|  |  |
| --- | --- |
| **Project TA(TargetActor)** | |
| **기능** | **개발자** |
| **멀티 플레이 엔진 구조 설계** | **황정인** |
| **네트워크 통신 개발** |
| **파일 입출력 개발** |
| **컨텐츠 개발** |
| **게임관련 데이터 추가** |
| **추가 정보** | |
| **Github :** [**https://github.com/HwangJeongIn/TargetActorGame/tree/AllInOne**](https://github.com/HwangJeongIn/TargetActorGame/tree/AllInOne)  **// 해당 저장소의 Project TargetActor 설명.txt 텍스트 파일에 파일 위치, 특징에 대해 설명하고 있다.**  **1차 게임 엔진 시연 영상 :** [**https://youtu.be/HPNiLHJIKCY**](https://youtu.be/HPNiLHJIKCY) | |
| **<목차>**  **1. 개요**  **1.1. 시스템 구조**  **- 주요 시스템 클래스**  **- App, ActorManager, ActorSystemManager, SpawnDataManager, GameDataManager**  **- 시스템의 생성과 해제**  **- initialize, open, close**  **- 공통 코드와 서버, 클라이언트 코드**  **- CommonXXX, ServerXXX, ClientXXX**  **1.2. ECS(Entity, Component, System) 모델**  **- Actor, ActorComponent, ActorSystem**  **- 오브젝트 풀**  **- ActorPool, ActorComponentPool**  **- 풀 객체 생명 주기**  **- onFromPool, onActive, onToPool**  **1.3. 멀티스레드 기반 구조**  **- 공유 자원과 동기화 기법**  **- SRW Lock**  **- ScopedLock 지원**  **- 병렬 로드**  **- 게임관련 데이터 로드 과정**  **- 컨텐츠 이벤트 관리**  **- 이벤트 큐, 이벤트 처리 과정**  **2. 네트워크 통신 모델**  **2.1. IOCP와 Overlapped I/O**  **- IOCP**  **- Overlapped I/O**  **- IOCP + Overlapped I/O**  **2.2. RPC 통신**  **- RPC의 사용 이유와 동작 방식**  **- IDL 파일과 IdlCompiler**  **- 간단한 원격함수 호출 예시**  **3. 게임관련 데이터**  **3.1. 데이터 형식**  **- Xml**  **- Binary**  **- 직렬화 도구 지원(MemoryBuffer와 Serializer)**  **3.2. 데이터 에디팅 툴**  **- Excel**  **- 에디팅 방식**  **- 추출 방식**  **- UE4 에디터**  **- 에디팅 방식**  **- 추출 방식**  **3.3. 게임관련 데이터 종류**  **3.3.1. GameData**  **- 에디팅 방식**  **- 로드 과정**  **3.3.2. PathPoint**  **- 에디팅 방식**  **- 로드 과정**  **- 동작 방식**  **3.3.3. SpawnData**  **- 에디팅 방식**  **- 로드 과정**  **- 동작 방식**  **3.3.4. Navigation**  **- UE4의 Recast/Detour Navigation, RecastNavMesh 추출 과정**  **3.3.5. Assets**  **- 로드 과정 동적 비동기 로드**  **- 로드조건**  **- 데이터 검증**  **4. 세부 기술**  **4.1. 게임 엔진과 UE4(클라이언트)과의 통신**  **- TAGameEvent**  **4.2. AI**  **- 행동 트리(BehaviorTree)**  **- AiBTNode, AiBTNodeComposite, AiBTNodeCondition, AiBTNodeExecution**  **4.3. Move**  **- Sector의 개념과 필요성**  **- Sector의 관리**  **- Sector 이동 로직 의사코드**  **4.4. Item**  **- Item 구조와 ItemSet**  **4.5. UI**  **- TAChunkUserWidget**  **4.6. Interaction**  **- 상호작용 가능한 액터 탐색**  **- InteractionButton과 InteractionObject**  **- 상호작용과 카메라 전환**  **5. 기타**  **5.1. 실수를 줄이기 위한 방법**  **- TA\_ASSERT\_DEV, TA\_LOG\_DEV, TA\_COMPILE\_DEV**  **5.2. 빠른 개발을 위한 방법**  **- 컴파일타임을 줄이기 위한 방법**  **- Pimpl패턴과 전방선언**  **- 매크로 스위치문 활용**  **- 템플릿 활용**  **5.3. 그외 기능들**  **- Enum <=> String 변환**  **- ToStringCast와 FromStringCast**  **6. 참고 자료** | |
| **1. 개요** | |
| **1.1. 시스템 구조** | |
| **- 주요 시스템 클래스**  뒤에서 세부 내용을 다룰 예정인 시스템은 간단한 구조를 소개하는 정도로 서술한다.  **\* App**  엔진에서 가장 최상위에 있는 시스템 클래스이다. 여기서 엔진에 존재하는 모든 시스템, 모든 풀, 각종 데이터 들이 할당되고 초기화가 일어난다. 또한 프로세스가 끝나면 모든 할당했던 메모리들을 해제시키는 출발점이다.  간단히 각 매니저들에 대해서 소개하면 다음과 같다.  - ActorEventTimer : 액터관련 이벤트를 관리한다.  - XXXActorManager : 액터풀과 컴포넌트풀, 섹터들을 관리한다.  - XXXActorSystemManager : 액터시스템들을 관리한다.  - XXXSpawnDataManager : 스폰관련 정보를 관리한다.  - GameDataManager : 게임데이터 관련 정보를 관리한다.  - IOCP : Overlapped I/O와 함께 통신, 이벤트 처리에 사용된다.    **\* ActorManager**  액터와 액터컴포넌트, 섹터들을 관리하는 역할을 하는 관리 클래스이다. 전반적인 풀을 관리하는 클래스인 ActorDataPool과 공간 분할 클래스인 Sectors를 멤버로 가지고 있다.  ActorDataPool는 액터 풀 클래스인 ActorPool과 액터 컴포넌트 풀 클래스인 ActorComponentPool를 가지고 있다. 액터와 액터 컴포넌트는 풀 객체에서 객체를 대여, 반납하면서 사용한다. ActorPool에 객체로 존재하는 Actor들은 기본적으로 ActorComponent들을 묶어주는 의미로 사용되며 ActorKey와 ActorType, 플레이어인 경우 네트워크 통신을 위한 Socket을 가지고 있다. ActorComponentPool에 존재하는 ActorComponent들은 기능을 수행하지는 않고 데이터로서 의미를 지닌다.  Sectors에는 액터가 이동할 수 있는 Sector들과 이를 관리하는 기능으로 구성되어 있다. Sector를 통해서 플레이어는 해당 장소에 어떤 액터가 있는지 조회할 수 있고, 반대로 해당 Sector에 존재하는 다른 액터에게 본인을 통보할 수 있는 기능도 존재한다.    **\* ActorSystemManager**  액터 컴포넌트에 접근하여 액터의 각종 기능을 수행하는 액터 시스템들을 관리하는 클래스이다. ActorSystems를 멤버로 가지고 있다.    **\* SpawnDataManager**  파일에서 정보를 읽어 지정된 위치에 액터를 생성하는 작업을 수행한다. SpawnData들을 파일에서 로드해서 가지고 있다.    **\* GameDataManager**  각 컨텐츠에서 필요한 게임 관련 데이터(이동속도, 공격력 등)들을 관리하는 클래스이다. GameData들을 파일에서 로드해서 멤버로 가지고 있으며, 각 컨텐츠에서 특정 GameData가 필요할 때 제공해준다.    **- 시스템의 생성과 해제**  여러 기능을 수행하는 시스템 클래스들이 일관성 있는 할당, 해제 과정을 지원하기 위해 크게 3가지의 공통 함수를 갖도록 설계하였다. 생성은 Initialize와 open으로 2단계로 구성되고 해제는 close를 통해 수행된다.  **\* initialize와 open**  생성 단계에서 Initialize는 메모리를 할당하고, open은 할당된 메모리를 추가로 초기화하는 역할을 한다.  간단하게 생성자에서 처리하지 않고 이러한 단계적 과정을 통해 생성하는 이유는 크게 세 가지이다.  1. 