여있다. 이 Sector를 통해서 플레이어는 해당 장소에 어떤 액터가 있는지 조회할 수 있고, 반대로 해당 장소의 액터에게 본인을 알릴 수도 있다.



## - ActorSystemManager

(ActorSystem의 자세한 설명은 다음 ECS구조 설명할 때 같이 하겠다.)

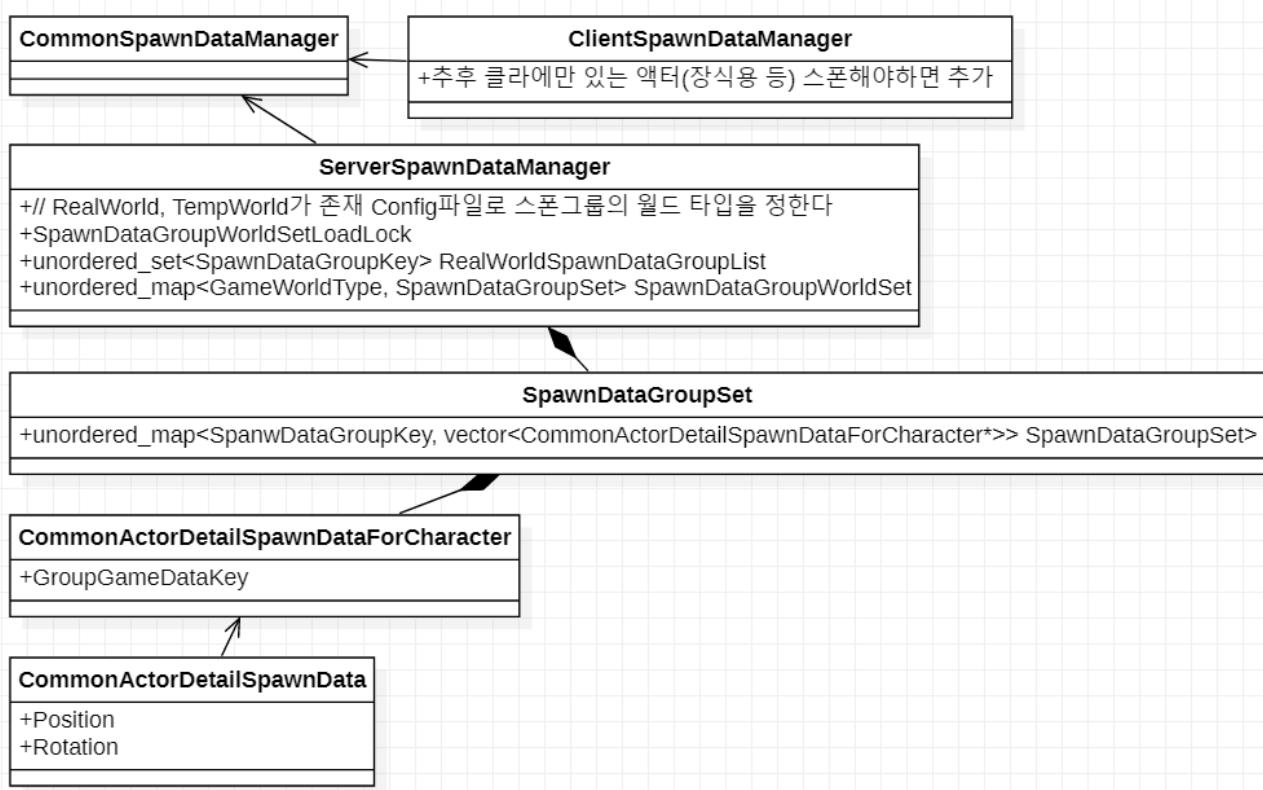
ActorSystems를 멤버로 가지고 있다. ActorSystems들은 간단하게 액터의 각종 기능을 담당한다.



## - SpawnDataManager

(SpawnData등 자세한 설명은 게임관련 데이터 설명할 때 같이 하겠다.)

SpawnData들을 파일에서 로드해서 가지고 있으며, 스폰관련 작업을 관리하는 역할을 한다.

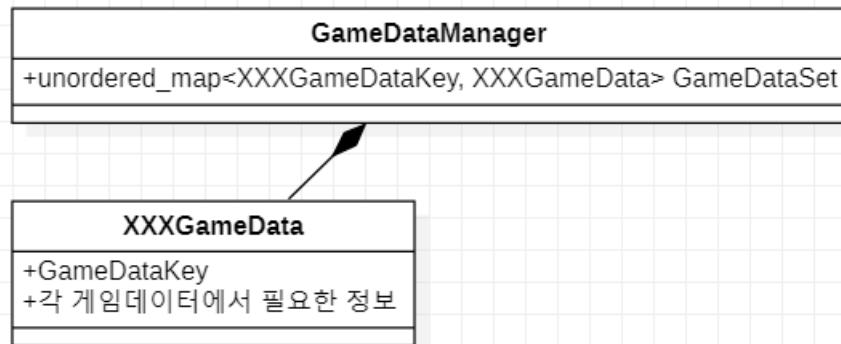


## - GameDataManager

(GameData등 자세한 설명은 게임관련 데이터 설명할 때 같이 하겠다.)

GameData들을 파일에서 로드해서 멤버로 가지고 있으며, 각 컨텐츠에서 GameData가 필요할 때

제공해준다.



## 1. 구조와 개념->기본 정보 설명

### - Common, Client, Server의 접두사의 의미

서버 클라이언트 구조를 만들기 위해서 공통적인 코드와 각각 사용해야 할 코드를 분리할 필요가 있다.

그래서 일단 공통적인 코드는 Common접두사를 붙인다.

그리고 클라이언트에서만 사용하는 코드는 Client접두사, 서버에서만 사용하는 코드는 Server접두사를 붙인 후, Common클래스를 상속받아서 사용하는 것으로 설계했다. 이런 식의 상속구조로 중복 코드를 최대한 없애고, 유지보수성을 높이며, 코드 가독성을 높이는 효과를 볼 수 있었다.

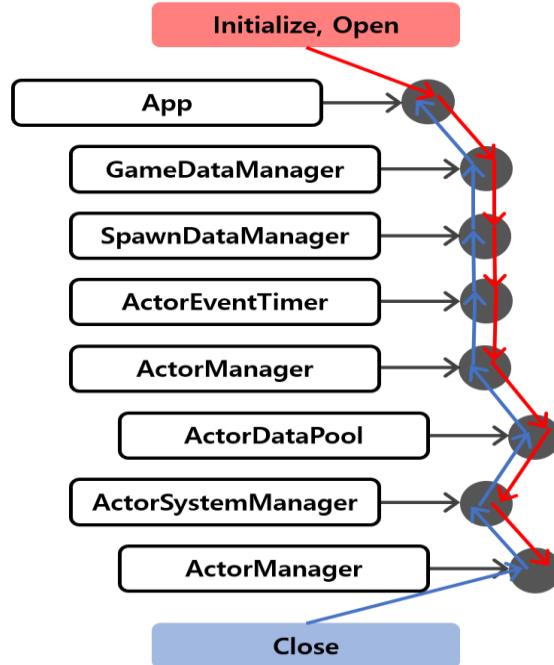
### - initialize, open, close 함수의 의미

여러 기능을 수행하고, 시스템을 가진 상위 클래스들은 크게 3가지의 함수를 공통적으로 갖도록 설계하였다. 일단 초기화단계에서는 Initialize와 open으로 2단계로 나누었고, 해제단계에서는 close를 사용하도록 하였다.

먼저 Initialize는 기본적으로 메모리를 할당하는 단계이고, open은 할당된 메모리를 가지고, 필요에 의해서 초기화하는 단계이다. 처음에는 그냥 할당, 초기화를 한번에 하면 되지 않을까라는 생각을 했는데, 프로젝트 구조가 거의 Common과 이를 상속받은 Server, Client구조이기 때문에 문제가 생길 수 있다.

예를 들어보면, ActorDataPool에서 Common레벨에서 처리해야 할 초기화는 액터풀의 FreeIndexes들을 넣어주는 작업이다. 하지만 할당 자체는 ServerActorDataPool이나 ClientActorDataPool에서 해줘야 한다. (각각 Server, Client에 맞는 ServerActor, ClientActor를 할당해야 하기 때문) 그렇게 되면, Common에서의 초기화 작업이 메모리가 할당되기전에 수행될 것이기 때문에 문제가 생긴다.

close는 프로세스가 끝날 때, 할당했던 메모리들을 해제하는 작업을 수행한다. 순서는 작은클래스에서 큰클래스으로 수행하고(예를 들어 ActorManager보다는 ActorPool의 close가 먼저 불린다.), 다른 곳에서 참조되는 클래스일 경우, 참조하고 있는 클래스에서 close가 호출된 이후에 수행하도록 작성한다.



### - 싱글프로세스, 멀티스레드 서버와 클라이언트

로딩 없는 오픈 월드 형식의 MMORPG 게임을 만들 예정이라서 싱글 프로세스, 멀티스레드 구조로 생각하고 만들었다. 당연히, 채널이동과 월드이동 로딩이 존재하지 않는다.

### - 공유 자원과 SRW Lock

멀티스레드 프로그래밍을 하면서 가장 신경 써야하는 부분은 역시 공유자원과 여러 관련된 이슈였다. 찾아보다가 유저모드 기반 동기화에서 SRW Lock을 공부하게 되었다. 이는 성능이 뛰어나고, 두 가지 방식(Shared, Exclusive)으로 공유자원에 대해서 락을 걸어줄 수가 있어서 편리하다. 특히 공유자원이 많이 변경되지는 않지만, 조회가 많이 이루어지는 곳에서는 매우 유용하다.  
예를 하나 들어보면 섹터에서는 액터가 많이 움직이지 않으면, 섹터안에 있는 액터의 리스트가 변하지 않지만, 안에 있는 액터가 움직일 때 해당 섹터에 있는 액터리스트를 모두 조회하면서 내가 움직이고 있다는 것을 알려야한다. 액터리스트 조회 시 Shared를 사용하면, 같이 Shared모드로 사용한 모든 곳에서 NonBlocking으로 처리할 수 있다.

엔진에서는 이를 쉽게 사용하기 위해서 Lockable이라는 상속받을 수 있는 클래스와 락객체를 사용할 수 있게 LockableObject를 만들어서 사용하고 있다.

### - ScopedLock의 필요성

개발하다가 느낀점인데, 공유자원에 접근할 때마다 Acquire, Release를 쌍으로 제대로 호출해줘야 하고, 만약 Release가 누락된다면, 데드락상태에 빠지게 된다. 이런 일이 없을 것 같지만, 의외로 코드가 길어지다보면 은근히 발생하는 상황이다. 이런 것들을 방지하기 위해서 스코프안에서만 락을 걸고, 스코프를 빠져나갈 때 락을 해제하는 방식으로 ScopedLock 클래스를 하나 만들었다. 이 클래스는 생성자에서 Lockable객체를 받으며, 거기에서 Shared모드나 Exclusive모드로 Acquire해준다. 그리고 소멸자에서는 Release 시켜준다.

## 1. 구조와 개념->ECS(Entity Component System) 구조

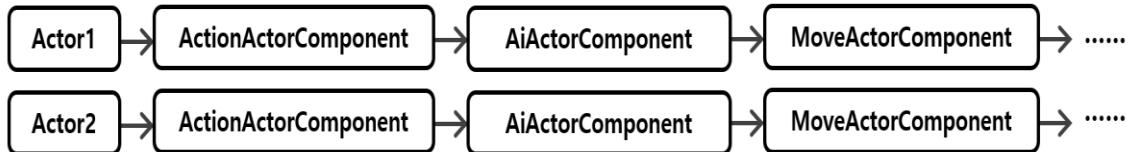
### - Actor, ActorComponent, ActorSystem

일반적으로 많이 사용하는 Actor가 ActorComponent의 실제 객체를 가지고 있고, 필요에 따라서 Actor가 ActorComponent에서 기능을 요청한다.

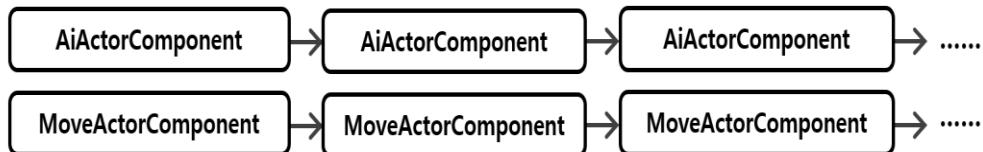
엔진에서는 일반적인 패턴이 아닌 ECS 패턴을 채택해서 사용하고 있다. Actor(Entity)는 현재 어떤 컴포넌트를 가지고 있는지만 알고 있고(엔진에서 ActorType에 따라 ActorComponentTypeList를 가지고 있다.), ActorComponent(Component)는 Getter, Setter와 데이터만 가지고 있으며, 실질적인 기능은 ActorSystem(System)에서 처리한다. 이런 식의 구조를 선택한 이유를 나열해 보면 다음과 같다.

1. 일단 구조 자체가 단순해지고, 가독성이 좋아진다.
2. 기존 상속구조(캐릭터는 어떤 컴포넌트를 들고 있고, 플레이어는 그를 상속받고 추가로 어떤 컴포넌트가 필요하고 ... 등등)에서 벗어나서 자유롭게 Entity를 정의할 수 있다. 간단히 어떤 타입의 컴포넌트를 가지고 있다고 정의만 하면 된다. 즉 확장성이 매우 좋아진다.
3. 시스템 단위(보통 하나의 시스템은 1개 많으면 2-3개의 컴포넌트를 담당한다고 보면 된다.)로 스레드를 나눠서 처리할 수 있으며, ActorComponentPool(ActorComponet)를 풀로 관리할 수 있는 이유도 순수하게 데이터만 존재하는 모두다 같은 객체이기 때문)에 있는 컴포넌트들을 모아서 처리하기 때문에 데이터 지역성이 매우 좋아진다.

### - AKKA Component Update



### - ECS Component Update

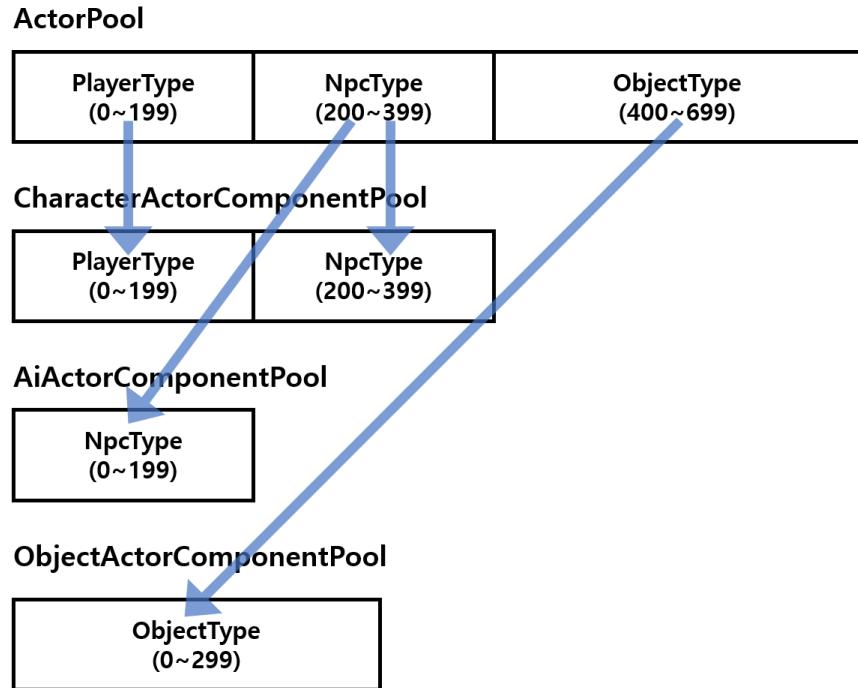


### - ActorPool과 ActorComponentPool

먼저 ActorPool 부터보면 ActorType에 따라서 사용할 수 있는 풀의 범위가 다르다. 예를들어 액터타입은 (Player, Npc, Object) 3가지가 있는데, Player는 0~199번, Npc는 200번~399번, Object는 400~599번 이런식으로 정해져있다. 이렇게 한 이유는 특정 ActorType에만 수행해야 할 것들이 있는 경우(플레이어 한테만 특정 이벤트를 보낸다든지 등) 해당 풀의 범위에서만 루프를 돌면서 처리하면 된다.

ActorComponent타입도 ActorPool처럼 액터 타입에 따라서 사용할 수 있는 풀의 범위가 다르지만, ActorPool과 다르게 특정 ActorType에서는 사용하지 않는 컴포넌트가 있기 때문에 StartIndexMap(처음 초기화 단계에서 생성되고, 불변하므로 Thread Safe하다.)이라는 멤버가 존재한다. 이 멤버를 통해서 어떤 ActorType이 해당 컴포넌트를 사용하고 있는지 알 수 있으며, 어느 범위의 ActorComponentPool을

사용하는지 알 수 있다.

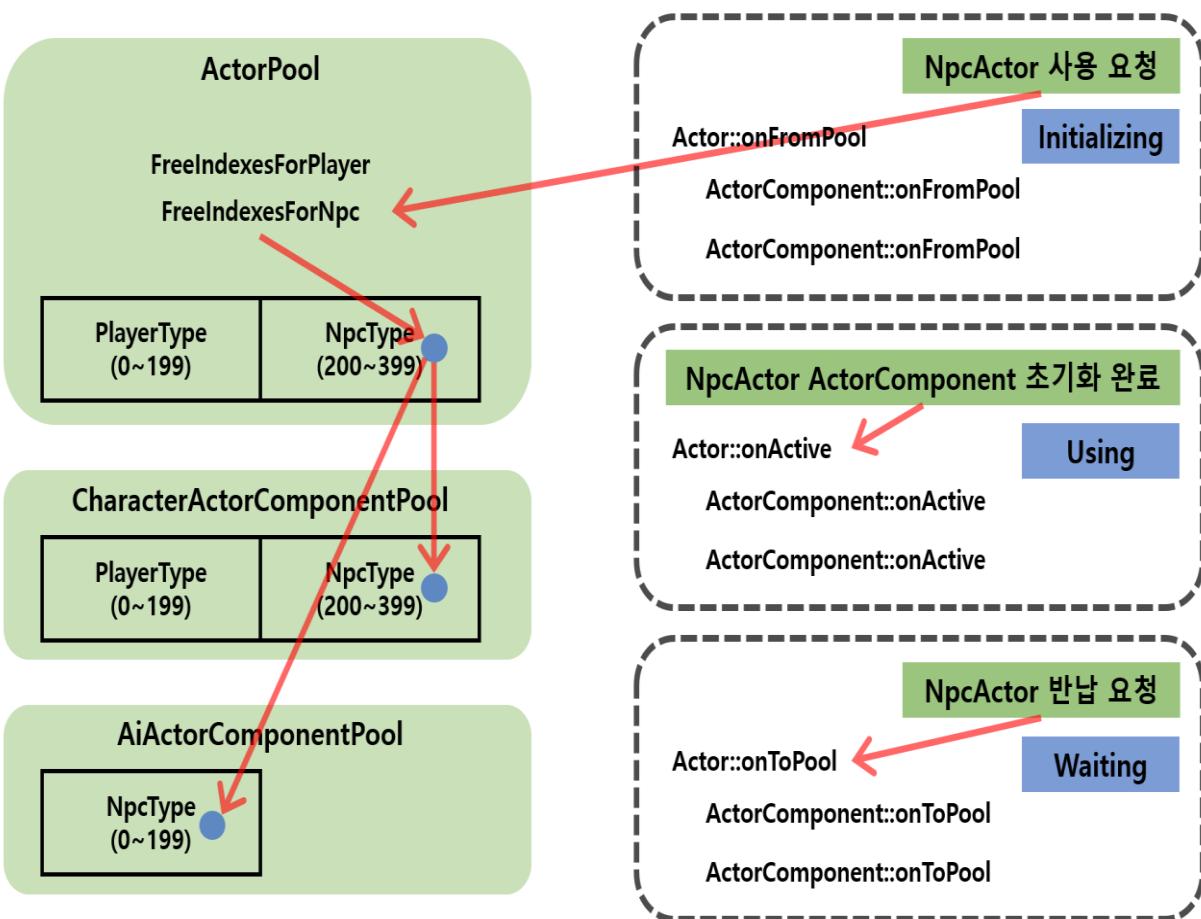


#### - **onToPool, onFromPool, onActive** 함수의 의미

풀에서 객체를 꺼내 쓰기 위해서는 몇 단계의 과정이 필요한데, 일단, 비어 있는 인덱스를 찾고(FreeIndexes에 저장되어 있음), 해당 풀에서 Actor와 ActorComponent를 사용하겠다고 마킹하는 것부터 시작한다. (PoolValueState::Initializing으로 초기화) 그 다음 onFromPool함수가 각 ActorComponent에서 호출된다.

