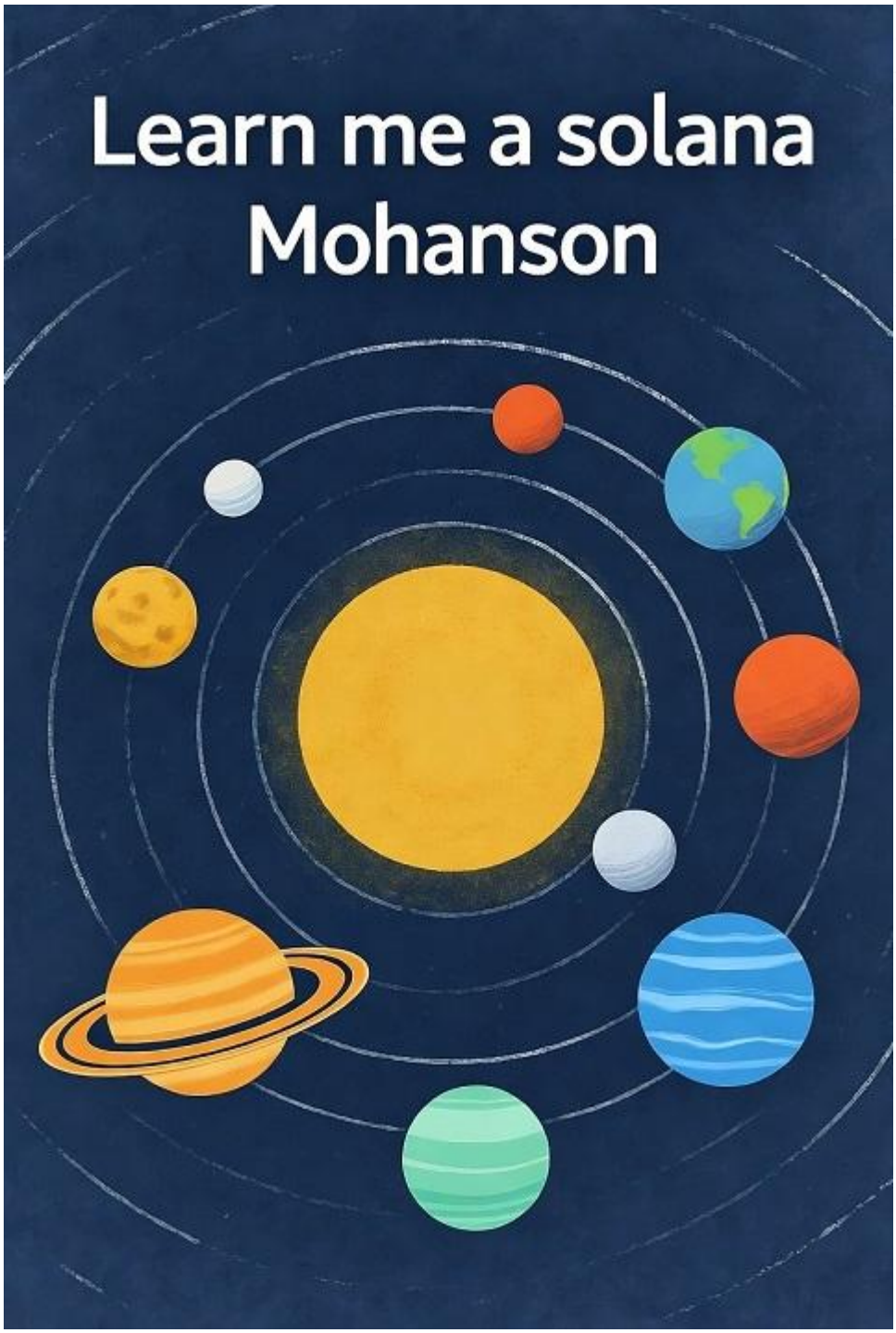
Learn Me A Solana

A comprehensive guide to Solana development

Mohanson

Table of contents

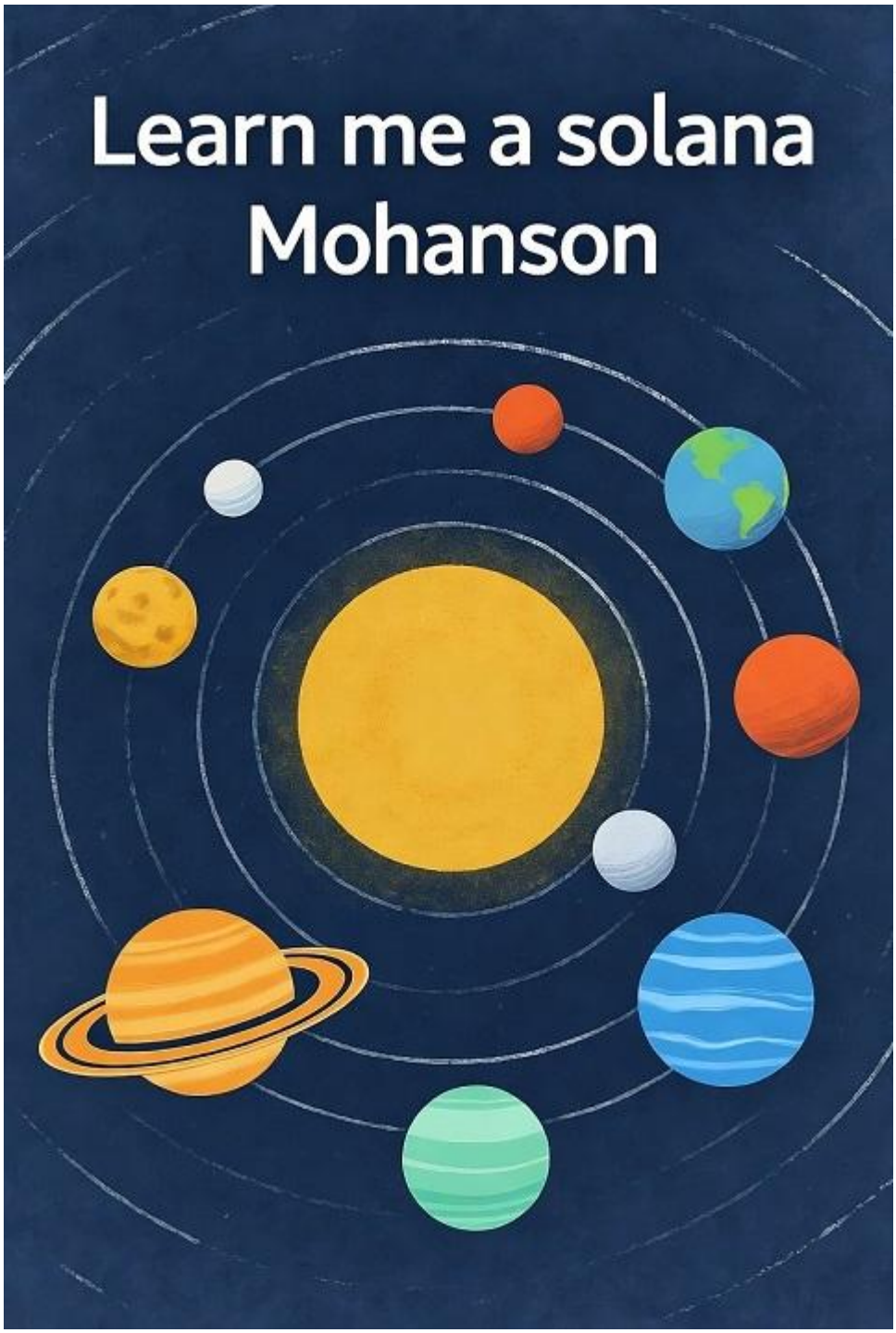


1. Solana/Learn Me A Solana

在这个科技日新月异的时代, 似乎人类变得比以往任何时候都焦虑和急躁. 当清晨的阳光穿过层层云彩, 在地上投下温暖的光影时, 我总会想起 2017 年, 那是我经历的第一个加密货币牛市. 我和我的朋友们会仔细研究各个项目的白皮书, 寻找他们有价值的地方, 并最终决策是否应该成为它们的早期投资者. 转眼间时间来到了 2025 年, solana 和 meme coin 成为了本轮牛市的主要叙事点. 如今, 一个代币项目的平均存活期可能仅仅只有一到两天, 寥寥数百个用户, 便会立即无人问津, 躺在谁也不会关心的角落等待死去.

这本 Learn Me A Solana, 不像那些关于区块链之类的常规读物那样引人入胜, 也不是一本能让您快速暴富的投资学著作. 它的内核非常简单: 我将向读者们展示 solana 构建的数字文明. Solana 的设计理念与比特币或以太坊并不完全一致, 甚至可以说它们的某些核心观念南辕北辙. 但无论我们是赞同或者是否定它, 首先我们必须先了解它.

或许是对未知的好奇, 或许是对可能的期待. 时隔多年, 我再次决定写一本书.



配套源码

课程主要使用 python 语言. 在第一章, 我们会使用到 pabtc, 它是一个比特币库. 在之后的章节, 我们主要使用到 solana 的 pxsol 库. 这两个库和本书的作者为同一个人.

- Pabtc: https://github.com/mohanson/pabtc
- Pxsol: https://github.com/mohanson/pxsol

语言

本书提供中文和英文版. 您可以根据自己的喜好选择在线阅读或者离线下载.

- English - Markdown
- English - PDF
- English - Web
- 中文 - Markdown
- 中文 - PDF
- 中文 - Web

小额赞助

- SOL: `Dd9i1im9RmaaxmdSrezjN4unqAPLF4HQGEgGrNDGKhtN`

搜索提示

- Solana 教程
- Solana 中文教程

2. Solana/私钥, 公钥与地址

2.1 Solana/私钥, 公钥与地址/导言

一位学生正困惑地望着电脑屏幕, 屏幕上显示着 solana 钱包的相关信息.

学生: "老师, 我已经学会了使用 solana 钱包发送和接收硬币, 但..."

学生欲言又止. 在刚才的课上, 教授在教学如何使用钱包.

教授: "哦, 我看得出来你非常疑惑."

学生: "老师, 我不是很明白这些术语的具体含义, 关于私钥, 公钥和地址, 尤其是它们之间的关系是什么样的. 能不能请您详细解释一下?"

教授: "当然可以! 让我们先从最基本的原理开始理解. 在 solana 的世界里, 每个钱包都像一个独特的账户, 里面存放着我们的硬币. 要访问这些硬币, 我们需要使用一种特殊的机制, 私钥."

学生: "哦, 老师, 那私钥是用来做什么的呢?"

教授: "没错! 私钥就像是我们保存在安全地方的钥匙. 它用于签名交易, 确保资金确实是我们自己转账的. 简单来说, 这就是我们对属于自己的 solana 硬币的一种**证明**. 有了私钥, 我们就可以告诉世界**这里是我的钱**."

学生: "明白了! 那公钥又是怎么回事呢?"

教授: "公钥就像是私钥的一个副本, 只不过它被公开到区块链上了. 每当我使用私钥创建一个交易时, 这个交易会被记录在 solana 的区块链上, 并伴随着我的公钥."

学生: "哦, 我有点明白了! 所以公钥就像是一串数字和字母的字符串, 它代表着我的钱包地址?"

教授: "正是如此! 每一个公钥都唯一对应着一个钱包地址. 地址就像是我的钱包的身份证, 这样, 别人看到这个地址就知道这是一个属于我自己的钱包."

学生: "那为什么多数时候我在网络上看到的是地址, 而不是公钥呢?"

教授: "这是因为地址更为直观和简洁. 它由字母和数字组成, 是其他人可以轻松访问的链接. 而公钥则是一串复杂的二进制代码, 不太适合直接在公共场合使用."

学生: "哦, 明白了! 那么, 总结一下, **私钥是用来签名交易的, 公钥则是私钥的一部分副本, 被记录在区块链上, 而地址则是基于公钥生成的一个独特的标识符**. 这样, 别人看到我的地址就能知道这是一个属于我自己的钱包."

教授: "完全正确! 这就是 solana 系统中私钥, 公钥和地址之间的关系. 希望通过这个简单的解释, 你们能更好地理解这些基础概念. 如果还有其他问题, 请随时提问!"

2.2 Solana/私钥, 公钥与地址/私钥

Solana 是一个高性能的区块链平台, 安全性方面自然不能忽视. 私钥是用于签名交易和确保资金安全的关键部分. 它是一个大数, 更具体地来说，是 0 到 2^{256} 之间的任何数字(包含前者, 不包含后者). 在内存中, 通常使用长度为 32 的字节数组来储存它.

私钥标识用户的所有资产. 在交易中, 通过私钥创建签名来证明用户拥有硬币的控制权, 以便将硬币转移给他人. 因此, 用户必须始终保管好自己的私钥, 遗忘私钥等于丧失了自己硬币的控制权, 泄露私钥等于将硬币的控制权共享给他人.

现代的大多数区块链钱包, 都隐藏了针对私钥的操作, 让用户自己操作私钥, 尤其是对技术不甚了解的用户, 其结果通常是灾难性的. 因此现代钱包更为流行一种叫做助记词的私钥变体, 当您看到助记词时, 应当了解它底层仍然是一个私钥.

回到我们的课程, 如果您想自己生成私钥, 你需要做的唯一一件事情就是生成一个长度为 32 的字节数组. 数组内的数据可以是任意的, 因此普遍的做法是寻找一个安全的随机源来填充数组. 下面演示几种不同的方式来生成 solana 的私钥.

python

使用代码生成私钥时, 最重要的一点是您不能使用伪随机数生成算法. 伪随机数是可复现的, 因此它们并不安全. 相反, 您应当使用密码学安全的真随机数. 假设操作系统始终能提供高质量, 加密安全的随机数据, 理想情况下由硬件熵源支持, 那么我们可以使用如下的代码来生成私钥.

```
import secrets

prikey = bytearray(secrets.token_bytes(32))
```

golang

```
package main

import (
    "crypto/rand"
)

func main() {
    prikey := make([]byte, 32)
    rand.Read(prikey)
}
```

rust

```
fn main() {
    let mut prikey = [0u8; 32];
    getrandom::fill(&mut prikey).unwrap();
}
```

私钥是 solana 生态系统中不可替代的最重要的组成部分. 不要使用来源不明的私钥, 也不要将您的私钥存储在互联网上, 例如网盘或在线笔记工具中. 通过遵循正确的生成和存储流程, 您可以有效管理和保护您的数字资产. 请始终记住: 安全第一!

2.3 Solana/私钥, 公钥与地址/私钥的密码学解释(一)

好吧, 这一章不是一篇易于理解的文章. 我将在这一章对 solana 的私钥做出一些密码学方面的解释, 这需要读者有一定的数学基础. 如果感觉到困难, 您完全可以跳过这一章内容, 我相信不会对后续学习产生任何阻碍. 作为我 solana 教程的一部分, 这篇文章花费了我最多的时间, 因此如果您下定决心阅读这篇文章, 相信看完之后会有一点的收获.

在正式进入密码学大门之前, 我们首先需要入门抽象代数. 代数主要研究的是运算规则. 一门代数, 其实都是从某种具体的运算体系中抽象出一些基本规则, 建立一个公理体系, 然后在这基础上进行研究. 一个集合再加上一套运算规则, 就构成一个代数结构.

2.3.1 群

代数里面的群(group)是由集合和二元运算(用符号 + 表示)构成的, 符合以下四个群公理的数学结构. 群公理的四个性质如下:

1. 加法封闭性: 对于群中的任意两个元素进行运算后, 结果仍然属于该群.
2. 加法结合律: 群中的运算满足结合律, 即对于群中的任意三个元素进行运算, 先计算前两个元素的运算结果, 然后再与第三个元素进行运算, 结果应该与先计算后两个元素的运算结果再与第三个元素进行运算的结果相同.
3. 加法单位元: 群中存在一个特殊的元素, 称为单位元, 对于群中的任意元素 a, 运算 a 与单位元的结果等于元素 a 本身, 即 a + e = e + a = a, 其中 e 表示单位元.
4. 加法逆元素: 群中的每个元素都有一个逆元素, 对于群中的任意元素 a, 存在一个元素 b, 使得 a + b = b + a = e, 其中 e 表示单位元.

如果我们添加第五条要求:

1. 加法交换律: a + b = b + a.

那么这个群就是**阿贝尔群**或**交换群**.

例: 整数集是否构成群? 自然数集呢?

答: 整数集是一个阿贝尔群. 自然数集不是一个群, 因为它不满足第四条群公理.

群拥有几个概念, 会在之后的文章中被提及.

1. 一个有限群的元素个数称为群的阶.
2. 一个群元素 p 的阶为最小的整数 k 使得 k * p 等于单位元.
3. 一个群元素 p 的阶如果等于群的阶, 则 p 称为该群的生成元, 且该群是一个循环群.

2.3.2 环

环在群的基础上多定义了一个群运算乘法, 用符号 x 表示.

1. 乘法封闭性: 对于环中的任意两个元素进行乘法运算后, 结果仍然属于该环.
2. 乘法结合律: 环中的乘法运算满足结合律, 即对于环中的任意三个元素进行乘法运算, 先计算前两个元素的乘法结果, 然后再与第三个元素进行乘法运算, 结果应该与先计算后两个元素的乘法结果再与第三个元素进行乘法运算的结果相同.
3. 乘法分配律: 环中的乘法运算对于加法运算满足左分配律和右分配律, 即对于环中的任意三个元素 a, b, c, 有 a * (b + c) = a * b + a * c 和 (b + c) * a = b * a + c * a.

2.3.3 域

域是一种代数结构, 由一个集合和两个二元运算(加法和乘法)组成. 域满足环的所有条件, 并且具有以下额外性质:

1. 乘法单位元: 域中存在一个特殊的元素, 称为乘法单位元, 对于域中的任意元素 a, a 与乘法单位元相乘的结果等于 a 本身, 即 a * 1 = 1 * a = a, 其中 1 表示乘法单位元.
2. 乘法逆元素: 对于域中的任意非零元素 a, 存在一个元素 b, 使得 a * b = b * a = 1, 其中 1 表示乘法单位元.

例: 整数集, 有理数集, 实数集和复数集是否构成域?

答: 整数集构成一个环但不构成域. 有理数集, 实数集和复数集均构成域.

2.3.4 素数有限域

有限域是包含有限个元素的域. 与其他域一样, 有限域是进行加减乘除运算都有定义并且满足特定规则的集合. 有限域最常见的例子是素数域, 也就是整数集对指定素数取模后构成的集合.

例: 有素数有限域为整数集对素数 23 取模, 求以下算式的值.

- 12 + 20
- 8 * 9
- 1 / 8

答:

- 12 + 20 = 32 % 23 = 9
- 8 * 9 = 72 % 23 = 3
- 由于 3 * 8 = 24 % 23 = 1, 因此 1 / 8 = 3

下面我们使用 python 代码实现一个素数有限域. 客观来讲, 它十分类似我们日常生活中使用的整数, 但区别在于我们需要对所有计算结果进行取模. 下面的代码拷贝自 pabtc 项目, 您可以使用 `pip install pabtc` 来获取这份代码.

```
import json
import typing


class Fp:
    # Galois field. In mathematics, a finite field or Galois field is a field that contains a finite number of elements.
    # As with any field, a finite field is a set on which the operations of multiplication, addition, subtraction and
    # division are defined and satisfy certain basic rules.
    #
    # https://www.cs.miami.edu/home/burt/learning/Csc609.142/ecdsa-cert.pdf
    # Don Johnson, Alfred Menezes and Scott Vanstone, The Elliptic Curve Digital Signature Algorithm (ECDSA)
    # 3.1 The Finite Field Fp

    p = 0

    def __init__(self, x: int) -> None:
        self.x = x % self.p

    def __add__(self, data: typing.Self) -> typing.Self:
        assert self.p == data.p
        return self.__class__(self.x + data.x)

    def __eq__(self, data: typing.Self) -> bool:
        assert self.p == data.p
        return self.x == data.x

    def __mul__(self, data: typing.Self) -> typing.Self:
        assert self.p == data.p
        return self.__class__(self.x * data.x)

    def __neg__(self) -> typing.Self:
        return self.__class__(self.p - self.x)
```

```
    def __repr__(self) -> str:
        return json.dumps(self.json())

    def __sub__(self, data: typing.Self) -> typing.Self:
        assert self.p == data.p
        return self.__class__(self.x - data.x)

    def __truediv__(self, data: typing.Self) -> typing.Self:
        return self * data ** -1

    def __pos__(self) -> typing.Self:
        return self.__class__(self.x)

    def __pow__(self, data: int) -> typing.Self:
        return self.__class__(pow(self.x, data, self.p))

    def json(self) -> str:
        return f'{self.x:064x}'

    @classmethod
    def nil(cls) -> typing.Self:
        return cls(0)

    @classmethod
    def one(cls) -> typing.Self:
        return cls(1)
```

使用代码验证上面的例题如下.

```
Fp.p = 23
assert Fp(12) + Fp(20) == Fp(9)
assert Fp(8) * Fp(9) == Fp(3)
assert Fp(8) ** -1 == Fp(3)
```

您可能注意到了, 有限域的除法是一个特殊情况. 当我们试图求 a / b 时, 我们实际上需要求的是 $a * b^{-1}$. 根据费马小定理, $b^{p-1} = 1$, 因此有 $b * b^{p-2} = 1$, 因此 $b^{-1} = b^{p-2}$. 所以 a / b 等价于 $a * b^{p-2}$.

2.4 Solana/私钥, 公钥与地址/私钥的密码学解释(二)

在我们深入探讨 solana 的私钥之前, 首先需要了解一种称为"公私钥密码学"的技术. 这是一种被广泛应用于现代信息安全领域的核心技术, 也是 solana 安全运行的基础. 公私钥密码学, 也被称为"非对称加密", 是一种基于数学算法的加密方法. 其核心在于使用一对密钥: 公钥和私钥.

公钥

- 公钥是可以公开使用的密钥, 它的主要作用是加密数据或验证签名.
- 有了公钥, 其他人可以安全地加密信息或对你的数据进行认证.

私钥

- 私钥是只有你一个人知道的密钥, 它的主要作用是解密信息或创建签名.
- 如果别人用你的公钥加密的数据, 只有你才能用你的私钥来解开并读取内容.

公私钥密码学的数学基础可以追溯到 20 世纪 70 年代. 最早的尝试之一是 diffie 和 hellman 在 1976 年提出的 "关键密钥交换协议", 但当时并没有广泛应用于实际系统. 随后, rsa 加密算法在 1977 年由 ron rivest, adi shamir 和 len adleman 提出, 成为公私钥密码学的经典方案. 该方案基于数论中的大质数分解难题, 被认为是安全的加密方法之一. 椭圆曲线密码学则是 rsa 的一种替代方案, 其优势在于使用更短的密钥长度即可达到相同的安全水平, 它的数学基础是椭圆曲线离散对数问题.

得益于比特币的发展, 比特币所采用的 secp256k1 椭圆曲线以及 ecdsa 签名算法在全球变得广为人知, 并产生了非常巨大的影响: 包括以太坊, ckb 等众多区块链项目在内, 都采用了和比特币相同的密码学算法. 但 solana 在这方面有点不一样, 它采用的是一种新型的椭圆曲线数字签名算法, 其曲线名称 ed25519, 签名算法为 eddsa.

我们不禁要思考, 促成 solana 改变的原因是什么? 为什么要和比特币不一样?

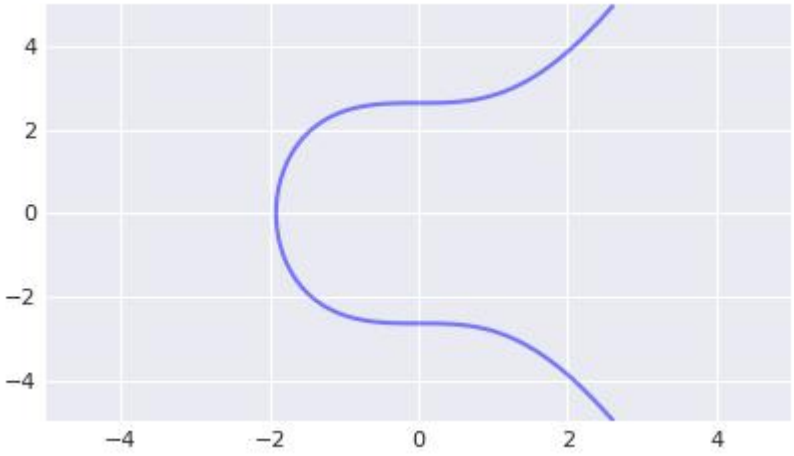
为了解释这方面的原因, 在接下来的几小节中, 我将带领读者深入学习比特币的 secp256k1 + ecdsa 签名方案, 以及更重要的: 该方案背后所隐藏着的巨大的安全性问题.

2.5 Solana/私钥, 公钥与地址/私钥的密码学解释(三)

Secp256k1 是比特币采用的标准椭圆曲线, 基于 koblitz 曲线($y^2 = x^3 + ax + b$). 其参数与美国安全局推荐使用的 p-256 类似, 但存在细微修改. 其表达式为

```
y² = x³ + 7
```

在实数域下, 它的图像是一个上下对称的曲线.



> P-256 是另一类被广泛使用的椭圆曲线, secp256k1 与其的区别只有参数不同.

2.5.1 Secp256k1 曲线

Secp256k1 实际上是一种基于素数有限域进行计算的技术. 该素数等于:

```
# Equals to 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
P = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
```

我们可以使用 python 来实现 secp256k1 的方程.

```
# Prime of finite field.
P = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f

class Fq(Fp):
    p = P

A = Fq(0)
B = Fq(7)


class Pt:

    def __init__(self, x: Fq, y: Fq) -> None:
        if x != Fq(0) or y != Fq(0):
            assert y ** 2 == x ** 3 + A * x + B
        self.x = x
        self.y = y
```

例: 有如下 (x, y), 请判断其是否位于 secp256k1 曲线上.

```
import pabtc

x = pabtc.secp256k1.Fq(0xc6047f9441ed7d6d3045406e95c07cd85c778e4b8cef3ca7abac09b95c709ee5)
y = pabtc.secp256k1.Fq(0x1ae168fea63dc339a3c58419466ceaeef7f632653266d0e1236431a950cfe52a)
```

答:

```
assert y ** 2 == x ** 3 + A * x + B
```

2.5.2 Secp256k1 点的加法

椭圆曲线上的点可以构成一个加法群. 规定椭圆曲线上给定两个不同的点 p 和 q, 其加法 r = p + q, 规则如下:

- 当 p == -q 时, $p(x_1, y_1) + q(x_2, y_2) = r(x_3, y_3)$, r 被称为单位元, 其中

```
x₃ = 0
y₃ = 0
```

- 当 p == +q 时, $p(x_1, y_1) + q(x_2, y_2) = r(x_3, y_3)$, 其中

```
x₃ = ((3 * x₁² + a) / (2 * y₁))² - x * x₁
y₃ = ((3 * x₁² + a) / (2 * y₁)) * (x₁ - x₃) - y₁
```

- 当 p != ±q 时, $p(x_1, y_1) + q(x_2, y_2) = r(x_3, y_3)$, 其中

```
x₃ = ((y₂ - y₁) / (x₂ - x₁))² - x₁ - x₂
y₃ = ((y₂ - y₁) / (x₂ - x₁)) * (x₁ - x₃) - y₁
```

2.5.3 Secp256k1 点的标量乘法

在定义了加法之后, 我们可以定义标量乘法. 给定一个点 p 以及标量 k, 则 p * k 数值上等于 k 个 p 相加的和. 椭圆曲线上的乘法可以分解为一系列的 double 和 add 操作. 例如, 我们要运算 151 * p, 直观上我们会认为要进行 150 次点相加运算, 但可以进行优化. 151 可以表示为二进制格式 10010111:

```
151 = 1 * 2⁷ + 0 * 2⁶ + 0 * 2⁵ + 1 * 2⁴ + 0 * 2³ + 1 * 2² + 1 * 2¹ + 1 * 2⁰
```

规定初始结果为 0. 我们从 10010111 的最低比特位开始, 如果为 1, 则结果加 p; 如果为 0, 令 p = 2p. 相关 python 代码如下所示:

```
def bits(n):
    # Generates the binary digits of n, starting from the least significant bit.
    while n:
        yield n & 1
        n >>= 1

def double_and_add(n, x):
    # Returns the result of n * x, computed using the double and add algorithm.
    result = 0
    addend = x
    for bit in bits(n):
        if bit == 1:
            result += addend
        addend *= 2
    return result
```

2.5.4 Secp256k1 生成元

我们人为规定一个特殊点, 叫做生成点 g, 椭圆曲线上的任意点都可以表示为 g 与一个标量 k 的乘积.

```
G = Pt(
    Fq(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798),
    Fq(0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8),
)
```

2.5.5 Secp256k1 的阶

椭圆曲线上的点是有限个数的, 这个数量被称作椭圆曲线的阶. 标量 k 的取值必须小于这个数字, 对于 secp256k1 来说, 这个值为

```
# The order n of G.
N = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
```

2.5.6 Secp256k1 完整代码

最终, 我们得到完整的 secp256k1 代码如下. 您可以在 pabtc.secp256k1 中找到这份代码.

```
# Prime of finite field.
P = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
# The order n of G.
N = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141


class Fq(Fp):

    p = P


class Fr(Fp):

    p = N


A = Fq(0)
B = Fq(7)


class Pt:

    def __init__(self, x: Fq, y: Fq) -> None:
        if x != Fq(0) or y != Fq(0):
            assert y ** 2 == x ** 3 + A * x + B
        self.x = x
        self.y = y

    def __add__(self, data: typing.Self) -> typing.Self:
        # https://www.cs.miami.edu/home/burt/learning/Csc609.142/ecdsa-cert.pdf
        # Don Johnson, Alfred Menezes and Scott Vanstone, The Elliptic Curve Digital Signature Algorithm (ECDSA)
        # 4.1 Elliptic Curves Over Fp
        x1, x2 = self.x, data.x
        y1, y2 = self.y, data.y
        if x1 == Fq(0) and y1 == Fq(0):
            return data
        if x2 == Fq(0) and y2 == Fq(0):
            return self
        if x1 == x2 and y1 == +y2:
```

```
            sk = (x1 * x1 + x1 * x1 + x1 * x1 + A) / (y1 + y1)
            x3 = sk * sk - x1 - x2
            y3 = sk * (x1 - x3) - y1
            return Pt(x3, y3)
        if x1 == x2 and y1 == -y2:
            return I
        sk = (y2 - y1) / (x2 - x1)
        x3 = sk * sk - x1 - x2
        y3 = sk * (x1 - x3) - y1
        return Pt(x3, y3)

    def __eq__(self, data: typing.Self) -> bool:
        return all([
            self.x == data.x,
            self.y == data.y,
        ])

    def __mul__(self, k: Fr) -> typing.Self:
        # Point multiplication: Double-and-add
        # https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication
        n = k.x
        result = I
        addend = self
        while n:
            b = n & 1
            if b == 1:
                result += addend
            addend = addend + addend
            n = n >> 1
        return result

    def __neg__(self) -> typing.Self:
        return Pt(self.x, -self.y)

    def __repr__(self) -> str:
        return json.dumps(self.json())

    def __sub__(self, data: typing.Self) -> typing.Self:
        return self + data.__neg__()

    def __truediv__(self, k: Fr) -> typing.Self:
        return self.__mul__(k ** -1)

    def __pos__(self) -> typing.Self:
        return Pt(self.x, +self.y)

    def json(self) -> typing.Self:
        return {
            'x': self.x.json(),
            'y': self.y.json(),
        }


# Identity element
I = Pt(
    Fq(0),
    Fq(0),
)
# Generator point
```

```
G = Pt(
    Fq(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798),
    Fq(0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8),
)
```

标量 k 就是所谓的 secp256k1 私钥, 而生成点与 k 的乘积, 即 g * k 表示 secp256k1 公钥. 从私钥计算公钥是十分容易的, 而想从公钥计算私钥是相当困难的.

2.5.7 习题

例: 已知比特币私钥为 `0x5f6717883bef25f45a129c11fcac1567d74bda5a9ad4cbffc8203c0da2a1473c`, 求公钥.

答:

```
import pabtc

prikey = pabtc.secp256k1.Fr(0x5f6717883bef25f45a129c11fcac1567d74bda5a9ad4cbffc8203c0da2a1473c)
pubkey = pabtc.secp256k1.G * prikey
assert(pubkey.x.x == 0xfb95541bf75e809625f860758a1bc38ac3c1cf120d899096194b94a5e700e891)
assert(pubkey.y.x == 0xc7b6277d32c52266ab94af215556316e31a9acde79a8b39643c6887544fdf58c)
```

2.6 Solana/私钥, 公钥与地址/私钥的密码学解释(四)

在数字化世界中, 如何在不泄露密钥的前提下验证他人身份, 一直是加密学领域的终极难题. 传统的口令和证书系统虽然安全, 但易受破解; 而公私钥体系则凭借强大的数学基础, 为数字信任提供了新的可能. 基于椭圆曲线 secp256k1 的 ecdsa 签名算法不仅能够证明信息的完整性, 更重要的是能够确保发送该信息的是身份持有者.

我们将在这一小节里学习 ecdsa 签名算法里的三个常用功能, 分别是签名, 验签以及公钥恢复.

2.6.1 Ecdsa 签名

我们经常会在日常生活中的各种文书上签下自己的名字, 这通常有两层含义: 一是告诉旁人是谁签了这份文书, 二是表示您同意和认可了文书上的文字内容. 数字签名功能类似写在纸上的普通签名, 但是使用了公私钥加密技术, 以用于鉴别数字信息的方法. 与普通签名不同, 数字签名通常可以防止信息被篡改.

基于 secp256k1, 可以实现一种叫做 ecdsa 的公私钥签名算法. 您使用私钥进行签名, 他人则使用您事先提供的公钥来验证您的签名.

> 前文回顾: secp256k1 私钥是一个任意标量 k, 公钥则是生成点 g 与 k 的乘积, 即 g * k.

签名步骤如下:

1. 使用哈希函数(例如 sha256)对信息进行哈希处理, 得到信息摘要 m.
2. 从 [1, n-1] 范围内选择一个随机整数 k. 其中 n 为椭圆曲线的阶.
3. 计算点 c = g * k 并将结果的 x 坐标记为 r. 如果 r 等于 0, 则为 k 选择不同的值并重复该过程.
4. 计算 $s = k^{-1} * (m + r * prikey)$ 的值, 其中 k^{-1} 是 k 的乘法逆元. 如果 s 等于 0, 则为 k 选择不同的值并重复该过程.
5. 消息的数字签名由 (r, s) 对组成.

实现代码如下:

```
import itertools
import secrets
import typing
import pabtc.secp256k1


def sign(prikey: pabtc.secp256k1.Fr, m: pabtc.secp256k1.Fr) -> typing.Tuple[pabtc.secp256k1.Fr, pabtc.secp256k1.Fr, int]:
    # https://www.secg.org/sec1-v2.pdf
    # 4.1.3 Signing Operation
    for _ in itertools.repeat(0):
        k = pabtc.secp256k1.Fr(max(1, secrets.randbelow(pabtc.secp256k1.N)))
        R = pabtc.secp256k1.G * k
        r = pabtc.secp256k1.Fr(R.x.x)
        if r.x == 0:
            continue
        s = (m + prikey * r) / k
        if s.x == 0:
            continue
        v = 0
        if R.y.x & 1 == 1:
            v |= 1
        if R.x.x >= pabtc.secp256k1.N:
            v |= 2
        return r, s, v
```

您可能发现在代码实现上, 签名函数不但返回了 (r, s), 还额外返回了一个 v 值. 这是恢复标识符, 用于从签名中确定签名者的公钥. 它使用了两个标志位, 最低标识位标志 c 的 y 轴坐标的奇偶位, 以便我们可以根据签名中的 r 来唯一还原 c 的实际值(椭圆曲线是关于 x 轴对称的

曲线, 每一个 x 都对应两个可能的 y 值). 另一个比特位用于确认 r 值是否发生过溢出, 因为椭圆曲线上的点的坐标范围是 0 到 P, 但在签名运算中我们会将 c 的 x 坐标转换为一个标量, 范围将缩小到 0 到 N, 因此可能会发生溢出截断.

> 您还记得 P 与 N 的取值吗? P 指的是素数域中的素数, N 指 secp256k1 椭圆曲线的阶, N 比 P 小.

```
P = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
N = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
```

2.6.2 Ecdsa 验签

验签是签名的反向运算, 其步骤如下:

1. 使用相同的哈希函数对收到的信息进行哈希处理, 得到信息摘要 m.
2. 检查签名值 r 和 s 是否在 [1, n-1] 范围内. 如果不在, 则签名无效.
3. 计算值 a = m * s^{-1}和 b = r * s^{-1}, 其中 s^{-1} 是 s 的乘法逆元.
4. 计算点 c = g * a + pubkey * b. 如果 c 等于单位元, 则签名无效.
5. 如果 c 的 x 坐标等于 r, 则签名有效, 否则无效.

我们实现代码如下:

```
import pabtc.secp256k1


def verify(pubkey: pabtc.secp256k1.Pt, m: pabtc.secp256k1.Fr, r: pabtc.secp256k1.Fr, s: pabtc.secp256k1.Fr) -> bool:
    # https://www.secg.org/sec1-v2.pdf
    # 4.1.4 Verifying Operation
    a = m / s
    b = r / s
    R = pabtc.secp256k1.G * a + pubkey * b
    assert R != pabtc.secp256k1.I
    return r == pabtc.secp256k1.Fr(R.x.x)
```

2.6.3 Ecdsa 公钥恢复

给定一个 ecdsa 签名 (r, s) 以及恢复标志符 v, 可以唯一确定签名者的公钥. 步骤如下:

1. 使用相同的哈希函数对收到的信息进行哈希处理, 得到信息摘要 m.
2. 通过恢复标志符 v 唯一还原 c 值.
3. 公钥 pubkey = (c * s - g * m) / r.

我们实现代码如下:

```
import pabtc.secp256k1


def pubkey(m: pabtc.secp256k1.Fr, r: pabtc.secp256k1.Fr, s: pabtc.secp256k1.Fr, v: int) -> pabtc.secp256k1.Pt:
    # https://www.secg.org/sec1-v2.pdf
    # 4.1.6 Public Key Recovery Operation
    assert v in [0, 1, 2, 3]
    if v & 2 == 0:
        x = pabtc.secp256k1.Fq(r.x)
    else:
        x = pabtc.secp256k1.Fq(r.x + pabtc.secp256k1.N)
    z = x * x * x + pabtc.secp256k1.A * x + pabtc.secp256k1.B
    y = z ** ((pabtc.secp256k1.P + 1) // 4)
    if v & 1 != y.x & 1:
```

```
        y = -y
    R = pabtc.secp256k1.Pt(x, y)
    return (R * s - pabtc.secp256k1.G * m) / r
```

再次提醒! 上文中所有出现的代码都公布在 github 上, 这样您可以随时查看, 参考和使用. 如果您有任何问题或需要进一步的帮助, 请随时告诉我!

- pabtc.ecdsa
- pabtc.secp256k1

2.6.4 习题

例: 有一条消息经过哈希处理为, 哈希值为 0x72a963cdfb01bc37cd283106875ff1f07f02bc9ad6121b75c3d17629df128d4e, 请使用私钥 0x01 对其进行签名, 验签以及公钥恢复.

答:

```
import pabtc

prikey = pabtc.secp256k1.Fr(1)
pubkey = pabtc.secp256k1.G * prikey
m = pabtc.secp256k1.Fr(0x72a963cdfb01bc37cd283106875ff1f07f02bc9ad6121b75c3d17629df128d4e)

r, s, v = pabtc.ecdsa.sign(prikey, m)
assert pabtc.ecdsa.verify(pubkey, m, r, s)
assert pabtc.ecdsa.pubkey(m, r, s, v) == pubkey
```

2.7 Solana/私钥, 公钥与地址/私钥的密码学解释(五)

Secp256k1 与 ecdsa 签名算法现在已经被广泛用于许多系统中了. 我知道椭圆曲线在密码学中有优势, 因为它们可以在较小的密钥长度下提供更高的安全性. 但是, 它们真的是安全且完美的算法吗?

从最近的新闻来看, 它们似乎有了一些问题. 美国国家标准与技术研究院认为 secp256k1 存在一些安全风险, 已经不建议使用, 作为替代, 美国建议使用另一条名为 secp256r1 的椭圆曲线. 另一方面, 比特币自身也在改变, 比特币在 2021 年引入了一种叫做 schnorr 的签名算法来尝试替代 ecdsa.

促成这些改变的本质原因是因为 ecdsa 签名算法自身的问题. 它极其容易受到攻击, 并造成过许多灾难性的后果. 在本节内容中, 我将带领大家回顾历史, 并尝试重现这些历史上的著名攻击.

2.7.1 随机数重用攻击

因为比特币的原因, secp256k1 椭圆曲线以及 ecdsa 签名算法变得无人不知, 无人不晓. 但其实在比特币之前, 它们也并非无人问津. 例如在 playstation 3 时代, 索尼就使用存储在公司总部的私钥将其 playstation 固件标记为有效且未经修改. Playstation 3 只需要一个公钥来验证签名是否来自索尼. 但不幸的是, 索尼因为他们糟糕的代码实现而遭到了黑客的破解, 这意味着他们今后发布的任何系统更新都可以毫不费力地解密.

在 fail0overflow 大会上, 黑客展示了索尼 ecdsa 的部分代码, 发现索尼让随机数的值始终保持 4, 这导致了 ecdsa 签名步骤中的随机私钥 k 始终会得到相同的值. Ecdsa 签名要求随机数 k 是严格随机的, 如果重复使用 k, 将直接导致私钥泄露. 这个攻击并不难, 因此小伙子们快来挑战一下吧!

```
def get_random_number():
  # Chosen by fair dice roll. Guaranteed to be random.
  return 4
```

例: 有以下信息, 求私钥 prikey.

- 信息 m_1, 及其签名 (r_1, s_1).
- 信息 m_2, 及其签名 (r_2, s_2).
- 信息 m_1 和 m_2 使用相同的随机数 k 进行签名, k 的具体数据则未知.

答:

$$
\begin{aligned}
s_1 &= (m_1 + prikey * r_1) / k \\
s_2 &= (m_2 + prikey * r_2) / k = (m_2 + prikey * r_1) / k \\
s_1 / s_2 &= (m_1 + prikey * r_1) / (m_2 + prikey * r_1) \\
prikey &= (s_1 * m_2 - s_2 * m_1) / (s_2 - s_1) / r_1
\end{aligned}
$$

这里有一个实际的例子可以帮助大家更直观的理解如何通过两个使用相同随机数 k 的签名来还原私钥.

```
import pabtc

m1 = pabtc.secp256k1.Fr(0x72a963cdfb01bc37cd283106875ff1f07f02bc9ad6121b75c3d17629df128d4e)
r1 = pabtc.secp256k1.Fr(0x741a1cc1db8aa02cff2e695905ed866e4e1f1e19b10e2b448bf01d4ef3cbd8ed)
s1 = pabtc.secp256k1.Fr(0x2222017d7d4b9886a19fe8da9234032e5e8dc5b5b1f27517b03ac8e1dd573c78)

m2 = pabtc.secp256k1.Fr(0x059aa1e67abe518ea1e09587f828264119e3cdae0b8fcaedb542d8c287c3d420)
r2 = pabtc.secp256k1.Fr(0x741a1cc1db8aa02cff2e695905ed866e4e1f1e19b10e2b448bf01d4ef3cbd8ed)
s2 = pabtc.secp256k1.Fr(0x5c907cdd9ac36fdaf4af60e2ccfb1469c7281c30eb219eca3eddf1f0ad804655)

prikey = (s1 * m2 - s2 * m1) / (s2 - s1) / r1
assert prikey.x == 0x5f6717883bef25f45a129c11fcac1567d74bda5a9ad4cbffc8203c0da2a1473c
```

2.7.2 心脏点攻击

心脏点攻击 (invalid curve attacks) 指攻击者通过生成不在标准曲线上的点, 通过这种方式绕过签名验证, 密钥生成, 或者其他基于曲线的操作.

在签名过程中, 攻击者可以通过某种方式构造一个无效的公钥. 该无效公钥与攻击者的私钥之间存在某种数学关系(例如, 攻击者通过伪造一个无效的公钥进行签名), 这使得攻击者能够生成一个看似有效的签名. 正常情况下, 签名验证算法会检查公钥是否在 secp256k1 曲线范围内. 如果公钥无效, 系统应该拒绝该签名. 但是, 假设系统没有进行充分的曲线点有效性检查, 攻击者可能会提交一个包含无效公钥和伪造签名的请求. 在某些情况下, 系统可能会错误地接受这个无效签名, 认为它是合法的. 攻击者的签名可能会通过系统的检查, 导致恶意的交易或操作被错误地认为是有效的, 从而执行某些非法操作, 比如转移资金或修改数据.