서버와 클라이언트 시스템에서 공통적인 코드를 처리할 때 순서가 중요한데 한 단계 과정으로 이를 처리하기 어렵다.  2. 복잡한 과정을 간단한 단계로 나눠 가독성을 높인다.  3. 과정에서 문제가 생겼을 경우 적절하게 에러 처리를 할 수 있다.  처리 순서가 문제가 되는 경우를 간단히 예로 들어보겠다.  엔진에는 액터의 풀 클래스인 ActorDataPool가 존재하고 이를 상속받은 ServerActorDataPool과 ClientActorDataPool를 사용하고 있다. 액터 할당은 상속받은 각 클래스에서 적절하게 ServerActor, ClientActor로 해주고 공통 처리 코드는 ActorDataPool에서 해준다. 생성자에서 처리하려고 하면 ServerActor, ClientActor가 생성되기 전에 ActorDataPool에서 공통 처리를 수행하게 된다. 제대로 할당되지도 않은 상황 때문에 문제가 된다. 만약 initialize와 open으로 나눈다면 ServerActorDataPool, ClientActorDataPool에서 initialize에서 각각 할당해주고 ActorDataPool의 open에서 공통 처리를 수행하면 끝이다.  **\* close**  close는 프로세스가 끝날 때, 할당했던 메모리들을 해제하는 작업을 수행한다. 순서는 작은 볼륨의 클래스에서 큰 볼륨의 클래스로 수행하고(예를 들어 ActorManager보다는 ActorDataPool의 close가 먼저 불린다.), 다른 곳에서 참조되는 클래스일 경우, 참조하고 있는 클래스에서 close가 호출된 이후에 수행하도록 작성되어 있다.    **- 공통 코드와 서버, 클라이언트 코드**  **\* CommonXXX, ServerXXX, ClientXXX**  서버와 클라이언트에서 공통적으로 사용하는 코드도 있고 각 위치에서만 사용해야하는 코드도 존재한다.  이를 분리하기 위해 공통 코드에는 Common 접두사를 붙여서 사용하고 서버에서만 사용되는 코드는 Server접두사 클라이언트에서만 사용되는 코드는 Client접두사를 붙여서 Common 코드를 상속받아 사용하는 것으로 정했다. 이러한 상속구조를 통해 중복 코드를 최대한 없애고, 유지보수성을 높이며, 코드 가독성을 높이는 효과를 볼 수 있었다. | |
| **1.2. ECS(Entity Component System) 모델** | |
| **- Actor, ActorComponent, ActorSystem**  엔진에서는 일반적인 객체 지향 모델이 아닌 ECS 모델을 사용하고 있다. ECS 모델은 크게 세 부분으로 나뉜다.  1. **Entity**(Actor)는 Component의 컨테이너로서 의미를 가진다.  2. **Comonpent**(ActorComponent)는 데이터를 제공한다.  3. **System**(ActorSystem)는 Component를 가지고 실질적인 기능을 제공한다.  간단한 동작 방식을 설명하자면 다음과 같다.  Actor는 ActorKey(객체 식별자)를 가지고 있어 통해 객체를 식별할 수 있고 ActorType를 가지고 있어서 이에 따라 해당 Actor가 어떤 ActorComponent를 보유하고 있는 지 확인할 수 있다. 필요에 따라서 ActorKey를 가지고 해당 Actor의 ActorComponent를 얻어서(ActorComponentPool에서 요청해야한다.) ActorSystem에서 실질적인 기능을 처리하게 된다.  이런 식의 구조를 선택한 이유를 나열해 보면 다음과 같다.  1. 일단 구조 자체가 단순해지고, 데이터와 기능을 분리하였기 때문에 가독성이 좋아진다. 또한 데이터를 따로 풀로 관리하여 데이터가 선형적으로 저장되기 때문에 컴포넌트들을 모아서 처리(Update 등)할 때 데이터 지역성의 이점을 누릴 수 있다.  // Unity에서는 여러 기능을 포함하고 있는 컴포넌트인 MonoBehaviour에서 기능을 분리함으로써 성능 향상을 봤다는 내용도 있다.  2. 기존 상속구조에서 벗어나서 자유롭게 Entity를 정의할 수 있다. 일반적인 객체 지향 모델에서는 프로젝트가 커질수록 상속구조가 복잡해지고 계층구조를 관리하기 어려워진다. 하지만 ECS 모델에서는 컴포넌트 기반으로 프로그램이 작성되기 때문에 확장성이 유지 보수성이 좋다.  3. 한 시스템 단위(보통 하나의 시스템은 1 - 3개의 컴포넌트를 담당)를 스레드로 나눠서 처리할 수 있다.    하지만 단점도 존재한다. ECS 구조의 컴포넌트 기반이라는 특징은 계층 구조에 대한 직관성이 떨어지고, 컴포넌트 간 복잡한 상호작용이 많이 일어날 수 있다. 이러한 단점에도 불구하고 다른 장점들이 더 매력적이라고 생각해서 선택하게 되었다.  **- 오브젝트 풀**  오브젝트 풀은 객체 사용 시 할당, 해제 과정 없이 이미 초기화된 풀에서 대여, 반납할 수 있도록 해주는 디자인 패턴이다. 이 패턴의 장점은 할당, 해제 작업으로 인한 성능 저하, 메모리 단편화를 방지하고 연속적인 메모리를 할당함으로써 데이터 지역성의 이점을 누릴 수 있다는 점이다.  엔진에서 ActorPool과 ActorComponentPool은 대여, 반납 기능을 기본적으로 제공하고 있다. 또한 ActorType에 따라 풀을 분할하여 사용하고 있다.  **\* ActorPool**  ActorPool은 Actor를 풀의 형식으로 저장하고 있다. Actor에는 ActorType(Player, Npc, Object)이라는 해당 Actor의 종류를 나타내는 정보가 존재하는데, 이 정보에 따라 저장되는 풀 내부에서의 위치가 결정된다. 예를 들어 Player는 0~199번, Npc는 200번~399번, Ojbect는 400~599번 이런 식으로 결정되어 있다. 이를 통해 특정 ActorType에만 수행해야 할 것들을 별도의 검색 없이 수행할 수 있다.  **\* ActorComponentPool**  ActorComponentPool도 ActorPool처럼 ActorType에 따라서 사용할 수 있는 풀의 범위가 다르지만, ActorPool과 다르게 특정 ActorType에서 사용하지 않는 컴포넌트가 있다. 이를 구분하기 위해 ActorType에 따라 사용하는 풀의 범위를 알 수 있는 StartIndexMap이라는 멤버가 존재한다. 이 멤버를 통해서 어떤 ActorType이 해당 컴포넌트를 사용하고 있는지 알 수 있으며, 어느 범위의 ActorComponentPool을 사용하는지 알 수 있다.    **\* 풀 객체 생명 주기**  **@ onFromPool**  풀에서 객체를 꺼내 쓰기 위해서는 몇 단계의 과정이 필요하다. 일단, 사용할 수 있는 풀 인덱스를 찾고(FreeIndexes에 저장되어 있다), 해당 풀에서 Actor와 ActorComponent가 사용 예정이라는 것을 마킹(PoolValueState::Initializing으로 초기화)하는 것부터 시작한다. 이 시점에 onFromPool함수가 각 ActorComponent에서 호출된다.  **@ onActive**  아직 초기화 되지 않는 단계이므로 initializeActorComponent함수를 통해서 Actor의 ActorComponent들의 모든 데이터들을 초기화를 진행한다. 이 후 onActive함수가 ActorComponent에서 호출된다. 그리고 최종적으로 Actor와 ActorComponent에 현재 사용하고 있다고 마킹(PoolValueState::Using으로 초기화)한다. 이렇게 함수를 2개로 나눈 이유는 Actor의 ActorComponent가 초기화 단계에서 필요한 작업과 초기화가 끝난 상태이면서 사용 전에 필요한 작업이 따로 있기 때문이다.  **@ onToPool**  onToPool함수는 객체가 풀에 반납될 때 호출되고, 이 후 다시 꺼내 사용할 수 있도록 FreeIndexes에 다시 등록해준다. 또한 풀 객체를 사용할 수 있게 마킹(PoolValueState::Waiting으로 초기화)한다. | |
| **1.3. 멀티스레드 기반 구조** | |