아직 초기화 되지 않는 단계이므로 initializeActorComponent함수를 통해서 Actor의 ActorComponent들의 데이터들을 모두 초기화 하고 나면, onActive함수가 ActorComponent에서 호출된다. 그리고 최종적으로 Actor와 ActorComponent에 현재 사용하고 있다고 마킹해준다. (PoolValueState::Using으로 초기화) 이렇게 함수를 2개로 나눈 이유는 Actor의 ActorComponent가 초기화단계에서 필요한 작업과 초기화가 끝난 상태이면서 사용 전에 필요한 작업이 따로 있기 때문이다.

onToPool함수의 경우에는 풀에 반납될 때 호출되고, 다시 꺼내 사용할 수 있도록 FreeIndexes에 해당 Actor가 추가되고, 사용할 수 있게 마킹해준다.(PoolValueState::Waiting으로 초기화)

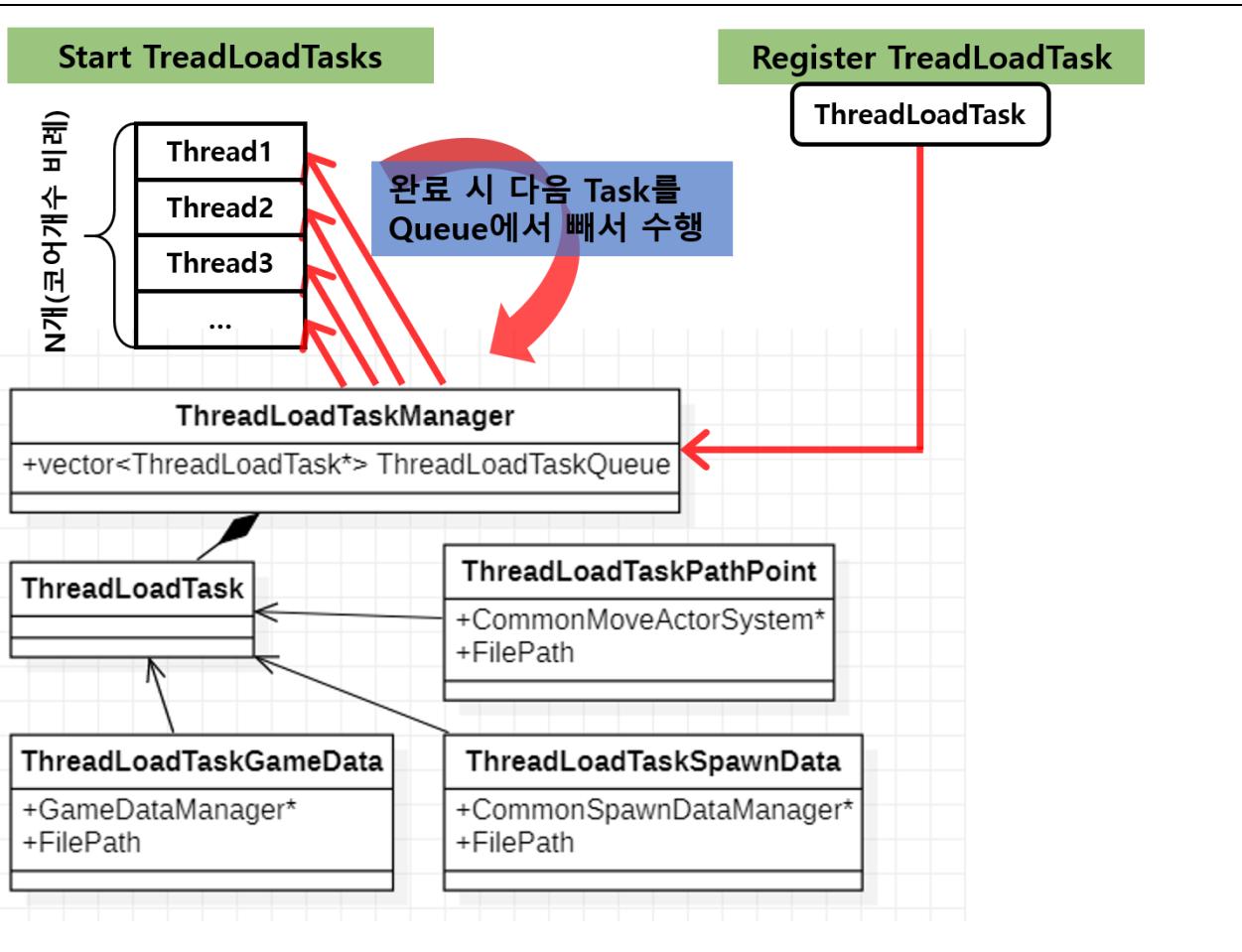


## 1. 구조와 개념->ThreadLoadTaskManager

### - 게임관련 데이터의 병렬적 로드

게임과 관련된 로드해야 할 데이터는 다양하고 개수도 많다. 그래서 효율적인 병렬적인 로드를 수행해줘야 한다. 효율적인 병렬적인 로드를 하기 위해서는 병렬처리 할 때 공유자원 접근을 최소화해야 한다. (아예 독립적이면 더 좋다.)

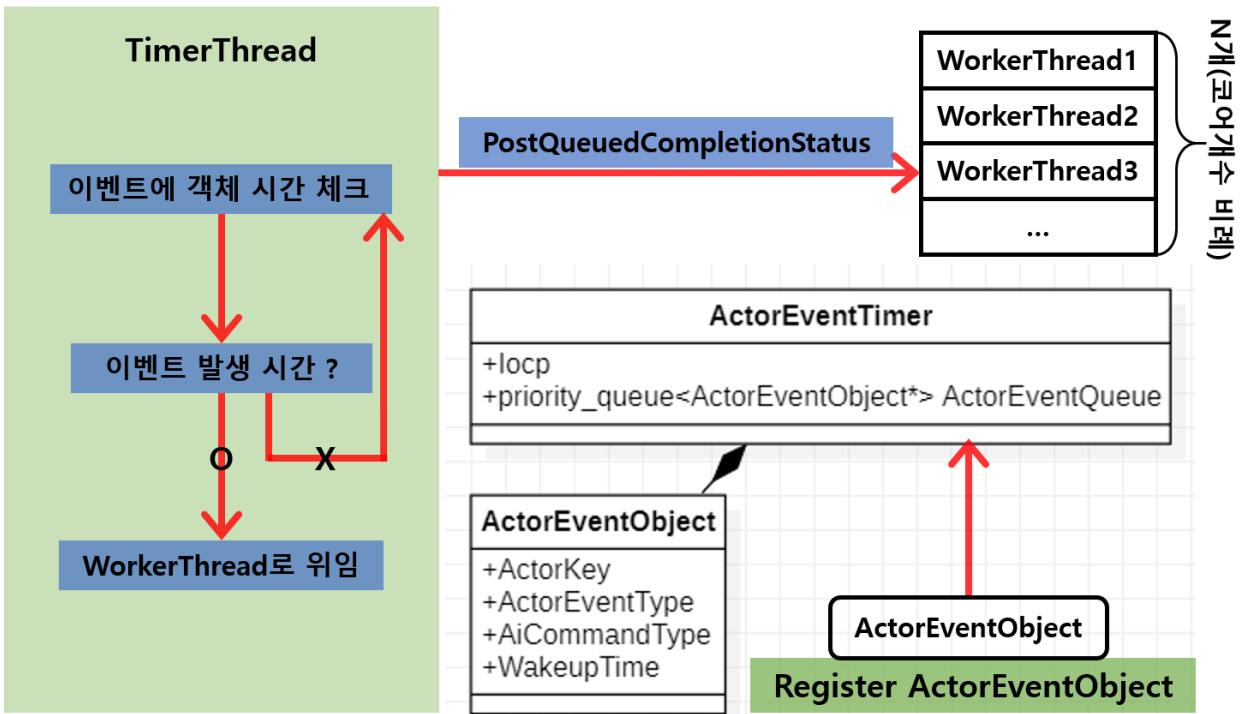
이런 병렬적 로드를 수행하기 위해서 ThreadLoadTaskManager 클래스를 만들었다. 파일이 수십 수백개가 있다고 가정해보면, 그 파일 개수만큼 스레드를 생성해서 한꺼번에 처리한다면, 코어수가 아무리 많은 CPU라도 스레드 수를 감당하지 못할 것이다. 그리고 그 많은 스레드들은 컨택스트 스위칭을 많이 하면서 성능을 악화시킨다. 이런 것을 방지하기 위해서 ThreadLoadTaskManager에서는 ThreadLoadTaskObject를 등록할 수 있도록 만들었고, 모든 ThreadLoadTaskObject의 등록이 끝나고, Start를 시켜주면 적절한 스레드를 생성해서 해당 Task를 처리하도록 설계하였다. ThreadLoadTaskObject에서는 함수를 호출할 객체, 함수호출 시 사용되는 입력 파라미터를 멤버로 갖고 있다.



## 1. 구조와 개념->ActorEventTimer

### - 이벤트 큐 관리, 액터 관련 이벤트를 처리

게임 진행중에는 게임 이벤트를 발생시킬 필요가 있다. 예를 들어서 몬스터가 죽었으면, 일정시간 뒤에 SpawnDataManager가 다시 스폰하도록 해야 한다. ActorEventTimer는 이런 게임이벤트들을 관리하는 클래스다. 스레드 하나(타이머 스레드라고 하겠다.)를 할당해서 일정시간마다 이벤트 큐(시간이 조금 남은 순서대로 정렬되어 큐에 들어가 있다. priority\_queue)에 지금 수행되어야 하는 이벤트가 있는지 계속해서 체크한다. 만약에 등록된 이벤트 중에 수행되어야 할 이벤트가 있다면 해당 이벤트를 직접 수행하지는 않고, 나머지 워커스레드들에게 PostQueuedCompletionStatus를 사용해서 위임한다. 직접 수행하게 되면, 타이머 스레드에서 병목현상이 발생할 수 있다.



## 2. 서버 클라이언트 통신->IOCP와 Overlapped I/O

### - IOCP

디바이스의 입출력 완료를 통보하기 위한 포트이다. 빠른 입출력 통보, 최적화된 쓰레드풀링 기술을 포함하고 있다. 기존에 디바이스의 개수에 한계가 있던 방식들(Event Select 등)과 다르게 디바이스(여기서는 Socket)와 IOCP를 연결하는 데 제한이 없다. 그리고 완료를 '통보' 해주기 때문에 계속해서 완료가 되었나 확인해볼 필요도 없다. IOCP에 연결된 디바이스의 입출력 작업이 완료가 되면, GetQueuedCompletionStatus를 통해 대기하고 있는 스레드에 알아서 통보해준다. 해당 스레드는 완료된 정보를 가지고 나머지를 처리하면 된다.

### - Overlapped I/O

간단한 동작원리는 소켓에 대해서 Overlapped 액세스를 걸어주고, Overlapped 액세스가 성공했는지 확인한 후 성공했으면 결과값을 얻어와서 나머지를 처리하는 식으로 동작한다.

Overlapped를 걸 때, 진행 중인 상태 현황을 보관하는 구조체와 수신된 데이터를 받을 데이터 블록이 필요한데, 이때 주의할 점은 Overlapped를 걸어 두었을 때 두 객체를 운영체제가 백그라운드에서 접근 중이므로 Overlapped I/O 전용 함수가 비동기로 하는 일이 완료될 때까지 두 객체를 건드리면 안된다.

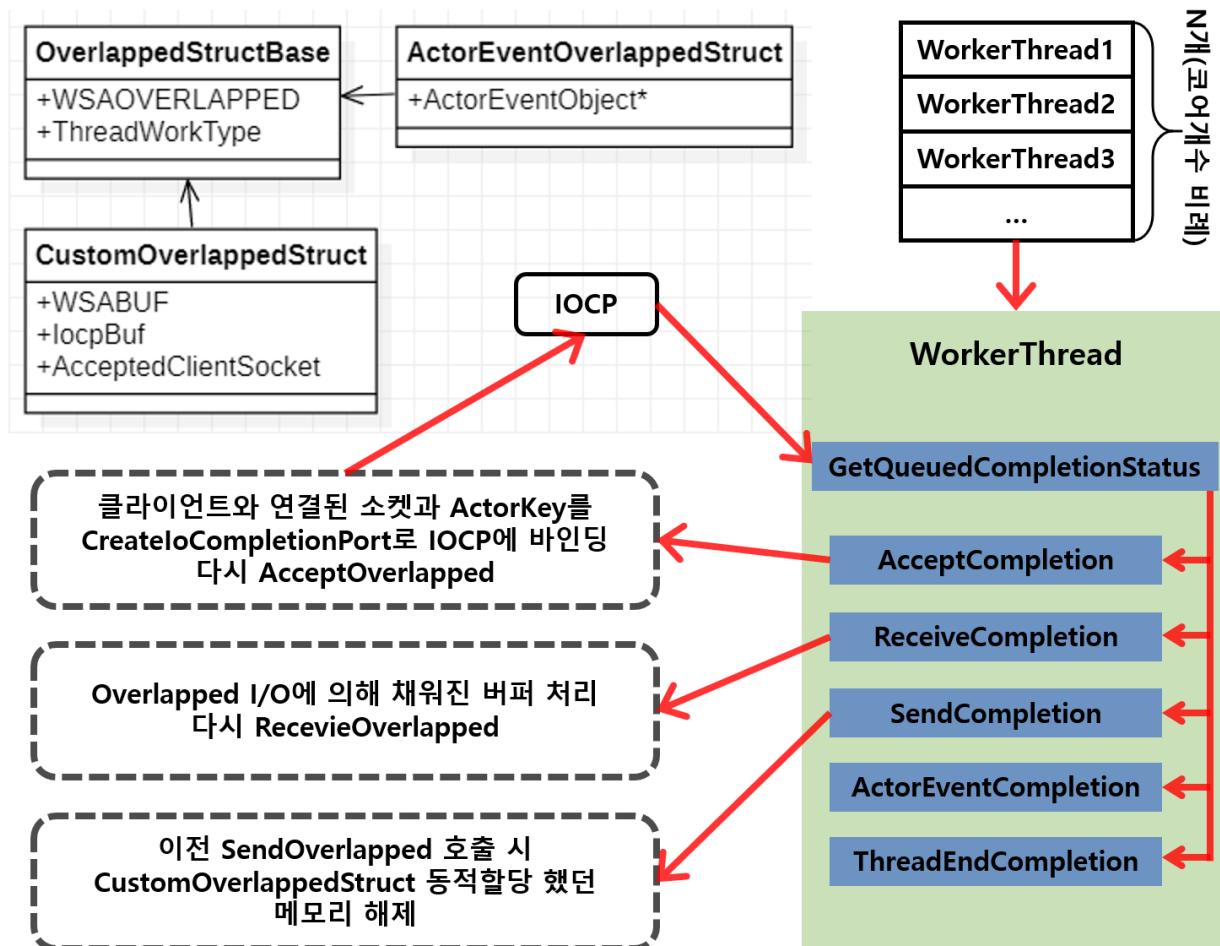
논블록킹 소켓이랑 비교했을 때, 코드가 깔끔해지고, 불필요한 시도를 계속 하지 않는다. (계속해서 send, receive를 호출하지 않고, 결과만 확인한다.) 근데 최종적으로 계속해서 send, receive를 호출하지 않을 뿐이지, GetOverlappedResult를 계속 호출해가면서 결과를 확인하기 때문에 한계가 있다.

### - IOCP와 Overlapped I/O의 조합

IOCP와 Overlapped I/O를 조합하면 성능이 매우 좋은 서버를 개발할 수 있다. 두 기능을 이용한 간단한 서버 클라이언트를 설명해보면, 일단 서버에서 리슨소켓을 IOCP에 추가하고, AcceptOverlapped를 걸어준다. 그리고 모든 WorkerThread에서 GetQueuedCompletionStatus로 대기해준다.

클라이언트가 접속하면 대기하고 있던 WorkerThread 중 하나가 깨어나서 Accept를 완료하고, 클라이언트와 연결된 소켓을 IOCP에 추가(CompletionKey는 해당 클라이언트를 구분할 수 있는 ActorKey로 설정하였다.)하고 ReceiveOverlapped를 걸어준다. 이제 클라이언트가 패킷을 보내면 다시 GetQueuedCompletionStatus를 통해서 대기하고 있던 WorkerThread 중 하나가 깨어나서 Overlapped I/O를 통해 채워진 버퍼를 읽어서 처리하면 된다.

SendOverlapped의 경우에는 한 소켓에 대해서 여러 개의 패킷을 동시에 보낼 수 있기 때문에, 동적 할당으로 Overlapped에 필요한 객체들을 할당하고, GetQueuedCompletionStatus을 통해서 완료가 통보되면 할당한 객체들을 삭제한다.



## 2. 서버 클라이언트 통신 ->RPC 통신

### - RPC통신의 필요성과 원리

일반적으로 서버 클라이언트 통신을 하기 위해서는 통신 내용에 따라 패킷객체를 정의해줘야 하고, 한쪽에서 그 패킷객체를 싸서 보내면 다른 쪽에서 패킷객체를 풀어서 내용에 따른 처리를 해줘야 한다.

근데 이렇게 개발을 패킷객체를 계속해서 수동으로 만들어야 하고, Pack, Unpack코드를 계속해서 작성해야 한다.

간단하게 원격함수호출(RPC, Remote Procedure Call)통신의 구현으로 해결할 수 있다.

번거로움도 줄이고, 코드가 간결해지고 상세코드는 IdlCompiler에 의해서 알아서 생성되기 때문에 직접

구현하지 않아서 실수를 줄일 수 있다.

