2단계_08. 아이템 시스템

수강일	@2023/07/11
이름	전영재
멘토	Min-Kang Song,현웅 최
작성 상태	Done
강의 시청 여부	☑

Contents

아이템 시스템

💡 **강의 목표**

- 트리거 박스를 활용한 아이템 상자의 구현
- 다양한 종류의 아이템에 대한 개별적인 습득 처리의 구현
- 소프트오브젝트 레퍼런스와 하드오브젝트 레퍼런스의 차이 이해

---

[1] 트리거 박스의 설정

박스 콜리전을 사용해 이벤트를 트리거 하는 액터를 만들어보자.

메쉬 컴포넌트의 콜리전은 NoCollision으로 설정해, 실질적인 충돌 이벤트는 트리거 박스에서만 사용하는 것이 핵심이다.

이제, 이벤트의 오버랩과 기능함수를 **델리게이트**를 통해 연결해주어야 하는데, 이것은 언리얼엔진의 컴포넌트 내에 이미 구현되어 있으니 활용하면 된다.

```
AABItemBox::AABItemBox()
{
  ...
  Trigger->OnComponentBeginOverlap.AddDynamic(this, &AABItemBox::OnOverlapBegin);
  ...
}
void AABItemBox::OnOverlapBegin(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 Ot
{ //함수의 인자형식에 주의하자. OnComponentBeginOverlap 델리게이트를 F12로 타고들어가면 해당 형식을 볼 수 있다.
  ...
}
```

데이터 애셋

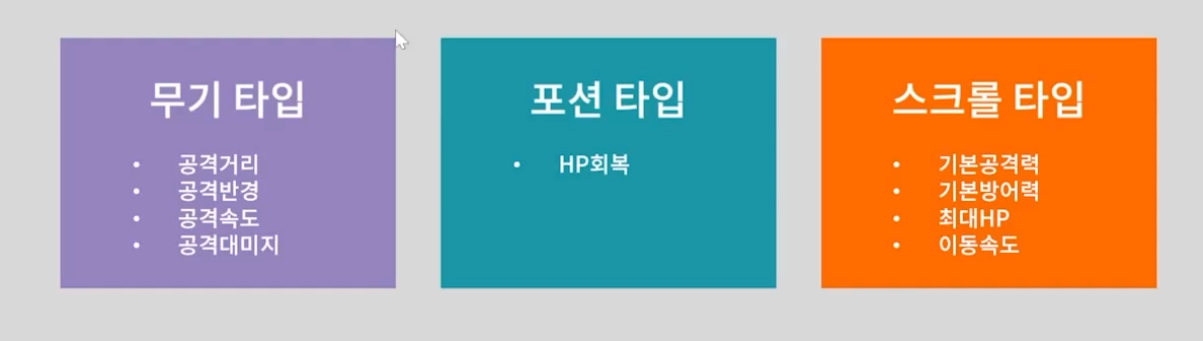
아이템 박스 내에 존재할 아이템에 대한 데이터를 데이터 애셋을 통해 만들어보자.
데이터 애셋 클래스 내에 Enum열거형을 정의해주는 것으로 에디터 내에서 해당 Enum을 사용할 수 있다.

```
#include "CoreMinimal.h"
#include "Engine/DataAsset.h"
#include "ABItemData.generated.h"

UENUM(BlueprintType)
enum class EItemType : uint8
{
  Weapon = 0,
  Potion,
  Scroll
};

UCLASS()
class ARENABATTLE_API UABItemData : public UPrimaryDataAsset
{
  GENERATED_BODY()

public:
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Type)
  EItemType Type;
};
```

