OpenZepplin

태그	ERC

- 태그: 작성하려는 글에 맞거나 연관되어 있는 태그를 선택합니다.
- 상태: 해당 글의 상태를 의미합니다.
  - 글을 쓸 예정으로 등록했다면 `작성 예정`, 글을 쓰는 중이거나 아직 완료한 것 같지 않다면 `작성 중`, 완료 후에는 `완료`로 표기합니다.

⚠가독성을 위해 개요와 결론을 작성하시는 것을 추천하지만, 느낌에 따라 자유롭게 작성하시면 됩니다!

개요

ERC-20에 코드구현을 위해 정보를 찾아보던 도중 OpenZepplin이라는 프레임워크를 찾았다.

본론

오픈제플린(OpenZepelin)이란

오픈제플린(OpenZeppelin)은 솔리디티 기반의 스마트 컨트랙트를 개발할 수 있는 표준 프레임워크입니다. 스크래치로 ERC20 토큰을 작성할 수는 있으나, 테스트를 거친 템플릿을 사용하여 더 안전하고 효율적인 컨트랙트를 빠르게 구축할 수 있습니다.

오픈제플린을 이용한 ERC-20

오픈제플린 ERC20은 다음과 같이 코드가 구성되어 있습니다.

- `IERC20`: 모든 ERC20 구현이 준수해야하는 인터페이스
- `IERC20Metadata`: `name`, `symbol`, `decimals` 등의 설정
- `ERC20`: ERC20의 실제 구현체

IERC20(OZ)

IERC는 ERC-20에 필요한 함수 등 구현해야할 목록으로 생각하면 됩니다. ERC20을 위해서는 다음 함수가 구현되어야 합니다. 즉, 거래소에 상장된 토큰은 아래의 함수가 모두 구현되어 있습니다.

- `totalSupply()`: **전체 공급량**. 전체 발행된 토큰양 반환
- `balanceOf(account)`: **잔고**. 특정 주소가 소유한 토큰 양 반환
- `approve(spender, amount)`: **승인**. 다른 사람에게 토큰 사용 허용. 토큰의 양을 확인하여 거래량 제한
- `allowance(owner, spender)`: **허용**. `owner`가 `spender`에게 허락한 토큰 개수 확인
- `transfer(to, amount)`: **전송.** 함수 호출 주소에서 개인 계정으로 송금
- `transferFrom(sender, recipient, amount)`: **대리 전송.** `approve` 이후 사용자 간 송금 제공

IERC2MetaData(OZ)

`IERC20MetaData`에는 ERC-20의 `name`, `symbol`, `decimals` 함수와 관련한 인터페이스입니다.

- `name()` : 토큰의 이름.
- `symbol()` : 토큰의 심볼. (이름의 축약어로 이해할 수 있음)
- `decimals()` : 소수점 단위 설정.

일반적으로 decimal은 18로 설정합니다. 이더(ether)와 단위와 같으며 대다수 이제 표준처럼 사용하고 있습니다.

ERC20(OZ)

ERC20 함수는 6개이지만 실제 코드에서 핵심적인 내용은 변수 3개와 함수 2개에 담겨있습니다.

변수

```
mapping(address => uint256) private _balances;

mapping(address => mapping(address => uint256)) private _allowances;

uint private _totalSupply;
```

맵핑(`mapping`) 변수는 C++에서 `map`, Java에서 `HashMap`, Python에서 `dict`, JavaScript에서 `Map` 의 역할을 하는 솔리디티 변수입니다.

> 💡 Solidity의 mapping은 해시맵을 사용합니다. 해시에 사용하는 단방향 함수는 `keccak256`입니다.

Solidity의 mapping은 해시맵을 사용합니다. 해시에 사용하는 단방향 함수는 `keccak256`입니다.

`_balances`는 주소와 각 주소가 보유한 토큰량을 쌍으로 들고 있는 변수이며, `_allowances`는 주소 A의 토큰을 주소 B가 대리 사용가능하며 이때 사용 가능한 토큰량을 저장하고 있는 변수입니다. 그리고 `_totalSupply`는 전체 발행된 토큰량을 저장합니다.

함수

- `_approve(owner, spender, amount)`: 승인 함수

```
function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}

/*
코드를 한 줄 씩 순서대로 살펴보면 다음과 같습니다.

보내는 주소와 받는 주소가 zero address인지 체크합니다.
_allowances에 owner- spender 매핑에 amount 값을 갱신합니다.
즉, 토큰 소유주-토큰 가용자 매칭과 가용 토큰양을 기록해둡니다.
Approval 이벤트를 발생시킵니다. (로그 기록)
*/
```

