Solidity智能合约开发知识

第5.1课：基础项目 - 简单代币合约

学习目标：理解ERC20代币标准、掌握代币合约的完整实现、学会授权机制的使用、能够部署和测试代币合约

预计学习时间：3小时

难度等级：基础实战

目录

---

1. ERC20代币标准概述

1.1 什么是ERC20

ERC20是以太坊上最广泛使用的代币标准。ERC20的全称是**Ethereum Request for Comment 20**，这是2015年由Fabian Vogelsteller提出的一个标准提案。

基本定义：

ERC20定义了可替代代币（Fungible Token）的统一接口。可替代代币意味着每个代币都是相同的，就像人民币一样，一张100元纸币和另一张100元纸币价值完全相同，可以互换。

规模和影响：

- 以太坊上已有超过50万个ERC20代币
- 日交易量超过百亿美元
- 99%的代币项目使用ERC20标准
- 几乎所有DeFi协议都基于ERC20

著名的ERC20代币：

代币	符号	类型	说明
Tether	USDT	稳定币	市值最大的稳定币
USD Coin	USDC	稳定币	Circle发行的美元稳定币
DAI	DAI	稳定币	MakerDAO的去中心化稳定币
Chainlink	LINK	功能代币	预言机网络代币
Uniswap	UNI	治理代币	Uniswap的治理代币

1.2 为什么需要代币标准

没有标准的问题：

如果每个代币项目都自己定义接口，会导致严重问题：

1. **钱包兼容性差**：无法统一支持所有代币
2. **交易所集成困难**：需要为每个代币定制代码
3. **DeFi协议无法通用**：不能通用处理不同代币
4. **开发成本高**：开发者需要学习各种不同接口
5. **用户体验差**：每个应用的操作方式都不同

有了ERC20标准的好处：

1. **统一接口**：一套代码支持所有ERC20代币
2. **无缝集成**：钱包、交易所、DApp都能自动支持
3. **降低成本**：开发和集成成本大幅降低
4. **用户体验好**：操作方式统一
5. **互操作性强**：不同应用和协议可以协同工作

这就是标准化的价值所在！

1.3 可替代代币 vs 不可替代代币

可替代代币（Fungible Token - ERC20）：

```
// 所有代币都相同
Alice的100个MTK = Bob的100个MTK
可以互换，价值相同
```

特点：

- 每个代币相同
- 可以分割
- 用于货币、积分、权益
- 标准：ERC20

不可替代代币（Non-Fungible Token - ERC721）：

```
// 每个代币都独特
Alice的NFT #1 ≠ Bob的NFT #2
不可互换，各有特点
```

特点：

- 每个代币唯一
- 不可分割
- 用于艺术品、收藏品、资产证明
- 标准：ERC721

1.4 ERC20标准的组成

ERC20标准定义了：

1. **6个核心函数**
    - 查询函数：totalSupply、balanceOf、allowance
    - 操作函数：transfer、approve、transferFrom
2. **2个必需事件**
    - Transfer事件：记录转账
    - Approval事件：记录授权
3. **3个可选元数据**
    - name：代币名称
    - symbol：代币符号
    - decimals：小数位数

---

2. ERC20核心函数

2.1 查询类函数

查询类函数不会改变合约状态，所以被标记为`view`。

totalSupply - 查询总供应量

函数签名：

```
function totalSupply() public view returns (uint256)
```

功能：返回代币的总发行量

示例：

```
contract TokenExample {
    uint256 public totalSupply;

    function totalSupply() public view returns (uint256) {
        return totalSupply;
    }
}

// 调用示例
uint256 supply = token.totalSupply();
// 如果总发行量是100万个，返回：1000000
```

使用场景：

- 查看代币总量
- 计算市值（总量 × 价格）
- 通缩代币查看剩余供应
- DeFi协议计算占比

balanceOf - 查询余额

函数签名：

```
function balanceOf(address account) public view returns (uint256)
```

功能：返回指定地址的代币余额

参数：

- `account`：要查询的账户地址

返回：该地址持有的代币数量

示例：

```
contract BalanceExample {
    mapping(address => uint256) public balanceOf;

    function balanceOf(address account) public view returns (uint256) {
        return balanceOf[account];
    }
}

// 调用示例
uint256 balance = token.balanceOf(aliceAddress);
// Alice有500个代币，返回：500
// 如果账户没有代币，返回：0
```

使用场景：

- 钱包显示余额
- 检查是否有足够代币

- 验证转账后余额
- DApp显示用户资产

allowance - 查询授权额度

函数签名：

```
function allowance(address owner, address spender) public view returns (uint256)
```

功能：返回被授权人可以使用授权人的代币数量

参数：

- `owner`：授权人的地址（代币所有者）
- `spender`：被授权人的地址（被允许使用代币的地址）

返回：授权的代币数量

示例：

```
contract AllowanceExample {
    mapping(address => mapping(address => uint256)) public allowance;

    function allowance(address owner, address spender)
        public view returns (uint256)
    {
        return allowance[owner][spender];
    }
}

// 调用示例
uint256 allowed = token.allowance(aliceAddress, uniswapAddress);
// Alice授权Uniswap使用50个代币，返回：50
// 如果没有授权，返回：0
```

使用场景：

- 检查剩余授权额度
- DApp显示授权状态
- 判断是否需要重新授权

2.2 操作类函数

操作类函数会改变合约状态，不能标记为`view`或`pure`。

transfer - 直接转账

函数签名：

```
function transfer(address to, uint256 amount) public returns (bool)
```

功能：从调用者账户转移代币到指定地址

参数：

- `to`：接收地址
- `amount`：转账数量

返回：`true`表示成功

示例：

```
function transfer(address to, uint256 amount) public returns (bool) {
    require(to != address(0), "Cannot transfer to zero address");
    require(balanceOf[msg.sender] >= amount, "Insufficient balance");

    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;

    emit Transfer(msg.sender, to, amount);

    return true;
}
```