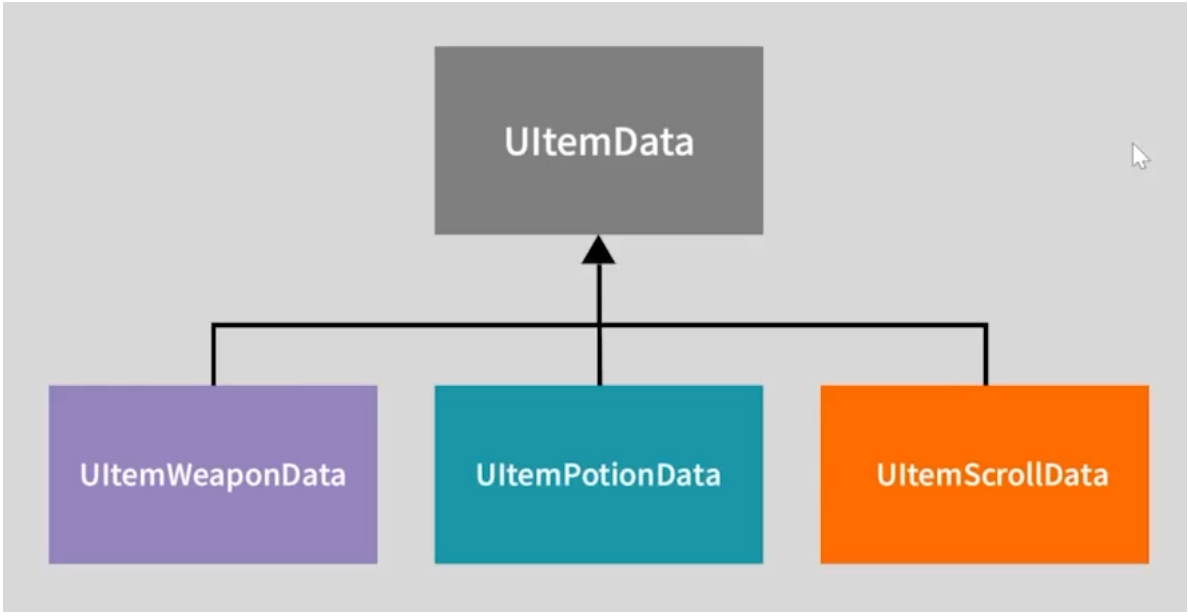
---

[2] 의존성 분리를 위한 설계 규칙

우리는 앞으로 다음과 같이 아이템 타입별 기능을 구현할 것이다.



그렇다면 아래와 같은 데이터 구조를 가지게 될것인데



게임의 특성 상, 앞으로 여러 타입의 아이템이 기획적으로 추가될 수 있다. 이때마다 새로운 함수를 만들고 로직을 추가해야 한다면 번거로운 작업이 될 것이다. 이를 방지하기 위해 **의존성을 분리한 데이터 구조**가 필요하다.

프로젝트 레이어

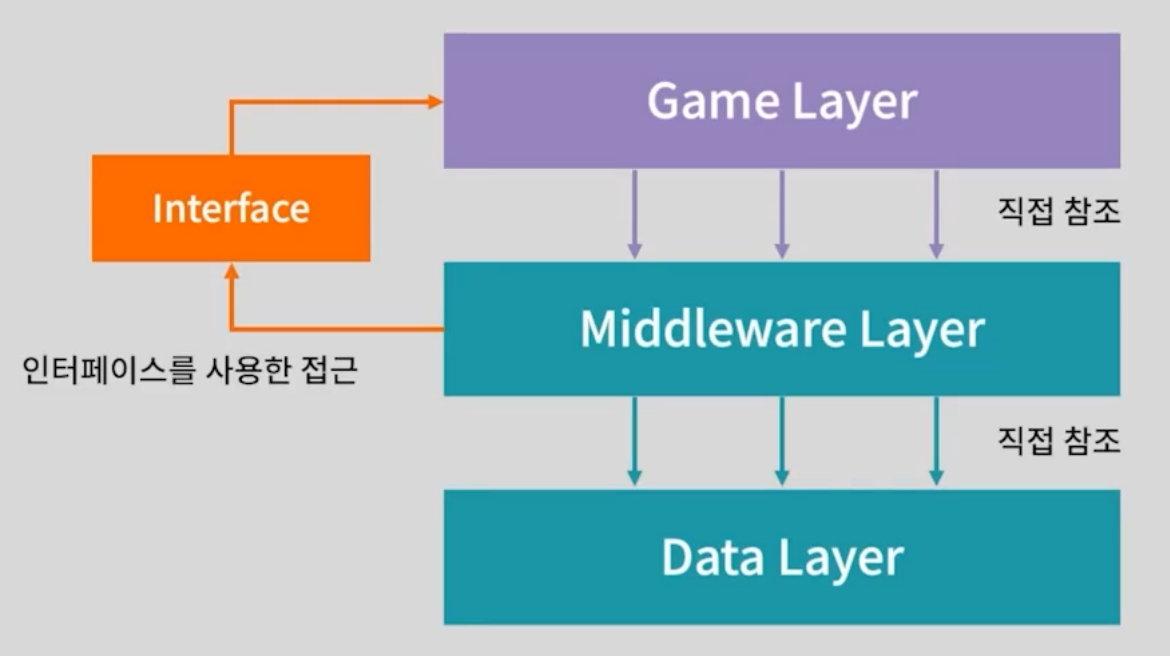
우선, 프로젝트의 레이어 구조에 대한 이해가 필요하다.

레이어란 간단히 프로젝트에 사용되는 데이터를 계층으로 구분지어 놓은 것이라 보면 된다

데이터 레이어란 스탯정보, 캐릭터 레벨 테이블과 같은 게임을 구성하는 **기본 데이터**들을 보관하는 레이어다.

미들웨어 레이어는 게임에 사용되는, 서비스를 제공하는 **독립적인 모듈**을 의미한다. 앞서 제작한 UI, 지금 제작중인 아이템, 앞으로 나올 AI들은 게임 전체적으로 사용되는 모듈이라고 볼 수 있다.