- `_transfer(from, to, amount)`: 전송 함수.

```
function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {
    require(from != address(0), "ERC20: transfer from the zero address");
    require(to != address(0), "ERC20: transfer to the zero address");
```

```
    _beforeTokenTransfer(from, to, amount);

    uint256 fromBalance = _balances[from];
    require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
    unchecked {
        _balances[from] = fromBalance - amount;
    }
    _balances[to] += amount;

    emit Transfer(from, to, amount);

    _afterTokenTransfer(from, to, amount);
}

/*
코드를 한 줄 씩 순서대로 살펴보면 다음과 같습니다.

보내는 주소와 받는 주소가 zero address인지 체크합니다.
_beforeTokenTransfer와 _afterTokenTransfer는
토큰 전송 이전과 이후 추가적인 내용을 작성할 수 있는 선택적 함수입니다. 생략해도 무방합니다.
from의 잔액을 불러옵니다.
잔액이 amount보다 많은지 체크합니다.
from에서는 보내는 양을 빼주고, to에 그만큼 더해줍니다.
Transfer 이벤트를 발생시킵니다. (로그 기록)
*/
```

결국 간단하게 설명하면 잔액 체크 후, 보내는 계좌에서 빼고 받는 계좌에서 더해주는 간단한 코드입니다.

이제 위 변수와 함수를 사용하면 ERC-20에 필요한 기능을 모두 구현할 수 있습니다.

- `totalSupply()`: `_totalSupply` 변수 사용
- `balanceOf()`: `_balances` 변수 사용
- `approve()`: `_approve` 함수 사용
- `allowance()`: `_allowances` 사용
- `transfer()`: `_transfer` 사용
- `transferFrom()`: (1) `allowance` 함수로 체크후, (2) `_approve` 함수 갱신, (3) `_transfer` 함수 사용

간단하게 적긴 했으나, 내부에는 여러 `require`문이 존재합니다. 내부 코드 점검에는 크게 3가지 포인트만 잊지 않으면 됩니다.

- 잔액과 전송되는 토큰량이 음수가 되게 하지 말 것.
- 데이터가 변경될 때, 연관되는 데이터를 모두 고려할 것.
- 불필요한 연산은 최소화 시킬 것. (예시. `_transfer`에서 `잔액>전송량`을 `require`로 체크했으므로 마이너스 연산에 `unchecked` 키워드를 사용하여 연산 효율성 증가)

추가구현

그렇다면 꼭 위에 정해진 함수만 작성해야할까요? 정답부터 말하면 그렇지 않습니다. 위 기능은 기본 기능이고, 토큰의 목적에 따라 다양한 함수를 추가적으로 구현할 수 있습니다. 오픈제플린에는 4개의 함수가 추가적으로 구현되어 있습니다.

- `increaseAllowance()` / `decreaseAllowance()`: `allowance` 양 증감
- `_mint`: 토큰 민팅. `_totalSupply`와 `_balances[owner]`에 토큰 동시 증가
- `_burn`: 토큰 소각. `_totalSupply`와 `_balances[owner]`에 토큰 동시 감소

ERC20 Extensions

오픈제플린에서도 ERC-20에 여러 추가 기능을 구현한 템플릿이 일부 존재합니다.

- `ERC20Burnable`: 토큰 소각과 대리 소각 가능
- `ERC20Capped`: 총 공급량 한정
- `ERC20Pausable`: 토큰 전송 일시 중지 가능
- `ERC20Snapshot`: 과거 상태를 저장하는 스냅샷 기능
- `ERC20Votes`: 투표 및 투표 위임 기능
- `ERC20VotesComp`: 투표 및 투표 위임 기능(Compound와 호환)
- `ERC20Wrapper`: 서로 다른 종류의 토큰으로 변환(민팅 및 소각 사용)
- `ERC20FlashMint`: 플래시론 지원

결론

OpenZeppelin을 사용하면 기본적인 ERC-20에 맞는 프로토콜 기능들은 제공해주나 추가 기능을 만들고 싶다면 다른 Extension을 사용하거나 직접 추가해서 구현해야함

참고

ERC-20 코드 살펴보기(feat. OpenZeppelin)
본 글은 OpenZeppelin 4.x 버전을 기반으로 작성되었습니다. ERC-20은 이더리움 블록체인 네트워크에서 정한 토큰의 표준 규격(세부적인 내용은 이전 포스팅 참고)을 의미합니다. ERC-20이 제공하는 기능 중 일부는 다음과 같습니다. * 한 계정에서 다른 계정으로 토큰 전송 * 계정의 현재 잔여 토큰(잔액) 확인 * 계정 내 토큰
https://ansubin.com/erc20/

ERC20 - OpenZeppelin Docs
An ERC20 token contract keeps track of fungible tokens: any one token is exactly equal to any other token; no tokens have special rights or behavior associated with them. This makes ERC20 tokens useful for things like a medium of exchange currency, voting rights, staking, and more.
https://docs.openzeppelin.com/contracts/4.x/erc20?ref=ansubin.com

OpenZeppelin
The standard for secure blockchain applications
https://www.openzeppelin.com/?ref=ansubin.com

openzeppelin-contracts/contracts/token/ERC20 at master · OpenZeppelin/openzeppelin-contracts
OpenZeppelin Contracts is a library for secure smart contract development. - OpenZeppelin/openzeppelin-contracts
https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token/ERC20?ref=ansubin.com