## 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 的第二节课，我们主要介绍以下内容:

1. cairo 开发环境的安装与配置，主要涉及 scarb 的安装与 vscode 的插件配置

2. cairo 语言的基础语法及测试，在此节中，我们将介绍 cairo 中的变量、基础数据类型、函数、控制流、所有权、结构体、泛型

3. starknet 账户配置与合约交互，此处我们主要使用 starkli 工具，并初步介绍 starknet 中的账户抽象

注意，在本节课中，我们所展示的所有操作可以在 Linux 和 MacOS 下进行，但没有在 Windows 下进行过测试。我个人使用的系统环境为 Win10 下的 WSL 系统。

## 环境配置

在这里，我使用一个空白的 github codespaces 为大家展示如何在 Linux 或 MacOS 下配置 cairo 环境。

```bash

uname -a

```

大家可以看到这是一个 ubuntu 22.04.1 的系统。

我们首先安装最重要的组件 `scarb` ，打开 [scarb 官网](https://docs.swmansion.com/scarb/download#install-via-installation-script) ，直接使用官方命令安装 `stable` 版本即可。安装成功后，我们直接运行 `scarb --version` 命令查看一下当前版本，此处我们使用了 `cairo 2.4.1` 版本。在 Cairo 2.4.1 中，第一次完整引入了对字符串的支持，我们会在后文使用这一特性。当然，目前主网暂时不支持 `cairo 2.4` 版本，但测试网已经支持了相关版本。

如果读者需要主网部署合约，请使用以下命令安装旧版本的 scarb :

```bash

curl --proto '=https' --tlsv1.2 -sSf https://docs.swmansion.com/scarb/install.sh | sh -s -- -v v2.3.1

```

我们在此处就不在展示了。

在后续智能合约编程中，我会使用兼容语法编写以避免主网无法部署的情况发生，如果您遇到任何版本问题，可以随时在群里联系我。对于 cairo 开发而言，遇到版本问题是每一个工程师必经之路。

接下来，我们继续配置 vscode 使其支持 cairo 语言。在拓展搜索栏内搜索 `cairo 1.0` 插件，然后安装即可。接下来，我们需要对其进行配置，我们只需要填入 `scarb` 地址即可。使用 `which scarb` 查询 `scarb` 的目录地址，然后填入配置项即可。

这里可以介绍一个有用的知识，即 scarb 实际上除了提供包管理功能，其内部已经内置了包含 cairo LSP 在内的所有工具，所以 scarb 实际上是一个 `all in one` 的开发环境。开发 cairo 理论上只需要安装此工具即可。但是需要注意 scarb 没有提供合约部署功能，这些我们需要使用其他工具，我们会在下一节课介绍合约部署。

我们初始化一个项目来看看配置是否有效。

使用 `scarb new hello\_world` 初始化一个简单项目，我们可以看到语法高亮已经生效了，定义跳转也是正常的。

然后，我们进行第一次 cairo 程序运行和测试，使用 `scarb cairo-run` 进行 cairo 语言运行，这里的返回值比较奇怪，我们使用 cairo 2.4 的新语法修正一下，删除 `main` 函数返回值 `felt252` ，然后加入 `println!` 宏打印输出值。

重新运行 `scarb cairo-run` ，我们发现此时输出更符合我们预期。

接下来，我们使用 `scarb test` 进行测试。显然，我们通过了测试，并可以看到测试报告的 gas 消耗。

## 基础语法

在介绍基础语法前，我们首先修改我们的 `lib.cairo` 使其符合我们需求:

```rust

fn main() {

}

#[cfg(test)]

mod tests {

#[test]

fn it\_works() {

}

}

```

### 变量

在 `cairo` 里，我们可以使用以下方法声明变量:

```rust

fn main() {

let x = 5;

println!("x is {}", x);

}

```

然后，我们可以尝试直接修改此变量，如下:

```rust

fn main() {

let x = 5;

println!("x is {}", x);

x = 6;

}