동작방식은 매우 간단한데, 그냥 한쪽에서 다른 쪽의 함수를 원격으로 호출할 수 있게 만들어 준 것이다. 통신을 하기위해서 일반 함수를 호출하듯, 원격함수를 호출해주면 끝이다.

원래라면 한 프로세스에서 다른 원격의 프로세스의 함수를 호출할 수 없다. 하지만 IdlCompiler가 그렇게 보이도록 코드를 생성해주는 것이다. 상세 코드는 일반적인 통신방법과 똑같다.

## - IdlCompiler

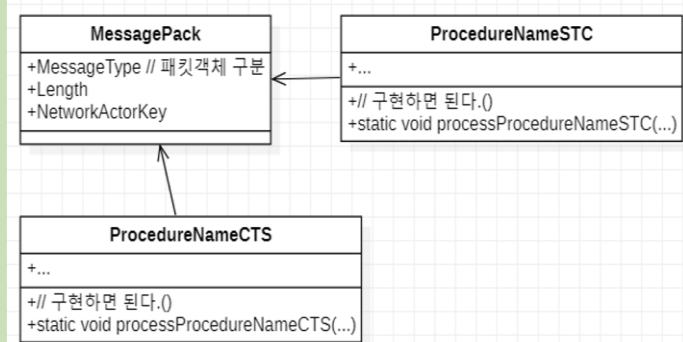
RPC통신에서는 호출 규약을 정의하는 IDL(Interface Definition Language)파일이 중요하다. 함수명, 인자 등이 포함되어있으며, 엔진에서는 IdlCompiler에 의해서 컴파일 되고, 파일들을(Stub코드) 생성한다.

```
[CTS]
{
    Login(const StringPack& id, const StringPack& password);
    MoveActor(const ActorKey& targetActorKey, const Vector& position);
    AttackTarget(const ActorKey& myActorKey, const ActorKey& targetActorKey);
    DetectTarget(const ActorKey& myActorKey, const uint32& targetActorType);
    MoveToTarget(const ActorKey& myActorKey, const ActorKey& targetActorKey);
    MoveToRandomPoint(const ActorKey& myActorKey);
    DropItem(const ItemSlotNo& slotNo, const int32& stackCount);
    UseItem(const ItemSlotNo& slotNo, const int32& stackCount);
}

[STC]
{
    CreateMainActor(const ActorKey& actorKeyToCreate);
    CreateActor(const uint32& actorType, const ActorKey& actorKeyToCreate);
    DestroyActor(const ActorKey& actorToDestroy);
    MoveComponentData(const ActorKey& targetActorKey, const CommonMoveComponentData& componentData);
    ActionComponentData(const ActorKey& targetActorKey, const CommonActionComponentData& componentData);
    AiComponentData(const ActorKey& targetActorKey, const CommonAiComponentData& componentData);
    CharacterComponentData(const ActorKey& targetActorKey, const CommonCharacterComponentData& componentData);
    InventoryComponentData(const ActorKey& targetActorKey, const CommonInventoryComponentData& componentData);
    ObjectComponentData(const ActorKey& targetActorKey, const CommonObjectComponentData& componentData);
    MoveActor(const ActorKey& targetActorKey, const Vector& position);
    ChangeHp(const ActorKey& attacker, const ActorKey& targetActorKey, const float& hpValue);
    KillActor(const ActorKey& targetActorKey);
    SetTargetActor(const ActorKey& myActorKey, const ActorKey& targetActorKey);
    DropItem(const ItemSlotNo& slotNo, const int32& stackCount);
    UseItem(const ItemSlotNo& slotNo, const int32& stackCount);
}
```

생성되는 것들은 인자들을 가지고 있는 패킷 객체, 원격함수(기능 구현은 직접 해야한다.), 원격함수를 호출하는 함수, MessageType에 따라 적절히 함수를 호출해주는 코드가 생성된다. 여기서 원격함수만 구현해서 사용하면 된다.

## 패킷객체



## MessageType에 따라 적절한 함수 호출 코드

```

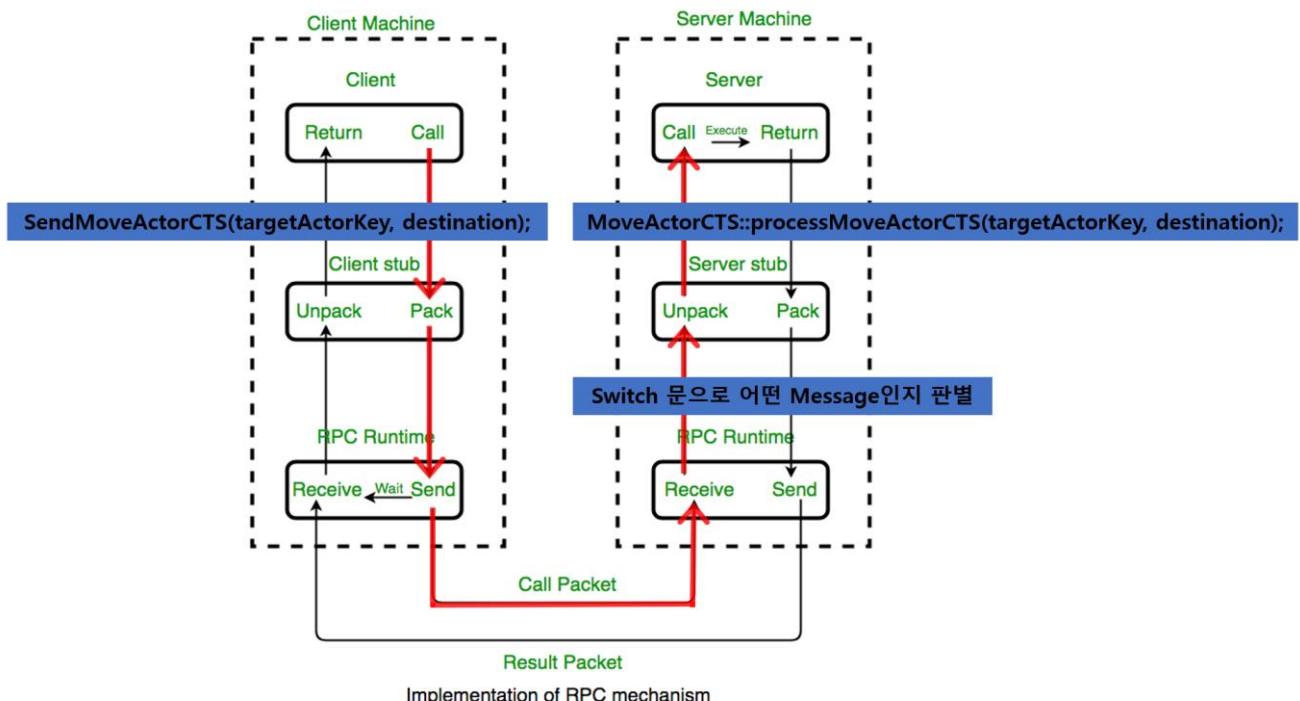
#define MESSAGE_SWITCH_CASE(TypeName)#
case MessageType::Type_Name:##
{#
    TypeName::processMessage(static_cast<const TypeName*>(message));#
}#
break;#
MESSAGE_SWITCH_CASE(CreateMainActorSTC)
MESSAGE_SWITCH_CASE(CreateActorSTC)
MESSAGE_SWITCH_CASE(DestroyActorSTC)
MESSAGE_SWITCH_CASE(MoveComponentDataSTC)
MESSAGE_SWITCH_CASE(ActionComponentDataSTC)
MESSAGE_SWITCH_CASE(AiComponentDataSTC)
MESSAGE_SWITCH_CASE(CharacterComponentDataSTC)
MESSAGE_SWITCH_CASE(InventoryComponentDataSTC)
MESSAGE_SWITCH_CASE(ObjectComponentDataSTC)
MESSAGE_SWITCH_CASE(MoveActorSTC)
MESSAGE_SWITCH_CASE(ChangeHpSTC)
MESSAGE_SWITCH_CASE(KillActorSTC)
MESSAGE_SWITCH_CASE(SetTargetActorSTC)
MESSAGE_SWITCH_CASE(DropItemSTC)
MESSAGE_SWITCH_CASE(UseItemSTC)
MESSAGE_SWITCH_CASE(LoginCTS)
MESSAGE_SWITCH_CASE(MoveActorCTS)
MESSAGE_SWITCH_CASE(AttackTargetCTS)
MESSAGE_SWITCH_CASE(DetectTargetCTS)
MESSAGE_SWITCH_CASE(MoveToTargetCTS)
MESSAGE_SWITCH_CASE(MoveToRandomPointCTS)
MESSAGE_SWITCH_CASE(DropItemCTS)
MESSAGE_SWITCH_CASE(UseItemCTS)
#endif MESSAGE_SWITCH_CASE
  
```

## 원격함수 호출 코드

<b>STC</b> <AllPacketServer.h> void SendProdecureNameSTC(...) ...	<b>CTS</b> <AllPacketClient.h> void SendProdecureNameCTS(...) ...
----------------------------------------------------------------------------	----------------------------------------------------------------------------

## - 간단한 원격함수 호출 예시

간단히 클라이언트에서 서버의 MoveActor라는 원격함수를 호출한다고 가정한다.



\* 백그라운드 이미지 출처 : greeks for greeks

## 3. 게임관련 데이터의 세이브와 로드->데이터 형식

## - Xml

게임관련 데이터의 대부분은 Xml파일로 되어있다. GameData, PathPoint, SpawnData 등이 그렇다. Xml로 만든 이유는 일단 Xml파일만 봤을 때 직관적으로 어떤 데이터인지 쉽게 눈에 들어오기 때문이다. Xml파일을 불러오는 과정은 몇 단계로 나뉜다.

1. 파일을 읽어서 파일 스트링을 가지고 온다.
2. '<', '>' 의 구분자들로 파일 스트링을 나눈다.
3. 나누어진 파일 스트링들 속에서 루트노드를 찾는다.
4. RootElement를 찾았으면 XmlElementStack을 만들고 가장 최상위에 RootElement를 넣는다.
5. 나누어진 파일스트링을 돌면서 ChildElement, Attribute를 처리한다.

5.1. 현재 스트링이 '<'이면 다음과 같이 처리한다.

5.1.1. '<' 다음에 '/'가 오면 XmlElement가 끝나는 것이다. XmlElementStack에서 빼준다.

```
// </XmlElementName>
```

5.1.2. '<' 다음에 '/'가 아니면 Attribute들을 파싱해준다.

5.1.1.1. '=', '\"', '\''의 구분자들로 해당 Attribute스트링을 나눈다.

5.1.1.2. '=' 구분자 기준으로 AttributeName과AttributeValue를 뽑아내서 저장한다.

5.1.1.3. 현재 XmlElementStack 최상단에 있는 Element에 해당 Element를 자식으로 추가한다.

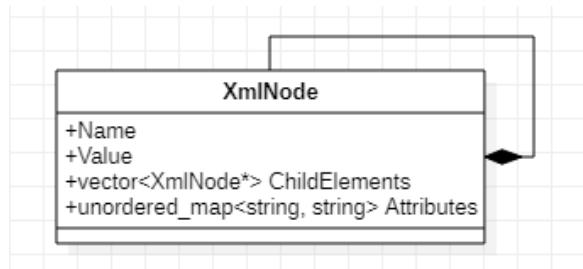
5.1.1.4. 해당 Element를 XmlElementStack 최상단에 넣는다.

5.2. 현재 스트링이 '>'이면 다음과 같이 처리한다.

5.2.1. 이전 스트링의 마지막문자가 '/'이면, XmlElement가 끝나는 것이다. XmlElementStack에서 빼준다.

```
// <XmlElementName Attribute1="Value1".... />
```

위의 과정을 거치고 나면 다음과 같은 RootElement를 얻을 수 있다. 이제 이를 가지고 각 데이터에 맞게 파싱을 진행하면 된다.



## - Binary

디버깅하기는 힘들지만, 가볍고 빠르게 바이너리 데이터를 로드, 세이브 할 수 있는 형식도 지원하기 위해서 엔진에서 해당 기능을 지원한다. Read, Write를 하기 위한 MemoryBuffer와 Serializer가 존재한다.

## - MemoryBuffer와 Serializer

MemoryBuffer는 말그대로 데이터를 담는 버퍼다. 최대크기가 존재하며, 현재 크기를 가지고 있다. 만약에 현재 크기가 최대크기를 넘어가게 되면 자동으로 버퍼를 키워준다. Owner로 Serializer를 붙여서 사용할 수도 있으며, MemoryBuffer 그 자체로도 사용 가능하다. 파일을 바이너리로 로드하거나 저장할 수 있고, 스트링으로 로드하거나 저장할 수 있는 기능을 담고 있다.

Serializer는 MemoryBuffer에 직렬화, 역직렬화를 편하게 하기 위해서 만들어졌다. SerializerMode::Read,

Write, WriteLog 등 몇가지 모드를 가지고 있으며, 비트플래그로 관리된다. 특히 WriteLog모드는 다른 모드들과 중첩해서 사용할 수 있으며, 현재까지 Serializer를 통해 이루어진 작업들이 로그로 저장되어, 디버깅에 유용하다.

242286	(620542) /Read => UInt16 : 0	242286	(620542) /Write => UInt16 : 0
242287	(620544) /Read => UInt16 : 52	242287	(620544) /Write => UInt16 : 52
242288	(620546) /Read => Int32 : 4	242288	(620546) /Write => Int32 : 4
242289	(620550) /Read => UInt16 : 0	242289	(620550) /Write => UInt16 : 0
242290	(620552) /Read => UInt16 : 0	242290	(620552) /Write => UInt16 : 0
242291	(620554) /Read => UInt16 : 0	242291	(620554) /Write => UInt16 : 0
242292	(620556) /Read => UInt16 : 0	242292	(620556) /Write => UInt16 : 0
242293	(620558) /Read => UInt16 : 0	242293	(620558) /Write => UInt16 : 0
242294	(620560) /Read => UInt16 : 0	242294	(620560) /Write => UInt16 : 0
242295	(620562) /Read => Int32 : 0	242295	(620562) /Write => Int32 : 0
242296	(620566) /Read => Float : 23959	242296	(620566) /Write => Float : 23959
242297	(620570) /Read => Float : 8900	242297	(620570) /Write => Float : 8900
242298	(620574) /Read => Float : -19427.5	242298	(620574) /Write => Float : -19427.5
242299	(620578) /Read => Float : 24187	242299	(620578) /Write => Float : 24187
242300	(620582) /Read => Float : 8900	242300	(620582) /Write => Float : 8900
242301	(620586) /Read => Float : -19220.4	242301	(620586) /Write => Float : -19220.4
242302	(620590) /Read => Float : 24403.6	242302	(620590) /Write => Float : 24403.6
242303	(620594) /Read => Float : 8900	242303	(620594) /Write => Float : 8900
242304	(620598) /Read => Float : -18992.4	242304	(620598) /Write => Float : -18992.4
242305	(620602) /Read => Float : 24608.8	242305	(620602) /Write => Float : 24608.8
242306	(620606) /Read => Float : 8900	242306	(620606) /Write => Float : 8900
242307	(620610) /Read => Float : -18836.6	242307	(620610) /Write => Float : -18836.6
242308	(620614) /Read => Float : 23757.6	242308	(620614) /Write => Float : 23757.6
242309	(620618) /Read => Float : 8900	242309	(620618) /Write => Float : 8900
242310	(620622) /Read => Float : -19562.4	242310	(620622) /Write => Float : -19562.4
242311	(620626) /Read => UInt16 : 4	242311	(620626) /Write => UInt16 : 4
242312	(620628) /Read => UInt16 : 0	242312	(620628) /Write => UInt16 : 0
242313	(620630) /Read => UInt16 : 1	242313	(620630) /Write => UInt16 : 1
242314	(620632) /Read => UInt16 : 2	242314	(620632) /Write => UInt16 : 2
242315	(620634) /Read => UInt16 : 3	242315	(620634) /Write => UInt16 : 3
242316	FileSize : 620636	242316	FileSize : 620636

Operator<< 를 여러 타입에 대해서 오버로딩 하고 있으며, 최종적으로 MemoryBuffer에 저장한다. 모드에 따라서 직렬화(WriteMode), 역직렬화(ReadMode) 가능하다.

### 3. 게임관련 데이터의 세이브와 로드->GameData

#### - Excel에서 에디팅

게임데이터는 원하는 데이터를 쉽고 빠르게 넣을 수 있어야 한다. Xml파일을 보면서 Attribute를 하나씩 추가하면 매우 번거로운 작업이 될 것이다. 그렇기 때문에 Excel의 매크로를 간단히 VisualBasic으로 작성해서 XmlMap을 만들고 그 XmlMap을 기준으로 Excel의 표를 Xml파일을 추출하도록 하였다.

Xml형식은 여러 개의 ChildElement를 GameDataElement하나에 넣는 형식이 아닌 GameDataElement안에 여러 Attribute를 가지고 있는 형식으로 작성했다.

#### - 로드 과정

Xml파일 하나를 기준으로 설명을 해보면,

1. 일단 GameDataName.Xml 파일을 파일 스트링으로 읽어서 RootElement형식으로 파싱을 진행한다.
2. RootElement의 ChildElement(하나하나가 GameData이다.)들을 순회하면서 XXXGameData를 만들어준다.
  - 2.1. 해당 ChildElement의 Attribute에서 'Key'의 값을 찾아서 생성된 XXXGameData의 \_key에 세팅한다.
  - 2.2. XXXGameData의 loadXml함수를 호출해서 Attribute에 있는 것들을 읽어서 처리한다.
    - 2.2.1. ChildElement->getAttribute("AttributeName") 와 같이 찾고, 처리한다.
    - 2.2.1.1. 꼭 있어야 하는 데이터와 없을 수 있는 데이터를 구분해서 코드를 작성한다.
    - 2.2.1.2. Enum 같은 경우는 CovertStringToEnum 함수를 사용한다. (뒤에서 설명)

2.2.1.3. 기타 변수(float, int32 등)은 FromStringCast<Type>(String)을 이용한다. (뒤에서 설명)

2.3. 초기화가 완료된 XXXGameDataKey를 key값으로 XXXGameData를 GameDataSet에 저장한다.

