## 概述

大家好，我是本节课的讲师 shew 。我是一名早期的 cairo 开发者，拥有较为丰富的 cairo 开发经验。但我目前的工作是为 defi 系统提供 solidity 编写的数值计算算法。

本节课是大家学习 cairo 的第三节课，我们主要介绍以下内容:

1. ERC20 智能合约的开发

2. ERC20 智能合约的 scarb 测试

3. ERC20 智能合约的部署

## 开发

与上节课介绍的 cairo 程序不同，智能合约比普通程序增加了以下四部分:

1. ABI 方便合约进行链上调用，规定用户可调用的外部函数

2. Storage 存储状态，正常程序可以直接调用操作系统写入，而区块链则需要使用特定操作码或函数写入

3. Event 事件，主要方便于链下索引，是避免链上状态空间膨胀的重要措施

4. Constructor 构造器，用于合约初始化

我们会以 ERC20 合约为例介绍以上几部分如何实现。

当然，第一步永远是创建项目，使用以下命令即可:

```bash

scarb new tintin\_erc20

```

接下来，我们可以简单创建一些模块以方便我们后期开发:

第一步，创建 `tests` 文件夹及其同名 cairo 文件 `tests.cairo`，并在 `tests` 文件夹内创建 `erc20\_test.cairo` ，并在 `tests.cairo` 内写入 `mod erc20\_test;`

第二步，创建 `erc20.cairo` 文件，并修改 `lib.cairo` 文件

```rust

mod erc20;

#[cfg(test)]

mod tests;

```

最后，我们需要为合约增加 starknet 区块链依赖，编辑 `Scarb.toml` 文件，加入以下内容:

```toml

[dependencies]

starknet = ">=2.4.0"

```

该配置意味着为此项目增加 `2.4.0` 及其以上版本的 starknet 支持。

### ABI

在编写具体的智能合约前，我们首先需要定义智能合约的 ABI 情况，关于 ERC20 所需要的 ABI ，大家应该都比较熟悉。我们首先编写 ABI 定义的框架:

```rust

#[starknet::interface]

trait IERC20<TContractState> {

}

```

此处使用宏 `#[starknet::interface]` 标明以下 `trait` 是一个 ABI 接口，而此处的 `TContractState` 是一个泛型参数，一般使用 `TContractState` 或 `TCS`

我们首先编写一个最简单的 `name` 函数的 ABI，如下:

```rust

fn name(self: @TContractState) -> felt252;

```

此处使用了快照类型，这代表 `name` 函数不会修改当前智能合约的状态空间，简单来说，就是当前函数不会写入任何数据，类似 solidity 中的 `view` 或 `pure` 函数。为兼容主网版本，这里依旧使用了 `felt252` 作为字符串返回。

其他如 `symbol` 、 `decimals` 和 `totalSupply` 函数类似，定义如下:

```rust

fn symbol(self: @TContractState) -> felt252;

fn decimals(self: @TContractState) -> u8;

fn totalSupply(self: @TContractState) -> u256;

```

而对于 `balanceOf` 和 `allowance`，定义如下:

```rust

fn balanceOf(self: @TContractState, account: ContractAddress) -> u256;

fn allowance(self: @TContractState, owner: ContractAddress, spender: ContractAddress) -> u256;

```

但此处出现报错，`Type not found.` 其含义为 `ContractAddress` 类型不存在，我们需要手动导入此类型。如下:

```rust

use starknet::ContractAddress;

```

众所周知，还有一些函数需要修改合约状态，如 `transfer` 等，对于这些函数，我们需要使用以下定义:

```rust

fn transfer(ref self: TContractState, to: ContractAddress, amount: u256) -> bool;

```

我们在 `self` 前增加了 `ref` 标识，此标识代表该函数可以修改当前合约状态。其他的函数定义类似，如下:

```rust

fn transferFrom(

ref self: TContractState, from: ContractAddress, to: ContractAddress, amount: u256

) -> bool;

fn approve(ref self: TContractState, spender: ContractAddress, amount: u256) -> bool;

fn mint(ref self: TContractState, amount: u256);