执行流程：

```
Alice调用 transfer(Bob, 100)
    ↓
检查：Bob地址有效？ ✓
    ↓
检查：Alice余额 >= 100？ ✓
    ↓
Alice余额 -= 100
Bob余额 += 100
    ↓
触发Transfer事件
    ↓
返回true
```

approve - 授权

函数签名：

```
function approve(address spender, uint256 amount) public returns (bool)
```

功能：授权指定地址使用调用者的代币

参数：

- `spender`：被授权人地址
- `amount`：授权数量

返回：`true`表示成功

示例：

```
function approve(address spender, uint256 amount) public returns (bool) {
    require(spender != address(0), "Cannot approve zero address");

    allowance[msg.sender][spender] = amount;

    emit Approval(msg.sender, spender, amount);

    return true;
}
```

重要特性：

1. **覆盖式授权**：新授权会覆盖旧授权
2. **不转移代币**：只设置额度，代币还在原账户
3. **可以取消**：设置为0即可取消授权
4. **需要谨慎**：授权给恶意合约会丢失代币

transferFrom - 授权转账

函数签名：

```
function transferFrom(address from, address to, uint256 amount) public returns (bool)
```

功能：使用授权额度，从授权人账户转移代币到指定地址

参数：

- `from`：代币所有者（授权人）
- `to`：接收者
- `amount`：转账数量

返回：`true`表示成功

示例：

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public returns (bool) {
    require(from != address(0), "From zero");
    require(to != address(0), "To zero");
    require(balanceOf[from] >= amount, "Insufficient balance");
    require(allowance[from][msg.sender] >= amount, "Insufficient allowance");

    balanceOf[from] -= amount;
    balanceOf[to] += amount;
    allowance[from][msg.sender] -= amount;

    emit Transfer(from, to, amount);
```

```
    return true;
}
```

执行流程：

```
Uniswap调用 transferFrom(Alice, Pool, 300)
    ↓
检查：Alice和Pool地址有效？ ✓
    ↓
检查：Alice余额 >= 300？ ✓
    ↓
检查：allowance[Alice][Uniswap] >= 300？ ✓
    ↓
Alice余额 -= 300
Pool余额 += 300
allowance[Alice][Uniswap] -= 300
    ↓
触发Transfer事件
    ↓
返回true
```

2.3 必需事件

Transfer事件

事件定义：

```
event Transfer(address indexed from, address indexed to, uint256 value);
```

参数：

- `from`：发送方地址（indexed）
- `to`：接收方地址（indexed）
- `value`：转账数量

触发时机：

1. 直接转账（transfer）
2. 授权转账（transferFrom）
3. 铸造代币（from = address(0)）
4. 销毁代币（to = address(0)）

indexed的作用：

```
// indexed参数可以被索引和过滤
event Transfer(address indexed from, address indexed to, uint256 value);

// 查询示例（使用Web3.js）
// 查询所有发送给Alice的转账
const events = await contract.getPastEvents('Transfer', {
    filter: { to: aliceAddress },
    fromBlock: 0
});
```

为什么使用indexed?

- 提高查询效率
- 区块链浏览器可以按地址过滤
- 钱包可以监听特定地址的事件
- DApp可以跟踪用户的交易历史

Approval事件

事件定义：

```
event Approval(address indexed owner, address indexed spender, uint256 value);
```

参数：

- `owner`：授权人地址（indexed）
- `spender`：被授权人地址（indexed）
- `value`：授权数量

触发时机：

调用approve函数时触发

使用场景：

```
// DApp监听授权事件
contract.on('Approval', (owner, spender, value) => {
    console.log(`${owner} approved ${spender} to spend ${value}`);
});
```

---

3. 授权机制详解

3.1 授权机制的工作原理

授权机制（Approval Mechanism）是ERC20标准的核心设计，也是很多初学者容易困惑的地方。

为什么需要授权机制?

问题场景：

Alice想在Uniswap上用USDT购买ETH。这个过程中，Uniswap合约需要获取Alice的USDT。

为什么不能用transfer?

```
// transfer只能由代币持有者自己调用
function transfer(address to, uint256 amount) public returns (bool) {
    // msg.sender必须是代币持有者
    balanceOf[msg.sender] -= amount;
    // ...
}

// Uniswap合约无法调用Alice的transfer
// 因为msg.sender会是Uniswap合约地址，不是Alice
```

问题的核心：

智能合约无法主动获取用户的代币。如果没有授权机制，合约就无法代表用户操作代币。

授权机制的解决方案：

1. 用户主动授权合约
2. 合约代表用户操作
3. 用户通过控制授权额度保持控制权

这是一种**委托代理模式**。

3.2 授权流程详解

让我们通过一个完整的场景来理解授权机制。

场景：Alice在Uniswap用USDT购买ETH

步骤1：Alice授权Uniswap

```
// Alice调用USDT合约的approve函数
usdt.approve(uniswapAddress, 1000);

执行过程：
1. allowance[Alice][Uniswap] = 1000
2. 触发Approval事件
3. 返回true

状态变化：
- Alice的USDT余额：不变（仍然是2000）
- Uniswap的授权额度：1000
- 代币位置：仍在Alice账户中
```

关键点：approve只是设置授权额度，并不转移代币！

步骤2：Uniswap使用授权

```
// Uniswap合约调用transferFrom
```

```
usdt.transferFrom(Alice, Pool, 500);

执行过程：
1. 检查：Alice余额 >= 500？ ✓（2000 >= 500）
2. 检查：allowance[Alice][Uniswap] >= 500？ ✓（1000 >= 500）
3. Alice余额 -= 500（2000 → 1500）
4. Pool余额 += 500
5. allowance[Alice][Uniswap] -= 500（1000 → 500）
6. 触发Transfer事件
7. 返回true

最终状态：
- Alice的USDT余额：1500（减少了500）
- Pool的USDT余额：500（增加了500）
- 剩余授权额度：500（被消耗了500）
```

关键点：授权额度会被消耗，不是一次性使用全部！

完整流程图：

```
初始状态：
Alice余额：2000 USDT
Pool余额：0 USDT
授权额度：0

    ↓ approve(Uniswap, 1000)