一个现实中的例子是 openssl 中的椭圆曲线验证漏洞. 2015 年, openssl 的一个版本(1.0.2 之前的版本)存在一个椭圆曲线验证漏洞. 攻击者可以通过构造一个无效的椭圆曲线点并将其用作公钥, 利用 openssl 的某些漏洞绕过验证, 进而攻击使用该库的系统. 这个漏洞被称为 cve-2015-1786, 它允许攻击者通过伪造无效的公钥来绕过签名验证. 同样的问题也曾发生在 bitocin core 使用的 ecdsa 库中, 早期版本的库没有对椭圆曲线点进行足够的检查.

在这个漏洞被修复之前, 攻击者可以在不进行正确验证的情况下, 绕过系统对曲线有效性的检查, 从而导致可能的拒绝服务或其他安全问题.

2.7.3 交易延展性攻击

Mt.Gox(门头沟) 一度是世界上最大的比特币交易所. 该公司总部位于东京, 估计 2013 年占比特币交易量的 70%. 2014 年, 门头沟交易所被黑客攻击, 造成了约 85 万枚比特币的损失. 在门头沟事件中, 黑客所采用的是一种名为交易延展性攻击(transaction malleability attack)的手法.

此次攻击的具体过程如下: 攻击者首先在门头沟发起一笔提现交易 a, 接着在交易 a 被确认之前通过篡改交易签名, 使得标识一笔交易唯一性的交易哈希发生改变, 生成伪造的交易 b. 之后, 交易 b 被区块链确认, 而交易所则收到了交易 a 失败的信息. 交易所误认为提现失败从而重新为攻击者构造一笔新的提现交易.

要使得攻击成立, 其核心是攻击者能够修改交易的签名部分(如输入的签名)或者其他非关键的字段, 从而改变交易的哈希值, 但不会改变交易的实际内容.

巧合的是, secp256k1 + ecdsa 确实存在一种十分便捷的方式, 使得攻击者可以修改签名结果的同时仍然能通过签名验证. 如果我们分析 ecdsa 验签算法, 会发现验签结果和签名 (r, s) 中的 s 值的符号是无关的. 为了验证这一点, 我们编写如下测试代码:

```
import pabtc

prikey = pabtc.secp256k1.Fr(1)
pubkey = pabtc.secp256k1.G * prikey
msg = pabtc.secp256k1.Fr(0x72a963cdfb01bc37cd283106875ff1f07f02bc9ad6121b75c3d17629df128d4e)

r, s, _ = pabtc.ecdsa.sign(prikey, msg)
assert pabtc.ecdsa.verify(pubkey, msg, r, +s)
assert pabtc.ecdsa.verify(pubkey, msg, r, -s)
```

在上述代码中, 我们使用私钥对一条消息进行了签名, 然后对签名中的 s 值取负号, 发现修改后的签名依然能通过 ecdsa 验证.

比特币在早期版本中存在这种攻击的风险, 攻击者通过延展性攻击破坏了交易的**不可篡改性**, 导致了严重的安全问题. 为了解决这一问题, 比特币在 segregated witness(segwit) 升级中做了改进, segwit 将交易的签名部分与其他数据分开存储, 使得即使攻击者篡改签名部分, 交易哈希也不再受影响, 从而解决了交易延展性问题.

这个问题在其他区块链系统中也有类似的影响, 因此许多项目都采取了类似 segwit 的解决方案, 来确保交易的完整性和可追溯性. 另一种解决方案是以太坊所采取的, 以太坊对签名中的 s 做了额外的要求, 要求 s 必须小于 `pabtc.secp256k1.N / 2`, 您可以 https://ethereum.github.io/yellowpaper/paper.pdf, Appendix F. Signing Transactions 找到以太坊针对交易延展性攻击的详细解决方式.

> 在古代, 如果我们把一枚金币敲变形之后, 虽然形状有所改变, 但质量却没有发生变化, 在市场交易中它仍然会被认可为一枚金币, 甚至您将金币敲成金块, 它依然会被认可, 这种特性呢被称为"延展性"或"可锻性".

有诗云:

门头交易所, 用户真是多.

比特币被盗, 大伙冷汗冒,

黑客改哈希, 交易无踪兆,

冷钱包空空, 财富随风飘.

2.7.4 旁路攻击

我坐飞机旁边有个大哥一直在看股票, 我俩聊了几句股票. 他说今年行情不好, 让我猜他亏了多少钱,

我说: "也就十来万吧." 大哥一愣, 问我: "你咋猜的呢?"

我说虽然你穿着衬衫西裤, 看着很商务, 但是却背了个瑞士军刀牌双肩包, 大老板有背这个的么?

一看你就是个跑业务的. 再看你戴了块阿玛尼这种杂牌子手表, 三十多岁的人了, 连个劳力士都没混上, 说明收入很一般.

你的衬衣是旧的, 但是熨得很板正, 领子也干净, 这都是你老婆给你收拾的. 你包上有个 hellokitty 小挂件, 这应该是你女儿给你挂的.

你自选股里都是一些 5G 移动芯片之类的股票, 你觉得自己很懂, 你应该是互联网企业上班的. 方方面面综合下来, 你的可支配资金也就 20-30 万, 结合今年的行情, 亏损 10 万左右. 再看看你这个黑眼圈和与年龄不成比例的稀疏发型, 压力不小.

你老婆应该还不知道你股票亏了这么多钱. 刚才看到你手机界面上还有炒虚拟币的软件, 在最后一位, 说明是最近刚刚下载的.

如果你股票再亏, 你就打算去炒虚拟币放手一搏, 但是你只会亏得更惨. 说完我点了下他手机炒股软件界面, 上面显示总投入 28 万, 当前亏损 10.2 万.

大哥沉默了, 一路上再也没跟我说一句话, 只是偶尔低头用食指关节揉一揉微微发红的的眼眶, 飞机餐的盒饭打开了, 但是没吃.

上述故事来自中国互联网, 最早出现在 2015 年, 由于被转载太多次, 因此作者实在不明. 在这个故事里, "我"就对"大哥"发动了一次**旁路攻击**. 大哥虽然没有向我透露任何关于自身的投资信息, 但是由于大哥的资产收益会影响大哥的穿着, 因此我们可以通过大哥的穿着来反向推断大哥的资产收益.

在密码学中, 所谓的旁路攻击(side-channel attacks), 就是一种利用设备执行任务时产生的物理或行为信息(如执行时间, 用电模式, 电磁辐射等)来破解密码或签名方案的方法. 对于 secp256k1 椭圆曲线和 ecdsa 签名方案, 这种攻击可能通过分析关键运算的执行特性来推断私钥.

在 Ecdsa 中, 签名过程涉及生成一个随机数 k, 然后用它来计算签名的一部分. 这个随机数的安全性至关重要, 如果 k 被泄漏, 攻击者就能通过它恢复私钥.

例: 有以下信息, 请计算 secp256k1 的私钥.

- 消息 `m = 0x72a963cdfb01bc37cd283106875ff1f07f02bc9ad6121b75c3d17629df128d4e`
- 随机数字 `k = 0x1058387903e128125f2715d7de954f53686172b78c3f919521ae4664f30b00ca`
- 签名
- `r = 0x75ee776c554b1dd5e1680a4cc9a3d0e8cb11400742d8af0222ce383e642f98db`
- `s = 0x35fd48c9157256558184e20c9392ff3c9517f9753e3745aede06cab285f4bc0d`

答: 根据 ecdsa 签名算法, 容易得到私钥计算公式为 `prikey = (s * k - m) / r`, 代入数字计算, 得到私钥为 1. 验证代码如下:

```
import pabtc

m = pabtc.secp256k1.Fr(0x72a963cdfb01bc37cd283106875ff1f07f02bc9ad6121b75c3d17629df128d4e)
k = pabtc.secp256k1.Fr(0x1058387903e128125f2715d7de954f53686172b78c3f919521ae4664f30b00ca)
r = pabtc.secp256k1.Fr(0x75ee776c554b1dd5e1680a4cc9a3d0e8cb11400742d8af0222ce383e642f98db)
s = pabtc.secp256k1.Fr(0x35fd48c9157256558184e20c9392ff3c9517f9753e3745aede06cab285f4bc0d)

prikey = (s * k - m) / r
assert prikey == pabtc.secp256k1.Fr(1)
```

随机数字 k 的计算涉及到椭圆曲线点乘和逆元操作(通常通过扩展欧几里得算法实现). 这些操作的时间可能会与 k 相关, 旁路攻击者可以测量执行时间差异来提取 k. 为了揭示原理, 我将尝试把攻击过程简化.

例: 有未知随机数字 k, 现在黑客通过某种手段可探测出 g * k 的执行时间, 请尝试是否可以得到随机数字 k 的一些信息.

答: 观察椭圆曲线上的点的乘法算法, 得出当 k 的比特位不同时, 会执行不同的操作. 当比特位为 0 时, 其计算量小于比特位为 1 时. 我们事先取两个不同的 k 值, 一个大多数位为 0, 另一个大多数位位 1, 计算它们的执行时间之差. 当有新的未知 k 进行计算时, 探测得到它的执行时间, 与前两个值进行比对, 可大致得到未知 k 其比特位为 1 的数量. 实验代码如下. 注意, 为了简化攻击步骤, 在实验代码中我们假设所有参与计算的 k 的第一个比特位始终为 1.

```
import pabtc
import secrets
import timeit

k_one = pabtc.secp256k1.Fr(0x8000000000000000000000000000000000000000000000000000000000000000)  # Has one '1' bits
k_255 = pabtc.secp256k1.Fr(0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff)  # Has 255 '1' bits
k_unknown = pabtc.secp256k1.Fr(max(1, secrets.randbelow(pabtc.secp256k1.N)) | k_one.x)  # The unknown k

a = timeit.timeit(lambda: pabtc.secp256k1.G * k_one, number=1024)
b = timeit.timeit(lambda: pabtc.secp256k1.G * k_255, number=1024)
c = timeit.timeit(lambda: pabtc.secp256k1.G * k_unknown, number=1024)

d = (c - a) / ((b - a) / 254)
print(d)
```

上述攻击过程是旁路攻击中的时间攻击(timing attacks), 如果要对该攻击做防护, 可以通过在代码中引入常量时间操作(constant-time operations)来避免泄露信息. 例如, 使用固定时间的加法和乘法, 防止时间差异被利用. 那么作为一道课后习题, 希望同学们可以自行尝试修改椭圆曲线上的乘法算法实现, 来避免老师上面提到的时间攻击.

在实际应用中, 为了避免密码学算法中的旁路攻击, 需要在算法, 硬件和软件层面做出多方面的安全优化. 不过由于 secp256k1 与 ecdsa 方案在设计时未充分考虑该攻击方式, 因此防护此类攻击非常困难且复杂.

2.7.5 小结

总之, 虽然 secp256k1 和 ecdsa 在许多应用中广泛使用, 并且它们在合理的实现和正确的使用下是相当安全的, 但也不能忽视它们存在的一些潜在漏洞. 得益于比特币的发展, secp256k1 和 ecdsa 名声大噪的同时也吸引了更多的密码学家和不怀好意的黑客们. 未来, 更多关于 sepc256k1 的一些攻击方式可能会逐步被发现并利用. 因此, 保持警惕, 及时更新以及遵循最新的安全最佳实践对于确保系统安全至关重要. 随着加密领域的不断进步, 目前已经有一些更加安全且高效的替代方案出现, 但无论如何, 理解并应对当前的风险, 仍然是我们每一个开发者的责任和挑战.

2.8 Solana/私钥, 公钥与地址/私钥的密码学解释(六)

Ed25519 是近些年来最火热的 secp256k1 替换品. 它的发展离不开一个意外. 25519 系列曲线自 2005 年发表以来, 在工业界原本一直处于无人问津的存在, 但转折发生在 2013 年, 斯诺登曝光棱镜计划后, 人们发现美国安全局力推的 p-256 曲线可能存在算法后门. 在此之后, 工业界开始尝试使用 25519 系列曲线来代替 p-256 系列曲线.

> Secp256k1 曲线属于类 p-256 曲线, 相较原版只有参数有少量修改.

目前来看, Ed25519 曲线的推广和应用无疑是十分成功的. 传统 web2 产品, 例如 github 和 binance, 都支持 ed25519 作为其 api 权限验证算法. 新兴的 web3 产品, 例如 solana, 也采用 ed25519 作为其核心签名算法.

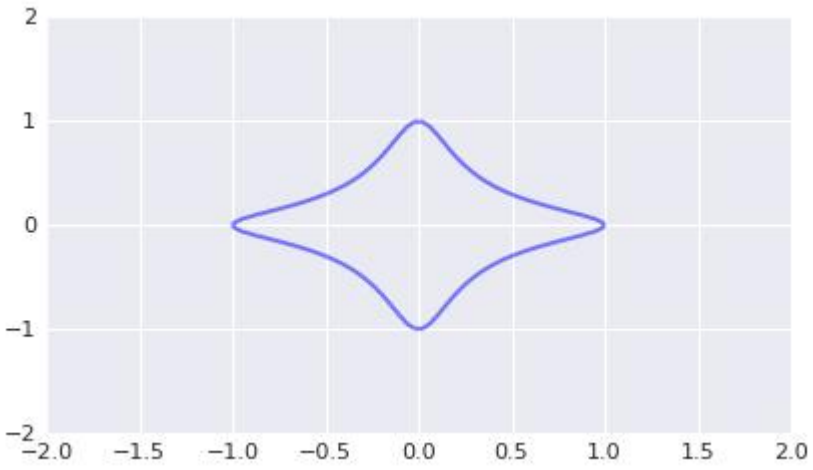
2.8.1 Ed25519 曲线

Ed25519 是一类**扭曲爱德华兹曲线**($ax^2 + y^2 = 1 + d * x^2 * y^2$), 其表达式为

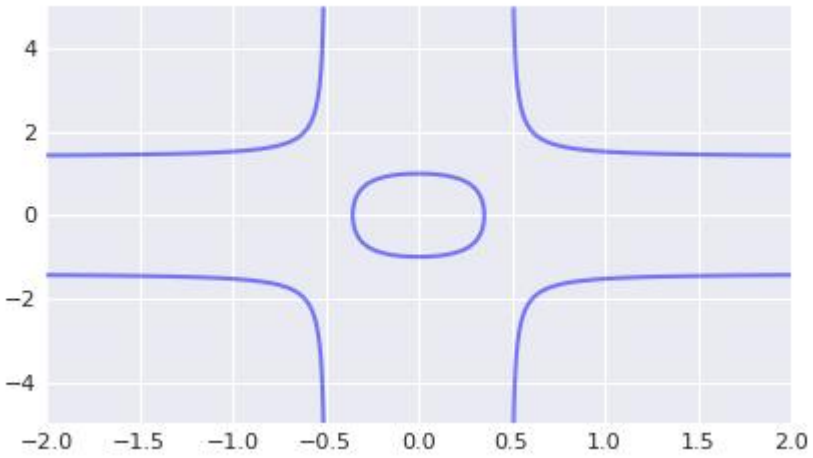
$$-x^2 + y^2 = 1 - (121665 / 121666) * x^2 * y^2$$

它和 secp256k1 一样基于素数域, 但其采用的素数 p 为 $2^{255} - 19$. 如果将该素数以 16 进制表示, 会发现其以 `ed` 结尾, 同时该椭圆曲线的阶的 16 进制表示同样以 `ed` 结尾. 因此 ed25519 既包含了作者爱德华兹的名字, 也隐含了该曲线的素数域和阶, 不得不说作者真是个取名小天才. 我等若是有作者一半的资质, 就不必为取变量名而烦恼了.

所谓扭曲爱德华兹曲线, 是指在**爱德华兹曲线**($x^2 + y^2 = 1 + d * x^2 * y^2$)上增加了常数项 a, 会因此"扭曲"了爱德华兹曲线. 原版的爱德华兹曲线是个非常漂亮的二元二次曲线, 例如当 d = -30 时, 爱德华兹曲线图形如下所示.



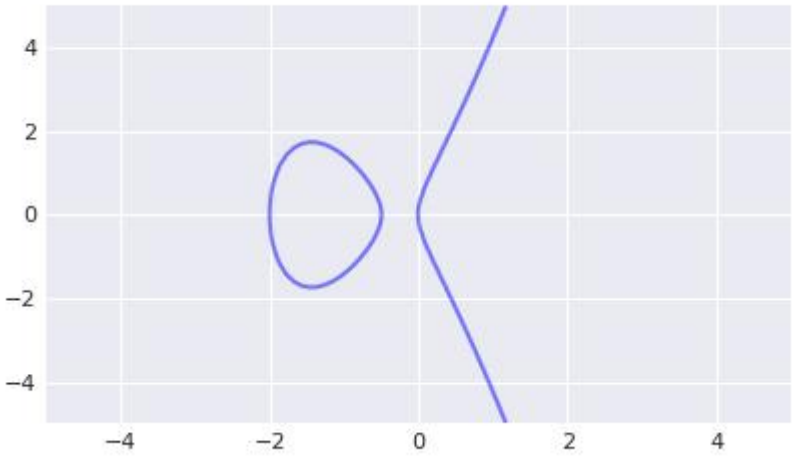
由于 ed25519 曲线的图形十分的不直观, 因此我们以 a = 8, d = 4 时的扭曲爱德华兹曲线做为替代图例, 其图形如下所示.



爱德华兹曲线是一种另类的椭圆曲线. 它在形式上简化了椭圆曲线上的点的加法运算, 使得实现更容易且计算效率更高.

所有扭曲爱德华兹曲线都与蒙哥马利曲线(b * y^2 = x^3 + a * x^2 + x)双向有理等价, ed25519 对应的蒙哥马利曲线称作 curve25519, 其表达式如下. 其图像同样非常不直观, 因此此处我们给出 a = 2.5, b=0.25 的替代图像.

```
y² = x³ + 486662 * x² + x
```



对于这些不同的椭圆曲线类型, 您可以这么理解: 椭圆曲线的一般形式是 y^2 = x^3 + ax + b, 在 1985 年由科布利茨和米勒分别独立提出. 在 1987 年, 蒙哥馬利证明了蒙哥马利曲线与椭圆曲线的一般形式双向有理等价, 因此蒙哥马利曲线也被称作椭圆曲线的蒙哥马利表示. 之后在 2005 年, 爱德华兹证明了扭曲爱德华兹曲线与蒙哥馬利曲线双向有理等价, 因此扭曲爱德华兹曲线也被称作椭圆曲线的扭曲爱德华兹表示.

例: 请判断下面的点是否位于 ed25519 上.

- `x = 0x1122e705f69819df8042c3a34d5294668f25830f41e9b585b2aa6b05ef4cc7e2`
- `y = 0x2a619802432fe95214ac6fed9d01dd149d197f1202e8c2698caab03831b8f2ee`

答: ed25519 与 secp256k1 的代码实现非常类似, 您可以在 pxsol.ed25519 中找到它的源码. 或者使用 `pip install pxsol` 安装完整的 solana 包.

```
import pxsol

x = pxsol.ed25519.Fq(0x1122e705f69819df8042c3a34d5294668f25830f41e9b585b2aa6b05ef4cc7e2)
y = pxsol.ed25519.Fq(0x2a619802432fe95214ac6fed9d01dd149d197f1202e8c2698caab03831b8f2ee)

assert pxsol.ed25519.A * x * x + y * y == pxsol.ed25519.Fq(1) + pxsol.ed25519.D * x * x * y * y
```

2.8.2 Ed25519 点的加法

与 secp256k1 曲线类似, Ed25519 上的点可以构成一个加法群. 规定 ed25519 上给定两个不同的点 p 和 q, 其加法 r = p + q, 规则如下:

```
x₃ = (x₁ * y₂ + x₂ * y₁) / (1 + d * x₁ * x₂ * y₁ * y₂)
y₃ = (y₁ * y₂ - a * x₁ * x₂) / (1 - d * x₁ * x₂ * y₁ * y₂)
```

代码实现如下:

```
A = -Fq(1)
D = -Fq(121665) / Fq(121666)

class Pt:

    def __init__(self, x: Fq, y: Fq) -> None:
        assert A * x * x + y * y == Fq(1) + D * x * x * y * y
        self.x = x
        self.y = y

    def __add__(self, data: typing.Self) -> typing.Self:
        # https://datatracker.ietf.org/doc/html/rfc8032#ref-CURVE25519
        # Points on the curve form a group under addition, (x3, y3) = (x1, y1) + (x2, y2), with the formulas
        #           x1 * y2 + x2 * y1               y1 * y2 - a * x1 * x2
        # x3 = -------------------------,   y3 = -------------------------
        #       1 + d * x1 * x2 * y1 * y2          1 - d * x1 * x2 * y1 * y2
        x1, x2 = self.x, data.x
        y1, y2 = self.y, data.y
        x3 = (x1 * y2 + x2 * y1) / (Fq(1) + D * x1 * x2 * y1 * y2)
        y3 = (y1 * y2 - A * x1 * x2) / (Fq(1) - D * x1 * x2 * y1 * y2)
        return Pt(x3, y3)
```

我们对比 secp256k1 的加法, 会发现 ed25519 上的加法算法是大幅简化的: 我们不需要额外的逻辑代码来判断 p 是否等于 ±q. 对于计算机而言, 每增加一个分支判断都会大幅度拖累 cpu 的运算速度, 因此 ed25519 曲线上的加法算法相比 secp256k1 是非常高效的.

2.8.3 Ed25519 点的标量乘法

在拥有加法后我们就能实现标量乘法, 这里不再赘述.

2.8.4 Ed25519 生成元

我们人为规定一个特殊点, 叫做生成点 g, 椭圆曲线上的任意点都可以表示为 g 与一个标量 k 的乘积.

```
G = Pt(
    Fq(0x216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a),
    Fq(0x6666666666666666666666666666666666666666666666666666666666666658),
)
```

例: 计算 g * 42 的值.

答:

```
import pxsol

p = pxsol.ed25519.G * pxsol.ed25519.Fr(42)

assert p.x == pxsol.ed25519.Fq(0x5dbe6cc3ccfe19f056f503fd5895e4ca00385a5f109126914b52446017318069)
assert p.y == pxsol.ed25519.Fq(0x4237066783c4352092fdf0de4df92cae7343f40939f32b3e195c834e99321ace)
```

2.9 Solana/私钥, 公钥与地址/私钥的密码学解释(七)

Eddsa 是一种使用基于扭曲爱德华兹曲线的 schnorr 签名变体的数字签名方案. 它旨在在不牺牲安全性的情况下比现有的数字签名方案更快.

2.9.1 Eddsa 点的编码

在 eddsa 中, 我们需要使用到一种特殊的曲线上的点的编码算法. 直观上讲, 曲线上的一个点由 x 和 y 两个值组成, 且 x 和 y 都 在 0 <= x,y < p 范围内, 因此我们需要使用 64 个字节来表示它. 但有办法将空间压缩到 32 个字节, 具体方法如下:

1. 由于 y < p, 因此 y 的最高有效位始终为 0.
2. 将 x 的最低有效位复制到 y 的最高有效位上, 并将结果以小端序编码为 32 个字节.

在这种编码方式下, 我们能得知 y 的具体数值以及 x 的奇偶性. 代码实现如下.

```
def pt_encode(pt: pxsol.ed25519.Pt) -> bytearray:
    # A curve point (x,y), with coordinates in the range 0 <= x,y < p, is coded as follows. First, encode the
    # y-coordinate as a little-endian string of 32 octets. The most significant bit of the final octet is always zero.
    # To form the encoding of the point, copy the least significant bit of the x-coordinate to the most significant bit
    # of the final octet.
    # See https://datatracker.ietf.org/doc/html/rfc8032#section-5.1.2
    n = pt.y.x | ((pt.x.x & 1) << 255)
    return bytearray(n.to_bytes(32, 'little'))
```

2.9.2 Eddsa 点的解码

Eddsa 点的解码是点的编码的逆运算. 步骤如下:

1. 首先, 将 32 字节数组解释为小端表示的整数. 此数字的第 255 位是 x 坐标的最低有效位, 表示了 x 值的奇偶性. 只需清除此位即可恢复 y 坐标. 如果 y >= p, 则解码失败.
2. 要恢复 x 坐标, 意味着曲线方程 $x^2 = (y^2 - 1) / (d * y^2 + 1)$ 成立. 令 $u = y^2 - 1$, $v = d * y^2 + 1$, 计算它的候选根 $x = (u / v)^((p + 3) / 8)$.
3. 现在有三种情况:
    1. 如果 x * x == u / v, 不做处理.
    2. 如果 x * x == u / v * -1, 则令 x = x * 2^((p - 1) / 4).
    3. 解码失败, 点不在曲线上.
4. 最后, 确定 x 的奇偶性. 如果奇偶性不一致, 则令 x = -x.

代码实现如下

```
def pt_decode(pt: bytearray) -> pxsol.ed25519.Pt:
    # Decoding a point, given as a 32-octet string, is a little more complicated.
    # See https://datatracker.ietf.org/doc/html/rfc8032#section-5.1.3
    #
    # First, interpret the string as an integer in little-endian representation. Bit 255 of this number is the least
    # significant bit of the x-coordinate and denote this value x_0. The y-coordinate is recovered simply by clearing
    # this bit. If the resulting value is >= p, decoding fails.
    uint = int.from_bytes(pt, 'little')
    bit0 = uint >> 255
    yint = uint & ((1 << 255) - 1)
    assert yint < pxsol.ed25519.P
    # To recover the x-coordinate, the curve equation implies x^2 = (y^2 - 1) / (d y^2 + 1) (mod p). The denominator is
    # always non-zero mod p.
    y = pxsol.ed25519.Fq(yint)
    u = y * y - pxsol.ed25519.Fq(1)
    v = pxsol.ed25519.D * y * y + pxsol.ed25519.Fq(1)
    w = u / v
```

```
    # To compute the square root of (u/v), the first step is to compute the candidate root x = (u/v)^((p+3)/8).
    x = w ** ((pxsol.ed25519.P + 3) // 8)
    # Again, there are three cases:
    # 1. If v x^2 = +u (mod p), x is a square root.
    # 2. If v x^2 = -u (mod p), set x <-- x * 2^((p-1)/4), which is a square root.
    # 3. Otherwise, no square root exists for modulo p, and decoding fails.
    if x*x != w:
        x = x * pxsol.ed25519.Fq(2) ** ((pxsol.ed25519.P - 1) // 4)
        assert x*x == w
    # Finally, use the x_0 bit to select the right square root. If x = 0, and x_0 = 1, decoding fails. Otherwise, if
    # x_0 != x mod 2, set x <-- p - x.  Return the decoded point (x,y).
    assert x != pxsol.ed25519.Fq(0) or not bit0
    if x.x & 1 != bit0:
        x = -x
    return pxsol.ed25519.Pt(x, y)
```

2.9.3 私钥

如前所述, Ed25519 的私钥是一个 32 字节的随机数, 通常通过安全的随机数生成器产生. 私钥是用户身份的核心, 需严格保密. 在 Ed25519 的实现中, 私钥并非用于直接签名, 而是通过哈希函数(sha-512)扩展为 64 字节的种子, 其中一部分作为标量用于生成公钥, 另一部分作为签名时的秘密标量. 这种设计增强了私钥的安全性, 防止因直接使用原始私钥而暴露风险.

私钥生成简单, 但其安全性依赖于随机数的质量. 如果随机数可预测, 攻击者可能通过暴力破解或伪造签名来威胁系统. 因此, 使用密码学安全的随机数生成器(如 /dev/urandom 或硬件随机数生成器)是生成私钥的关键.

```
import secrets

prikey = bytearray(secrets.token_bytes(32))
```

2.9.4 公钥

Ed25519 的公钥的长度同样是 32 字节. 32 字节公钥通过以下步骤生成.

1. 使用 sha-512 对 32 字节私钥进行哈希处理, 生成 64 个字节的哈希结果. 只有前 32 个字节用于生成公钥.
2. 清除第一个字节的最低三位, 清除最后一个八位字节的最高位, 并设置最后一个字节的第二高位.
3. 将上述数据解释为小端序整数, 形成秘密标量 a. 执行标量乘法 g * a, 并将结果进行点的编码.

```
def pubkey(prikey: bytearray) -> bytearray:
    assert len(prikey) == 32
    h = hash(prikey)
    a = int.from_bytes(h[:32], 'little')
    a &= (1 << 254) - 8
    a |= (1 << 254)
    a = pxsol.ed25519.Fr(a)
    return pt_encode(pxsol.ed25519.G * a)
```

Ed25519 的公钥生成过程是单向的: 从私钥可以快速计算出公钥, 但从公钥无法反推出私钥, 这种不可逆性是椭圆曲线离散对数问题的核心保障. 公钥的作用是公开身份, 任何人都可以使用公钥来验证签名. 由于 Ed25519 的设计高效, 公钥生成和使用都非常快速, 非常适用于高性能场景.

2.9.5 签名

签名用于证明消息的真实性和完整性. 签名过程的输入是私钥(一个 32 字节的数组)和任意大小的消息 m, 签名过程如下:

1. 使用 sha-512 对私钥(32 字节)进行哈希计算. 按照前一节的描述, 从哈希的前半部分构造秘密标量 a, 以及对应的公钥 pubkey. 将哈希摘要的后半部分记为 prefix.
2. 计算 sha-512(prefix + m), 其中 m 是待签名的消息. 将得到的 64 字节哈希解释为一个小端序整数 r.
3. 计算点 g * r, 并对结果进行点的编码, 记为 digest.
4. 计算 sha-512(digest + pubkey + m), 并将得到的 64 字节摘要解释为一个小端序整数 h.
5. 计算 s = r + a * h.
6. 将 digest 和 s 的小端序编码连接起来, 形成 64 字节签名.

```
def sign(prikey: bytearray, m: bytearray) -> bytearray:
    # The inputs to the signing procedure is the private key, a 32-octet string, and a message M of arbitrary size.
    # See https://datatracker.ietf.org/doc/html/rfc8032#section-5.1.6
    assert len(prikey) == 32
    h = hash(prikey)
    a = int.from_bytes(h[:32], 'little')
    a &= (1 << 254) - 8
    a |= (1 << 254)
    a = pxsol.ed25519.Fr(a)
    A = pxsol.ed25519.G * a
    pubkey = pt_encode(A)
    prefix = h[32:]
    r = pxsol.ed25519.Fr(int.from_bytes(hash(prefix + m), 'little'))
    R = pxsol.ed25519.G * r
    digest = pt_encode(R)
    h = pxsol.ed25519.Fr(int.from_bytes(hash(digest + pubkey + m), 'little'))
    s = r + a * h
    return digest + bytearray(s.x.to_bytes(32, 'little'))
```

2.9.6 验签

验签是验证签名的过程, 用于确认消息未被篡改且确实由持有对应私钥的人签署. Ed25519 的验签需要消息 m, 签名 v 和公钥 pubkey, 步骤如下:

1. 先将签名 v 拆分为两个 32 字节数组. 将前半部分记为 digest, 解码为点 r, 将后半部分解码为整数 s. 将公钥 pubkey 解码为点 a. 如果任何解码失败(包括 s 超出范围), 则签名无效.
2. 计算 sha-512(digest + pubkey + m), 并将 64 位字节摘要解释为小端整数 h.
3. 检查是否满足群方程 g * s == r + a * h

```
def verify(pubkey: bytearray, m: bytearray, g: bytearray) -> bool:
    # Verify a signature on a message using public key.
    # See https://datatracker.ietf.org/doc/html/rfc8032#section-5.1.7
    assert len(pubkey) == 32
    assert len(g) == 64
    A = pt_decode(pubkey)
    digest = g[:32]
    R = pt_decode(digest)
    s = pxsol.ed25519.Fr(int.from_bytes(g[32:], 'little'))
    h = pxsol.ed25519.Fr(int.from_bytes(hash(digest + pubkey + m), 'little'))
    return pxsol.ed25519.G * s == R + A * h
```

例: 假设您有私钥 `833fe62409237b9d62ec77587520911e9a759cec1d19755b7da901b96dca3d42`, 请对消息 `sha-512('abc')` 进行签名并验证签名.

答:

```
import pxsol
import hashlib

prikey = bytearray.fromhex('833fe62409237b9d62ec77587520911e9a759cec1d19755b7da901b96dca3d42')
pubkey = pxsol.eddsa.pubkey(prikey)
msg = hashlib.sha512(b'abc').digest()
sig = pxsol.eddsa.sign(prikey, msg)
assert sig[:32].hex() == 'dc2a4459e7369633a52b1bf277839a00201009a3efbf3ecb69bea2186c26b589'
assert sig[32:].hex() == '09351fc9ac90b3ecfdfbc7c66431e0303dca179c138ac17ad9bef1177331a704'
assert pxsol.eddsa.verify(pubkey, msg, sig)
```

2.10 Solana/私钥, 公钥与地址/私钥的密码学解释(八)

这是一项了不起的成就: 我们花了整整八章的时间来学习 Ed25519. 我们能坚持到这里, 相信未来无论什么艰难困阻都无法阻挡我们了. 我最后想和您谈一下 Ed25519 的优势, 有些优势是显而易见的, 而有些优势是隐藏的.

发明 Ed25519 的最大原因就是为了取代美国国家标准系列椭圆曲线. 爱德华兹本人对其的评价是:

1. 完全开放设计, 算法各参数的选择直截了当, 非常明确, 没有任何可疑之处, 相比之下, 目前广泛使用的椭圆曲线是美国国家标准系列标准, 方程的系数是使用来历不明的随机种子生成, 而至于这个种子的来历没有资料介绍.
2. 安全性高, 一个椭圆曲线加密算法就算在数学上是安全的, 在实用上也并不一定安全, 有很大的概率通过缓存, 时间, 恶意输入摧毁安全性, 而 Ed25519 系列椭圆曲线经过特别设计, 尽可能的将出错的概率降到了最低, 可以说是实践上最安全的加密算法. 例如, 任何一个 32 位随机数都是一个合法的 Ed25519 公钥, 因此通过恶意数值攻击是不可能的, 算法在设计的时候刻意避免了某些分支操作, 这样在编程的时候可以不使用 if , 减少了不同 if 分支代码执行时间不同的时序攻击概率, 相反, 美国国家标准系列椭圆曲线算法在实际应用中出错的可能性非常大, 而且对于某些理论攻击的免疫能力不高.

从开发者的角度来看, 它具有一些额外的隐藏优点.

1. Ed25519 签名过程不依赖随机数生成器. 因此我们可以不必假设必须存在一个安全的随机数发生器. 密码学基本原理: 假设越少, 问题越少!
2. Ed25519 的私钥空间是 0 <= prikey <= 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff, 而 secp256k1 的私钥空间是 0 <= prikey <= 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141. 您可以看到 Ed25519 的私钥空间是非常规整的, 这能避免许多潜在的错误.
3. Ed25519 公钥只有 32 字节, 而 secp256k1 的非压缩表示公钥是 64 字节, 压缩表示公钥是 33 字节. 在系统底层里具有一些隐藏的好处. 例如常用的内存拷贝函数 memcpy, 通常会对长度小于等于 32 的字节数组使用特别的 small copy 算法, 典型例子如 glibc 的 aarch64 代码中 https://github.com/bminor/glibc/blob/master/sysdeps/aarch64/memcpy.S. 在一些高级语言中也会额外照顾长度小于等于 32 的字节数组, 例如 rust 就只会为这类小数组实现 clone 和 copy.
4. Ed25519 曲线上的点群运算是完备的, 也即对于所有的点群中元素都成立, 计算时无需做额外的判断, 意味着运算时不需要对不受信的外部值做昂贵的点的验证.
5. Ed25519 签名机制本身安全性不受哈希碰撞的影响.

2.11 Solana/私钥, 公钥与地址/公钥

在 solana 区块链中, 公钥是一个由加密算法生成的唯一标识符, 通常与私钥成对出现. **私钥由用户持有并严格保密, 而公钥则可以公开分享, 用来表示用户的身份或账户地址**. 与比特币或以太坊的地址不同, solana 的公钥本身就是账户的标识符, 而不是通过哈希函数生成的派生地址.

2.11.1 什么是公钥

Solana 使用的是 ed25519 椭圆曲线加密算法, 这是一种高安全性和高效性的签名算法. 每个公钥是一个 32 字节的字节数组, 通常以 base58 编码格式展示给用户. 例如, 一个典型的 solana 公钥看起来像是:

```
6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
```

例: 有 solana 地址 `6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt`, 求该地址对应的公钥. 公钥结果以 16 进制表示.

答: Solana 的地址就是其公钥. 因此只需将地址字符串以 base58 解码即可. 为了解决这个问题, 我们首先通过 `pip install pxsol` 命令安装 `pxsol` 库. 这是一个对人类友好的 solana python 库. 使用该库提供的 base58 功能即可解码 base58 字符串.

```
import pxsol

pubkey = pxsol.base58.decode('6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt')
print(pubkey.hex()) # 4cb5abf6ad79fbf5abbccafcc269d85cd2651ed4b885b5869f241aedf0a5ba29
```

2.11.2 公钥的生成

Solana 的公钥和私钥对是通过 ed25519 算法生成的. 大致生成过程如下:

1. 私钥生成: 首先生成一个随机的 32 字节数组.
2. 私钥派生: 通过私钥和 ed25519 相关算法, 生成一个 64 字节的派生私钥.
3. 公钥派生: 从派生私钥中通过数学运算派生出对应的 32 字节公钥.

如果您已经完整阅读过前几节, 那么您应该对这其中的每一个步骤都了如指掌. 如果您没有阅读过, 那也不要紧, 因为 pxsol 已经为您实现了一切!

例: 有以 base58 表示的私钥 `11111111111111111111111111111112`, 求其对应的公钥.

答:

```
import pxsol

prikey = pxsol.core.PriKey.base58_decode('11111111111111111111111111111112')
pubkey = prikey.pubkey()
print(pubkey) # 6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
```

2.11.3 公钥的应用

Solana 的公钥在网络中有多种用途, 有下面的几种常见使用场景.

账户标识

在 solana 中, 每个账户都由一个公钥唯一标识. 账户可以存储 sol(solana 的原生代币), 代币, 程序数据等. 无论是普通用户账户还是智能合约(称为程序账户), 都以公钥作为地址. 在 solana 中, 账号的地址就是其公钥的 base58 表示, 因此用户实际上是通过公钥接收 sol 或其他代币. 因此, 您可以将公钥分享给他人, 让他们向您的账户转账. **公钥可以公开, 但私钥必须严格保密**.

> 在转账时, 仔细核对公钥, 避免因输入错误导致资金损失.

交易验签

当用户发起一笔交易(如转账, 调用程序等), 需要使用对应的私钥对交易进行签名. 网络通过公钥验证签名的有效性, 确保交易是由合法账户持有人发出的.

权限管理

Solana 中的账户支持权限分离. 例如, 一个公钥可以被指定为"只读"权限, 而另一个公钥拥有"签名"权限. 这种设计在去中心化应用和团队账户管理中非常有用.

程序交互

在 solana 上, 智能合约(称为程序, program)也有自己的公钥. 用户通过指定程序的公钥调用其功能, 公钥在此起到定位和验证的作用.

2.12 Solana/私钥, 公钥与地址/地址

嗨, 伙计, 我早就告诉过你, solana 的地址就是它公钥的 base58 表示……

不过嘛, 我还是忍不住想再啰嗦一遍这个知识点, 毕竟这是搞技术的人都要知道的基础知识嘛.

哪天你要是在公钥和地址这两个概念上闹出了笑话, 说出 "我是通过看这种垃圾博客文章来学习的 solana" 这种话, 要是传到作者的耳朵里, 想必作者的脸色也会不好看.

所以我特意创作了这一小节, 并将其伪装成技术文章, 塞入目录列表中.

记住这个知识点, 以后看到那些奇怪的字母和数字, 别忘了它们既是公钥, 也是地址.

2.13 Solana/私钥, 公钥与地址/地址伪装转账攻击与虚荣地址

Solana 作为一个高性能的区块链平台, 因其快速交易和低费用吸引了大量用户和开发者. 然而, 随着其普及, 针对用户的攻击手法也变得更加多样化. 从 2023 年开始, 一种名为 "地址伪装转账攻击" 的攻击方式开始在网络上大量出现. 这是一种利用地址相似性迷惑受害者的黑客策略.

2.13.1 地址伪装转账攻击

地址伪装转账攻击是指黑客生成一个与受害者已有地址前几位和后几位高度相似的 solana 地址(通常称为"虚荣地址"), 然后利用该地址向受害者发送少量代币或资产. 这种攻击旨在诱骗受害者误以为该转账来自合法来源, 进而促使受害者在后续交互中将资金转移到黑客控制的地址.

例如, 假设受害者的钱包地址是:

```
7x9kPqM...XyZ3mN
```

黑客可能生成一个虚荣地址, 如:

```
7x9kaA2...QxZ3mN
```

两者的开头和结尾几乎相同, 只有中间部分略有差异. 由于用户在日常使用中往往只检查地址的首尾, 这种相似性极易导致混淆.

攻击的实施方式非常简单, 首先黑客使用专门的工具生成大量 solana 公钥私钥对, 直到找到一个与目标地址首尾相似的地址. Solana 的地址是基于公钥的 32 字节 base58 编码, 生成特定模式的地址需要较多的计算能力, 但并非不可行. 之后黑客从伪装地址向受害者发送少量 sol 或代币(如 0.001 sol), 并可能附带一个看似合法的备注, 例如"测试转账"或"退款". 受害者在钱包中看到这笔转账时, 可能误以为是自己认识的地址发来的. 受害者可能尝试"归还"这笔资金, 或在后续交互中不小心将大额资金发送到伪装地址. 黑客则通过监控链上交易, 迅速转移收到的资金.

这种攻击在 solana 生态中已被多次报告. 例如, 2023 年, 一些用户在收到不明来源的小额转账后, 因未仔细核对地址而将资金转回伪装地址, 导致损失数千美元的资产. 由于 solana 交易是不可逆的, 一旦资金转出, 用户几乎无法追回.

> 请仔细核对完整地址: 不要仅依赖首尾几位, 始终检查地址的完整字符串, 尤其是在发送大额交易时.

2.13.2 虚荣地址

我们尝试使用 pxsol 来生成一个虚荣地址. 我们的受害者是 `6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt`, 而我们要完成的目标是获得一个公私钥对, 使其地址的前两位和后两位与受害者地址相同. 功能脚本如下.

```
import pxsol

target = '6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt'

for _ in range(1 << 32):
    prikey = pxsol.core.PriKey.random()
    pubkey = prikey.pubkey()
    pubkey_base58 = pubkey.base58()
    if pubkey_base58[:2] == target[:2] and pubkey_base58[-2:] == target[-2:]:
        print(prikey)
        print(pubkey)
        break
```

我在一台家用机上运行此脚本, 花费了约 3 个小时获得了一个满足要求的私钥和公钥.

```
prikey = D44NkELmJ6pQ13qDpA983i82iPDy9yi6VPJ3xRHv3v1R
pubkey = 6Ak9ou3WLs8uxme4vqMyNsUHiyY4inK3wLRtwgCC7uWt
```

Python 不以性能见长: 如果您可以使用 c, rust 或者 go 语言的 ed25519 库来进行这个工作, 相信速度应该会有上千倍的提升.

2.14 Solana/私钥, 公钥与地址/Base58

Base58 是一种将二进制数据转换为人类可读文本的编码方式, 因其字符集包含 58 个字符而得名. 它最初由比特币的创始人中本聪设计, 用于比特币地址的生成, 后来被广泛应用于区块链, 加密货币和其他技术领域.

2.14.1 Base58 的起源

Base58 的起源可以追溯到 2008 年, 当时中本聪发布了比特币白皮书并开始开发比特币协议. 在设计比特币地址时, 他需要一种方法将复杂的公钥哈希转换为简洁, 用户友好的字符串. 传统的 base64 编码虽然高效, 但包含了容易混淆的字符(如 0 和 O, I 和 l)以及特殊符号(+ 和 /), 不适合手动输入或视觉识别.

为了解决这个问题, 中本聪在 2009 年提出了 base58 编码方案. 他从 base64 的 64 个字符中剔除了 0, O, I 和 l, 并去除了特殊符号, 最终定义了一个由 58 个字符组成的集合:

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

此外, 中本聪还引入了一种带校验位的 base58 变体, 通过添加额外的校验位增强了错误检测能力.

2.14.2 Base58 的工作原理

Base58 编码的过程类似于进制转换.

1. 将二进制数据视为一个大整数.
2. 用这个整数反复除以 58, 取余数映射到字符集中的字符.
3. 重复此过程直到商为 0, 生成最终字符串.

但有一点需要额外注意, 我们需要将十六进制值开头的每个零字节 (0x00) 转换为 base58 中的 1. 在数字开头放置零不会增加其大小(例如 0x12 与 0x0012 相同), 因此当我们转换为 base58 时, 开头的任何额外零都不会影响结果. 为了确保前导零对结果有影响, base58 编码包含一个手动步骤, 将所有前导 0x00 转换为 1.

例: 有十六进制数据 `ef5557e913d5e13e9390a2fb0eeca75d739eccd5249dc174587669db471ca1f2df10d7e17a`, 获取它的 base58 表示.