```

在此处，我们可以简单讨论一些历史遗留问题。在 cairo 0 开发中，cairo 使用了与 solidity 一致的命名规范，但是上节课展示了新版本命名规范与 solidity 的命名规范显然不同。我观察到市场上大部分钱包仍是只兼容 solidity 的命名规范，所以上文使用了 solidity 命名规范而没有使用目前 cairo 1 的规范。

### 存储

在介绍存储前，我们需要首先定义智能合约，如下:

```rust

#[starknet::contract]

mod erc20 {

}

```

此时编译提醒我们需要创建存储结构体。(`Plugin diagnostic: Contracts must define a 'Storage' struct.` )

我们直接创建存储结构体，如下:

```rust

#[starknet::contract]

mod erc20 {

#[storage]

struct Storage {

\_name: felt252,

\_symbol: felt252,

\_decimals: u8,

\_total\_supply: u256,

}

}

```

而对于映射类似，比如记录用户余额的 `\_balances` ，我们需要进行以下定义:

```rust

\_balances: LegacyMap::<ContractAddress, u256>,

```

此处也需要增加 `use starknet::ContractAddress; ` 的导入语句。

更加复杂的是多重映射，如 `allownaces` ，其定义如下:

```rust

\_allowances: LegacyMap::<(ContractAddress, ContractAddress), u256>,

```

我们使用 `(ContractAddress, ContractAddress)` 两个地址作为键来检索用户的授权额度。此处有一个重要的知识点，我们可以看到 `u256` 作为了映射中的值使用。而基于上节课的学习，大家应该知道 `u256` 是一个结构体。那么一个结构体如何作为映射中的值使用呢？

```rust

#[derive(Copy, Drop, Hash, PartialEq, Serde, starknet::Store)]

struct u256 {

low: u128,

high: u128,

}

```

我们可以看到此处的派生了 `starknet::Store` 宏，为任意结构体派生这个宏后，该结构体就可以作为映射中的值使用。

而此处派生的 `Hash` 宏，则为该结构体增加了作为映射中的键的能力。我们可以编写如下结构体:

```

#[derive(starknet::Store, Hash, Drop)]

struct test {

name: felt252,

symbol: felt252

}

```

然后在 `storage` 内加入以下映射:

```rust

\_test: LegacyMap::<test, test>

```

此处我们发现出现报错 `Variable not dropped.`，为 `test` 增加宏即可:

```rust

#[derive(starknet::Store, Hash, Drop)]

struct test {

name: felt252,

symbol: felt252

}

```

### 事件

事件是智能合约的重要组成部分之一，在 cairo 中，事件的编写是基于枚举类型和结构体的。

我们首先需要创建一个枚举类型，要求枚举类型中列出所有的合约事件:

```rust

#[event]

#[derive(Drop, starknet::Event)]

enum Event {

Transfer: Transfer,

Approval: Approval,

}

```

此处为 `Event` 派生了 `Drop` 和 `starknet::Event` 属性，其中后者较为重要，代表此枚举类型与事件有关而不是仅仅是一个枚举类型。

接下来，我们需要具体定义 `Transfer` 和 `Approval` 事件，如下:

```rust

#[derive(Drop, starknet::Event)]

struct Transfer {

#[key]

from: ContractAddress,

#[key]

to: ContractAddress,

value: u256,

}

#[derive(Drop, starknet::Event)]

struct Approval {

#[key]

owner: ContractAddress,

#[key]

spender: ContractAddress,

value: u256,

}

```

与 `Event` 的定义类似，但是 `Transfer` 和 `Approval` 事件都是以结构体定义且使用了 `#[key]` 标识，此标识类似 solidity 中的 `index` 属性。

### 构造器

智能合约需要构造器初始化一些基本属性，cairo 中的构造器较为简单，如下:

```rust

#[constructor]

fn constructor(ref self: ContractState, name: felt252, symbol: felt252, decimals: u8, ) {

self.\_name.write(name);

self.\_symbol.write(symbol);

self.\_decimals.write(decimals);

}