| 현재는 로딩 없는 오픈 월드 형식의 MMORPG 게임을 만들 목적으로 만든 게임 엔진이기 때문에 싱글프로세스, 멀티스레드 구조로 동작한다. 채널이동과 월드이동이 존재하지 않는다.  **- 공유 자원과 동기화 기법**  멀티스레드 프로그래밍에서는 공유 자원에 대한 읽고 쓰기가 자유롭지 않다. 상황에 따라서 스레드에서 안전하게 접근할 수 있도록 동기화 기법을 적절하게 사용해야 한다.  **\* SRW Lock**  유저모드 기반 동기화 기법 중 SRW Lock라는 도구가 존재한다. SRW Lock은 성능이 뛰어나고, 두 가지 방식(Shared, Exclusive)으로 동기화 기법을 제공하고 있다. 특히 공유 자원이 많이 변경되지는 않지만, 조회가 많이 이루어지는 곳에서 Shared모드는 유용하게 사용될 수 있다.  예를 하나 들어보면 섹터에서는 액터가 많이 움직이지 않으면, 섹터안에 있는 액터의 리스트가 변하진 않지만, 안에 있는 액터가 움직일 때 해당 섹터에 있는 액터리스트를 모두 조회하면서 내가 움직이고 있다는 것을 알려야한다. 액터리스트 조회 시 Shared 모드로 락을 걸어 사용했을 때 모든 경로에서 해당 액터의 리스트를 조회만 하고 있는 상황에서는 모두 블로킹 없이 처리할 수 있다.  엔진에서는 이를 추상화하고 인터페이스를 통해 쉽게 사용할 수 있도록 SRW Lock의 래퍼 클래스인 Lockable를 제공하고 있으며 이를 상속받은 LockableObject를 락 객체로 사용하고 있다.  **\* ScopedLock 지원**  락을 걸고 풀기 위해 Acquire, Release를 페어로 호출해줘야 한다. 만약 Release가 누락되거나 Acquire를 연속으로 호출하게 되면 쉽게 데드락 상태에 빠지게 된다. 이는 개발을 진행하면서 코드가 길어지면서 충분히 발생할 수 있는 상황이다.  이를 방지하기 위해서 엔진에서는 해당 스코프안에서 안전하게 락을 걸 수 있는 도구인 ScopedLock 클래스를 제공하고 있다. 이 클래스는 생성자에서 Shared모드나 Exclusive모드로 Acquire해주고 소멸자에서는 Release 함으로써 직접 Acquire, Release 호출하면서 발생하는 실수를 방지해준다.  **- 병렬 로드**  게임과 관련된 로드해야 할 데이터는 다양하고 개수도 많다. 이를 싱글 스레드에서 처리하고자 한다면 시간이 너무 오래 걸린다. 멀티스레드 환경에서 효율적으로 로드를 수행하기 위해서는 데이터 경쟁 상태를 최소화하고 (완전히 독립적이면 더 좋다.), 적절한 스레드 수를 사용해야한다. 엔진에서는 일정한 스레드 수에서 효율적인 병렬 로드를 지원하기 위해 병렬 로드 시스템 클래스인 ThreadLoadTaskManager를 제공하고 있다.  **\* 게임관련 데이터 로드 과정**  먼저 데이터를 로드하기 위해서 먼저 ThreadLoadTaskManager에 태스크를 등록하여야 한다. 태스크 객체 클래스인 ThreadLoadTaskObject를 사용하여 ThreadLoadTaskManager에 태스크를 등록할 수 있는데, 이 객체는 프로시저를 호출할 객체, 함수호출 시 사용되는 인자들을 멤버로 가지고 있다. 처리할 태스크를 등록 후 ThreadLoadTaskManager에서 로드를 시작하면 등록한 태스크들에 대해 스레드가 적절히 생성되어 태스크를 하나씩 처리한다.  파일이 수십 수백 개가 존재하고 그 파일 개수만큼 스레드를 생성해서 한꺼번에 처리한다면, 코어수가 아무리 많은 CPU라도 스레드 수를 감당하지 못할 것이다. 이를 방지하기 위해서 적절한 수의 스레드로 등록된 태스크들을 FCFS 방식으로 스케줄링하여 처리한다.  로드 과정을 그림으로 나타내면 다음과 같다.    **- 컨텐츠 이벤트 관리**  게임 진행 중에는 이벤트를 발생시킬 필요가 있다. 예를 들어서 몬스터가 사망 시에는 일정 시간 후 SpawnDataManager가 다시 스폰하도록 해야 한다. ActorEventTimer는 이런 액터의 이벤트들을 관리하는 시스템 클래스이다.  **\* 이벤트 큐, 이벤트 처리 과정**  이벤트를 처리하기 위해 스레드 하나(타이머 스레드)를 할당해서 일정시간마다 이벤트 큐(시간에 따라 정렬된 우선순위 큐)에 지금 수행되어야 하는 이벤트가 있는 지 체크한다. 만약에 등록된 이벤트 중에 수행되어야 할 이벤트가 있다면 해당 이벤트를 직접 수행하지는 않고, 나머지 워커 스레드들에게 PostQueuedCompletionStatus를 사용해서 다른 스레드가 병렬처리 할 수 있도록 위임한다. 직접 수행하지 않는 이유는 타이머 스레드에서 병목 현상을 방지하기 위함이다. | |
| **2. 네트워크 통신 모델** | |
| **2.1. IOCP와 Overlapped I/O** | |
| **- IOCP**  디바이스의 입출력 완료를 통보하기 위한 포트이다. 빠른 입출력 통보, 최적화된 스레드풀링 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 통보해주기 때문에 계속해서 완료 여부를 지속적으로 확인할 필요 없다.  간단한 사용 예시를 들어보면 소켓을 IOCP에 연결해두면 소켓의 수신 작업이 완료되면, GetQueuedCompletionStatus를 통해 대기하고 있는 스레드를 깨워준다. 해당 스레드는 수신된 정보를 가지고 패킷을 처리하면 된다.  Windows 기반 플랫폼에서만 사용 가능하고 프로그램 구현이 복잡해지며 공유 메모리에 대한 동기화를 적절히 처리해야 한다는 주의 사항이 존재한다.  **- Overlapped I/O**  Overlapped I/O는 비동기 I/O에 사용되는 Windows API이다. 진행 중인 상태 현황을 보관하는 구조체와 수신된 데이터를 받을 데이터 블록이 필요하며 이 두 데이터는 Overlapped을 걸어두었을 때 운영체제가 백그라운드에서 접근하기 때문에 비동기로 수행하는 일이 완료될 때까지 유지되어야 한다.  간단한 동작 방식은 소켓에 대해서 Overlapped 액세스를 걸어주고, Overlapped 액세스가 성공했는지 확인한 후 성공했으면 결과값을 얻어와서 나머지를 처리하는 식으로 동작한다.  논블록킹 소켓이랑 비교했을 때, 코드가 깔끔해지고, 불필요한 시도를 계속 하지 않는다. (계속해서 send, receive를 호출하지 않고, 결과만 확인한다.) 하지만 결론적으로 send, receive를 호출하지 않지만 GetOverlappedResult를 계속 호출하면서 결과를 확인해야 하기 때문에 한계가 있다.  **- IOCP + Overlapped I/O**  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 워커 스레드에서 GetQueuedCompletionStatus로 대기해준다.  클라이언트가 접속하면 대기하고 있던 워커 스레드 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 워커 스레드 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, 동적할당으로 Overlapped에 필요한 객체들을 할당하고, GetQueuedCompletionStatus을 통해서 완료가 통보되면 할당한 객체들을 삭제한다. | |
| **2. RPC 통신** | |
| **- RPC의 사용 이유와 동작 방식**  일반적으로 서버와 클라이언트가 통신을 하기 위해서는 목적에 따라 각각 패킷객체를 정의해야하고 한쪽에서 그 패킷객체를 직렬화하여 보내면 다른 쪽에서 역직렬화하여 패킷객체를 읽어서 그에 따라 처리를 해줘야 한다. 이 방식을 사용하였을 때 패킷객체는 필요할 때마다 수동으로 만들어야 하고, 직렬화 / 역직렬화 코드를 계속해서 작성해야한다.  엔진에서는 이를 단순화하기 위해 원격함수호출(RPC, Remote Procedure Call)통신을 지원하고 있다. 이를 사용하면 일반 함수를 호출하듯 원격함수를 호출하면 원격에 있는 프로세스에서 함수가 호출된다. 사용자는 IDL파일에 해당함수를 정의하고 컴파일 후, 호출하려는 원격 프로세스의 함수만 구현하면 되기 때문에 매우 간편하다.  일반적으로 한 프로세스에서 다른 원격 프로세스의 함수를 호출할 수 없다. 이런 일이 가능해 보이는 이유는 함수 호출 사이에 일반적인 패킷처리 로직 코드가 자동으로 생성되기 때문이다. 엔진에서는 IdlCompiler가 IDL 파일을 읽어 내부 패킷 처리 코드를 자동 생성하도록 구현되어 있다. 이 덕분에 사용자는 패킷 처리 로직을 신경쓰지 않아도 된다.  **- IDL 파일과 IdlCompiler**  엔진에서 RPC통신을 지원하기 위해 호출 규약을 정의하는 IDL(Interface Definition Language)파일이 존재한다. 이 파일에는 함수명, 인자 등이 포함되어 있으며, 엔진에서는 이를 컴파일할 수 있는 IdlCompiler를 제공하고 있다. IDL 파일들은 IdlCompiler에 의해 컴파일되고, 파일들을(Stub코드) 생성하게 된다.  IDL 파일의 예시는 다음과 같다.    이를 통해 인자들을 가지고 있는 패킷 객체, 원격에 존재하는 프로세스의 함수(기능 구현은 직접 해야한다.), 원격함수, 패킷 객체에 따라 적절히 함수를 호출해주는 코드가 생성된다. 여기서 원격에 존재하는 프로세스의 함수만 구현해서 사용하면 된다.    **- 간단한 원격함수 호출 예시**  간단히 클라이언트에서 서버의 MoveActor라는 원격함수를 호출한다고 가정한다.    \* 백그라운드 이미지 출처 : greeks for greeks | |
| **3. 게임관련 데이터** | |