状态1：
Alice余额：2000 USDT
Pool余额：0 USDT
授权额度：1000 ←（已授权）

    ↓ transferFrom(Alice, Pool, 500)

最终状态：
Alice余额：1500 USDT ←（减少500）
Pool余额：500 USDT ←（增加500）
授权额度：500 ←（消耗500）
```

3.3 授权机制的实际应用

应用场景1：去中心化交易所（Uniswap）

```
用户想用USDT买ETH：
1. 用户授权Uniswap使用USDT
2. Uniswap调用transferFrom从用户账户取USDT
3. Uniswap给用户发送ETH
```

应用场景2：流动性挖矿

```
用户想质押代币挖矿：
1. 用户授权挖矿合约使用代币
2. 挖矿合约调用transferFrom锁定用户代币
3. 用户获得挖矿奖励
```

应用场景3：NFT购买

```
用户想用USDT购买NFT：
1. 用户授权NFT市场合约使用USDT
2. 市场合约调用transferFrom扣除USDT
3. 市场合约转移NFT给用户
```

应用场景4：借贷协议（Compound/Aave）

```
用户想抵押代币借款：
1. 用户授权借贷合约使用代币
2. 借贷合约调用transferFrom锁定抵押品
3. 用户获得借款
```

3.4 授权安全注意事项

危险做法：无限授权

```
// 危险：授权最大值
token.approve(contract, type(uint256).max);
// 相当于把全部代币的控制权交给了合约
```

问题：

- 如果合约有漏洞，所有代币都可能被盗
- 授权一次永久有效，风险持续存在
- 恶意合约可以随时转走全部代币

安全做法：按需授权

```
// 安全：只授权需要的数量
token.approve(uniswap, 100);  // 只授权本次交易需要的100个

// 使用后撤销授权
token.approve(uniswap, 0);    // 撤销授权
```

授权安全原则：

1. **最小授权**：只授权实际需要的数量
2. **使用后撤销**：完成操作后立即撤销授权
3. **只授权可信合约**：只对经过审计的知名合约授权
4. **定期检查**：定期检查并撤销不再需要的授权
5. **使用授权管理工具**：使用Revoke.cash等工具管理授权

真实案例：

许多用户因为无限授权损失了资金：

- 2021年某DeFi协议被攻击，用户损失数百万美元
- 攻击者利用用户的无限授权转走代币
- 只有撤销授权的用户幸免

教训：永远不要给不熟悉的合约无限授权！

---

4. 代币合约实现

4.1 合约结构设计

完整的ERC20代币合约结构：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract MyToken {
    // 1. 代币基本信息
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;

    // 2. 状态变量
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    // 3. 所有者（用于权限控制）
    address public owner;

    // 4. 事件
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    // 5. 修饰符
    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can call this");
        _;
    }

    // 6. 构造函数
    constructor(
        string memory _name,
        string memory _symbol,
        uint8 _decimals,
        uint256 _initialSupply
    ) {
        name = _name;
```

```
        symbol = _symbol;
        decimals = _decimals;
        totalSupply = _initialSupply * 10**_decimals;
        owner = msg.sender;
        balanceOf[msg.sender] = totalSupply;
        emit Transfer(address(0), msg.sender, totalSupply);
    }

    // 7. 核心函数（见下文）
}
```

4.2 状态变量详解

代币基本信息：

```
string public name;       // 代币名称，如："My Token"
string public symbol;     // 代币符号，如："MTK"
uint8 public decimals;    // 小数位数，通常为18
uint256 public totalSupply;  // 总供应量
```

为什么decimals通常是18?

1. **与ETH一致**：1 ETH = 10^18 wei
2. **精度足够**：18位小数可以表示非常小的金额
3. **行业惯例**：大多数代币都用18
4. **简化计算**：与ETH保持一致便于计算

例外情况：

- USDT：6位小数（与美元的分单位一致）
- USDC：6位小数（同上）
- WBTC：8位小数（与比特币一致）

余额映射：

```
mapping(address => uint256) public balanceOf;
```

这个映射存储了每个地址的代币余额：

- 键（key）：地址
- 值（value）：该地址拥有的代币数量

授权映射（双层映射）：

```
mapping(address => mapping(address => uint256)) public allowance;
```

这是一个嵌套的mapping，存储授权关系：

- 第一层键：授权人地址
- 第二层键：被授权人地址
- 值：授权数量

理解方式：`allowance[Alice][Uniswap]`表示"Alice授权给Uniswap的数量"

4.3 构造函数实现

构造函数在合约部署时执行一次，用于初始化代币。

```
constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals,
    uint256 _initialSupply
) {
    // 设置代币信息
    name = _name;
    symbol = _symbol;
    decimals = _decimals;

    // 计算总供应量
    totalSupply = _initialSupply * 10**_decimals;

    // 设置所有者
    owner = msg.sender;

    // 将所有代币分配给部署者
    balanceOf[msg.sender] = totalSupply;

    // 触发Transfer事件（从零地址到部署者）
    emit Transfer(address(0), msg.sender, totalSupply);
}
```

总供应量计算：

```
totalSupply = _initialSupply * 10**_decimals;

// 例如：
// _initialSupply = 1000
// _decimals = 18
// totalSupply = 1000 * 10^18 = 1000000000000000000000

// 这表示1000个代币，但实际存储的是最小单位
```

为什么从零地址触发Transfer事件?

```
emit Transfer(address(0), msg.sender, totalSupply);
```

- `address(0)`表示代币是新铸造的
- 这是ERC20标准的约定
- 区块链浏览器会识别这是初始分配
- 与销毁代币（to = address(0)）相对应

4.4 transfer函数实现

```
function transfer(address to, uint256 amount) public returns (bool) {
    // 1. 检查接收地址
    require(to != address(0), "Cannot transfer to zero address");

    // 2. 检查余额
    require(balanceOf[msg.sender] >= amount, "Insufficient balance");

    // 3. 更新余额
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;

    // 4. 触发事件
    emit Transfer(msg.sender, to, amount);

    // 5. 返回成功
    return true;
}
```