答:

```
import pxsol

data = bytearray.fromhex('ef5557e913d5e13e9390a2fb0eeca75d739eccd5249dc174587669db471ca1f2df10d7e17a')
print(pxsol.base58.encode(data)) # 92EW9Qnnov7V3QLqToHsFNyEnQ6vvJdYiLgBTfLCv3J5XJjnh1K
```

2.14.3 Base58 的代码实现

您可以在 pxsol.base58 找到一份 base58 的简单 python 实现.

```
# Copyright (C) 2011 Sam Rushing
# Copyright (C) 2013-2014 The python-bitcoinlib developers
#
# This file is part of python-bitcoinlib.
#
# It is subject to the license terms in the LICENSE file found in the top-level
# directory of this distribution.
#
# No part of python-bitcoinlib, including this file, may be copied, modified,
# propagated, or distributed except according to the terms contained in the
# LICENSE file.
```

```
# Base58 encoding and decoding

B58_DIGITS = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'


def encode(b: bytearray) -> str:
    # Encode bytes to a base58-encoded string
    assert isinstance(b, bytearray)
    # Convert big-endian bytes to integer
    n = int.from_bytes(b)
    # Divide that integer into bas58
    res = []
    while n > 0:
        n, r = divmod(n, 58)
        res.append(B58_DIGITS[r])
    res = ''.join(res[::-1])
    # Encode leading zeros as base58 zeros
    czero = 0
    pad = 0
    for c in b:
        if c == czero:
            pad += 1
        else:
            break
    return B58_DIGITS[0] * pad + res


def decode(s: str) -> bytearray:
    # Decode a base58-encoding string, returning bytes.
    if not s:
        return bytearray()
    # Convert the string to an integer
    n = 0
    for c in s:
        n *= 58
        assert c in B58_DIGITS
        digit = B58_DIGITS.index(c)
        n += digit
    # Convert the integer to bytes
    res = bytearray(n.to_bytes(max((n.bit_length() + 7) // 8, 1)))
    # Add padding back.
    pad = 0
    for c in s[:-1]:
        if c == B58_DIGITS[0]:
            pad += 1
        else:
            break
    return bytearray(pad) + res
```

2.15 Solana/私钥, 公钥与地址/公私钥对

在 solana 生态系统, 尤其是钱包应用中, 您可能更经常遇到一个叫做公私钥对的概念. 所谓的公私钥对, 其本质上就是将私钥(在前)与公钥(在后)简单的链接起来. 这个被链接起来的字节串经常以不同的名称和形象出现在网络上, 迷惑着前赴后继的新手们.

2.15.1 Keypair(id.json)

Solana 的官方发行包中存在一个命令行钱包实现, 可以前往源代码仓库下载并安装它: https://github.com/anza-xyz/agave

在安装完成后, 您应当能找到一个叫做 solana-keygen 的工具, 我们可以使用该工具生成一个新的钱包:

```
$ solana-keygen new
```

工具会生成一个新的密钥对, 并将公钥和私钥保存在一个 json 文件中(通常是 ~/.config/solana/id.json). 您会看到类似以下的输出:

```
Wrote new keypair to /home/ubuntu/.config/solana/id.json
==========================================================================
pubkey: 6ASf...GWt
==========================================================================
Save this seed phrase and your BIP39 passphrase to recover your new keypair:
smart mutual resist shrimp fever parrot suit kidney public unhappy fringe kiwi
==========================================================================
```

文件中保存的是一个长度为 64 的字节数组. 该数组前 32 位是私钥, 后 32 位则是该私钥对应的公钥.

```
$ cat ~/.config/solana/id.json

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 76, 181, 171, 246, 173, 1
```

例: 以上 id.json 所对应的私钥是什么? 账户地址是什么?

答:

```
import pxsol

idjson = bytearray([
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
    0x4c, 0xb5, 0xab, 0xf6, 0xad, 0x79, 0xfb, 0xf5, 0xab, 0xbc, 0xca, 0xfc, 0xc2, 0x69, 0xd8, 0x5c,
    0xd2, 0x65, 0x1e, 0xd4, 0xb8, 0x85, 0xb5, 0x86, 0x9f, 0x24, 0x1a, 0xed, 0xf0, 0xa5, 0xba, 0x29,
])

prikey = pxsol.core.PriKey(idjson[:32])
pubkey = pxsol.core.PubKey(idjson[32:])
assert prikey.pubkey() == pubkey
print(prikey) # 1111111111111111111111111111112
print(pubkey) # 6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
```

2.15.2 Keypair 的 base58 表示

在 solana 生态里, 普通用户可能更多的使用浏览器钱包, 例如流行的 phantom 钱包. 这类钱包的一个常用功能就是可以导入或者导出私钥.

但是这类浏览器钱包大都犯了一个错误: 他们导入或导出的实际上是公私钥对, 而不是私钥, 尽管您在页面上看到的提示词语是私钥.

就作者所观察的几个钱包, 当您试图导入"私钥"时, 应当复制到输入框中的是公私钥对的 base58 表示; 而钱包导出的"私钥"其实也是公私钥对的 base58 表示.

例: 小明使用 solana-keygen 生成了一个新的钱包, 他应当如何将该钱包导入到 phantom 钱包里?

答: 使用 base58 编码 ~/.config/solana/id.json 文件中的数据, 得到字符串, 复制到 phantom 钱包的"私钥"框框里即可. 代码如下:

```
import pxsol

idjson = bytearray([
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
    0x4c, 0xb5, 0xab, 0xf6, 0xad, 0x79, 0xfb, 0xf5, 0xab, 0xbc, 0xca, 0xfc, 0xc2, 0x69, 0xd8, 0x5c,
    0xd2, 0x65, 0x1e, 0xd4, 0xb8, 0x85, 0xb5, 0x86, 0x9f, 0x24, 0x1a, 0xed, 0xf0, 0xa5, 0xba, 0x29,
])
print(pxsol.base58.encode(idjson)) # 11111111111111111111111111111111PPm2a2NNZH2EFJ5UkEjkH9Fcxn8cvjTmZDKQQisyLDmA
```

例: 有私钥 `11111111111111111111111111111112`, 应当如何将其导入到 phantom 钱包里?

答: 我们通过私钥来生成 keypair 的 base58 表示.

```
import pxsol

prikey = pxsol.core.PriKey.base58_decode('11111111111111111111111111111112')
pubkey = prikey.pubkey()
print(pxsol.base58.encode(prikey.p + pubkey.p))
# 11111111111111111111111111111111PPm2a2NNZH2EFJ5UkEjkH9Fcxn8cvjTmZDKQQisyLDmA
```

2.15.3 Keypair 的 base64 表示

Solana 在历史上还存在过一种以 base64 表示的公私钥对格式, 该格式是将公私钥对以 base64 编码而成.

提醒: 以 **base64** 编码而成.

当您使用 solana 生态里的应用时, 应当特别当心这种格式. 我在使用某个 solana 钱包时, 数次尝试导入私钥无果, 遂无能狂怒研究其源代码, 发现其采用了这种编码格式.

幸运的是, 经我考证, 该格式仅在 solana 极早期有过小范围的使用, 在当前时间节点下除非是考古人员否则很难再遇到它.

3. Solana/交易

3.1 Solana/交易/导言

学生: "教授, 我最近在研究 solana 区块链, 看到有很多提到交易这个概念. 我知道它是用来转账 sol 的, 但到底什么是 solana 的交易? 它的作用是什么?"

教授: "好问题! 其实 solana 的交易不仅仅是转账 sol 那么简单. 它是区块链世界中所有操作的核心. 当你在 solana 上做任何事情, 无论是转账, 调用智能合约, 还是与去中心化应用互动, 你都在发起一笔交易. 所以, 可以把交易看作是你与 solana 区块链交互的通行证."

学生: "哦, 原来是这样. 那它的结构是什么样的? 我听说 solana 交易比一些其他区块链要快, 是不是和交易的结构也有关系?"

教授: "非常对! Solana 的交易结构其实比很多区块链要简单高效. 它包含几个关键部分."

1. 签名: 每笔交易都需要由发送者签名. 这就像你在合同上签字, 确保你是合法发起交易的人.
2. 账户信息: 交易会包含参与交易的账户, 最基本的是发件人和收件人的账户, 还有可能涉及其他账户, 比如智能合约的账户.
3. 指令: 指令是交易的具体操作. 如果你是转账, 那就只有一个简单的转账指令. 但如果你是在调用智能合约, 这部分可能就复杂多了, 包含一系列指令, 告诉 solana 网络该如何处理.
4. 费用: 每笔交易都会有一定的费用, 这就是你支付给网络的通行费, 用来奖励验证交易的节点.
5. 附加数据: 有时你还会在交易中加上一些额外的数据, 比如智能合约的参数或其他特定的设置.

学生: "看来 solana 的交易就像是一个经过精心打包的包裹, 里面有所有的信息, 准备送到目标地址!"

教授: "很形象! 正是这样. Solana 的交易结构设计得很紧凑, 让整个网络能够快速处理大量交易."

学生: "这听起来像 solana 在效率方面做了很多优化. Solana 的交易怎么做到这么快呢?"

教授: "这个就涉及到 solana 的核心技术了. Solana 的高性能来源于几项创新, 最重要的就是历史证明和交易并行处理."

学生: "哇, 简直像是给区块链装上了高速公路, 大家都能快速通行!"

教授: "说得好! Solana 的设计就像给所有交易开辟了快速车道, 而没有交通堵塞. 所以, Solana 能够在几乎没有延迟的情况下处理大量交易."

学生: "这真是比我想象的要复杂多了!"

教授: "哈哈, 区块链的世界远比它看起来的要有趣. Solana 的交易只是其中的一个组成部分, 但它是让这个生态系统高速运转的关键. 继续探索吧, 你会发现更多惊人的东西."

3.2 Solana/交易/货币面值

Solana 网络的原生加密货币称为 sol. 它类似于其他区块链的代币, 如比特币的 btc, 以太坊的 eth 等等, 用来支付交易费用, 参与网络治理, 以及充当一种价值存储工具.

Solana 网络中最小的计量单位被称作 lamport. 类似于比特币中的"聪", solana 也需要有一种更小的单位来精确表示非常小的交易金额. 1 sol 等于 10^9 lamport. 这使得在处理微小交易时, 不会因为单位限制而引发精度丢失问题.

例: 0.33 sol 等于多少 lamport?

答: 0.33 * 10^9 = 330000000 lamport. 库 `pxsol.denomination` 中定义了 sol 和 lamport 的面值, 可以使用如下代码进行转换.

```
import pxsol

print(int(0.33 * pxsol.denomination.sol))
# 330000000
```

Lamport 是 sol 的子单位, 类似于美元与美分之间的关系(1 美元 = 100 美分). 在 solana 网络中, 所有的余额和交易量都是以 lamport 来表示的, 但用户通常以 sol 为单位与网络交互, 进行交易或转账时, sol 是用户界面上显示的单位.

关于 lamport 这个名称的由来, solana 创始人 anatoly yakovenko 在设计时决定用 "lamport" 来致敬计算机科学领域的著名学者 leslie lamport. Leslie lamport 是分布式系统和并发计算领域的先驱, 他的研究成果对现代区块链技术有着重要的影响. 特别是在分布式系统中的时间同步问题和共识机制方面, lamport 的贡献为 solana 的高效共识算法(历史证明)提供了理论支持.

至于为什么选择 sol 作为代币的名称, 作者没有找到确切的证据, 但能略微猜测一二:

1. 币圈项目通常使用三个字母作为代币名称, 在 solana 开发阶段, sol 恰巧是一个未被使用的名称.
2. 容易让用户联想到"太阳"(solar)的意思, 寓意一个充满活力, 不断成长并提供光明的未来.

在最后, 我会给您一些建议. 您在 solana 网络中进行一切活动时, 都应当特别注意:

1. Lamport 是 solana 网络的真实存在的最小单位, sol 则更多是一种象征符号, 并不真实存在.
2. 作为开发者, 您总是应该操作 lamport, 而仅在最后展示给用户时, 将其转换为 sol.

3.3 Solana/交易/构建本地开发环境

交易是 solana 系统中最为核心的部分之一. 在本章节中, 我们将深入分析 solana 的各种交易形式, 包括如何创建交易, 为交易签名, 以及将交易发送至验证者节点, 使其成为 solana 系统永久交易记录的一部分.

在区块链上执行交易需要支付手续费, 而在主网上进行测试成本较高. 因此, 搭建一个本地 solana 开发链对于开发和测试至关重要. 本文将详细介绍如何在本地搭建 solana 开发链, 并为您的账户获取 sol 空投, 以便用于测试和开发.

3.3.1 获取 solana 官方发布包

2024 年 3 月 2 日, Solana 经历了一次重大的人事变动: solana 客户端的开发工作从 solana labs 移交至 anza 团队. 该团队 fork 了 solana 的源代码, 此后所有开发工作均在新 fork 的仓库中进行.

您可以通过当时的新闻报道了解更多细节: https://solana.com/news/solana-labs-anza-fork.

这次变动导致原有的 solana 仓库失效, 新 fork 的仓库则命名为 agave. 因此, 现在 solana 唯一的官方仓库地址为: https://github.com/anza-xyz/agave.

在该仓库的发布页面找到二进制发布包, 下载并解压. 以 ubuntu 系统为例, 可使用以下命令完成操作:

```
$ wget https://github.com/anza-xyz/agave/releases/download/v2.0.20/solana-release-x86_64-unknown-linux-gnu.tar.bz2
$ tar -xvf solana-release-x86_64-unknown-linux-gnu.tar.bz2
$ cd solana-release
```

> 注意: 本文撰写时, agave 的最新版本为 v2.0.20. 但随着时间推移, 该版本可能已不是最新版本. 请始终选择发布页面上显示的最新版本!

安装完成后, 可通过以下命令检查 Solana 是否成功安装:

```
$ solana --version
# solana-cli 2.0.20 (src:20a8749f; feat:607245837, client:Agave)
```

如果显示 solana 的版本信息, 说明安装成功.

3.3.2 启动 solana 本地开发链

Solana 提供了一个名为 solana-test-validator 的工具, 可以帮助我们在本地启动一个 solana 开发链. 执行以下命令来启动本地测试链:

```
$ solana-test-validator
```

此命令将启动一个本地的 solana 区块链实例, 默认情况下, 它会在端口 8899 上提供 api 接口. 你可以看到类似下面的输出, 表示本地链正在运行:

```
# Identity: H3SmeomgugZP3AYzgeLuL9ZfHQrUWahFBkzeiwdzCmaE
# Genesis Hash: B1Kc8gnygWEW6ZXxV4STaheM9WAWNYqukTLuxoiCWjUX
# Version: 2.0.20
# Shred Version: 57683
# Gossip Address: 127.0.0.1:1024
# TPU Address: 127.0.0.1:1027
# JSON RPC URL: http://127.0.0.1:8899
# WebSocket PubSub URL: ws://127.0.0.1:8900
```

3.3.3 配置 solana 命令行工具

为了让 solana 命令行工具与本地链进行交互, 您需要设置 solana 使用本地链的配置. 运行以下命令:

```
$ solana config set --url http://localhost:8899
```

3.3.4 创建 solana 钱包

执行以下命令:

```
$ solana-keygen new
```

这将生成一个新的密钥对并保存到 ~/.config/solana/id.json 文件中. 你可以通过以下命令查看该账户的公钥:

```
$ solana address
# 6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
```

例: 通过上述方式创建的钱包是随机的, 不同用户生成的私钥和地址均不同. 回顾第一章内容, 若需让 solana 命令行工具使用私钥为 0x01 的账户, 应如何操作?

答: id.json 文件存储了账号的公私钥对. 我们编辑 ~/.config/solana/id.json, 将其内容替换为以下代码生成的目标公私钥对:

```
import pxsol

prikey = pxsol.core.PriKey.int_decode(0x01)
pubkey = prikey.pubkey()
print(list(prikey.p + pubkey.p))
```

3.3.5 获取 solana 测试硬币

在本地链上, 您可以通过请求空投 sol 来为钱包账户提供一些测试资金. 使用以下命令为你的账户获取空投 sol:

```
$ solana airdrop 10
```

此命令将为你当前配置的账户空投 10 个 sol. 如果你需要空投更多 sol, 可以调整数字, 例如:

```
$ solana airdrop 100
```

您还可以为空投至指定账户, 添加目标地址作为参数, 例如:

```
$ solana airdrop 10 6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
```

你可以通过以下命令查看账户的余额, 确认空投是否成功:

```
$ solana balance
```

如果一切顺利, 你将看到账户中已经有了空投的 sol.

3.3.6 注意事项

1. 本地开发链会消耗一定量的内存, CPU 和磁盘空间. 连续运行 24 小时可能占用约 8g 磁盘空间, 请提前规划.
2. 本地开发链的数据可能不会保存, 重启后需要重新请求空投. 但您不需要重新创建账号.
3. 如果你想回到主网或其他网络(如 devnet), 可以用 `solana config set --url <URL>` 切换, 例如: `solana config set --url https://api.devnet.solana.com`

3.3.7 准备就绪

至此, 您已成功搭建本地 solana 开发链, 并为账户空投了 sol 测试代币.

接下来是……

3.4 Solana/交易/使用内置钱包进行转账

Pxsol 的内置钱包允许读者在 python 中直接管理 solana 账户, 通过私钥派生公钥(钱包地址), 从而执行如查看余额, 交易转账等操作.

3.4.1 创建您的私钥

要使用 pxsol 的内置钱包, 首先需要一个私钥对象. 有多种途径来初始化您的私钥. 以下是几种创建私钥的方式:

从字节数组

```
import pxsol

prikey = pxsol.core.PriKey(bytearray([0x00] * 31 + [0x01]))
assert prikey.pubkey().base58() == '6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt'
```

从 base58 字符串

```
import pxsol

prikey = pxsol.core.PriKey.base58_decode('11111111111111111111111111111112')
assert prikey.pubkey().base58() == '6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt'
```

从 hex 字符串

```
import pxsol

prikey = pxsol.core.PriKey.hex_decode('0000000000000000000000000000000000000000000000000000000000000001')
assert prikey.pubkey().base58() == '6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt'
```

从整数

```
import pxsol

prikey = pxsol.core.PriKey.int_decode(0x01)
assert prikey.pubkey().base58() == '6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt'
```

从 keypair(base58)

```
import pxsol

prikey = pxsol.core.PriKey.wif_decode('11111111111111111111111111111111PPm2a2NNZH2EFJ5UkEjkH9Fcxn8cvjTmZDKQQisyLDmA')
assert prikey.pubkey().base58() == '6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt'
```

以上所有方法都是严格等价的.

3.4.2 使用内置钱包进行转账

Pxsol 的子模块 `pxsol.wallet` 实现了一个简单但是功能强大的内置钱包.

Ada 正在泰国享受他的假期. 他进入了一家海鲜餐厅, 美美的吃上了一顿. 在结账的时候, 他注意到这家餐厅允许顾客使用 solana 支付账单. Ada 决定试一试, 他现在需要向店主 bob 支付 1 sol. 为了实现这个过程, ada 首先用自己的私钥初始化了自己的钱包. 要完成这笔交易, ada 还需要完成两个步骤:

- 填写 bob 的 solana 公钥(地址).
- 要发送的金额, 以 lamport 表示.

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
bob = pxsol.core.PubKey.base58_decode('8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH')
ada.sol_transfer(bob, 1 * pxsol.denomination.sol)
```

> 绝密信息: bob 的地址所对应的私钥为 0x02.

执行代码, ada 的交易就将发送到 solana 网络, 任何错误都将不可逆转. 因此, ada 仔细检查了地址和金额, 确保万无一失. Pxsol 的钱包会构建一笔交易, 从 ada 获取 1 sol 的资金并发送到 bob 的地址, 最后使用 ada 的私钥签署交易.

在完成交易之后, ada 检查了自己钱包的余额.

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
print(f'ada: {ada.sol_balance() / pxsol.denomination.sol} sol')
# ada: 499999998.999995 sol
```

细心的 ada 发现, 自己的余额减少的数字略微大于 1 sol, 这其间的差值是该交易所支付的手续费.

Solana 拥有无与伦比的交易确认速度: 通常在您发出交易的瞬间, 交易就会被确认. 因此 pxsol 的内置钱包采用了同步交易确认方式: 只有当交易被网络确认, `sol_transfer()` 方法才会返回.

3.5 Solana/交易/交易详情

在上一节中, ada 向 bob 转移了 1 sol. 我们想查看这笔交易的详细内容, 为此, 我们使用 `pxsol.rpc.get_signatures_for_address` 来查询 ada 过去一段时间内的所有交易历史. 接口将返回交易的序列化格式数据, 对其进行反序列化就能得到一个 `pxsol.core.Transaction` 对象.

```
import base64
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))

for e in pxsol.rpc.get_signatures_for_address(ada.pubkey.base58(), {'limit': 32}):
    tx_meta = pxsol.rpc.get_transaction(e['signature'], {'encoding': 'base64'})
    tx_byte = base64.b64decode(tx_meta['transaction'][0])
    tx = pxsol.core.Transaction.serialize_decode(tx_byte)
    print(tx)
```

> 如果存在多个交易, 交易将按照交易时间倒序排序输出.

我们检索到 ada 的原始交易, 对其进行解码, 格式化并查看它包含的内容. 结果如下所示:

```
{
    "signatures": [
        "3NPdLTf2Xp1XUu82VVVKgQoHfiUau3wGPTKAhbNzm8Rx5ebNQfHBzCGVsagXyQxRCeEiGr1jgr4Vn32UEAx1Aov3"
    ],
    "message": {
        "header": [
            1,
            0,
            1
        ],
        "account_keys": [
            "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
            "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
            "11111111111111111111111111111111"
        ],
        "recent_blockhash": "6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx",
        "instructions": [
            {
                "program": 2,
                "account": [
                    0,
                    1
                ],
                "data": "3Bxs3zzLZLuLQEYX"
            }
        ]
    }
}
```

实际的交易看起来和我们期待的不太一样. 您可能注意到, account_keys 的前两个条目中包含了 ada 和 bob 的地址, 但第三个条目一连串的 1 又表示的什么呢? Ada 所发送的 1 sol 输入又在哪里?

别担心, 接下来的时间里, 我会向您介绍交易中的每一个字段.

3.6 Solana/交易/签名与验证

3.6.1 示例交易

> 为了避免读者不停切换页面, 我将待分析的交易复制在了每一小节的开头.

```
{
    "signatures": [
        "3NPdLTf2Xp1XUu82VVVKgQoHfiUau3wGPTKAhbNzm8Rx5ebNQfHBzCGVsagXyQxRCeEiGr1jgr4Vn32UEAx1Aov3"
    ],
    "message": {
        "header": [
            1,
            0,
            1
        ],
        "account_keys": [
            "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
            "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
            "11111111111111111111111111111111"
        ],
        "recent_blockhash": "6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx",
        "instructions": [
            {
                "program": 2,
                "account": [
                    0,
                    1
                ],
                "data": "3Bxs3zzLZLuLQEYX"
            }
        ]
    }
}
```

3.6.2 签名

一个典型的 solana 交易分为两大核心部分: 签名和消息. 签名是交易的"通行证". 签名通过私钥生成, 确保交易的真实性和不可篡改性. 消息部分则明确指定操作细节, 通过这种分离, solana 可以并行处理大量交易, 极大提升吞吐量.

每个签名的长度均为 64 字节, 签名数组的长度取决于 message.header 中指定的签名数量(稍后会解释). 在这个例子中, 只有一个签名, 表明交易由一个账户发起并验证.

> 多签名交易(例如需要多个账户授权)会包含多个签名.

签名由账户的私钥对序列化后的交易的"消息"部分进行签名生成.

例: 请使用 ada 的私钥 0x01 对上述交易进行签名, 并验证签名结果是否和交易中的签名一致.

答:

```
import pxsol

tx = pxsol.core.Transaction.serialize_decode(bytearray([
    0x01, 0x76, 0x7a, 0xe2, 0x66, 0x60, 0xc1, 0x42, 0x94, 0x1a, 0x59, 0x61, 0xf6, 0xde, 0xc7, 0x23,
    0x7c, 0xae, 0x73, 0x3e, 0xdf, 0xe6, 0x51, 0x7c, 0x37, 0xfb, 0xb8, 0x48, 0x1f, 0x46, 0xbb, 0xb5,
    0x3c, 0xe3, 0x00, 0xe7, 0x14, 0xb4, 0x78, 0x40, 0x14, 0x2c, 0x93, 0xa4, 0xe6, 0x60, 0x0c, 0x50,
    0xfd, 0xa9, 0x75, 0x60, 0xab, 0x64, 0x1d, 0xb0, 0xce, 0x19, 0x55, 0x9b, 0x25, 0x1d, 0x66, 0xdf,
```

```
    0x04, 0x01, 0x00, 0x01, 0x03, 0x4c, 0xb5, 0xab, 0xf6, 0xad, 0x79, 0xfb, 0xf5, 0xab, 0xbc, 0xca,
    0xfc, 0xc2, 0x69, 0xd8, 0x5c, 0xd2, 0x65, 0x1e, 0xd4, 0xb8, 0x85, 0xb5, 0x86, 0x9f, 0x24, 0x1a,
    0xed, 0xf0, 0xa5, 0xba, 0x29, 0x74, 0x22, 0xb9, 0x88, 0x75, 0x98, 0x06, 0x8e, 0x32, 0xc4, 0x44,
    0x8a, 0x94, 0x9a, 0xdb, 0x29, 0x0d, 0x0f, 0x4e, 0x35, 0xb9, 0xe0, 0x1b, 0x0e, 0xe5, 0xf1, 0xa1,
    0xe6, 0x00, 0xfe, 0x26, 0x74, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x57, 0xe9, 0x77, 0x4a, 0x3c, 0xad, 0x5c, 0x33, 0xf1, 0xfb, 0x6b,
    0x37, 0xa0, 0x3d, 0x4f, 0x00, 0x9a, 0x31, 0x09, 0x81, 0x18, 0xd2, 0xce, 0xae, 0xbf, 0x43, 0x0a,
    0xf3, 0x01, 0xad, 0x25, 0x0d, 0x01, 0x02, 0x02, 0x00, 0x01, 0x0c, 0x02, 0x00, 0x00, 0x00, 0x00,
    0xca, 0x9a, 0x3b, 0x00, 0x00, 0x00, 0x00,
]))

s = pxsol.eddsa.sign(bytearray([0x00] * 31 + [0x01]),  tx.message.serialize())
assert s == tx.signatures[0]
```

3.6.3 验证过程

当交易提交到 solana 网络时, 验证节点首先会检查 `tx.signatures` 的数量是否与 `tx.message.header` 匹配, 然后使用账户的公钥(来自 `tx.message.account_keys`) 验证每个签名是否有效.

在我们的例子中, 签名对应 `tx.message.account_keys` 中的第一个账户: `6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt`. 这个账户是交易的发起者, 且支付了交易费用.

例: 验证上述交易的签名有效.

答:

```
import pxsol

tx = pxsol.core.Transaction.serialize_decode(bytearray([
    0x01, 0x76, 0x7a, 0xe2, 0x66, 0x60, 0xc1, 0x42, 0x94, 0x1a, 0x59, 0x61, 0xf6, 0xde, 0xc7, 0x23,
    0x7c, 0xae, 0x73, 0x3e, 0xdf, 0xe6, 0x51, 0x7c, 0x37, 0xfb, 0xb8, 0x48, 0x1f, 0x46, 0xbb, 0xb5,
    0x3c, 0xe3, 0x00, 0xe7, 0x14, 0xb4, 0x78, 0x40, 0x14, 0x2c, 0x93, 0xa4, 0xe6, 0x60, 0x0c, 0x50,
    0xfd, 0xa9, 0x75, 0x60, 0xab, 0x64, 0x1d, 0xb0, 0xce, 0x19, 0x55, 0x9b, 0x25, 0x1d, 0x66, 0xdf,
    0x04, 0x01, 0x00, 0x01, 0x03, 0x4c, 0xb5, 0xab, 0xf6, 0xad, 0x79, 0xfb, 0xf5, 0xab, 0xbc, 0xca,
    0xfc, 0xc2, 0x69, 0xd8, 0x5c, 0xd2, 0x65, 0x1e, 0xd4, 0xb8, 0x85, 0xb5, 0x86, 0x9f, 0x24, 0x1a,
    0xed, 0xf0, 0xa5, 0xba, 0x29, 0x74, 0x22, 0xb9, 0x88, 0x75, 0x98, 0x06, 0x8e, 0x32, 0xc4, 0x44,
    0x8a, 0x94, 0x9a, 0xdb, 0x29, 0x0d, 0x0f, 0x4e, 0x35, 0xb9, 0xe0, 0x1b, 0x0e, 0xe5, 0xf1, 0xa1,
    0xe6, 0x00, 0xfe, 0x26, 0x74, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x57, 0xe9, 0x77, 0x4a, 0x3c, 0xad, 0x5c, 0x33, 0xf1, 0xfb, 0x6b,
    0x37, 0xa0, 0x3d, 0x4f, 0x00, 0x9a, 0x31, 0x09, 0x81, 0x18, 0xd2, 0xce, 0xae, 0xbf, 0x43, 0x0a,
    0xf3, 0x01, 0xad, 0x25, 0x0d, 0x01, 0x02, 0x02, 0x00, 0x01, 0x0c, 0x02, 0x00, 0x00, 0x00, 0x00,
    0xca, 0x9a, 0x3b, 0x00, 0x00, 0x00, 0x00,
]))

assert pxsol.eddsa.verify(tx.message.account_keys[0].p, tx.message.serialize(), tx.signatures[0])
```

3.7 Solana/交易/序列化与反序列化

3.7.1 示例交易

> 为了避免读者不停切换页面, 我将待分析的交易复制在了每一小节的开头.


```
{
    "signatures": [
        "3NPdLTf2Xp1XUu82VVVKgQoHfiUau3wGPTKAhbNzm8Rx5ebNQfHBzCGVsagXyQxRCeEiGr1jgr4Vn32UEAx1Aov3"
    ],
    "message": {
        "header": [
            1,
            0,
            1
        ],
        "account_keys": [
            "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
            "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
            "11111111111111111111111111111111"
        ],
        "recent_blockhash": "6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx",
        "instructions": [
            {
                "program": 2,
                "account": [
                    0,
                    1
                ],
                "data": "3Bxs3zzLZLuLQEYX"
            }
        ]
    }
}
```


3.7.2 序列化与反序列化

Solana 的交易在签名, 传播和存储时, 它们是序列化的. 序列化是指将数据结构转换为字节流的过程, 以便在网络中传输或存储. 反序列化则是相反的过程, 将字节流重新解析为原始的数据结构. Solana 使用一种高效的二进制格式来实现序列化和反序列化, 这种格式在性能和数据紧凑性之间取得了平衡.

Solana 的序列化过程基于一种自定义的二进制编码格式, 设计目标是高效且无歧义. 该序列化方式没有名字, 但特点明显且十分简单. 大体上来说, 它基于以下两个基本规则:

- 紧凑编码: solana 使用变长整数(compact-u16)来表示长度字段. 例如, 账户数量或签名数量会根据实际值的大小动态编码, 而不是固定占用 2 个字节. 这种方式下减少了不必要的空间浪费.
- 顺序存储: 交易的各个部分按固定顺序排列, 例如先存储签名数量, 再存储签名数据, 然后是消息内容. 这种顺序性简化了反序列化的逻辑.

例如, 一个简单的转账交易在序列化后可能是这样的字节流:

1. 签名数量(1 字节或更多, 变长编码).
2. 签名数据(每个 64 字节, 按顺序排列).
3. 消息头部(3 字节).
4. 账户数量(1 字节或更多, 变长编码).
5. 账户地址(每个 32 字节, 按顺序排列).
6. 最近区块哈希(32 字节).
7. 指令数量(1 字节或更多, 变长编码).
8. 指令内容
    1. 程序索引(1 字节).
    2. 账户数量(1 字节或更多, 变长编码).
    3. 账户索引(每个 1 字节, 按顺序排列).
    4. 参数长度(1 字节或更多, 变长编码).
    5. 参数数据.

3.7.3 使用变长编码表示长度

Solana 序列化算法中采用的变长整数编码称作 compact-u16. 该算法的核心思想是将 16 位整数(最大值 65535)用 1 到 3 个字节表示, 具体字节数取决于数值的大小. 它的编码规则类似于传统 vlq 编码, 通过在每个字节中使用 7 位来存储实际数据, 并用最高位(第 8 位)作为"延续位"来指示是否需要读取后续字节.

具体编码过程如下:

- 小于 128(0x7f)的值, 只需 1 个字节. 最高位设为 0, 表示没有后续字节. 剩余 7 位存储数值.

例: 数值 5(二进制 00000101)编码是多少?

答:

```
import pxsol

assert pxsol.core.compact_u16_encode(5) == bytearray([0x05])
```

- 128 到 16383(0x3fff)的值, 需要 2 个字节. 第一个字节的最高位设为 1, 表示有后续字节; 低 7 位存储数值的低 7 位. 第二个字节的最高位设为 0, 表示结束; 低 7 位存储数值的剩余部分.

例: 数值 132(二进制 10000100)编码是多少?

答: 第一个字节: 0x84(10000100, 延续位 1, 数据 0000100). 第二个字节: 0x01(00000001, 延续位 0, 数据 0000001).

```
import pxsol

assert pxsol.core.compact_u16_encode(132) == bytearray([0x84, 0x01])
```

- 大于 16383 的值, 需要 3 个字节. 前两个字节的延续位都设为 1, 分别存储低 14 位. 第三个字节的延续位设为 0, 存储剩余部分.

例: 数值 65535(二进制 11111111 11111111)编码是多少?

答:

```
import pxsol

assert pxsol.core.compact_u16_encode(65535) == bytearray([0xff, 0xff, 0x03])
```

在 solana 中, 交易内部的数据通常较小, 长度一般会在 128 以内. 使用 compact-u16, 这些值可以用单个字节表示, 而不是固定使用 2 个字节, 从而减少传输和存储成本.

3.7.4 习题

例: 您能从序列化的十六进制交易形式手动解码 ada 的交易吗? 交易数据如下.

```
01767ae26660c142941a5961f6dec7237cae733edfe6517c37fbb8481f46bbb53ce300e714b4784
0142c93a4e6600c50fda97560ab641db0ce19559b251d66df04010001034cb5abf6ad79fbf5abbc
cafcc269d85cd2651ed4b885b5869f241aedf0a5ba297422b9887598068e32c4448a949adb290d0
f4e35b9e01b0ee5f1a1e600fe2674000000000000000000000000000000000000000000000000
0000000000000057e9774a3cad5c33f1fb6b37a03d4f009a31098118d2ceaebf430af301ad250d0
1020200010c0200000000ca9a3b00000000
```

答:

```
01                                                                 tx.signatures.length
767ae26660c142941a5961f6dec723....7560ab641db0ce19559b251d66df04 tx.signatures[0]
01                                                                 tx.message.header[0]
00                                                                 tx.message.header[1]
01                                                                 tx.message.header[2]
03                                                                 tx.message.account_keys.length
4cb5abf6ad79fbf5abbccafcc269d85cd2651ed4b885b5869f241aedf0a5ba29 tx.message.account_keys[0]
7422b9887598068e32c4448a949adb290d0f4e35b9e01b0ee5f1a1e600fe2674 tx.message.account_keys[1]
0000000000000000000000000000000000000000000000000000000000000000 tx.message.account_keys[2]
57e9774a3cad5c33f1fb6b37a03d4f009a31098118d2ceaebf430af301ad250d tx.message.recent_blockhash
01                                                                 tx.message.instructions.length
02                                                                 tx.message.instructions[0].program
02                                                                 tx.message.instructions[0].account.length
00                                                                 tx.message.instructions[0].account[0]
01                                                                 tx.message.instructions[0].account[1]
0c                                                                 tx.message.instructions[0].data.length
0200000000ca9a3b00000000                                           tx.message.instructions[0].data
```

3.7.5 注意事项

- 变长编码 compact-u16 仅用于表示长度. 交易中的程序索引, 账户索引值则是直接使用 u8 来表示.

3.8 Solana/交易/账户与权限

3.8.1 示例交易

> 为了避免读者不停切换页面, 我将待分析的交易复制在了每一小节的开头.

```
{
    "signatures": [
        "3NPdLTf2Xp1XUu82VVVKgQoHfiUau3wGPTKAhbNzm8Rx5ebNQfHBzCGVsagXyQxRCeEiGr1jgr4Vn32UEAx1Aov3"
    ],
    "message": {
        "header": [
            1,
            0,
            1
        ],
        "account_keys": [
            "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
            "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
            "11111111111111111111111111111111"
        ],
        "recent_blockhash": "6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx",
        "instructions": [
            {
                "program": 2,
                "account": [
                    0,
                    1
                ],
                "data": "3Bxs3zzLZLuLQEYX"
            }
        ]
    }
}
```

3.8.2 账户列表

在 solana 交易中, `tx.message.account_keys` 是所有参与账户的列表. 看看我们的示例:

```
"account_keys": [
    "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
    "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
    "11111111111111111111111111111111"
]
```

在示例中, 我们的交易总共涉及到三个账户, 这三个账户分别是:

1. `6ASf5Ec...`: ada, 转账的发起者, 且支付了交易费用.
2. `8pM1DN3...`: bob, 转账的接收者.
3. `1111111...`: solana 系统程序(system program). 用于处理基础操作如转账.

账户列表定义了"谁参与"了这笔交易, 您可以这么理解这笔简单的交易:

1. ada 向 bob 转移了 1 sol, 这个操作会导致二人账户的余额发生变化, 因而显然他们的账户参与了交易.
2. 转账操作需要调用 solana 系统程序中的转账函数, 这个步骤会读取系统程序的代码, 因此认为系统程序也间接参与了交易.

账户列表的顺序不是任意的, 我们继续分析.

3.8.3 权限

在 solana 的交易结构中, 账户不仅是参与者, 还承载了权限和角色的定义. 账户**是否需要签名**, **是否可以被修改**, 都直接影响交易的执行逻辑. 这些权限的设置, 与交易的 `tx.message.header` 字段密切相关.

是否需要签名?

签名账户是交易的"授权者", 通常负责支付费用或批准操作. 在我们的例子中, ada 需要对交易进行签名, 而 bob 和 solana 系统程序不需要.

是否可写?

账户是否可写决定了它在交易中是否能被修改. 只读账户只能被读取, 不能更改状态. 在我们的例子中, ada 和 bob 是可写的, 但 solana 系统程序是只读的.

我们可以使用两个比特位来表示账户的权限. 第 0 个比特位表示是否可写, 第 1 个比特位表示是否需要签名, 如此, 账户权限能被表示为 0 到 3 之间的一个数字. 账户列表必须按照权限从高到低排列:

账户索引	地址	需要签名	可写	权限(0-3)	角色
0	`6ASf5Ec...`	是	是	3	ada (付款方)
1	`8pM1DN3...`	否	是	1	bob (接收者)
2	`1111111...`	否	否	0	系统程序

账户列表的权限以一种压缩方式被保存在 `tx.message.header` 中.

```
"header": [1, 0, 1]
```

这些数字与 `tx.message.account_keys` 中的账户列表结合, 决定了每个账户的权限状态. 它是一个包含三个数字的数组, 分别表示:

- 需要签名账户数: 这里是 1, 表示需要 1 个签名, 与 `tx.signatures` 数组长度一致.
- 需要签名, 只读账户数: 这里是 0.
- 无需签名, 只读账户数: 这里是 1.

以程序代码来描述这里的逻辑的话, 可以表示如下:

- `tx.header[0]` 为权限 >= 2 的账户数量.
- `tx.header[1]` 为权限 == 2 的账户数量.
- `tx.header[2]` 为权限 == 0 的账户数量.

只读账户无需状态变更, 这一特性使得 solana 能够充分利用并行处理能力, 从而显著提升交易吞吐量. 并行执行交易一直是区块链行业面临的一大技术难题, 因为交易通常会涉及对账户状态的修改, 而不同的执行顺序可能会导致截然不同的结果. 为了确保一致性和正确性, 传统区块链系统通常采用单核单线程的方式, 按顺序逐一处理交易. 这种方法虽然简单可靠, 但极大地限制了性能的扩展, 尤其是在高负载场景下. Solana 通过识别只读账户并优化状态管理, 打破了这一瓶颈, 实现了更高效的交易处理机制.

许多公链也试图解决这个问题, 以获得更高的 qps, 例如以太坊在 eip-2930 中就打了一个补丁, 允许用户显示指定该笔交易访问了哪些账户数据, 以便节点可以并行处理互不相关的交易. 不过从作者的视角来看, 这个补丁显然十分的不成功, 在本书编写期间, 这种方式并没有在以太坊生态系统里被广泛采用. 相比之下, solana 为账号引入的权限系统, 似乎是一种更为高级和优雅的设计.

3.9 Solana/交易/区块哈希与时效性

3.9.1 示例交易

> 为了避免读者不停切换页面, 我将待分析的交易复制在了每一小节的开头.

```
{
    "signatures": [
        "3NPdLTf2Xp1XUu82VVVKgQoHfiUau3wGPTKAhbNzm8Rx5ebNQfHBzCGVsagXyQxRCeEiGr1jgr4Vn32UEAx1Aov3"
    ],
    "message": {
        "header": [
            1,
            0,
            1
        ],
        "account_keys": [
            "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
            "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
            "11111111111111111111111111111111"
        ],
        "recent_blockhash": "6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx",
        "instructions": [
            {
                "program": 2,
                "account": [
                    0,
                    1
                ],
                "data": "3Bxs3zzLZLuLQEYX"
            }
        ]
    }
}
```

3.9.2 最近区块哈希

在 solana 交易中, `tx.message.recent_blockhash` 是一个看似不起眼却至关重要的字段. 它确保交易不会无限期有效, 也防止重复执行交易.

```
"recent_blockhash": "6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx"
```

这个值由交易创建时从网络节点实时获取. 代码如下.

```
import pxsol

print(pxsol.rpc.get_latest_blockhash({})['blockhash'])
# 6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx
```

在交易中引入最近区块哈希有以下几个目的:

1. solana 要求交易在某个时间窗口内提交(通常几分钟, 具体取决于网络配置). 如果交易中提交的最近区块哈希太旧, 交易会被拒绝.
2. 防止重放攻击. 每次交易绑定一个唯一的区块哈希, 即使交易内容相同, 使用不同的哈希也能区分它们, 避免重复执行.
3. 它将交易锚定到时间线, 确保网络能够快速排序和验证交易.

3.9.3 习题

例: 尝试将 ada 的交易重新发送, 看看会发生什么?

答:

```
import base64
import pxsol

tx = bytearray([
    0x01, 0x76, 0x7a, 0xe2, 0x66, 0x60, 0xc1, 0x42, 0x94, 0x1a, 0x59, 0x61, 0xf6, 0xde, 0xc7, 0x23,
    0x7c, 0xae, 0x73, 0x3e, 0xdf, 0xe6, 0x51, 0x7c, 0x37, 0xfb, 0xb8, 0x48, 0x1f, 0x46, 0xbb, 0xb5,
    0x3c, 0xe3, 0x00, 0xe7, 0x14, 0xb4, 0x78, 0x40, 0x14, 0x2c, 0x93, 0xa4, 0xe6, 0x60, 0x0c, 0x50,
    0xfd, 0xa9, 0x75, 0x60, 0xab, 0x64, 0x1d, 0xb0, 0xce, 0x19, 0x55, 0x9b, 0x25, 0x1d, 0x66, 0xdf,
    0x04, 0x01, 0x00, 0x01, 0x03, 0x4c, 0xb5, 0xab, 0xf6, 0xad, 0x79, 0xfb, 0xf5, 0xab, 0xbc, 0xca,
    0xfc, 0xc2, 0x69, 0xd8, 0x5c, 0xd2, 0x65, 0x1e, 0xd4, 0xb8, 0x85, 0xb5, 0x86, 0x9f, 0x24, 0x1a,
    0xed, 0xf0, 0xa5, 0xba, 0x29, 0x74, 0x22, 0xb9, 0x88, 0x75, 0x98, 0x06, 0x8e, 0x32, 0xc4, 0x44,
    0x8a, 0x94, 0x9a, 0xdb, 0x29, 0x0d, 0x0f, 0x4e, 0x35, 0xb9, 0xe0, 0x1b, 0x0e, 0xe5, 0xf1, 0xa1,
    0xe6, 0x00, 0xfe, 0x26, 0x74, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x57, 0xe9, 0x77, 0x4a, 0x3c, 0xad, 0x5c, 0x33, 0xf1, 0xfb, 0x6b,
    0x37, 0xa0, 0x3d, 0x4f, 0x00, 0x9a, 0x31, 0x09, 0x81, 0x18, 0xd2, 0xce, 0xae, 0xbf, 0x43, 0x0a,
    0xf3, 0x01, 0xad, 0x25, 0x0d, 0x01, 0x02, 0x02, 0x00, 0x01, 0x0c, 0x02, 0x00, 0x00, 0x00, 0x00,
    0xca, 0x9a, 0x3b, 0x00, 0x00, 0x00, 0x00,
])

pxsol.rpc.send_transaction(base64.b64encode(tx).decode(), {})

# Exception: {
#     'code': -32002,
#     'message': 'Transaction simulation failed: This transaction has already been processed',
#     'data': {
#         'accounts': None,
#         'err': 'AlreadyProcessed',
#         'innerInstructions': None,
#         'logs': [],
#         'replacementBlockhash': None,
#         'returnData': None,
#         'unitsConsumed': 0,
#     }
# }
```

3.10 Solana/交易/指令

3.10.1 示例交易

> 为了避免读者不停切换页面, 我将待分析的交易复制在了每一小节的开头.


```
{
    "signatures": [
        "3NPdLTf2Xp1XUu82VVVKgQoHfiUau3wGPTKAhbNzm8Rx5ebNQfHBzCGVsagXyQxRCeEiGr1jgr4Vn32UEAx1Aov3"
    ],
    "message": {
        "header": [
            1,
            0,
            1
        ],
        "account_keys": [
            "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
            "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
            "11111111111111111111111111111111"
        ],
        "recent_blockhash": "6vAwzjtGMrN3mJ8o7iGVDjMM46e2AnctqmjvLbqtESrx",
        "instructions": [
            {
                "program": 2,
                "account": [
                    0,
                    1
                ],
                "data": "3Bxs3zzLZLuLQEYX"
            }
        ]
    }
}
```


3.10.2 交易中的指令

Solana 的交易就像是信封, 而指令就是信封里的内容, 每个指令都告诉 solana 网络如何处理这笔交易. 就像我们把信放进信封里一样, 之后我们会将信封蜡封, 然后盖上自己的腊封印章. 最后别忘记了, 您还需要付邮费!

