ERC20_Docs

Upside Academy 1기 nullorm (정효영)

▼ 목차

ERC20

About ERC20

ERC20에는 두 개의 event와 여섯 개의 메서드가 필요하다.

```
function name() public view returns (string) // optional
function symbol() public view returns (string) // optional
function decimals() public view returns (uint8) // optional

function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uin
t256 balance)
function transfer(address _to, uint256 _value) public retur
ns (bool success)
function transferFrom(address _from, address _to, uint256 _
value) public returns (bool success)
function approve(address _spender, uint256 _value) public r
eturns (bool)
function allowance(address _owner, address _spender)public
view returns(uint256)
```

```
event Transfer(address indexed _from, address indexed _to,
uint256 _value)
event Approval(address indexed _owner, address indexed _spe
nder, uint256 _value)
```

이러한 메서드와 이벤트들을 기반으로 ERC20은 대체가능 Token(↔ NFT) 에 관한 기본적인 사항들을 정의한다. 이를 통해 여러 다른 제품 및 서비스에서 상호 운용 가능한 토큰 application을 구축할 수 있도록 한다.

constructor

```
constructor(string memory name_, string memory symbol_) EIP712(name_, "1") {
    _name = name_;
    _symbol = symbol_;
    _mint(msg.sender, 100 ether);
}
```

ERC20에서는 name과 symbol을 정해준다.

그리고, testcode에서, 두 계정에 각 50 ether씩 토큰을 주는 과정이 포함되어있기 때문에, msg.sender에게 100 ether만큼의 토큰을 mint해준다.

Mappings / Variables

```
uint256 private _totalSupply;
string private _name;
string private _symbol;
mapping(address => uint256) private _balances;
mapping(address => mapping(address => uint256)) private _al
lowances;
```

`_totalSupply, _name, _symbol` 은 각각 토큰의 총 발행량, 이름, 심볼을 저장하는 변수이다.

- `_balances` : address가 가지고 있는 토큰의 총 수량을 나타낸다.
- `_allowance` : owner가 spender에게 얼마만큼의 토큰을 approve했는지를 저장하는 매핑.

Getter Functions

```
function symbol() public view virtual returns (string memor
y) {
    return _symbol;
}

function decimals() public view virtual returns (uint8) {
    return 18;
}

function totalSupply() public view virtual returns (uint25
6) {
    return _totalSupply;
}

function balanceOf(address account) public view virtual ret
urns (uint256) {
    return _balances[account];
}

function allowance(address owner, address spender) public v
iew virtual returns (uint256) {
    return _allowances[owner][spender];
}
```

위에서 소개했던 매핑, 변수들의 값을 return해주는 getter function들이다.

Main Executable Functions

```
function transfer(address to, uint256 value) public virtual
whenNotPaused() returns (bool) {
    address owner = msg.sender;
    _transfer(owner, to, value);
    return true;
}

function transferFrom(address from, address to, uint256 val
ue) public virtual whenNotPaused() returns (bool) {
```

```
    address spender = msg.sender;
    _spendAllowance(from, spender, value); // allowance를 소
비하여 transferFrom을 함
    _transfer(from, to, value);
    return true;
}

function approve(address spender, uint256 value) public vir
tual whenNotPaused() returns (bool) {
    address owner = msg.sender;
    _approve(owner, spender, value);
    return true;
}
```

- `transfer`: `token`의 `owner`(`msg.sender`)가 `to`에게 `value`만큼의 토큰을 전송해준다.
- `transferFrom`: 제3자(`spender`)가 `from`의 `token`을 `to`에게 transfer한다.
- `approve`: `spender`가 `transferFrom`을 할 수 있도록 `approve`하는 함수.