实现要点：

要点1：零地址检查

```
require(to != address(0), "Cannot transfer to zero address");
```

零地址（0x0000000000000000000000000000000000000000）是特殊地址：

- 发送到零地址的代币永久丢失
- 类似于"黑洞地址"
- 必须防止误操作

要点2：余额检查

```
require(balanceOf[msg.sender] >= amount, "Insufficient balance");
```

防止透支：

- 确保发送者有足够代币
- 防止余额变为负数
- Solidity 0.8.0+会自动检查下溢

要点3：更新顺序

```
balanceOf[msg.sender] -= amount;  // 先减
balanceOf[to] += amount;          // 后加
```

这是CEI模式（Checks-Effects-Interactions）的应用：

- 先更新状态

- 再进行外部交互
- 防止重入攻击

要点4：触发事件

```
emit Transfer(msg.sender, to, amount);
```

事件的作用：

- 钱包监听事件更新余额
- 区块链浏览器显示交易记录
- DApp跟踪代币流动
- 前端实时通知用户

4.5 approve函数实现

```
function approve(address spender, uint256 amount) public returns (bool) {
    // 1. 检查被授权人地址
    require(spender != address(0), "Cannot approve zero address");

    // 2. 设置授权额度
    allowance[msg.sender][spender] = amount;

    // 3. 触发事件
    emit Approval(msg.sender, spender, amount);

    // 4. 返回成功
    return true;
}
```

关键特性：

特性1：覆盖式授权

```
// 第一次授权
approve(uniswap, 100);  // allowance[Alice][Uniswap] = 100

// 第二次授权（覆盖）
approve(uniswap, 200);  // allowance[Alice][Uniswap] = 200（不是300）
```

新授权会完全覆盖旧授权，不是累加。

特性2：取消授权

```
// 取消授权：设置为0
approve(uniswap, 0);  // allowance[Alice][Uniswap] = 0
```

特性3：不转移代币

```
// 执行approve后
// 代币仍然在授权人账户中
// 只是设置了一个"额度"
```

4.6 transferFrom函数实现

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public returns (bool) {
    // 1. 检查地址有效性
    require(from != address(0), "From zero");
    require(to != address(0), "To zero");

    // 2. 检查余额
    require(balanceOf[from] >= amount, "Insufficient balance");

    // 3. 检查授权额度
    require(allowance[from][msg.sender] >= amount, "Insufficient allowance");

    // 4. 执行转账
    balanceOf[from] -= amount;
    balanceOf[to] += amount;

    // 5. 减少授权额度
    allowance[from][msg.sender] -= amount;

    // 6. 触发事件
    emit Transfer(from, to, amount);

    // 7. 返回成功
    return true;
}
```

关键理解：

谁在调用?

```
// Alice授权Uniswap使用500个代币
alice.approve(uniswap, 500);

// Uniswap合约调用transferFrom
// msg.sender = Uniswap合约地址
uniswap.transferFrom(alice, pool, 300);

// 检查：allowance[alice][uniswap] >= 300
// msg.sender是Uniswap，所以检查的是Uniswap的授权额度
```

三方关系：

```
Alice（授权人）
    ↓ 授权
Uniswap（被授权人/调用者）
    ↓ 执行转账
Pool（接收者）
```

检查逻辑：

```
// 从from账户转出，需要检查：
// 1. from的余额是否足够
require(balanceOf[from] >= amount);

// 2. msg.sender（调用者）是否被from授权
require(allowance[from][msg.sender] >= amount);

// 这两个条件都满足才能转账
```

---

5. 扩展功能：Mint和Burn

5.1 Mint - 铸造代币

Mint（铸造）功能用于增加代币供应量，创造新的代币。

函数实现：

```
function mint(address to, uint256 amount) public onlyOwner {
    // 1. 检查接收地址
    require(to != address(0), "Cannot mint to zero address");

    // 2. 增加总供应量
    totalSupply += amount;

    // 3. 增加接收者余额
    balanceOf[to] += amount;

    // 4. 触发Transfer事件（from为零地址）
    emit Transfer(address(0), to, amount);
}
```

Mint的特点：

1. **增加总供应**：`totalSupply += amount`
2. **凭空创造**：代币从零地址"铸造"出来
3. **需要权限**：通常只有owner可以铸造
4. **from为零地址**：`Transfer(address(0), to, amount)`

使用场景：

场景1：稳定币发行（USDC）

```
用户存入100美元到Circle
    ↓
Circle调用mint函数
    ↓
铸造100个USDC给用户
    ↓
totalSupply增加100
```

场景2：游戏代币奖励

```
玩家完成任务
    ↓
游戏合约调用mint
    ↓
铸造奖励代币给玩家
```

场景3：流动性挖矿

```
用户质押LP代币
    ↓
挖矿合约定期mint
    ↓
铸造收益代币给用户
```

场景4：社区激励

```
用户贡献内容
    ↓
DAO投票通过奖励
    ↓
铸造代币给贡献者
```

5.2 Burn - 销毁代币

Burn（销毁）功能用于减少代币供应量，永久销毁代币。

函数实现：

```
function burn(uint256 amount) public {
    // 1. 检查余额
    require(balanceOf[msg.sender] >= amount, "Insufficient balance to burn");

    // 2. 减少总供应量
    totalSupply -= amount;

    // 3. 减少调用者余额
    balanceOf[msg.sender] -= amount;

    // 4. 触发Transfer事件（to为零地址）
    emit Transfer(msg.sender, address(0), amount);
}
```

Burn的特点：

1. **减少总供应**：`totalSupply -= amount`
2. **永久消失**：代币发送到零地址，无法恢复
3. **任何人可调用**：通常不需要权限（销毁自己的代币）
4. **to为零地址**：`Transfer(msg.sender, address(0), amount)`

使用场景：

场景1：稳定币赎回（USDC）

```
用户赎回100美元
    ↓
调用burn销毁100个USDC
    ↓
Circle发送100美元给用户
    ↓
totalSupply减少100
```

场景2：通缩机制