```

在这里，我们可以看到如何对单一变量存储进行写操作，使用 `write` 方法即可。

### 具体实现

在完成上述所有工作后，我们剩余的工作事实上与编写普通的 cairo 程序区别不大。但是还是存在一些特殊的语法，首先编写框架:

```rust

#[external(v0)]

impl IERC20Impl of super::IERC20<ContractState> {

}

```

此处的 `#[external(v0)]` 宏意味着以下对 `IERC20` 的实现都为外部函数，即用户可以直接调用这些函数。而内部函数的实现，我们会在后面介绍。

此时，我们可以看到编译器报错 `Not all trait items are implemented. Missing: 'name', 'balanceOf', 'symbol', 'decimals', 'totalSupply', 'allowance', 'transfer', 'transferFrom', 'approve', 'mint'.`。该报错提醒我们 ABI 内的函数没有实现。注意，cairo 是严格的，其 ABI 定义的函数必须全部实现，且不允许实现 ABI 内不存在的函数。

我们首先实现一些简单的函数:

```rust

fn name(self: @ContractState) -> felt252 {

self.\_name.read()

}

```

`name` 函数展示了如何读取单个变量的存储值，`symbol` `decimals` 和 `totalSupply` 函数写法类似:

```rust

fn symbol(self: @ContractState) -> felt252 {

self.\_symbol.read()

}

fn decimals(self: @ContractState) -> u8 {

self.\_decimals.read()

}

fn totalSupply(self: @ContractState) -> u256 {

self.\_total\_supply.read()

}

```

对于 `balanceOf` 和 `allowance` 函数，其实现与 `name` 等有所不同，因为这些函数涉及到了映射类型的读取。其读取逻辑为输入键、返回键对应的值。实现如下:

```rust

fn balanceOf(self: @ContractState, account: ContractAddress) -> u256 {

self.\_balances.read(account)

}

fn allowance(

self: @ContractState, owner: ContractAddress, spender: ContractAddress

) -> u256 {

self.\_allowances.read((owner, spender))

}

```

此处的 `allowance` 较为特殊，我们刚刚也提到，`allowance` 是使用元组作为键的，允许我们使用多个键查询某一个值。

完成上述工作后，我们介绍一下合约测试，但是正如目前编译器报错，当我们没有完整实现接口时，编译是无法通过的，所以此处我们增加一些空的实现:

```rust

fn mint(ref self: ContractState, amount: u256) {}

fn approve(ref self: ContractState, spender: ContractAddress, amount: u256) -> bool {

true

}

fn transfer(ref self: ContractState, to: ContractAddress, amount: u256) -> bool {

true

}

fn transferFrom(

ref self: ContractState, from: ContractAddress, to: ContractAddress, amount: u256

) -> bool {

true

}

```

之后，我们可以在 `src/tests/erc20\_test.cairo` 文件内编写第一个测试，我们希望测试构造器函数能否正常运行，编写如下代码:

```rust

use tintin\_erc20::erc20::ERC20;

use tintin\_erc20::erc20::IERC20Dispatcher;

use tintin\_erc20::erc20::IERC20DispatcherTrait;

use tintin\_erc20::erc20::ERC20::{Event, Approval};

use starknet::contract\_address::ContractAddress;

use starknet::syscalls::deploy\_syscall;

use core::test::test\_utils::assert\_eq;

const NAME: felt252 = 'Test';

const SYMBOL: felt252 = 'TET';

const DECIMALS: u8 = 18\_u8;

#[test]

fn test\_initializer() {

let mut calldata = array![NAME, SYMBOL, DECIMALS.into()];

let (erc20\_address, \_) = deploy\_syscall(

ERC20::TEST\_CLASS\_HASH.try\_into().unwrap(), 0, calldata.span(), false

)

.unwrap();

let erc20\_token = IERC20Dispatcher { contract\_address: erc20\_address };

assert\_eq(@erc20\_token.name(), @NAME, 'Name should be NAME');

assert\_eq(@erc20\_token.symbol(), @SYMBOL, 'Symbol should be SYMBOL');

assert\_eq(@erc20\_token.decimals(), @18\_u8, 'Decimals should be 18');

}