- 正如信封中可以添加多封信件一样, 交易中也允许添加多个指令.
- 交易的签名就像腊封印章.
- 交易手续费就像信件邮费.

Solana 交易中的每个指令都执行特定的功能. 在示例交易中, ada 发送了一笔 sol 转账, 这笔交易背后其实就包含了一个转账指令. 这个指令告诉 solana 将指定金额从一个账户转移到另一个账户.

Solana 的交易指令有很多种, 像转账指令就是其中之一. 每种指令都由一个程序(也就是智能合约)来执行. 每个指令由三部分组成:

- program: 目标程序的索引(对应 account_keys 中的位置).
- account: 涉及的账户索引数组(对应 account_keys 中的位置).
- data: 传递给程序的二进制数据.

在示例中:

- `"program": 2`: 表示调用 `tx.message.account_keys[2]`, 即系统程序 `1111111...`.
- `"account": [0, 1]`: 表示涉及了两个账户, `tx.message.account_keys[0]` 和 `tx.message.account_keys[1]`.
- `"data": "3Bxs3zzLZLuLQEYX"`: 表示具体操作.

我们将 data 数据进行 base58 解码, 得到其十六进制表示的数据为 `0200000000ca9a3b00000000`. 系统程序 `1111111...` 会对数据进行解析, 前 4 个字节会被解析为内部函数索引, 后 8 个字节解释为转账的金额. 在示例中, 我们的内部函数索引为 2, 表示转账, 金额则为 1000000000 lamport.

```
import pxsol

data = pxsol.base58.decode('3Bxs3zzLZLuLQEYX')
assert data.hex() == '0200000000ca9a3b00000000'

assert int.from_bytes(data[:4], 'little') == 2
assert int.from_bytes(data[4:], 'little') == 1 * pxsol.denomination.sol
```

不同的指令对于 account 和 data 有不同的要求, 这点在您使用除转账之外的其它指令时, 需要尤其注意. 在下一节中, 我将向您简单介绍 solana 的系统程序以及它所包含的内部函数.

3.11 Solana/交易/系统程序

Solana 的系统程序 `1111111...` 是 solana 中最核心的程序, 负责提供基本的区块链账户管理功能. 其主要功能包括创建账户, 余额转移, 数据存储等功能. 通过在交易中调用它, 您可以实现各种复杂功能, 这也是 solana 可编程性的体现.

系统程序所支持的指令列表定义在 https://github.com/anza-xyz/solana-sdk/blob/c654e5f556ad3e22679fe9757da1bf5c9486e2f1/system-interface/src/instruction.rs#L68-L252.

您可以看到, 指令数据被定义为一个枚举类型, 并使用 bincode 编码. Bincode 是一种紧凑型编码方式, 特点是编码对象的大小将等于或小于对象在正在运行的 rust 程序中占用的内存大小.

Bincode 编码规范详见 https://docs.rs/crate/bincode/2.0.1/source/docs/spec.md.

粗略的讲, 我们目前只需要关注 bincode 中以下几条规则:

- 任何数字以小端序编码.
- 枚举拥有索引值, 被视作 u32. 编码时首先编码索引值, 然后再编码其值.

以下是对系统程序中前三条主要指令的分析. 余下的指令, 相信您在本节课程结束后有能力自行阅读学习. 加油, 奥里给!

3.11.1 创建账户

作用: 用于创建一个新的账户, 并将其所有权指定为系统程序.

位置: `SystemInstruction::CreateAccount`

涉及账户:

账户	权限	说明
资金提供者	3	必须拥有足够的 sol 余额来支付创建新账户所需的费用
新账户	3	-

> 还记得我们是如何表示账户权限的吗? 我们使用两个比特位, 第 0 个比特位表示是否可写, 第 1 个比特位表示是否需要签名.

数据格式:

名称	类型	说明
index	u32	约定为 0
lamports	u64	分配给新账户的 sol 数量
space	u64	新账户分配的存储空间大小, 以字节为单位
owner	pubkey	新账户的所有者程序的公钥

3.11.2 分配

作用: 将账户的所有权转移到指定的程序. Solana 的账户模型要求每个账户必须有一个所有者, 我们日常使用的 solana 账户, 其所有者是系统程序.

位置: `SystemInstruction::Assign`

涉及账户:

账户	权限	说明
目标账户的公钥	3	目标账户的公钥

数据格式:

名称	类型	说明
index	u32	约定为 1
owner	pubkey	目标程序的公钥

3.11.3 转移

作用: 从一个账户向另一个账户转移 sol 余额.

位置: `SystemInstruction::Transfer`

涉及账户:

账户	权限	说明
发送方公钥	3	-
接收方公钥	1	-

数据格式:

名称	类型	说明
index	u32	约定为 2
lamports	u64	要转移的 sol 数量

3.11.4 其它内置程序

Solana 系统程序的执行方式是"基于账户"的, 系统程序通过修改账户的状态和存储的数据来执行任务. 除了系统程序外, solana 还内置了一些其它程序:

- 代币程序. Solana 的代币程序允许用户创建自己的代币, 进行代币的转移, 铸造和销毁等操作. 如果您参与过 solana 链上代币交易的话, 会频繁与此程序交互. 程序源码: https://github.com/solana-program/token-2022.
- 质押程序. Solana 的质押程序允许用户将 sol 代币质押到网络中, 从而参与验证过程. 质押机制是 solana 共识算法的一部分, 允许用户通过质押来获取网络的奖励.
- 租赁程序. Solana 的租赁程序用于管理网络中账户的存储租赁机制. Solana 的账户存储空间是有限的, 因此用户在创建账户时需要支付一定的"租赁费用", 以确保他们的账户数据能够在网络中得到存储.

3.11.5 习题

例: Solana 系统程序的公钥是多少? 请以十六进制表示.

答: Solana 系统程序的公钥是一个全部为 0 的 32 字节数组. 它是一个"黑洞"账户, 理论上没有与之对应的私钥.

```
import pxsol

pubkey = pxsol.core.PubKey.base58_decode('11111111111111111111111111111111')
assert pubkey.hex() == '0000000000000000000000000000000000000000000000000000000000000000'
```

例: 如果 ada 要转账 2 sol 给 bob, 他应当如何构造指令数据?

答: 我们可以按照上述规则手工拼凑数据, 也可以借助 `pxsol.program` 模块完成数据的构建. 此处仅演示第二种方法:

```
import pxsol

data = pxsol.program.System.transfer(2 * pxsol.denomination.sol)
assert pxsol.base58.encode(data) == '3Bxs3zxH1DZVrsVy'
```

3.12 Solana/交易/手工构造交易

夫交易者, 数字雕琢而出, 财富流动之诗也.

匠人之手, 运字节于毫端, 化混沌为有序, 宛若琢玉成器, 精妙不可言.

本节课程中, 我们会学习如何以手工方式构造一笔 solana 转账交易. 通过这个过程, 希望您能更加深入了解 solana 的交易结构.

3.12.1 我们的目标是

假设 ada 要向 bob 支付 2 sol, 同时 bob 要向 cuc 支付 1 sol. 请在一个交易里实现 ada 和 bob 二人的要求.

3.12.2 定义交易的参与者

这笔交易总共有四个参与者, 分别如下:

- `ada`: 私钥为 `1`.
- `bob`: 私钥为 `2`.
- `cuc`: 公钥为 `HPYVwAQmskwT1qEEeRzhoomyfyupJGASQQtCXSNG8XS2`.

在 solana 中, 转账是通过系统程序完成的, 它的地址是固定的:

- `系统程序`: `11111111111111111111111111111111`.

> 绝密信息: cuc 的地址所对应的私钥为 0x03.

3.12.3 构造指令

Solana 的交易由一条或多条指令组成. 对于本交易, 我们需要构造两个转账指令.

Solana 交易中的指令仅存储程序和账户的索引, 而具体的账户信息(公钥和权限)却并不存储在指令中. 这让我们在构建交易时十分的被动. 为此, pxsol 实现了两个辅助数据结构:

- `pxsol.core.AccountMeta`. AccountMeta 包含账户公钥以及其账户权限. 在本示例中, ada 与 bob 的权限应当是需要签名且可写, cuc 的权限应当是无需签名且可写.
- `pxsol.core.Requisition`. Requisition 自身包含一条交易指令的全部数据, 并且能在稍后构建交易时, 将自身"编译"为使用索引的指令.

构造如下代码:

```
import pxsol

ada = pxsol.core.PriKey.int_decode(1)
bob = pxsol.core.PriKey.int_decode(2)
cuc = pxsol.core.PubKey.base58_decode('HPYVwAQmskwT1qEEeRzhoomyfyupJGASQQtCXSNG8XS2')

# Transfer from ada to bob, 2 sol
r0 = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
r0.account.append(pxsol.core.AccountMeta(ada.pubkey(), 3))
r0.account.append(pxsol.core.AccountMeta(bob.pubkey(), 1))
r0.data = pxsol.program.System.transfer(2 * pxsol.denomination.sol)

# Transfer from bob to cuc, 1 sol
r1 = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
r1.account.append(pxsol.core.AccountMeta(bob.pubkey(), 3))
```

```
r1.account.append(pxsol.core.AccountMeta(cuc, 1))
r1.data = pxsol.program.System.transfer(1 * pxsol.denomination.sol)
```

3.12.4 组建交易

使用 `pxsol.core.Transaction.requisition_decode` 方法, 将上文定义的两个请求编译为指令并组装在一个交易里. 该方法的第一个参数表示应该由谁支付手续费, 第二个参数则是 Requisition 列表. 在本示例中, 我们决定由 ada 付出交易手续费.

```
tx = pxsol.core.Transaction.requisition_decode(ada.pubkey(), [r0, r1])
```

3.12.5 获取最近区块哈希

使用 rpc 接口获取 solana 的最近区块哈希.

```
tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
```

3.12.6 签名交易

组装好交易后, ada 和 bob 需要用他们的私钥对交易进行签名. 此步骤需注意签名顺序与账户列表顺序应当匹配. 在本示例中, ada 的签名要放置在 bob 的签名之前. 您可以根据以下两条规则简单评估签名的顺序:

1. 支付手续费的账户总是在签名列表最前.
2. 可写的账户总在只读账户之前.
3. 按照账户在交易中出现的先后顺序排列.

代码调用如下:

```
tx.sign([ada, bob])
```

3.12.7 最终交易结构

使用 `print(tx)` 可以将交易完整打印如下.

```
{
    "signatures": [
        "2DNYcExSuLB1BgkB7p3gSFEuWvwgnCbcBXtENBgU9tXGQdfknvST4c3U1uQ7AEAwbEc6D1qzxMQhjdiTQytE3A24",
        "42hH4QE2r9w4yRE9CQGZq7N16Pr4712sPU6myqQNAaaAxo8A8F7HB9d46By4EVXbmRJVYMNHSgdHfXmv9XY4TFud"
    ],
    "message": {
        "header": [
            2,
            0,
            1
        ],
        "account_keys": [
            "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
            "8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH",
            "HPYVwAQmskwT1qEEeRzhoomyfyupJGASQQtCXSNG8XS2",
            "11111111111111111111111111111111"
        ],
        "recent_blockhash": "FSLe6dD3NxjJacCSW9P3LSyzpZd6H4SHSUCcCFUaTQwj",
        "instructions": [
            {
                "program": 3,
                "account": [
                    0,
                    1
```

```
                ],
                "data": "3Bxs3zzLZLuLQEYX"
            },
            {
                "program": 3,
                "account": [
                    1,
                    2
                ],
                "data": "3Bxs3zvX19cRxrhM"
            }
        ]
    }
}
```

3.12.8 提交交易

将签名后的交易序列化为字节, 通过 solana 的 rpc 接口 `send_transaction` 提交到网络. 网络将会验证交易的签名, 检查余额并执行转账.

```
txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
assert pxsol.base58.decode(txid) == tx.signatures[0]
pxsol.rpc.wait([txid])
```

几秒钟后, bob 和 cuc 的账户就会各自多了 1 sol!

3.12.9 完整代码

```
import base64
import pxsol

ada = pxsol.core.PriKey.int_decode(1)
bob = pxsol.core.PriKey.int_decode(2)
cuc = pxsol.core.PubKey.base58_decode('HPYVwAQmskwT1qEEeRzhoomyfyupJGASQQtCXSNG8XS2')

r0 = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
r0.account.append(pxsol.core.AccountMeta(ada.pubkey(), 3))
r0.account.append(pxsol.core.AccountMeta(bob.pubkey(), 1))
r0.data = pxsol.program.System.transfer(2 * pxsol.denomination.sol)

r1 = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
r1.account.append(pxsol.core.AccountMeta(bob.pubkey(), 3))
r1.account.append(pxsol.core.AccountMeta(cuc, 1))
r1.data = pxsol.program.System.transfer(1 * pxsol.denomination.sol)

tx = pxsol.core.Transaction.requisition_decode(ada.pubkey(), [r0, r1])
tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
tx.sign([ada, bob])
txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
assert pxsol.base58.decode(txid) == tx.signatures[0]
pxsol.rpc.wait([txid])
```

> 使用两个辅助数据结构: pxsol.core.AccountMeta 和 pxsol.core.Requisition 让手工构造 solana 交易变得简单和有趣!

3.13 Solana/交易/手续费

如果说区块链世界中最大的用户痛点的是什么, 那么交易手续费一定在其中. 以太坊的高 gas 费, 比特币的高网络拥堵费用都让不少人望而却步. 那么作为近年来备受瞩目的区块链平台, solana 的交易手续费是如何运作的?

3.13.1 交易手续费的基本构成

Solana 的交易手续费主要由两部分组成:

- 基础费用: 这是每笔交易必须支付的最低费用, 用于补偿网络验证者处理交易的成本.
- 优先费用(可选): 用户可以选择支付额外的费用, 以提高交易的优先级, 从而在网络繁忙时更快被打包和确认.

与以太坊等区块链不同, solana 的手续费不依赖于复杂的 gas 机制, 而是采用了一种更简单, 更可预测的收费模式. 基础费用是固定的, 并且非常低廉, 通常以 solana 的原生代币 sol 计价.

交易手续费隐式指定, 默认由交易中的第一个签名账户提供. 一个交易需要支付的总基础费用等于基础费用与签名数量的乘积.

3.13.2 具体费用

截至本文撰写时间, 即 2025 年 3 月 28 日, solana 的基础交易费用通常在 0.000005 sol 左右. 以 sol 的市场价格为例, 假设 1 sol = 137 美元, 那么一笔交易的成本大约是 0.0007 美元. 即使在网络高峰期, 用户选择支付优先费用, 总体成本也很少超过几美分. 相比之下, 以太坊的 gas 费在网络繁忙时可能高达数美元甚至数十美元, 而比特币的转账费用也常在 1-5 美元之间波动. Solana 的低成本优势显而易见.

Solana 的交易手续费之所以能保持在极低水平, 主要得益于其独特的技术架构和设计理念. Solana 的核心创新之一是其"历史证明"机制, 结合塔式拜占庭容错共识算法, 使得网络每秒可以处理数万笔交易. 相比之下, 以太坊当前主网的吞吐量仅为 15-30 tps. 更高的吞吐量意味着单位成本被大幅摊薄.

> Solana 的这种优化并非没有代价: 从本质上来说, solana 的做法是通过牺牲去中心化来获得更高的交易处理性能.

3.13.3 手续费的分配与燃烧机制

Solana 的手续费有一部分会被"燃烧", 即永久销毁, 以减少 sol 的总供应量, 从而在长期内可能对代币价值产生正向影响. 具体来说: 50% 的手续费被燃烧, 这部分直接从流通中移除. 50% 的手续费分配给验证者, 作为对维护网络安全的奖励.

3.13.4 添加优先费用

执行交易时, 网络会消耗以计算单位(compute unit)为单位的计算资源. 交易最多可使用 1,400,000 个计算单位, 默认情况下, 每条指令限制为 200,000 个计算单位. 您可以通过在交易中包含一条 `set_compute_unit_limit` 指令来请求特定的计算单位限制.

```
rq = pxsol.core.Requisition(pxsol.program.ComputeBudget.pubkey, [], bytearray())
rq.data = pxsol.program.ComputeBudget.set_compute_unit_limit(200000)
```

您可以为每个计算单位支付一点优先费用, 如果您要这么做, 通过在交易中包含一条 `set_compute_unit_price` 指令就可以了.

```
rq = pxsol.core.Requisition(pxsol.program.ComputeBudget.pubkey, [], bytearray())
rq.data = pxsol.program.ComputeBudget.set_compute_unit_price(1)
```

交易优先费用以 micro lamport 计价, 其换算规则是 1,000,000 micro lamport = 1 lamport.

> 绝密信息: 将交易优先费用指定为 1, 就能让您的交易轻松优先于未设置交易优先费用的交易! 网络上有一些实时追踪 solana 交易优先手续费的工具, 可能会建议您设置上千的费用, 但实际上, 您完全不需要支付如此庞大的费用!

3.13.5 习题

例: ada 准备向 bob 转账 1 sol, 但此时 ada 意识到网络非常拥堵, 因此他决定为这笔交易添加一点优先费用!

答:

```
import base64
import pxsol

ada = pxsol.core.PriKey.int_decode(1)
bob = pxsol.core.PubKey.base58_decode('8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH')

r0 = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
r0.account.append(pxsol.core.AccountMeta(ada.pubkey(), 3))
r0.account.append(pxsol.core.AccountMeta(bob, 1))
r0.data = pxsol.program.System.transfer(1 * pxsol.denomination.sol)

r1 = pxsol.core.Requisition(pxsol.program.ComputeBudget.pubkey, [], bytearray())
r1.data = pxsol.program.ComputeBudget.set_compute_unit_price(1)

tx = pxsol.core.Transaction.requisition_decode(ada.pubkey(), [r0, r1])
tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
tx.sign([ada])
txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
assert pxsol.base58.decode(txid) == tx.signatures[0]
pxsol.rpc.wait([txid])
```

4. Solana/账户模型

4.1 Solana/账户模型/导言

课堂上, 一位学生坐在座位上, 转动着手中的钢笔, 目光在教授的讲解中穿梭. 他低头观察手里的笔, 眉头微微皱起, 似乎在思考, 随即抬起头, 目光集中在教授的讲解上.

教授将课件放到讲台上, 目光轻松地落在学生身上, 示意他提问.

学生: "教授, 我想, 我已经开始了解一点 solana 了, 我感觉它的账户模型挺特别的. 能不能给我一个简单的概述, 帮助我搞明白到底什么是 solana 的账户? 我的 sol 余额到底是如何存储在 solana 链上的?"

教授: "哈哈, 当然! Solana 的账户模型可以说是非常简洁又高效. 你可以把 solana 的账户想象成一个个数据存储仓库, 每个账户负责存储一些数据, 但它们自己并不包含执行逻辑."

学生: "您的意思是, 账户只负责存数据, 执行逻辑的是别的地方对吧?"

教授轻轻点了点头.

教授: "是的! 根据账户中存储的数据的不同, solana 的账户可以分为三大类: **普通账户**, **程序账户**和**数据账户**. 普通账户就是你常见的钱包账户, 它们存储用户的 sol 余额, 简单直接, 只有存钱和转账的功能. 而程序账户则像是执行智能合约的操作中心, 它们存储程序代码. 最后数据账户则负责存储程序的执行状态, 控制着应用程序的行为."

教授顿了顿, 又补充说道.

教授: "严格来说, 普通账户是一种特殊的数据账户, 他们被系统程序管理着."

学生: "您所说的管理是什么意思?"

教授: "这是 solana 的另一个亮点. 每个数据账户的数据是由它的所有者控制, 程序账户则由部署它的开发者管理, 确保了每个账户的安全性和隔离性."

学生: "让我总结一下: 链上所有数据都存储在 solana 账户里. 根据账户的具体功能, 我们可以在功能上将它们分为普通账户, 程序账户和数据账户三大类. 普通账户负责存储和转账 sol 余额; 程序账户用来存储开发者部署的智能合约程序代码; 数据账户则负责存储智能合约运行中产生的数据."

教授: "是的, solana 的账户模型虽然简单, 但背后的设计非常巧妙. 如果你深入了解, 会发现它在很多方面都做出了创新的优化, 让区块链不仅更快, 而且更具可扩展性. 这是 solana 的创新点之一."

4.2 Solana/账户模型/账户数据结构

在 solana 中, 每个账户基本上是一个个"数据存储单元". 这些账户不仅用于存储 sol(solana 的原生代币), 还可以存储一些额外的数据, 比如智能合约的合约代码以及状态.

根据账户中存储的数据的不同, solana 的账户可以分为三大类:

- 普通账户: 也就是我们常用的钱包账户, 用于存储和转账 sol 余额.
- 程序账户: 用于存储智能合约的逻辑代码.
- 数据账户: 用于存储智能合约的状态数据.

值得注意的是, solana 的原生代币 sol 由一个特殊的程序账户, 即系统程序管理, 因此我们也可以说, 普通账户是一种特殊的数据账户.

Solana 账户的数据结构比较简单, 我们可以通过 rpc 来查询一个账户的详细信息. 我们只需要知道钱包地址, 就能通过 rpc 请求获取该账户的详细数据.

首先, 假设我们要查询的钱包地址是 `6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt`. 我们可以使用 solana 的 get_account_info rpc 方法来查询该钱包地址的账户信息. 方法如下:

```
import json
import pxsol

prikey = pxsol.core.PriKey.int_decode(1)
pubkey = prikey.pubkey()  # 6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
result = pxsol.rpc.get_account_info(pubkey.base58(), {})
print(json.dumps(result, indent=4))
```

当然我们也可以不借助 pxsol 而直接使用 curl 来构造请求, 其最终结果是一致的.

```
curl -X POST http://127.0.0.1:8899 \
    -H "Content-Type: application/json" \
    -d '{
        "jsonrpc": "2.0",
        "id": 1,
        "method": "getAccountInfo",
        "params": ["6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt"]
    }'
```

两种方式都能得到返回数据如下:

```
{
    "data": [
        "",
        "base64"
    ],
    "executable": false,
    "lamports": 500000000000000000,
    "owner": "11111111111111111111111111111111",
    "rentEpoch": 0,
    "space": 0
}
```

这些返回的数据字段的含义是什么呢? 让我们一一解释.

- `data`: 该字段包含了账户存储的实际数据. 数据是以 base64 编码的形式返回的, 因此你需要解码它才能看到实际内容.
  - 对于普通钱包账户, 通常这个字段是空的, 或者它包含一些额外的信息, 比如账户的状态.
  - 对于程序账户, 该字段存储了智能合约的代码.
  - 对于数据账户, 该字段包含了智能合约的状态信息.
- `executable`: 该字段是一个布尔值, 表示该账户是否是智能合约账户. 如果是智能合约账户, 这个字段会为 true, 意味着账户可以执行一些操作. 对于普通账户和数据账户, 这个字段通常是 false, 因为它只是存储数据并不执行代码.
- `lamports`: 该字段表示账户的余额. Solana 的最小货币单位是 lamport, 1 sol 等于 10^9 lamport. 在这个例子中, 500000000000000000 lamport 等于 500000000 sol. 所以你可以通过这个字段看到账户当前的 sol 余额.
- `owner`: 该字段表示这个账户是由哪个程序控制的. 对于普通钱包账户, 这个字段通常是 `11111111111111111111111111111111`, 它表示 solana 系统程序管理着该账户. 对于数据账户而言, 该字段就会是它所关联着的程序账户.
- `rentEpoch`: 一个遗留字段, 源自 solana 曾经有一个机制定期从账户中扣除 lamport. 虽然此字段仍然存在于账户类型中, 但自从 solana v1.14 版本周期性租金收取功能被彻底弃用后, 它已不再使用.
- `space`: 简单表示 `data` 字段的长度. 一个账户最多可以存储 10 兆的数据.

您可以看到, 不同类型的账户, 其底层其实共享着完全相同的数据结构, 我们更多是从功能上对账户类型进行区分. 与其他区块链相比, solana 的账户模型简化了许多复杂的管理机制(当然, 老实说, 也带来了一些新的问题, 要解释清楚这件事情相当复杂...).

4.3 Solana/账户模型/未花费交易输出模型与账户模型

想象一下, 你正在处理两种不同的银行账户系统: 一个是储蓄罐式的, 另一个是传统账户式的. 这两种设计虽然都能帮助你管理资产, 但它们的工作方式有很大不同.

这两种不同的思想, 衍生出了区块链世界中的两种截然不同的设计方向: 未花费交易输出模型(Unspent transaction output)和账户模型.

4.3.1 未花费交易输出模型

未花费交易输出模型最早由比特币引入, 并成为比特币区块链的核心设计.

想象你有一个零钱罐, 里面装着很多不同面额的硬币. 每当你收钱时, 它们就会作为独立的硬币放入零钱罐中, 每个硬币代表你能花费的金额. 而当你需要支付某个商品时, 你并不从账户余额中直接扣钱, 而是拿出足够的硬币来支付, 然后将剩余的部分重新放回零钱罐里. 换句话说, 每次你交易, 都是拆分或者合并这些硬币.

在比特币中, 这些硬币就是未花费交易输出. 每个交易输出就是一个独立的金额, 它只能花费一次, 并且当你花费它时, 系统会创建新的交易输出. 例如, 你收到的 100 个比特币可能会分成 10 个 10 比特币的未花费交易输出, 而当你需要花费其中 45 个比特币时, 你就要选择将这 5 个 10 比特币的未花费交易输出合并起来, 然后将剩余的 5 个比特币作为新的未花费交易输出返回给自己.

您可以将上述过程再次用零钱罐类比: 您的零钱罐中有 10 张 10 美元的纸币, 现在您需要花费 45 美元, 因此您选择从零钱罐中取出 5 张 10 美元, 然后将商家的 5 美元找零重新存入自己的零钱罐中.

未花费交易输出模型的特点就像零钱罐一样:

- 您可以拥有多个零钱罐.
- 每笔交易都像是从零钱罐里挑出合适的硬币来支付, 支付后剩余的硬币又被放回零钱罐中.
- 每个硬币都是独立的, 互不相连, 因此有一定的隐私性, 别人无法知道你所有的硬币加起来有多少.
- 灵活性高, 因为你可以随意组合和拆分硬币.

4.3.2 账户模型

账户模型是目前被更多区块链所使用的一种记账方式. 账户模型最初来源于以太坊区块链, 如今被很多其他区块链项目用来记录交易和状态变化. 与零钱罐不同, 账户模型就像你在银行开了一个账户, 每次你存入或取出资金时, 银行只会更新你账户的余额. 你不需要知道具体是哪个硬币被拿走或者存入了, 只要账户上的数字变了, 你就可以放心地管理你的资产.

在账户模型中, 资产直接存储在账户中, 而不是分散在多个交易输出中. 每当你进行一笔交易时, 区块链系统只需要更新你的账户余额. 这就像你转账给朋友时, 只需要指定金额, 银行会自动从你的账户中扣除相应的钱, 而无需你指定具体的硬币.

Solana 正是采用这种账户模型. 当你向朋友转账时, 系统只会减少你账户中的余额, 并将相同金额添加到你朋友的账户中.

账户模型的特点就像银行账户一样:

- 交易过程简单直接, 你只需关心账户余额的增减.
- 相比于未花费交易输出模型, 账户模型的隐私性较差, 因为账户的余额随时可以查询, 容易被追踪.
- 更适合处理复杂的交易逻辑, 特别是智能合约, 因为这些合约本质上就是在账户之间管理资金流转.

4.3.3 思考

从比特币到 solana, 两种账户模型各有千秋, 设计背后的理念和适用场景不同. 站在作者的个人角度来看, 我认为比特币设计的未花费交易输出模型更加具有设计上的美感, 但于此同时又不得不承认其在用户使用上并不友好. 那么读者对这两者是怎么思考的呢?

4.4 Solana/账户模型/所有权和权限控制

Solana 的账户数据结构中拥有一个 `owner` 字段, 该字段指向一个程序账户, 作为该账户的所有者. 只有所有者程序可以更改账户的 data 字段和从账户余额中扣除余额.

Solana 账户的实际功能依赖于账户所有者设定的规则, 我们以现实中的银行系统进行类比:

Ada 在泰国汇商银行(siam commercial bank)开设了一个银行账户, 并在账户中存入了一些资金. 在这个场景下:

- Ada 拥有账户内资金的所有权和有限使用权.
- Ada 账户的所有权仍归属银行. 银行账户是依附于银行的工具, 是银行的附属和延伸.
- Ada 只允许执行银行允许其执行的操作, 例如转账, 汇款等. Ada 无法销毁账户内的资金, 因为银行不允许她这么做.

我们再次强调一遍: 每个 solana 账户都与一个程序账户相关联, 而该程序的设计会定义账户如何被访问和操作. 所有者程序账户可以是 solana 系统程序, 也可以是用户自己部署的智能合约, 这种设计下使得 solana 非常灵活, 可以实现针对账户非常细粒度的控制.

4.5 Solana/账户模型/普通钱包账户

在 solana 网络上, 使用最广泛的账户类型就是普通钱包账户. 这些账户通常由钱包应用生成, 帮助用户存放和管理 sol. 此页面列举出了常见的 solana 钱包, 您可以根据您的喜好任意选择您喜欢且信任的钱包应用.

不过我依然有一些建议, 也许可以帮助在区块链世界冲浪的您:

1. 始终在官网下载钱包应用, 并且不要相信搜索引擎. 依记得在 2022 年左右, 网络上开始出现大量钓鱼网站, 引导用户下载被黑客修改过的钱包应用. 这些钓鱼网站甚至购买了搜索引擎的排名, 使得用户搜索钱包名称时, 钓鱼网站搜索结果可以排序在官网之上.
2. 选择拥有良好声誉的开发团队的钱包应用. 开发团队不应当有黑历史, 或者来路不明.
3. 选择拥有更多用户数量的钱包应用.

4.5.1 创建普通钱包账户

创建这样一个账户其实很简单, 唯一需要做的事情就是生成一个私钥. 通过私钥派生出的公钥就是您钱包的地址, 别人可以用它给您转钱; 私钥则是您控制钱包的密码, 只有您知道它, 别人不能随便用. 通过 pxsol, 使用私钥 0x02 生成一个新的账户如下.

```
import pxsol

bob = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x02))
ret = pxsol.rpc.get_account_info(bob.pubkey.base58(), {})
print(ret) # None
```

新创建的钱包账户此时并未被 solana 网络记录, 因此当上述代码尝试使用 get_account_info 查询账户信息时, 返回的结果是 `None`.

向这个账户转账 sol 可以"激活"这个账户.

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
bob = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x02))

# Transfer 1 sol from ada to bob.
ada.sol_transfer(bob.pubkey, 1 * pxsol.denomination.sol)

ret = pxsol.rpc.get_account_info(bob.pubkey.base58(), {})
print(ret)
# {
#     "data": ["", "base64"],
#     "executable": false,
#     "lamports": 1000000000,
#     "owner": "11111111111111111111111111111111",
#     "rentEpoch": 18446744073709551615,
#     "space": 0,
# }
```

您可以看到, 此时 bob 的账户信息已经被记录到了 solana 网络上. 账户的所有者是 `1111111...`, 也就是 solana 系统程序.

4.5.2 销毁普通钱包账户

如果您不再需要某个钱包账户了, 或者想要清理账户, 您可以销毁它. 销毁钱包账户其实并不复杂. 假如我们想要销毁上面创造的 bob 账户, 只需要把账户里的 sol 全部转移到另一个账户里. 没有余额的账户将立即被销毁.

转账的命令是这样的:

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
bob = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x02))

# Transfer all lamports from bob to ada.
bob.sol_transfer(ada.pubkey, bob.sol_balance() - 5000)

ret = pxsol.rpc.get_account_info(bob.pubkey.base58(), {})
print(ret)
# None
```

Bob 在构造交易时, 必须预留 5000 lamport 手续费, 然后将剩余 sol 全部转账给 ada. 待转账完成后, bob 账户将被立即销毁, 您将无法在 solana 网络上查询到 bob 账户的信息.

您可以轻松地创建和销毁 solana 上的普通钱包账户. 通常来说, 创建钱包账户是很常见的操作, 而销毁账户则多发生在您不再使用某个账户时, 或者您想清理一下自己的账户列表.

4.6 Solana/账户模型/程序账户

Solana 上的程序账户就是一个部署了 solana 程序(智能合约)的账户. 程序部署完成后, 它就拥有一个"程序账户地址", 其他人可以通过这个地址来调用它. Solana 上的程序通常是用 rust 写的, 需要先编译成 bpf 字节码格式. 在之后的章节中我们将详细讨论 solana 上的智能合约, 现在先让我们专注于存储程序的容器, 也就是程序账户.

4.6.1 部署程序

作为一名老练的开发者, 我们已经在职业生涯中编写并运行过大量的 hello world 代码. 今天也不例外, 我们会尝试在 solana 网络上部署并运行一段简单的代码. 恰巧, pxsol 的资源目录中保存了一份 hello world 代码, 您可以通过以下命令下载它:

```
$ wget https://raw.githubusercontent.com/mohanson/pxsol/refs/heads/master/res/hello_solana_program.so
```

这个 `hello_solana_program.so` 文件, 这就是您的程序代码, 它将被上传到 solana 链上.

```
import pathlib
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))

program_data = pathlib.Path('hello_solana_program.so').read_bytes()
program_pubkey = ada.program_deploy(bytearray(program_data))
print(program_pubkey) # 3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja
print(pxsol.rpc.get_account_info(program_pubkey.base58(), {}))
# {
#     "data": [
#         "AgAAAKqeeWx5rwCATKoazfymul8X00alxPltuX+elp+32dxO",
#         "base64"
#     ],
#     "executable": true,
#     "lamports": 1141440,
#     "owner": "BPFLoaderUpgradeab1e11111111111111111111111",
#     "rentEpoch": 18446744073709551615,
#     "space": 36
# }
```

我们在代码中, 简单的使用 `program_deploy()` 即可将程序部署到 solana 网络上. 在上述例子里, 程序被部署在了地址为 `3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja` 的程序账户里.

4.6.2 程序账户的权限和状态

部署后的程序账户由 bpf upgradeable loader, 即 `BPFLoaderUpgradeab1e11111111111111111111111` 所拥有, 这个 loader 控制是否可以升级程序.

账户信息里的 `executable` 被标记为 true, 表示它是一个程序账户, 可以执行代码.

Solana 包含少量原生程序, 这些程序是运行验证器节点所必需的. 与第三方程序不同, 原生程序是 solana 网络的一部分. 我们之前提到的用于 sol 转账的 solana 系统程序, 以及 bpf upgradeable loader 都是 solana 原生程序.

此页面列举了 solana 当前存在的全部原生程序.

4.6.3 调用程序

Solana 里的程序就像是个链上工具人, 只要您发出合法的指令, 它就能帮你完成一些预设的基础活!

每次您要调用链上程序, 就要给它发送一个交易, 交易中包含一个指令, 指令的目标程序是它, 然后告诉它您想干嘛.

在我们的 `hello_solana_program.so` 程序里, 这个程序会向任何调用自己的用户发送一条"你好"的消息. 让我们试试调用它!

```
import base64
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))

rq = pxsol.core.Requisition(pxsol.core.PubKey.base58_decode('3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja'), [], bytearray
tx = pxsol.core.Transaction.requisition_decode(ada.pubkey, [rq])
tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
tx.sign([ada.prikey])
txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
pxsol.rpc.wait([txid])
r = pxsol.rpc.get_transaction(txid, {})
for e in r['meta']['logMessages']:
    print(e)

# Program 3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja invoke [1]
# Program log: Hello, Solana!
# Program log: Our program's Program ID: 3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja
# Program 3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja consumed 11759 of 200000 compute units
# Program 3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja success
```

在第二行输出, 我们收到了来自程序发出的 `Hello, Solana!` 消息.

Hello, Solana!

4.6.4 等等, 程序在那儿?

我们在部署程序后就立即查询了程序账户的信息, jsonrpc 接口返回信息如下:

```
{
    "data": [
        "AgAAAKqeeWx5rwCATKoazfymul8X00alxPltuX+elp+32dxO",
        "base64"
    ],
    "executable": true,
    "lamports": 1141440,
    "owner": "BPFLoaderUpgradeab1e11111111111111111111111",
    "rentEpoch": 18446744073709551615,
    "space": 36
}
```

但事情似乎有点不对劲. 程序账户里存储的数据 data 是不是有点...太少了? 毕竟我们程序的本体可是有足足 38936 字节那么大!

```
$ ls hello_solana_program.so
# -rwxrwxr-x  1 ubuntu ubuntu 38936 Sep 13  2024 hello_solana_program.so
```

实际上, 在这个例子里, 程序账户存储的是"程序元信息", 而不是程序代码本体!

因为历史原因, solana 支持两种部署模式:

模式	所有者	描述
不可升级程序	bpf loader	字节码直接存进程序账户的 data 里
可升级程序	bpf upgradeable loader	程序账户只是壳子, 真正的字节码存放在另一个叫做 program data account 的账户里

不可升级程序在 solana 网络上已经事实上被弃用, 因此 pxsol 不再支持不可升级程序, 正因如此您部署的是一个可升级的 solana 程序, 这时候程序账户(就是你部署出来的那个地址)里的 data 其实并不直接存储整个 bpf 字节码, 而是一个指向 program data account 的"指针".

我们解码 data 数据 `AgAAAKqeeWx5rwCATKoazfymul8XO0alxPltuX+elp+32dxO` , 得到:

```
import base64

data = base64.b64decode('AgAAAKqeeWx5rwCATKoazfymul8X00alxPltuX+elp+32dxO')
print(data.hex())
# 02000000aa9e796c79af00804caa1acdfca6ba5f17d346a5c4f96db97f9e969fb7d9dc4e
```

这个数据结构由 bpf upgradeable loader 管理, 格式大致是:

```
pub enum UpgradeableLoaderState {
    /// Account is not initialized.
    Uninitialized,
    /// A Buffer account.
    Buffer {
        /// Authority address
        authority_address: Option<Pubkey>,
        // The raw program data follows this serialized structure in the
        // account's data.
    },
    /// An Program account.
    Program {
        /// Address of the ProgramData account.
        programdata_address: Pubkey,
    },
    // A ProgramData account.
    ProgramData {
        /// Slot that the program was last modified.
        slot: u64,
        /// Address of the Program's upgrade authority.
        upgrade_authority_address: Option<Pubkey>,
        // The raw program data follows this serialized structure in the
        // account's data.
    },
}
```

- `02000000` 表示当前的枚举类型索引.
- `aa9e796c79af00804caa1acdfca6ba5f17d346a5c4f96db97f9e969fb7d9dc4e` 表示的则是 program data account 地址.

这一次, 我们查询 program data account 的账户信息, 得到消息如下:

```
import pxsol

program_data_pubkey_byte = bytearray.fromhex('aa9e796c79af00804caa1acdfca6ba5f17d346a5c4f96db97f9e969fb7d9dc4e')
program_data_pubkey = pxsol.core.PubKey(program_data_pubkey_byte)

r = pxsol.rpc.get_account_info(program_data_pubkey.base58(), {})
print(r)
# {
#     "data": [
#         "AwAAACwBAAAAAAAAAUy...AAAAAAAAAAAAAAAAAAAAAA==",
#         "base64"
#     ],
```

```
#     "executable": false,
#     "lamports": 543193200,
#     "owner": "BPFLoaderUpgradeab1e11111111111111111111111",
#     "rentEpoch": 18446744073709551615,
#     "space": 77917
# }
```

可以确认, `hello_solana_program.so` 的字节码确实直接塞在了 data 里.

4.7 Solana/账户模型/数据账户

在 solana 网络上, 程序的可执行代码与程序状态被存储在不同的账户中. 这类似于操作系统通常将程序和其数据分开存储在不同的文件中.

一个带状态的程序如果要在链上"安家落户", 它得找个地方放自己的变量, 状态或配置数据.

不过, 在 solana 上, 程序本身不能自己创建数据账户, 它只能让调用者, 也就是发交易的人, 在执行时预先签名并通过程序指令来由系统程序代创建, 并将数据账户的所有者设定为您的自定义程序, 这样您的自定义程序就能在数据账户的 data 里写入, 修改或者删除数据.

4.7.1 如何手动创建数据账户

如果您准备创建一个数据账户, 您首先需要明确以下三件事情.

- 数据账户的所有者程序是谁?
- 数据账户的空间是多少? 也就是您预期数据账户里会被写入多少数据?
- 数据账户的初始余额是多少?

创建账户需要使用 solana 系统程序的 create_account 指令. 我们举一个简单的例子, 对此进行演示.

例: 创建一个新的随机账户, 要求:

- 所有者程序指定为系统程序, 即 `11111111111111111111111111111111`.
- 数据账户的空间为 64 字节.
- 数据账户的初始余额为 1 sol.

答: 代码如下:

```
import base64
import json
import pxsol
import random

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
tmp = pxsol.wallet.Wallet(pxsol.core.PriKey.random())

rq = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
rq.account.append(pxsol.core.AccountMeta(ada.pubkey, 3)) # Funding account
rq.account.append(pxsol.core.AccountMeta(tmp.pubkey, 3)) # The new account
rq.data = pxsol.program.System.create_account(
    pxsol.denomination.sol, # Initial lamports
    64, # Data size for the new account
    pxsol.program.System.pubkey # Owner
)

tx = pxsol.core.Transaction.requisition_decode(ada.pubkey, [rq])
tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
tx.sign([ada.prikey, tmp.prikey])
txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
pxsol.rpc.wait([txid]) # Waiting for the transaction to be processed

r = pxsol.rpc.get_account_info(tmp.pubkey.base58(), {})
print(json.dumps(r, indent=4))
# {
#     "data": [
#         "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA==",
```

```
#         "base64"
#     ],
#     "executable": false,
#     "lamports": 1000000000,
#     "owner": "11111111111111111111111111111111",
#     "rentEpoch": 18446744073709551615,
#     "space": 64
# }
```

4.7.2 Ada 和"泰铢币"的烦恼

Ada 的假期非常愉快, 但她也有一点小烦恼.

她虽然可以使用 sol 来支付她的餐费与娱乐费用, 但由于在现实世界中, 几乎所有商品都是以法币定价的, 因此每当她要付费时, 都需要查询 sol 当前的市价, 以决定应当转账多少 sol 给商家.

为此, 她准备在 solana 上写一个叫"泰铢币"的程序, 它不是真的代币, 但程序里能记录谁拥有多少"泰铢币".

她写好了程序, 并部署了它. 接着, 她用自己的钱包创建了一个数据账户, 专门用来存储自己的泰铢币余额. 比如她创建了一个账户地址: `ThbAdaBalance111...`, 这个账户归她的程序控制, owner 是她的程序地址, data 里记录着她的余额, 比如 1000 泰铢币.

Ada 向朋友 bob 推荐了这个程序并邀请他参与测试. 朋友 bob 也创建好了自己的数据账户, 接着, 他想给 ada 转 100 泰铢币. 他找到了 ada 的钱包地址, 开心地打开程序, 准备发钱……

但是他懵了:

"等等, ada 存储泰铢币的账户地址是啥? 不是她的钱包地址吗?"

麻烦来了, solana 的数据账户是可以随便创建的, 跟钱包地址无关! Ada 可能用任意地址当做自己泰铢币的数据账户地址, 只要她预先创建好, 让程序控制就行了.

这下 bob 可郁闷了, 每次给人转账之前还得"问一嘴你那个...那个数据账户地址是多少?"

Ada 心想: 如果可以从我的普通钱包账户确定性的派生出泰铢币的数据账户地址就好了!

4.7.3 程序派生地址

为了解决这个问题, solana 引入了一种特殊地址叫 pda(program derived address), 也就是程序派生地址. 它是从程序公钥和自定义的种子(通常是用户的普通钱包地址)推导出来的地址, 而且这个地址是唯一且可预测的.

也就是说, 在上面的例子里, 只要 bob 知道泰铢币的程序地址和 ada 的普通钱包地址, bob 就能算出 ada 的泰铢币数据账户地址, 而不需要 ada 告诉他.