Internal Functions

```
function _transfer(address from, address to, uint256 value)
internal {
    require(from != address(0), "Invalid Sender");
    require(to != address(0), "Invalid Receiver");
    _update(from, to, value);
}

function _approve(address owner, address spender, uint256 v
alue) internal virtual {
    require(owner != address(0), "Invalid Approver");
    require(spender != address(0), "Invalid Spender");
    _allowances[owner][spender] = value;
}

function _update(address from, address to, uint256 value) i
nternal {
    if (from == address(0)) { // mint
```

```
            _totalSupply += value;
        }
        else
        {
            uint256 fromBalance = _balances[from];
            require(fromBalance >= value, "Insufficient Balanc
e");
            unchecked { // require에서 검사했기 때문에, unchecked로
해도 됨
                _balances[from] = fromBalance - value;
            }
        }

        if (to == address(0)) { // burn
            unchecked {
             // value가 totalsupply보다 작거나 같기 때문에, 오버플로우
안일어남.
                _totalSupply -= value;
            }
        } else {
            unchecked {
             // balance에서 value를 더해도 totalsupply보다 작거나 같
기 때문에, 오버플로우 안일어남
                _balances[to] += value;
            }
        }
    }
```

- `_transfer` : `from` 과 `to` 의 잔액을 `update` 해줌
- `_approve` : `_allowance` 매핑을 업데이트하여 `approve` 한 것을 기록
- `_update` : 모든 토큰 수량 관련 연산이 이루어지는 함수

  `from` 주소가 0이라면 `mint` 가 되어 totalSupply를 올려주고,

  `to` 주소가 0이라면 `burn` 되어 totalSupply를 빼준다.

  둘 모두 0이 아니라면, `_balances` 매핑을 업데이트 해준다.

```
function _mint(address account, uint256 value) internal {
    require(account != address(0), "Invalid Receiver");
    _update(address(0), account, value);
}

function _burn(address account, uint256 value) internal {
    require(account != address(0), "Invalud Sender");
    _update(account, address(0), value);
}
```

- `_mint` : 토큰을 새로 발행
- `_burn` : 토큰을 소각

```
function _spendAllowance(address owner, address spender, ui
nt256 value) internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    require(currentAllowance >= value, "Insufficient Allowa
nce");
    _approve(owner, spender, currentAllowance - value);
}
// currentAllowance가 type(uint).max일 경우에 spend를 하지 않도
록 되어있는데,
// gas 소모 때문에 그렇게 되어있던 것 같다.
```

- 이 함수는 `transferFrom()`에서 `approve`를 통해 `allowance`된 수량을 이용한 뒤에 사용한 만큼의 `allowance`를 깎아주는 함수이다.

ERC20Pausable

말그대로 컨트랙트의 기능을 멈추도록 하는 기능을 제공한다. openzeppelin의 구현체를 보고 참고하였다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ERC20Pausable {
    bool private _paused;
```

```
    address private owner;
    modifier onlyOwner() {
        require(msg.sender == owner, "not owner");
        _;
    }
    modifier whenNotPaused() {
        _requireNotPaused();
        _;
    }
    modifier whenPaused() {
        _requirePaused();
        _;
    }
    function paused() public view virtual returns (bool) {
        return _paused;
    }
    function _requireNotPaused() internal view virtual {
        require(!paused(), "Expect not paused");
    }
    function _requirePaused() internal view virtual {
        require(paused(), "Expect paused");
    }
    function pause() public virtual whenNotPaused onlyOwner
() {
        _paused = true;
    }
    function unpause() public virtual whenPaused onlyOwner
() {
        _paused = false;
    }
}
```

- testcode에서 owner가 아니라면 pause를 하지 못하도록 하는 테스트가 있는데, 그것 때문에, onlyOwner modifier를 만들어주었다.
- pause기능은 whanPau`s`ed와 `whenNotPaused` modifier를 통해 구현되는데, 이는 `_paused`변수가 true인지, false인지에 따라 작동하게 된다. `_paused` 변수가 true라면, `whenNotPaused` modifier가 붙은 함수는 작동하지 않게 된다.

이 때문에, 위의 ERC20함수를 보면, `whenNotPaused()` modifier가 붙어있는 것을 볼 수 있다.

ERC20Permit

Permit은 오프체인에서 approve에 대한 서명을 전달함으로써 토큰의 owner가 직접 approve를 하지 않고, 서명을 전달해주면 이를 받은 서명자가 대신 approve를 해주는 기능을 제공한다.

이런식으로 오프체인 등에서 "미리 서명된 데이터"를 이용하는 것은 이미 블록체인 상에서 널리 사용되고있는 방식이라고 한다. 따라서 이후에도 분명 언젠가 활용할 일이 있을 것이기 때문에, 깊게 공부해보려고 한다.

여기서 주의할 점은, 아무 다른 토큰에서 `permit`한 서명을 이용해 다른 토큰에서도 이용할 수 있으면 안되기 때문에, 이를 겹치지 않도록 조절 해주어야 한다.