### 3. 게임관련 데이터의 세이브와 로드->UE4에서 에디팅

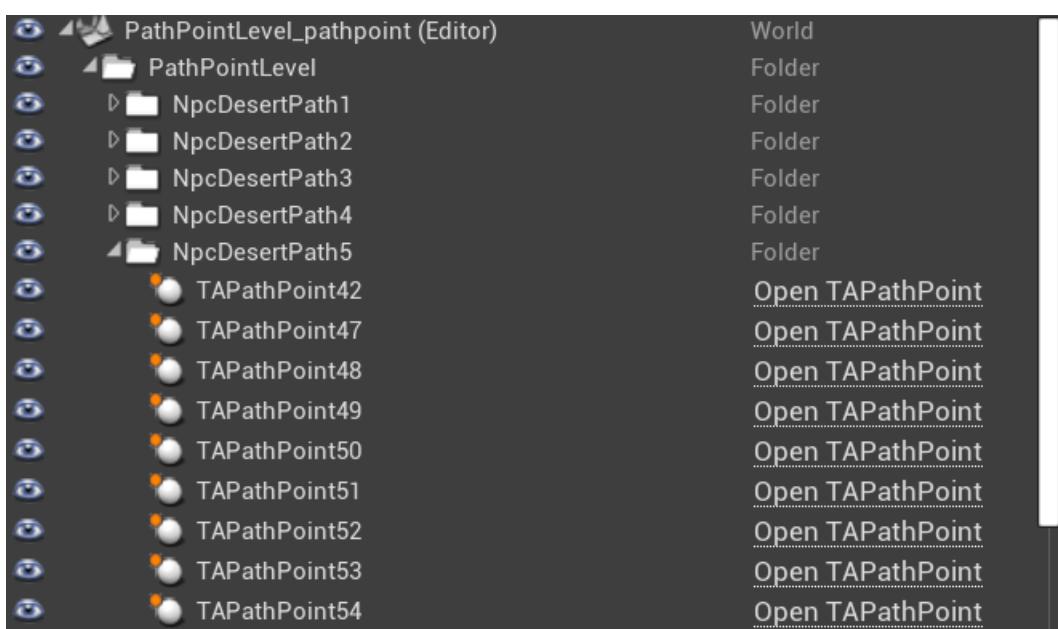
#### - 데이터들을 배치할 수 있는 레벨과 레벨태그

현재 엔진에서 UE4의 가장 상위 레벨은 Root 레벨이다. 그 아래에 DesertLevel\_map, DesertLevel\_spawndata, DesertLevel\_pathpoint 등이 존재한다. Root 레벨의 하위 레벨들은 뒤에 LevelName\_XXX(앞으로 레벨태그라고 부르겠다.)가 붙어 있는데, 현재 3가지의 형식이 있고, 역할은 다음과 같다.

- map : 일반적인 오브젝트 배치레벨이다. 레벨 스트리밍 볼륨에 의해서 인게임에서 스트리밍 된다.
- pathpoint : 경로 데이터를 에디팅하기 위한 레벨, 인게임에서 스트리밍 되지 않는다.
- spawndata : 스폰정보를 에디팅하기 위한 레벨, 인게임에서 스트리밍 되지 않는다.

// 액터는 서버에서 스폰하라고 내려준 액터들만 스폰시킨다.

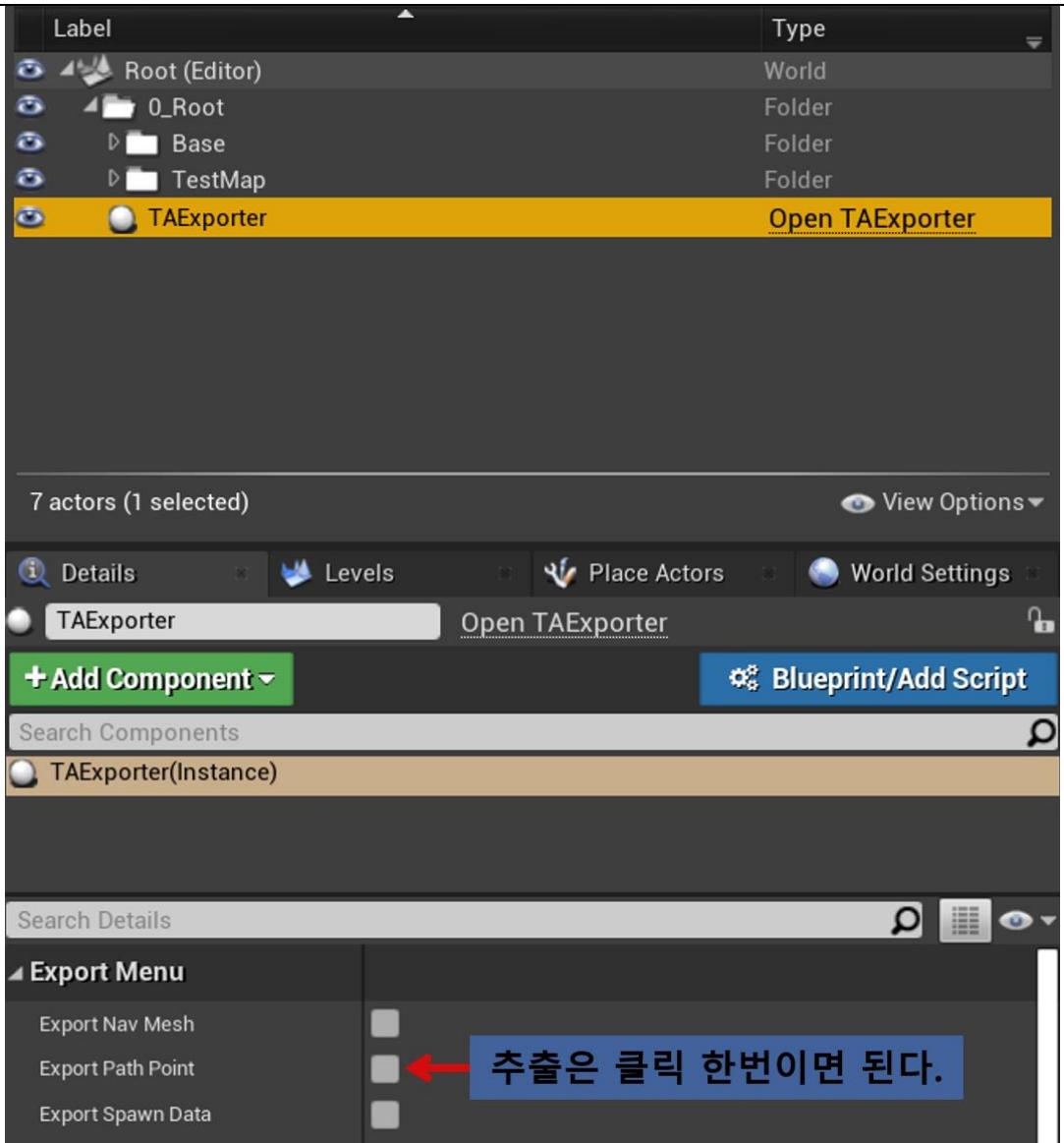
pathpoint와 spawndata의 레벨태그를 가진 레벨의 경우 최상위 폴더아래 하위 폴더를 만들고 거기에 pathpoint는 TAPathPoint 액터를 배치, spawndata는 TASpawnData 액터를 배치하면 된다. 아래는 pathpoint의 경로정보들을 폴더로 나누어서 TAPathPoint를 배치한 모습이다.



#### - 에디터에서 Play 없이 클릭 한번으로 각 데이터를 추출

일단 추출할 데이터들을 에디팅을 하기 위해서는 For\_Editing이라는 것을 정의 해 놓아야 사용할 수 있도록 하였다. 그 이유는 에디팅에 필요한 모듈과 함수들이 게임 패키징 할 때, 사용할 수 없기 때문이다.

Root 레벨에는 데이터 추출을 위한 특별한 액터 하나가 존재한다. 그 액터는 TAEporter라는 액터인데, 이 액터가 모든 추출작업을 담당한다. 예를 들어서 ExportPathPoint를 클릭하면 레벨태그가 pathpoint인 레벨들을 쭉 찾아서 해당 레벨의 하위폴더까지 모두 추출해준다.



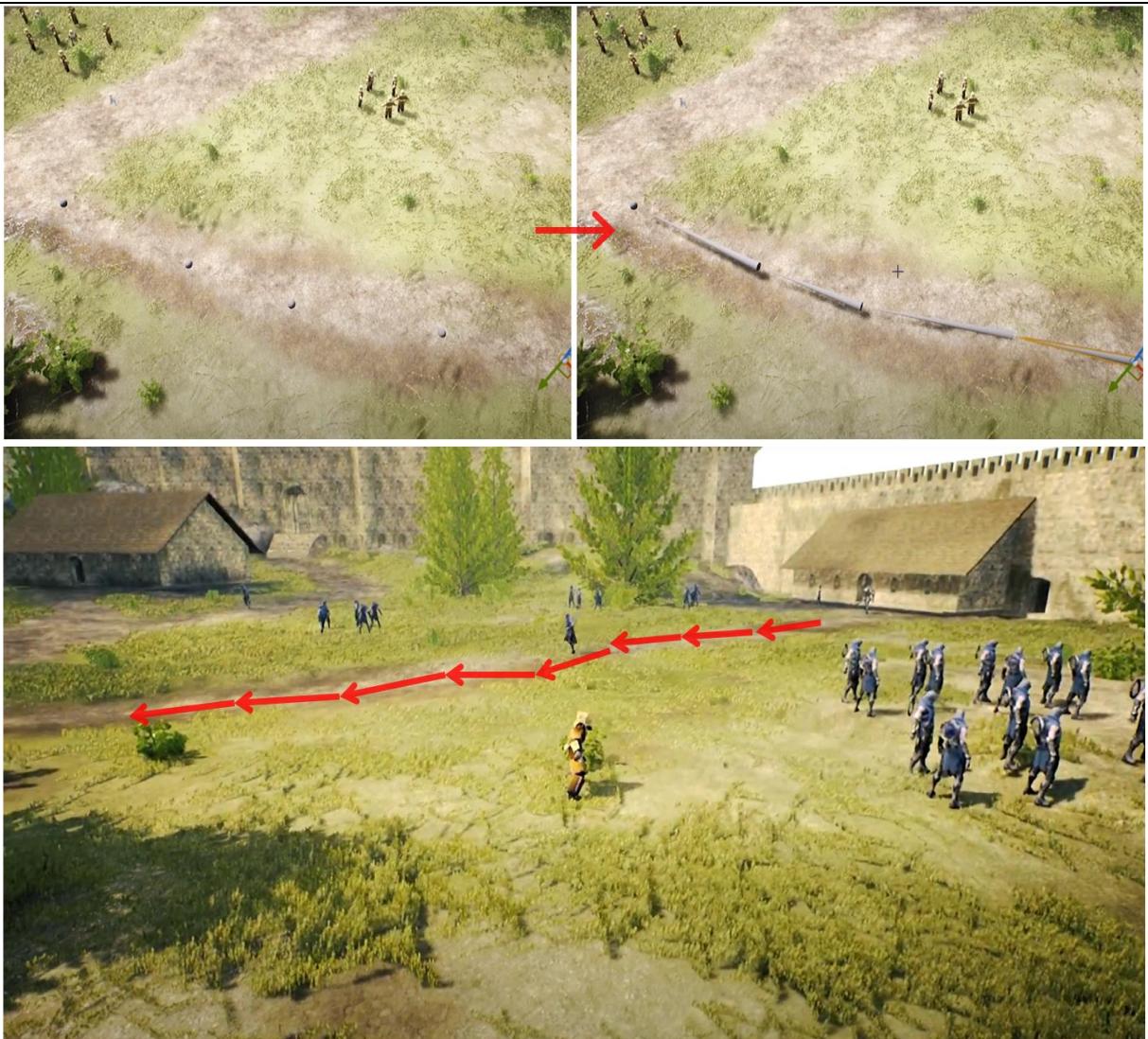
### 3. 게임관련 데이터의 세이브와 로드->PathPoint

#### - PathPoint 데이터가 필요한 이유

현재 RecastNavigationMesh가 서버쪽에는 로드가 된 상태이기 때문에 그냥 DestinationPoint 하나만 있으면 되지 않을까라는 생각도 했는데, 특정 포인트를 확정적으로 지나는 Npc경로를 만들고 싶어서 엔진에서 지원하게 만들었다.

#### - UE4에서 에디팅

앞에서 나온 내용과 같이 pathpoint 레벨태그를 가진 레벨을 만들고, 그 레벨에 폴더를 만들고 TAPathPoint 객체를 순서대로 배치하면된다. 가장 처음에 위치한 TAPathPoint가 시작점이다. TAExporter를 통해서 PathPoint들을 추출하면 해당레벨의 이름.xml 파일로 추출된다. 공중에 TAPathPoint가 위치할 경우 AttachToTheGround 클릭을 통해서 땅에 붙일 수 있는 기능을 제공한다.



### - 로드 과정

추출된 xml파일들은 기본적으로 위치 정보의 나열이라고 보면 된다. 이 파일들을 병렬적으로 모두 로드해서 ServerMoveActorSystem의 `unordered_map<PathPointPathKey, PathPointPath*>` PathPointPathSet에 보관하고 있다. 파일들을 읽어서 먼저 PathPoint들을 생성하고 이를 가지고 PathPointPath객체를 생성한다.

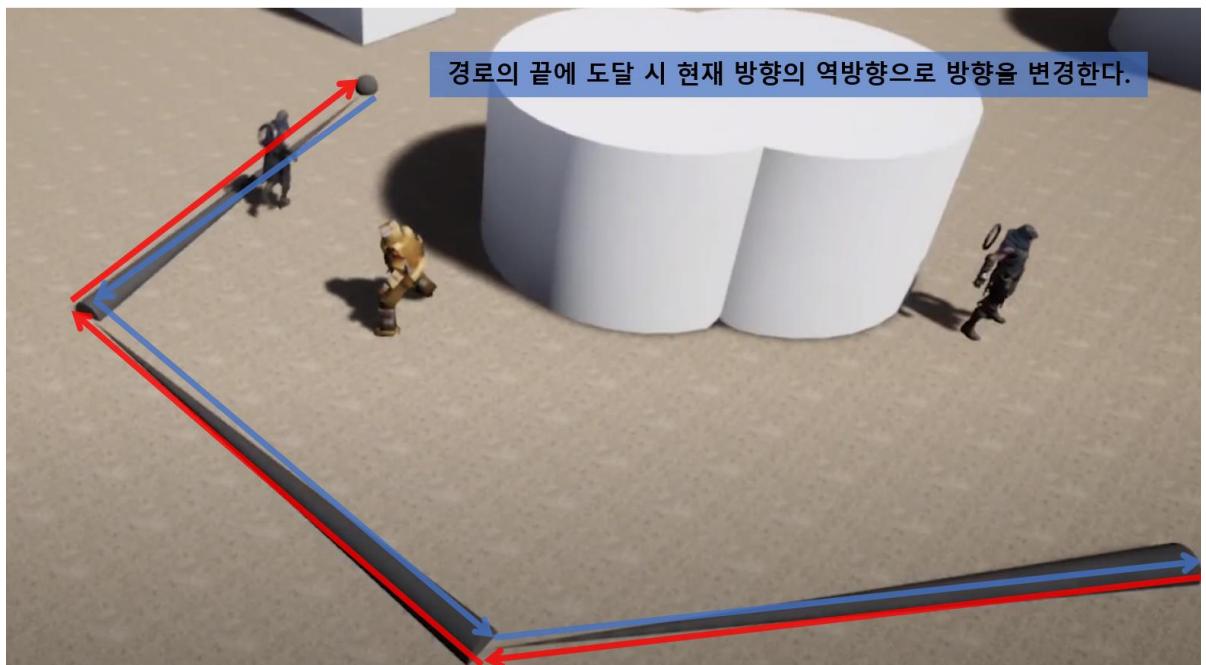
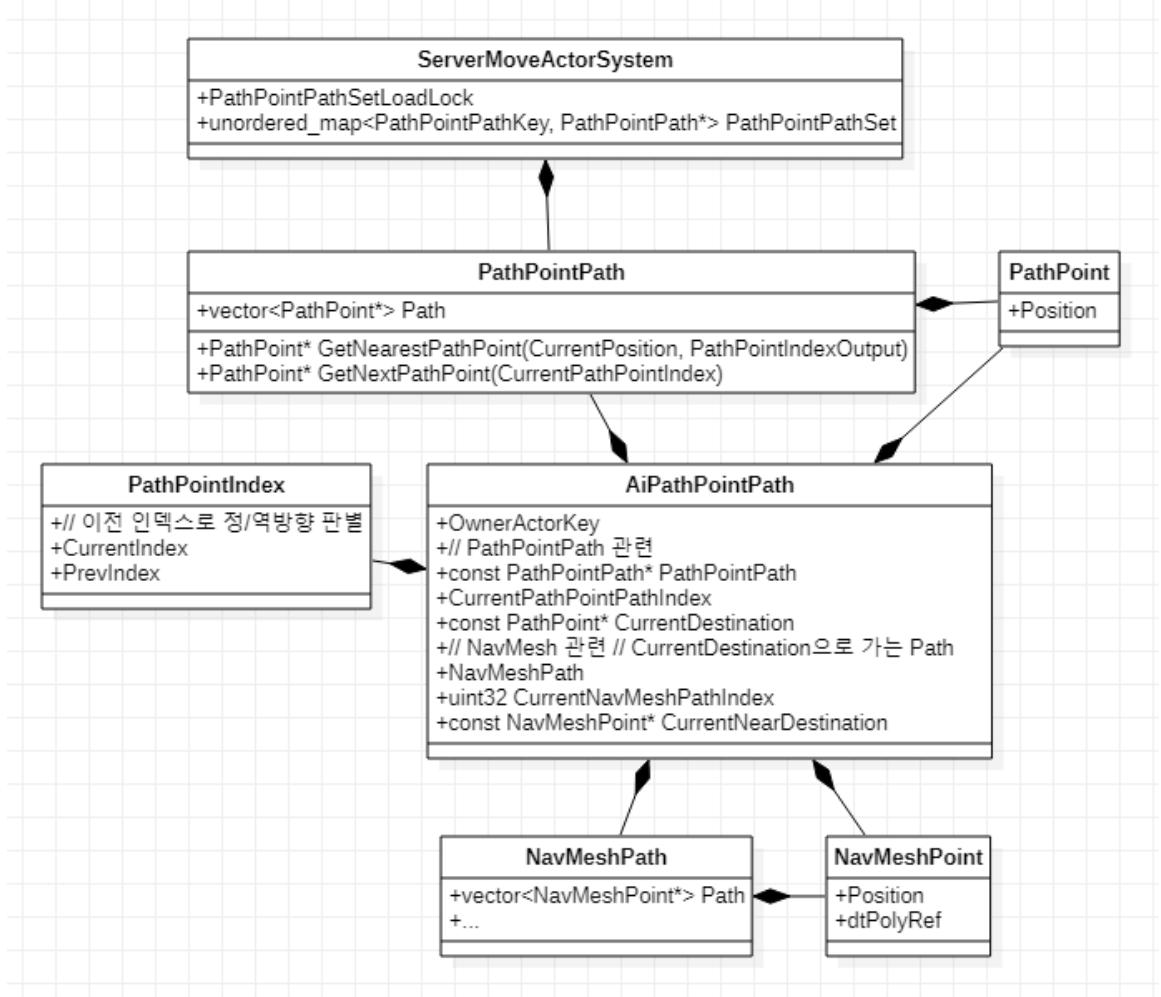
그리고 잠깐 락을 걸어서(다른 스레드도 같이 로드 중) PathPointPathSet에 PathPointPathKey(파일이름 해시값)와 함께 삽입해준다.

이때 PathPointPath객체는 PathPoint들을 배열로 가지고 있는 클래스이며, 위치에 따라서 가장 가까운 PathPoint를 찾아주거나, PahtPoint들을 정방향으로 순회하고, 끝까지 갔을 때 다시 역방향으로 순회할 수 있도록 도와주는 역할을 한다.