게임 레이어는 캐릭터, 게임모드와 같이 **게임의 로직**을 구체적으로 구현하는데 사용되는 레이어다.



이때, **상단의 레이어는 하단의 레이어들을 직접 참조**할 수 있지만, 역으로 **하단은 상단을 인터페이스 등을 통해 참조**하도록 설계하는 것이 **의존성 분리**의 첫걸음이다.

쉽게 생각하면, 캐릭터의 애니메이션을 넣을때 당연히 스켈레탈 메쉬에 애님BP를 참조하여 넣게 된다. 그러나 애님BP는 캐릭터에 대한 참조가 필요없다. 똑같이 애님BP를 작성할때도, BP내에서는 어떤 애니메이션을 참조할 지 애니메이션 애셋에 대한 참조가 필요하다. 그러나 애니메이션 애셋은 애님BP에 대한 참조가 필요 없다.

예외의 경우로 애님BP에서 캐릭터의 Velocity등을 필요로 할 때가 있다. 이 경우 인터페이스를 사용한 접근을 써야한다는 것이다.

Wrapper구조체

아이템 타입 별 기능을 구현할 때, `Switch` 문을 통해 분기를 나눌 수도 있지만 본 강의에서는 **배열**을 사용했다. 델리게이트에 바인드 되어 작동하는 함수 내에 새로운 델리게이트들의 배열을 만들고, 인자로 받은 배열번호(=(uint8)Type)에 해당하는 델리게이트를 실행하는 방식이다.

델리게이트는 배열을 넘겨줄 수 없기에 배열번호를 받아 사용하는 것이며, 해당 배열내에 델리게이트를 담기 위해 클래스 내에 커스텀 **구조체**를 생성하였다.

언리얼엔진 내 **Wrapper구조체**를 만드는 방식을 알아보자.

```
ABCharacterBase.h
DECLARE_DELEGATE_OneParam(FOnTakeItemDelegate, class UABItemData* /*InItemData*/);
USTRUCT(BlueprintType)
struct FTakeItemDelegateWrapper
{
  GENERATED_BODY() //언리얼의 GENERATED_BODY 매크로 사용
  FTakeItemDelegateWrapper() {} //생성자 구현
  FTakeItemDelegateWrapper(const FOnTakeItemDelegate& InItemDelegate) : ItemDelegate(InItemDelegate) {} //인자를 받는 버전의 생성자
  FOnTakeItemDelegate ItemDelegate; //델리게이트 인자
```

```
};
...
protected:
UPROPERTY()
  TArray<FTakeItemDelegateWrapper> TakeItemActions;

  virtual void TakeItem(class UABItemData* InItemData) override;
  virtual void EquipWeapon(class UABItemData* InItemData);
  virtual void DrinkPotion(class UABItemData* InItemData);
  virtual void ReadScroll(class UABItemData* InItemData);
```

```
AABCharacterBase::AABCharacterBase()
{
  ...
  // Item Actions
  //CreateUObject를 통해 델리게이트에서 바로 인스턴스를 생성해서 넘겨준다.
  TakeItemActions.Add(FTakeItemDelegateWrapper(FOnTakeItemDelegate::CreateUObject(this, &AABCharacterBase::EquipWeapon)));
  TakeItemActions.Add(FTakeItemDelegateWrapper(FOnTakeItemDelegate::CreateUObject(this, &AABCharacterBase::DrinkPotion)));
  TakeItemActions.Add(FTakeItemDelegateWrapper(FOnTakeItemDelegate::CreateUObject(this, &AABCharacterBase::ReadScroll)));
}
void AABCharacterBase::TakeItem(UABItemData* InItemData)
{
  if (InItemData)
  {
    TakeItemActions[(uint8)InItemData->Type].ItemDelegate.ExecuteIfBound(InItemData);
  }
}
```

델리게이트 전달의 전체적인 흐름을 살펴보면,

1. 아이템박스의 박스콜리전에 닿게되면, 박스콜리전의 OnOverlapBegin을 발동시키고, 인터페이스를 통해 캐릭터의 TakeItem을 발동시키며 자신의 Type을 전해준다.
2. TakeItem은 전해받은 Type을 통해 TakeItemActions라는 델리게이트가 담긴 배열을 작동시킨다. 이때 전해받은 Type이 해당 배열의 몇번째 배열을 사용하게 될지 정하는 지표가 된다.

솔직히 말해서 이렇게까지 할 필요가 있나싶다. 어쩌피 델리게이트가 Character클래스 내의 함수들에게 연결되어 있는데 그냥 직접 함수를 호출하는 방식이 더 간편하다고 생각된다.

> 💡 나중에 TakeItem을 컴포넌트화 시켜 ABCharacterBase 외부 클래스로 만들면효율적일것 같다.