```
bytes32 private constant PERMIT_TYPEHASH = keccak256("Permi
t(address owner,address spender,uint256 value,uint256 nonc
e,uint256 deadline)");
```

이 코드를 통해 PERMIT_TYPEHASH를 만들고,

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public virtual {
    require(block.timestamp <= deadline, "Expired Signatur
e");
    bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHA
SH, owner, spender, value, _useNonce(owner), deadline));
    bytes32 hash = _hashTypedDataV4(structHash);
    address signer = ECDSA.recover(hash, v, r, s);
    require(signer == owner, "INVALID_SIGNER");
```

```
        _approve(owner, spender, value);
    }
```

이러한 방식으로 permit()이 구현되어있는 컨트랙트에 서명값을 전달할 수 있다.

1. deadline을 명시해주어야 한다.

   동일 종류의 토큰에 대하여 deadline이 지나기 전에 permit을 한번 더 실행할 수 있게 된다면, owner는 permit을 두 번 실행시킨 꼴이 될 것이다.

2. structHash를 통해 owner가 지정한 spender, block.timestamp, nonce, deadline이 맞는지 확인하기 위해 해싱을 한다. (서명 과정에서도 이 값들에 대하여 hashing을 한 값에 대해 서명을 한다.)

3. 위에서 언급한 것 처럼, 아무 다른 토큰에서 permit한 서명을 이용해 다른 토큰에서도 이용할 수 있으면 안되기 때문에, 토큰의 정보에 대한 메타데이터 또한 같이 해싱을 해주는 `_hashTypedDataV4`를 실행해준다.

   이는 openzeppelin의 `EIP712.sol`을 사용하였다.

4. 이러한 hash와 v, r, s를 이용하여 서명 생성자의 주소를 `ecrecover` 한다.

5. 이 주소가 owner(서명 생성자)의 주소와 같은지를 검사한다.

6. `approve`를 해준다.

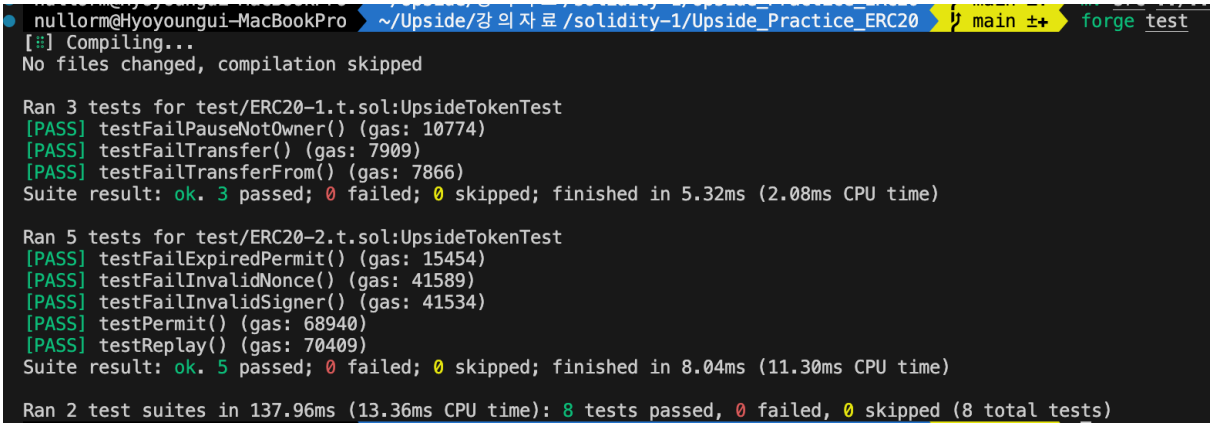
이러한 과정을 구현하는 데 초점을 맞추었다.

(+) EIP712: Typed structured data hashing and signing

EIP712는 구조화된 데이터의 해싱 및 서명을 위한 표준이 정의되어있다.

하지만, 일단 제출이 우선이므로 제출을 한 후에 추가적인 공부를 진행해보도록 해야겠다.

Test



test들을 모두 잘 통과한 것을 볼 수 있다.