```

为了进行测试，我们导入了大量依赖，其中最核心的依赖为 `IERC20Dispatcher` 和 `IERC20DispatcherTrait` 。大家可能感觉我们似乎没有在 ERC20 合约内实现这两个模块，实际上，这两个模块是由 IERC20 接口衍生获得的，是 编译器自动生成的。

此处的核心之一是 `deploy\_syscall` 函数，此函数定义如下:

```rust

extern fn deploy\_syscall(

class\_hash: ClassHash,

contract\_address\_salt: felt252,

calldata: Span<felt252>,

deploy\_from\_zero: bool,

) -> SyscallResult<(ContractAddress, Span<felt252>)> implicits(GasBuiltin, System) nopanic;

```

该函数的作用是基于合约的 class hash 部署合约，对于 class hash 的具体含义，我们会在之后介绍。简单来说，Starknet 的合约部署分为两步，第一步需要 `declare` 合约获得合约的 class hash，而第二步则为使用 class hash 来部署合约。

最后，我们使用了 `assert\_eq` 进行相等性判断，我们可以看到 `assert\_eq` 函数是特殊的，其参数类型为快照类型，这意味着我们需要手动使用 `@` 进行类型转化。

最后，运行 `scarb test` 获取测试结果。

然后，我们开始关注一些可以修改智能合约状态空间的函数，其中 `approve` 较为简单，如下:

```rust

fn approve(ref self: ContractState, spender: ContractAddress, amount: u256) -> bool {

let owner = get\_caller\_address();

self.\_allowances.write((owner, spender), amount);

self.emit(Event::Approval(Approval { owner, spender, value: amount }));

true

}

```

此处使用了 `get\_caller\_address`，其功能类似 solidity 中的 `msg.sender`，但该函数需要手动导入:

```rust

use starknet::get\_caller\_address;

```

此处展示了如何对映射类型存储变量写入数据，简单来说就是调用 `write` 函数，第一个参数为键而第二个参数为值。最后，我们展示了如何释放事件，此处使用同名变量的遮盖作用，完整写法如下:

```rust

fn approve(ref self: ContractState, spender: ContractAddress, amount: u256) -> bool {

let owner = get\_caller\_address();

self.\_allowances.write((owner, spender), amount);

self.emit(Event::Approval(Approval { owner: owner, spender: spender, value: amount }));

true

}

```

但由于变量名与结构体成员名称同名，我们可以直接书写 `Approval { owner, spender, value: amount }` 来简化事件释放。

完成 `approve` 函数编写后，我们进行函数的相关测试。在进行测试前，我们首先编写辅助函数 `setUp` ，此函数会帮助我们完成合约部署和初始化工作，并设置合约的调用者，该函数实现如下:

```rust

fn setUp() -> (ContractAddress, IERC20Dispatcher, ContractAddress) {

let caller = contract\_address\_const::<1>();

set\_contract\_address(caller);

let mut calldata = array![NAME, SYMBOL, DECIMALS.into()];

let (erc20\_address, \_) = deploy\_syscall(

ERC20::TEST\_CLASS\_HASH.try\_into().unwrap(), 0, calldata.span(), false

)

.unwrap();

let mut erc20\_token = IERC20Dispatcher { contract\_address: erc20\_address };

(caller, erc20\_token, erc20\_address)

}

```

此处需要注意 `set\_contract\_address` 和 `contract\_address\_const` 函数都需要使用以下语句导入:

```rust

use starknet::testing::set\_contract\_address;

use starknet::contract\_address\_const;

```

该函数的作用是将我们编写函数的地址修改为 `contract\_address\_const::<1>()` 地址。使用 `set\_contract\_address` 后，对 ERC20 合约的所有调用对会在 `contract\_address\_const::<1>()` 地址发起。这里也涉及到 cairo 测试的基本原理，cairo 的测试可以认为是我们将测试代码和 ERC20 代币合约一起部署到测试环境中，测试代码对 ERC20 代币合约进行调用，所以此处我们使用 `set\_contract\_address` 通过修改测试函数的地址实现修改 ERC20 代币合约内的 `get\_caller\_address` 的值。