| **3.1. 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 때문 개발 단계에서 유용하게 사용될 수 있다. (추후 성능을 위해 바이너리 파일로 추출하여 사용할 듯하다.)  Xml파일을 읽어 프로그램 내 객체(엔진에서 XmlNode 클래스 객체)로 만드는 과정은 다음과 같은 단계로 이루어진다.  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 AttributeName과 AttributeValue를 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 데이터의 특성에 맞게 파싱하면 된다..    **- Binary**  디버깅은 힘들지만 성능을 위해 가볍고 빠른 바이너리 데이터를 로드, 세이브 할 수 있는 기능을 엔진에서 지원하고 있다. Read, Write를 하기 위한 MemoryBuffer와 Serializer가 존재한다.  **\* 직렬화 도구 지원(MemoryBuffer와 Serializer)**  바이너리 파일의 직렬화 수행을 위해 MemoryBuffer와 Serializer 같은 직렬화 도구를 지원하고 있다.  MemoryBuffer는 문자 그대로 데이터를 담는 버퍼의 역할을 한다. 용량, 현재 위치 등과 같은 정보를 가지고 있다. 만약 사용 중에 용량을 초과하게 되면 자동으로 버퍼를 키워준다. 독립적으로 사용할 수 있고, 직렬화 로직을 사용하기 위해 Serializer와 함께 사용될 수 있다. 파일을 바이너리 또는 문자열 형식으로 로드하거나 세이브할 수 있는 기능을 지원한다.  Serializer는 MemoryBuffer에 직렬화, 역직렬화를 수월하게 수행하기 위해 지원하고 있다. Operator<< 를 여러 타입에 대해서 오버로딩 하고 있으며, 최종적으로 MemoryBuffer에 저장한다. 모드에 따라서 직렬화(Write 모드), 역직렬화(Read 모드) 가능하다.Read, Write, WriteLog 등의 직렬화 모드를 지원하고 있으며 이러한 모드는 비트플래그로 관리된다. 모드 중 WriteLog 모드는 다른 모드들과 중첩해서 사용할 수 있으며, 현재까지 Serializer를 통해 이루어진 작업들이 로그로 저장되어, 디버깅에 유용하게 사용될 수 있다.  다음 사진은 WriteLog 모드를 사용했을 때 저장된 로그 파일이다. | |
| **3.2. 데이터 에디팅 툴** | |
| **- Excel**  **\* 에디팅 방식**  엔진에서 사용되는 데이터 중 GameData는 원하는 데이터를 쉽고 빠르게 넣을 수 있어야 한다. Xml파일을 보면서 Attribute를 하나씩 추가하면 매우 번거로운 작업이 될 것이다. 이를 방지하기 위해 Excel에서 GameData를 에디팅 할 수 있는 기능을 지원한다.  사용방법은 Excel의 특정 행에 해당 데이터의 속성의 이름을 작성하고 각 행마다 속성 값을 입력하면 된다. 각 데이터를 식별하고 중복 데이터를 방지하기 위해서 Key라는 속성은 반드시 추가되어 있어야 하고 속성 값도 작성되어 있어야 한다.  **\* 추출 방식**  Excel에서 Xml 파일을 추출하기 위해 VisualBasic으로 간단히 매크로를 작성해서 지원하고 있다. 이 매크로를 이용하면 XmlMap을 만들고 그 XmlMap가지고 표의 최상위 행(Attribute들)을 매핑해서, Xml파일을 추출할 수 있다.  **- UE4 에디터**  **\* 에디팅 방식**  엔진에서 일부 데이터들은 UE4의 에디터를 활용하여 에디팅할 수 있도록 지원하고 있다. Npc들의 스폰 정보를 담고 있는 SpawnData와 Npc들의 경로 정보를 담고 있는 PathPoint의 경우 실제 맵을 보면서 에디팅 해야 하기 때문이다.  일단 UE4에서 데이터들을 에디팅을 하기 위해서는 For\_Editing라는 매크로를 정의해야 사용할 수 있도록 제한하였다. 이는 에디팅에 필요한 모듈과 함수들을 게임 패키징 시 사용할 수 없기 때문에 코드를 분리한 것이다.  다음으로 데이터의 성격에 따라 특정 태그의 붙은 특정 레벨에 배치함으로써 데이터를 추가할 수 있다. 이러한 레벨의 이름은 ‘레벨이름\_태그이름’의 형식으로 만들고 최상위 Root 레벨의 하위 레벨로 추가하여야 한다. 태그의 종류는 다음과 같다.  - map : 일반적인 오브젝트 배치레벨이다. 레벨 스트리밍 볼륨에 의해서 인게임에서 스트리밍 된다.  - pathpoint : Npc의 경로 정보를 에디팅하기 위한 레벨, 인게임에서 스트리밍 되지 않는다.  - spawndata : Npc의 스폰정보를 에디팅하기 위한 레벨, 인게임에서 스트리밍 되지 않는다.  예를 들어 다음과 같이 에디팅용 레벨을 만들 수 있다.  Ex) DesertLevel\_map, DesertLevel\_spawndata, DesertLevel\_pathpoint  마지막으로 에디팅 기능을 가진 액터인 TAPathPoint, TASpawnData를 해당 레벨에 적절하게 배치한다.  // pathpoint는 TAPathPoint 액터를 배치, spawndata는 TASpawnData 액터를 배치  아래는 pathpoint의 경로정보들을 폴더로 나누어서 TAPathPoint를 배치한 모습이다.    **\* 추출 방식**  Root 레벨에는 데이터 추출을 위한 TAExporter라는 특별한 액터 하나가 존재한다. 이 액터는 UE4 에디터에서 에디팅 가능한 모든 데이터의 추출 버튼을 제공한다. 예를 들어서 ExportPathPoint를 클릭하면 태그가 pathpoint인 레벨들을 모두 찾아서 해당 레벨의 하위폴더를 순회하여 데이터를 모두 추출해준다. | |
| **3.1. 게임관련 데이터 종류** | |
| **3.3.1. GameData** | |
| GameData는 게임 플레이에 필요한 여러 가지 정보를 의미한다. 예를 들어 특정 몬스터의 공격력, 이동 속도와 같은 정보들이 저장된다.  **- 에디팅 방식**  위에서 설명한 것과 같이 필요한 GameData만큼 Excel파일을 만들어서, 각 GameData에서 필요한 속성들을 표의 최상위 행에 추가하고, 각 속성마다 값을 설정하고 추출하여 사용하면 된다.  **- 로드 과정**  Xml파일 하나를 기준으로 설명을 해보면 다음과 같다.  1. 일단 GameDataName.Xml 파일을 파일 스트링으로 읽어서 RootElement(XmlNode 클래스 객체)형식으로 파싱을 진행한다.  2. RootElement의 ChildElement(각각 하나의 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 Attribute Value와 없을 수 있는 Attribute Value를 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. | |
| **3.3.2. PathPoint** | |
| PathPoint는 Npc의 경로 정보를 저장하고 있는 데이터이다. 현재 RecastNavigationMesh를 UE4에서 추출하여 서버측에서 사용하고 있기 때문에 목적지 정보만 저장하면 되지 않을 까라는 생각도 하였지만 특정 지점들을 확정적으로 지나는 Npc경로가 필요할 수 있기 때문에 엔진에서 지원하고 있다.  **- 에디팅 방식**  앞에서 나온 내용과 같이 pathpoint 레벨태그를 가진 레벨을 만들고, 그 레벨에 폴더를 만들고 TAPathPoint객체를 순서대로 배치하면된다. 가장 처음에 위치한 TAPathPoint가 시작점이다. TAExporter를 통해서 PathPoint들을 추출하면 해당레벨의이름.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 스레드마다 잠시 락을 걸고(다른 스레드와 함께 병렬 로드를 진행하기 때문) PathPointPathSet에 PathPointPathKey(파일이름 해시값)와 함께 삽입해준다.  이때 PathPointPath객체는 PathPoint들을 배열로 가지고 있는 클래스이며, 현재 위치에서 가장 가까운 PathPoint를 찾아주거나, PahtPoint들을 정방향으로 순회하고, 끝까지 갔을 때 다시 역방향으로 순회할 수 있도록 도와주는 역할을 수행한다.  **- 동작 방식**  1. 특정 Npc의 ServerAiActorComponent가 초기화 될 때 AiGameData의 속성 중 PathPointPathKey가 설정되어 있으면 경로를 초기화한다.  ServerMoveActorSystem에서 PathPointPathKey로 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. | |
| **3.3.3. SpawnData** | |