```
每笔转账收取1%手续费
    ↓
手续费自动burn
    ↓
代币越来越稀缺
    ↓
理论上价值增加
```

场景3：购买服务

```
用户用代币购买NFT
    ↓
代币被burn销毁
    ↓
用户获得NFT
    ↓
代币总量减少
```

场景4：回购销毁

```
项目方回购代币
    ↓
调用burn销毁
    ↓
减少市场供应
    ↓
提升代币价值
```

5.3 Mint vs Burn对比

特性	Mint（铸造）	Burn（销毁）
总供应	增加	减少
权限要求	需要（onlyOwner）	不需要（任何人）
Transfer事件	from = address(0)	to = address(0)
可逆性	可逆（可以burn）	不可逆（永久消失）
安全风险	滥发导致通胀	误操作无法恢复
典型用途	发行、奖励、质押收益	回购、销毁、赎回、通缩

完整代码示例：

```solidity
contract TokenWithMintBurn {
    string public name = "My Token";
    string public symbol = "MTK";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    address public owner;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
```

```
    constructor(uint256 _initialSupply) {
        owner = msg.sender;
        totalSupply = _initialSupply * 10**decimals;
        balanceOf[msg.sender] = totalSupply;
        emit Transfer(address(0), msg.sender, totalSupply);
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner");
        _;
    }

    function transfer(address to, uint256 amount) public returns (bool) {
        require(to != address(0), "Zero address");
        require(balanceOf[msg.sender] >= amount, "Insufficient balance");

        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;

        emit Transfer(msg.sender, to, amount);
        return true;
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        require(spender != address(0), "Zero address");

        allowance[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public returns (bool) {
        require(from != address(0), "From zero");
        require(to != address(0), "To zero");
        require(balanceOf[from] >= amount, "Insufficient balance");
        require(allowance[from][msg.sender] >= amount, "Insufficient allowance");

        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        allowance[from][msg.sender] -= amount;

        emit Transfer(from, to, amount);
        return true;
    }

    // 铸造功能
    function mint(address to, uint256 amount) public onlyOwner {
```

```
        require(to != address(0), "Cannot mint to zero address");

        totalSupply += amount;
        balanceOf[to] += amount;

        emit Transfer(address(0), to, amount);
    }

    // 销毁功能
    function burn(uint256 amount) public {
        require(balanceOf[msg.sender] >= amount, "Insufficient balance");

        totalSupply -= amount;
        balanceOf[msg.sender] -= amount;

        emit Transfer(msg.sender, address(0), amount);
    }
}
```

---

6. 部署和测试

6.1 在Remix中部署

步骤1：创建合约文件

1. 打开Remix IDE：https://remix.ethereum.org
2. 在File Explorer中创建新文件：`MyToken.sol`
3. 复制完整的合约代码到文件中

步骤2：编译合约

1. 点击左侧的"Solidity Compiler"图标
2. 选择编译器版本：0.8.19或更高
3. 点击"Compile MyToken.sol"按钮
4. 确保没有错误，只有警告可以忽略

步骤3：准备部署参数

构造函数需要4个参数：

参数	类型	示例值	说明
_name	string	"My Token"	代币名称（带引号）
_symbol	string	"MTK"	代币符号（带引号）
_decimals	uint8	18	小数位数（数字）
_initialSupply	uint256	1000	初始供应量（数字）

注意：

- 字符串参数需要用引号：`"My Token"`
- 数字参数不需要引号：`18`
- 实际总供应量 = _initialSupply × 10^18

步骤4：部署合约

1. 点击左侧的"Deploy & Run Transactions"图标
2. 环境选择：Remix VM (Shanghai)
3. 账户：使用默认的Account 0
4. 合约选择：MyToken
5. 在Deploy旁的输入框填入参数：

   ```
   "My Token","MTK",18,1000
   ```

6. 点击"Deploy"按钮
7. 等待部署完成

步骤5：验证部署

部署成功后，在下方"Deployed Contracts"中可以看到合约实例。

点击展开合约，可以看到所有公开函数和变量：

- name: 返回"My Token"
- symbol: 返回"MTK"
- decimals: 返回18
- totalSupply: 返回1000000000000000000000（1000 × 10^18）
- owner: 返回部署者地址
- balanceOf: 输入地址查询余额

6.2 测试transfer函数

测试场景：Alice向Bob转账100个代币

步骤1：查看初始余额

```
1. 在balanceOf中输入Account 0的地址
2. 点击call按钮
3. 应该显示：100000000000000000000（1000个代币）
```

步骤2：执行转账

```
1. 确保当前账户是Account 0（部署者）
2. 找到transfer函数
3. 填入参数：
   - to：复制Account 1的地址
   - amount：100000000000000000000（100个代币）
4. 点击transact按钮
5. 等待交易确认
```

步骤3：验证结果

```
1. 查询Account 0余额：
   balanceOf(Account 0) = 900000000000000000000 (900个)

2. 查询Account 1余额：
   balanceOf(Account 1) = 100000000000000000000 (100个)

3. 查看事件日志：
   应该有Transfer事件
   from：Account 0
   to：Account 1
   value：100000000000000000000
```

测试负面场景：

测试1：余额不足

```
1. 切换到Account 1（只有100个代币）
2. 尝试转账200个代币给Account 2
3. 应该失败，显示："Insufficient balance"
```

测试2：转账到零地址

```
1. 在to参数中填入：0x0000000000000000000000000000000000000000
2. 尝试转账
3. 应该失败，显示："Cannot transfer to zero address"
```

6.3 测试授权机制

测试场景：Alice授权Bob，Bob代Alice转账给Carol

步骤1：Alice授权Bob

```
1. 切换到Account 0（Alice）
2. 找到approve函数
3. 填入参数：
   - spender：Account 1的地址（Bob）
   - amount：500000000000000000000（500个代币）
4. 点击transact
5. 交易成功
```

步骤2：验证授权

```
1. 找到allowance函数
2. 填入参数：
   - owner：Account 0的地址
   - spender：Account 1的地址
3. 点击call
4. 应该返回：50000000000000000000
```

步骤3：Bob使用授权