编写完成辅助函数后，我们需要真正编写 `approve` 的测试代码，如下:

```rust

#[test]

fn test\_approve() {

let (caller, erc20\_token, erc20\_address) = setUp();

let spender = contract\_address\_const::<2>();

let amount = 2000\_u256;

erc20\_token.approve(spender, amount);

assert\_eq(@erc20\_token.allowance(caller, spender), @amount, 'Approve should eq 2000');

}

```

我们要注意到 `approve` 函数会释放事件，所以对事件的检测也应该包含在测试中，如下:

```rust

assert\_eq(

@starknet::testing::pop\_log(erc20\_address).unwrap(),

@Event::Approval(Approval { owner: caller, spender: spender, value: amount }),

'Approve Emit'

)

```

但此处，我们发现编译器给出了 `Trait has no implementation in context: core::traits::PartialEq::<tintin\_erc20::erc20::ERC20::Event>` 的报错，这说明我们的事件结构体缺失某些属性的实现，我们可以直接使用 `derive` 进行衍生。如下:

```rust

#[derive(Drop, PartialEq, starknet::Event)]

```

此处使用了 `pop\_log` 函数，该函数的作用是弹出当前合约的事件，我们可以认为合约所有的事件构成了一个列表,`pop\_log` 函数就是弹出此列表中的最早的一个事件。

对于 `mint` 函数的实现较为简单，实现如下:

```rust

fn mint(ref self: ContractState, amount: u256) {

let sender = get\_caller\_address();

self.\_total\_supply.write(self.\_total\_supply.read() + amount);

self.\_balances.write(sender, self.\_balances.read(sender) + amount);

self

.emit(

Event::Transfer(

Transfer { from: core::Zeroable::zero(), to: sender, value: amount }

)

)

}

```

此处需要零地址，我们使用了 `core::Zeroable::zero()` 函数获得，此函数的作用就是获取各个类型的 0，该函数我们在上一节课内也有所使用。

而其测试则较为简单，如下:

```rust

#[test]

fn test\_mint() {

let (caller, erc20\_token, \_) = setUp();

let amount = 2000\_u256;

erc20\_token.mint(amount);

assert\_eq(@erc20\_token.balanceOf(caller), @amount, 'Mint 2000');

}

```

此处我们没有为 `mint` 的测试:

```rust

#[test]

fn test\_mint() {

let (caller, erc20\_token, erc20\_address) = setUp();

let amount = 2000\_u256;

erc20\_token.mint(amount);

assert\_eq(@erc20\_token.balanceOf(caller), @amount, 'Mint 2000');

assert\_eq(

@starknet::testing::pop\_log(erc20\_address).unwrap(),

@Event::Transfer(Transfer { from: core::Zeroable::zero(), to: caller, value: amount }),

'Transfer Emit'

)

}

```

我们继续编写 `transfer` 逻辑，如下:

```rust

fn transfer(ref self: ContractState, to: ContractAddress, amount: u256) -> bool {

let from = get\_caller\_address();

self.\_balances.write(from, self.\_balances.read(from) - amount);

self.\_balances.write(to, self.\_balances.read(to) + amount);

self.emit(Event::Transfer(Transfer { from, to, value: amount }));

true

}

```

也是一个较为简单的逻辑，我们需要减少 `from` 的余额，并增加 `to` 地址的余额即可。

对于 `transfer` 的正向测试是简单的，如下:

```rust

#[test]

fn test\_transfer() {

let (from, erc20\_token, \_) = setUp();

let to = contract\_address\_const::<2>();

let amount = 2000\_u256;

erc20\_token.mint(amount);

erc20\_token.transfer(to, amount);

assert\_eq(@erc20\_token.balanceOf(from), @core::Zeroable::zero(), 'Balance from = 0');

assert\_eq(@erc20\_token.balanceOf(to), @amount, 'Balance to = 2000');

}

```

此处，我们需要对 `transfer` 进行逆向测试，判断 `transfer` 在用户转移大于其余额资产时是否会产生报错，如下:

```rust

#[test]

fn test\_err\_transfer() {

let (from, erc20\_token, \_) = setUp();

let to = contract\_address\_const::<2>();

let amount = 2000\_u256;

erc20\_token.mint(amount);

erc20\_token.transfer(to, 3000\_u256);

}

```

使用 `scarb test` 运行测试，发现出现预期报错:

```bash

failures:

tintin\_erc20::tests::erc20\_test::test\_err\_transfer - Panicked with (0x753235365f737562204f766572666c6f77 ('u256\_sub Overflow'), 0x454e545259504f494e545f4641494c4544 ('ENTRYPOINT\_FAILED')).