所以, ada 把系统升级了一下, 以后每个用户的泰铢币余额都必须保存在 pda 上.

例: 假设 ada 的泰铢币程序地址是 `F782pXBcfvHvb8eJfrDtyD7MBtQDfsrihSRjvzwuVoJU`, 那么 ada 关于这个程序的 pda 地址是多少?

答: 在 pxsol 中, 我们可以通过 `derive_pda()` 函数来计算一个程序对于特定账户的 pda 地址.

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
thb = pxsol.core.PubKey.base58_decode('F782pXBcfvHvb8eJfrDtyD7MBtQDfsrihSRjvzwuVoJU')
pda = thb.derive_pda(ada.pubkey.p)
print(pda) # HCPe787nPq7TfjeFivP9ZvZwejTAq1PGGzch93qUYeC3
```

4.7.4 小结

数据账户是程序用来存储自身状态数据的地方. 理论上, 任何账户都能作为数据账户. 不过, solana 上的开发者通常更习惯使用 pda 来确定性的生成数据账户.

您知道吗? 在上一小节中我们部署的 `hello_solana_program.so` 程序的 bpf 字节码, 实际上也位于一个 pda 账户内, 您可以使用下面的代码进行验证:

```
import pxsol

program = pxsol.core.PubKey.base58_decode('3EwjHuke6N6CfWPQdbRayrMUANyEkbondw96n5HJpYja')
program_data = pxsol.core.PubKey.hex_decode('aa9e796c79af00804caa1acdfca6ba5f17d346a5c4f96db97f9e969fb7d9dc4e')
assert pxsol.program.LoaderUpgradeable.pubkey.derive_pda(program.p) == program_data
```

有了 pda, 链上生活真是舒服多了!

4.8 Solana/账户模型/程序派生地址算法解析

在 solana 上开发程序, 您早晚会需要使用程序派生地址(pda 地址). 简单说, 这是一种专属于您程序的数据账户地址, 可以用来保存数据, 但它跟普通钱包地址不一样, 它**没有私钥**, 别人也没法控制它, 只有您的程序能指挥它做事.

4.8.1 如何生成

程序派生地址是程序根据一组种子(可以是字符串, 钱包地址等)加上自己的公钥算出来的一个地址. 它不是随机生成的, 而是可以预先计算, 可预测的. 这个地址不会对应任何私钥, 所以没人能签名控制它, 包括程序作者本人.

这对去中心化程序(比如链上钱包, 订单簿, 投票系统)来说特别重要, 因为开发者经常要给每个用户分配一个账户, 但不想让他们自己管理这个账户的密钥, 否则用户可以直接手动修改数据账户内的数据为任意数据.

上个小节中有个例子:

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
thb = pxsol.core.PubKey.base58_decode('F782pXBcfvHvb8eJfrDtyD7MBtQDfsrihSRjvzwuVoJU')
pda = thb.derive_pda(ada.pubkey.p)
print(pda) # HCPe787nPq7TfjeFivP9ZvZwejTAq1PGGzch93qUYeC3
```

这样几行代码就能生成一个程序派生地址, 通常我们会把它当作用户在该程序下的数据存储账户.

4.8.2 算法解释

Solana 的密钥对是 ed25519 曲线上的点, 包含一个公钥和对应的私钥. 公钥用作链上账户的地址.

程序派生地址是一个通过预定义输入集有意生成的点, 必须位于 ed25519 曲线之外. 位于 ed25519 曲线之外的点没有有效的对应私钥, 无法执行普通意义上的签名操作. 相应的, solana 网络为程序派生地址开了个特殊的后门, 程序可以为自己的派生地址进行模拟签名操作.

生成程序派生地址算法过程如下:

1. 把种子和程序公钥拼在一起.
2. 给它加一个叫 bump 的值(从 255 开始向下尝试).
3. 每次拼完后算哈希, 看看算出来的地址在不在椭圆曲线上.
4. 若地址不在曲线上, 说明这个地址不可能被签名控制, 可以用作程序派生地址.

详细代码实现如下:

```
class PubKey:
    # Solana's public key is a 32-byte array. The base58 representation of the public key is also referred to as the
    # address.

    def __init__(self, p: bytearray) -> None:
        assert len(p) == 32
        self.p = p

    def derive_pda(self, seed: bytearray) -> typing.Self:
        # Program Derived Address (PDA). PDAs are addresses derived deterministically using a combination of
        # user-defined seeds, a bump seed, and a program's ID.
        # See: https://solana.com/docs/core/pda
        data = bytearray()
        data.extend(seed)
```

```
        data.append(0xff)
        data.extend(self.p)
        data.extend(bytearray('ProgramDerivedAddress'.encode()))
        for i in range(255, -1, -1):
            data[len(seed)] = i
            hash = bytearray(hashlib.sha256(data).digest())
            # The pda should fall off the ed25519 curve.
            if not pxsol.eddsa.pt_exists(hash):
                return PubKey(hash)
        raise Exception
```

4.8.3 习题

例: Bump 是干嘛的?

答: 它是用来避免地址冲突的调节器. 如果某个种子组合生成的地址不合法, 程序就试着调 bump, 直到找到一个合法地址.

例: 程序派生地址能自己发交易吗?

答: 不行. 程序派生地址没有私钥, 因此它不能直接发交易, 只能被程序调用和控制.

4.9 Solana/账户模型/租赁与租赁豁免机制

在 solana 上, 每个账户都要消耗存储空间, 而这些空间背后是集群中的节点在为您存数据. 所以, solana 设计了一个租赁机制: 您要为您在链上存的每一字节数据付租金. 但又因为频繁续租太麻烦, 也设计了租赁豁免制度.

4.9.1 租赁

Solana 的账户不是免费的, 它占用存储资源, 而节点们要保存这些数据. 因此, solana 要求每个账户必须有一部分 sol 用作数据存储租金.

这套机制的核心规则是:

- 账户存储数据越多, 占用的空间越大, 所需的租金也越多.
- 租金是预付的, 按照一定周期定时从您的账户中扣款.
- 如果账户余额不足, 或者长期空着不用, 系统可以把它回收掉.

这笔租金实际上是您存在账户余额里的一部分 sol, solana 运行时会根据当前租金费率计算出租期.

设计租赁机制的根本原因是为了防止账户垃圾, 这些垃圾账户可能被有意或无意的创建:

- 用户创建大量空账户, 从不清理.
- 合约部署后遗留无用数据.
- 程序错误导致堆积大量临时账户.

如果账户没有成本, 链上的状态会无限膨胀, 最终影响性能和稳定性. 通过租赁机制, solana 迫使用户对链上状态负责, 用经济手段做资源管理.

4.9.2 租赁豁免

Solana 也知道频繁续租会很麻烦, 所以它设计了租赁豁免机制: 只要您账户里的 sol 足够多, 超过一个阈值, 系统就默认您预付了永久租金, 您的账户就不会被回收了.

假设您创建了一个数据账户, 占用了 100 字节. 系统会根据当前的租金费率算出一笔豁免租金, 比如 0.002 sol. 您只要在账户里放进去这 0.002 SOL, 系统就视作豁免, 系统永远不会向您再收租, 您的账户也永远不会被删掉.

这套机制设计得非常实用, 带来了几个好处:

- 开发者不需要给账户设置租期, 续租逻辑.
- 用户不需要关心账户什么时候过期.
- 合约运行更加稳定.

Solana 从一开始就采用了租赁和租赁豁免并存的机制. 但在 simd-0084 提案中, 开发团队提出移除周期性租金收取机制. 根据 solana stack exchange 上的讨论, 这一变更于 2023 年 12 月 6 日前部署完成.

因此, solana 已经彻底移除了对租金收取的实际处理逻辑. 也就是说: 租金仍然作为一个概念存在于 solana 网络中, 但系统已经不会再周期性地收取租金或因为账户余额不足而删除账户. 实际上, 现在所有账户都相当于租赁豁免, 只要创建出来了就不会被删.

> 事实上, 我花了很多时间确认了这一修改的真实性. 请注意, solana 官方文档以及他人的博客文章可能仍包含部分过时信息, 建议结合上述社区讨论和提案确认最新状态.

4.9.3 租赁周期

从技术上讲, 每个账户都会记录自己的上次租金计费时间和余额. Solana 每个 epoch(大概两天)会进行一次租金结算. 如果某个账户没达到豁免门槛, 长期没有被访问, 且余额又不足以继续支付租金, 那它就可能被标记为"可清除", 节点会对账户数据进行清理, 释放状态空间.

如果您是程序开发者, 一般只要给数据账户设置成"租赁豁免", 后续就不用管它了.

> 请注意, 周期性租金收取的实际处理逻辑已经从 solana 网络里彻底移除.

4.9.4 租赁豁免资金计算

账户初始化时就要考虑豁免金额. 您需要提前估算账户空间大小, 并把足够的 sol 存进去. 租赁费率由 solana 网络实时计算, 您可以通过 get_minimum_balance_for_rent_exemption rpc 接口来获取当前存储指定大小的数据时需要的租赁豁免资金数量.

例: 如果小明要在账户中存储 100 字节的数据, 他需要的租赁豁免资金是多少?

答: 当前需要 1586880 lamport. 注意这个值可能会随着时间而发生变动.

```
import pxsol

print(pxsol.rpc.get_minimum_balance_for_rent_exemption(100, {}))
# 1586880
```

4.9.5 开发者要注意的细节

Solana 的账户租赁机制是为了控制状态存储成本, 让链上的数据有代价的存储. 同时, 为了开发方便, 又引入了租赁豁免机制, 只要您存够了钱, 就可以让账户永久保留.

对开发者来说, 最重要的是在初始化账户时正确设置余额, 以达成租赁豁免.

4.10 Solana/账户模型/尚未深入探讨的问题

Solana 的账户模型为性能和可组合性做出了大胆设计, 它不像以太坊那样以合约为中心, 而是以账户为核心, 将代码, 状态, 权限等元素彻底解耦. 通过本章内容, 我们已经了解了 solana 中各种账户的结构, 类型, 使用方式, 以及如何通过程序派生地址实现权限控制等机制.

然而, 即便掌握了这些知识, 在真实的开发中, 仍有不少细节和边界问题, 值得进一步探索.

4.10.1 尚未深入探讨的问题

- 程序账户能否主动修改数据账户的数据? Solana 的运行模型并不允许程序主动操作任意账户, 而是依赖调用者明确传入相关账户. 也就是说, 即使你拥有某个账户的所有权, 如果这个账户没有在指令中作为 writable 传入, 也无法修改它的状态. 这对新开发者来说是一个常见误解, 也是调试失败的重要原因.
- 复杂数据结构的布局与序列化约定. 数据账户内允许存放任意数据, 但如何高效, 安全地布局这些数据, 特别是变长结构与嵌套结构, 是一种工程技巧. 错误的布局可能导致租金过高, 数据解析出错, 甚至程序逻辑脆弱.
- 账户之间的组合调用与跨程序协作. 真实应用中, 常常需要多个账户协同工作. 如何构造模块化的账户结构, 处理跨账户调用中的权限传递, 是构建可扩展程序框架的关键.
- 并发访问下的账户锁定与失败处理机制. 虽然 solana 支持并行执行, 但开发者仍需自己管理账户之间的冲突, 特别是在热账户(如高频交易账户)上. 理解并发失败的原因, 合理设计账户划分策略, 是提升程序吞吐的前提.

4.10.2 下一步: 深入智能合约, 理解账户模型

在接下来的学习或开发中, 我们将尝试挑战一些新的任务, 通过这些新的任务来更加深入的理解 solana 的账户模型.

Solana 的账户模型, 不仅仅是一种存储机制, 更是一种控制执行, 构建权限边界, 提升性能的系统设计哲学.

账户是你与程序之间的协议, 也是你与系统之间的契约.

理解账户, 不只是为了写代码, 更是为了驾驭一套为高性能设计而生的思维方式.

5. Solana/程序开发入门

5.1 Solana/程序开发入门/导言

学生: "老师, 我刚刚读了那篇**Ada 和泰铢币的烦恼**的小故事, 感觉超有趣! 原来在 solana 上写个泰铢币程序也能遇到地址管理这么现实的问题啊."

老师: "哈, 那可是很多 solana 初学者的第一道关卡. 程序能跑不稀奇, 数据怎么管理才是关键."

学生: "我特别好奇她是怎么用 pda 解决问题的. 我以前学过以太坊, 我可以直接在账户里存储任意 k/v 数据, 举例来说, 我可以设计 key 为一个账户地址, value 为一个数字, 这样就能记录下谁有多少钱. 但在 solana, 好像得自己写代码处理这一切?"

老师: "没错. Solana 的智能合约不会自动帮你记谁有多少钱, 你得自己设计数据账户, 定义数据结构和权限."

学生: "感觉这就像搭积木, 但又要自己画图纸. 那泰铢币程序也是完全自己实现余额以及转账逻辑的?"

老师: "是的, ada 给每个用户分配一个 pda 作为存储他们泰铢余额的账户地址, 这样就不需要手动告诉别人钱在哪儿. 这一招在 solana 上非常常见."

学生: "我想自己试着写一个! 不一定是泰铢币, 也许我来搞个泡泡币? 每次互动就冒一个泡泡那种……"

老师(笑): "有创意! 那我们这章就从最基础的 solana 程序写法讲起. 先带你从如何将数据存储上链开始, 然后慢慢建出你自己的泡泡宇宙."

学生: "好耶!"

5.2 Solana/程序开发入门/搭建 Rust 开发环境

在区块链的世界里, solana 一直以快和高并发闻名. 如果您有点 web3 的背景, 大概也听说过它的一些不同: 它不用 evm, 不用 solidity, 合约用 rust 写, 部署成 bpf 字节码.

> 我们已经有太多 evm 项目了!

刚听上去可能有点劝退, 但其实一旦上手, 您会发现 solana 合约系统非常清爽, 性能极高, 而且资源管理也非常工程化和模板化. 只要您成功编写了第一个能正常运行的程序, 您几乎就能编写 solana 生态系统里的所有种类的程序.

本篇文章就带你一步步搭建 solana 智能合约的开发环境, 为后续学习和动手实验做好准备.

5.2.1 为什么用 Rust 写 Solana 合约

Solana 的运行时不是基于 evm, 而是基于 berkeley packet filter(bpf), 它是一种经过验证的适合在内核或沙盒中运行的字节码格式.

Berkeley packet filter, 诞生于上世纪 90 年代初(是的, 它的年纪比许多年轻黑客还大), 最初是为 unix 系统上的网络数据包捕获而生的. 简单来说, 它是一种高效的网络数据包过滤机制, 让内核能够决定哪些数据包该被扔掉, 哪些该传给用户态程序, 好比门卫大叔判断谁能进小区. 它的核心就是一个小巧玲珑的虚拟机, 这台虚拟机能运行一套简单的指令集(和 x86, arm 这些庞然大物比, bpf 的指令集简直像个袖珍机器人). 之后的时间里, 开发者们渐渐发现可以依赖 bpf 的指令集做更多事情, 于是 bpf 演化成了一个超级沙盒执行环境.

Rust 是目前社区最成熟的用来编写 solana bpf 合约的语言, 原因有几个:

- Rust 编译器对内存和类型检查非常严格, 能减少低级错误.
- Rust 的性能接近 c/c++, 但更安全.
- 最重要的: solana 官方 sdk 用 rust 编写, 生态完善.

所以, 不管你是 web 开发者, 链上合约老手还是系统程序员, 只要你想写 solana 合约, rust 是唯一且最佳选择.

5.2.2 安装 Rust 工具链

如果你已经是 rust 用户, 可以跳过这一步. 否则, 前往 rust 安装页面 根据提示安装 rust. 对于 linux, macos 以及 windows 上的 linux 子系统而言, 其安装命令都是一样的:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

安装后, 运行:

```
$ rustup --version
$ cargo --version
```

确认一切就绪.

5.2.3 Anchor 开发框架(可选但目前不推荐)

Anchor 是 solana 合约的一个开发框架. 它有一些优点, 例如自带合约模板, 自动处理 pda 账户权限, 支持集成测试等. 但是对于新手来说, 过早使用框架进行合约开发, 可能无法真正掌握 solana 合约的底层逻辑.

鉴于多数 solana 合约开发教程或文档, 均习惯基于框架进行教学, 因此我特地在这里告诉您, 本章节后续的内容不会依赖 anchor 框架, 您现在也没有必要安装 anchor.

5.3 Solana/程序开发入门/一个允许用户自由存储数据的链上程序

在 solana 上写程序, 可不只是写代码这么简单. 您需要理解账户模型, 租赁机制, 权限控制, 还有数据存储的经济成本. 这套项目教程, 正是为了带您一步步理解并实现一个真实又不复杂的链上应用.

我们要做的, 是一个链上数据存储器.

5.3.1 背景动机

假设您是某个去中心化应用或工具平台的用户, 您希望能把一段数据(比如一份文档, 一个配置文件, 一份合同或者一个游戏存档)存在链上.

1. 您希望数据是您的, 不能被别人覆盖.
2. 您希望以后可以随时更新这份数据.
3. 您不希望因为数据长度变化, 导致程序崩掉或账户被清除.
4. 您也不想付出不必要的存储费.

这时候, 我们就可以做一个用户托管数据仓库, 每个用户的数据单独保存在自己的一个 pda 中, 按需付费, 灵活更新.

5.3.2 项目目标

我们要实现的是一个支持任意用户创建, 更新, 扩容和缩容数据账户的 solana 程序, 具体包括两个功能:

1. 用户可以初始化自己的数据账户. 程序会为用户创建一个 pda 作为数据存储账户; 用户可以指定初始内容, 系统会根据长度自动分配所需的存储空间; 数据账户的所有 lamport 和权限归用户所有, 程序只负责校验并初始化账户.
2. 用户可以随时更新数据内容. 用户可以把数据换成新的内容; 如果新数据更长, 程序会检查用户是否附带了足够的 lamport 以保持租赁豁免; 如果新数据更短, 程序允许把多余的 lamport 从数据账户中提取出来, 退还给用户.

5.3.3 技术细节

程序根据用户主钱包地址生成 pda 账户, 保证每个用户有且仅有一个对应的数据账户. 整个 pda 账户的 data 区域用于存储用户输入的原始字节流. 同时需要处理 solana 的租赁机制, 程序使用系统程序接口, 计算给定长度数据所需的最小租金, 检查当前 pda 账户的余额是否满足租赁豁免要求, 如果不够, 用户必须补 lamport; 如果超额, 允许用户提取差额. 最终, 程序将把用户提交的任意数据写入 pda 账户的 data 字段里.

5.3.4 小结

这个程序看似简单, 但它涉及了 solana 开发中几乎所有关键知识点: pda 账户, 账户创建, lamport 管理, 系统调用, 权限校验, 动态数据处理…… 是一个非常理想的实战练习项目.

如果您已经安装好开发环境, 让我们从下一章开始, 一步步把这个链上数据存储器构建起来!

5.4 Solana/程序开发入门/搭建初始目录结构

这篇文章, 我们来搭建一个干净的纯 rust 的 solana 程序项目目录结构, 作为我们的链上数据存储器项目的初始目录结构.

5.4.1 新建空白项目

我们先创建一个普通的 rust 库项目:

```
$ cargo new --lib pxsol-ss
$ cd pxsol-ss
```

这个项目会包含我们的主程序逻辑, 也就是将要部署到链上的合约. 项目名称 `pxsol-ss` 是 `pxsol-simple-storage` 的缩写.

5.4.2 配置编译目标

编辑 Cargo.toml, 添加如下设置:

```
[lib]
crate-type = ["cdylib", "lib"]
```

这个是告诉 cargo 我们要生成哪几种类型的 crate.

其中 `cdylib` 表示编译为一个 c 兼容的动态库. Solana 要求合约以 cdylib 形式编译为 `.so` 文件, 才能部署到链上. 这个 `.so` 文件会通过 `cargo build-bpf` 生成.

另外 `lib` 表示我们还希望编译成普通的 rust 库, 即 `.rlib`. 这有助于在本地测试时, 把合约逻辑当作普通 rust 模块来调用, 也方便写单元测试或集成测试.

总结的说: 您需要 cdylib 来部署, lib 来开发和测试.

5.4.3 添加依赖

我们需要引入 solana 的核心 sdk:

```
[dependencies]
solana-program = "1.18.0"
```

5.4.4 项目目录结构参考

最终的目录结构看起来可能像这样:

```
pxsol-ss/
├── Cargo.toml
└── src/
    └── lib.rs
```

其中 Cargo.toml 中的内容为

```
[package]
name = "pxsol-ss"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib", "lib"]
```

```
[dependencies]
solana-program = "1.18"
```

5.4.5 创建 lib.rs 骨架

在 `src/lib.rs` 中, 先写一个最简单的入口:

```
solana_program::entrypoint!(process_instruction);

pub fn process_instruction(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    solana_program::msg!("Hello Solana!");
    Ok(())
}
```

这个程序目前什么也不做, 只会在调用时打印一句话. 但它已经可以被编译成 solana 支持的 bpf 程序了.

5.4.6 尝试编译

运行下面命令, 进行交叉编译:

```
$ cargo build-sbf -- -Znext-lockfile-bump
```

如果一切正常, 你会在 `target/deploy/` 目录下看到 `pxsol_ss.so` 文件, 这就是可以部署到 solana 的程序文件.

注意参数 `-Znext-lockfile-bump` 是一个临时参数, 因为 solana v1.18 依赖于 rustc 1.75, 如果您本地的 rust 版本大于 1.75, 存在一些兼容性问题, 因此需要传入该参数. 当您阅读本书时, 随着版本的更新, 此兼容问题很可能已经修复, 因此您也许可以尝试看看不加入该临时参数. 关于该问题的详细解释, 可以参考该 github 页面.

5.4.7 小结

到这里, 我们已经完成了一个最基本的, 使用原生 rust 编写的 solana 程序框架搭建. 这是实现我们用户自托管链上数据存储器的第一步.

下一步我们将实现账户派生和数据存储的逻辑, 开始真正和 solana 的账户模型打交道.

5.5 Solana/程序开发入门/入口函数解释

Solana 的每一笔交易中都包含一个或多个指令. 一个指令是对链上某个程序账户的调用, 包含三部分, 源码:

```
pub struct Instruction {
    /// Pubkey of the program that executes this instruction.
    pub program_id: Pubkey,
    /// Metadata describing accounts that should be passed to the program.
    pub accounts: Vec<AccountMeta>,
    /// Opaque data passed to the program for its own interpretation.
    pub data: Vec<u8>,
}
```

所以, 当一个指令被执行时, 它就会被送给您写的程序的入口函数 `process_instruction()` .

5.5.1 程序入口函数签名

```
solana_program::entrypoint!(process_instruction);

pub fn process_instruction(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    solana_program::msg!("Hello Solana!");
    Ok(())
}
```

我们一一解释.

参数 `program_id: &solana_program::pubkey::Pubkey` 是当前这个合约(程序账户)本身的地址. 在链上, 每个部署的程序都有一个账户地址(公钥). 当交易指令调用这个程序时, solana 会把它写进 `program_id` 字段中. 您可以用它来做校验, 比如检查某个账户是否是由这个程序创建的 pda. 比如:

```
let expected_pda = solana_program::pubkey::Pubkey::create_program_address(&[seed], program_id)?;
```

参数 `accounts: &[solana_program::account_info::AccountInfo]` 是指令中涉及的账户列表, 对应 `Instruction.accounts` 里的 `Vec<AccountMeta>` 项. 这些账户由调用方指定, 并且程序不能随便添加账户, 只能用这些已传入的账户.

每个 AccountInfo 包含:

1. 账户地址 (key)
2. 是否是签名者 (is_signer)
3. 是否是可写账户 (is_writable)
4. Lamports 余额 (lamports)
5. 数据 (data)
6. 所属程序 (owner)
7. ...

程序通常需要自己根据账户位置索引来解读这些账户. 比如:

```
let account_user = &accounts[0];
let account_user_pda = &accounts[1];
```

注意账户的顺序非常重要, 您必须和调用方传入的顺序一一对应.

参数 `data: &[u8]` 是调用方自定义的指令数据. 通常我们会自己设计一个结构, 然后用 borsh, serde 等序列化方案或手动解析来从字节数组中提取内容. 您可以把它理解为"程序要执行什么操作 + 参数", 类似函数调用时传参.

这个模式和其他链, 比如以太坊的函数调用, 很不一样, solana 追求高性能, 所以要求调用方把所有要用的账户和数据一次性传进来, 程序自己不会查链, 也不扫账户, 而是基于这些参数做纯函数式的运算.

5.6 Solana/程序开发入门/创建数据账户并使其达成租赁豁免

我们开始实现链上数据存储器的第一个功能. 用户首次使用该存储器, 并尝试写入数据到自己的专属数据账户中时. 我们要完成以下几个功能:

1. 用户第一次上传时, 程序会帮他创建一个 pda 数据账户.
2. 上传的数据长度可以自定义.
3. 创建的账户会自动达成租赁豁免, 避免日后被清理.

5.6.1 涉及到的账户

在编写具体功能之前, 我们先思考一下该功能会涉及到哪些账户.

1. 用户的普通钱包账户. 创建 pda 数据账户需要使用到用户的普通钱包账户作为种子, 同时需要用户的普通钱包账户提供 lamport 以达成数据账户租赁豁免. 该账户的权限应当是可写, 需要签名.
2. 用户新生成的数据账户. 我们会新建一个 pda 数据账户并在此账户中写入数据. 该账户的权限应当是可写, 无需签名.
3. 系统账户. 只有系统账户才能创建新的账户. 该账户的权限应当是只读, 无需签名.
4. Sysvar rent 账户. Solana 通过 sysvar 帐户向程序公开各种集群状态数据. 在本例子中, 我们需要知道需要多少 lamport 才能使数据账户达成租赁豁免, 而这个数字可能由集群动态改变. 因此需要访问 sysvar rent 账户. 该账户的权限应当是只读, 无需签名. 您可以访问此页面 了解更多关于 sysvar 账户的信息.

总结账户列表如下:

账户索引	地址	需要签名	可写	权限(0-3)	角色
0	...	是	是	3	用户的普通钱包账户
1	...	否	是	1	用户的数据账户
2	`1111111111...`	否	否	0	System
3	`SysvarRent...`	否	否	0	Sysvar rent

从入口函数 process_instruction 的 accounts 参数中获取各个账户信息代码如下:

```
let accounts_iter = &mut accounts.iter();
let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
let account_data = solana_program::account_info::next_account_info(accounts_iter)?;
let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program system
let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program sysvar rent
```

5.6.2 计算租赁豁免

Solana 提供了一个函数可以查询系统规定的租赁豁免门槛:

```
let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(data.len());
```

参数 `data.len()` 是您准备在 pda 账户中存储的字节数, 返回值 rent_exemption 是为租赁豁免所需的 lamport 数量.

5.6.3 派生 PDA 数据账户地址

您需要使用 `solana_program::pubkey::Pubkey::find_program_address` 来获取 pda 账户地址以及其 bump 值. 在本示例中, 我们只需要使用到 bump 的值.

```
let bump_seed = solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).1;
```

5.6.4 判断 PDA 是否已经存在

Solana 的 sdk 里并没有直接提供可供判断一个账户是否存在的函数, 因此我们使用以下方式来进行判断. 这行代码的依据是任何存在的账户都必须达成租赁豁免, 因此存在的账户的余额必不可能为零.

```
if **account_data.try_borrow_lamports().unwrap() == 0 {
    // Data account is not initialized.
}
```

5.6.5 创建 PDA 账户

您需要使用系统程序 `solana_program::system_instruction::create_account` 创建账户.

```
solana_program::system_instruction::create_account(
    account_user.key,
    account_data.key,
    rent_exemption,
    data.len() as u64,
    program_id,
)
```

由于 pda 没有私钥, 不能自己签名, 所以要用程序的签名种子进行签名.

```
solana_program::program::invoke_signed(
    &solana_program::system_instruction::create_account(
        account_user.key,
        account_data.key,
        rent_exemption,
        data.len() as u64,
        program_id,
    ),
    accounts,
    &[&[&account_user.key.to_bytes(), &[bump_seed]]],
)?;
```

Solana rust sdk 中有一个与 `invoke_signed()` 函数非常相似的 `invoke()` 函数, 它们的作用都是用于执行一个指令, 但是功能上存在细微的差异. 在这个例子中, 我们要操作的账户是 pda, 也就是说这个账户没有私钥, 不能真正签名, 但您(程序)作为它的所有者, 有权代表它执行操作. 这个时候就不能用普通的 `invoke()`, 而是要用 `invoke_signed()`, 让 solana 系统知道: "这个账户虽然没有签名, 但我是它的创建者, 我现在代表它签名了".

完成! 您现在拥有了一个租赁豁免的 pda 数据账户.

5.6.6 写入数据

最后, 我们向数据账户写入数据. 很简单, 对吧?

```
account_data.data.borrow_mut().copy_from_slice(data);
```

5.6.7 完整代码

```
#![allow(unexpected_cfgs)]

use solana_program::sysvar::Sysvar;

solana_program::entrypoint!(process_instruction);

pub fn process_instruction(
```

```
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    let accounts_iter = &mut accounts.iter();
    let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_data = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program system
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program sysvar rent

    let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(data.len());
    let bump_seed = solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).1;

    // Data account is not initialized. Create an account and write data into it.
    if **account_data.try_borrow_lamports().unwrap() == 0 {
        solana_program::program::invoke_signed(
            &solana_program::system_instruction::create_account(
                account_user.key,
                account_data.key,
                rent_exemption,
                data.len() as u64,
                program_id,
            ),
            accounts,
            &[&[&account_user.key.to_bytes(), &[bump_seed]]],
        )?;
        account_data.data.borrow_mut().copy_from_slice(data);
        return Ok(());
    }
    Ok(())
}
```

5.7 Solana/程序开发入门/数据账户内容更新及动态租赁调节

Solana 的账户存储需要租金, 数据越长, 租金越贵. 如果数据更新后变长了, 数据账户租金不够, 账户就不再租赁豁免; 如果数据更新后变短了, 那用户其实多交了租金, 程序可以把多余的部分还给用户!

本篇文章就来教您如何实现动态租赁调节.

5.7.1 更新数据账户内容

链上账户可以使用 `.data.borrow_mut()` 来更新内容, 但大小不能变, 所以通常需要重新创建或使用 `.realloc()` 重分配数据账户空间.

```
// Realloc space.
account_data.realloc(data.len(), false)?;
// Overwrite old data with new data.
account_data.data.borrow_mut().copy_from_slice(data);
```

5.7.2 租金补足

如果新数据比老数据更长, 您需要使用系统程序 `solana_program::system_instruction::transfer` 为 pda 账户添加更多租金.

```
// Fund the data account to let it rent exemption.
if rent_exemption > account_data.lamports() {
    solana_program::program::invoke(
        &solana_program::system_instruction::transfer(
            account_user.key,
            account_data.key,
            rent_exemption - account_data.lamports(),
        ),
        accounts,
    )?;
}
```

5.7.3 租金退款

如果新数据比老数据更短, 则从 pda 账户中获取退款. 获取退款不需要执行系统程序, 您只需要修改两个账户中的余额即可.

```
// Withdraw excess funds and return them to users. Since the funds in the pda account belong to the program, we do
// not need to use instructions to transfer them here.
if rent_exemption < account_data.lamports() {
    **account_user.lamports.borrow_mut() = account_user.lamports() + account_data.lamports() - rent_exemption;
    **account_data.lamports.borrow_mut() = rent_exemption;
}
```

您可能好奇, 为什么这个时候不需要使用 `solana_program::system_instruction::transfer`? 问题的答案在于权限. 您是否还记得每个数据账户都拥有所有者程序? 所谓的所有者程序, 就是能自由操控数据账户而不需要额外的权限.

- 租金补足过程中, 程序转移了您的钱包账户的资金, 因此必须得到您的授权.
- 租金退款过程中, 程序转移了自己控制的数据账户的资金, 因此无需您授权.

5.8 Solana/程序开发入门/完整链上代码

在本小节中, 我们给出完整链上数据存储器的代码.

```
#![allow(unexpected_cfgs)]

use solana_program::sysvar::Sysvar;

solana_program::entrypoint!(process_instruction);

pub fn process_instruction(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    let accounts_iter = &mut accounts.iter();
    let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_data = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program system
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program sysvar rent

    let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(data.len());
    let bump_seed = solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).1;

    // Data account is not initialized. Create an account and write data into it.
    if **account_data.try_borrow_lamports().unwrap() == 0 {
        solana_program::program::invoke_signed(
            &solana_program::system_instruction::create_account(
                account_user.key,
                account_data.key,
                rent_exemption,
                data.len() as u64,
                program_id,
            ),
            accounts,
            &[&[&account_user.key.to_bytes(), &[bump_seed]]],
        )?;
        account_data.data.borrow_mut().copy_from_slice(data);
        return Ok(());
    }

    // Fund the data account to let it rent exemption.
    if rent_exemption > account_data.lamports() {
        solana_program::program::invoke(
            &solana_program::system_instruction::transfer(
                account_user.key,
                account_data.key,
                rent_exemption - account_data.lamports(),
            ),
            accounts,
        )?;
    }
    // Withdraw excess funds and return them to users. Since the funds in the pda account belong to the program, we do
    // not need to use instructions to transfer them here.
    if rent_exemption < account_data.lamports() {
        **account_user.lamports.borrow_mut() = account_user.lamports() + account_data.lamports() - rent_exemption;
        **account_data.lamports.borrow_mut() = rent_exemption;
    }
    // Realloc space.
```

```
    account_data.realloc(data.len(), false)?;
    // Overwrite old data with new data.
    account_data.data.borrow_mut().copy_from_slice(data);

    Ok(())
}
```

5.9 Solana/程序开发入门/编译并部署程序

5.9.1 编译

使用下面的命令编译程序代码.

```
$ cargo build-sbf -- -Znext-lockfile-bump
```

5.9.2 部署程序

使用下面的 python 代码部署目标程序上链:

```
import pathlib
import pxsol

# Enable log
pxsol.config.current.log = 1

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))

program_data = pathlib.Path('target/deploy/pxsol_ss.so').read_bytes()
program_pubkey = ada.program_deploy(bytearray(program_data))
print(program_pubkey) # DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E
```

在部署过程中, 您会在日志中看到大量的交易. 在 solana 上部署一个程序和其他区块链(如以太坊)相比, 流程略有不同. 部署过程分为多个步骤, 每一步基本上对应一个或多个交易:

1. 创建一个程序账户.
2. 分段上传程序代码(分片写入). Solana 的单笔交易大小有限, 一个交易序列化后最多不超过 1232 字节, 而你的程序代码可能有几万字节或更多. 所以必须把 bpf 字节码分片后, 分多次交易写入账户的数据区.
3. 所有字节都写完之后, 需要最后一步: 调用 bpf loader 程序的 finalize 方法, 把账户标记为 finalized. 从这个时候开始, 它才会变成一个真正的 solana 程序了.

虽然流程复杂一些, 但这是高性能设计的一部分.

5.10 Solana/程序开发入门/程序交互

现在, 我们的链上数据存储器已经被部署在 `DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E` 地址上. 我们尝试写入自己的数据到程序里.

5.10.1 写入数据到账户

写数据的过程是通过一个 solana 交易来完成的. 您可以这样写入数据:

```
import base64
import pxsol

pxsol.config.current.log = 1

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))


def save(user: pxsol.wallet.Wallet, data: bytearray) -> None:
    prog_pubkey = pxsol.core.PubKey.base58_decode('DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E')
    data_pubkey = prog_pubkey.derive_pda(user.pubkey.p)
    rq = pxsol.core.Requisition(prog_pubkey, [], bytearray())
    rq.account.append(pxsol.core.AccountMeta(user.pubkey, 3))
    rq.account.append(pxsol.core.AccountMeta(data_pubkey, 1))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.SysvarRent.pubkey, 0))
    rq.data = data
    tx = pxsol.core.Transaction.requisition_decode(user.pubkey, [rq])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([user.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
    r = pxsol.rpc.get_transaction(txid, {})
    for e in r['meta']['logMessages']:
        print(e)

if __name__ == '__main__':
    save(ada, b'The quick brown fox jumps over the lazy dog')
```

```
# 2025/05/27 10:17:23 pxsol: transaction send signature=oCF2esfLeM7iu8MsR5wgBPatVXGt9Dq7TSzLpwWuMjooeDBeHMtSc8ukuqmPcaMrzz
# 2025/05/27 10:17:23 pxsol: transaction wait unconfirmed=1
# 2025/05/27 10:17:23 pxsol: transaction wait unconfirmed=0
# Program DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E invoke [1]
# Program DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E consumed 2903 of 200000 compute units
# Program DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E success
```

5.10.2 读取链上的数据

读取数据就是查询自己的 pda 数据账户中存储数据的过程. 这个过程不需要构造交易, 只需要借助 rpc 接口查询即可.

```
import base64
import pxsol

pxsol.config.current.log = 1

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))


def load(user: pxsol.wallet.Wallet) -> bytearray:
    prog_pubkey = pxsol.core.PubKey.base58_decode('DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E')
    data_pubkey = prog_pubkey.derive_pda(user.pubkey.p)
```

```
    info = pxsol.rpc.get_account_info(data_pubkey.base58(), {})
    return base64.b64decode(info['data'][0])


if __name__ == '__main__':
    print(load(ada).decode()) # The quick brown fox jumps over the lazy dog
```

5.10.3 更新链上的数据

我们只需要重新调用一次 `save()` 并写入不同的数据, 就能覆盖链上已有的数据, 程序会为我们自动实现新的租赁豁免. 完整代码如下.

```
import base64
import pxsol

pxsol.config.current.log = 1

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))


def save(user: pxsol.wallet.Wallet, data: bytearray) -> None:
    prog_pubkey = pxsol.core.PubKey.base58_decode('DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E')
    data_pubkey = prog_pubkey.derive_pda(user.pubkey.p)
    rq = pxsol.core.Requisition(prog_pubkey, [], bytearray())
    rq.account.append(pxsol.core.AccountMeta(user.pubkey, 3))
    rq.account.append(pxsol.core.AccountMeta(data_pubkey, 1))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.SysvarRent.pubkey, 0))
    rq.data = data
    tx = pxsol.core.Transaction.requisition_decode(user.pubkey, [rq])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([user.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
    r = pxsol.rpc.get_transaction(txid, {})
    for e in r['meta']['logMessages']:
        print(e)


def load(user: pxsol.wallet.Wallet) -> bytearray:
    prog_pubkey = pxsol.core.PubKey.base58_decode('DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E')
    data_pubkey = prog_pubkey.derive_pda(user.pubkey.p)
    info = pxsol.rpc.get_account_info(data_pubkey.base58(), {})
    return base64.b64decode(info['data'][0])


if __name__ == '__main__':
    save(ada, b'The quick brown fox jumps over the lazy dog')
    print(load(ada).decode()) # The quick brown fox jumps over the lazy dog
    save(ada, '片云天共远, 永夜月同孤.'.encode())
    print(load(ada).decode()) # 片云天共远, 永夜月同孤.
```

5.11 Solana/程序开发入门/升级程序

在 solana 中, 我们可以替换已部署的程序的代码. 不过要注意, solana 程序的部署和更新存在几个不同的版本演进, 在早期的版本里, 程序是不可升级的.

5.11.1 如何升级程序

升级程序可使用 `pxsol.wallet.Wallet.program_update()` 接口完成. 您可以修改我们的链上数据存储器代码, 然后使用如下的命令来升级链上程序版本. 升级程序不会修改程序的地址.

```
import pathlib
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))
program_pubkey = pxsol.core.PubKey.base58_decode('DVapU9kvtjzFdH3sRd3VDCXjZVkwBR6Cxosx36A5sK5E')
program_data = pathlib.Path('target/deploy/pxsol_ss.so').read_bytes()
ada.program_update(program_pubkey, program_data)
```

5.11.2 Solana 程序最早是不可更改的

Solana 早期负责**管理程序的程序**叫做 bpf loader, 也叫做 `v1`. 使用它部署的程序, 您的程序会被标记为所有数据不可写, 所有代码不可修改, 同时程序账户的 executable 被设置为 true. 所以这种部署方式下, 程序不能更新! 如果您想升级, 就只能部署一个新的程序, 并更新所有依赖它的账户或前端逻辑, 这无疑非常麻烦.

5.11.3 演进史

Solana 的 bpf loader 是一个特殊的原生程序, 专门用来加载并执行您上传的 bpf 字节码. Solana 发展过程中经历了三个主要版本的 loader:

Loader 名称	地址	特点
BPFLoader (v1)	`BPFLoader1111111111111111111111111111111111`	最早期版本, 部署后不可升级
BPFLoader2 (v2)	`BPFLoader2111111111111111111111111111111111`	支持更高效的加载, 但依旧不可升级
BPFLoaderUpgradeable (v3)	`BPFLoaderUpgradeab1e11111111111111111111111`	当前默认, 支持程序升级和权限控制

也就是从 v3 开始, solana 引入了可升级程序的概念, 并将其作为**默认部署**方式.

5.11.4 可升级程序的结构图

Pxsol 默认使用 v3 来部署程序. 这会自动创建两个账户:

账户类型	作用
Program account	主地址, 对外暴露的程序入口
Program data account	存储实际代码字节码, 可变

```
Program ID (e.g., MyProgram111...)
│
├──> Program Account
│      └── owner: BPFLoaderUpgradeable
│      └── executable: true
│      └── points to:
│
└──> ProgramData Account
       └── contains: .so 字节码
       └── contains: upgrade_authority pubkey
```

所谓的升级程序, 其实就是修改第二个账户的内容.

5.12 Solana/程序开发入门/获取完整源码

源码我已经打包好放上 github 啦!

如果你懒得跟着一步步敲代码(我懂你), 可以直接去看我准备好的示例项目. 地址在这儿, 不用谢我, 除非你想请我喝杯奶茶.

我知道许多开发者喜欢咖啡, 但对于我而言, 奶茶总是最好的.

```
$ git clone https://github.com/mohanson/pxsol-ss
$ cd pxsol-ss
```

```
$ python make.py deploy
# 2025/05/20 16:06:38 main: deploy program pubkey="T6vZUAQyiFfX6968XoJVmXxpbZwtnKfQbNNBYrcxkcp"
```

注意到程序地址会被保存在 `res/info.json` 中, 后续操作会直接从此文件获取程序地址.

```
# Save some data.
$ python make.py save "The quick brown fox jumps over the lazy dog"

# Load data.
$ python make.py load
# The quick brown fox jumps over the lazy dog.

# Save some data and overwrite the old data.
$ python make.py save "片云天共远, 永夜月同孤."
# Load data.
$ python make.py load
# 片云天共远, 永夜月同孤.
```

6. Solana/泰铢币

6.1 Solana/泰铢币/导言

学生: "教授, 我最近在想, 我们之前写的简单的链上数据存储器, 是否能扩展成一个能执行转账的泰铢币程序? 似乎每个人只需要在自己的数据账户中记录自己的余额就可以了, 对吧?"

老师: "哈哈, 你已经走到一个非常关键的阶段了. 其实, 任何一个链上程序, 本质上都是一个状态机. 你想实现什么功能, 仅取决于你怎么去解释数据."

学生: "对啊, 我认为, 只需要程序给每个用户创建一个数据账户, 存他们自己的余额."

老师: "完全正确. 你可以继续想想, 泰铢币程序需要实现哪些指令?"

学生: "可以这么简单开始, 程序支持两个指令, 分别是铸造和转移. 前者增加代币总供应量, 后者则在两个账户之间转移代币."

老师: "别忘记了, 你还需要明确涉及的账户列表."

学生: "是的, 教授. 我想我对 solana 程序的设计有更深刻的认识了. 我们总是需要遵循**先设计数据格式, 然后设计指令以及最后明确账户列表**这三个步骤."

老师: "很棒! 你已经开始触类旁通了. 那么泰铢币程序就作为你这周的家庭作业了!"

学生: "太好了! 我这就开始画图纸, 然后一步步把它写出来."

6.2 Solana/泰铢币/进化之路

当我们在区块链世界中编写去中心化应用时, 往往都是从最简单的链上数据存储器起步. 大概在 8 年前, 我第一次接触到区块链世界, 我看到的第一个教程就是教学如何在以太坊上编写一个数据存储器. 如今, 我成为了一个新的教程编写者, 当我思考我应该选择哪个应用作为我的教学例子时, 我立即想到了它, 我必须承认, 这是一种开源精神的传承.

我很喜欢一句话: **算法 + 数据结构 = 程序**. 我认为即使是去中心化应用也遵循这个道理. 当您理解如何在链上存储任意数据后, 您就能通过调整算法来实现任意您想实现的程序.

> Algorithms + Data Structures = Programs 是 N. Wirth 老爷子的经典著作.

链上数据存储器的本质是用一个数据账户, 在链上存储用户自己的任意信息.

我们如果想把它发展成一个"泰铢币"程序, 只需要从数据格式, 指令交互, 账户管理上这三个方面做一些改变. 下面, 我们就从这些角度, 看看它是怎么从数据存储器一步步进化的.