```

我们发现编译器检测到了一个错误，其含义为无法为不可变变量重新赋值，这说明 cairo 中使用 `let` 声明的任何一个变量都是不可变的。我们可以使用以下方法修正此错误:

```rust

fn main() {

let mut x = 5;

println!("x is {}", x);

x = 6;

println!("x is {}", x);

}

```

此处的 `mut` 即代表 `x` 是一个可修改的变量而不是常量。在此处，我们也可以看到 cairo 的哲学是尽可能声明不可变变量。

在此处，我们也可以看到 `x` 的类型被推导为 `felt252` 类型，这是 cairo 中最基础的类型之一，但是我们需要注意 `felt252` 的除法运算是特殊的，其具有一些诡异特性，所以 cairo 标准库没有实现 felt252 的除法。更多内容我们会在后文介绍数据类型时为大家介绍。

当然，`cairo` 中也存在名副其实的常量，其定义如下:

```rust

const ONE\_HOUR\_IN\_SECONDS: u32 = 3600;

fn main() {

let x = 5;

println!("x is {}", x);

println!("ONE\_HOUR\_IN\_SECONDS is {}", ONE\_HOUR\_IN\_SECONDS);

}

```

此处我们使用了类型注释，这部分内容我们会在马上介绍。和其他语言类似，cairo 中的常量使用大写字母且单词之间使用下划线。我们可以在 [这里](https://github.com/starkware-libs/cairo/blob/main/docs/reference/src/components/cairo/modules/language\_constructs/pages/naming-conventions.adoc) 找到所有的命名规范。

### 数据类型

Cairo 是一门静态语言，其要求在编译过程中所有变量的类型都是确定的。大部分情况下，cairo 可以自行推导出变量的类型，如上文给出的 `x` 变量，编译器将其自动推导为 `felt252`，但是有些情况下需要我们手动指定类型，如下:

```rust

fn main() {

let x = 5;

let y = 10;

println!("x is {}", y / x);

println!("ONE\_HOUR\_IN\_SECONDS is {}", ONE\_HOUR\_IN\_SECONDS);

}

```

我们可以看到编译器自动推导 `x` 和 `y` 的类型均为 `felt252`，这导致除法无法完成。此时，我们可以直接指定类型。

```rust

fn main() {

let x: u8 = 5;

let y = 10;

println!("x is {}", y / x);

println!("ONE\_HOUR\_IN\_SECONDS is {}", ONE\_HOUR\_IN\_SECONDS);

}

```

在这里，我们仅指定了 `x` 的类型，而 `y` 的类型则交给编译器自动推导。

在 cairo 中，存在 u8 u16 u32 u64 u128 u256 等类型的整数，分别代表 8 16 32 64 128 和 256 位的整数，这些整数具有符合直觉的四则运算，如除法会自动向下取整。此处可以需要注意 `u256`，因为 cairoVM 实际上仅支持最大 252 bit 数据，所以 `u256` 是由 2 个 `u128` 拼接获得的，我们会在后续课程中看到这一点对函数调用的影响。

```rust

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

struct u256 {

low: u128,

high: u128,

}

```

对于数值型类型，我们有一种更加快捷的声明方案，如下:

```rust

fn main() {

let x = 5\_u8;

let y = 7;

println!("x is {}", y / x);

println!("ONE\_HOUR\_IN\_SECONDS is {}", ONE\_HOUR\_IN\_SECONDS);

}

```

我们可以使用 `\_u8` 等后缀声明某一个数值类型。

对于大部分开发者而言，类型转换是一个比较复杂的问题。给出以下代码:

```rust

fn main() {

let x = 5\_u8;

let y = 7\_u16;

println!("x is {}", y / x);

}

```

报错显示类型不匹配，预期 `x` 为 `u16` 类型但传入了 `u8` 类型。这里需要进行一次类型转换，在 Cairo 中，最常见的类型转换方案就是调用 `into` 函数，如下:

```rust

fn main() {

let x = 5\_u8;

let y = 7\_u16;

println!("x is {}", y / x.into());

}