```

此处，我们可以使用以下方法预期该错误发生:

```

#[should\_panic(expected: ('u256\_sub Overflow', 'ENTRYPOINT\_FAILED', ))]

```

最后，我们编写 `transferFrom` 的逻辑:

```rust

fn transferFrom(

ref self: ContractState, from: ContractAddress, to: ContractAddress, amount: u256

) -> bool {

let caller = get\_caller\_address();

let allowed: u256 = self.\_allowances.read((from, caller));

if (allowed != BoundedInt::max()) {

self.\_allowances.write((from, caller), allowed - amount);

self

.emit(

Event::Approval(

Approval { owner: from, spender: caller, value: allowed - amount }

)

);

}

self.\_balances.write(from, self.\_balances.read(from) - amount);

self.\_balances.write(to, self.\_balances.read(to) + amount);

self.emit(Event::Transfer(Transfer { from, to, value: amount }));

true

}

```

此处需要引入 `use core::integer::BoundedInt;` ，该函数提供了所有数值类型的最大值。关于 `transferFrom` 的核心逻辑是在用户 `approve` 最大值时，进行 `transferFrom` 不应该扣除用户的授权额度。

我们应该对其两个分支都进行测试:

首先测试最简单的情况，如下:

```rust

#[test]

fn test\_transferFrom() {

let amount = 2000;

let (owner, erc20\_token, \_) = setUp();

let from = contract\_address\_const::<2>();

let to = contract\_address\_const::<3>();

erc20\_token.mint(amount);

erc20\_token.approve(from, amount);

set\_contract\_address(from);

erc20\_token.transferFrom(owner, to, 1000);

assert(erc20\_token.balanceOf(owner) == 1000, 'Balance owner == 1000');

assert(erc20\_token.balanceOf(to) == 1000, 'Balance to == 1000');

assert(erc20\_token.allowance(owner, from) == 1000, 'Approve == 1000')

}

```

然后，我们需要进行一次对于 `transferFrom` 的测试，如下:

```rust

#[test]

fn test\_transferFrom() {

let amount = 2000;

let (owner, erc20\_token, erc20\_address) = setUp();

let from = contract\_address\_const::<2>();

let to = contract\_address\_const::<3>();

erc20\_token.mint(amount);

erc20\_token.approve(from, amount);

set\_contract\_address(from);

erc20\_token.transferFrom(owner, to, 1000);

assert(erc20\_token.balanceOf(owner) == 1000, 'Balance owner == 1000');

assert(erc20\_token.balanceOf(to) == 1000, 'Balance to == 1000');

assert(erc20\_token.allowance(owner, from) == 1000, 'Approve == 1000');

assert\_eq(

@starknet::testing::pop\_log(erc20\_address).unwrap(),

@Event::Transfer(Transfer { from: core::Zeroable::zero(), to: owner, value: amount }),

'First Transfer Emit'

);

assert\_eq(

@starknet::testing::pop\_log(erc20\_address).unwrap(),

@Event::Approval(Approval { owner, spender: from, value: 2000 }),

'First Approve Emit'

);

assert\_eq(

@starknet::testing::pop\_log(erc20\_address).unwrap(),

@Event::Approval(Approval { owner, spender: from, value: 1000 }),

'Transfer Approve Emit'

);

assert\_eq(

@starknet::testing::pop\_log(erc20\_address).unwrap(),

@Event::Transfer(Transfer { from: owner, to, value: 1000 }),

'TransferFrom Emit'

);

}

```

如上文所述，`pop\_log` 是从最早释放的事件开始弹出，所以此处检查了 `mint` 时的 `Transfer` 事件，第一次授权时的 `Approve` 事件和 `transferFrom` 调用期间的 `Transfer` 和 `Approve` 事件。