```
1. 切换到Account 1（Bob）
2. 找到transferFrom函数
3. 填入参数：
   - from：Account 0的地址（Alice）
   - to：Account 2的地址（Carol）
   - amount：30000000000000000000（300个代币）
4. 点击transact
5. 交易成功
```

步骤4：验证结果

```
1. Account 0余额减少300：
   60000000000000000000（600个）

2. Account 2余额增加300：
   30000000000000000000（300个）

3. 剩余授权额度减少300：
   allowance(Account 0, Account 1) = 20000000000000000000（200个）
```

测试授权不足：

```
1. Account 1现在只有200授权额度
2. 尝试使用300个代币
3. 应该失败，显示："Insufficient allowance"
```

6.4 测试Mint和Burn

测试Mint（如果实现了）：

```
步骤1：切换到Account 0（owner）

步骤2：调用mint函数
- to：Account 1的地址
- amount：500000000000000000000（500个）

步骤3：验证结果
- totalSupply增加500
- Account 1余额增加500
- 查看Transfer事件（from应该是零地址）
```

测试Burn（如果实现了）：

```
步骤1：切换到Account 1

步骤2：调用burn函数
- amount：200000000000000000000（200个）

步骤3：验证结果
- totalSupply减少200
- Account 1余额减少200
- 查看Transfer事件（to应该是零地址）
```

6.5 多账户交互测试

完整测试流程：

```
初始状态：
Account 0：1000个代币（owner）
Account 1：0个代币
Account 2：0个代币

操作1：Account 0转账给Account 1
transfer(Account 1, 100)
结果：Account 0：900, Account 1：100

操作2：Account 0授权Account 1
approve(Account 1, 500)
结果：allowance[0][1] = 500

操作3：Account 1代Account 0转账给Account 2
transferFrom(Account 0, Account 2, 300)
结果：
- Account 0：600（减少300）
- Account 2：300（增加300）
- allowance[0][1] = 200（消耗300）

操作4：Owner铸造代币给Account 2
mint(Account 2, 500)
结果：
```

```
- totalSupply: 1500（增加500）
- Account 2: 800（增加500）

操作5：Account 1销毁自己的代币
burn(50)
结果：
- totalSupply: 1450（减少50）
- Account 1: 50（减少50）

最终状态：
Account 0: 600
Account 1: 50
Account 2: 800
totalSupply: 1450
```

6.6 使用区块链浏览器

查看事件日志：

在Remix的控制台中，每次交易后可以看到详细的事件信息：

```
Transaction Details:
├─ Status: Success
├─ Gas Used: 51,234
├─ Transaction Hash: 0x123...
└─ Logs:
    └─ Transfer(from: 0xABC..., to: 0xDEF..., value: 10000000000000000000)
```

理解事件参数：

- `indexed`参数在日志中单独存储，可以高效查询
- 非indexed参数存储在data字段
- 最多3个indexed参数

---

7. 安全实践

7.1 常见安全问题

问题1：整数溢出/下溢

Solidity 0.8.0之前的问题：

```
// 0.7版本及以前：危险
uint256 balance = 100;
balance -= 200;  // 下溢变成极大值
// 结果：balance = 2^256 - 100（而不是报错）
```

Solidity 0.8.0+的保护：

```
// 0.8版本及以后：安全
uint256 balance = 100;
balance -= 200;  // 自动检查，交易回滚
// 错误："Arithmetic operation underflowed or overflowed"
```

结论：使用Solidity 0.8.0或更高版本！

问题2：重入攻击

虽然基础ERC20代币不涉及外部调用，但如果添加额外功能需要注意。

遵循CEI模式：

```
function withdraw(uint256 amount) public {
    // 1. Checks - 检查
    require(balanceOf[msg.sender] >= amount, "Insufficient balance");

    // 2. Effects - 更新状态（先更新）
    balanceOf[msg.sender] -= amount;

    // 3. Interactions - 外部调用（后调用）
    payable(msg.sender).transfer(amount);
}
```

问题3：Approve竞态条件

问题描述：

```
Alice授权Bob 100个代币
    ↓
Alice想改为授权50个
    ↓
Alice调用：approve(Bob, 50)
    ↓
但Bob在Alice的交易确认前
快速调用transferFrom使用100个
    ↓
然后Alice的交易确认
Bob又获得50个授权
    ↓
Bob总共使用了150个（100+50）
```

解决方案1：先设为0再设新值

```
// 安全做法
token.approve(spender, 0);       // 先撤销
token.approve(spender, newAmount);  // 再授权新额度
```

解决方案2：使用increaseAllowance和decreaseAllowance

```
function increaseAllowance(address spender, uint256 addedValue)
    public returns (bool)
{
    allowance[msg.sender][spender] += addedValue;
    emit Approval(msg.sender, spender, allowance[msg.sender][spender]);
    return true;
}

function decreaseAllowance(address spender, uint256 subtractedValue)
    public returns (bool)
{
    require(
        allowance[msg.sender][spender] >= subtractedValue,
        "Decreased allowance below zero"
    );
    allowance[msg.sender][spender] -= subtractedValue;
    emit Approval(msg.sender, spender, allowance[msg.sender][spender]);
    return true;
}
```

问题4：零地址检查缺失

```
// 危险：没有检查零地址
function badTransfer(address to, uint256 amount) public returns (bool) {
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;  // to可能是零地址
    return true;
}

// 安全：检查零地址
function goodTransfer(address to, uint256 amount) public returns (bool) {
    require(to != address(0), "Cannot transfer to zero address");
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;
    return true;
}
```

7.2 使用OpenZeppelin库

OpenZeppelin是经过专业审计的智能合约库，提供了安全可靠的ERC20实现。

为什么使用OpenZeppelin?

1. **经过审计**：专业安全审计
2. **久经考验**：被数千个项目使用
3. **持续维护**：及时修复已知问题
4. **功能完整**：包含各种扩展功能
5. **最佳实践**：代码质量高

使用OpenZeppelin实现ERC20：

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract MyToken is ERC20, Ownable {
    constructor(uint256 initialSupply) ERC20("My Token", "MTK") {
        _mint(msg.sender, initialSupply * 10**decimals());
    }

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }

    function burn(uint256 amount) public {
        _burn(msg.sender, amount);
    }
}
```