6.2.1 账户模型: 从简单数据到余额账户

在最初的存储器中, 数据账户的结构很简单, 用户可以存储任意格式和长度的数据. 每个用户都有自己专属的数据账户, 合约只要校验 pda 地址和用户签名即可写入数据.

到了泰铢币程序, 我们就得让数据账户不仅仅是一个可以任意读写的个人空间, 而是真正的余额账户. 我们规定数据账户中只能存储一个 64 位无符号型整数, 且以大端序进行编码.

这样, 每个用户的数据账户就好像是在代币合约账本里的子账户, 明确记载了该用户拥有多少泰铢币.

6.2.2 两个指令: 铸造和转账

在链上数据存储器阶段, 程序只有一个存储或更新数据的指令. 现在我们需要基于这个指令, 开发出两个新的指令:

1. 铸造: 由铸造权限持有者(通常是合约部署者)发起, 为所有者铸造新的泰铢币: 也就是货币增发.
2. 转账: 用户 ada 转账泰铢币给用户 bob, 要求用户 ada 签名确认, 并更新双方的数据账户(余额账户).

这两个指令不仅要对余额账户读写, 还要进行基本的检查:

1. 铸造: 只能由授权账户发起.
2. 转账: 校验发送方余额是否足够, 并小心处理**整数溢出**问题.

设计两条指令的接收数据格式. 简单来说, 泰铢币程序只接收 9 个字节的数据, 第一个字节用于区分您是想铸造还是转账, 剩余的字节表示为铸造代币的数量或转账代币的数量.

1. 铸造: `0x00` + u64
2. 转账: `0x01` + u64

6.2.3 账户列表

每个指令都要明确声明它用到的账户(accounts 参数), 否则无法在 solana 运行. 需要额外注意的地方在于, 如果用户还不存在数据账户, 我们需要为他创建新的数据账户.

总结账户列表如下:

铸造

账户索引	地址	需要签名	可写	权限(0-3)	角色
0	...	是	是	3	铸造权限所有者的普通钱包账户
1	...	否	是	1	铸造权限所有者的数据账户
2	`1111111111...`	否	否	0	System
3	`SysvarRent...`	否	否	0	Sysvar rent

转账

账户索引	地址	需要签名	可写	权限(0-3)	角色
0	...	是	是	3	发送者的普通钱包账户
1	...	否	是	1	发送者的数据账户
2	...	否	否	0	接收者的普通钱包账户
3	...	否	是	1	接收者的数据账户
4	`1111111111...`	否	否	0	System
5	`SysvarRent...`	否	否	0	Sysvar rent

6.2.4 不是结束

跟以太坊的 erc20 不同, solana 的合约世界非常灵活. 我们需要管理铸造的权限. 在本教程中, 我们选择把铸造权限写死在合约里, 当然您也可以单独搞个"权限账户"来管理铸造权限.

您也可以随时添加别的功能, 比如销毁或者批量转账, 这些功能虽然不是很常用, 但对于某些场景至关重要, 例如您想批量空投代币到上百万个用户: 如果没有批量转账功能, 这花费的手续费以及时间很可能是您无法接受的.

从最初的链上数据存储器, 到一个真正的泰铢币程序, 关键在于:

- 数据结构的演化. 从简单的字节串演进到余额账户结构.
- 指令的演化. 从简单存储更新变成铸造和转账.
- 账户列表的演化.

世界由您来定义, 见证您的泰铢币的诞生!

6.3 Solana/泰铢币/核心机制实现

这篇文章介绍泰铢币的实现原理, 核心机制和背后的一些趣事点.

6.3.1 指令路由

泰铢币的合约主函数 `process_instruction()` , 像个小开关盒子:

- 当第一个字节是 `0x00` , 就执行铸造操作, ada 亲自印钞, 往自己的账户里塞钱.
- 当第一个字节是 `0x01` , 就执行两个账户之间的转账操作.

切换指令全靠这一个字节, 简单粗暴, 也非常有 solana 的狂野风格.

```
#![allow(unexpected_cfgs)]

use solana_program::sysvar::Sysvar;

solana_program::entrypoint!(process_instruction);

pub fn process_instruction_mint(
    _: &solana_program::pubkey::Pubkey,
    _: &[solana_program::account_info::AccountInfo],
    _: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    Ok(())
}

pub fn process_instruction_transfer(
    _: &solana_program::pubkey::Pubkey,
    _: &[solana_program::account_info::AccountInfo],
    _: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    Ok(())
}

pub fn process_instruction(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    assert!(data.len() >= 1);
    match data[0] {
        0x00 => process_instruction_mint(program_id, accounts, &data[1..]),
        0x01 => process_instruction_transfer(program_id, accounts, &data[1..]),
        _ => unreachable!(),
    }
}
```

6.3.2 创建数据账户

在每次转账或铸币之前, 合约都会检查目标 pda 数据账户有没有被初始化. 如果没有的话, 立刻用 `invoke_signed()` 调用 `solana_program::system_instruction::create_account()` 创建账户并帮 pda 数据账户交齐租金, 保证租赁豁免.

数据账户里写上 8 字节的 u64::MIN, 表示 0 泰铢余额.

这个自动开户逻辑非常贴心, 让用户转账时不用先自己去初始化自己的数据账户. 铸造指令与转账指令初始化 pda 数据账户代码如下:

```
pub fn process_instruction_mint(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    let accounts_iter = &mut accounts.iter();
    let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_user_pda = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program system
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program sysvar rent

    // Data account is not initialized. Create an account and write data into it.
    if **account_user_pda.try_borrow_lamports().unwrap() == 0 {
        let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(8);
        let bump_seed =
            solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).1;
        solana_program::program::invoke_signed(
            &solana_program::system_instruction::create_account(
                account_user.key,
                account_user_pda.key,
                rent_exemption,
                8,
                program_id,
            ),
            accounts,
            &[&[&account_user.key.to_bytes(), &[bump_seed]]],
        )?;
        account_user_pda.data.borrow_mut().copy_from_slice(&u64::MIN.to_be_bytes());
    }
}
```

```
pub fn process_instruction_transfer(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    let accounts_iter = &mut accounts.iter();
    let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_user_pda = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_into = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_into_pda = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program system
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program sysvar rent

    // Data account is not initialized. Create an account and write data into it.
    if **account_into_pda.try_borrow_lamports().unwrap() == 0 {
        let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(8);
        let bump_seed =
            solana_program::pubkey::Pubkey::find_program_address(&[&account_into.key.to_bytes()], program_id).1;
        solana_program::program::invoke_signed(
            &solana_program::system_instruction::create_account(
                account_user.key,
                account_into_pda.key,
                rent_exemption,
                8,
                program_id,
            ),
```

```
            accounts,
            &[&[&account_into.key.to_bytes(), &[bump_seed]]],
        )?;
        account_into_pda.data.borrow_mut().copy_from_slice(&u64::MIN.to_be_bytes());
    }
}
```

6.3.3 只有 Ada 能印钱

别以为谁都能在 ada 的世界里印泰铢币! 在铸造操作的开头, 我们来一段硬性校验:

```
assert_eq!(*account_user.key, solana_program::pubkey!("6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt"));
```

只能 ada 本人签名, 才能铸币. 别想偷懒, 别想作弊, 防止通胀从根本做起(注: 此限制对 ada 无效)!

铸造流程也很简单, 首先读取 ada 的余额, 之后交易 data 参数里取出要铸造的金额, 两数相加, 写回 pda 数据账户. 在这个例子里, 数字以大端序存储.

```
// Mint.
let mut buf = [0u8; 8];
buf.copy_from_slice(&account_user_pda.data.borrow());
let old = u64::from_be_bytes(buf);
buf.copy_from_slice(&data);
let inc = u64::from_be_bytes(buf);
let new = old.checked_add(inc).unwrap();
account_user_pda.data.borrow_mut().copy_from_slice(&new.to_be_bytes());
```

6.3.4 转账指令

对于转账操作的话, 先把收款方的 pda 账户初始化好(如果还没开过户), 之后读取发送方和接收方 pda 数据账户里的余额, 接着从交易 data 里取出转账金额, 双方余额做加减, 最后写回各自的 pda 数据账户.

要注意的是, 转账操作时必须验证发送人的 pda 账户确实属于发送人, 防止让他人扣了您的钱!

```
let account_need_pda =
        solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).0;
assert_eq!(account_user_pda.key, &account_need_pda);
```

Rust 的 `.checked_sub()` 和 `.checked_add()` 有溢出检测, 可以防止你搞个负数变成链上亿万富翁. 转账流程如下:

```
// Transfer.
let mut buf = [0u8; 8];
buf.copy_from_slice(&account_user_pda.data.borrow());
let old_user = u64::from_be_bytes(buf);
buf.copy_from_slice(&account_into_pda.data.borrow());
let old_into = u64::from_be_bytes(buf);
buf.copy_from_slice(&data);
let inc = u64::from_be_bytes(buf);
let new_user = old_user.checked_sub(inc).unwrap();
let new_into = old_into.checked_add(inc).unwrap();
account_user_pda.data.borrow_mut().copy_from_slice(&new_user.to_be_bytes());
account_into_pda.data.borrow_mut().copy_from_slice(&new_into.to_be_bytes());
Ok(())
```

6.4 Solana/泰铢币/完整链上代码

在本小节中, 我们给出完整泰铢币的代码.

```
#![allow(unexpected_cfgs)]

use solana_program::sysvar::Sysvar;

solana_program::entrypoint!(process_instruction);

pub fn process_instruction_mint(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    let accounts_iter = &mut accounts.iter();
    let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_user_pda = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program system
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program sysvar rent

    // Only Ada can mint more Thai Baht.
    assert_eq!(*account_user.key, solana_program::pubkey!("6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt"));

    // Data account is not initialized. Create an account and write data into it.
    if **account_user_pda.try_borrow_lamports().unwrap() == 0 {
        let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(8);
        let bump_seed =
            solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).1;
        solana_program::program::invoke_signed(
            &solana_program::system_instruction::create_account(
                account_user.key,
                account_user_pda.key,
                rent_exemption,
                8,
                program_id,
            ),
            accounts,
            &[&[&account_user.key.to_bytes(), &[bump_seed]]],
        )?;
        account_user_pda.data.borrow_mut().copy_from_slice(&u64::MIN.to_be_bytes());
    }

    // Mint.
    let mut buf = [0u8; 8];
    buf.copy_from_slice(&account_user_pda.data.borrow());
    let old = u64::from_be_bytes(buf);
    buf.copy_from_slice(&data);
    let inc = u64::from_be_bytes(buf);
    let new = old.checked_add(inc).unwrap();
    account_user_pda.data.borrow_mut().copy_from_slice(&new.to_be_bytes());
    Ok(())
}

pub fn process_instruction_transfer(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
```

```
    let accounts_iter = &mut accounts.iter();
    let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_user_pda = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_into = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_into_pda = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program system
    let _ = solana_program::account_info::next_account_info(accounts_iter)?; // Program sysvar rent

    let account_need_pda =
        solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).0;
    assert_eq!(account_user_pda.key, &account_need_pda);

    // Data account is not initialized. Create an account and write data into it.
    if **account_into_pda.try_borrow_lamports().unwrap() == 0 {
        let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(8);
        let bump_seed =
            solana_program::pubkey::Pubkey::find_program_address(&[&account_into.key.to_bytes()], program_id).1;
        solana_program::program::invoke_signed(
            &solana_program::system_instruction::create_account(
                account_user.key,
                account_into_pda.key,
                rent_exemption,
                8,
                program_id,
            ),
            accounts,
            &[&[&account_into.key.to_bytes(), &[bump_seed]]],
        )?;
        account_into_pda.data.borrow_mut().copy_from_slice(&u64::MIN.to_be_bytes());
    }

    // Transfer.
    let mut buf = [0u8; 8];
    buf.copy_from_slice(&account_user_pda.data.borrow());
    let old_user = u64::from_be_bytes(buf);
    buf.copy_from_slice(&account_into_pda.data.borrow());
    let old_into = u64::from_be_bytes(buf);
    buf.copy_from_slice(&data);
    let inc = u64::from_be_bytes(buf);
    let new_user = old_user.checked_sub(inc).unwrap();
    let new_into = old_into.checked_add(inc).unwrap();
    account_user_pda.data.borrow_mut().copy_from_slice(&new_user.to_be_bytes());
    account_into_pda.data.borrow_mut().copy_from_slice(&new_into.to_be_bytes());
    Ok(())
}

pub fn process_instruction(
    program_id: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    data: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    assert!(data.len() >= 1);
    match data[0] {
        0x00 => process_instruction_mint(program_id, accounts, &data[1..]),
        0x01 => process_instruction_transfer(program_id, accounts, &data[1..]),
        _ => unreachable!(),
    }
}
```

6.5 Solana/泰铢币/程序交互

6.5.1 编译并部署程序

在之前的文章中, 我们已经展示过如何编译以及部署程序, 此处不再赘述, 仅**再次**给出相关步骤和代码如下.

使用下面的命令编译程序代码.

```
$ cargo build-sbf -- -Znext-lockfile-bump
```

使用下面的 python 代码部署目标程序上链.

```
import pathlib
import pxsol

# Enable log
pxsol.config.current.log = 1

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(0x01))

program_data = pathlib.Path('target/deploy/pxsol_thaibaht.so').read_bytes()
program_pubkey = ada.program_deploy(bytearray(program_data))
print(program_pubkey) # 9SP6msRytNxeHXvW38xHxjsBHspqZERDTMh5Wi8xh16Q
```

此处泰铢币部署地址为 `9SP6msRytNxeHXvW38xHxjsBHspqZERDTMh5Wi8xh16Q`.

6.5.2 铸造代币

铸造新泰铢币的过程是通过一个 solana 交易来完成的. Ada 可以这样为自己铸造新的 100 个泰铢币. 您可能需要注意下 `data` 的构造, 它的长度为 9 个字节, 第一个字节为 0, 代表铸造操作.

另外要注意, 只有 ada 有权利铸造新的代币, 此权限已经在泰铢币的链上程序中被强制硬编码.

```
import base64
import pxsol


def mint(user: pxsol.wallet.Wallet, amount: int) -> None:
    assert user.pubkey.base58() == '6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt' # Is ada?
    prog_pubkey = pxsol.core.PubKey.base58_decode('9SP6msRytNxeHXvW38xHxjsBHspqZERDTMh5Wi8xh16Q')
    data_pubkey = prog_pubkey.derive_pda(user.pubkey.p)
    rq = pxsol.core.Requisition(prog_pubkey, [], bytearray())
    rq.account.append(pxsol.core.AccountMeta(user.pubkey, 3))
    rq.account.append(pxsol.core.AccountMeta(data_pubkey, 1))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.SysvarRent.pubkey, 0))
    rq.data = bytearray([0x00]) + bytearray(amount.to_bytes(8))
    tx = pxsol.core.Transaction.requisition_decode(user.pubkey, [rq])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([user.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
    r = pxsol.rpc.get_transaction(txid, {})
    for e in r['meta']['logMessages']:
        print(e)


if __name__ == '__main__':
```

```
    ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
    mint(ada, 100)
```

6.5.3 查询余额

使用 rpc 接口查询自己的数据账户中的数据, 并将其转换为 64 位无符号整数, 该数字即表示用户的泰铢币余额.

```
import base64
import pxsol

def balance(user: pxsol.core.PubKey) -> int:
    prog_pubkey = pxsol.core.PubKey.base58_decode('9SP6msRytNxeHXvW38xHxjsBHspqZERDTMh5Wi8xh16Q')
    data_pubkey = prog_pubkey.derive_pda(user.p)
    info = pxsol.rpc.get_account_info(data_pubkey.base58(), {})
    return int.from_bytes(base64.b64decode(info['data'][0]))

if __name__ == '__main__':
    ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
    print(balance(ada.pubkey))
```

6.5.4 转账

Ada 向 bob 转账 50 泰铢币, 转账完成后, 查询双方的余额.

```
import base64
import pxsol


def balance(user: pxsol.core.PubKey) -> int:
    prog_pubkey = pxsol.core.PubKey.base58_decode('9SP6msRytNxeHXvW38xHxjsBHspqZERDTMh5Wi8xh16Q')
    data_pubkey = prog_pubkey.derive_pda(user.p)
    info = pxsol.rpc.get_account_info(data_pubkey.base58(), {})
    return int.from_bytes(base64.b64decode(info['data'][0]))


def transfer(user: pxsol.wallet.Wallet, into: pxsol.core.PubKey, amount: int) -> None:
    prog_pubkey = pxsol.core.PubKey.base58_decode('9SP6msRytNxeHXvW38xHxjsBHspqZERDTMh5Wi8xh16Q')
    upda_pubkey = prog_pubkey.derive_pda(user.pubkey.p)
    into_pubkey = into
    ipda_pubkey = prog_pubkey.derive_pda(into_pubkey.p)
    rq = pxsol.core.Requisition(prog_pubkey, [], bytearray())
    rq.account.append(pxsol.core.AccountMeta(user.pubkey, 3))
    rq.account.append(pxsol.core.AccountMeta(upda_pubkey, 1))
    rq.account.append(pxsol.core.AccountMeta(into_pubkey, 0))
    rq.account.append(pxsol.core.AccountMeta(ipda_pubkey, 1))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.SysvarRent.pubkey, 0))
    rq.data = bytearray([0x01]) + bytearray(amount.to_bytes(8))
    tx = pxsol.core.Transaction.requisition_decode(user.pubkey, [rq])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([user.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
    r = pxsol.rpc.get_transaction(txid, {})
    for e in r['meta']['logMessages']:
        print(e)


if __name__ == '__main__':
```

```
ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
bob = pxsol.core.PriKey.int_decode(2).pubkey()
transfer(ada, bob, 50)
print(balance(ada.pubkey))
print(balance(bob))
```

6.6 Solana/泰铢币/获取完整源码

源码我已经打包好放上 github 啦!

如果你懒得跟着一步步敲代码(我懂你), 可以直接去看我准备好的示例项目. 地址在这儿, 不用谢我, 除非你想请我喝杯奶茶.

我知道许多开发者喜欢咖啡, 但对于我而言, 奶茶总是最好的.

> 有时候人生就像一部小说, 总得给我们点儿 déjà vu(既视感) 的惊喜.

```
$ git clone https://github.com/mohanson/pxsol-thaibaht
$ cd pxsol-thaibaht
```

```
$ python make.py deploy
# 2025/05/20 16:06:38 main: deploy program pubkey="9SP6msRytNxeHXvW38xHxjsBHspqZERDTMh5Wi8xh16Q"
```

注意到程序地址会被保存在 `res/info.json` 中, 后续操作会直接从此文件获取程序地址.

```
# Mint 21000000 Thai Baht for Ada
$ python make.py mint 21000000

# Show ada's balance
$ python make.py balance 6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
# 21000000

# Transfer 100 Thai Baht to Bob
$ python make.py transfer 100 8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH

# Show ada's balance
$ python make.py balance 6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt
# 20999900
# Show bob's balance
$ python make.py balance 8pM1DN3RiT8vbom5u1sNryaNT1nyL8CTTW3b5PwWXRBH
# 100
```

7. Solana/SPL Token

7.1 Solana/SPL Token/导言

在上一节课后, 学生花了许多的课外时间, 尝试来为泰铢币程序引入更多功能, 例如记录泰铢币的总发行量等. 随着功能越加越多, 学生变得日益憔悴.

学生: "教授, 我最近回头看我写的改进版的泰铢币程序, 感觉自己像是在拿牙签造摩天楼……"

教授: "这形容挺贴切的, 你写程序确实写得很有毅力, 但也确实有点原始社会的味道."

学生: "铸造要自己写, 转账还得手动加减数值, 我还得统计各种不同的数据, 我都快变成链上会计师了. 是不是其实有现成的工具能帮我偷个懒?"

教授战术后仰.

教授: "当然有, 孩子! Solana 可不是荒岛求生, 它可是区块链的大都会生活. 我们有 spl token, 专业, 安全, 钱包识别率 100%, 就像宜家的现成家具, 自己组装就完了."

学生: "哇, 这是个新名词! 那么什么是 spl token?"

教授: "这是 solana 网络上的一种代币标准. 记住, 它不仅仅是标准, 甚至有现成的代码."

学生: "我是不是可以把我那套泰铢币逻辑搬过去? 用 spl token 重新定义."

教授: "你不但可以, 而且应该! 你那些手写的 pda 和整数加法逻辑, 放在 spl 眼里, 那都是它两年前就吃剩下的冷饭."

学生: "哈哈, 那我赶紧升级一下, 不然我的泡泡币还没发出去, 光 debug 就先气跑投资人了."

教授: "很好, 来, 这一节我们就从 spl token 开始, 带你走上发行**标准规范**的泰铢币的康庄大道!"

7.2 Solana/SPL Token/历史与核心规范概览

让我们把时间退回到 2017 年, 在这一年, 以太坊上的 erc-20 正式成为以太坊生态的标准. 它定义了一套标准化的智能合约接口, 极大地降低了开发者创建代币的复杂性. 通过统一的规范, 开发者无需从头构建代币合约, 只需遵循 erc-20 标准即可快速发行代币.

该规范直接推动了 2017 年以及 2021 年的两次加密货币牛市. 第一次的行业热点是 ico(首次代币发行), 吸引了大量资金和开发者进入以太坊生态, 奠定了以太坊作为智能合约平台领导者的地位. 第二次的行业热点是 defi(去中心化金融), 我们现在熟知的几乎所有 defi 玩法(如 uniswap, compound, aave 等) 都爆发于 2021 年, 它们全都依赖 erc-20 代币进行价值传递, 流动性提供和链上治理.

我们可以肯定的说, 代币是以太坊生态发展的最关键推动力.

现在, 问题来到了 solana. 作为行业的年轻挑战者, solana 应当如何设计并实现一个代币标准?

7.2.1 破局之道

与以太坊不同, 基于 solana 账户模型的关系, solana 本身并不内建"一个钱包账户有多少余额"这样的逻辑. 开发者必须使用 pda 数据账户手动实现代币的存储, 增发与转账. 但这显然容易出错, 也容易重复造轮子. 用户还必须面临一个最大的问题: 不同的代币, 所支持的指令可能是不同的.

为了解决这个问题, solana 团队创建了 spl token, 作为统一的代币标准. 它参考了以太坊的 erc20 规范, 为开发者提供标准接口, 来促进生态兼容性. 这样钱包, 去中心化交易所, 去中心化金融等应用都可以通用识别代币.

得益于 solana 生态支持原生程序, 因此开发者无需自己部署 spl token 的代码, 相反开发者只需要创建一个被称为 spl 铸造账户的账户, 用于存储该代币的基本信息, 就可以创建出一个自己的 spl token.

7.2.2 查询链上代币信息

我们以 solana 上使用较为广泛的 usdc 代币为例, 该代币的铸造账户地址为 `EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v`, 您可以通过浏览器查询到该代币, 一些重要信息如代币总量, 小数点精度, 名称等如下.

Overview	**Value**
Current Supply	7761611891.221625
Decimals	6
data.name	USD Coin
data.symbol	USDC

根据最新数据, 截至2025 年 06 月, solana 网络上总共有一千三百万个spl token, 其中 pump.fun 一家就铸出了一千一百万个 token. 历史峰值出现在 2024 年 10 月 24 日, 仅仅一天就铸造了 3.6 万个新 spl token.

在 solana 创建一个新的代币几乎不花钱, 任何有一定经验的开发者都能在 1 秒之内就能完成. 同时像 pump.fun 以及 let's bonk 等网站上线的一键发币功能也吸引了大量非开发者用户, 助推了 spl token 的数量暴涨.

> 没时间解释了, 让我们铸出来再说!

7.2.3 发展历史

Solana 刚上线时, spl token 是以最小可行产品方式发布(v1). 该版本仅支持铸造, 销毁和转账基本操作. 这时的 spl token 主要目标是跑通代币系统, 尚不追求高级功能.

随着 raydium 等去中心化交易所的出现, spl token 成为了 solana 各类资产流通的基石(v2). 这一阶段的重点是规范了钱包的支持, 并引入 token metadata 概念. 该功能最初由 metaplex 提供, 目的是为扩充 spl token v1 版本的功能, 允许开发者为自己的代币设置名字, 符号, 图像等信息.

从 2023 至今, 最近的 spl token v3(也称 token-2022)开始引入更强大的功能, 例如允许冻结账户, 转账钩子(用于支持合约式回调). 但这也意味着 token-2022 开始与旧版 spl token 不完全兼容, 一些老旧的钱包和应用可能尚未完全支持. 虽然如此, 如果您正试图创造一种新代币, 从 token-2022 入手, 是一条最省心的路径, 本课程也将默认采用 token-2022 规范进行讲解.

7.2.4 核心规范概览

定义一个 spl token 需要使用到两个核心组件.

- **代币程序**: 代币的核心程序, 负责代币的创建, 转移, 销毁等操作. 有两个版本:
  - `TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb`. 新版, 即 token-2022.
  - `TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA`. 旧版, 目前已经基本弃用. 如果您在浏览器上查找一些极早期发行的代币, 可能会见到它.
- **铸造账户**: 定义代币的基础数据如总供应量, 小数位数等. 每个 spl token 都有一个唯一的铸造账户. 从 token-2022 开始, 您可以在此账户中添加"扩展数据", 最常用的扩展为代币元数据信息, 可用于存储该 spl token 的人类可读信息, 如名字, 符号, 图标 url 等.

一个普通铸造账户的整体数据结构大致上如下所示.

```
┌──────────────────────────────────────┐
│            Mint Account              │
|--------------------------------------|
|      decimals:         u8            |
|      supply:           u64           |
|      mint_authority:   Pubkey        |
|      freeze_authority: Pubkey        |
|--------------------------------------|
|      ... ...                         |
|--------------------------------------|
|      name: "Thai Baht Coin"          |
|      symbol: "THB"                   |
|      uri: "http://accu.cc/..."       |
└──────────────────────────────────────┘
```

基于一些历史原因, spl token 被设计为如此, 如果您有机会深入阅读其代码, 可能会发现它内部有着极重的历史包袱. 我们不得不使用一种非常复杂的数据结构来在铸造账户里存储一个代币的信息. 铸造账户里的基础数据是代币的基础, 用来管理代币供应和权限. 元数据则是基于"旧"的铸造账户结构, 为了保持一定的兼容性而附加上去的, 它像是一张包装纸, 描述代币的名字, logo 等供人类阅读的信息.

Solana 已经铺好路, 只等您发车.

7.3 Solana/SPL Token/创建您的代币

Pxsol 的内置钱包集成了一个简单且通用的接口, 用于创建您的 spl token (以及执行通用的 spl 操作).

7.3.1 创建代币

创建新代币的过程十分简单. 示例代码如下:

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
spl = ada.spl_create(
    'PXSOL',
    'PXS',
    'https://raw.githubusercontent.com/mohanson/pxsol/refs/heads/master/res/pxs.json',
    9,
)
print(spl) # 2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF
```

上述代码运行后, 在 `2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF` 地址上创建了一个新的代币. 该账户通常被称作代币的"铸造账户". 函数方法 `spl_create()` 接收四个参数, 分别是:

- `name`: 代币名称.
- `symbol`: 代币符号, 通常是名称的缩写, 就像 btc 之于 bitcoin 一样.
- `uri`: 代币元数据的 json 文件网络地址.
- `decimals`: 小数位, 例如 decimals=9 时, 1000000000 表示 1 个代币.

代币元数据的 json 文件的典型结构如下. 您需要将此 json 文件上传到公共可访问的服务器, 例如 arweave 或 ipfs, 然后在创建代币时将该文件地址作为参数传入 uri. 文件里的 `image` 字段非常重要, 通常关系到您的代币如何在钱包, 去中心化交易所等应用里的展示图像.

```
{
    "name": "PXSOL",
    "symbol": "PXS",
    "description": "Proof of study https://github.com/mohanson/pxsol",
    "image": "https://raw.githubusercontent.com/mohanson/pxsol/refs/heads/master/res/pxs.png"
}
```

7.3.2 交易费用

创建一个新的 spl token, 大约需要 0.004 sol 的租金以及 0.00001 sol 的交易费用. 截止至 2025 年 6 月, 总价值大概 $0.6 美元, 四舍五入等于不要钱!

7.4 Solana/SPL Token/铸造账户解析

铸造账户是用于定义和管理代币的核心账户类型. 铸造账户存储了代币的基本信息, 例如代币的供应量, 小数位数, 铸造权限以及冻结权限. 在 token-2022 升级中, 铸造账户还可以包含扩展字段, 以支持额外的功能, 例如代币元数据, 代币转账手续费等. 目前使用最广泛的是两个代币扩展: 元数据指针扩展以及元数据扩展.

我们使用下面的代码解析上一小节我们创建的代币.

```
import base64
import pxsol

info = pxsol.rpc.get_account_info('2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF', {})
data = bytearray(base64.b64decode(info['data'][0]))
mint = pxsol.core.TokenMint.serialize_decode(data)
print(mint)
# {
#     "auth_mint": "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
#     "supply": 0,
#     "decimals": 9,
#     "inited": true,
#     "auth_freeze": "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
#     "extensions": {
#         "metadata_pointer": {
#             "auth": "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
#             "hold": "2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF"
#         },
#         "metadata": [
#             {
#                 "auth_update": "6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt",
#                 "mint": "2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF",
#                 "name": "PXSOL",
#                 "symbol": "PXS",
#                 "uri": "https://raw.githubusercontent.com/mohanson/pxsol/refs/heads/master/res/pxs.json",
#                 "addition": {}
#             }
#         ]
#     }
# }
```

7.4.1 基础字段解析

- `auth_mint`: 铸造权限. 表示有权铸造(发行)新代币的账户地址. 此账户可以调用链上指令来增加代币供应量. 在本例中, 铸造权限归属于地址 `6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt`.
- `supply`: 当前代币的总供应量. 当前此代币供应量为 0, 表明尚未铸造任何代币.
- `decimals`: 代币的小数位数, 决定了代币的最小单位. 例如, decimals=9 表示代币的最小单位是 $1/10^9$(即 0.000000001). 这与以太坊上 erc-20 代币的 decimals 字段类似.
- `inited`: 表示铸造账户是否已初始化. 只有正确初始化的铸造账户可用于代币操作.
- `auth_freeze`: 冻结权限表示有权冻结或解冻代币账户的地址. 冻结功能可用于暂停代币的转账操作, 通常用于合规性或安全管理. 在本例中, 冻结权限与铸造权限相同, 均为 `6ASf5EcmmEHTgDJ4X4ZT5vT6iHVJBXPg5AN5YoTCpGWt`.

7.4.2 扩展字段解析

Token-2022 程序的一大特点是支持扩展功能, 这些功能通过 `extensions` 字段实现. 在本例中, 铸造账户包含了两种扩展: `metadata_pointer` 与 `metadata`.

扩展 `metadata_pointer` 表示为元数据指针, 也就是指明当前代币的元数据信息存储在哪里. 字段 `auth` 表明谁有权限修改元数据指针, 字段 `hold` 表明元数据的存储账户地址. 我们可以将元数据存储在铸造账户中, 也可以将元数据存储在另一个独立账户里. 不过就我来看, 允许将元数据信息和铸造信息分开存储更多是为了维持与旧版本的兼容性, 现实中您不应该这么做.

扩展 `metadata` 表示为元数据. 您可以很容易通过字段名来分析出大部分字段的用处, 因此我不再做多解释. 需要额外注意的是其中的 `addition` 字段, 它是一个 key/value 结构, 用于为代币提供更多的标识信息, 可便于在钱包或去中心化应用中展示.

7.5 Solana/SPL Token/铸造代币和查询代币余额

这节课我们来谈谈铸币. 不过在谈历史之前, 我想先来谈谈铸币权的一些有趣小历史.

在接触到比特币之后, 我对经济学产生了广泛的兴趣, 尤其是对古代中国经济学. 古代中国的货币大体上经历了从贵金属货币(如金银)到纸币, 之后纸币信用广泛崩溃, 社会重新回到贵金属货币这样一个轮回.

三国时期, 约公元 220-280, 刘备为筹措军费, 铸造"直百五铢"钱, 面值远超实际金属价值(约 30 倍), 大量掠夺民间财富, 使得蜀汉短短几十年, 从隆中对描述的"沃野千里, 天府之土"变为"百姓凋瘁", 此等变化甚至引得蜀中大儒谯周作仇国论. 三国志吴书亦有记载说"入其朝不闻正言, 经其野民皆菜色".

一千两百年后, 明太祖朱元璋为统一货币, 减少铜钱依赖, 发行纸币"大明宝钞", 同时立法禁止金银流通. 大明宝钞不由朝廷发行, 而是由内府(一个非政府机构, 其存在只为服务皇帝个人)发行. 在宝钞发行初期, 尚能勉强维持其价值, 到其儿子朱棣时期, 一千两的大明宝钞只能在黑市换得两百两银, 以至于民间社会开始拒收大明宝钞.

此等例子数不胜数. **垄断铸币权的人一定会滥发新币**.

不管怎么说, 本教程毕竟不是历史书, 因此让我们收束一下发散的思维, 想一想我们如何来铸造新币吧!

7.5.1 铸造代币和查询代币余额

您可以通过简单的代码为任意用户增发指定数量的代币余额. 注意: 只有代币的创建者有权限增发代币!

使用 `spl_mint()` 方法增发的代币数量需考虑小数位, 例如, 若 decimals=9, 则 1000000000 表示 1 个代币.

类似的, 使用 `spl_balance()` 方法查询您的代币余额时, 函数将返回一个数组: 数组第 0 位表示您拥有的代币数量, 第 1 位表示该代币的小数位.

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
spl = pxsol.core.PubKey.base58_decode('2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF')

ada.spl_mint(spl, ada.pubkey, 100000000 * 10 ** 9)
print(ada.spl_balance(spl)) # [100000000000000000, 9]
```

7.5.2 关联代币账户

与我们之前写的泰铢币程序一样, 您的代币实质上存储于一个 pda 账户里. 我们习惯称呼该账户为关联代币账户(associated token account, ata), 它是用于存储和管理用户持有 spl token 的特殊账户.

- 每个关联代币账户唯一对应于一个普通钱包账户和一个代币的铸造账户.
- 由程序自动生成, 遵循确定性地址规则, 确保同一个钱包和代币组合始终生成相同的关联代币账户地址.
- 存储用户的代币余额.

当您使用 pxsol 的 `spl_mint()` 方法增发代币时, 如果目标用户还没有关联代币账户, pxsol 则会自动为其生成.

我们尝试使用 rpc 接口查询一个关联代币账户内存储的数据, 获得返回数据如下.

```
import base64
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
spl = pxsol.core.PubKey.base58_decode('2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF')
ata = ada.spl_account(spl)
```

```
info = pxsol.rpc.get_account_info(ata.base58(), {})
data = base64.b64decode(info['data'][0])
print(data.hex())
```

返回数据总共长 170 个字节, 我们对其进行详细分析.

```
11c447d79a76ef38a896d72fe54b373bab14dcba868425645a1670180e656780 代币的地址
4cb5abf6ad79fbf5abbccafcc269d85cd2651ed4b885b5869f241aedf0a5ba29 用户的普通钱包地址
00008a5d78456301                                                 余额, 小端序表示, 为 100000000 * 10 ** 9
00000000                                                         是否设置代理地址 (delegate), 0=未设置
0000000000000000000000000000000000000000000000000000000000000000 代理地址
01                                                               账户状态(0=未初始化, 1=已初始化, 2=冻结)
00000000                                                         是否为原生代币, 0=非原生代币, 符合预期
0000000000000000                                                 若为原生代币时, 记录其租赁豁免阈值
0000000000000000                                                 委托金额
00000000                                                         是否允许关闭帐户
0000000000000000000000000000000000000000000000000000000000000000 关闭帐户的权限拥有者地址
02                                                               账户类型(0=未初始化, 1=铸造账户, 2=持仓账户)
0700                                                             代币扩展(7=不可改变账户所有权)
0000                                                             代币扩展(填充, 无实际意义)
```

您可以参考以下两个源代码链接, 这有助于您加深对关联代币账户的理解.

- 基础账户类型
- 账户类型以及扩展字段

在多数情况下, 我们只会关注前三个字段, 也就是代币的地址, 用户的普通钱包地址以及余额.

7.5.3 习题

例: Bob 是否被允许铸造新代币? 请编写代码尝试.

答:

```
import pxsol

bob = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(2))
spl = pxsol.core.PubKey.base58_decode('2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF')

bob.spl_mint(spl, bob.pubkey, 100000000 * 10 ** 9)
# Exception: {
#     'code': -32002,
#     'message': 'Transaction simulation failed: Attempt to debit an account but found no record of a prior credit.',
#     'data': {
#         'accounts': None,
#         'err': 'AccountNotFound',
#         'innerInstructions': None,
#         'logs': [],
#         'replacementBlockhash': None,
#         'returnData': None,
#         'unitsConsumed': 0
#     }
# }
```

7.6 Solana/SPL Token/转账

Solana 是区块链世界的代币流动超级引擎. 凭借其极低的交易确认时间以及几乎算是免费的交易手续费, 让代币转账变得如丝般顺滑.

7.6.1 转账

您可以使用 `spl_transfer()` 发送任意 spl token 给任意用户, 但记得考虑代币的小数位. 此函数会自动检测目标用户是否存在 ata 账户, 如果没有的会, 则会为其进行自动创建.

```
import pxsol

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
bob = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(2))
spl = pxsol.core.PubKey.base58_decode('2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF')

print(ada.spl_balance(spl)) # [100000000000000000, 9]
ada.spl_transfer(spl, bob.pubkey, 100 * 10 ** 9)
print(ada.spl_balance(spl)) # [99999900000000000, 9]
print(bob.spl_balance(spl)) # [100000000000, 9]
```

7.6.2 成为空投之王

在 solana 网络上, 您可能经常遇见有陌生人给您转账 spl token. 这种由陌生人或广泛用户分发代币的行为通常被称为"空投". 空投是项目方或个人免费向特定群体或不特定群体的钱包地址发送代币的行为, 这种行为在 solana, 以太坊等区块链生态中非常常见, 旨在推广项目或增加代币的流通性.

> 这种行为常见于 meme 币项目，一些散户会通过代币的持有者人数来判断代币的活跃性或潜力, 项目方通过空投可以短时间提升这个值.

例: 请向随机的 1000 个地址空投您的代币, 成为空投之王!

答:

```
import pxsol
import random

pxsol.config.current.log = 1

ada = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(1))
spl = pxsol.core.PubKey.base58_decode('2CMXJX8arHRsiiZadVheRLTd6uhP7DpbaJ9hRiRMSGcF')

for _ in range(1000):
    dst = pxsol.core.PriKey.random().pubkey()
    ada.spl_transfer(spl, dst, 100 * 10 ** 9)
```

7.7 Solana/SPL Token/指令详解(一)

在本课中, 我们将深入解读 `pxsol.wallet.Wallet` 类中的 `spl_create()` 方法. 这是一个高级接口, 用于在 solana 区块链上创建一个带有元数据的 spl token, 并初始化相关权限与租金豁免等功能.

7.7.1 源码

> 为了避免读者不停切换页面, 我将待分析的代码复制在了这里.

```
def spl_create(self, name: str, symbol: str, uri: str, decimals: int) -> pxsol.core.PubKey:
    # Create a new token.
    mint_prikey = pxsol.core.PriKey.random()
    mint_pubkey = mint_prikey.pubkey()
    mint_size = pxsol.program.Token.size_extensions_base + pxsol.program.Token.size_extensions_metadata_pointer
    # Helper function to tack on the size of an extension bytes if an account with extensions is exactly the size
    # of a multisig.
    assert mint_size != 355
    addi_size = pxsol.program.Token.size_extensions_metadata + len(name) + len(symbol) + len(uri)
    mint_lamports = pxsol.rpc.get_minimum_balance_for_rent_exemption(mint_size + addi_size, {})
    r0 = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
    r0.account.append(pxsol.core.AccountMeta(self.pubkey, 3))
    r0.account.append(pxsol.core.AccountMeta(mint_pubkey, 3))
    r0.data = pxsol.program.System.create_account(mint_lamports, mint_size, pxsol.program.Token.pubkey)
    r1 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
    r1.account.append(pxsol.core.AccountMeta(mint_pubkey, 1))
    r1.data = pxsol.program.Token.metadata_pointer_extension_initialize(self.pubkey, mint_pubkey)
    r2 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
    r2.account.append(pxsol.core.AccountMeta(mint_pubkey, 1))
    r2.account.append(pxsol.core.AccountMeta(pxsol.program.SysvarRent.pubkey, 0))
    r2.data = pxsol.program.Token.initialize_mint(decimals, self.pubkey, self.pubkey)
    r3 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
    r3.account.append(pxsol.core.AccountMeta(mint_pubkey, 1))
    r3.account.append(pxsol.core.AccountMeta(self.pubkey, 0))
    r3.account.append(pxsol.core.AccountMeta(mint_pubkey, 0))
    r3.account.append(pxsol.core.AccountMeta(self.pubkey, 2))
    r3.data = pxsol.program.Token.metadata_initialize(name, symbol, uri)
    tx = pxsol.core.Transaction.requisition_decode(self.pubkey, [r0, r1, r2, r3])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([self.prikey, mint_prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
    return mint_pubkey
```

7.7.2 实现流程拆解

调用 `spl_create()` 方法, 实质上会组装并发送一笔交易, 该交易包含四条链上指令, 分别封装在 requisition 中, 依次执行如下.

指令 1: 创建铸造账户

```
r0 = pxsol.core.Requisition(pxsol.program.System.pubkey, [], bytearray())
r0.account.append(pxsol.core.AccountMeta(self.pubkey, 3))
r0.account.append(pxsol.core.AccountMeta(mint_pubkey, 3))
r0.data = pxsol.program.System.create_account(mint_lamports, mint_size, pxsol.program.Token.pubkey)
```

上述代码为新 spl token 分配一个租金豁免的账户. 账户大小需要包括基础数据与扩展数据. 参数 `mint_lamports` 是租金豁免的最低 lamports, 通过 rpc 查询获得.

指令 2: 初始化元数据扩展指针

```
r1 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
r1.account.append(pxsol.core.AccountMeta(mint_pubkey, 1))
r1.data = pxsol.program.Token.metadata_pointer_extension_initialize(self.pubkey, mint_pubkey)
```

上述代码启用 token-2022 的扩展字段: metadata pointer. 这是 token-2022 的一个特性, 允许你在铸造账户上挂载额外的元数据结构. 在稍后的指令中, 我们将实际去挂载数据, 该指令当前仅做申明使用. 除此扩展之外, token-2022 实际上还支持另外几十个不同的扩展, 您可以在此页面了解更多.

指令 3: 初始化铸造账户

```
r2 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
r2.account.append(pxsol.core.AccountMeta(mint_pubkey, 1))
r2.account.append(pxsol.core.AccountMeta(pxsol.program.SysvarRent.pubkey, 0))
r2.data = pxsol.program.Token.initialize_mint(decimals, self.pubkey, self.pubkey)
```

上述代码在创建出来的铸造账户里设置代币的小数精度, 铸造权限和冻结权限. 多数情况下, 将这些权限设置为发布者本人就可以了. 初始化完成后, 您才能开始铸币.

指令 4: 初始化元数据

```
r3 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
r3.account.append(pxsol.core.AccountMeta(mint_pubkey, 1))
r3.account.append(pxsol.core.AccountMeta(self.pubkey, 0))
r3.account.append(pxsol.core.AccountMeta(mint_pubkey, 0))
r3.account.append(pxsol.core.AccountMeta(self.pubkey, 2))
r3.data = pxsol.program.Token.metadata_initialize(name, symbol, uri)
```

上述代码将元数据(名称, 符号, 地址)直接写入铸造账户的元数据扩展字段中.

7.8 Solana/SPL Token/指令详解(二)

在上一节课中, 我们学习了如何使用 `spl_create()` 创建一个带有元数据的 spl token. 在本节课, 我们将深入学习另一个至关重要的操作: `spl_mint()` 是如何将 token 铸造并分发给目标地址(或自己)的?

7.8.1 源码

> 为了避免读者不停切换页面, 我将待分析的代码复制在了这里.

```
def spl_mint(self, mint: pxsol.core.PubKey, recv: pxsol.core.PubKey, amount: int) -> None:
    # Mint a specified number of tokens and distribute them to self. Note that amount refers to the smallest unit
    # of count, For example, when the decimals of token is 2, you should use 100 to represent 1 token. If the
    # token account does not exist, it will be created automatically.
    recv_ata_pubkey = Wallet.view_only(recv).spl_account(mint)
    r0 = pxsol.core.Requisition(pxsol.program.AssociatedTokenAccount.pubkey, [], bytearray())
    r0.account.append(pxsol.core.AccountMeta(self.pubkey, 3))
    r0.account.append(pxsol.core.AccountMeta(recv_ata_pubkey, 1))
    r0.account.append(pxsol.core.AccountMeta(recv, 0))
    r0.account.append(pxsol.core.AccountMeta(mint, 0))
    r0.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    r0.account.append(pxsol.core.AccountMeta(pxsol.program.Token.pubkey, 0))
    r0.data = pxsol.program.AssociatedTokenAccount.create_idempotent()
    r1 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
    r1.account.append(pxsol.core.AccountMeta(mint, 1))
    r1.account.append(pxsol.core.AccountMeta(recv_ata_pubkey, 1))
    r1.account.append(pxsol.core.AccountMeta(self.pubkey, 2))
    r1.data = pxsol.program.Token.mint_to(amount)
    tx = pxsol.core.Transaction.requisition_decode(self.pubkey, [r0, r1])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([self.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
```

7.8.2 实现流程拆解

方法 `spl_mint()` 会组装一笔交易, 内含两条链上指令.