```

`into` 函数会自动根据所需要的类型进行类型转换，所有的数值类型都可以使用 `into` 实现自动的向上类型转化，如此处的 `u8` 转化 `u16` 等。可能大家好奇是否有办法进行 `u16` 转化 `u8` 呢？答案是可以的，但是众所周知，`u16` 转 `u8` 是一种危险操作，直接使用 `into` 会出现报错。

```rust

fn main() {

let x = 5\_u16;

let y = 7\_u8;

println!("x is {}", y / x.into());

}

```

我们需要使用另一种方法实现，如下:

```rust

fn main() {

let x = 5\_u16;

let y = 7\_u8;

println!("x is {}", y / x.try\_into().unwrap());

}

```

此处的 `try\_into` 用于 “危险” 的数值类型转化，如此处的 `u16` 转 `u8`，该函数会根据转化是否成功返回 `Option<T>` 类型，该类型定义如下:

```rust

pub enum Option<T> {

Some: T,

None,

}

```

我们可以直接使用 `unwrap` 解包装获得 `T` 类型，当然，如果 `Option<T>` 为 `None` 则直接报错。

在介绍完数值类型后，我们介绍一下 cairo 中的字符串类型。目前 cairo 中存在短字符串和长字符串两类，所谓短字符串即使用 `felt252` 存储字符，仅支持 252 位。而长字符串则是 `cairo 2.4` 引入的新类型。

我们首先展示短字符串，如果您希望编写的合约现在可以部署到主网中，使用短字符串仍是唯一选择。

```rust

fn main() {

let x = 'Hello World';

println!("x is {}", x);

}

```

我们可以看到输出为 `87521618088882533792115812` ，这实际是就是 `Hello World` 的 ascii 编码，这里我们也可以进一步认识到短字符串本质上就是 `felt252`

最新版本的 cairo 增加了长字符串支持，但目前由于主网暂时无法使用，我们在此处仅进行简单介绍:

```rust

fn main() {

let x = 25\_u16;

println!("{}", format!("https://example.com/{}", x));

let y: ByteArray = "Hello World";

println!("{}", y)

}

```

此处使用 `format!` 宏进行字符串格式化，目前已支持数值类型向字符串的转化。声明字符串类型则需要使用 `ByteArray` 类型注释和双引号。

我们会在后续文章内介绍 bool 类型以及元组类型。

### 函数

在上文中，我们只使用了 `main` 函数进行开发，显然，在正常开发中，我们也需要定义其他函数。我们首先编写 `sum` 函数:

```rust

fn main() {

println!("{}", sum\_three(1, 2, 3))

}

fn sum\_three(a: u32, b: u32, c: u32) -> u32 {

a + b + c

}

#[cfg(test)]

mod tests {

use super::sum\_three;

#[test]

fn it\_works() {

assert(sum\_three(1, 2, 3) == 6, 'Sum Fail');

}

}

```

在此处，我们使用 `fn` 声明了 `sum\_three` 函数，并设置了三个参数，这里需要注意，在 `cairo` 里编写函数需要给定函数返回值的类型。然后，编写函数体，cairo 默认返回最后一个表达式的值，而不需要使用 `return` 关键词。这是与 `rust` 语言一致的。

在此处，我们也对函数进行测试，使用 `assert` 判断结果是否正确，如果 `sum\_three(1, 2, 3)` 不等于 6 ，该断言则会报错，报错返回为 `Sum Fail`。

### 控制流

`if` 语句与其他语言类似，如下:

```rust

fn min(a: u32, b: u32) -> u32 {

if a <= b {

a

} else {

b

}

}

```

如果存在多个需要判断的条件则需要使用 `else if` 语句，如下:

```rust

fn min(a: u32, b: u32, c: u32) -> u32 {

if (a <= b) & (a <= c) {

a

} else if (b <= a) & (b <= c) {

b

} else {

c

}

}