### - 동작 원리

- 특정 Npc의 ServerAiActorComponent가 초기화 될 때 PathPointPathKey가 있으면 Path를 초기화한다. ServerMoveActorSystem에서 해당 Key로 PathPointPath를 받아와서 AiPathPointPath를 생성해준다. 이때 AiPathPointPath는 RecastNavMesh를 통해서 PathPointPath의 PathPoint들을 순회하는 기능을 가진 클래스이다. PathPointPath를 가지고 있고, 현재위치에서 다음 목적지 PathPoint까지의 NavMeshPath를 가지고 있다.

2. 이 Npc가 AiTick마다 MoveToPathPoint를 수행한다면, 처음에는 가장 가까운 PathPoint를 찾는다.
3. 가장 가까운 PathPoint를 목적지로하는 NavMeshPath를 생성하고, NavMeshPoint들을 차례로 이동한다.
4. NavMeshPath의 마지막 NavMeshPoint까지 도달했을 때 다음 PathPoint를 찾고 NavMeshPath 생성한다.
5. PathPointPath는 마지막 PathPoint에 도달하면 방향을 반대로 바꿔준다.
6. 3~5번이 반복된다.



### 3. 게임관련 데이터의 세이브와 로드->SpawnData

#### - UE4에서 에디팅

앞에서 나온 내용과 같이 spawndata 레벨태그를 가진 레벨을 만들고, 그 레벨에 폴더를 만들고 TASpawnData 객체를 원하는 위치에 배치하면 된다. 추출되는 정보는 Position, Rotation, GroupGameDataKey가 추출된다. GroupGameData는 어떤 캐릭터형 액터를 스폰할지 결정하는 식별자라고 보면 된다. 만약에 DesertLevel\_spawndata 레벨에 Common이라는 폴더를 만들고 TASpawnData를 배치하면, DesertLevel\_Common.xml 파일로 추출된다. AttachToTheGround 클릭을 통해서 땅에 붙일 수 있는 기능을 제공한다.

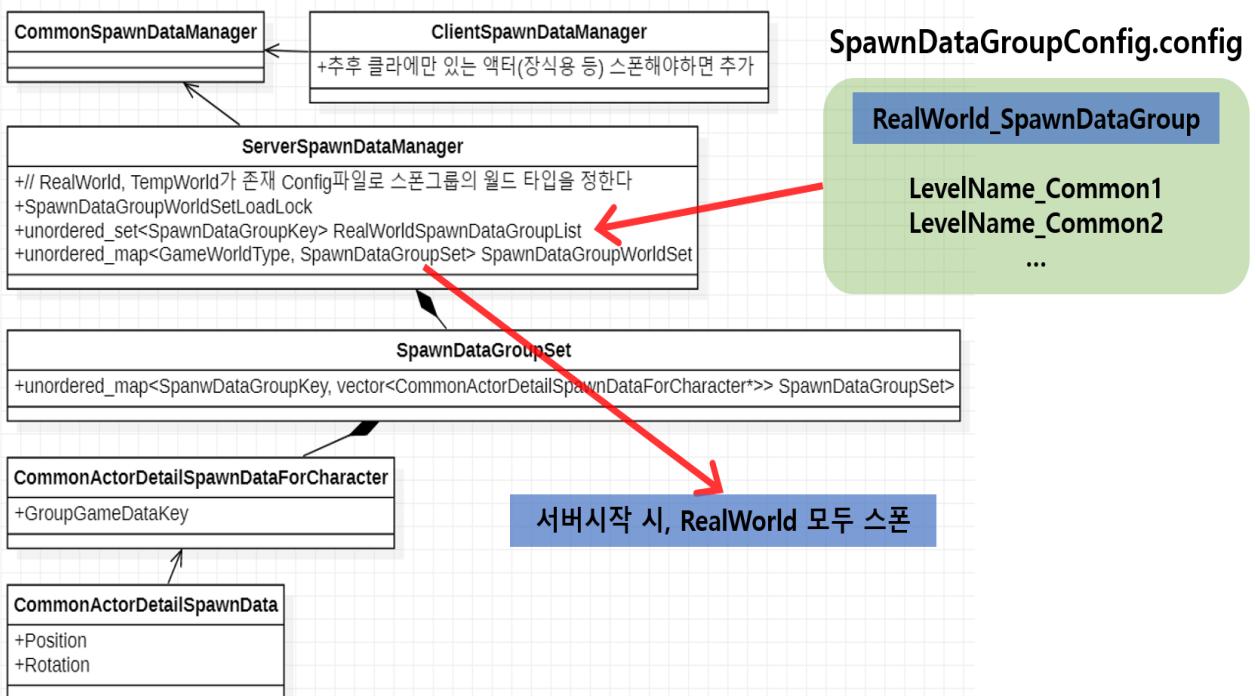
#### - 로드 과정

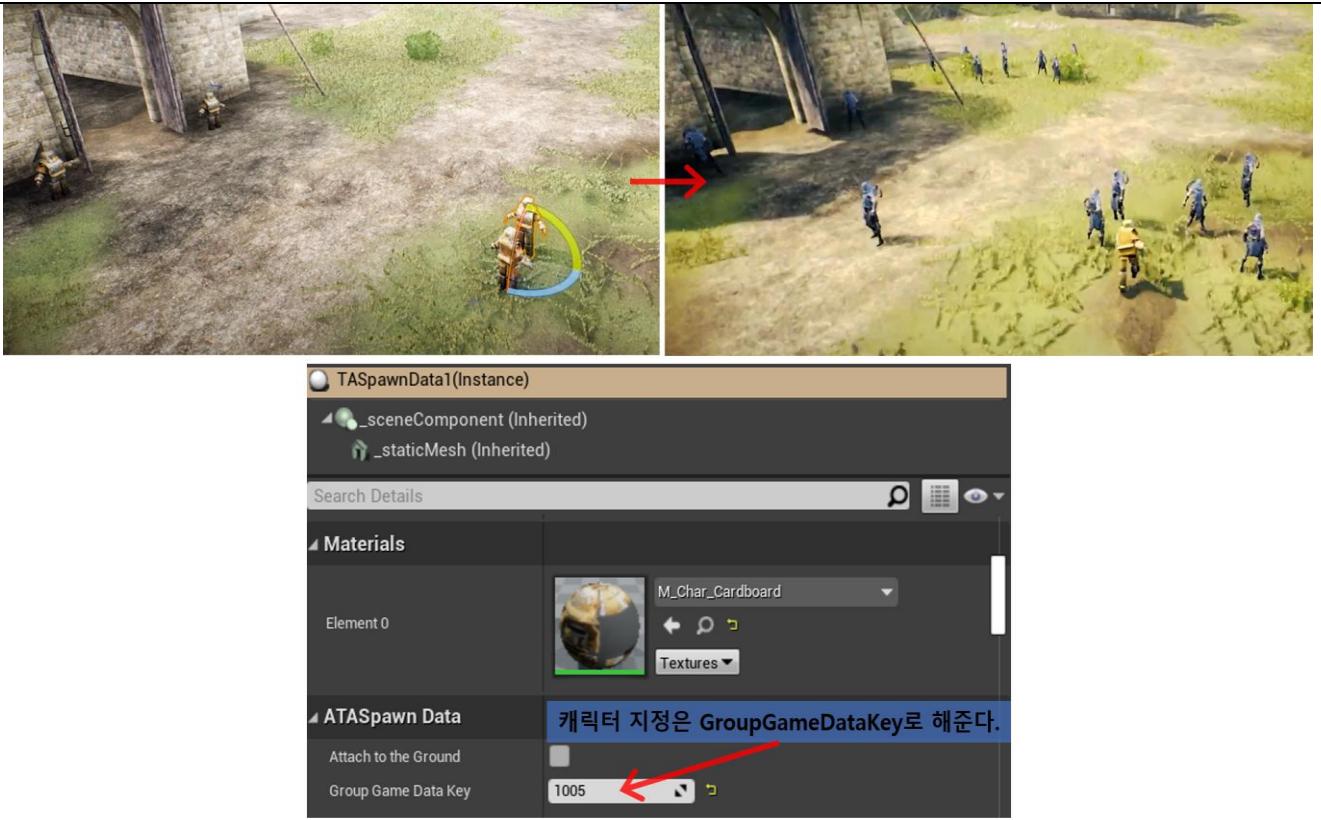
일단 서버에서 SpawnData 폴더 있는 xml파일들과 config파일을 모아준다. config파일은 어떤 xml파일이 ReadWorld에 스폰될지 결정해주는 구성정보를 가지고 있는 파일이라고 보면 된다. 여기서 RealWorld에 존재하는 액터들은 인게임에서 그 위치에서 상주하는 액터라고 보면 된다. 만약 공격을 받아 죽어도, SpawnData에 의해서 다시 스포너된다. 이와 다른 개념으로 TempWorld에 포함되는 액터들이 존재하는데, 이는 필요에 의해서만 스폰 되는 액터들을 말한다. (시나리오 진행을 위한 액터 등)

그렇게 config파일에 의해서 RealWorld월드에 스폰될 xml파일들을 판별해 내면, 파일을 병렬적으로 로드한다. 파일내에 있는 모든 SpawnData(Position, Rotation, GroupGameDataKey)를 로드해서 객체로 만들고 잠깐 락을 걸고 unordered\_map<GameWorldType, SpawnDataGroupSet> SpawnDataGroupWorldSet에 삽입한다.

#### - 동작 원리

그렇게 처음 모든 SpawnData를 로드하고 나면 서버를 시작할 때 RealWorld에 존재하는 모든 액터들을 스폰시켜준다.





### 3. 게임관련 데이터의 세이브와 로드->Recast/Detour Navigation

#### - UE4의 RecastNavMesh

UE4 에디터 상에서 직접 RecastNavMesh를 쉽게 생성할 수 있으며, Property도 쉽게 조정할 수 있도록 에디팅 환경이 구축되어 있다. 그렇게 만들어진 RecastNavMesh 정보를 그대로 직렬화해서 서버에서 역직렬화 해서 사용하는 것으로 만들어야겠다는 생각을 하였다.

가장 첫번째로 해야할 점은 현재 생성된 저 NavMesh의 데이터들이 어디에 저장되어 있는지 파악하는 일인데, 여러 디버깅과 코드확인 결과 UE4에서 생성된 RecastNavMesh 정보들은 UNavigationSystemV1->ARecastNavMesh->FPIImplRecastNavMesh->dtNavMesh(Recast / Detour 라이브러리에 포함된 클래스)에 저장되어 있었다. 해당 정보는 에디터에서 플레이 하고 있지 않아도, 생성되어 있는 정보이기 때문에, 바로 추출이 가능하다. 이를 추출한 파일을 똑같은 로직으로 역직렬화해서 서버에서 사용 중이다. (Recast / Detour 라이브러리도 엔진에서 std형식에 맞춰 조금 변경한 후 그대로 사용중이다.)

### 3. 게임관련 데이터의 세이브와 로드->UE4에서 리소스 로드

#### - 리소스의 동적 비동기 로드

##### - 동적 비동기 로드조건

확인 결과, 동적 비동기를 수행하기 위해서는 두 가지 조건이 필요하다.

첫번째 조건은 비동기로 에셋을 로드하기 위해 FStreamableManager가 필요한데, 이는 여走路에서 사용할 일이 많을 것 같아서 TAGameInstance에 멤버로 추가하였다. 그리고 SoftObjectPath라는 변수가 필요하다. 이 변수는 간단하게 에셋의 파일경로라고 보면 된다. 사용하는 방법은 FStreamableManager->RequestAsyncLoad(SoftObjectPath, FStreamDelegate::CreateUObject(콜백함수를 부를 객체, &콜백함수)) 이런 식으로 작성해주면 작업이 완료 되었을 때, 해당 콜백함수를 통해서 성공 실패 여부를 알려주고,

성공했다면 로드된 객체를 얻어서 사용할 수 있다.

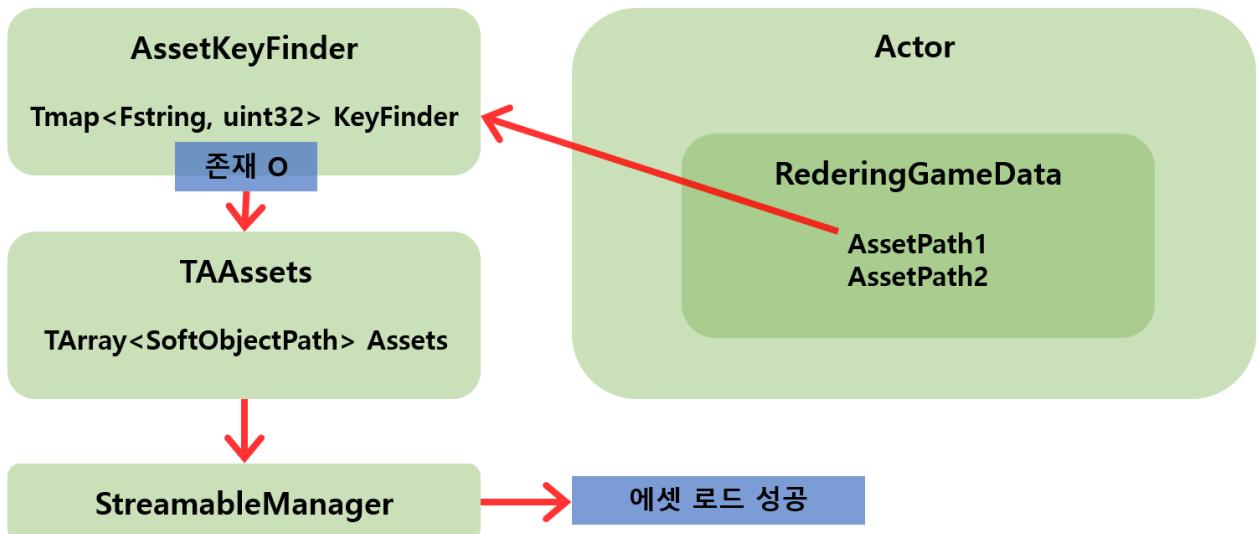
두 번째 조건은 해당 SoftObjectPath가 이미 로드 되어있어야 한다는 것이다. 미리 로드 하기 위해서 TAGameClientSetting 모듈을 하나 더 만들고 그 안에 TAAAssets이라는 에셋들의 SoftObjectPath들을 담고 있는 UObject 클래스 하나를 만든다. (이 SoftObjectPath들은 Config/DefaultTAAAssets.ini 파일을 통해서 추가 된다.) 그리고 TAGameClient에서 TAAAssets을 사용하게 하기 위해서 TAGameClientSetting 모듈이 더 빨리 로드 되도록 TAGameClient.uproject 파일에 해당 모듈의 json 값을 수정한다.

"LoadingPhase": "PreDefault"로 설정하면 "Default"로 설정한 TAGameClient 모듈보다 더 빨리 로드된다.

두 번째 조건에서 DefaultTAAAssets.ini 파일에 에셋 경로를 추가해야 하나 의아해서, 해당 파일에 에셋 경로를 추가하지 않고, 경로 문자열을 그대로 들고와서 SoftObjectPath로 만들고 그 객체로 그대로 동적 비동기 로드를 시켜봤는데, 작동하지 않았다.

### - 데이터 검증을 위한 방법

UE4에서 여러 가지 리소스를 로드하기 위해서는 액터에 따라서 어떤 리소스를 사용할지 정해줘야 한다. 그 정보는 RederingGameData가 가지고 있는데, 이안에 있는 에셋 경로들로 에셋들을 로드하기 전에 해당 에셋이 DefaultTAAAssets.ini 파일에 정의되어 있는지 확인해봐야 한다. 이를 위해서 TAGameInstance에서 해당 에셋이 있는지(있다면 TAAAsset에 있는 SoftObjectPath들 중 몇 번째 인덱스에 있는지) 없는지에 대해서 판별하기 위해서 TMap< FString, uint32> AssetKeyFinder를 유지하고 있다.

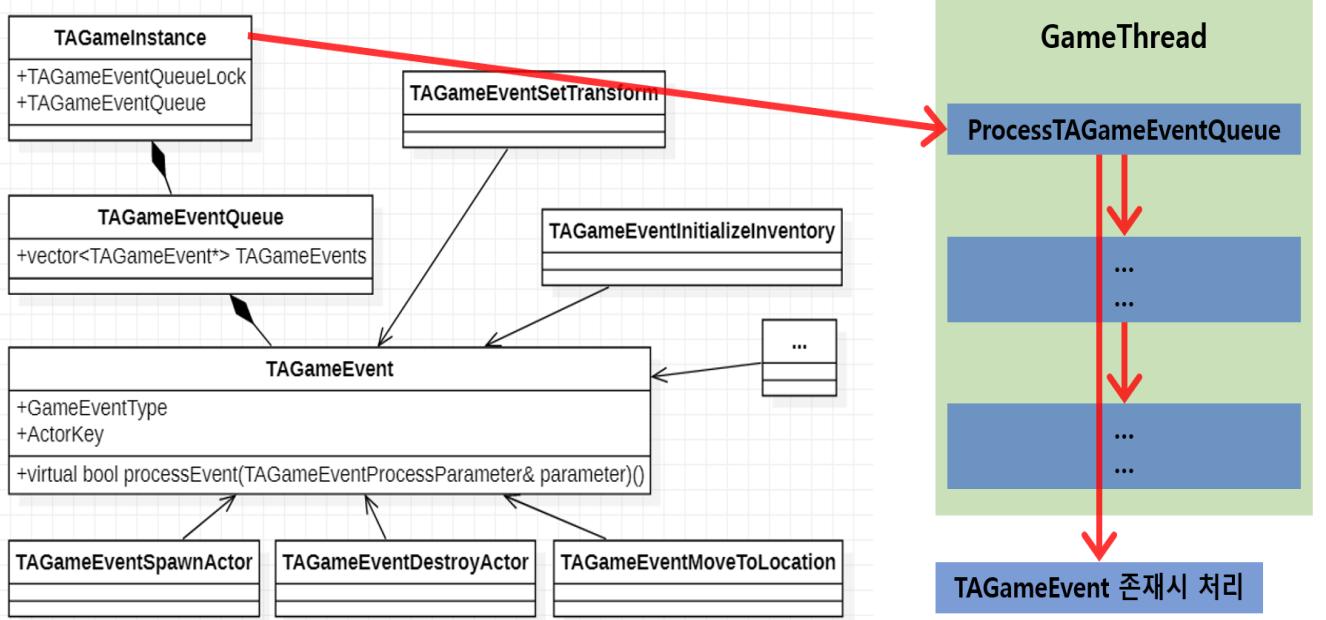


## 4. 컨텐츠->TA게임엔진과 UE4의 통신

### - TAGameEvent

UE4에서 게임루프는 싱글스레드(게임스레드)에서 돌아간다. 근데 엔진의 TAGameClient는 멀티스레드로 동작한다. 그렇기 때문에 TAGameClient에서 무작정 UE4에 있는 메모리에 접근하면 안된다. (데이터 레이스 발생) TAGameEventQueue를 이용해서 UE4와 TAGameClient가 통신한다.