OpenZeppelin vs 手写对比：

特性	手写实现	OpenZeppelin
代码量	~100行	~20行
安全性	需要自己保证	专业审计
功能完整性	基础功能	包含扩展功能
维护成本	需要自己维护	库自动更新
学习价值	高（理解原理）	中（学习使用）
生产使用	不推荐	强烈推荐

建议：

- 学习阶段：手写实现，理解原理
- 生产环境：使用OpenZeppelin库

7.3 安全检查清单

在部署代币合约前，检查以下安全项：

基础安全：

- [ ] 使用Solidity 0.8.0+（自动溢出检查）
- [ ] 所有address参数检查非零地址
- [ ] 所有amount参数检查余额充足
- [ ] transferFrom检查授权额度

授权安全：

- [ ] approve函数检查spender非零
- [ ] 提供increaseAllowance和decreaseAllowance
- [ ] 文档中警告无限授权的风险

权限控制：

- [ ] mint函数使用onlyOwner
- [ ] 敏感函数有适当的权限检查
- [ ] 考虑多签或时间锁控制

事件完整性：

- [ ] 所有状态改变都触发相应事件
- [ ] Transfer事件参数正确
- [ ] Approval事件参数正确

代码质量：

- [ ] 遵循CEI模式
- [ ] 错误信息清晰
- [ ] 代码注释完整
- [ ] 通过编译无警告

测试覆盖：

- [ ] 测试正常转账
- [ ] 测试授权机制
- [ ] 测试边界条件
- [ ] 测试错误场景

---

8. 实战练习

练习1：创建自己的代币

任务：

使用今天学到的知识，创建一个ERC20代币。

要求：

1. 设置自己的名称和符号
2. 实现所有核心功能（transfer、approve、transferFrom）
3. 添加mint和burn功能
4. 部署到Remix并测试
5. 测试所有功能和错误场景

参考参数：

- 名称：选择一个有意义的名称
- 符号：2-5个字符
- 小数位数：18
- 初始供应量：1000-10000

练习2：批量转账功能

任务：

为代币合约添加批量转账功能。

要求：

1. 函数签名：`batchTransfer(address[] memory recipients, uint256[] memory amounts)`
2. 检查数组长度一致
3. 检查总金额不超过余额
4. 限制批量大小（≤50）
5. 正确触发Transfer事件

代码框架：

```
function batchTransfer(
    address[] memory recipients,
    uint256[] memory amounts
) public returns (bool) {
    // TODO: 实现批量转账
    // 1. 检查数组长度
    // 2. 限制批量大小
    // 3. 计算总金额
    // 4. 检查余额
    // 5. 执行转账
}
```

参考答案：

```
function batchTransfer(
    address[] memory recipients,
    uint256[] memory amounts
) public returns (bool) {
    require(recipients.length == amounts.length, "Length mismatch");
    require(recipients.length <= 50, "Batch too large");

    uint256 totalAmount = 0;
    for (uint256 i = 0; i < amounts.length; i++) {
        totalAmount += amounts[i];
    }

    require(balanceOf[msg.sender] >= totalAmount, "Insufficient balance");

    for (uint256 i = 0; i < recipients.length; i++) {
```

```
        require(recipients[i] != address(0), "Invalid address");
        require(amounts[i] > 0, "Invalid amount");
    }

    for (uint256 i = 0; i < recipients.length; i++) {
        balanceOf[msg.sender] -= amounts[i];
        balanceOf[recipients[i]] += amounts[i];
        emit Transfer(msg.sender, recipients[i], amounts[i]);
    }

    return true;
}
```

练习3：暂停功能

任务：

实现合约暂停/恢复功能。

要求：

1. 添加`paused`状态变量
2. 实现`pause`和`unpause`函数（onlyOwner）
3. 在transfer、approve、transferFrom中检查paused状态
4. 暂停时阻止所有转账操作

代码框架：

```
bool public paused = false;

modifier whenNotPaused() {
    require(!paused, "Contract is paused");
    _;
}

function pause() public onlyOwner {
    paused = true;
}

function unpause() public onlyOwner {
    paused = false;
}

function transfer(address to, uint256 amount)
    public whenNotPaused returns (bool)
{
    // 转账逻辑
}
```

练习4：使用OpenZeppelin（选做）

任务：

使用OpenZeppelin库重新实现代币合约。

步骤：

1. 在Remix中导入OpenZeppelin库
2. 继承ERC20和Ownable合约
3. 实现mint和burn功能
4. 对比手写实现和库实现的区别

参考代码：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

contract MyTokenOZ is ERC20, Ownable, ERC20Burnable {
    constructor(uint256 initialSupply) ERC20("My Token", "MTK") {
        _mint(msg.sender, initialSupply * 10**decimals());
    }

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
}
```

---

9. 常见问题解答

Q1：为什么要乘以10的decimals次方?

答：因为Solidity没有浮点数，需要用整数表示小数。

例子：

```
// 如果decimals = 18, initialSupply = 1000
totalSupply = 1000 * 10**18 = 1000000000000000000000

// 这个大数字表示1000个代币
// 最小单位是1（1 wei）
// 1个代币 = 10^18个最小单位
```

类比：

- 1元人民币 = 100分
- 1 ETH = 10^18 wei
- 1个代币 = 10^decimals个最小单位

Q2：Transfer事件为什么from/to是零地址?

答：零地址有特殊含义。

from为零地址：代币铸造

```
emit Transfer(address(0), to, amount);
// 表示代币是新创造的，不是从某个地址转来的
```

to为零地址：代币销毁

```
emit Transfer(from, address(0), amount);
// 表示代币被销毁了，发送到"黑洞"
```

Q3：为什么approve会覆盖而不是累加?

答：这是设计决策，覆盖式更安全。

如果是累加式：