```

此处我们使用 `&` 进行了联合条件判断，当然，cairo 事实上也支持 `||` OR 联合条件判断。

除了 `if` 外，另一个重要的控制流是循环，在 cairo 里，目前仅支持 `loop` 循环。我们重新回到 `fib` 序列，如下:

```rust

fn fib(mut a: felt252, mut b: felt252, mut n: felt252) -> felt252 {

loop {

if n == 0 {

break a;

}

n = n - 1;

let temp = b;

b = a + b;

a = temp;

}

}

```

此处的 `mut` 代表该参数是可变的，与变量声明时的 `mut` 作用是一致的。

编写 `loop` 的核心是编写 `break` 语句以判断跳出循环的条件。目前 cairo 不支持 `for` 循环等，我们会在后文介绍数组类型时进一步提到 `loop` 循环的作用。当然，cairo 实际上更推崇递归代替循环，但递归代替循环对于很多开发者而言过于抽象，所以 cairo 引入了 `loop` 语法简化循环创建。

### 数组

数组类型是 cairo 中最常见的集合类型，我们可以看一下标准库内对数组相关的函数实现:

1. `append` 函数用于在原数组后增加元素，此处的 `ref` 等关键词会在后续课程介绍

2. `pop\_front` 函数则是用于在原数组内获取元素

3. `get` 和 `at` 都用于获取指定索引的元素，但是两者的返回值类型不同

4. `len` 用于获取数组长度

5. `span` 一种特殊的用于所有权的函数，我们会在后续课程内介绍

我们首先展示常规的数组创建以及 `sum` 求和，如下:

```rust

fn main() {

let a = array![1, 2, 3];

println!("{}", sum\_array(a));

}

fn sum\_array(mut arr: Array<u32>) -> u32 {

let mut sum: u32 = 0;

loop {

match arr.pop\_front() {

Option::Some(current\_value) => {

sum += current\_value;

},

Option::None => {

break;

},

};

};

sum

}

```

此处使用了 `pop\_front` 从数组内获取元素，此处注意获取到的元素为 `Option<T>` 类型，我们需要对其进行 `match` 匹配操作，当我们获得 `Option::None` 时就说明当前数组遍历完成，可以直接跳出循环。

### 所有权

所有权是 cairo 中最核心的概念之一，与 rust 不同，目前的 cairo 的所有权都是比较简单的。我们首先看一下当前的所有权系统，编写以下代码:

```rust

fn main() {

let a = array![1, 2, 3];

println!("{}", sum\_array(a));

a.len();

}

```

我们可以看到此处编译器已经给出了报错 `Variable was previously moved.` 变量所有权已经被转移。这是因为我们在 `sum\_array` 中使用了 `a` 变量，其所有权被转移到了 `sum\_array` 函数中，我们无法继续在 `main` 函数中使用此变量。

我们可以看到另一个例子:

```rust

fn main() {

let (a, b, c) = (1, 2, 3);

let d = sum\_three(a, b, c);

println!("{}", a);

}

fn sum\_three(a: u256, b: u256, c: u256) -> u256 {

a + b + c

}

```

此处大家可能发现为什么 `a` 的所有权没有转移？这是因为对于 `felt252` 和整数类型而言，其都实现了 `Copy` 属性，即当该值被调用时都会自动复制一份作为参数传入另一个函数。总结来说，对于简单类型，如数值类型，我们不需要考虑所有权问题。

现在我们继续探索如何解决上述 `sum\_array` 问题，我们可以使用两种方案解决:

1. 快照传参

2. 可变引用传参

在继续讨论之前，我们首先简化一下上述案例:

```rust

fn main() {

let a = array![1, 2, 3];

println!("{}", return\_len(a));

println!("{}", a.len());

}

fn return\_len(arr: Array<u32>) -> u32 {

arr.len()

}