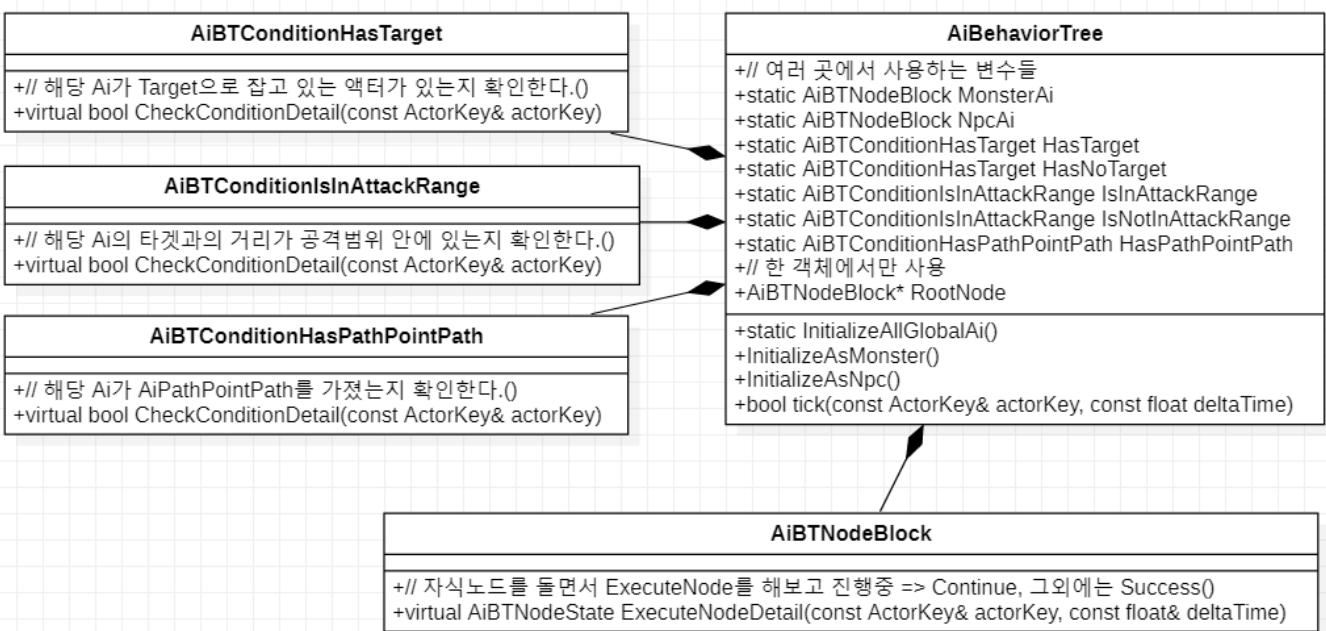
동작 원리를 간단히 설명해보면 먼저 TAGameEventQueue는 UE4의 TAGameInstance에 존재한다. 엔진의 TAGameClient에서 UE4로 메시지를 보내야 할 때, TAGameEventQueue에 이벤트를 추가한다. (당연히 동기화를 위한 락이 필요하다.) 그리고 나면 TAGameInstance가 매 Tick마다 TAGameEventQueue를 확인해서 이벤트가 있으면 처리한다.



## 4. 컨텐츠->AI

### - BehaviorTree

엔진에서는 Ai처리를 BehaviorTree로 하고 있다. BehaviorTree에는 크게 AiBTNode, AiBTNodeComposite, AiBTNodeState, AiBTNodeCondition, AiBTNodeExecution으로 이루어져 있고, BehaviorTree의 일반 멤버 변수는 AiBTNodeBlock 하나로 되어 있다. 그리고 액터들의 다양한 Ai를 초기화 하기 위해 여러 개의 static AiBTNodeCondition(AiBTNode를 실행하는 조건), static AiBTNodeBlock(자식 AiBTNode를 가지며 순차적으로 순회하는 노드)를 가진다. (각 내용의 자세한 설명은 뒤에서 할 예정이다.)



### - AiBTNode

가장 일반적으로 사용되는 BehaviorTree의 노드로, BehaviorTree는 루트 AiBTNode부터 시작해 많고 다양한 여러 개의 AiBTNode로 구성되어 있다. BehaviorTree를 실행시키면 이 루트 AiBTNode부터

시작해서 그 아래 있는 자식 AiBTNode까지 ExecuteNode를 실행한다.

AiBTNode는 멤버로 AiBTNodeCondition배열과 AiBTNodeExecution배열을 가진다.

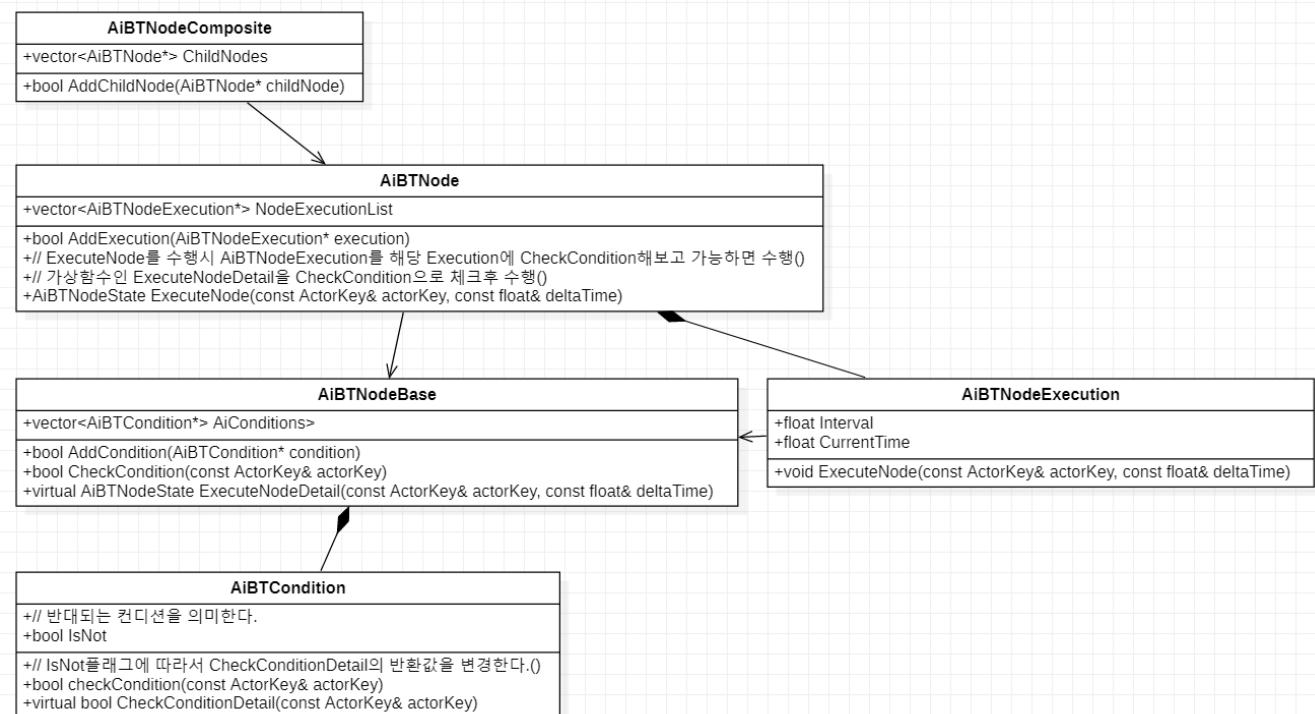
AiBTNodeCondition은 간단히 해당 노드를 실행할 조건이 되는지 체크하는 객체다. 그래서 해당 배열에서 각 AiBTNodeCondition객체들을 순회하면서 하나라도 조건이 안되면 해당 노드가 실패하게 된다.

AiBTNodeExecution은 해당 노드가 실행될 때 같이 실행되는 동작을 담고 있는 객체다.

그리고 ExecuteNodeDetail이라는 가상함수를 가지는데, 이 함수는 ExecuteNode할 때

AiBTNodeCondition이 모두 충족되었을 때, 수행된다. 즉 AiBTNode를 상속받은 객체의 실제 구현내용은 ExecuteNodeDetail이라는 함수를 오버라이딩해서 넣어주면 된다.

그리고 현재 노드의 실행결과를 나타내는 AiBTNodeState라는 것이 존재한다. ExecuteNode 수행 시, 해당 노드가 성공할 때 AiBTNodeState::Success, 실패할 때 AiBTNodeState::Failure, 진행 중일 때 AiBTNodeState::Continue를 반환한다. 이 반환 결과에 따라서 다양한 방식으로 동작하는 AiBTNodeComposite가 존재하고, 이를 조합해서 통해서 Ai를 구성한다.



## - AiBTNodeComposite

AiBTNodeComposite는 자식 노드를 가질 수 있는 노드다. 자식 노드의 ExecuteNode 결과 값에 따라서 다양한 방식으로 동작하며, 종류는 3가지로 나뉜다. 3종류 모두 자식노드가 AiBTNodeState::Continue를 반환하면 그대로 반환한다.

### 1. Selector

: 자식노드의 ExecuteNode결과 값이 하나라도 성공 시 AiBTNodeState::Success을 반환하고, 모두 실패 시 AiBTNodeState::Failure을 반환한다.

### 2. Sequence

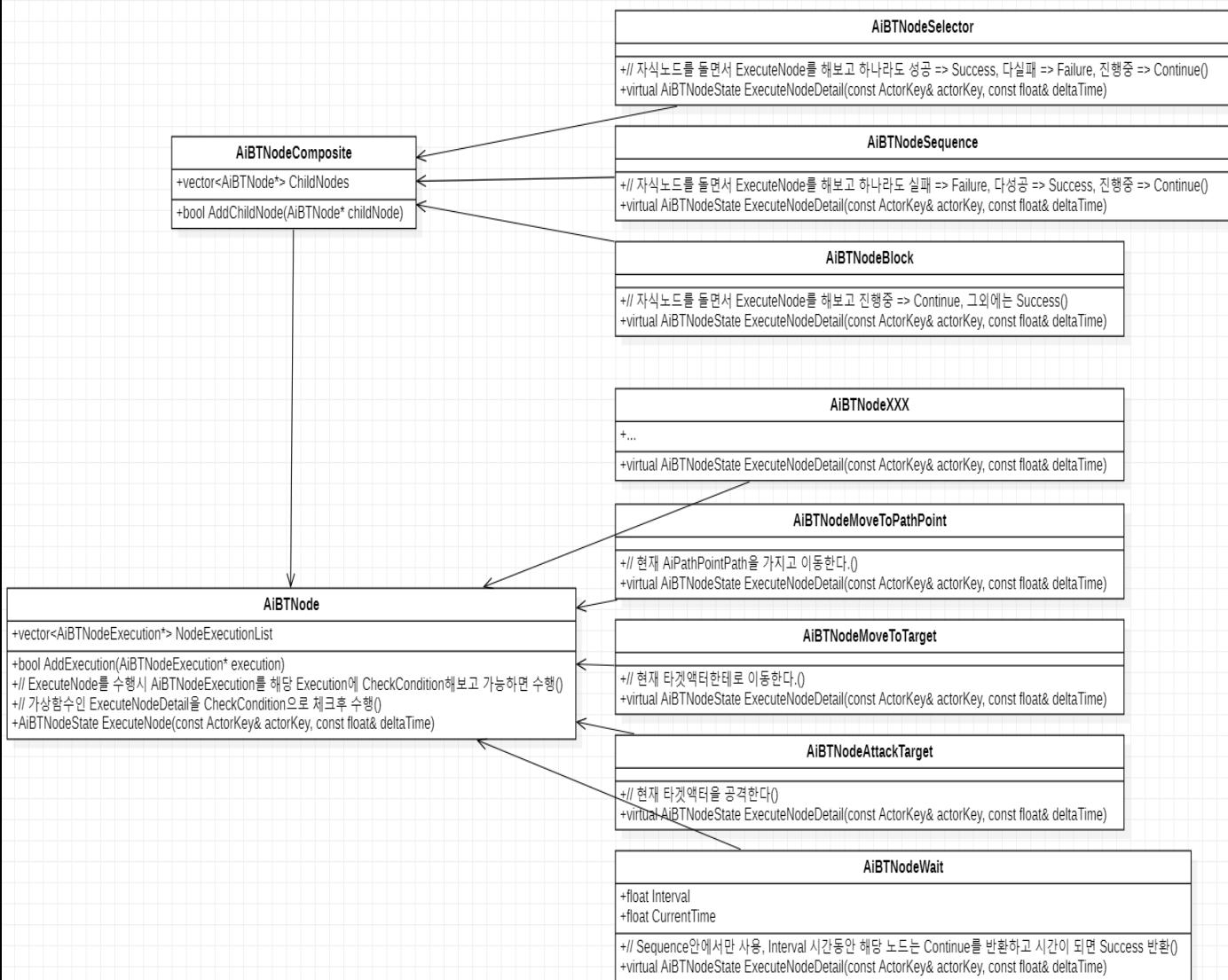
: 자식노드의 ExecuteNode결과 값이 하나라도 실패 시 AiBTNodeState::Failure을 반환하고, 모두 성공 시

AiBTNodeState::Success을 반환한다.

### 3. Block

: 자식노드의 ExecuteNode결과 값에 상관없이 무조건 AiBTNodeState::Success을 반환한다.

현재 있는 AiBTNode들과 상속구조를 나열해보면 다음과 같다.



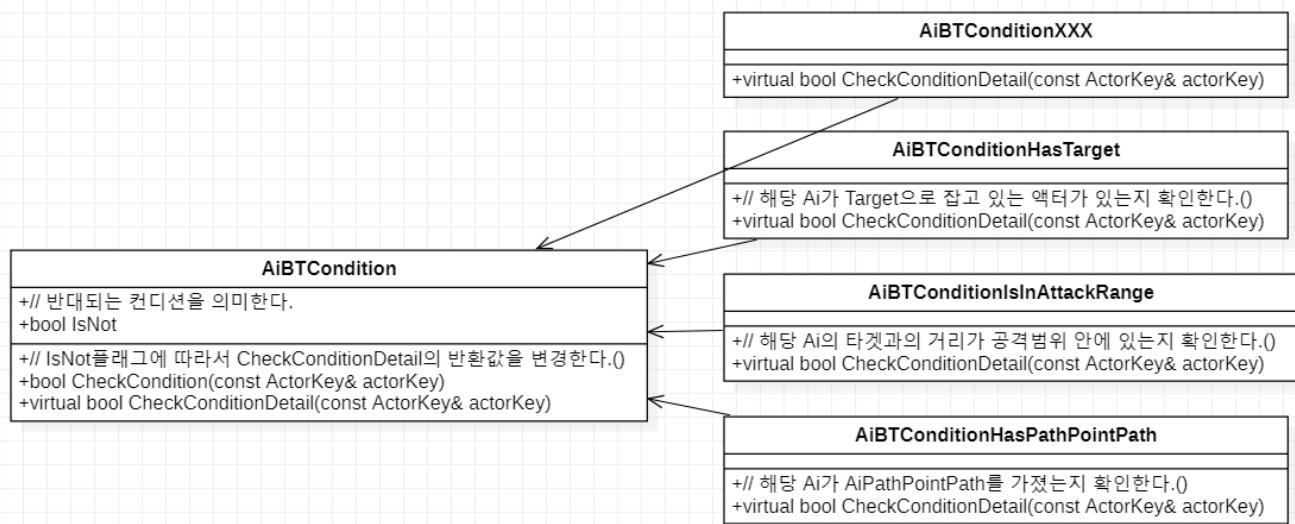
### - AiBTNodeCondition

AiBTNode가 ExecuteNode를 수행할 때 AiBTNodeCondition들을 가장 먼저 체크한다. 공통적으로 많이 사용하는 조건은 BehaviorTree의 static 변수로 존재한다.

멤버변수로 IsNot이라는 플래그 하나가 존재한다. 이는 해당 컨디션의 값을 부정으로 바꿔주는 플래그이다. 만약 AiBTConditionHasTarget이라는 조건에서 IsNot플래그가 켜져있으면 타겟을 가지지 않는 조건이 된다.

멤버함수로 CheckCondition과 가상함수인 CheckConditionDetail이 존재하는데, CheckCondition은 CheckConditionDetail을 수행해보고 IsNot플래그에 의해 최종적으로 true/false를 결정해 반환한다. 즉 AiBTNodeCondition을 상속받은 클래스는 CheckConditionDetail만 오버라이딩 해주면 된다.

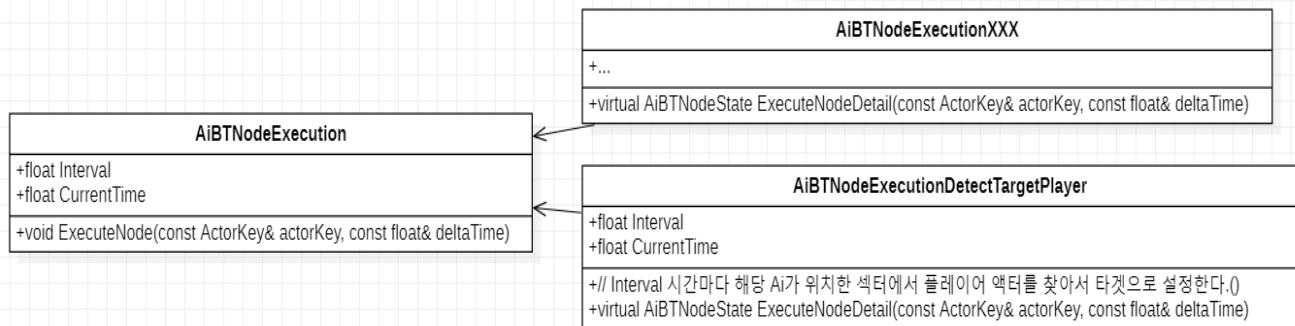
현재 있는 조건들과 상속구조를 나열해보면 다음과 같다.



### - AiBTNodeExecution

AiBTNode가 ExecuteNode를 수행할 때 AiBTNodeExecution도 같이 수행한다. 이는 노드를 순회할 때 조건에 따라서 주기적으로 해야 할 것들을 넣어주는 객체라고 보면 된다. AiBTNodeExecution도 AiBTNodeBase를 상속받고 있기 때문에, AiBTNodeCondition을 추가해서 특정 조건에서만 수행되도록 설정할 수 있다.

현재 있는 실행객체들과 상속구조를 나열해보면 다음과 같다.



## 4. 컨텐츠->Move

### - Sector이동과 통보

액터들의 이동에는 Sector라는 개념이 빠질 수 없다. Sector는 정방형의 일정한 크기의 영역이라고 보면 된다. 인게임에서 월드는 이 Sector의 집합으로 이루어져 있다. 액터들은 항상 한 Sector에 포함되어 있다. 액터가 다른 Sector로 넘어갈 만큼 움직이면, 이동한 새로운 Sector의 액터리스트에 해당 액터가 추가되고, 이전 Sector에서는 해당 액터가 삭제된다.

Sector 개념이 필요한 이유는 간단히 말하면 최적화를 위해서다.