| SpawnData는 Npc들의 스폰정보를 저장하고 있는 데이터이다. 액터의 정보, 위치, 회전 값 등을 포함하고 있다. 주로 서버에서 SpawnData를 기반으로 액터들을 스폰하고 클라이언트에서는 서버에서 통보해준 액터를 스폰한다.    **- 에디팅 방식**  앞에서 나온 내용과 같이 spawndata 레벨태그를 가진 레벨을 만들고, 그 레벨에 폴더를 만들고 TASpawnData객체를 원하는 위치에 배치하면 된다. 추출되는 정보는 Position, Rotation, GroupGameDataKey가 추출된다. GroupGameData는 어떤 캐릭터형 액터를 스폰할지 결정하는 식별자이다. 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 SpanwData 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 스폰된다. 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 해당 데이터를 프로그램 내의 객체로 만들고 잠깐 락을 걸고 unordered\_map<GameWorldType, SpawnDataGroupSet> SpawnDataGroupWorldSet에 삽입한다.  **- 동작 방식**  처음 모든 SpawnData를 로드하고 나면 서버를 시작할 때 RealWorld에 존재하는 모든 액터들을 로드한 SpawnData를 기반으로 스폰한다. | |
| **3.3.4.** **Navigation** | |
| **- UE4의 Recast/Detour Navigation, RecastNavMesh 추출 과정**  이미 UE4 에디터 상에서 직접 RecastNavMesh를 쉽게 생성할 수 있으며, Property도 쉽게 조정할 수 있도록 에디팅 환경이 구축되어 있다. 이렇게 에디팅 환경이 잘 갖추어진 RecastNavMesh정보를 그대로 직렬화해서 서버에서 역직렬화 해서 사용하고 있다.  추출 하기 위한 간단한 과정은 다음과 같다.  가장 먼저 현재 생성된 NavMesh의 데이터들이 어디에 저장되어 있는지 파악하는 일인데, 여러 디버깅과 코드확인 결과 UE4에서 생성된 RecastNavMesh정보들은 UNavigationSystemV1->ARecastNavMesh->FPImplRecastNavMesh->dtNavMesh(Recast / Detour 라이브러리에 포함된 클래스)에 저장되어 있는 것을 확인할 수 있었다. 해당 정보는 에디터에서 플레이 하고 있지 않아도, 생성되어 있는 정보이기 때문에, 바로 추출이 가능하였고 이를 추출한 파일을 똑같은 로직으로 역직렬화해서 서버에서 사용 중이다. (Recast / Detour 라이브러리도 엔진에서 std형식에 맞춰 조금 변경한 후 그대로 사용 중이다.) | |
| **3.3.5. Assets** | |
| **- 동적 비동기 로드**    **\* 로드조건**  동적 비동기를 수행하기 위해서는 두 가지 조건이 필요하다.  1. 비동기로 에셋을 로드하기 위한 FStreamableManager라는 클래스와 에셋의 파일 경로인 SoftObjectPath 객체가 필요하다.  사용하는 방법은 다음과 같다.  FStreamableManager->RequestAsyncLoad(SoftObjectPath, FStreamDelegate::CreateUObject(콜백함수를 부를 객체, 콜백함수)) 이와 같은 형식으로 작성하면 비동기 로드 작업이 완료 되었을 때, 해당 콜백함수를 통해서 성공 실패 여부를 알려주고, 성공했다면 로드된 객체를 얻어서 사용할 수 있다.  2. SoftObjectPath가 미리 로드 되어있어야 한다. 이를 실현시키기 위해서 SoftObjectPath를 포함하는 새로운 모듈을 만들어 SoftObjectPath를 사용하는 모듈보다 먼저 로드 해야한다.  다음과 같이 처리하였다.  2.1. TAGameClientSetting모듈을 하나 더 만들고 그 안에 TAAssets이라는 에셋들의 SoftObjectPath들을 담고있는 UObject클래스 하나를 만든다.  2.2. SoftObjectPath들은 Config/DefaultTAAssets.ini 파일을 통해서 추가 한다.  2.3. TAGameClient에서 TAAssets을 사용하게 하기 위해서 TAGameClientSetting모듈이 더 빨리 로드 되도록 TAGameClient.uproject파일에 해당 모듈의 json 값을 수정한다. “LoadingPhase”:”PreDefault”로 수정하면 “Default”로 설정한 TAGameClient모듈보다 더 빨리 로드되도록 하였다.  두번째 조건에서 무조건 미리 로드된 SoftObjectPath에만 동작하는 점이 의아해서 한 파일에 대한 에셋 경로를 추가하지 않고, 경로 문자열을 그대로 들고 와서 SoftObjectPath로 만들고 그 객체로 그대로 동적 비동기 로드를 시켜봤는데, 동작하지 않았다.  **\* 데이터 검증**  액터에 따라 다양한 UE4의 리소스(Mesh 정보 등)를 사용해야 한다. 이를 위해서 액터마다 렌더링 정보를 담고있는 RenderingGameData를 가지고 있는데, 이 데이터 안에 UE4의 에셋 경로를 문자열 형식으로 담고 있고, 이 경로를 기반으로 에셋을 사용할 수 있게 된다.  여기서 RenderingGameData안에 있는 에셋 경로가 유효한지 판단하기 위해 UE4안에서 정의된 에셋을 에셋 경로를 <에셋 경로, 정의된 에셋> 형식으로 저장하고 있다. 해당 TMap에 존재한다면 유효한 에셋 경로임을 알 수 있다. | |
| **4. 세부 요소기술 설명** | |
| **4.1.** **게임 엔진과 UE4(클라이언트)의 통신** | |
| **- TAGameEvent**  게임 클라이언트로 UE4을 사용하고 있기 때문에 멀티스레드 기반 C++ 엔진과 UE4 간 통신 방법이 필요하다.  UE4에서 게임 루프는 싱글스레드(게임스레드)에서 돌아간다. 근데 엔진의 TAGameClient는 멀티스레드로 동작한다. 그렇기 때문에 TAGameClient에서 무작정 UE4에 있는 메모리에 접근하면 데이터 경쟁 이슈가 발생한다. 이를 해결하기 위해 이벤트를 쌓아두고 처리할 수 있는 TAGameEventQueue를 만들어서 UE4와 TAGameClient가 통신하도록 하였다.  동작 방식을 간단히 설명해보면 먼저 TAGameEventQueue는 UE4의 TAGameInstance에 존재한다. 엔진의 TAGameClient에서 UE4로 메시지를 보내야 할 때, TAGameEventQueue에 이벤트를 추가한다. 이 과정은 멀티스레드로 동작하기 때문에 동기화를 위한 락이 필요하다. 이후 TAGameInstance가 매 프레임마다 TAGameEventQueue를 확인해서 쌓여있던 이벤트들을 처리한다. | |
| **4.2. AI** | |
| **- 행동 트리(BehaviorTree)**  엔진에서는 Ai 로직은 행동 트리로 구성되어 있다.  행동 트리에 대해 간단히 설명하자면 Ai를 구현하는데 사용되는 트리 형식의 설계 기법으로 일반적으로 많이 사용하는 FSM과 비교했을 때 계층 구조로 되어있기 때문에 행동의 우선 순위를 결정할 수 있고, Ai 흐름을 유연하게 컨트롤할 수 있다. 또한 FSM과 비교했을 때 여러 가지 상태를 복합적으로 가질 수 있으며, 복잡한 AI일수록 FSM과 대조적으로 코드가 간결해지고, 유지 보수에 강점이 있다.  엔진에서 제공하는 행동 트리 클래스인 AiBehaviorTree은 크게 다음과 같은 요소로 구성되어 있다. (뒤에서 조금 세부적으로 설명할 예정이기 때문에 간단히 설명하겠다.)  \* **AiBTNode** : 조건, 실행 목록을 가진 가장 기본 노드  \* **AiBTNodeComposite** : Selector, Sequence, Block 같은 특정 조건을 가진 분기점 노드(자식 노드 소유)  \* **AiBTNodeState** : 노드의 상태(Success, Failure, Continue)  \* **AiBTNodeCondition** : 모든 노드에 포함되어 있는 노드의 조건  \* **AiBTNodeExecution** : 해당 노드를 실행할 때 수행되는 동작    그리고 액터들의 다양한 Ai에서 사용되는 조건들을 재사용하기 위해 여러 개의 static AiBTNodeCondition을 가지고 있으며, Ai를 재사용하기 위해 여러 개의 static AiBTNodeBlock(루트 노드 역할)를 가진다.    **\* AiBTNode**  가장 일반적으로 사용되는 AiBehaviorTree의 노드를 의미하며 거의 모든 노드는 AiBTNode를 상속받고 있다.  AiBehaviorTree는 루트 AiBTNode부터 시작해 많고 다양한 여러 개의 AiBTNode로 구성되어 있다. AiBehaviorTree를 실행시키면 이 루트 AiBTNode부터 시작해서 그 아래 있는 자식 AiBTNode까지 ExecuteNode 함수를 호출한다.  AiBTNode는 멤버로 AiBTNodeCondition배열과 AiBTNodeExecution배열을 가진다. AiBTNodeCondition은 조건을 검사하는 객체로 해당 노드를 실행할 수 있는지 판별할 때 사용된다. 