```
Alice授权100
又授权100
结果：200

但如果Alice想改为50呢?
无法精确控制额度
```

覆盖式的优势：

```
Alice授权100
想改为50
直接调用：approve(spender, 50)
结果：50（清晰明确）

想取消授权
调用：approve(spender, 0)
结果：0（完全取消）
```

Q4：transferFrom为什么要减少授权额度?

答：授权额度是一次性的，使用后应该消耗。

如果不减少：

```
Alice授权Bob 100个
Bob使用50个
如果授权额度不减少，Bob可以无限次使用
这违反了授权的初衷
```

正确做法：

```
Alice授权Bob 500个
Bob使用300个
剩余授权：200个
Bob最多还能使用200个
```

Q5：为什么mint需要权限，burn不需要?

答：因为风险不同。

Mint的风险：

- 任何人都能mint会导致无限通胀
- 代币价值归零
- 需要严格控制

Burn的风险：

- 只能销毁自己的代币
- 不影响其他人
- 类似于"扔钱"，虽然浪费但不伤害他人

结论：

- Mint影响所有持币者，需要权限
- Burn只影响自己，不需要权限

Q6：public和external在ERC20中如何选择?

答：根据标准和使用场景选择。

ERC20标准规定使用public：

```
// ERC20标准接口
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 amount) external returns
(bool);
}
```

但实际实现中：

- 如果合约内部需要调用这些函数：使用`public`
- 如果只给外部调用：使用`external`（省gas）

大多数ERC20实现使用`public`，因为：

- 灵活性更高
- 内部也可以调用
- Gas差异不大

Q7：如何在前端显示正确的代币数量?

答：需要除以10的decimals次方。

合约中存储：

```
balanceOf[Alice] = 1500000000000000000000
// 这是1500个代币（decimals=18）
```

前端显示：

```
const balance = await token.balanceOf(aliceAddress);
const decimals = await token.decimals();

// 转换为人类可读的数字
const displayBalance = balance / (10 ** decimals);
console.log(displayBalance);  // 1500

// 或使用ethers.js
const formatted = ethers.utils.formatUnits(balance, decimals);
console.log(formatted);  // "1500.0"
```

---

10. 知识点总结

ERC20标准

核心组成：

- 6个核心函数
- 2个必需事件
- 3个可选元数据

函数分类：

查询函数（view）：

- `totalSupply()`：总供应量
- `balanceOf(address)`：查询余额
- `allowance(address, address)`：查询授权

操作函数：

- `transfer(address, uint256)`：直接转账
- `approve(address, uint256)`：授权
- `transferFrom(address, address, uint256)`：授权转账

授权机制

工作流程：

1. 用户调用approve授权合约
2. 合约调用transferFrom使用授权
3. 授权额度被消耗

安全原则：

- 按需授权，不要无限授权
- 使用后撤销授权
- 只授权可信合约

Mint和Burn

Mint（铸造）：

- 增加总供应量
- 需要权限控制
- from为零地址

Burn（销毁）：

- 减少总供应量
- 任何人可销毁自己的代币
- to为零地址

安全实践

关键原则：

1. 使用Solidity 0.8.0+
2. 遵循CEI模式
3. 检查零地址
4. 清晰的错误消息
5. 生产环境使用OpenZeppelin

---

11. 学习检查清单

完成本课后，你应该能够：

ERC20标准理解：

- [ ] 理解什么是ERC20标准
- [ ] 知道为什么需要代币标准
- [ ] 掌握6个核心函数
- [ ] 理解2个必需事件

授权机制：

- [ ] 理解授权机制的工作原理
- [ ] 知道为什么需要授权
- [ ] 会使用approve和transferFrom
- [ ] 理解授权的安全风险

代币实现：

- [ ] 会实现完整的ERC20代币
- [ ] 会设计状态变量
- [ ] 会实现transfer函数
- [ ] 会实现approve和transferFrom
- [ ] 会添加mint和burn功能

部署和测试：

- [ ] 会在Remix中部署代币
- [ ] 会测试所有功能
- [ ] 会进行多账户交互测试
- [ ] 会查看事件日志

安全意识：

- [ ] 理解常见安全问题
- [ ] 知道如何防范风险
- [ ] 了解OpenZeppelin库
- [ ] 掌握安全检查清单

---

12. 下一步学习

完成本课后，建议：

1. **部署到测试网**
    - 申请Sepolia测试网ETH
    - 部署代币到测试网
    - 在Etherscan上验证合约
    - 测试网上进行真实交互
2. **学习OpenZeppelin**
    - 研究OpenZeppelin的ERC20实现
    - 学习各种扩展功能
    - 理解安全最佳实践
3. **研究真实项目**
    - 分析USDT、USDC的合约代码
    - 学习顶级项目的实现方式
    - 理解生产级代码的特点
4. **准备进阶学习**
    - 准备学习第6课：合约继承
    - 了解更多DeFi协议
    - 学习代币经济模型

下节课预告：第6.1课 - 合约继承

我们将学习：

- 单继承和多重继承
- super关键字的使用
- 构造函数继承
- 函数重写（override）
- 抽象合约和接口

---

13. 扩展资源

官方资源：

- ERC20标准：https://eips.ethereum.org/EIPS/eip-20
- OpenZeppelin ERC20：https://docs.openzeppelin.com/contracts/4.x/erc20
- Solidity文档：https://docs.soliditylang.org

学习资源：

- Solidity by Example - ERC20：https://solidity-by-example.org/app/erc20/
- OpenZeppelin Contracts：https://github.com/OpenZeppelin/openzeppelin-contracts

工具推荐：

- Remix IDE：在线开发环境
- Etherscan：区块链浏览器
- Revoke.cash：授权管理工具
- Hardhat：专业开发框架

实战学习：
研究真实的ERC20代币：

- USDT合约：https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7
- USDC合约：https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48
- UNI合约：https://etherscan.io/token/0x1f9840a85d5af5bf1d1762f925bdaddc4201f984

安全资源：

- Smart Contract Security Best Practices
- Common ERC20 Vulnerabilities
- OpenZeppelin Security Audits