指令 1: 创建关联代币账户

```
r0 = pxsol.core.Requisition(pxsol.program.AssociatedTokenAccount.pubkey, [], bytearray())
r0.account.append(pxsol.core.AccountMeta(self.pubkey, 3))
r0.account.append(pxsol.core.AccountMeta(recv_account_pubkey, 1))
r0.account.append(pxsol.core.AccountMeta(recv, 0))
r0.account.append(pxsol.core.AccountMeta(mint, 0))
r0.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
r0.account.append(pxsol.core.AccountMeta(pxsol.program.Token.pubkey, 0))
r0.data = pxsol.program.AssociatedTokenAccount.create_idempotent()
```

上述代码为接收人自动创建与之相关联的关联代币账户. 使用 `create_idempotent()` 保证即使目标账户已经存在, 指令也不会报错导致交易失败, 这是 `AssociatedTokenAccount.create_idempotent()` 指令区别于 `AssociatedTokenAccount.create()` 最重要的一点. 您可以这样理解这条指令的逻辑:

- 如果目标关联代币账户已经存在, 则正常退出.
- 如果目标关联代币账户还未存在, 则创建账户, 并正常退出.

> 正如它的名字所描述的, create_idempotent() 是幂等的. 幂等性是数学和计算机学中的一个概念. 在计算机中, 幂等性是指: 用户对于同一个操作发起一次请求或者多次请求, 获得的结果都是一致的. 不会因为请求多次出现异常情况.

指令 2: 铸造代币

```
r1 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
r1.account.append(pxsol.core.AccountMeta(mint, 1))
r1.account.append(pxsol.core.AccountMeta(recv_account_pubkey, 1))
r1.account.append(pxsol.core.AccountMeta(self.pubkey, 2))
r1.data = pxsol.program.Token.mint_to(amount)
```

上述代码将一定数量的代币铸造到接收者账户. 铸造代币需要拥有铸造权限, 在 `pxsol.wallet.Wallet` 的设计里, 代币的铸造权限拥有者默认是代币发布者本人.

铸造代币将会增加代币的总供应量.

7.9 Solana/SPL Token/指令详解(三)

本小节带你深入理解 spl 转账操作背后的链上逻辑, 帮助你掌握在 solana 上安全自动地转移 spl token 的正确方式. 方法 `spl_transfer()` 主要完成以下两件事:

- 确保接收者的关联代币账户存在. 如果没有，自动创建.
- 从当前钱包的关联代币账户转账指定数量的代币到接收者账户.

7.9.1 源码

> 为了避免读者不停切换页面, 我将待分析的代码复制在了这里.

```
def spl_transfer(self, mint: pxsol.core.PubKey, recv: pxsol.core.PubKey, amount: int) -> None:
    # Transfers tokens to the target. Note that amount refers to the smallest unit of count, For example, when the
    # decimals of token is 2, you should use 100 to represent 1 token. If the token account does not exist, it will
    # be created automatically.
    self_ata_pubkey = self.spl_account(mint)
    recv_ata_pubkey = Wallet.view_only(recv).spl_account(mint)
    r0 = pxsol.core.Requisition(pxsol.program.AssociatedTokenAccount.pubkey, [], bytearray())
    r0.account.append(pxsol.core.AccountMeta(self.pubkey, 3))
    r0.account.append(pxsol.core.AccountMeta(recv_ata_pubkey, 1))
    r0.account.append(pxsol.core.AccountMeta(recv, 0))
    r0.account.append(pxsol.core.AccountMeta(mint, 0))
    r0.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    r0.account.append(pxsol.core.AccountMeta(pxsol.program.Token.pubkey, 0))
    r0.data = pxsol.program.AssociatedTokenAccount.create_idempotent()
    r1 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
    r1.account.append(pxsol.core.AccountMeta(self_ata_pubkey, 1))
    r1.account.append(pxsol.core.AccountMeta(recv_ata_pubkey, 1))
    r1.account.append(pxsol.core.AccountMeta(self.pubkey, 2))
    r1.data = pxsol.program.Token.transfer(amount)
    tx = pxsol.core.Transaction.requisition_decode(self.pubkey, [r0, r1])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([self.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
```

7.9.2 实现流程拆解

方法 `spl_transfer()` 会组装并发送一笔包含两条链上指令的交易.

指令 1：创建关联代币账户

```
r0 = pxsol.core.Requisition(pxsol.program.AssociatedTokenAccount.pubkey, [], bytearray())
r0.account.append(pxsol.core.AccountMeta(self.pubkey, 3))
r0.account.append(pxsol.core.AccountMeta(recv_ata_pubkey, 1))
r0.account.append(pxsol.core.AccountMeta(recv, 0))
r0.account.append(pxsol.core.AccountMeta(mint, 0))
r0.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
r0.account.append(pxsol.core.AccountMeta(pxsol.program.Token.pubkey, 0))
r0.data = pxsol.program.AssociatedTokenAccount.create_idempotent()
```

上述代码与 `spl_mint()` 的第一条指令功能以及作用相同.

指令 2：转账

```
r1 = pxsol.core.Requisition(pxsol.program.Token.pubkey, [], bytearray())
r1.account.append(pxsol.core.AccountMeta(self_ata_pubkey, 1))
r1.account.append(pxsol.core.AccountMeta(recv_ata_pubkey, 1))
r1.account.append(pxsol.core.AccountMeta(self.pubkey, 2))
r1.data = pxsol.program.Token.transfer(amount)
```

上述代码将一定数量的代币从当前账户转移到接收者账户. 转账操作最终会由 token-2022 程序验证并计入双方余额变化.

7.10 Solana/SPL Token/后记

在我写作本章的时候, 我的创作火花突然被点燃了. 我开始思考一件事情: 我们只在本地开发网络学习 solana 的方式真的是最好的吗? 是否应该尝试在 solana 主网上做一些事情? 这样我和我的读者可以共同合作完成一场教学课程, 您可以查询区块链历史记录, 知道我在过去做了什么, 您也能看到同为读者的其它人正在做什么, 没有比这更加棒的事情了!

通过这几个月的高强度写作, solana 让我相信, 普通人也能通过代币讲述自己的故事.

我的下一阶段计划建立在 solana 主网络上, 我将向读者介绍如何在主网运营一个代币项目: 包括发行代币, 创建代币的流动性池以及进行代币空投. 我希望这些代币不仅是迷因的载体, 更是学习的象征. 教学用的代币的空投会基于"学习证明", 任何学习过本教程并实际动手实验的读者都能通过创建一个学习证明来申请获得一些代币空投, 这个过程将是自动完成的. 每当有人持有这枚代币, 就仿佛在说: "我探索过 solana 的世界, 我学习过这份教程, 我是这个生态的一部分." 这枚代币将在您的钱包中熠熠生辉.

- 学习证明并非由我开具给您, 相反, 由您自己生成.
- 您可以交易空投获得的代币, 像其它 meme 币一样.
- 教学用代币可以被无限增发, 确保其只有纪念意义.

8. Solana/在主网发行您的代币

8.1 Solana/在主网发行您的代币/导言

学生兴奋地冲进教室.

学生: "教授! 我决定了, 我要把我写的泡泡币部署到 solana 主网上!"

教授: "孩子你确定不改个名字吗, 这听起来就有点泡沫经济那味儿. 话说你打算怎么启动它?"

学生: "我已经有一个 token-2022 的 mint 合约了. 我想让大家能买能卖, 还想搞点空投, 做个网站, 再上一些 dex, 走向 web3 的人生巅峰!"

教授: "慢着, 先让我来检查下你的泡泡币. 嗯, 你的泡泡币为何没有图标? 记得设置泡泡币的图标链接. 人类是视觉动物, 没头像, 没人敢买!"

学生: "我想我可能错误的设置了 url, 让我来修复好它."

教授: "很好. 下一步你准备如何分发你的泡泡币? 这个币现在可只有你一个持有人."

学生: "我想给认识的朋友们发币庆祝, 空投 1000 枚泡泡币! 之后我想搞个索取网站, 让陌生的用户可以连钱包领取."

教授: "非常正确!"

学生: "我希望大家能自由交易泡泡币. 我需要用 raydium 创建一个交易池子!"

教授: "创建交易池子需要提供初始流动性. 确保你有足够资金."

学生: "我还需要建立信任, 我担心有人说泡泡币是 scam."

教授: "没错. 你可以锁仓自己的代币, 向社区公开. 别忘记写白皮书和计划书, 明确用途, 通胀机制, 预期发展等."

学生: "教授, 我懂了!"

8.2 Solana/在主网发行您的代币/从测试网到主网的迁移

经过数日的奋战, 我们的泰铢币, 泡泡币, 猫猫币以及乌龟币们终于在本地网络上稳定运行, 增发和交易等功能都一切顺利. 现在, 是时候考虑在主网上正式发布您的代币了!

从本地网络到主网, 通常来说我们只需要换一个 url. 这个过程看起来简单, 实际上一点也不复杂.

8.2.1 常用环境对比

在 solana 上, 我们常见的环境有三种:

网络	特点	用途
Localnet	本地节点, 速度最快	单机测试, 开发初期验证
Devnet	公共测试网, 稳定性好	合约测试, 代币实验
Mainnet Beta	主网, 所有真实资产运行的地方	真金白银就在这里

在 pxsol 里, 使用下面的代码来切换以上三种环境.

```
import pxsol

pxsol.config.current = pxsol.config.develop # Localnet, 默认
pxsol.config.current = pxsol.config.testnet # Devnet
pxsol.config.current = pxsol.config.mainnet # Mainnet Beta
```

虽然它们的接口基本一致, 但有几个关键差异, 必须留意.

- 本地节点和公共测试网的 sol 是免费的, 但主网上的 sol 是有价值的. 您需要使用真金白银去购买 sol.
- 有许多黑帽和白帽盯着您的合约: 测试时漏掉的权限问题, 到了主网就是真 rug 事故.
- 节点速率限制: 主网节点普遍对 rpc 请求有更高限制, 频繁调用可能导致 ip 被 ban.

8.2.2 预估费用

发布一个完整的代币系统, 通常涉及以下操作:

操作	费用 (约估)
创建 mint 账户	0.002 sol 左右
创建 metadata 账户	0.01 sol 左右(存储多)
创建初始 token account	0.002 sol 左右
初始化池子(如 raydium)	0.01 sol ~ 0.05 sol
空投合约部署	0.5 个 sol 以上

所以准备至少 1 sol 预算是基本线. 在 2025 年 7 月, 这约等于 200 美元.

8.2.3 速率限制

在主网上发送高频交易请求时, rpc 限速可能是您最大的障碍. Pxsol 默认使用公共 rpc, 其拥有较高的速率限制, 因此您可能会发现当把 pxsol 配置为主网后, 其性能会严重下降: 但这并不是 pxsol 的问题, 而是 pxsol 在规避公共 rpc 的速率限制.

您可以通过以下代码, 自定义 rpc 请求的地址以及每次 rpc 请求后的冷却时间.

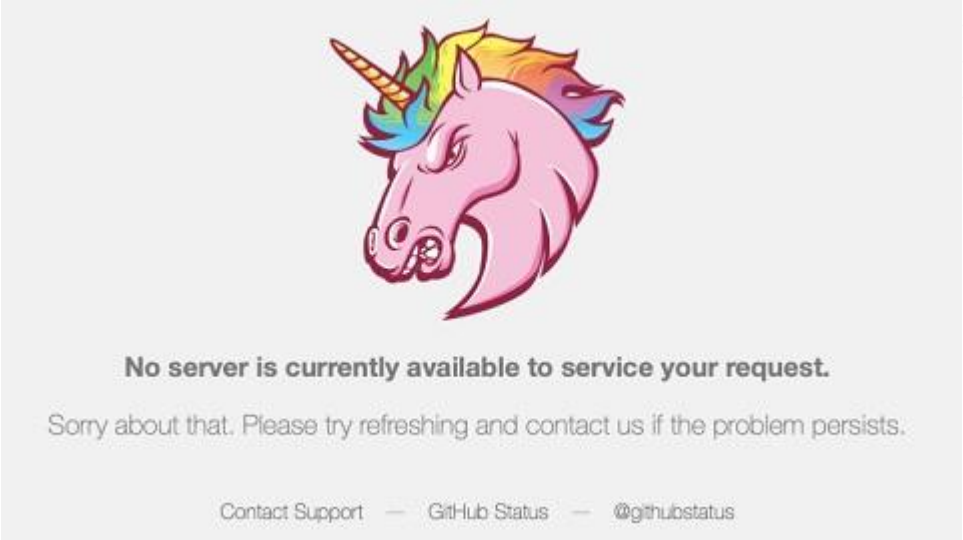
```
import pxsol

pxsol.config.current = pxsol.config.mainnet
pxsol.config.current.rpc.qps = 1
pxsol.config.current.rpc.url = 'https://api.mainnet-beta.solana.com'
```

您也可以使用一些付费的节点, 例如 helius 或者 triton one 这种由服务提供商提供的节点, 或者直接搭建自己的主网 fullnode, 成本较高但完全自由.

8.2.4 图片与元数据托管建议

没人希望自己的代币图标指向一个 404. 主网运行时, 元数据托管尤其重要. 推荐的做法是使用 arweave 或 ipfs 储存图像, 也可以使用 github, 但使用 github 时偶尔会失效.

> 您成功召唤了 github 独角兽!

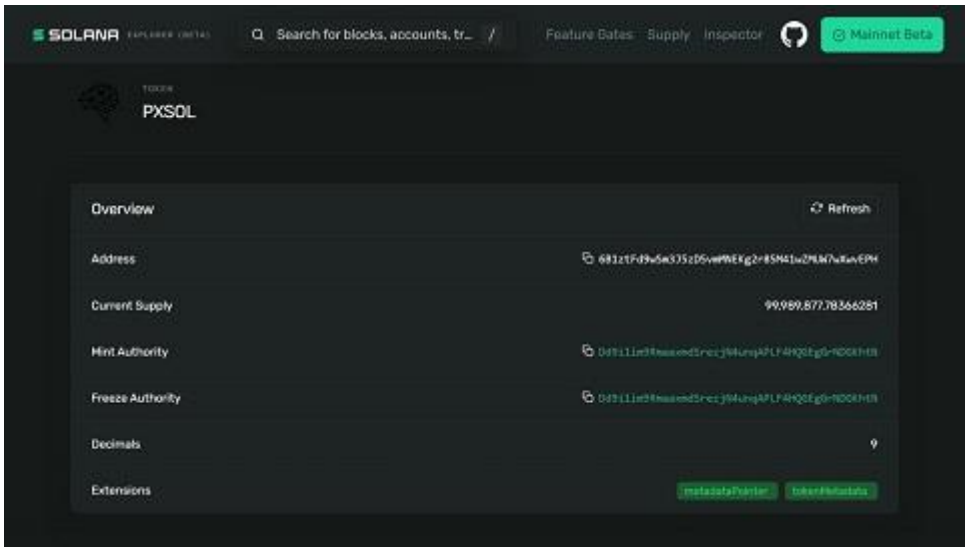


8.3 Solana/在主网发行您的代币/在主网发行您的代币

与在本地网络发行代币的步骤类似, 我们使用以下的代码将代币发行在主网上. 记得切换网络使用 `pxsol.config.current = pxsol.config.mainnet`.

```
import pxsol

# Switch to mainnet.
pxsol.config.current = pxsol.config.mainnet

you = pxsol.wallet.Wallet(pxsol.core.PriKey.base58_decode('Put your private key here'))
spl = you.spl_create(
    'PXSOL',
    'PXS',
    'https://raw.githubusercontent.com/mohanson/pxsol/refs/heads/master/res/pxs.json',
    9,
)
print(spl) # 6B1ztFd9wSm3J5zD5vmMNEKg2r85M41wZMUW7wXwvEPH
```

您可以在浏览器里浏览我刚刚新发行的 pxs 代币.



在发行代币后可以为自己铸造一些初始代币, 例如一亿枚.

```
import pxsol

pxsol.config.current = pxsol.config.mainnet
you = pxsol.wallet.Wallet(pxsol.core.PriKey.base58_decode('Put your private key here'))
spl = pxsol.core.PubKey.base58_decode('6B1ztFd9wSm3J5zD5vmMNEKg2r85M41wZMUW7wXwvEPH')
you.spl_mint(spl, you, 100000000 * 10 ** 9)
```

完成上述步骤后, 您可以通过钱包工具导入您的代币, 验证代币是否已经生效. 打开钱包, 查看您的代币是否正确显示吧!

8.4 Solana/在主网发行您的代币/上架去中心化交易所

在区块链世界里, 去中心化交易所(dex)是一种无需信任中介, 直接进行交易的场所. 在传统的中心化交易所(如 binance, coinbase)中, 所有交易都通过交易所的中央服务器进行撮合, 交易所负责资产的保管和转账. 然而, 这样的中心化交易存在信任风险, 一旦交易所被攻击或发生运营问题, 用户的资金就可能遭受损失.

去中心化交易所则不同, 它们基于区块链技术, 通过智能合约和自动做市商算法, 使用户可以在无需中介的情况下直接交易代币. 用户的资金不再存储在中心化平台上, 而是由区块链上的智能合约管理.

在 solana 生态下, raydium 是其中最受欢迎的 dex 之一. 本小节会带您一步步了解如何将您的代币上架到 raydium, 并为其提供流动性.

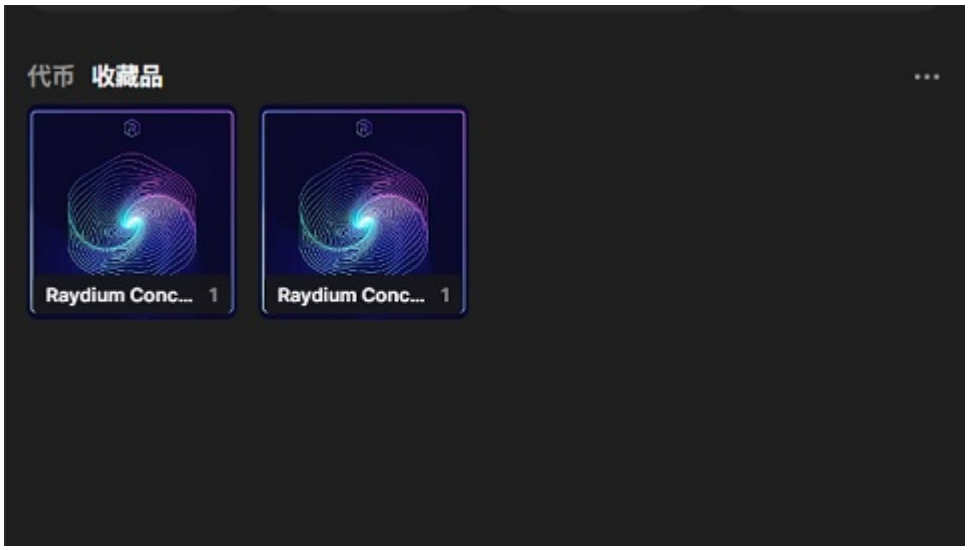
8.4.1 将代币上架到 raydium

为了让用户能够交易您的代币, 您首先需要为其创建一个交易池. 交易池是由两种代币组成的流动性池, raydium 采用自动做市商模型来提供流动性.

上一小节我们已经在主网部署了一个 pxs 代币, 现在我准备让它和 sol 进行交易, 因此我需要为这两个代币创建一个池子. 在网页上的操作步骤十分简单:

1. 打开 https://raydium.io/liquidity-pools/
2. 点击右上角 create 按钮.
3. 选择 pxs 代币与 sol 代币, 设置交易费率.
4. 设置 pxs 代币的初始价格.
5. 完成!

由于我为自己的代币提供了流动性, raydium 会为我发放 lp token, 这些代币代表我在流动性池中的份额. 您应当妥善保管这些 lp token, 当您试图撤出流动性时需要提供这些 lp token 以取回您的资金.



8.4.2 在 raydium 上交易 pxs

一旦流动性池创建完毕, 用户就可以通过 raydium 进行交易. Raydium 使用自动做市商模型, 在没有传统订单簿的情况下通过池子进行代币交换. 用户可以通过 raydium 的界面选择代币对, 输入想要交易的数量, 并通过一个简单的 swap 操作完成交易.

您可以通过此链接交易 pxs. 我为 pxs 创建了一个很小的交易池, 您可以在此交易池中随意买卖 pxs. 由于流动性较小, 交易可能会产生较大的滑点, 导致交易价格不如预期.

在后续章节中, 我会实现一个关于 pxs 代币的空投合约. 您可以领取免费的 pxs 空投并在此池子中卖出以换取 sol.

敬请期待.

8.5 Solana/在主网发行您的代币/设计空投规则

在主网运营一个代币项目, 代币分发一直是个大难题. 空投是一种流行的大幅度增加代币持有人数的方式, 可以帮助新项目吸引用户.

我为 pxs 代币设计了一种十分简单的空投方式, 旨在教学并启发读者设计自己的复杂空投规则. 不同于传统的空投方式, pxs 代币的空投被计划空投给任何"学习过此教程"的用户. 空投通过智能合约自动分发的, 没有手动转账的繁琐过程. 读者只需支付一笔非常低的 solana 网络手续费, 即可免费领取固定数量的 pxs 代币.

在接下来的课程中, 我们会实现一个空投程序, 该程序会为每一个和它交互的用户空投 5 pxs, 没有限制也没有上限.

不过, 请注意, 理论上, 任何人都可以领取无限数量的 pxs 代币, 但实际的隐性限制是 solana 网络的手续费. 换句话说, 5 枚 pxs 的实际价值等于一次 solana 交易的手续费, 在今天约等于 0.001 美元.

8.6 Solana/在主网发行您的代币/由程序控制的代币

空投就像是给大家发糖果, 可是, 要实现一个自动发糖的程序(我们这里指的是空投程序), 首先必须确保程序有足够的糖果库存. 这就像是, 你想在派对上分发糖果, 首先得保证自己手里有糖果!

在 solana 上, 代币不会直接存储在钱包或程序里, 而是通过关联代币账户来管理. 由程序控制的代币也需要通过关联代币账户来管理和转移.

8.6.1 代币转账流程

我们再来复习一下关联代币账户. 代币的转账并不是直接发生在用户的钱包和接收账户之间, 而是通过关联代币账户来完成的. 你可以把它想象成你大钱包里的一个小专用钱包盒, 每个小钱包盒都对应一种代币.

代币转账是打开您的大钱包后, 再打开您的小钱包, 然后发送到另一个用户的小钱包里.

通常来讲, 要实现由程序控制的代币这个功能, 需要有以下三个账户层级.

- 程序账户. 这儿的程序账户是就是我们的空投程序, 控制着代币的所有权. 它负责分配和转移代币. 程序账户通过执行 invoke 或 invoke_signed 指令来与其他账户交互.
- 程序 pda 账户. 程序账户通常不能直接拥有代币, 程序账户可以通过派生一个 pda 账户来存储代币. 这个 pda 账户是通过程序地址和特定的种子生成的. 派生账户的签名权限是由程序账户通过 `invoke_signed` 来控制的.
- 关联代币账户. 实际存储代余额的账户. 关联代币账户由程序的 pda 账户派生而来.

简单来说, 账户之间的关系: 程序账户 -> 程序 pda 账户 -> 关联代币账户.

真正拥有代币所有权的是程序 pda 账户, 但得益于 pda 账户的特性, 程序账户可以代替其 pda 账户签名, 因此程序账户也就变相拥有了代币的所有权.

8.6.2 工作原理详解

在 solana 中, 程序账户通过签名来控制 pda 账户, 使得程序能够代替 pda 账户进行代币管理. 这是由于 pda 账户本身没有私钥, 它的签名操作只能通过与程序账户相关的签名种子来完成.

假设你有一个程序, 它需要把代币转给用户. 那么过程如下.

1. 程序账户通过 `invoke_signed` 指令向 pda 账户发起转账请求.
2. pda 账户存储着代币, 程序通过签名种子签署转账, 完成从 pda 账户到用户关联代币账户的代币转移.

核心代码示例:

```
let account_seed = &[];
let account_bump = solana_program::pubkey::Pubkey::find_program_address(&[account_seed], account_mana.key).1;
solana_program::program::invoke_signed(
    &spl_token_2022::instruction::transfer_checked(
        &account_spl.key,
        &account_mana_spla.key,
        &account_mint.key,
        &account_user_spla.key,
        &account_mana_auth.key,
        &[],
        5000000000,
        9,
    )?,
    accounts,
```

```
    &[&[account_seed, &[account_bump]]],
)?;
```

8.7 Solana/在主网发行您的代币/实现空投程序

我们要实现的空投程序主要包含两个功能.

1. 任意调用该空投程序的用户, 程序会自动为其创建一个关联代币账户.
2. 转账 5 pxs 给他.

合约程序实现如下. 程序内部会调用两条指令来实现上述的功能.

```
solana_program::entrypoint!(process_instruction);

pub fn process_instruction(
    _: &solana_program::pubkey::Pubkey,
    accounts: &[solana_program::account_info::AccountInfo],
    _: &[u8],
) -> solana_program::entrypoint::ProgramResult {
    let accounts_iter = &mut accounts.iter();
    let account_user = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_user_spla = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_mana = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_mana_auth = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_mana_spla = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_mint = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?;
    let account_spl = solana_program::account_info::next_account_info(accounts_iter)?;
    let _ = solana_program::account_info::next_account_info(accounts_iter)?;

    solana_program::program::invoke(
        &spl_associated_token_account::instruction::create_associated_token_account_idempotent(
            &account_user.key,
            &account_user.key,
            &account_mint.key,
            &account_spl.key,
        ),
        accounts,
    )?;
    let account_seed = &[];
    let account_bump = solana_program::pubkey::Pubkey::find_program_address(&[account_seed], account_mana.key).1;
    solana_program::program::invoke_signed(
        &spl_token_2022::instruction::transfer_checked(
            &account_spl.key,
            &account_mana_spla.key,
            &account_mint.key,
            &account_user_spla.key,
            &account_mana_auth.key,
            &[],
            5000000000,
            9,
        )?,
        accounts,
        &[&[account_seed, &[account_bump]]],
    )?;

    Ok(())
}
```

编译程序后, 使用下面的代码将其部署在主网.

```
import pxsol
pxsol.config.current = pxsol.config.mainnet

user = pxsol.wallet.Wallet(pxsol.core.PriKey.base58_decode('Put your private key here'))
with open('target/deploy/pxsol_spl.so', 'rb') as f:
    data = bytearray(f.read())
mana = user.program_deploy(data)
print(mana) # HgatfFyGw2bLJeTy9HkVd4ESD6FkKu4TqMYgALsWZnE6
```

我们的空投合约在主网上的部署地址为 `HgatfFyGw2bLJeTy9HkVd4ESD6FkKu4TqMYgALsWZnE6`.

简要分析下该空投程序的涉及账户:

账户	权限	说明
用户账户	3	必须拥有足够的 sol 余额来支付创建新账户所需的费用
用户关联代币账户	1	/
程序账户	0	主网地址 `HgatfFyGw2bLJeTy9HkVd4ESD6FkKu4TqMYgALsWZnE6`
程序 pda 账户	0	主网地址 `5yAqR4gSYfs7CqpR4mgN5DNT4xczwiATuybaAa33xGip`
程序 pda 关联代币账户	1	主网地址 `G7C9Px4x1G5YE2NmUHG6BeuqavoDsQQsHGpfs7nvMcq9`
spl token mint 账户	0	主网地址 `6B1ztFd9wSm3J5zD5vmMNEKg2r85M41wZMUW7wXwvEPH`
系统程序	0	主网地址 `11111111111111111111111111111111`
原生程序: Token-2022	0	主网地址 `TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb`
原生程序: 关联代币程序	0	主网地址 `ATokenGPvbdGVxr1b2hvZbsiqW5xWH25efTNsLJA8knL`

最后, 不要忘记, 我们还需要将一些代币转账到**程序 pda 账户**. 转账操作如下:

```
import pxsol
pxsol.config.current = pxsol.config.mainnet

user = pxsol.wallet.Wallet(pxsol.core.PriKey.base58_decode('Put your private key here'))
pubkey_mint = pxsol.core.PubKey.base58_decode('6B1ztFd9wSm3J5zD5vmMNEKg2r85M41wZMUW7wXwvEPH')
pubkey_mana = pxsol.core.PubKey.base58_decode('HgatfFyGw2bLJeTy9HkVd4ESD6FkKu4TqMYgALsWZnE6')
pubkey_mana_seed = bytearray([])
pubkey_mana_auth = pubkey_mana.derive_pda(pubkey_mana_seed)
user.spl_transfer(pubkey_mint, pubkey_mana_auth, 90000000 * 10**9)
```

我们初始转入九千万 pxs 代币给空投程序. 您可以通过此页面查看空投程序当前拥有的 pxs 代币余额.

8.8 Solana/在主网发行您的代币/获取空投

Pxs 空投合约的设计允许任意调用该合约的用户获取 5 pxs 的空投, 因此我们编写脚本如下. 该脚本的功能只是简单的调用一次空投合约.

```
import argparse
import base64
import pxsol

# Apply for PXS airdrop on the mainnet.

pxsol.config.current = pxsol.config.mainnet
pxsol.config.current.log = 1

parser = argparse.ArgumentParser()
parser.add_argument('--prikey', type=str, help='private key')
args = parser.parse_args()

user = pxsol.wallet.Wallet(pxsol.core.PriKey.int_decode(int(args.prikey, 0)))
pubkey_mint = pxsol.core.PubKey.base58_decode('6B1ztFd9wSm3J5zD5vmMNEKg2r85M41wZMUW7wXwvEPH')
pubkey_mana = pxsol.core.PubKey.base58_decode('HgatfFyGw2bLJeTy9HkVd4ESD6FkKu4TqMYgALsWZnE6')
pubkey_mana_seed = bytearray([])
pubkey_mana_auth = pubkey_mana.derive_pda(pubkey_mana_seed)
pubkey_mana_spla = pxsol.wallet.Wallet.view_only(pubkey_mana_auth).spl_account(pubkey_mint)
rq = pxsol.core.Requisition(pubkey_mana, [], bytearray())
rq.account.append(pxsol.core.AccountMeta(user.pubkey, 3))
rq.account.append(pxsol.core.AccountMeta(user.spl_account(pubkey_mint), 1))
rq.account.append(pxsol.core.AccountMeta(pubkey_mana, 0))
rq.account.append(pxsol.core.AccountMeta(pubkey_mana_auth, 0))
rq.account.append(pxsol.core.AccountMeta(pubkey_mana_spla, 1))
rq.account.append(pxsol.core.AccountMeta(pubkey_mint, 0))
rq.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
rq.account.append(pxsol.core.AccountMeta(pxsol.program.Token.pubkey, 0))
rq.account.append(pxsol.core.AccountMeta(pxsol.program.AssociatedTokenAccount.pubkey, 0))
rq.data = bytearray()
tx = pxsol.core.Transaction.requisition_decode(user.pubkey, [rq])
tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
tx.sign([user.prikey])
pxsol.log.debugln(f'main: request pxs airdrop')
txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
pxsol.rpc.wait([txid])
tlog = pxsol.rpc.get_transaction(txid, {})
for e in tlog['meta']['logMessages']:
    pxsol.log.debugln(e)
pxsol.log.debugln(f'main: request pxs airdrop done')
```

您可以在 pxsol 项目的源码中找到这份脚本. 脚本已经默认配置到主网, 运行脚本后, 我们的空投合约就将发送 5 pxs 至您的账户!

```
$ git clone https://github.com/mohanson/pxsol
$ cd pxsol
$ python example/pxs_airdrop.py --prikey 0xYOUR_MAINNET_PRIVATE_KEY
```

8.9 Solana/在主网发行您的代币/获取完整源码

源码我已经打包好放上 github 啦!

如果你懒得跟着一步步敲代码(我懂你), 可以直接去看我准备好的示例项目. 地址在这儿, 不用谢我, 除非你想请我喝杯奶茶.

我知道许多开发者喜欢咖啡, 但对于我而言, 奶茶总是最好的.

```
$ git clone https://github.com/mohanson/pxsol-spl
$ cd pxsol-spl
```

您可以在本地开发网络发行代币并部署该空投合约:

```
$ python make.py deploy
# 2025/05/19 11:42:11 main: deploy mana pubkey="344HRAgWWiLuhUWTm9YNKWfhV5fWK26vx45vMxA9HyCE"
```

生成一个随机账户, 并发送空投:

```
$ python make.py genuser
# 2025/05/19 11:45:11 main: random user prikey="Dk5y9WDhMiX83VDPTfojkWgXt6KuBAYhQEgVRAKYGLYG"

$ python make.py --prikey Dk5y9WDhMiX83VDPTfojkWgXt6KuBAYhQEgVRAKYGLYG airdrop
# 2025/05/19 11:45:24 main: request spl airdrop done recv=5.0
```

9. Solana/经济系统

9.1 Solana/经济系统/导言

学生: "教授, 你知道吗, 我决定长期持有一些 sol 代币, 我对它的未来有很大的信心! 就像古话所说, 有时候, 梭哈是一种智慧."

教授: "哇, 看来你已经有了投资的眼光! 那你打算什么时候开始买更多 sol 来扩大你的持仓呢?"

学生: "其实, 我已经买了一些. 可是你知道的, 我对它的未来也有点担心. 就像那些币圈里常见的项目, 项目方可以随意增发, 就好像我前段时间发行的泡泡币, 通货膨胀严重, 一不小心就可能导致价值缩水. 我不想买了 sol 之后几年下来它变得一文不值."

教授推了推眼镜, 心想: 自从你发行了泡泡币就每天给自己增发新币, 这不归零才是奇迹.

教授: "嗯, 通胀问题确实是很多数字货币面临的挑战之一. 但是 solana 有自己的一些机制来控制这个问题. 比如 solana 的通货膨胀率是逐年下降的. 你有没有了解过它的经济模型呢?"

学生: "哦? 通胀率逐年下降? 那就是说, 随着时间的推移, sol 变得更值钱了吗? 听起来不错!"

教授: "并非如此, sol 每年仍然会增发, 但增发速度会逐年下降. 直到大约 10 年后, 会达到一个稳定的水平. 因此 solana 不像比特币, 它的供应没有上限, 仍然是一个通胀型的加密货币."

学生: "那说起来, 我无法被动的等着 solana 因通缩而升值了. 所以 solana 的价值增长是不是更多取决于它能否被广泛应用呢? 如果大规模的应用落地, sol 的价值是不是就能水涨船高? "

教授: "完全正确! 区块链的价值最终还是来源于它的应用场景. Solana 已经吸引了大量的开发者和项目, 而随着去中心化金融, nft, 以及其他创新应用的不断发展, solana 网络的使用量会不断增加."

学生: "听你这么一说, 我感觉自己买 sol 的决策更有信心了! 看来 solana 的未来可期啊! "

教授: "没错, solana 的未来非常有潜力. 但记住, 投资始终要有长期的眼光, 不要被短期的波动吓到. 现在就让我们一起深入探讨 solana 的经济模型, 看看它是如何保证其长期价值的吧!"

学生: "太好了, 期待这节课! 我准备好深入研究了!"

9.2 Solana/经济系统/典型案例分析

这一小节简要谈谈区块链上的经济系统. 对于大多数区块链项目来说, 经济系统是其根基. 系统要是设计的巧妙, 那么确实可以支持项目的长期高速发展; 要是疏忽大意了, 项目就只能化作春泥更护花了.

Steem 通胀与治理危机 2016–2020

Steem 是一个基于区块链的去中心化社交媒体平台, 用户可以因创建或策展(例如点赞, 评论)优质内容而获得加密货币奖励. Steem 每年约有 9% 的代币通胀, 其中大部分通过投票挖矿奖励内容创作者和持币者. 但是由于高通胀导致强大的抛压, 且奖励分配机制被少数大户操纵, 社区信心下降, 代币长期下跌, 活跃用户减少, 最终被 tron 创始人孙宇晨收购, 引发治理大战.

BitConnect 崩盘 2018

BitConnect 是一个高收益投资项目. 它承诺通过借贷计划和自有的交易机器人为投资者提供高达 1% 的日收益(月复利可达 40% 左右), 该机器人据称利用加密货币市场的波动性获利. 但实际上其所谓收益来源实际上是后来的用户资金, 属于庞氏经济模型, 监管介入后平台关停, 代币在几天内归零, 投资者损失惨重.

EOS 资源模型失败 2018–2021

EOS 可以说是 2018 年最受人瞩目的项目: 所有人都在期待它是不是另一个以太坊. 它设计了一个机制, 用户需质押 eos 获取计算资源才能使用链上应用. 这种设计下, 由于链上计算资源价格极度波动, 用户体验极差, 普通用户在高峰期几乎无法交易, 导致大量应用流失. 最后的结果是 eos 价格持续下行, 作者当时也是冲 eos 的一员, 最终作者的 eos 资产缩水了 100 倍.

Terra / UST 崩盘 2022

币圈有句俗语: 世界上最不稳定的东西就是算法稳定币.

Terra 是一个算法稳定币项目, 所谓的算法稳定币, 指的是通过"某种高明的算法来保证代币与美元是一一对应的". 具体来说, terra 其美元稳定币 ust 通过 luna 的铸造与销毁来维持锚定 1 美元, 依赖套利者平衡价格. 但问题在于当市场信心崩溃时, 套利机制无法阻止 luna 被无限增发, 最终进入"死亡螺旋". 作者也是有幸经历了它归零的那个瞬间, 当时正值夜里, 当第二天清晨时, 其价格已经从 100 多美元跌到接近零.

从这些案例可以看出, 区块链经济系统设计失败常见的几个坑:

- 过高通胀导致长期抛压, 价格难稳
- 激励结构单一导致用户只为奖励来, 奖励一停就走
- 过于相信市场调节, 极易在极端行情下失效
- 庞氏结构, 完全依赖新资金补贴老资金
- 最后最奇葩的死法是系统设计的没有大问题, 但用户体验太差导致没人使用

9.2.1 Solana/经济系统/概述

Solana 的经济设计核心目标是通过**通胀**, **质押**和**交易费用**三大机制, 激励**验证者节点**和**质押者**为网络提供计算, 存储和安全保障.

我们先来谈谈两个 solana 系统里的两个核心角色.

- 验证者是运行 solana 软件的服务器, 负责处理交易, 验证区块链的状态并生成新的区块. 它们是 solana 网络的工作者, 维护网络的运行. 验证者通过处理交易获得交易费用和区块奖励(来自通胀奖励). 如果有质押者委托 sol 给该验证者, 验证者还可以从质押奖励中抽取一定比例的佣金(由验证者自行设定).
- 质押者是持有 sol 代币的用户, 他们将自己的 sol 委托(delegate)给某个验证者节点, 以支持网络的去中心化和安全性. 质押者可以获取一定的质押奖励.

三个核心机制维持整个体系:

- 通胀: 用于向验证者和质押者持续发放奖励, 从而保持节点运营的积极性.
- 质押机制: 让持币者通过委托代币给验证者来参与网络安全, 同时获得收益, 防止流动性泛滥.
- 交易费用: 既是对网络资源的使用补偿, 也是对投机性攻击(如垃圾交易)的防御手段.

其实你可以看到, solana 的经济系统设计与现实中的美元/美债系统十分相似: 通过通胀释放流动性, 通过质押回收流动性, 维持一个动态平衡. 通过这些机制的协同运作, solana 追求在代币供应无限增长与代币价值稳定之间找到平衡点, 使得网络既能高速发展, 又能保持经济的可持续性.

本系列文章将带你深入了解 solana 经济系统的运作逻辑, 从通胀曲线到质押回报, 从交易费分配到长期供需关系, 帮助你全面认识这一机制.

9.3 Solana/经济系统/创世块(一)

Solana 区块链于 2020 年 3 月 16 日正式上线, 其创世块是区块链的第一个区块, 标志着网络的启动. 创世块本身不包含源代码, 而是一个初始状态的快照, 定义了网络的初始参数, 账户状态和代币分配等. Solana 的创世块可以通过区块链浏览器查看, 但如果想具体查看创世块的交易或账户需要通过节点或相关工具查询.

在 solana 里, 创世块与其说是一个块, 称它为一个配置文件似乎更为妥当.

您可以通过区块链浏览器"查询"创世块, 但除了最基本的数据外, 您无法从浏览器中得到更多有意义的信息: https://explorer.solana.com/block/0.

9.3.1 创世块代币分发

Solana 代币初始分发在网络启动时通过创世块完成. 根据 solana 白皮书, 创世块中的 sol 主要分配给了以下几类参与者:

1. 种子轮和早期投资者: 包括风险投资机构(如 a16z, multicoin capital 等)和早期支持者.
2. 团队和基金会: solana labs 团队和 solana 基金会保留了一部分代币, 用于开发和生态系统建设.
3. 社区和奖励: 部分代币分配给社区激励, 验证者奖励等.

根据**非官方来源的**的公开信息和社区讨论, solana 的初始代币分配大致如下:

1. 种子轮和私募: 约 25-30% 分配给早期投资者.
2. 团队: 约 12.5%.
3. 基金会/生态系统: 约 10-15%.
4. 社区/奖励: 剩余部分用于验证者奖励, 空投和其他社区激励.

我们谨慎的看待以上信息.

9.3.2 获取创世块数据

为了更加真实的研究 solana 的经济系统, 我决定直接分析创世块中的数据. 幸运的是, 创世块信息是**半公开**的. 您能从网络上得到创世块的数据, 但同时网络上却缺少教程来教您分析它, 也没有任何网页或图表展示它, 因此我称它为半公开. 本文将试图改变这一点.

首先我们下载创世块数据的压缩档并解压它, 获得 `genesis.bin` 文件.

```
$ mkdir ledger
$ cd ledger
$ wget https://api.mainnet-beta.solana.com/genesis.tar.bz2
$ tar -jxvf genesis.tar.bz2
$ ll

# drwxrwxr-x   2 ubuntu ubuntu   4096 Aug  8 18:24 ./
# drwxrwxrwt 101 ubuntu ubuntu  65536 Aug  8 18:28 ../
# rw-r--r--    1 ubuntu ubuntu 132347 Mar 16  2020 genesis.bin
```

该文件使用 bincode 编码了一个账户列表, 我们需要编译一个工具来分析 `genesis.bin` 文件. 下载 solana 源码库, 并编译其中的 `ledger-tool` 工具.

```
$ git clone https://github.com/anza-xyz/agave
$ cd ledger-tool
$ cargo build
```

执行编译出来的 `agave-ledger-tool` 并在参数中指定我们刚才保存 `genesis.bin` 的目录:

```
$ agave-ledger-tool genesis --ledger ledger --accounts --output json
```

我们得到了创世块中获得初始分配的所有账号列表. 该列表中共保存了 431 个账户, 在这里只截取头尾两个账户显示如下.


```
{
  "accounts": [
    {
      "pubkey": "13LeFbG6m2EP1fqCj9k66fcXsoTHMMtgr7c78AivUrYD",
      "account": {
        "lamports": 153333633390909120,
        "data": [
          "AQAAAIDVIgAAAAAAEz9W2TmIqMRgQ0rrgC89RndskM9PojDPkwTMSiLZq/lkVvBa571/bSeULSeR8aaRDAXCdHyD8RGsnvQTimS4AgAAAAAAAA
          "base64"
        ],
        "owner": "Stake11111111111111111111111111111111111111",
        "executable": false,
        "rentEpoch": 0,
        "space": 200
      }
    },
    ... ...
    {
      "pubkey": "JCo7ptMT38iZdjXXdxD8Ye79dGBBkMi2nttnv965k3pE",
      "account": {
        "lamports": 104166666268940,
        "data": [
          "AQAAAIDVIgAAAAAAHo5ufv6LXJUFQplRqRqFRQY4YFx4p81H20CZkWDY9q37k5kcYLD2AC8789vlnsUxaVMBz0/FeathVFoOL2wFSwAAAAAAAA
          "base64"
        ],
        "owner": "Stake11111111111111111111111111111111111111",
        "executable": false,
        "rentEpoch": 0,
        "space": 200
      }
    },
}
```


您可以直接通过地址 https://raw.githubusercontent.com/mohanson/pxsol/refs/heads/master/res/genesis.json 下载我预先解析好的 json 文件.

```
$ wget https://raw.githubusercontent.com/mohanson/pxsol/refs/heads/master/res/genesis.json
```

9.4 Solana/经济系统/创世块(二)

我们使用下面的代码汇总一下创世块中所有账号的余额, 得到 sol 在创世块中的初始发行总额是 500000000 sol.

```
import json
import pathlib
import pxsol

genesis = json.loads(pathlib.Path('genesis.json').read_text())
accounts = genesis['accounts']

lamports = sum([e['account']['lamports'] for e in accounts])
print(lamports / pxsol.denomination.sol) # 500000000.0
```

之后, 我们通过 solscan.io 查询当日 sol 的总供应量. 得知截至 2025 年 9 月 2 日, sol 的总供应量为 608757550 枚(包含流通供应量 540915225 枚以及非流通供应量). 也就是说, 在经历 5.5 年的运营之后, 市面上流通的绝大部分 sol (500000000 / 540915225 = 92.4%) 仍然都来自创世块. 根据这个信息我们可以至少得到两个有用的分析:

- 现在流通中的 sol 几乎都来自创世块 431 个账号的卖出.
- 通胀率极低, 根据公式 `(540915225 / 500000000) ** (1 / 5.5)` 计算得到真实年通胀率为 1.44%.