最后，我们编写 `transferFrom` 的另一个分支测试:

```rust

#[test]

fn test\_MAXApproveTransfer() {

let max: u256 = BoundedInt::max();

let amount = 2000;

let (owner, erc20\_token, \_) = setUp();

let from = contract\_address\_const::<2>();

let to = contract\_address\_const::<3>();

erc20\_token.mint(amount);

erc20\_token.approve(from, max);

set\_contract\_address(from);

erc20\_token.transferFrom(owner, to, 1000);

assert(erc20\_token.allowance(owner, from) == max, 'Max Approve invarient')

}

```

### 内部函数

在上文编写智能合约时，我们发现转移用户代币并释放事件的操作在 `transfer` 和 `transferFrom` 内都有出现，所以我们考虑将其抽象为一个函数，但是该函数应不允许用户从外部调用，所以此处我们涉及到了 cairo 智能合约开发中的重要一步，即如何实现内部函数？

我们首先实现内部函数的框架:

```rust

#[generate\_trait]

impl StorageImpl of StorageTrait {

}

```

此处的 `#[generate\_trait]` 宏的作用是根据 `StorageImpl` 的具体实现给出 `StorageTrait` 的代码。

我们直接编写 `transfer\_helper` 函数，如下:

```rust

fn transfer\_helper(

ref self: ContractState,

sender: ContractAddress,

recipient: ContractAddress,

amount: u256

) {

self.\_balances.write(sender, self.\_balances.read(sender) - amount);

self.\_balances.write(recipient, self.\_balances.read(recipient) + amount);

self.emit(Transfer { from: sender, to: recipient, value: amount });

}

```

此时我们可以检查一下 `StorageTrait` 的定义，如下:

```rust

trait StorageTrait {

fn transfer\_helper(

ref self: ContractState,

sender: ContractAddress,

recipient: ContractAddress,

amount: u256

);

}

```

我们可以看到编译器自动生成了函数定义的 `trait` ，这大大减少了模板代码的编写。

接下来，我们可以将 ERC20 合约内的代币转移部分更换为此内部函数，首先修改 `transfer` 函数，修改后结果如下:

```rust

fn transfer(ref self: ContractState, to: ContractAddress, amount: u256) -> bool {

let from = get\_caller\_address();

self.transfer\_helper(from, to, amount);

true

}

```

此处使用 `self.transfer\_helper` 调用内部函数。

`transferFrom` 的修改也与之类似。

## 部署

修改 `Scarb.toml` 增加编译目标:

```toml

[[target.starknet-contract]]

allowed-libfuncs = true

```

我们需要进行合约部署，首先使用 `scarb test` 确保合约可以正常通过全部测试，然后使用 `scarb build` 编译整个项目。

在上次课程内，我们安装了旧版本的 `starkli` ，但在本次课程内，由于我们智能合约的版本较新，所以此处需要使用最新版本的 `starkli`，使用 `starkliup` 更新版本即可。

然后配置 `starkli` 必要的环境变量:

```bash

export STARKNET\_KEYSTORE=~/.starknet\_accounts/key.json

export STARKNET\_RPC=https://free-rpc.nethermind.io/goerli-juno

export STARKNET\_ACCOUNT=~/.starknet\_accounts/starkli.json

```

使用 `starkli declare` 交易部署 class hash，如下:

```bash

starkli declare target/dev/tintin\_erc20\_ERC20.contract\_class.json

```

终端会出现如下内容:

![Declare Output](https://blogimage.4everland.store/starkliDeclareOutput.png)

但是我们一般会遇到以下输出:

```bash

Not declaring class as it's already declared. Class hash:

0x03a03848d08e2eb8ebffa1da26ef8f64175ad73aed6ed5c75bd7b0c4f8ebed4b

```

这是因为一个智能合约只能在 starknet 内注册一次。

使用以下命令部署合约:

```bash

starkli deploy 0x03a03848d08e2eb8ebffa1da26ef8f64175ad73aed6ed5c75bd7b0c4f8ebed4b str:HELLO str:HE 18

```

此处使用 `str` 作为前缀代表参数的类型，而 `starkli` 工具会在后台自动转化类型。