```

与 `sum\_array` 类似，此时参数所有权被转移给了 `return\_len` 导致后续 `a.len();` 报错。我们首先解决此问题，最简单的方法就是快照传参，如下:

```rust

fn main() {

let a = array![1, 2, 3];

println!("{}", return\_len(@a));

println!("{}", a.len());

}

fn return\_len(arr: @Array<u32>) -> u32 {

arr.len()

}

```

使用 `@` 符号可以获得某一个变量的不可变引用，如此就可以规避所有权问题。但是需要特殊注意的是，对于 `@Array<u32>` 而言，我们通常使用其另一个写法，如下:

```rust

// Span.

struct Span<T> {

snapshot: @Array<T>

}

```

> 125 行

此处值得注意 `Span<T>` 实现的功能远远多于 `@Array<T>` ，所以大家请尽可能多使用 `Span<T>` 而不是更加原始的 `@Array<T>`。

有了此基础，我们可以尝试将 `sum\_array` 修改一下，如下:

```rust

fn sum\_array(mut arr: Span<u32>) -> u32 {

let mut sum: u32 = 0;

loop {

match arr.pop\_front() {

Option::Some(current\_value) => {

sum += \*current\_value;

},

Option::None => {

break;

},

};

};

sum

}

```

此处需要注意 `Span<u32>` 的 `pop\_front` 方法获得的是 `Option<@T>` 类型，所以此处需要使用 `\*` 进行解引用。此处可能比较奇怪，不可变引用实际上也可以被标注为 `mut` 进行修改。

进行上述操作后，我们就可以修复之前错误。

```rust

fn main() {

let a = array![1, 2, 3];

println!("{}", sum\_array(a.span()));

println!("{}", a.len());

}

```

通过观察输出，我们可以发现在 `sum\_array` 中对 `a` 参数的修改没有返回到 `main` 函数的变量 `a` 中。

> 这里可以介绍一下 `@` 快照类型的本质，其本质实际上就是将原数据复制一份用于函数操作

接下来我们可以使用一下另一个策略，即输入可变引用，如下:

```rust

fn main() {

let mut a = array![1, 2, 3];

println!("Sum is {}", sum\_array(ref a));

println!("Length is {}", a.len());

}

fn sum\_array(ref arr: Array<u32>) -> u32 {

let mut sum: u32 = 0;

loop {

match arr.pop\_front() {

Option::Some(current\_value) => {

sum += current\_value;

},

Option::None => {

break;

},

};

};

sum

}

```

此处我们使用 `ref` 标识，使用关键词的变量意味着其所有权不会变化，但函数对其修改会返回至参数上。运行代码，我们可以观察到其输出，如下:

```bash

Sum is 6

Length is 0

Run completed successfully, returning []

```

由于 `sum\_array` 中使用了 `pop\_front` 函数导致 `a` 数组被清空，这一函数内的变化也被传导到了 `main` 函数内，使其长度改变为 0

在此处，我们可以总结 cairo 中的所有权规则如下:

1. 当使用简单的非集合类型时，无需关注所有权问题

2. 当遇到所有权问题且不希望函数修改原变量值，使用 `@` 或 `Span<T>` 进行快照传参

3. 当遇到所有权问题且希望函数直接修改原变量值的情况下，使用 `ref` 传参

### 结构体

结构体是 cairo 内最常见的类型之一，也存在许多特性，在本节课中，我们可能无法完整介绍所有特性，此处仅介绍一些较为重要的属性。

我们首先给出一个结构体的定义:

```rust

struct Rectangle {

width: u64,

height: u64,

}

fn main() {

let rectangle = Rectangle { width: 30, height: 10, };

let area = area(@rectangle);

println!("Area: {}", area);

}

fn area(rectangle: @Rectangle) -> u64 {

\*rectangle.width \* \*rectangle.height

}

