Item System

게임 개발에서 중요한 백엔드 서비스이다. DB에 저장되고 생성과 소멸이 잦고 중요한 재화이기 때문에 성능과 안정성을 만족하는 알고리즘과 구조에 기반해야 한다.

요구 사항

- 트랜잭션
  - 생성, 소멸, 이동, 거래, 속성
- 트래킹

아이템은 인벤토리에 보관된다. 아이템은 드랍, 구매, 뽑기를 통해 생성된다. 아이템은 버리기, 제련, 강화 실패로 소멸한다. 아이템은 개인 거래, 경매장, 우편을 통해 이동한다. 아이템은 강화, 소켓 장착 등으로 속성이 변경된다.

다양한 속성과 다양한 인벤토리, 다양한 동작 방식이 있고 게임 내 중요 재화이기 때문에 서비스 중 트래킹이 가능해야 한다.

개념들

아이템 타잎과 인스턴스 아이템 옵션 (속성) 아이템 소켓 인벤토리와 슬롯 동작들(operations):

- 생성(드랍, 구매, 뽑기)과 획득
- 이동(개인 거래, 경매장, 우편)과 획득 및 소멸
- 버리기와 소멸
- 강화와 속성 변경
- 소켓 장착과 속성 변경
  가치 함수
- value(item) --> R+
- 각 아이템 인스턴스의 가치를 매기는 함수
- 아이템 속성에 대한 함수 (결국, R^n에서 R^+로의 함수)

평가 기준

동작의 트랜잭션 (ACID) 동작의 응답 속도 동작의 오류 처리 동작의 안전성 (보안) 동작의 트래킹 수준

인스턴스화 (생성과 변경)

원칙들:

- 인스턴스는 고유하다
- 인스턴스는 변경되지 않는다. (생성/소멸/속성 변경만 있다)
  - 단, 개수로 관리되는 경우는 대표 인스턴스가 변경된다.
- 트래킹 가능하다.

개별 아이템이 고유한 객체일 경우가 있고 돈, 물약과 같이 개수를 갖는 경우가 있다. 개수를 갖는 경우에도 인스턴스는 고유해야 한다.

인벤토리와 슬롯

모든 아이템은 인벤토리에 있다. 인벤토리는 여러 종류가 있다. 별도 테이블을 두고 관리할 지, 아니면 한 테이블에서 관리할 지 결정이 필요하다.

여기서는 구현의 한 예인 하나의 TblItem에서 관리하는 경우를 중심으로 살펴본다.

인벤토리는 길드, 캐릭터, 계정, 우편함 등 다양한 타잎으로 나눠서 처리한다.

작은 차이의 처리

인벤토리마다 동작이 달라지는 것들이 있다. 이럴 경우 클래스를 분리하거나 함수를 분리하거나 if 문으로 나누어 처리한다. if 문으로 처리하게 되면 코드가 복잡해지고 실수가 잦아질 수 있다. 클래스를 분리하는 경우 클래스 종류가 많고 코드를 따라가기가 어려워질 수 있다. 이 중간에 함수를 사용하는 방식이 있다.

C++에서 다형성은 컴파일 시 타잎 기반, constexpr을 사용하여 컴파일 시와 실행 시 모두 적용하는 방법, 타잎 정보에 기초하여 실행 시 함수를 선택하는 방법, 클래스를 분리하여 다형성을 이용하는 방법이 있다. 각 방법의 장단점이 있다.

요청을 받아 조절하는 함수에서 인벤토리를 타잎별로 찾아서 처리하고 개별 인벤토리 타잎에 세부 구현을 갖는 방식이 중요해 보인다. 클래스 계층이 없으면 코드가 너무 복잡해지고 하는 일이 많아진다.

- InvenComp
  - 인벤토리 관련 이벤트의 처리 함수들 담당
- Inven
  - Default
  - PlatinumInven
  - EquipInven
  - RebuyInven
  - MailInven
  - ExchangeInven
  - QuestInven
  - StorageInven
  - CashInven
  - KnightageInven
  - MakingInven
- Slot
  - 인벤토리를 슬롯으로 구성된다.
  - Slot의 특성은 구성되며 하위 클래스를 갖는다.
- Item
  - 하위클래스?
- ItemBag
  - 동적인 아이템 정보 이동용 아이템 가방