이 조건 객체 배열 순회하면서 하나 이상의 조건을 만족하지 못한다면 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. AiBTNode를 상속받은 객체의 실제 구현내용은 ExecuteNodeDetail이라는 함수를 오버라이딩하여 사용한다.  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 구성하게 된다.    **\* AiBTNodeComposite**  AiBTNodeComposite는 자식 노드를 가질 수 있는 노드의 종류로 트리에서 분기점 역할을 하는 노드이다.  자식 노드의 ExecuteNode 결과 값에 따라서 다양한 방식으로 동작하며, 종류는 3가지로 나뉜다. 3종류 모두 자식노드가 AiBTNodeState::Continue를 반환하면 그대로 반환하고 나머지 상황에서는 다음과 같이 동작한다.  \* **Selector** : 자식노드의 ExecuteNode 결과 값이 하나라도 성공 시 AiBTNodeState::Success을 반환하고, 모두 실패 시 AiBTNodeState::Failure을 반환한다.  \* **Sequence** : 자식노드의 ExecuteNode 결과 값이 하나라도 실패 시 AiBTNodeState::Failure을 반환하고, 모두 성공 시 AiBTNodeState::Success을 반환한다.  \* **Block** : 자식노드의 ExecuteNode 결과 값에 상관없이 무조건 AiBTNodeState::Success을 반환한다.  현재 있는 AiBTNode들과 상속구조를 나열해보면 다음과 같다.    **\* AiBTNodeCondition**  AiBTNodeCondition은 노드의 조건을 검사하는 객체이다. 이 조건 객체를 통해 해당 노드가 실행될 수 있는지 판단한다. 모든 노드는 조건 객체의 리스트를 들고 있다. 공통적으로 사용되는 조건을 정의하기 위해 static 멤버 변수로도 정의하고 있다.  멤버변수로 IsNot라는 플래그가 존재하는데 이를 통해 조건의 반대 조건을 쉽게 정의할 수 있다. 예를 들어 AiBTConditionHasTarget이라는 조건에서 IsNot가 켜져있으면 타겟을 가지지 않는 조건이 된다.  멤버함수로 CheckCondition과 가상함수인 CheckConditionDetail이 존재하는데, CheckCondition은 CheckConditionDetail를 수행해보고 IsNot에 따라 true, false를 결정하게 된다. 즉 AiBTNodeCondtion을 상속받은 클래스는 CheckConditionDetail을 오버라이딩하여 구현하면 된다.  현재 있는 조건들과 상속구조를 나열해보면 다음과 같다.    **\* AiBTNodeExecution**  AiBTNodeExecution은 해당 노드를 실행할 때 수행되는 동작을 의미한다. 노드를 순회할 때 조건에 따라서 주기적으로 해야 할 것들을 정의하기 위해서 사용된다.  AiBTNodeExecution도 AiBTNodeBase를 상속받고 있기 때문에, AiBTNodeCondition을 추가해서 특정 조건에서만 수행되도록 설정할 수도 있다.  현재 있는 실행객체들과 상속구조를 나열해보면 다음과 같다. | |
| **4.3. Move** | |
| **- Sector의 개념과 필요성**  엔진에서는 액터들이 존재하는 월드 공간을 Sector라는 개념으로 분할하여 사용하고 있다. Sector는 정방형의 일정한 크기의 영역으로 정의된다. 월드에 스폰된 액터들은 항상 하나의 Sector에 포함되어 있다.  Sector개념이 필요한 이유는 간단히 말하면 최적화를 위해서다. 두 가지 상황에 대해 예를 들어보겠다.  1. Sector개념이 없을 때 한 몬스터 액터가 플레이어 액터를 타겟으로 잡기 위해서는 월드에 있는 모든 플레이어 액터들에 대해 거리를 계산하면서 타겟을 찾아야한다. (거리 외에 다른 조건도 많을 것이다.)  2. 멀티플레이어 게임 프로그래밍 관점에서 내가 움직이면 상대방에게 내가 움직이고 있다는 것을 실시간으로 통보해야 한다. 그래야 상대방 클라이언트에서도 움직이는 내 캐릭터를 렌더링할 수 있다. 만약 Sector라는 개념이 없다면, 내 캐릭터 액터가 움직일 때마다 현재 시야에서 보이지도 않는 모든 플레이어에게 내 위치를 통보해야 할 것이다.  이런 상황에서 Sector라는 개념이 존재한다면 Sector안에 있는 액터로 한정할 수 있다  **- Sector의 관리**  이러한 공간 분할이 정상적으로 동작하기 위해서는 Sector에 속해있는 액터리스트를 항상 최신 상태로 유지하여 액터들에게 정보를 제공해야 한다.  액터가 Sector를 사용하기 위한 조건은 다음과 같다.  1. 액터가 Sector의 경계를 넘어가면 이동한 새로운 Sector의 액터리스트에 해당 액터를 추가하고, 이전 Sector에서는 해당 액터를 삭제한다.  2. 액터의 시야 범위를 속해 있는 Sector로 한정하면 인접한 Sector들의 경계에 존재하는 액터들의 경우 거리가 가까워도 볼 수 없기 때문에 인접한 Sector들도 볼 수 있도록 해야한다.  2번 조건 그림으로 표현하면 다음과 같다.      2번의 조건인 인접 Sector 액터들이 보이도록 하는 것은 별로 어렵지 않다. 현재 Sector기준으로 팔 방향에 존재하는 Sector를 같이 처리하면 된다. (액터는 속한 Sector를 포함하여 총 9개의 Sector를 본다.)  이제 1번 조건을 만족시키기 위해 다음과 같은 로직을 사용한다고 생각해보자.  1. 이동할 때 서로 이동하는 사실을 통보해준다.  2. Sector를 이동했을 때, 일단 새로 생긴 Sector들의 액터들에게 나를 생성하라고 통보하고, 나도 새로 생긴 Sector들의 액터들을 생성하면 된다.  3. 지워진 Sector들에는 액터들에게 나를 지우라고 통보하고, 나도 지워진 Sector들의 액터들을 지워준다.  하지만 위의 방식은 멀티스레드 환경에서 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.  이를 그림으로 나타내면 다음과 같다.    이를 해결하기 위해 그렇기 때문에 액터는 현재 해당 액터가 보고 있는 액터들인 ‘ViewList’라는 것을 관리한다.  만약에 위에 그림과 같은 상황이 나타나도 A가 움직이면서 인접Sector들의 액터들에게 움직이는 것을 알려주려고 봤는데, 인접 Sector의 액터인 B가 아직 ViewList에 없다면, A한테 B를 생성시켜줄 수 있다. 반대로 B의 ViewList에 A가 없으면 생성시켜줄 수 있다. 결론적으로 ViewList는 멀티스레드 때문에 누락된 액터들을 뒤늦게 라도 처리해주는 역할을 하는 것이다. 플레이어 액터뿐 아니라 Npc도 ViewList를 관리해야 하는데, 그 이유는 ViewList에 플레이어가 들어온 경우 AI를 활성화시켜줘야 하기 때문이다.  **- Sector 이동 로직 의사코드**  나를 MyActor라고 하고, 현재 루프를 도는 Actor를 TargetActor라고 한다.  새로운 인접 Sector들의 액터들을 NewActors, 이전의 인접 Sector들의 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. | |
| **4.4. Item** | |
| **- Item 구조와 ItemSet**  Item은 기본적으로 기본 정보와 세부 정보로 나누어져 있다.  \* 기본 정보 : 변하지 않는 불변 정보(ItemType, RenderingGameData 등)  \* 세부 정보 : 변할 수 있는 가변 정보(Durability, EnchantLevel 등)  따라서 같은 아이템이라도 세부 정보에 따라서 서로 다른 아이템이 될 수 있다.  ItemSet은 Item들을 담고 있는 클래스이다. ItemSetType으로 ContainerType과 SlotType을 선택할 수 있도록 설계하였는데, 이에 따라 담을 수 있는 아이템이 결정된다.  \* ContainerType : ItemSet의 모든 슬롯에 담을 수 있는 ItemType의 제한이 없기 때문에 다양한 Item을 담을 수 있어야 하는 인벤토리나 보물상자 등을 구현하기 위해 사용된다.  \* SlotType : 특정 슬롯에 담을 수 있는 ItemType이 한정되어 있기 때문에 장비 착용 슬롯 등을 구현하기 위해 사용된다. | |
| **4.5. UI** | |