```

在此处，考虑到 `Rectangle` 是一个复杂类型且我们不希望转移其所有权，所以使用了 `@` 快照类型，但是我们可以看到此处的代码报错 `Variable not dropped.`，其含义为变量无法被丢弃。由于此处我们没有使用函数消耗变量的所有权，所以在 `main()` 函数中 `rectangle` 一直存在，且由于该结构体没有实现 `Drop` 属性使其无法被丢弃，这是一个错误。我们可以借助结构体的宏解决此问题，如下:

```rust

#[derive(Drop)]

struct Rectangle {

width: u64,

height: u64,

}

```

使用 `#[derive(Drop)]` 将为结构体自动衍生 `Drop` 属性使其可以被丢弃。

我们接下来尝试编写一个内置宏，我们的目标是使用 `println!` 打印 `rectangle` 变量，如下:

```rust

fn main() {

let rectangle = Rectangle { width: 30, height: 10, };

let area = area(@rectangle);

println!("Area: {}", area);

println!("Rectangle: {}", rectangle)

}

```

报错为 `Trait has no implementation in context: core::fmt::Display::<hello\_world::Rectangle>`，我们直接为 `Rectangle` 实现此 `trait` 如下:

```rust

impl RectangleDisplay of core::fmt::Display::<Rectangle> {

fn fmt(self: @Rectangle, ref f: core::fmt::Formatter) -> Result<(), core::fmt::Error> {

write!(f, "width: ")?;

core::fmt::Display::fmt(self, ref f)?;

write!(f, "height: ")?;

core::fmt::Display::fmt(self.height, ref f)

}

}

```

此处的 `write!` 就是输出指定内容的宏，这里也使用了 `?` 进行错误传递。

我们也可以为结构体附属一些函数，如下:

```rust

#[derive(Drop)]

struct Rectangle {

width: u64,

height: u64,

}

trait RectangleTrait {

fn area(self: @Rectangle) -> u64;

}

impl RectangleImpl of RectangleTrait {

fn area(self: @Rectangle) -> u64 {

(\*self.width) \* (\*self.height)

}

}

fn main() {

let rect1 = Rectangle { width: 30, height: 50, };

println!("Area: {}", rect1.area());

}

```

此处使用了 `RectangleTrait` 为 `Rectangle` 结构体提供了 `area` 函数。

### 泛型

我们希望将 `Rectangle` 设置为任何类型都可以使用，此时就涉及到泛型。我们首先定义泛型的结构体:

```rust

#[derive(Drop)]

struct Rectangle<T> {

width: T,

height: T,

}

```

然后，我们需要定义泛型函数 `area` 如下:

```rust

trait RectangleTrait<T> {

fn area(self: @Rectangle<T>) -> T;

}

impl RectangleImpl<T> of RectangleTrait<T> {

fn area(self: @Rectangle<T>) -> T {

\*self.width \* \*self.height

}

}

```

我们可以看到此处报错为 `Trait has no implementation in context: core::traits::Mul::<T>` 即此处使用的泛型参数 `T` 没有实现 `core::traits::Mul::<T>` 属性，我们可以使用以下泛型规范修复此问题:

```rust

impl RectangleImpl<T, +Mul<T>> of RectangleTrait<T> {

fn area(self: @Rectangle<T>) -> T {

\*self.width \* \*self.height

}

}

```

然后我们发现另一个报错 `note: Trait has no implementation in context: core::traits::Copy::<T>`，我们也如法炮制修复一下:

```rust

impl RectangleImpl<T, +Mul<T>, +Copy<T>> of RectangleTrait<T> {

fn area(self: @Rectangle<T>) -> T {

\*self.width \* \*self.height

}

}

```

最后，我们可以看到所有报错消失。

与结构体的 `trait` 不同，泛型函数有另一套定义方法，我们使用 `sum\_array` 函数为例介绍泛型函数，我们首先对其进行泛型定义:

```rust

fn sum\_array<T>(mut arr: Span<T>) -> T {

let mut sum = 0;

loop {

match arr.pop\_front() {

Option::Some(current\_value) => { sum += \*current\_value; },

Option::None => { break; },

};

};

sum

}