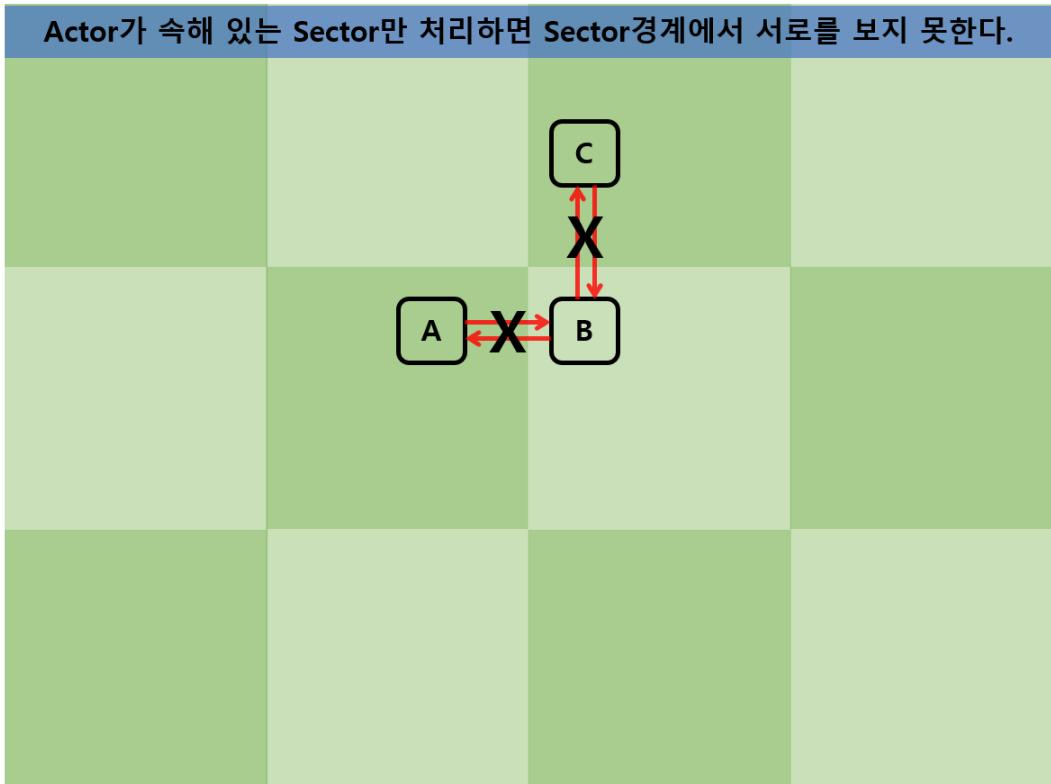
간단히 예시를 몇 개 들어보면, Sector 개념이 없다면 몬스터 액터가 플레이어 액터를 타겟으로 잡기 위해서 월드에 있는 모든 플레이어 액터리스트를 가지고 와서 거리를 비교해가면서 타겟을 찾아야 한다.

또 멀티플레이어 게임 프로그래밍 관점에서 예를 들어보면, 멀티플레이어 게임에서는 내가 움직이면

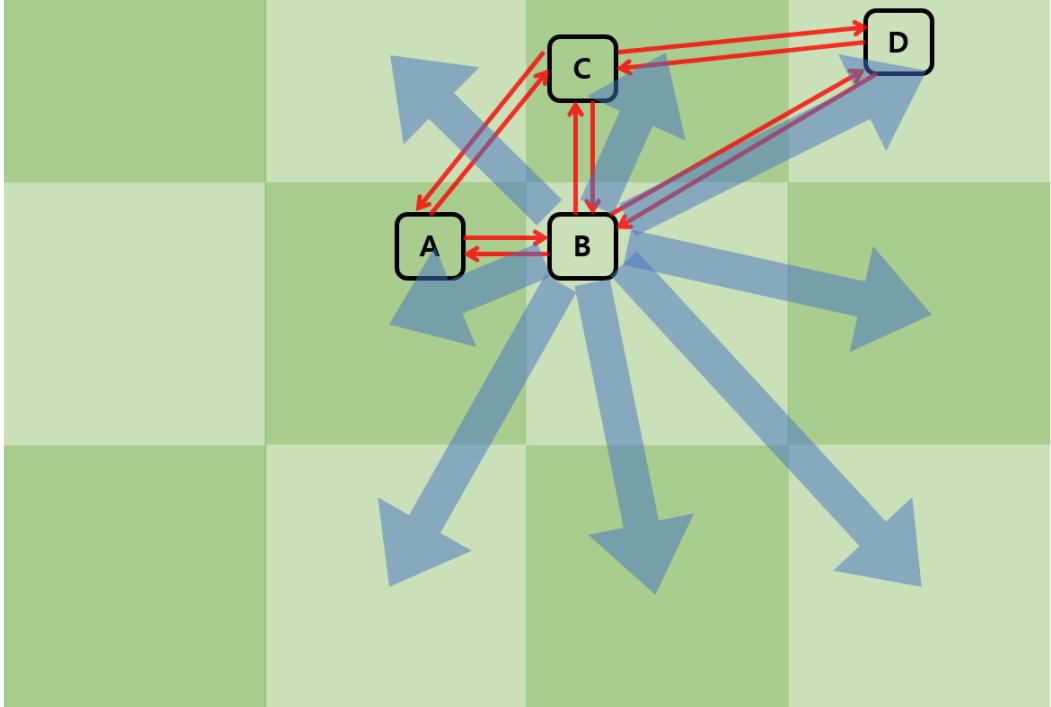
상대방에게 내가 움직이고 있다는 것을 실시간으로 알려 줘야한다. 그래야 상대방 클라이언트에서도 움직이는 내 캐릭터가 렌더링 될 것이다. 만약 Sector라는 개념이 없다면, 내 캐릭터 액터가 움직일 때마다 현재 시야에서 보이지도 않는 모든 플레이어에게 패킷을 쏴 줘야 한다.

이런 상황들을 막기 위해서 Sector개념이 필요하다. Sector가 있는 경우에, 위의 두 가지 경우 모두 Sector안에 있는 액터로 한정시킬 수 있다.

그렇다면 액터가 속해있는 Sector의 액터리스트만 받아서 처리하면 될까 싶었지만, 그렇진 않았다. 그렇게 할 경우에 현재 Sector의 경계에 있다고 가정하면 바로 앞에 있지만 처리 못하는 경우도 생긴다. 그래서 현재 Sector와 주위 Sector도 같이 처리한다.



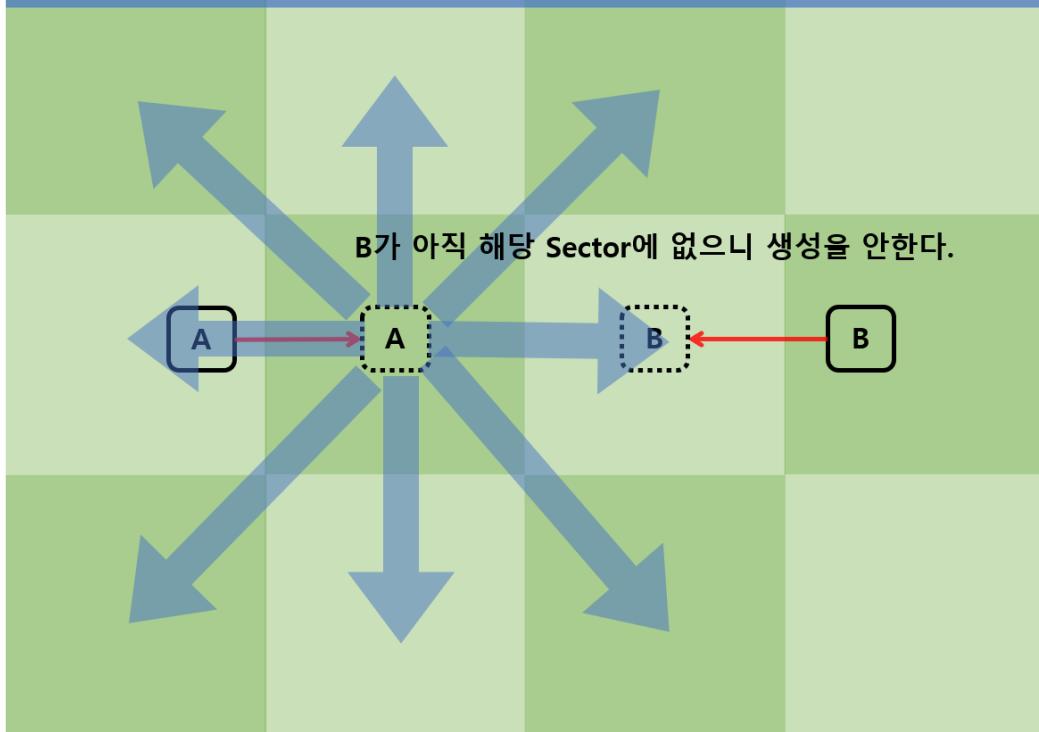
Actor가 속해 있는 Sector와 주위의 Sector까지 처리한다.



이제 그렇다면 움직일 때 서로 움직인다고 알려주기만 하면 된다. Sector를 이동했을 때, 일단 새로 생긴 Sector들의 액터들에게 나를 생성하라고 말해주고, 나도 새로 생긴 Sector들의 액터들을 생성하면 된다. 지워진 Sector들은 액터들에게 나를 지우라고 말해주고, 나도 지워진 Sector들의 액터들을 지우면 된다.

이렇게 하면 될 것 같았지만, 멀티스레드에서 환경에서는 문제가 생길 수 있다. 예를 들어 A, B가 동시에 Sector를 이동하고, A, B가 서로 인접 Sector에 위치하게 되는 경우에 문제가 생긴다. 서로 Sector이동 시에 A입장에서는 새로운 인접Sector에 B가 없을 수 있고, B입장에서는 새로운 인접 Sector에 A가 없을 수 있다. 그렇게 되면 서로 인접Sector에 위치하고 있어도 서로를 모를 수 있다. 이는 거의 동시에 Sector를 이동하는 상황에 발생한다.

A, B가 동시에 Sector를 이동하면 인접Sector에 위치해도 서로 모를 수 있다.



그렇기 때문에 액터는 ViewList라는 것을 관리한다. 이는 현재 액터가 보고 있는 액터들을 나타낸다. 만약 위에 그림과 같은 상황이 나타나도, A가 움직이면서 인접Sector들의 액터들에게 움직이는 것을 알려주려고 봤는데, 인접Sector의 액터인 B가 아직 ViewList에 없다면, A한테 B를 생성시켜줄 수 있다. 반대로 B의 ViewList에 A가 없으면 생성시켜줄 수 있다. 결론적으로 ViewList는 멀티스레드 때문에 누락된 액터들을 뒤늦게라도 처리해주는 역할을 하는 것이다. 플레이어 액터뿐 아니라 Npc도 ViewList를 관리해야 하는데, 그 이유는 ViewList에 플레이어가 들어온 경우 AI를 활성화시켜줘야 하기 때문이다.

### - Sector이동 로직 의사코드

나를 MyActor라고 하고, 현재 루프를 도는 Actor를 TargetActor라고 한다.

새로운 인접Sector들의 액터들을 NewActors, 이전의 인접Sector들의 액터들을 OldActors라고 한다.

<OldActors존재X, NewActors존재O>

```
// MyActor 갱신  
MyActor ViewList에 TargetActor 추가  
    MyActor가 Player면 TargetActor 생성 패킷  
    MyActor가 Npc이고, TargetActor가 Player이면 MyActor AI 활성화
```

```
// TargetActor 갱신  
TargetActor ViewList에 있으면  
    TargetActor가 Player면 MyActor 이동 패킷  
TargetActor ViewList에 없으면  
    TargetActor ViewList에 MyActor추가  
        TargetActor가 Player면 MyActor 생성 패킷  
        TargetActor가 Npc이고, MyActor가 Player이면 TargetActor AI 활성화
```

<OldActors존재O, NewActors존재X>

```
// MyActor 갱신
MyActor ViewList에 TargetActor 제거
    MyActor가 Player면 TargetActor 제거 패킷
    MyActor가 Npc이고, TargetActor가 Player이고, TargetActor가 마지막 Player면 MyActor AI 비활성화

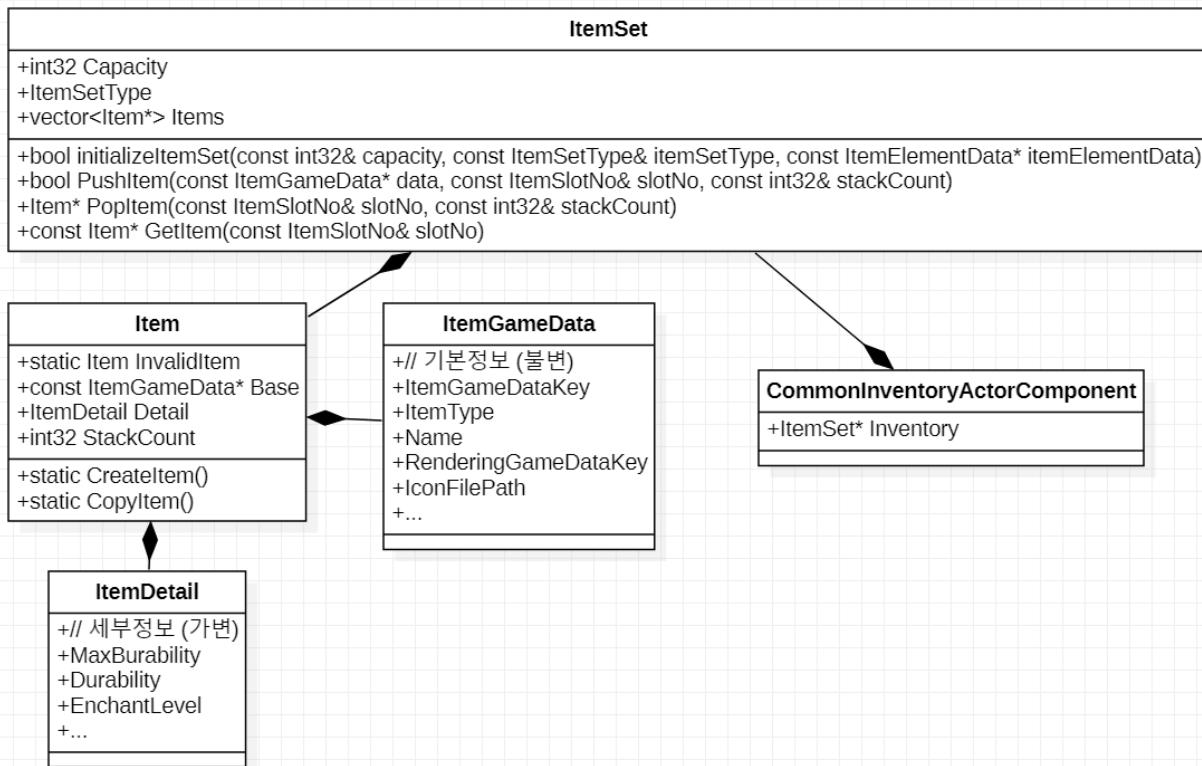
// TargetActor 갱신
TargetActor ViewList에 있으면
    TargetActor ViewList에 MyActor 제거
        TargetActor가 Player면 MyActor 제거 패킷
        TargetActor가 Npc이고, MyActor가 마지막 Player이면 TargetActor AI 비활성화
TargetActor ViewList에 없으면
    다른 스레드에서 이미 처리된 것이다.
```

## 4. 컨텐츠->Item

### - Item과 ItemSet

Item은 기본적으로 기본정보와 세부정보로 나누어져 있다. 기본정보는 ItemType과 RenderingGameData 등 불변정보이고, 세부정보는 Durability, EnchantLevel 등 가변정보를 담고 있다. 같은 아이템이라도 세부정보에 따라서 서로 다른 아이템이 될 수 있다.

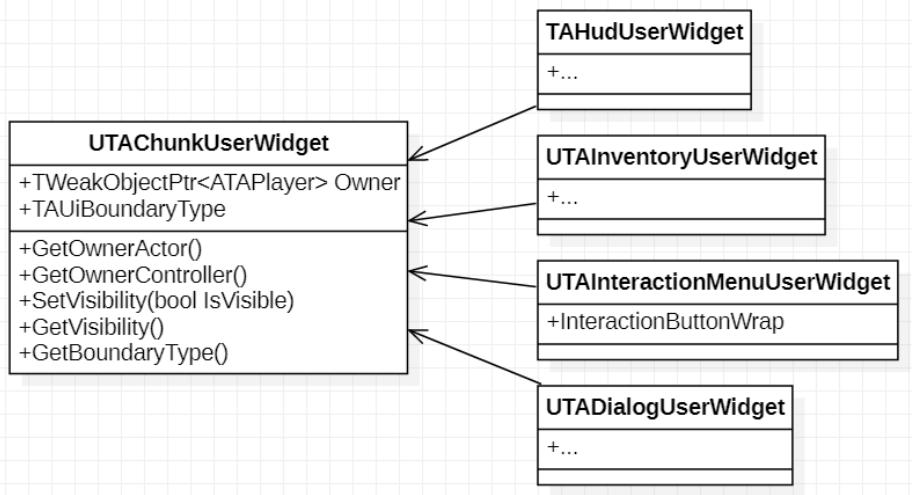
ItemSet은 Item들을 담고 있는 클래스이다. ItemsetType으로 ContainerType과 SlotType이 존재하는데, ContainerType은 ItemSet의 모든 ItemSlot에 담을 수 있는 ItemType의 제한이 없다. 그래서 다양한 Item을 담을 수 있어야 하는 Inventory나, 보물상자 등에 사용된다. SlotType은 특정 슬롯에 담을 수 있는 ItemType이 한정되어 있다. 그래서 장비 착용 슬롯을 구현하는데 적합하다.



## 4. 컨텐츠->UI

## - TACchunkUserWidget

UI 프로그래밍하다보면, 많은 코드에서 액터에 접근해야 할 필요가 있고, 꺼다 켰다 하는 작업이 매우 많아진다. 그때마다 그런 코드를 모두 작성하면 소스코드가 매우 복잡해질 것 같았다. 그렇기 때문에 그런 기능들을 모아둔 UserWidget을 하나 만들었다. Viewport에 바로 추가되는 최상위 UserWidget같은 경우 TACchunkUserWidget을 상속받아서 사용한다. 최상위 UserWidget 하위에 있는 ChildUserWidget의 경우는 최상위 UserWidget을 통해서 TACchunkUserWidget에 접근해서 필요기능을 사용하면 된다.