위 정도를 기본 타잎으로 갖추고 다형성을 결정한다.

```
ErrorCode InvenComp::GetMakingMaterials(IndexItem item, std::size_t count, ItemList& items)
{
    // Story: MakingInven에 대해 item에 해당하는 아이템 목록을 count 만큼 가져온다.
    RETURN_IF(count == 0, ErrorCode::ItemInvalidMakingRequestCount);
    RETURN_IF(!IsValidItemIndex(item), ErrorCode::ItemInvalidItemIndex);

    // Given: 재질 아이템이 제작 인벤에 count 만큼 있을 경우
    // Do: 제작 인벤들에서 해당 아이템들을 count만큼 찾아서 items에 넣는다.
    // Then: items에 count 개수 만큼 아이템이 들어 있다.

    int required = count; // req

    // Do의 구현 :
    // - 각 제작 인벤에 대해
    for ( auto& inven : GetMakingInvenIterator() )
    {
        // - 제작 재료를 필요한 만큼 가져온다.
        required -= inven->FindItemsRequired( item, required, items );
        // - 해당 재료는 items에 넣는다.

        if ( required <= 0 ) // 다 채웠으면 완료
        {
            return ErrorCode::Success;
        }
    }

    return ErrorCode::ItemInsufficientMakingItems;
}
```

GetMakingInvenIterator()는 shared_ptr 형태로 관리되는 inven 포인터들에 대해 vector로 갖고 있도록 구현하면 쉽게 구현할 수 있다.

FindItemsRequired() 함수는 Inven의 기본 기능으로 구현할 수 있다.

Inven이 클래스가 아니고 컴포넌트에서만 처리할 경우 재사용 가능한 함수가 누적되지 않는다. 컴포넌트는 인벤토리들을 대상으로 동작하기 때문에 개별 인벤토리 처리 함수를 넣으면 함수 개수가 많아져서 부담이 생기기 때문에 서술형으로 쭉 적는 경향이 생긴다.

Item은 돌아다니기 때문에 shared_ptr로 관리해야 한다. inven은 검색이나 모으기를 위해 shared_ptr로 관리되는 것이 좋다. GetMakingInvenIterator() 구현이나 여러 개의 인벤을 만들기 위해 필요하다.

트랜잭션

동작의 트랜잭션을 관리하는 것이 트래킹이다. 트래킹은 accounting이다. accounting의 대상을 판별하는 식별자와 트랜잭션의 구성 요소가 핵심이다. 식별자는 인스턴스이며 컴퓨터에서는 고유한 키이다.

트랜잭션은 양이 많아 DB에 저장하기는 어렵고 로그로 처리해야 한다. 기업마다 쌓여있던 많은 양의 회계 장부를 생각하면 그 보다 훨씬 많은 트랜잭션이 생산되는 게임에서는 어쩔 수 없는 선택으로 보인다.

로그로 쌓더라도 DB처럼 검색이 가능해야 한다. 기간별 회계 장부를 보듯이.

여러 아이템의 처리

아이템의 분해와 조합 등으로 여러 아이템에 걸친 동작이 발생할 경우 개발 아이템에 대한 처리를 묶어서 해야 한다. 이런 경우를 위해 개별 아이템에 대한 DB 처리와 결과 처리를 묶어 동작을 구성하고 처리하는 방식이 괜찮아 보인다.

ItemTransaction의 타잎별 구현을 하고 DB 처리와 게임 처리를 동시에 묶는다. ItemTransaction을 개별적으로 처리하는 내용은 ItemProc으로 만들고 ItemProc은 DB 처리와 게임 처리를 위한 함수를 갖는다. 개별적으로 DB에 대한 처리, 게임에 대한 처리를 Commit, Rollback으로 한다.

비동기로 처리되므로 같은 아이템에 대한 여러 요청이 올 수 있어 일단 게임에서 플래그로 Lock을 걸고 (모든 동작 안 됨, 인벤토리 슬롯 이동도 안 됨) 결과가 오면 Commit 후 락을 푸는 방식으로 동작 가능해 보인다.

클래스

ItemTransaction

- Create()
- Execute()
- Commit()
- Rollback()

ItemExtractTransaction, ItemTradeTransaction 등 DB에서 트랜잭션으로 처리해야 할 단위들로 하위 클래스로 만들고 Create()에서 ItemProc들을 추가하고 Execute에서 ItemProc::Execute()하고 Commit()에서 게임 쪽 반영하며 Rollback()에서 Item Lock을 푼다.

ItemProc

- Create()
- Execute()
- Commit()
- Rollback()

개별 아이템에 대한 생성, 삭제, 변경 Proc만 있으면 된다.

정리

위의 아이디어에 기초해서 개념화를 더 진행해야 한다. 코드에서 Lock이 걸린 아이템은 사용하지 못 하도록 인터페이스나 체크가 쉬워야 한다. 임시 보관소로 옮겨서 보이지 않도록 하는 방법이 가장 안전해 보인다.