---

[3] 소프트 레퍼런싱

지금까지 멤버변수는 `TObjectPtr`로 선언했다. 이럴 경우, 액터가 생성될 때 **자동으로 메모리에 생성**된다. 이것을 **'하드 레퍼런싱'**이라고 한다.

그러나 맵 로딩과 동시에 메모리에 생성되기에 하드 레퍼런싱을 남용하는 것은 메모리 사용량에 부담이 된다.

그래서 필요는 하되 곧바로 사용하지 않는 멤버변수는 `TSoftObjectPtr`로 선언하는 것이 좋다.

이것을 **'소프트 레퍼런싱'** 이라고 한다.

애셋 대신에 애셋주소 문자열이 지정되며, 우리가 필요할 때 애셋을 로딩해온다.

사용법

사용에 주의할 것은, 해당 멤버변수는 말그대로 필요할 때 로드가 되기 때문에 일반적인 코드를 사용해서는 에러가 발생한다.

다음과 같이 해당 멤버변수(`WeaponMesh`)가 준비되어 있는지(`IsPending`) 여부를 확인한 후 안되었다면 로드해주고(`LoadSynchronous`) 사용해야 한다.

```
void AABCharacterBase::EquipWeapon(UABItemData* InItemData)
{
  UABWeaponItemData* WeaponItemData = Cast<UABWeaponItemData>(InItemData);
  if (WeaponItemData)
  {
    if (WeaponItemData->WeaponMesh.IsPending())
    {
      WeaponItemData->WeaponMesh.LoadSynchronous();
    }
    Weapon->SetSkeletalMesh(WeaponItemData->WeaponMesh.Get());
```

```
  }
}
```

📌 **Summary**

- 기믹 구현을 위한 트리거 구조의 설계
- 데이터 애셋을 확장시킨 아이템 데이터 구조 관리
- 의존성 분리를 위한 프로젝트 레이어에 대한 이해
- 메모리 최적화를 위한 소프트 레퍼런싱의 사용법

0n. 아이템 시스템 강의 과제

❓ Q1. 현재 프로젝트의 클래스를 데이터, 미들웨어, 게임의 세 가지 레이어로 한번 정리하고 이들의 의존 관계를 줄이기 위한 규칙을 만들어보시오.

- 데이터: 캐릭터의 움직임을 표현하기 위한 애니메이션 애셋. 이들은 애셋 그자체로 타 클래스에 대한 참조를 필요로 하지 않는다.
- 미들웨어: 위 애셋들을 모아서 상황에 맞는 움직임을 구성한 애니메이션 BP
  - IsFire, IsBuilding 등 캐릭터의 상태를 사용하는 경우가 있다. 이 때를 위해 PlayerCharacter의 인터페이스를 작성해 이들을 가져와야 한다.
- 게임: 플레이어 캐릭터는 위 애님BP를 가져와 움직임으로 사용한다.

❓ Q2. 현재 프로젝트에 소프트 오브젝트 레퍼런스를 도입하고, 최초 맵 로딩시 메모리 사용량을 이전과 이후로 비교해보시오.

플레이어는 무기를 가지고 몬스터에 대항한다. 하지만 이 무기는 필드에서 상점에서 구매하여 사용해야 한다. 그렇기에 게임 시작과 동시에는 필요로 하지 않는다.

그렇기에 캐릭터의 EquippedWeapon을 소프트 레퍼런싱을 사용해 구입하는 경우에 로드한다면 메모리 사용량을 절약할 수 있을 것이다.

```
Cmd: Obj List Class=SkeletalMesh
Obj List: Class=SkeletalMesh
Objects:
                                                                                                  Object      NumKB
                                                SkeletalMesh /Game/Assets/AnimStarterPack/UE4_Mannequin/Mesh/SK_Mannequin.SK_Mannequin    3965.04
                                                             SkeletalMesh /Game/Assets/EnemyMaynard/SKM_EnemyMaynard.SKM_EnemyMaynard    3500.86
                                                                                                    Class     Count      NumKB      MaxKB   ResExcKB  ResExcD
                                                                                              SkeletalMesh         2    7465.90    7466.25    2011.04
2 Objects (Total: 7.291M / Max: 7.291M / Res: 1.964M | ResDedSys: 0.048M / ResDedVid: 0.000M / ResUnknown: 1.916M)
```

사용시 무기의 메쉬는 메모리에 올라가 있지 않을 것은 확인할 수 있었다.

하지만 무기 메쉬의 양이 많지 않았기에 이에 대한 큰 체감은 느낄 수 없었다.(로딩이 짧아진다거나 하는!)

MMO장르를 목표한다면 확실히 사용되는 메모리의 양의 차이가 날 것이니, 그 때는 드라마틱한 차이를 느낄 수 있을 것이라 예상된다.