## 4. 컨텐츠->Interaction

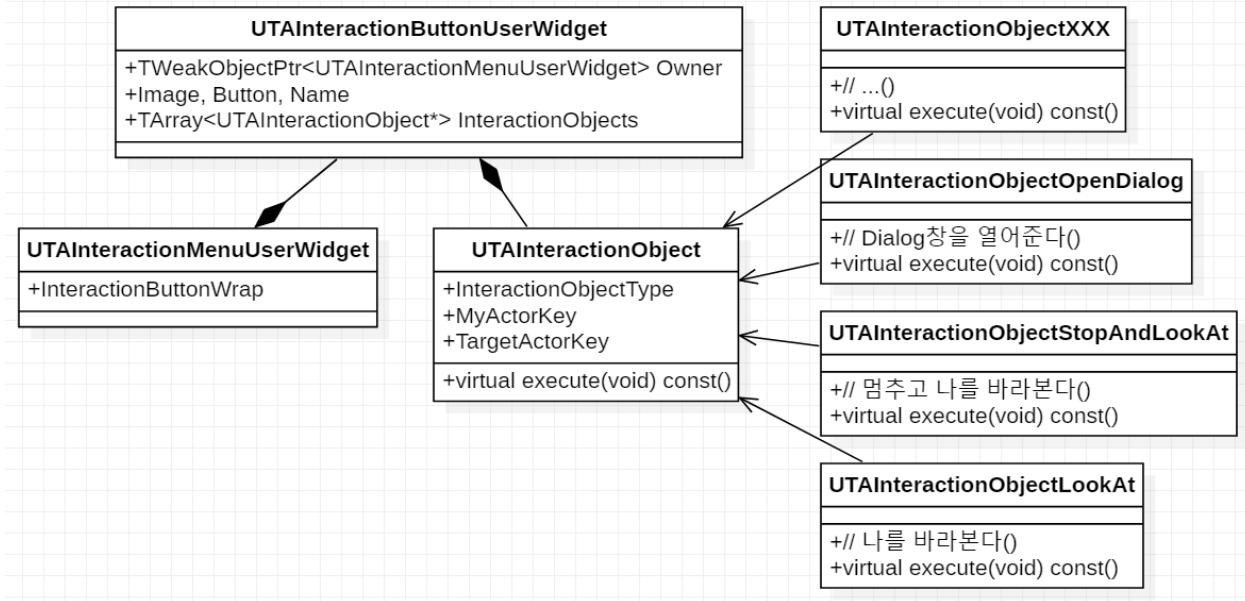
### - InteractionActor

인게임에서 Interaction가능한 Actor를 찾는 일은 TAGameInstance가 수행한다. TAGameInstance는 서버에서 내려준 Actor들을 UE4형식의 액터로 스폰해서 리스트로 보관하고 있다. TAGameInstance의 tick을 수행할 때 일정시간마다, 해당 Actor목록을 돌면서 Interaction가능 거리에 있는 가장 가까운 Actor를 InteractionActor로 설정한다.

### - InteractionButton과 InteractionObject

TAGameInstance에서 InteractionActor가 세팅되면 어떤 Interaction을 할 수 있는지 버튼으로 띄워서 보여줘야 한다. 어떤 버튼을 띄울지는 CharacterGameData에 InteractionType 리스트를 보고 결정한다. InteractionType 개수에 따라서 InteractionButton 개수가 결정된다.

그렇게 InteractionButton을 띄우고 나면 해당 버튼을 클릭했을 때(또는 바인딩된 키보드 키를 눌렀을 때), 그에 따른 동작을 해야 한다. 그래서 InteractionButton안에는 InterationObject라는 특정기능을 수행하는 객체들의 배열이 있다. 클릭 시 이 객체들을 Execute시켜주는 형식으로 Interaction의 실제 동작을 정의했다.



## - Interaction과 Camera 전환

말하는 Interaction을 수행 시 Dialog가 나오고, 카메라는 해당 Npc를 포커스 해야한다. 또 말하는 도중에는 플레이어의 키보드 입력을 막아야한다.

해당 Npc를 포커스하기 위해서 먼저 CameraMode를 FocusedAndControlBlocked로 변경 후, Camera를 Npc 액터기준으로 바라보는 방향으로 조금 이동하고 Npc액터를 바라보도록 설정한다. 그리고 CameraDirtyFlag를 켜준다. 그러면 Player Tick에서 보간작업이 이루어지면서 자연스럽게 전환된다.

키보드 입력에 따라서 플레이어가 움직이는 로직은 모두 MoveAndRotateCharacterByInput이라는 함수를 통해서 들어오는데, 현재 CameraMode가 FocusedAndControlBlocked로 설정되어 있으면 여기서 입력을 무시하면된다.

## 5. 기타->실수를 줄이기 위한 방법

### - TA\_ASSERT\_DEV, TA\_LOG\_DEV, TA\_COMPILE\_DEV

엔진에서는 4가지 유용한 개발을 도와주는 매크로가 있다.

**TA\_LOG\_DEV** : 간단히 문자열을 출력한다.

파라미터로 문자열로 넘겨주며, 문자열을 그대로 콘솔창에 출력해준다. UE4라면 UE4로그에 찍어준다.

**TA\_ASSERT\_DEV** : Expression을 확인해보고, 그 결과에 따라 로그출력과 Break를 걸어준다.

파라미터로 Expression과, 문자열을 넘겨주며, Expression이 false일 때, 문자열출력과 DebugBreak()를 수행한다. UE4라면 UE4로그에 Error 형식으로 로그를 찍는다. Expression이 true면 아무것도 해주지 않는다.

**TA\_COMPILE\_DEV** : Expression을 확인해보고, 그 결과에 따라서 컴파일 에러를 발생시킨다.

파라미터로 들어온 Expression의 결과가 false일 때, char msg[-1]와 같은 컴파일 에러를 발생시키는 코드를 생성한다.

**TA\_TEMP\_DEV** : 임시코드를 표시하기 위해서 사용하면 된다.

프로젝트가 커지다 보면 내가 임시코드를 어디에 작성했는지 잊어버리게 된다. 그때 **TA\_TEMP\_DEV**의 정의를

주석 처리하고 컴파일해보면 해당 위치를 알 수 있다.

## 5. 기타->빠른 개발을 위한 방법

### - 컴파일타임을 줄이기 위한 방법

#### - Pimpl패턴과 전방선언

프로그래밍을 하다보면, 사용하지도 않는 파일을 인클루드하는 경우가 많다. 그리고 그런 인클루드는 여러 해더에 가려져 있어서 눈에 잘 안보이는 경우가 많다. 그런 인클루드를 막기 위해서는 근본적으로 해더파일에서 다른 해더파일의 인클루드 수를 최대한 줄여야 한다.

하나 유용한 특징은 포인터형 레퍼런스형은 내부 구현을 사용하지 않는다면 전방선언만으로 인클루드 없이 컴파일 가능하다는 점이다. 그렇게 전방선언을 최대한 해서 해더파일의 인클루드를 줄인다.

Pimpl패턴 역시 포인터를 활용하는 방식이다. 일반적으로 클래스 멤버변수로 다른 클래스의 객체를 가질 때, 다음과 같이 선언한다.

```
#include "ClassName2.h"
"
ClassName1
{
...
...
Private:
    ClassName2 _className2Object;
}
```

여기서 ClassName2 \_className2Object;이 부분을 ClassName2\* \_className2Object;로 바꿔주고 ClassName1해더 앞에 ClassName2를 인클루드 하는 대신 class ClassName2;라고 선언해주는 것이다. 그렇게 하면 ClassName2해더파일을 인클루드하지 않고도 정의할 수 있고, 이는 컴파일타임을 줄이는데 도움을 준다.

### - 매크로 스위치문 활용

프로그래밍을 하다 보면 하기 싫어도 어쩔 수 없이 비슷한 코드를 작성해야 할 때가 있다. 예를 들어서 각 ActorComponentPool에서 ActorComponent를 얻어오는 코드다. ActorComponent타입이 추가될 때마다 해당 코드 블록을 계속 작성해야 하면, 귀찮아지고 앞 코드블록을 복사해서 쓰다가 실수가 생길 수도 있다. 다음과 같이 매크로로 코드블록을 작성해두면 편하게 매크로로 추가할 수 있다.

```

switch (componentType)
{
#define RETURN_COMPONENTS(Type, PoolName) \
    case ActorComponentType::Type: \
        { \
            return PoolName->getActorComponent<Server##Type##ActorComponent>(actorType, relativeGroupIndex); \
        } \
        break;

    RETURN_COMPONENTS(Move, _moveComponentPool)
    RETURN_COMPONENTS(Action, _actionComponentPool)
    RETURN_COMPONENTS(Ai, _aiComponentPool)
    RETURN_COMPONENTS(Character, _characterComponentPool)
    RETURN_COMPONENTS(Inventory, _inventoryComponentPool)
    RETURN_COMPONENTS(Object, _objectComponentPool)

#undef RETURN_COMPONENTS

    case ActorComponentType::Count:
    default:
        {
            TA_ASSERT_DEV(false, "다른 타입이 들어왔습니다.");
        }
}

TA_COMPILE_DEV(6 == static_cast<uint8>(ActorComponentType::Count), "여기도 추가해주세요");

```

### - 템플릿 적극 활용

템플릿을 활용하면, 모든 비슷한 코드를 하나하나 작성할 필요없이 쉽고, 정확하게 프로그래밍 할 수 있다. 위에 매크로 설명 경우와 비슷하게 비슷한 로직을 가진 클래스 객체들을 처리할 때 템플릿으로 처리하면 된다.

## 5. 기타->유용한 기능들

### - Enum <=> String 변환

프로그래밍을 하다 보면 Enum과 String을 서로 변환해야 할 경우가 있다. 각 경우를 하나씩 예로 들어보면 먼저 Enum -> String의 경우는 Serializer에서 데이터 처리에 대한 로그를 작성할 때, 데이터들이 각각 어떤 자료 형인지 로그파일에 기록해야 한다. 그때 TADebugType(Int8, Int16, Float ...)를 문자열("Int8", "Int16", "Float" ...)로 바꿔야한다.

String -> Enum의 경우는 ItemGameData를 xml파일에서 로드할 때 "ItemType::Consumable"와 같은 문자열을 ItemType::Consumable로 바꿔야한다.

이런 변환을 진행하기 위해서 추가해줘야 하는 것이 있는데 unordered\_map<string, EnumType> EnumTypeConverter이다. 다음과 같이 정의하면 나머지 ConvertEnumToString, ConvertStringToEnum 함수가 템플릿으로 자동으로 생성되게 코드를 작성해 두었다.

```

const std::unordered_map<std::string, GameWorldType> GameWorldTypeConverter
{
    {"RealWorld", GameWorldType::RealWorld}
    , {"TempWorld", GameWorldType::TempWorld}
};

TA_COMPILE_DEV(2 == static_cast<uint8>(GameWorldType::Count), "여기도 확인해주세요");

const std::unordered_map<std::string, ItemType> ItemTypeConverter
{
    { "All", ItemType::All }
    ,{ "Consumable", ItemType::Consumable }
    ,{ "RHand", ItemType::RHand }
    ,{ "LHand", ItemType::LHand }
};

TA_COMPILE_DEV(4 == static_cast<uint8>(ItemType::Count), "여기도 확인해주세요");

```

## - ToStringCast와 FromStringCast

String과 여러가지 자료형의 변환을 지원하는 템플릿 함수다. ostringstream과 istringstream을 사용하여 내부를 구현하고 있다.

## 6. 학습자료->문자열 관련 정리

개발환경이 Window 이기 때문에 그 기준으로 작성한다.

### 1. 아스키(ASCII) 코드

(American Standard Code for Information Interchange)의 약자

컴퓨터는 숫자(0과1)만 인식할 수 있기 때문에, 숫자와 문자를 대응시킨 것

영문 알파벳을 사용하는 대표적인 문자 인코딩

7bit 인코딩(128개의 문자 표현) + 1bit(Parity Bit)로 총 8bit로 사용.

총 128개 문자[출력 불가능한 제어 문자(33개) + 공백을 비롯한 출력 문자(95개)]

### 2. ANSI

영어만을 고려하여 만들어진 ASCII에서 다른 언어를 지원해야 할 필요가 생겨 만들어졌다.

ASCII 기반으로 만들어진 표준 규격으로 언어(CodePage)마다 코드표가 따로 존재한다.

(American National Standard Institute)의 약자

ASCII의 확장판 : 앞의 7bit(ASCII와 동일) + 1bit(CodePage) = 총 8bit로 사용.

여기서 CodePage란, 각 언어별로 code값을 주고 code마다 다른 문자열 표를 의미하도록 약속한 것이다.

CodePage는 CP949, EUC-KR 등 여러 가지가 있다.

영어만 사용하거나 ASCII를 사용할 경우 세계 어디에서나 사용에 문제가 없다.

영어 외 다른 언어를 사용할 경우 ANSI는 Code Page를 동일하게 맞춰야 한다.

Code Page가 다를 경우 의도와 다른 결과가 나올 수 있다.

대부분 ANSI Code Page들은 1바이트지만 아시아 몇 Code Page만 1-2바이트를 차지한다.

### 3. 멀티바이트(Multi-Byte)

위의 ASCII 문자는 8bit 즉, 1byte로 표현하는 문자이다.

그런데 ASCII 문자표를 보면 알 수 있듯이, 영어, 숫자, 기타 부호 등 128개 문자만 사용할 수 있는데 한글, 일본어와 같은 문자를 표현하기 위해서 1바이트 더 추가해서 총 2바이트로 문자 집합을 구성하여 사용할 수 있게 하였고 이를 ISO-2200에 정의하게 되었다.

여기서 주의해야 할 것은 2byte를 사용하는 것이 멀티바이트가 아니라, 문자 하나를 표현하는데 있어 다양한 바이트 수를 사용하는 방식이 멀티바이트인 것이다. 즉, 멀티바이트는 가변 문자열이라고 생각하면 된다.

영어와 같이 아스키 코드에 포함되는 문자 1byte

그 외 다른 언어처럼 아스키 코드에 포함되지 않는 문자 2byte처럼 다르게 할당해주는 방식이다.

특정 문자 집합마다 CodePage가 존재한다. 이런 의미에서 ANSI와 멀티바이트는 비슷한 개념이다.

#### 4. 유니코드(Unicode)

모든 문자(영어 포함)를 항상 2byte로 표현, 하지만 전 세계의 모든 문자를 2byte 문자로 1:1 매핑해 놓은 단일 코드표로 특정 인코딩 방식을 가리키는 것은 아니다. UTF(Unicode Transformation Format)는 유니코드 기반으로 적절히 조작해 byte를 문자로 변환하는 인코딩 방식이다.

그러나 보통, Unicode Encoding이라 하면 Windows, Java에서는 UTF-16을, 나머지 시스템에서는 UTF-8을 가리킨다. 혼용할 수 있으니 조심해야 한다.

인터넷, 파일 등 UTF-8형식을 많이 쓰는데 Windows는 UTF-16으로 지원한다.

#### 5. 정리, 추가 내용

\* 일반적인 경우 vs MS

<일반적인 경우>

유니코드 : UTF-8, -16, -32

<MS>

유니코드 : UTF-16

멀티바이트(ANSI) // MS에서는 A버전의 WindowsAPI를 사용 가능한 인코딩을 의미한다.

// 개념적으로 보면 UTF-8도 멀티바이트지만 MS에선 지원하지 않는다.

- CodePage들 // ex) CP949, EUC-KR

\* UTF-8같은 문자열의 저장되는 방식

문자에 따라서 사용하는 바이트 수가 다르기 때문에, 몇 개의 바이트를 사용할지 결정하는 로직은 간단하게 최상위 비트가 설정되어 있으면 다음 바이트도 사용 이런 식으로 되어있다.

\* SBCS, MBCS, WBCS

SBCS(Single Byte Character Set): 한 문자를 1byte로 표현하는 방식 ( ex) ASCII )

MBCS(Multi Byte Character Set): 한 문자를 다양한 byte로 표현하는 방식, SBCS 방식으로 표현할 수 없는

언어를 표현하기 위해 ( ex) CP949, EUC-KR )

WBCS(Wide Byte Character Set): 한 문자를 2byte로 표현하는 방식, UNICODE(UTF-16 = WBCS) ( ex )  
wchar\_t, L"hello" )

SBCS -> ASCII 코드

MBCS -> ASCII 코드와 Unicode를 혼용하여 사용하는 방식

WBCS -> Unicode

\* MS의 유니코드인 UTF-16으로 변환하는 방법

ANSI/UTF-8 to UNICODE(MS에서 UTF-16) => MultiByteToWideChar

UNICODE(MS에서 UTF-16) to ANSI/UTF-8 => WideCharToMultiByte

\* MS에서 UTF-8을 사용하기 위한 방법

// A버전 함수도 아래와 비슷한 로직으로 되어있다.

1. MultiByteToWideChar를 사용해서 UTF-8 > UTF-16으로 변환

2. W버전의 함수 호출 + 추가 작업

3. 최종적으로 WideCharToMultiByte호출해서 UTF-16 > UTF-8로 변환

\* ANSI or UNICODE(UTF-16 = WBCS) 지원

ANSI를 쓸 이유가 없다. 무조건 UNICODE(UTF-16 = WBCS)을 사용하고, 일반적인 문자 인코딩을 지원할 수 없기 때문에 A버전은 필요 없다.

MS에 T를 붙인 ANSI(MBCS) / UNICODE(UTF-16 = WBCS)를 지원하는 여러가지 형식의 것들(TCHAR 등)이 존재하는데, 명시적으로 쓰는게 좋을 것 같다.

// <https://stackoverflow.com/questions/234365/is-tchar-still-relevant/3002494#3002494>

## 6. 학습자료->UE4

\*\* 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 null로 초기화 된다.

다른 곳에서 AActor 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 UActorComponent 또는 AActor 레퍼런스에만 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 TArray 또는 다른 언리얼 컨테이너 클래스에 저장해야 한다. 이렇게 하면 일방적으로 삭제되고 허상포인터로 남지 않고 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++
```

### \* 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 때문에 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 접근 가능

### \* 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 고려하지 않은 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 Pawn을 통해서(AddMovementInput) 한다.

#### \* Pawn, Controller 기타 정보

##### - Pawn

bUseControllerRotationYaw : 컨트롤러 회전의 Yaw회전 값과 폰의 Yaw회전값은 기본적으로 연동되어 있는데, 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)

편집 가능

- (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 Protected)Access = true) : private 나 protected 변수를 외부에 노출

#### \* Pawn vs Controller

Controller는 플레이어의 의지를 전달한다. 게임의 물리적인 요소를 고려하지 않은 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 감지하는 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 발생 가능

Hit 이벤트 : 블록 반응에서 이벤트가 발생한다.

Query : 두 물체의 충돌 영역이 서로 겹치는지 테스트하는 설정, 충돌 영역의 겹침을 감지하는 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 생성해야 한다.

#### \*\* 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 빌드 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 다른
2. 프로젝트명.Target.cs, 프로젝트명Editor.Target.cs에 해당 모듈 추가해야 dll 생성한다.
3. 기본적으로 Public(외부 모듈에 공개할 파일)폴더에 헤더, 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 뒀어서 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.

```
#pragma once  
① #include "MyUObject.generated.h"  
② UCLASS()  
③ class UMyUObject : public UObject  
{  
    GENERATED_BODY() ⑤  
};
```

1. 클래스이름.generated.h 를 반드시 가장 마지막에 include 시켜주어야 합니다.
2. 클래스 선언 전에 UCLASS 매크로를 사용해야 합니다.
3. 언리얼 오브젝트의 접두사는 U, A 그리고 S 가 있습니다. 액터 기반이라면 A 를 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.

```
#pragma once  
  
#include "MyUObject.generated.h"  
① UCLASS()  
② class MYGAME_API UMyUObject : public UObject  
{  
    GENERATED_BODY()  
  
    public:  
        UMyUObject(); ②  
        UPROPERTY()  
        FString MyName; ③  
  
        UFUNCTION()  
        FString GetName(); ④  
};
```

1. 모듈이름\_API 매크로는 상황에 따라 부가 설정을 해주는 매크로입니다. 예를 들어 윈도우 에디터에서 사용할 수 있게 dll로 빌드할 때에는 이 매크로는 MS 의 DLL 문법인 \_\_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.