```

此时报错显示 `Trait has no implementation in context: core::traits::AddEq::<T>`，我们直接增加此 `trait` ，如下:

```rust

fn sum\_array<T, +AddEq<T>>(mut arr: Span<T>) -> T {}

```

此时 `let mut sum = 0;` 报错，我们需要初始化一个泛型下的零，此时就需要使用 `let mut sum = core::Zeroable::zero();` 函数初始化泛型下的零。此时又出现了 `Trait has no implementation in context: core::zeroable::Zeroable::<T>` 报错，我们需要手动增加此 `trait` ，如下:

```rust

fn sum\_array<T, +AddEq<T>, +core::Zeroable<T>>(mut arr: Span<T>) -> T {

```

此时报错提醒我们泛型参数未实现 `Drop::<T>`，我们也修改函数定义已消除此错误:

```rust

fn sum\_array<T, +AddEq<T>, +core::Zeroable<T>, +Drop<T>>(mut arr: Span<T>) -> T {

```

最后，我们发现解引用报错 `Cannot desnap a non copyable type.`，这说明我们需要增加另一个 `trait`，如下:

```rust

fn sum\_array<T, +AddEq<T>, +core::Zeroable<T>, +Drop<T>, +Copy<T>>(mut arr: Span<T>) -> T {

```

## starkli

`starkli` 工具是目前最常用的合约交互和部署工具，提供了大量 starknet 上的常用功能。在本节课中，我们主要进行 starkli 的账户配置和合约交互。

第一步安装 `starkli` 使用以下命令:

```bash

curl https://get.starkli.sh | sh

starkliup -v v0.1.20

```

接下来，创建私钥保存仓库:

```bash

mkdir ~/.starknet\_accounts

starkli signer keystore new ~/.starknet\_accounts/key.json

```

此处可以将密钥文件的位置写入环境变量:

```bash

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

```

初始化账户并预计算账户地址:

```bash

starkli account oz init ~/.starknet\_accounts/starkli.json

```

配置 RPC 地址:

```bash

export STARKNET\_RPC=https://starknet-testnet.public.blastapi.io

```

部署账户:

```bash

starkli account deploy /home/codespace/.starknet\_accounts/starkli.json

```

配置环境变量:

```bash

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

```

调用合约的 `name` 函数:

```bash

starkli call 0x0091efcd6807d63d83ceb1ce1912c039d7533cfbe54e820711ce406e726b2d4a name

```

调用 mint 函数:

```bash

starkli invoke 0x0091efcd6807d63d83ceb1ce1912c039d7533cfbe54e820711ce406e726b2d4a mint u256:1000

```

获取账户余额:

```bash

starkli call 0x0091efcd6807d63d83ceb1ce1912c039d7533cfbe54e820711ce406e726b2d4a balanceOf 0x07f65c594867fb1b65cd90da48fdb2e80bccd06caa9c4db995410ef66b0f757b

```

作业:

1.判断以下程序的返回值:

```rust

fn main() {

let mut x = 5;

add\_one(ref x);

println!("{}", x)

}

fn add\_one(ref x: u32) -> u32 {

x + 1

}

```

A. 6

B. 5

C. 0

D. 上述代码无法通过编译

2.判断以下程序的返回值:

```rust

fn main() {

let mut x = array![1, 2, 3];

let y = pop\_len(ref x);

}

fn pop\_len(ref arr: Array<felt252>) -> felt252 {

match arr.pop\_front() {

Option::Some(current\_value) => {

current\_value

},

Option::None => {

0

},

}

}

```

A. x = 3, y = 0

B. x = 2, y = 1

C. x = 2, y = 3

D. x = 3, y = 1

3.已知当前代码报错 `Variable not dropped.` 如何修复以下代码:

```rust

struct Wallet<T> {

balance: T

}

fn main() {

let w = Wallet { balance: 3 };

}

```

A. 使用 `#[derive(Copy)]` 为结构体派生宏

B. 使用 `#[derive(Drop)]` 为结构体派生宏

C. 使用 `#[derive(PartialEq)]` 为结构体派生宏