| **- TAChunkUserWidget**  UI 관련 코드를 작성하다 보면 많은 UI에서 소유하는 액터에 접근하거나 활성화, 비활성화 하는 작업이 많았다. 이러한 코드를 각각 전부 작성하지 않고 기본적으로 제공하기 위해 공통 기능을 모아둔 UserWidget인 TAChunkUserWidget이라는 기본 UI 클래스를 만들었다.  Viewport에 바로 추가되는 최상위 UserWidget같은 경우 TAChunkUserWidget을 상속받아서 사용하고 최상위 UserWidget 하위에 있는 ChildUserWidget의 경우는 최상위 UserWidget을 통해서 TAChunkUserWidget에 접근해서 필요기능을 사용하는 식으로 사용하고 있다. | |
| **4.6.** **Interaction** | |
| **- 상호작용 가능한 액터 탐색**  게임을 진행 하다 보면 대화, 채집 등을 위해 액터와 상호작용 해야 하는 경우가 있다. 상호작용한 액터를 찾기 위해 TAGameInstance에서는 엔진 서버에서 내려준 액터들을 UE4 형식의 액터로 스폰해서 리스트로 보관하고 있다.  TAGameInstance의 일정시간마다, 해당 액터리스트를 순회하면서 상호작용 가능 거리에 있는 가장 가까운 액터를 InteractionActor로 설정하고 UI를 띄워준다.  **- InteractionButton과 InteractionObject**  TAGameInstance에서 InteractionActor가 결정되면 어떤 상호작용을 할 수 있는지 버튼UI를 띄워줘야 한다. 해당 액터의 CharacterGameData에 존재하는 InteractionType 리스트를 보고 버튼들을 결정한다. 이때 InteractionType 개수에 따라 InteractionButton 개수가 결정된다.  그렇게 InteractionButton을 띄우고 나면 해당 버튼을 클릭했을 때(또는 바인딩된 키보드 키를 눌렀을 때), 그에 따른 동작을 해야 한다. 예를 들어 대화라는 상호작용을 하면 하던 것을 멈추고 이쪽을 돌아보고 다이얼로그 창을 열어야 한다. 이를 실현하기 위해 InteractionButton안에는 InterationObject라는 특정 기능을 수행하는 객체들의 배열이 있다. 상호작용 시 이와 연관된 InteractionObject들은 순차적으로 수행된다.    **- 상호작용과 카메라 전환**  대화라는 상호작용을 수행할 때는 다이얼로그 창이 열리고 카메라는 해당 Npc를 포커스해야 한다. 또한 말하는 도중에는 플레이어의 키보드 입력을 막아야한다.  해당 Npc를 포커스하기 위해서 먼저 CameraMode를 FocusedAndControlBlocked로 변경 후, 카메라를 Npc 액터기준으로 바라보는 방향으로 조금 이동하고 Npc액터를 바라보도록 설정한다. 그리고 CameraDirtyFlag를 켜준다. 그러면 Player Tick에서 보간 작업이 이루어지면서 자연스럽게 전환된다.  키보드 입력에 따라서 플레이어가 움직이는 로직은 모두 MoveAndRotateCharacterByInput이라는 함수를 통해서 들어오도록 작성되어 있는데 현재 CameraMode가 FocusedAndControlBlocked로 설정되어 있을 때 이 함수를 수행하지 않고 리턴 한다. | |
| **5. 기타** | |
| **5.1. 실수를 줄이기 위한 방법** | |
| **- TA\_ASSERT\_DEV, TA\_LOG\_DEV, TA\_COMPILE\_DEV**  엔진에서는 실수를 줄이기 위해 4가지 유용한 매크로를 지원하고 있다.  \* **TA\_LOG\_DEV** : 간단히 문자열을 출력한다.  인자로 문자열로 넘겨주고 그대로 콘솔창에 출력한다. UE4라면 UE4로그에 출력한다.  \* **TA\_ASSERT\_DEV** : 표현식에 따라서 로그 출력과 Break를 걸어준다.  인자로 표현식과 문자열을 넘겨주고 표현식이 false일 때만 문자열 출력과 DebugBreak()를 수행한다. UE4라면 UE4로그에 Error 형식의 로그를 출력한다.  \* **TA\_COMPILE\_DEV** : 표현식에 따라서 컴파일 에러를 발생시킨다.  인자로 들어온 표현식이 false일 때만 char msg[-1]와 같은 컴파일 에러를 발생시키는 코드를 생성한다.  \* **TA\_TEMP\_DEV** : 임시 코드를 표시하기 위해서 사용된다.  프로젝트를 진행하다 보면 나중에 반드시 수정해야 하는 임시 코드를 작성해야 할 수 있다.  임시 코드 리마인드 차원에서 임시 코드에 TA\_TEMP\_DEV을 작성하고 나중에 확인할 때 TA\_TEMP\_DEV 정의를 주석 처리하고 컴파일해보면 해당 위치를 알 수 있다. | |
| **5.2. 빠른 개발을 위한 방법** | |
| **- 컴파일타임을 줄이기 위한 방법**  **\* Pimpl패턴과 전방선언**  프로그래밍을 하다 보면 사용하지도 않는 파일을 인클루드하여 사용할 때가 있다. 이러한 인클루드는 여러 해더에 가려져 있어서 눈에 잘 안보이는 경우가 많다. 사용하지 않는 파일을 인클루드하면 수정하지도 않았지만 다시 컴파일 되는 파일이 많아진다. 이러한 현상을 막기 위해 근본적으로 헤더파일에서 다른 헤더파일의 인클루드를 최대한 줄여야 한다.  여기서 활용할 수 있는 사실은 포인터형 레퍼런스형은 내부 구현을 사용하지 않는다면 인클루드 하지 않고 전방선언만으로 컴파일 가능하다는 점이다. 이를 활용하여 전방선언을 최대한 활용하여 헤더파일 종속성을 줄이고 컴파일 시간을 줄일 수 있다.  Pimpl 패턴은 포인터와 전방선언을 활용하여 종속성을 줄여 컴파일 시간을 줄이는 패턴이다. 예를 들어보면 일반적으로 클래스 멤버변수로 다른 클래스의 객체를 소유할 때 다음과 같이 작성한다.  #include “ClassName2.h”  ClassName1  {  …  …  Private:  ClassName2 \_className2Object;  }  여기서 ClassName2 \_className2Object;이 부분을 ClassName2\* \_className2Object;로 바꿔주고 ClassName1해더 앞에 ClassName2를 인클루드 하는 대신 class ClassName2;로 선언하면 ClassName2해더파일 인클루드 없이 정의할 수 있고 결과적으로 컴파일 시간을 줄일 수 있다.  **- 매크로 스위치문 활용**  비슷한 로직을 구현하는 소스코드를 작성하다 보면 어쩔 수 없이 비슷한 코드를 작성하는 경우가 있다. 예를 들어서 각 ActorComponentPool에서 ActorComponent를 얻어 오는 코드에서 ActorComponent타입이 추가될 때마다 해당 코드 블록을 계속 작성해야 한다. 이를 빠르고 정확하게 하기 위해 코드 블록을 매크로로 정의해서 사용할 수 있다.    **- 템플릿 활용**  오버로딩 대신 템플릿을 활용하면 타입이 정해지지 않아도 소스코드를 작성할 수 있고 동일한 소스코드를 실수 없이 작성할 수 있다. 다만 특정 타입에 대해 예외가 생길 수 있기 때문에 이러한 타입을 처리할 때는 std::enable\_if를 활용하여 템플릿 인스턴스 생성을 제한하거나 특수화를 사용하여 적절하게 처리해줘야 한다. | |
| **5.3.** **그외 기능들** | |
| **- Enum <=> String 변환**  프로그래밍을 하다 보면 Enum과 String을 서로 변환해야 할 경우가 있다.  \* Enum -> String의 경우 : Serializer에서 데이터 처리에 대한 로그를 작성할 때, 데이터들이 각각 어떤 자료 형인지 로그파일에 기록해야 한다. 여기서 TADataType(Int8, Int16, Float …)를 문자열(“Int8”, “Int16”, “Float” …)로 변환해야 한다.  \* String -> Enum의 경우 : ItemGameData를 xml파일에서 로드할 때 “ItemType::Consumable”와 같은 문자열을 ItemType::Consumable로 변환해야 한다.  이런 변환을 진행하기 위해서 unordered\_map<string, EnumType> 형식으로 Enum과 String을 저장하고 템플릿과 매크로를 활용하여 ConvertEnumToString, ConvertStringToEnum함수를 생성해서 사용하고 있다.    **- ToStringCast와 FromStringCast**  String과 여러가지 자료형의 변환을 자원하는 템플릿 함수다. ostringstream과 istringstream을 사용하여 구현하고 있다. | |
| **6. 참고 자료** | |
| **<링크>**  **프로그래밍(서버, 기타 등)**  <https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsasocketa>  <https://docs.microsoft.com/en-us/windows/win32/api/mswsock/nf-mswsock-acceptex>  <https://docs.microsoft.com/en-us/windows/win32/api/mswsock/nc-mswsock-lpfn_connectex>  <https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup>  <https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsasend>  <https://docs.microsoft.com/ko-kr/sql/connect/odbc/cpp-code-example-app-connect-access-sql-db?view=sql-server-ver15>  <https://docs.microsoft.com/ko-kr/cpp/cpp/noexcept-cpp?view=msvc-160&viewFallbackFrom=vs-2019>  <https://docs.microsoft.com/ko-kr/cpp/cpp/declspec?view=msvc-160&viewFallbackFrom=vs-2019>  <https://stackoverflow.com/questions/1372480/c-redefinition-header-files-winsock2-h>  <https://stackoverflow.com/questions/62256738/visual-studio-2019-c-and-stdfilesystem>  <https://stackoverflow.com/questions/5100283/how-do-i-setup-visual-studio-to-register-some-defines-globally>  <https://stackoverflow.com/questions/922360/why-cant-i-make-a-vector-of-references>  <https://stackoverflow.com/questions/2170688/private-virtual-method-in-c>  <https://stackoverflow.com/questions/17871880/should-i-use-wchar-t-when-using-utf-8>  <https://stackoverflow.com/questions/2259544/is-wchar-t-needed-for-unicode-support>  <https://softwareengineering.stackexchange.com/questions/286676/algorithm-data-structure-to-answer-what-recipes-can-i-make-with-this-set-of-ing>  <https://www.gamasutra.com/view/feature/131505/toward_more_realistic_pathfinding.php>  <http://codevba.com/excel/XmlMap.htm#.YIlVOLUzabh>  <http://wiki.hash.kr/index.php/%ED%8A%B8%EB%9E%9C%EC%9E%AD%EC%85%98>  <http://wiki.hash.kr/index.php/%EC%BF%BC%EB%A6%AC>  <https://ko.wikipedia.org/wiki/%EC%8A%A4%ED%95%80%EB%9D%BD> <https://ko.wikipedia.org/wiki/%ED%8A%9C%ED%94%8C>  <https://ko.wikipedia.org/wiki/%ED%8F%B4%EB%A7%81_(%EC%BB%B4%ED%93%A8%ED%84%B0_%EA%B3%BC%ED%95%99)>  <https://namu.wiki/w/UTF-8>  <https://ju3un.github.io/encodng-unicode-ansi/>  <https://gameprogrammingpatterns.com/object-pool.html>  <https://thispointer.com/how-to-use-unordered_set-with-user-defined-classes-tutorial-example/>  <http://rpclib.net/gettingstarted/>  <https://ko.wikipedia.org/wiki/%EB%8B%A4%EC%A4%91%ED%99%94_(%ED%86%B5%EC%8B%A0)>  <https://modoocode.com/306#page-heading-9>  <https://modoocode.com/215>  <https://popcorntree.tistory.com/8>  <https://popcorntree.tistory.com/39?category=813523>  <http://sweeper.egloos.com/3003333>  <http://egloos.zum.com/sweeper/v/3003805>  <http://egloos.zum.com/sweeper/v/3147987>  <https://junshock5.tistory.com/28>  <https://jhnyang.tistory.com/262>  <https://lacti.github.io/2013/08/20/simple-cpp-rpc/>  <https://nesoy.github.io/articles/2019-07/RPC>  <https://modoocode.com/284>  <https://ozt88.tistory.com/23>  [https://ozt88.tistory.com/20](https://ozt88.tistory.com/20#comment5563178)  <https://ozt88.tistory.com/19>  <https://ozt88.tistory.com/25?category=123069>  <https://coding-factory.tistory.com/245>  **Unreal Engine**  <https://docs.unrealengine.com/ko/InteractiveExperiences/Networking/Actors/RPCs/index.html>  <https://docs.unrealengine.com/ko/ProductionPipelines/DevelopmentSetup/CodingStandard/index.html>  <https://docs.unrealengine.com/ko/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/ActorLifecycle/index.html>  <https://docs.unrealengine.com/ko/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Objects/Optimizations/index.html>  <https://docs.unrealengine.com/ko/InteractiveExperiences/UMG/UserGuide/WidgetTypeReference/index.html>  <https://docs.unrealengine.com/ko/InteractiveExperiences/UMG/UserGuide/Slots/index.html>  <https://docs.unrealengine.com/en-US/Basics/Levels/LevelsWindow/index.html>  <https://docs.unrealengine.com/ko/ProgrammingAndScripting/ProgrammingWithCPP/Assets/AsyncLoading/index.html>  <https://github.com/recastnavigation/recastnavigation>  <https://blog.gamedev.tv/creating-unreal-engine-ui-with-umg-and-c/>  <https://www.gamedev.net/blogs/entry/2210775-more-recast-and-detour/>  <https://www.ue4community.wiki/memory-management-6rlf3v4i>  <https://www.ue4community.wiki/legacy/assets-naming-convention-qqp2b5m1#Folders_by_categories>  <https://dawnarc.com/2018/07/ue4-tsharedptr-tweakobjectptr-and-tuniqueptr/>  <https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection>  <https://issues.unrealengine.com/issue/UE-87619>  <http://egloos.zum.com/sweeper/v/3205731>  <https://m.blog.naver.com/destiny9720/220934094119>  **<책>**  - Windows via C/C++  - Effective C++  - Effective Modern C++  - 게임 서버 프로그래밍 교과서  - 멀티플레이어 게임프로그래밍  - 이득우의 언리얼 C++ 게임 개발의 정석  - 게임 프로그래밍 패턴  - 자료구조 알고리즘 운영체제 기본서 등등 | |