Solana 社区中有一些批评认为 solana 的初始分发中代币分配较为集中, 早期投资者和团队持有了绝大部分份额. 我们使用代码来验证一下这个说法是否属实.

```
import json
import pathlib

genesis = json.loads(pathlib.Path('genesis.json').read_text())
accounts = genesis['accounts']
accounts.sort(key=lambda x: -x['account']['lamports'])

lamports_total = sum([e['account']['lamports'] for e in accounts])
lamports_top10 = sum([e['account']['lamports'] for e in accounts[:10]])
lamports_top20 = sum([e['account']['lamports'] for e in accounts[:20]])
lamports_top50 = sum([e['account']['lamports'] for e in accounts[:50]])

print(lamports_top10 / lamports_total) # 0.87
print(lamports_top20 / lamports_total) # 0.92
print(lamports_top50 / lamports_total) # 0.94
```

得到分析结果如下:

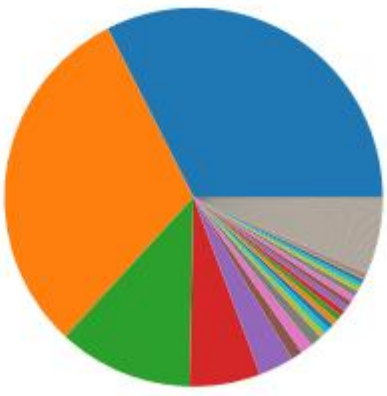
- 创世块中的前 10 个地址占有了 87% 的总初始分发额.
- 创世块中的前 20 个地址占有了 92% 的总初始分发额.
- 创世块中的前 50 个地址占有了 94% 的总初始分发额.

看起来社区中的批评还是比较中肯的. 这里给出前 10 地址的浏览器链接. 我们相信这些地址应当大多数来自现实世界中的风险投资机构和 solana 基金会/开发者团队, 但本文无意真的去分析这些账户的现实主体, 因此在文章中仅仅简单列出.

- https://solscan.io/account/APnSR52EC1eH676m7qTBHUJ1nrGpHYpV7XKPxgRDD8gX
- https://solscan.io/account/13LeFbG6m2EP1fqCj9k66fcXsoTHMMtgr7c78AivUrYD
- https://solscan.io/account/GK2zqSsXLA2rwVZk347RYhh6jJpRsCA69FjLW93ZGi3B
- https://solscan.io/account/8HVqyX9jebh31Q9Hp8t5sMVJs665979ZeEr3eCfzitUe
- https://solscan.io/account/HbZ5FfmKWNHC7uwk6TF1hVi6TCs7dtYfdjEcuPGgzFAg
- https://solscan.io/account/14FUT96s9swbmH7ZjpDvfEDywnAYy9zaNhv4xvezySGu
- https://solscan.io/account/9huDUZfxoJ7wGMTffUE7vh1xePqef7gyrLJu9NApncqA
- https://solscan.io/account/C7C8odR8oashR5Feyrq2tJKaXL18id1dSj2zbkDGL2C2
- https://solscan.io/account/AYgECURrvuX6GtFe4tX7aAj87Xc5r5Znx96ntNk1nCv
- https://solscan.io/account/AogcwQ1ubM76EPMhSD5cw1ES4W5econvQCFmBL6nTW1

最后绘制一个饼图来直观显示创世块中各地址的份额比例.

```
import json
import matplotlib.pyplot as plt
import pathlib

plt.style.use('seaborn-v0_8-darkgrid')
plt.figure(figsize=(4.8, 2.7))

genesis = json.loads(pathlib.Path('genesis.json').read_text())
accounts = genesis['accounts']
accounts.sort(key=lambda x: -x['account']['lamports'])

x = [e['account']['lamports'] for e in accounts]

plt.pie(x)
plt.axis('equal')
plt.show()
```



注意上图中灰色的区域是排名在后的几百个账户的叠加显示, 并非代表一个账户.

9.5 Solana/经济系统/通胀奖励

按照最初设计, solana 网络通过预定的通胀机制来发行新代币, 这些新代币按比例分配给基金会, 验证者和委托者. 通胀率会从最初的 8% 开始, 每年以 15% 的速度递减, 目标是长期稳定在 1.5%. Solana 基金会固定得到通胀的 5% 金额, 直到持续领取达 7 年时间.

可以通过 rpc 实时查询 solana 的当前通胀率. 在当前时间点来说, 其通胀率为 `0.04317705363505517`. 另外您可以发现 `foundation` 的值是 `0.0`, 也就是说当前基金会已经不再默认获得 5% 的通胀, 即使时间还未满 7 年.

```
import pxsol

pxsol.config.current = pxsol.config.mainnet
print(pxsol.rpc.get_inflation_rate())

# {
#   'epoch': 842,
#   'foundation': 0.0,
#   'total': 0.04317705363505517,
#   'validator': 0.04317705363505517,
# }
```

您可以在 https://github.com/anza-xyz/solana-sdk/blob/9decd857f019cc4c8dd89f4b3811ea56b0ac5c8e/inflation/src/lib.rs#L30-L34 找到关于 solana 通胀率的参数. 但遗憾的是, 我目前没有找到 solana 基金会奖励是什么时候从 5% 改为 0% 的具体信息, 但我猜测应该是在 solana 验证者社区投票后实施的. 具体时间可能是在相关论坛, 网页建立之前进行的, 例如 2021 或 2022 年的某次投票.

Solana 的通胀率基于一个固定的初始通胀率, 每年以一定比例递减, 直到达到长期目标通胀率. 具体公式如下:

$$I_t = \max(I_0 \times (1-r)^t, I_e)$$

其中:

- I_t: 第 t 年的通胀率.
- I_0: 初始通胀率(8%)
- I_e: 目标通胀率(1.5%)
- r: 年度递减率(15%)
- t: 时间(以年为单位, 从通胀机制启动开始计算)

通胀率决定了每年新增的 sol 代币数量, 以当前流通供应量为基准, 乘以通胀率就是当年的新增代币数量.

例: 假设 2025 年总供应量为 6 亿 sol, 通胀率 I_t=4.3%, 问当年预估新增代币数量.

答: `600000000 × 0.043 = 25800000`.

要注意, solana 的通胀是依据总供应量来计算的, 而非流通供应量. 这意味着即使有部分代币被锁定或未流通, 它们仍然会被计入通胀计算中. 这有助于确保网络的长期稳定性和安全性, 因为更多的代币被分配给验证者和质押者, 以激励他们维护网络. 但与此同时, 这也会使得通胀的实际影响更为复杂, 因为流通中的代币数量可能与总供应量存在显著差异.

另外在实际操作中, solana 其实并非按照现实中的"年"来计算通胀率和分配奖励. Solana 的基础时间单位是 epoch, 一个 epoch 通常持续约 2-3 天(多数情况下您可以粗略认为是 2 天), 每个 epoch 包含一定数量的 slot(时间槽), 通常约为 432000 个 slot. 通胀奖励会在每个 epoch 结束时分发给验证者和质押者.

9.6 Solana/经济系统/手续费与手续费燃烧

Solana 的手续费体系采用两层结构, 主要由基础费用和优先费用组成, 辅以存储租金机制, 以支持网络的高效运行和抗垃圾交易能力.

9.6.1 基础费用

每笔 solana 交易都需要支付基础费用, 用于补偿验证者处理交易所需的计算资源. 基础费用固定为每个签名 5000 lamports. 以 2025 年 sol 价格约为 200 美元为例, 5000 lamports 相当于约 0.001 美元, 即千分之一美元. 这种固定费率设计使得 solana 的交易成本在网络拥堵时仍保持稳定, 与以太坊等基于拍卖的费用模型形成鲜明对比.

Solana 的费用燃烧机制是其经济模型的重要组成部分, 旨在通过减少 sol 的流通供应来支持代币的长期价值. 基础费用的 50% (即2500 lamports/签名)会被燃烧, 永久移除出流通供应, 剩余 50% 分配给处理交易的验证者.

在本文写作时, 根据 solana compass 上的数据, 在过去 24 小时时间里, 用户总计支付了 1675.48 sol 的基础费用, 也就是说有 837.74 sol 被同时燃烧销毁. 按此估算, 年度燃烧量约为 305775 sol. 若以 2025 年 sol 价格 200 美元计算, 年度燃烧价值约为 61155000 美元.

这种燃烧机制类似于以太坊的 eip-1559, 通过减少代币供应来增强 sol 作为价值存储的潜力.

9.6.2 优先费用

优先费用是可选的附加费用, 用户可以通过支付额外的 sol 来提高交易被当前领导者(即验证者)优先处理的概率. 优先费用的计算公式为 `计算单元限制 x 计算单元价格` .

其中, 计算单元限制是交易可使用的最大计算资源, 上限为 140 万个计算单元, 默认 200000 个. 计算单元价格由用户指定. 根据 SIMD-0096, 优先费用完全由处理交易的验证者收取, 不进行燃烧.

在本文写作时, 根据 solana compass 上的数据, 在过去 24 小时时间里, 用户总计支付了 3268.76 sol 的优先费用, 且 66.42% 的交易用户都支付了优先费用.

9.6.3 存储租金

在 solana上, 存储账户数据需要支付租金, 费用与账户占用空间成正比. 租金费用在账户关闭时可退还, 旨在激励用户清理不必要的链上数据. 50% 的租金费用会被燃烧, 剩余 50% 分配给验证者.

租金费用相当于将一半 sol 永久移出流动池, 并将剩下一半 sol **暂时**移出流动池, 除非用户决定删除自己的账户, 否则这部分租金将永远不会参与流通. 根据 coinlaw 的报告, solana 目前每周新增约 200000 个钱包, 平均每日新增约 28571 个账户.

> More than 200,000 new wallets are created on Solana each week, indicating robust organic adoption.

假设一个账户的平均数据大小为 165 字节(spl token 账户的典型大小), 那么使用以下代码估算下每日新增租金消耗约为 83 sol.

```
import pxsol

pxsol.config.current = pxsol.config.mainnet

lamport = pxsol.rpc.get_minimum_balance_for_rent_exemption(128 + 165, {}) * 28571
sol = lamport / pxsol.denomination.sol
print(sol) # 83.71760136
```

不过事实上, 如果一名开发者要在 solana 网络上部署程序账户, 需要支付的租金要远远大于一个普通用户账户. 因此这个 83 sol 数据是极端低估的.

9.6.4 总结

Solana 的手续费规则通过固定基础费用, 可选优先费用和存储租金的组合, 实现了低成本, 可预测和高效率的交易体验. 理论上来说, 通过手续费燃烧机制, 可以有效减少 sol 的流通供应, 增强代币的经济价值. 然而, 以实际数据来说, 被燃烧掉的 sol 只占每年通胀增发的极小部分(数据参考上一章节), 也就是说想依靠手续费燃烧扭通胀为通缩, 短期看来是不可能的.

9.7 Solana/经济系统/验证者的成本和预期收益

成为 solana 验证者是参与 solana 网络的重要方式. 验证者可以赚取 solana 通胀增发的代币, 以及收取用户交易中附加的优先手续费. 在本文中, 我们将讨论成为一个 solana 验证者所涉及的成本以及预期的收益.

9.7.1 验证者的职责

Solana 采用历史证明共识机制. 您也许知道"比特币矿工"在维护比特币节点的运行, 在 solana 的网络里, 这个维护节点运行的角色被称为"验证者". 验证者在 solana 网络中的主要职责包括:

- 验证交易: 验证者检查并确认交易和区块的有效性.
- 生成区块: 验证者参与投票达成共识, 生成新区块.
- 保障网络安全: 验证者通过参与共识, 确保网络的完整性, 防止恶意篡改.

9.7.2 验证者的硬件开销

在本文编写期间, solana 网络共有 1029 名验证者, 您可以查看此页面来实时跟踪验证者的数量. 成为 solana 验证者需要一些技术准备和硬件配置.

针对技术细节, 官方的详细教程位于 https://docs.anza.xyz/operations/, 但本文中我们主要关心验证者的相关经济行为, 因此我们首先评估一下要成为验证者大概需要多少投入. 根据官方说法, 验证者的硬件要求为:

- 高速互联网连接: 至少 100 Mbps 或更高的网络带宽, 以处理大量交易.
- 可靠的电力供应: 保证服务器 24/7 在线.
- 处理器: 12 核及以上, 至少 3.0 GHz
- 内存: 256 GB 或更多
- 硬盘: NVMe SSD, 至少 1TB 存储

这个配置要求已经远远超过普通家用电脑, 另外 solana 网络对节点在线也有要求: 如果一个节点掉线:

- Solana 的验证者通过投票来确认区块并获得奖励. 如果节点掉线或无法及时投票, 验证者将错过相应的投票奖励.
- 验证者的质押收益与节点的活跃度和表现直接相关. 掉线会导致节点的投票成功率下降, 从而减少分配给该节点的质押奖励.
- Solana 网络的验证者排名和选择机制部分依赖于节点的性能指标, 频繁掉线的节点可能会在排名中下降, 影响其吸引更多质押的能力. 这对验证者的长期收益和网络地位有间接影响.

我们可以得出一个近似正确的结论. 要成为 solana 验证者, 我们只有两种选择: 自建机房, 或购买云服务器. 我很难评估自建机房需要的花销, 因此这里只考虑购买云服务器的方式.

根据硬件要求, 亚马逊上适合运行 solana 验证者节点的实例类型, 比较合适且低价的是 `m8g.16xlarge` 这一款, 其价格约 $3.15955 每小时, 或者 $1498.78 每月. 以上的价格并未包含存储. Solana 验证者需要高速 NVMe SSD 存储, 推荐至少 1TB, 但实际可能需要 2TB 或更多. 亚马逊的 EBS(Elastic Block Store) 存储费用如下: gp3 卷(通用 SSD) 每 GB 每月 $0.08. 那么 2TB(2000GB)存储就需要 2000 × $0.08 = $160/月. 如果需要更高的吞吐量(如 io2 卷), 费用可能增加到 $200-$300/月.

Solana 验证者节点的数据传输量非常大, 平均每月可能达到 60-100TB(主要是出站流量, 入站流量通常免费). 亚马逊的数据传输费用通常每 GB $0.09(美国地区). 假设每月 80TB 出站流量, 那么流量费用是 80000 × $0.09 = $7200. 带宽成本是运行 solana 验证者节点的主要开销, 亚马逊的流量费用在全球角度上来讲是比较便宜的, 如果在中国大陆, 每 GB 的流量费用可能是亚马逊的数倍甚至数十倍.

至此, 我们估算总硬件成本为 $9000 每月, 且这个花销仅仅是成为 solana 验证者的最低配置需求.

9.7.3 验证者的投票费用

Solana 验证者有项重要工作就是投票, 且验证者需要为每笔投票交易支付费用, 每个 epoch(约 2-3 天)投票成本约为 2.16 sol(每个投票交易花费手续费 0.000005 sol, 432000 slots/epoch). 以 sol 价格 $200 计算: 每日约 1.1 sol × $200 = $220/天, 折合每月大概 $4950.

Solana 基金会委托计划(SFDP)可能为新验证者覆盖第一年的部分投票费用(第一季度报销 100%, 之后逐渐减少), 这可以显著降低初期成本.

9.7.4 质押

Solana 网络本身对验证者的最低质押数量没有严格规定, 但为了获得投票权并有效参与网络, 验证者通常需要质押一定数量的 sol. 一般来说, 质押的 sol 数量会影响验证者的投票权重, 进而影响其被选中验证交易的概率. 而验证者只有被选中才能获得通胀奖励. 因此为了收支平衡, 验证者必须质押足够多数量的 sol 来提高自己被选中的几率, 以获得足够的通胀奖励来覆盖掉硬件成本和投票费用.

不过幸运的是, 验证者不仅可以质押自己的 sol, 还可以接受其他用户的委托质押. 这意味着即使你自己没有大量 sol, 也可以通过吸引其他人的质押来增加你的验证者节点的总质押量.

Solana 通胀奖励在每个 epoch 结束时根据验证者的质押权重和投票表现分配给验证者及其委托者. 当前(截至 2025 年 9 月), solana 的年化通胀率约为 4.4%, 折合每个月的通胀率为 0.366%. 当前 solana 总供应量为 6.08 亿, 也就是每月约新增 2228052 个 sol. 要覆盖硬件开销和投票费用, 验证者大约需要获得 75 个 sol, 也就是占通胀新增代币的 `75 / 2228052 = 0.00336%`. 当前 solana 网络里总计有 4 亿 sol 在质押, 因此验证者至少需要质押(或者委托质押) `400000000 * 0.0000336 = 13440` 个 sol.

我们打开 https://solscan.io/validator, 分析所有验证者的质押数量, 统计显示大约有 80% 的验证者质押数量高于这个数字. 因此我估计, **80% 的 solana 验证者是盈利的, 剩下 20% 的验证者目前收支平衡甚至可能处于亏损运营**. 关于部分验证者正在亏损这个观点, 有一些其它数据可以佐证, 最直观的证据就是验证者数量在逐渐减少, 过去一年里, 验证者数量从 1300 多减少到现在的 1029 个, 恰好是在一年的时间了淘汰掉了大约 20% 的验证者.

注意有一部分的验证者收益目前还没有统计, 也就是手续费收入. 但我们在前面的文章中已经证明了手续费占通胀的比例极小, 因此我在计算验证者的收益时直接忽略了这部分. 是否将这部分收益纳入统计不会大幅的影响上文的结论.

9.7.5 结论

与比特币矿工类似, solana 验证者也是强者恒强的. 不同的是, 比特币比拼的是算力, 而 solana 比拼的是 sol 的质押数量. 这两者都是一个完全充分竞争的市场. 过去十几年, 比特币的算力越来越集中到几家大型矿场, 目前 solana 的质押也开始集中到少数几个大型验证者节点. 这种集中化到底是否是一个理想状态, 仍然值得进一步观察.

对于希望支持 solana 网络并获得奖励的用户来说, 成为验证者是一种有吸引力的方式, 但同时也意味着您需要投入巨大的初始资金: $15000/月的硬件成本以及至少 13440 个 sol 的质押. 在后文中, 我们会尝试使用一种替代方法, 让您无需付出如此巨大的成本也能获取 solana 的通胀奖励.

9.8 Solana/经济系统/质押

对于无力独自承担验证者节点运行成本的用户, solana 提供了质押机制, 为持币者提供获取通胀收入的机会.

9.8.1 什么是质押

所谓质押, 本质上是一种权益委托. 由于运行验证者节点对普通用户来说难度较高, 许多用户会选择将手中的 sol 委托给某个验证者. 验证者获得通胀奖励后, 会根据用户所持有的"权益证明"将奖励分发给用户.

9.8.2 技术实现

如前所述, 验证者只需记录用户权益并分配奖励, 因此并没有"官方"的质押程序. 不同验证者可以自行开发质押程序, 并采用不同的底层技术实现.

一种最直观的做法是, 用户将 sol 存入某个链上程序, 程序记录用户的账户地址和存入的 sol. 当验证者获得奖励后, 根据链上数据按比例将奖励转账给用户. 这种方式效率较低, 但在区块链早期的矿池中曾被广泛采用. 另一个问题是转账需要消耗大量手续费, 可能用户的收益还不够支付一次转账的费用.

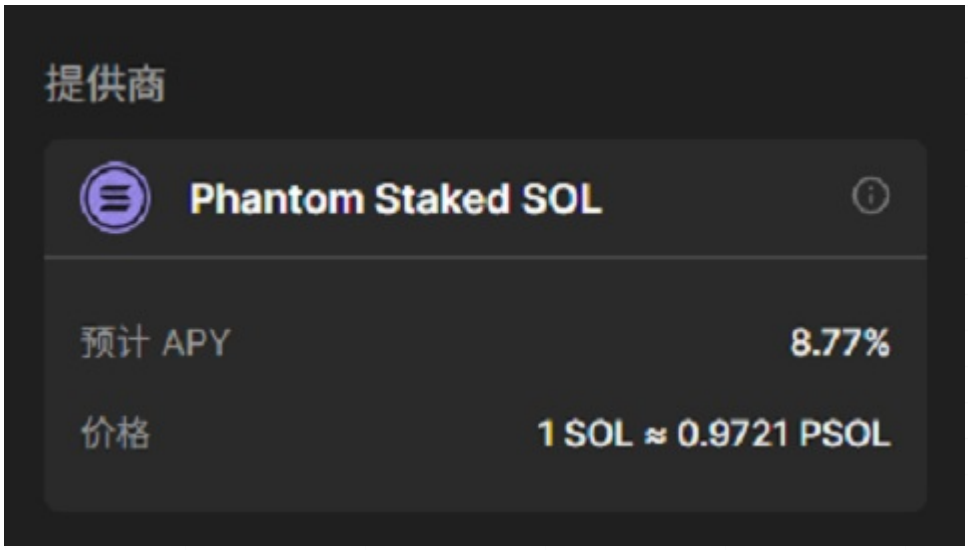
目前主流做法是: 当用户存入 sol 时, 链上程序会发放一定数量的"权益代币". 当验证者获得通胀奖励时, 会将获得的 sol 存入链上程序的权益池, 这会使用户手中的权益代币价格上升. 当用户准备赎回质押资产时, 只需卖出手中的权益代币, 即可取回 sol 本金及利息. 用户不仅可以将权益代币卖给链上程序, 还可以转账或出售给其他用户. 这种玩法其实就是目前世界上最受人关注的金融创新: 权益代币化.

9.8.3 选择质押池

Solana 支持多种钱包, 大多数钱包都支持质押功能. 常见的钱包包括:

- Phantom: 非常受欢迎的 solana 钱包, 支持质押和管理 sol 代币.
- Sollet: 一个开源的 solana 钱包.
- Solflare: 另一款常用的 solana 钱包, 支持质押和代币管理.
- ...

一般来说, 您只需使用上述钱包, 即可在主界面找到质押入口. 下图为 phantom 钱包的质押页面, 显示预期年化收益率为 8.77%. 但作者对该数据有所疑问, 因为目前 solana 的名义通胀率约为 4.3%, 全网质押率约 65%, 理论上质押者能获得的最大通胀收益率为 `4.3% / 0.65 = 6.61%`. 即使加上 phantom 的额外奖励(如手续费, 优先手续费和交易重排序奖励, 即 MEV), 粗略估算最高也只能达到 7%, 很难达到 8.77%.



> MEV(最大可提取价值)是指矿工, 验证者或其他区块链参与者通过重新排序, 包含或排除区块内交易, 从而在区块奖励和交易费用之外获得的额外利润. 最初称为"矿工可提取价值"(Miner Extractable Value), 随着其在不同区块链生态系统中的广泛应用, 演变为"最大可提取价值".

事实上, 我查阅了目前几个最大质押池的收益率, 发现大多数都在 6%~7% 左右. 截至 2025 年 9 月 2 日, 部分大池数据如下:

- Binance: 约 5.88%
- Jito: 约 6.73%
- Jpool: 约 9%, 该池表现远超平均值
- Marinade Finance: 约 7.04%
- Phantom: 约 8.77%
- Solblaze: 约 7.02%
- ... ...

目前我尚未找到 phantom 钱包超出常规的收益率的合理解释, 有机会会尝试详细分析 8.77% 这一数字, 但目前还是专注于本文内容.

9.8.4 分散风险

为降低风险, 建议将 sol 分散质押到多个验证者池. 这不仅可以减少单一验证者节点出现问题时的风险, 还能通过不同池的奖励最小化收益波动.

请注意, 您质押资产的安全性完全取决于验证者提供的质押程序. 质押程序可能存在漏洞, 甚至被恶意植入后门, 因此选择有良好声誉和高透明度的验证者非常重要.

如果您对 solana 质押感兴趣, 建议先小规模试水, 逐步积累经验, 再逐步扩大质押规模. 在区块链网络中, 谨慎行事总是明智的.

9.9 Solana/经济系统/社区治理中的争议

这一篇文章我想主要讨论一些偏探讨性质的问题. 这些问题不会有标准答案.

Solana 的经济系统在 solana 高速发展的过程中, 逐渐暴露出一些结构性问题, 尤其是在通胀机制和质押激励方面. 本文将深入分析 solana 经济系统中的主要争议点, 特别是围绕 simd-0228 提案的讨论, 以及社区在通胀率和质押比例方面的治理动态.

9.9.1 核心争议

高通胀对非质押者的稀释效应

Solana 目前的名义通胀率是 4.3%, 但由于质押率只有 65%, 因此实际上质押者的实际年化收益率大约是 6.61%, 这意味着非质押者的持币价值每年被稀释约 6.61%. 这种高通胀率对非质押者构成了显著的经济压力, 可能影响他们继续持有 sol 的意愿.

65% 的质押率其实并不健康. 三分之二的 sol 持有者都质押了资产以获取利息收益, 这使得市场上流通的 sol 数量大幅减少, 导致市场结构变得脆弱.

验证者对通胀奖励的依赖

验证者的收益几乎全部来自通胀奖励, 而非网络正常运行产生的交易手续费. 目前来看, 我认为验证者对通胀奖励的依赖性还在增加. 随着未来质押率上升和通胀率下降, 可能导致验证者收入减少, 进而验证者数量下降, 影响网络的安全性和去中心化程度.

9.9.2 被否决的 SIMD-0228 提案: 动态通胀机制

为了解决上述问题, solana 社区在 2025 年 3 月提出了 simd-0228 提案, 旨在引入动态通胀机制.

该提案是 solana 历史上参与率最高的一次投票, 接近代币总供应量的 50%, 但最终投票结果否定了该提案, 反对票以微弱优势获胜.

该提案由 multicoin capital 的 tushar jain 和 vishal kankani 主导, 支持者包括 anza 的首席经济学家 max resnick. 由于该提案未能通过, 因此您只能在 github 的 pull request 中找到该提案的具体内容: https://github.com/solana-foundation/solana-improvement-documents/pull/228.

该提案核心内容是将 sol 的年通胀率与质押率挂钩:

- 当质押率超过 50% 时, 通胀率降低;
- 当质押率低于 50% 时, 通胀率提高.

尽管 simd-0228 提案旨在优化 solana 的经济模型, 但社区内存在不同的意见. 支持者观点认为认为动态通胀机制可以降低 sol 的通胀率, 提升其价值稳定性, 吸引更多的 defi 资金流入. 而反对者观点认为基于市场的反馈调节机制, 可能产生"死亡螺旋", 也就是形成"通胀-抛售-更通胀"的恶性循环, 最终影响网络的安全性.

9.9.3 社区治理中投票权的不平等

由 simd-0228 提案引发的讨论, 标志着 solana 社区在经济治理方面迈出了重要一步. 尽管提案最终未通过, 但其引发的讨论促使社区更加关注通胀机制, 质押激励和网络安全之间的平衡. 但是, 在跟踪这一事件时, 作者也发现了社区治理中同样存在的结构性问题.

在 solana 的社区治理模式中, 投票权分配并不平等. 质押者只能通过其委托的验证者参与投票, 而验证者则代表其质押者行使投票权. 验证者投票的权重通常与他们质押的 sol 数量成正比, 质押者通过将他们的 sol 委托给验证者来间接参与投票. 这种机制导致了以下问题:

- 绝大多数质押者只关心质押收益率, 并不关心所委托验证者对某个提案的看法. 因此在重要投票前, 验证者可以通过"补贴"的形式人为抬高质押收益率, 以此贿赂质押者, 来吸引更多资金来扩充自己的投票权. 甚至更加过分的是, 部分验证者在重要投票前不与质押者沟通, 而是自行决定投票方向. 例如在关于 simd-0228 的投票中, 就有多个头部验证者在没有与质押者有任何沟通的情况下, 就直接投了赞成票/反对票.
- 大户持有大量代币, 掌握过多投票权, 影响项目发展方向. 对多数人而言, 只关注自身利益, 而忽视网络整体健康发展.

9.9.4 不是结束

虽然存在诸多争议和不完美, 但这正是 solana 不断成长和适应市场变化的关键因素. 这个世界上最可怕的不是有反对意见, 而是没有反对意见. 通过持续的讨论和辩论, 社区能够识别问题, 探索解决方案, 并推动项目向前发展. 这次围绕 simd-0228 的讨论, 只是 solana 经济治理旅程中的一个章节.

未来, 我希望看到 solana 社区在治理机制上进行更多的创新, 以确保所有持币者的声音都能被公平地听到. 这可能包括引入更透明的投票过程, 提高质押者的参与度, 以及探索新的经济激励机制.

10. Solana/更多开发者工具

10.1 Solana/更多开发者工具/导言

期末前的倒数第二节课, 教室里阳光有些慵懒. 学生把笔在指间转得飞快, 人却在发呆.

学生: "教授, 我写了点合约, 也会发交易了. 我想我对 solana 网络有个基本的了解了. 但我总觉得缺点什么?"

教授: "你的疑惑是正常的, 因为你缺少对 solana 社区工具的了解. 你不好奇吗? 现在 solana 开发团队在做什么, 社区里的其它人在做什么, 社区里还有哪些好用的工具?"

学生: "先说 anchor 吧, 我总听人说它是 solana 链上程序开发的神器?"

教授: "没错! Anchor 是为 solana rust 合约准备的开发框架. 它提供了账户校验, 序列化, idl, 测试框架等一整套工具, 极大降低了心智负担. 用 anchor, 你能更清晰地描述程序接口, 自动生成客户端, 写测试也像搭乐高一样. 如果你要写链上程序, anchor 是首选."

学生: "那如果我想实现一个前端网站要跟合约交互, 该用谁?"

教授: "这就轮到 solana/web3.js 登场了. 它是浏览器和 node.js 里最常用的 sdk, 和钱包生态(如 phantom)贴合紧密. 创建交易, 签名, 发送, 订阅事件, 前端一条龙."

学生: "我还想要偷窥, 不, 是查询链上数据, 该用谁?"

教授: "那就是 solana-py. 它是 python 社区的主流 sdk, 适合写运维脚本, 数据处理, 离线任务, 也能发交易, 查账户, 跑批量任务. 和数据科学工具链亲和力很强. 如果你要把区块链数据丢进 pandas 里揉一揉, python 会让你事半功倍. 当然, 我更加建议你使用 pxsol, 因为它是本书作者亲手打造的, 专门为 pythoner 而生的 sdk."

学生: "听起来三位各司其职."

教授: "还有各种区块浏览器与 rpc 服务: 快速确认交易与日志, 排查问题就靠它们."

学生: "这么多工具, 我怕学起来会很难."

教授: "记住: 工具的价值不在于完全学会如何使用, 而在当你想解决问题时, 你知道该用谁."

学生举起拳头比了个加油的手势.

学生: "谢谢教授!"

教授: "去做点有趣的东西吧. 下次上课, 你来给我演示."

10.2 Solana/更多开发者工具/Anchor 环境搭建

在本书的中篇内容里, 我们将第一个想法带上 solana, 我们编写了一个可以存储任意数据的简单数据存储程序. 我们使用原生 rust 编写了这个程序, 但过程中需要我们直面账户校验, 序列化和客户端打包这些杂事, 往往会消磨兴致. Anchor 出场的意义, 正是把这些粗重活接过去, 让你把精力放在要实现什么, 而不是怎么让代码"跑"起来.

Anchor 是一种为 solana 区块链设计的开发框架, 用于快速, 安全地构建和部署链上程序. 它通过提供工具和抽象来简化开发流程, 包括自动处理账户和指令数据的序列化, 内置安全检查, 生成客户端库以及提供测试工具.

我们在这里用 Anchor 重写那个数据存储程序, 让你体会它的魔力. 我们不会在这里做说明书式的工具介绍, 如果您需要它, 请参考官方文档. 我们只会准备一张干净的工作台来组装代码, 让你专注于实现核心功能. 你会看到 anchor 的核心心智模型, 完成一次从零到一的本地运行, 并学会辨认路上的几个小坑.

10.2.1 环境搭建

如果你的机器还没有这些工具, 请先安好: rust, solana cli, node.js 与 yarn, 以及 anchor 本体. 下面的命令可以直接复用; 若你已有其一, 可跳过相应小节.

安装 anchor(使用 avm 管理版本):

```
$ cargo install --git https://github.com/coral-xyz/anchor avm --locked --force
$ avm install latest
$ avm use latest
$ anchor --version
```

准备 solana cli 与本地链:

```
$  sh -c "$(curl -sSfL https://release.solana.com/stable/install)"
$ solana --version
$ solana config set --url http://127.0.0.1:8899
$ solana-test-validator -r
```

准备 node.js 与 yarn, 因为 anchor 的测试与客户端默认使用 ts:

```
$ npm install -g yarn
```

本章节的配套代码在这里. 如果你正在阅读该配套仓库, `Anchor.toml` 已预设本地网络与钱包路径, `tests/` 里也放好了 ts 的测试脚本. 进入仓库根目录, 装上依赖即可:

```
$ yarn install
```

小提示: 第一次跑本地链时, 别忘了给默认钱包要点启动资金.

```
$ solana airdrop 2
```

10.2.2 创建项目

我们先使用 anchor 搭一个最小可用的程序, 看看它长什么样.

```
$ anchor init pxsol-ss-anchor
$ cd pxsol-ss-anchor
```

脚手架会生成一套目录:

- `programs/<name>/src/lib.rs` 是合约入口. 你会看到 `#[program]` 模块和一两个演示方法.
- `Anchor.toml` 是配置中心, 记录 program id, 要连接的集群, 测试脚本等.
- `tests/` 放着 ts 测试, 等会儿它会代表客户端来"按按钮".

先试着构建它:

```
$ anchor build
```

如果你还没启动本地链, 开一个终端让验证器常驻:

```
$ solana-test-validator -r
```

接着跑一次测试:

```
$ anchor test --skip-local-validator
```

这条命令做了三件事:

1. 构建 rust 程序
2. 把它部署到本地链
3. 运行 `tests/` 下的 ts 测试用例

10.2.3 如何开始

当我们开始实现真正的业务, 可以沿着这条最小路径前进:

1. 在 `programs/<name>/src/lib.rs` 里新增一个方法, 先写出期望的 accounts 结构与约束
2. 在 `tests/` 写一个最小的调用脚本, 跑 `anchor test` 观察失败信息
3. 循环填写逻辑代码, 补齐账户, 空间与权限, 并时刻调整测试脚本, 直到测试通过
4. 最后接入前端或后端服务

当你跨过这些门槛, anchor 就会像一把顺手的扳手. 你不用每天都去记 torx 和内六角的尺寸, 只管拧紧你真正关心的那颗螺丝.

10.3 Solana/更多开发者工具/Anchor 里的简单数据存储合约

本章节的配套代码在这里.

这一节, 我们用 anchor 实现一个数据存储合约, 走一遍从建模到程序构建的过程. 你会看到三个关键点: 账户心智模型, 两条指令(init/update), 以及动态重分配与租金的细枝末节. 代码出自 `programs/pxsol-ss-anchor/src/lib.rs`, 但我们以文字的方式来理解它.

10.3.1 数据存储格式设计

我们知道用户数据实际上是存储在 pda 程序扩展账户里的. 在我们使用原生 rust 编写该程序时, 我们其实并没有对 pda 账户里的数据格式做过多约束, 只要能序列化与反序列化就行. 但在 anchor 里, 我们可以定义一个结构体来描述它, 并用 `#[account]` 标记它. 这种做法可以方便我们后续的开发, 也方便我们对链上数据的分析和理解.

```
#[account]
pub struct Data {
    pub auth: Pubkey, // The owner of this pda account
    pub bump: u8,     // The bump to generate the PDA
    pub data: Vec<u8> // The content, arbitrary bytes
}

impl Data {
    pub fn space_for(data_len: usize) -> usize {
        // 8 (discriminator) + 32 (auth) + 1 (bump) + 4 (vec len) + data_len
        8 + 32 + 1 + 4 + data_len
    }
}
```

方法 `space_for()` 用来计算账户所需空间. 这里的空间由五部分组成. 我们需要使用该函数来计算账户的租赁豁免金额.

10.3.2 指令: 初始化程序扩展账户

我们设计了两条指令: `init` 和 `update`. `init` 用来初始化程序扩展账户, `update` 用来更新内容. 下面我们逐一分析它们的实现.

指令 init 做了三件事: 记住谁是拥有者, 记录 bump, 并把内容置空.

```
pub fn init(ctx: Context<Init>) -> Result<()> {
    let account_user = &ctx.accounts.user;
    let account_user_pda = &mut ctx.accounts.user_pda;
    account_user_pda.auth = account_user.key();
    account_user_pda.bump = ctx.bumps.user_pda;
    account_user_pda.data = Vec::new();
    Ok(())
}
```

账户约束里, `init` 会在第一次调用时分配账户与租金, 由拥有者 `payer = user` 支付:

```
#[derive(Accounts)]
pub struct Init<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(
        init,
        payer = user,
        seeds = [SEED, user.key().as_ref()],
        bump,
        space = Data::space_for(0)
    )]
```

```
    pub user_pda: Account<'info, Data>,
    pub system_program: Program<'info, System>,
}
```

此时程序扩展账户里的数据字段是空的, 但已具备完整身份与归属, 并达成了租赁豁免.

10.3.3 指令: 存储或更新数据

更新内容时, 我们允许程序扩展账户变大或变小. 变大需要补齐租金, 变小则把多出来的 lamports 退给拥有者. 逻辑可以读作三步: 验权, 重分配, 找零. Anchor 框架会帮我们处理租赁豁免与扣费, 但找零需要我们自己来做. 也就是当新数据比老数据大时, 我们不需要做什么, anchor 会自动帮我们补齐租赁豁免资金; 但当新数据比老数据小时, 我们要把多出来的 lamports 退给拥有者.

```
pub fn update(ctx: Context<Update>, data: Vec<u8>) -> Result<()> {
    let account_user = &ctx.accounts.user;
    let account_user_pda = &mut ctx.accounts.user_pda;
    // Authorization: only the stored authority can update.
    require_keys_eq!(account_user_pda.auth, account_user.key(), PxsolError::Unauthorized);
    // At this point, Anchor has already reallocated the account according to the `realloc = ...` constraint
    // (using `new_data.len()`), pulling extra lamports from auth if needed to maintain rent-exemption.
    account_user_pda.data = data;
    // If the account was shrunk, Anchor won't automatically refund excess lamports. Refund any surplus (over the
    // new rent-exempt minimum) back to the user.
    let account_user_pda_info = account_user_pda.to_account_info();
    let rent = Rent::get()?;
    let rent_exemption = rent.minimum_balance(account_user_pda_info.data_len());
    let hold = **account_user_pda_info.lamports.borrow();
    if hold > rent_exemption {
        let refund = hold.saturating_sub(rent_exemption);
        // Transfer lamports from PDA to user using the PDA as signer.
        let signer_seeds: &[&[u8]] = &[SEED, account_user.key.as_ref(), &[account_user_pda.bump]];
        let signer = &[signer_seeds];
        let cpictx = CpiContext::new_with_signer(
            ctx.accounts.system_program.to_account_info(),
            system_program::Transfer { from: account_user_pda_info.clone(), to: account_user.to_account_info() },
            signer,
        );
        // It's okay if refund equals current - min_rent; system program enforces balances.
        system_program::transfer(cpictx, refund)?;
    }
    Ok(())
}
```

相配套的账户约束清晰地约定了该指令的一些策略细节.

```
#[derive(Accounts)]
#[instruction(new_data: Vec<u8>)]
pub struct Update<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(
        mut,
        seeds = [SEED, user.key().as_ref()],
        bump = user_pda.bump,
        realloc = Data::space_for(new_data.len()),
        realloc::payer = user,
        realloc::zero = false,
        constraint = user_pda.auth == user.key() @ PxsolError::Unauthorized,
    )]
```

```
    pub user_pda: Account<'info, Data>,
    pub system_program: Program<'info, System>,
}
```

10.3.4 常用技巧与注意事项

- 授权必检: `require_keys_eq!(...)`
- PDA 做签名者: 用 `new_with_signer`, seeds 里别忘了 `bump`.
- 伸缩成本与上限: 一次极大扩容会受限制, 必要时分片或多次更新.
- 资金来源: 扩容的 rent 差额由 `user` 出, 余额不足会失败.

10.3.5 收束

我们 anchor 版本的数据存储器很平凡, 却把 anchor 最常用的几块能力都连接在了一起: 账户约束, 动态重分配, pda 代签. 把它跑通之后, 我们可以继续加上一些更加复杂的逻辑. 代码总共只有不到 100 行, 但它是个很好的起点. 您应该能很快阅读懂它, 因此在这里不再过多赘述.

10.4 Solana/更多开发者工具/Anchor 测试框架

当你写下第一行合约代码, 测试就是它开口说的第一句话. 我们希望它既能在框架下顺畅表达, 也能在底层协议里自证严谨. 本节把测试当成一段小旅程: 先用 anchor 自带的 ts 测试框架走一条铺好的大道, 再用 python 下的 pxsol 客户端走一条原野小路(直接按二进制协议构造交易数据).

目标很朴素: 在本地链上, 初始化一个数据存储器, 多次更新内容, 然后把它读回来确认数据无误. 路径与代码都在仓库的 `tests/` 目录里.

10.4.1 TypeScript

这条路最省心. 你只需要告诉 anchor: 我要哪个程序, 要调用这个程序的哪个指令, 带上哪些账户与参数. 其余的编解码与账户核验, 由 anchor 和 idl 替你完成.

> Anchor 的 idl 会在你第一次构建程序时自动生成. 它记录了程序 id, 每个指令的账户与参数, 以及每个账户的数据结构. 你可以把它想象成一个桥梁, 连接链上程序与链下客户端.

我们的测试很比较简单, 先调用一次 init, 然后调用两次 update, 每次都传入不同长度的内容. 每次调用后, 我们都 fetch 一次账户数据, 确认内容正确.

```
import * as anchor from "@coral-xyz/anchor";
import { Program } from "@coral-xyz/anchor";
import { PxsolSsAnchor } from "../target/types/pxsol_ss_anchor";

describe("pxsol-ss-anchor", () => {
  // Configure the client to use the local cluster.
  anchor.setProvider(anchor.AnchorProvider.env());
  const program = anchor.workspace.pxsolSsAnchor as Program<PxsolSsAnchor>;
  const provider = anchor.getProvider() as anchor.AnchorProvider;
  const wallet = provider.wallet as anchor.Wallet;
  const walletPda = anchor.web3.PublicKey.findProgramAddressSync(
    [Buffer.from("data"), wallet.publicKey.toBuffer()],
    program.programId
  )[0];

  it("Init with content and then update (grow and shrink)", async () => {
    // Airdrop SOL to fresh authority to fund rent and tx fees
    await provider.connection.confirmTransaction(await provider.connection.requestAirdrop(
      wallet.publicKey,
      2 * anchor.web3.LAMPORTS_PER_SOL
    ), "confirmed");

    const poemInitial = Buffer.from("");
    const poemEnglish = Buffer.from("The quick brown fox jumps over the lazy dog");
    const poemChinese = Buffer.from("片云天共远, 永夜月同孤.");
    const walletPdaData = async (): Promise<Buffer<ArrayBuffer>> => {
      let walletPdaData = await program.account.data.fetch(walletPda);
      return Buffer.from(walletPdaData.data);
    }

    await program.methods
      .init()
      .accounts({
        user: wallet.publicKey,
        userPda: walletPda,
        systemProgram: anchor.web3.SystemProgram.programId,
      })
      .signers([wallet.payer])
```

```
      .rpc();
    if (!(await walletPdaData()).equals(poemInitial)) throw new Error("mismatch");

    await program.methods
      .update(poemEnglish)
      .accounts({
        user: wallet.publicKey,
        userPda: walletPda,
        systemProgram: anchor.web3.SystemProgram.programId,
      })
      .signers([wallet.payer])
      .rpc();
    if (!(await walletPdaData()).equals(poemEnglish)) throw new Error("mismatch");

    await program.methods
      .update(poemChinese)
      .accounts({
        user: wallet.publicKey,
        userPda: walletPda,
        systemProgram: anchor.web3.SystemProgram.programId,
      })
      .signers([wallet.payer])
      .rpc();
    if (!(await walletPdaData()).equals(poemChinese)) throw new Error("mismatch");
  });
});
```

运行:

```
# 自动构建, 部署到本地链并运行 ts 测试
$ anchor test
```

10.4.2 Python Pxsol

这条路更贴近协议本身. 我们会亲手排列账户列表, 拼接 8 字节方法 discriminator, 再把 4 字节小端长度与原始字节流接在后头. 它适合跨语言集成, 或在没有 anchor 客户端的环境里验算每一步.

代码如下:

```
import argparse
import base64
import pxsol


parser = argparse.ArgumentParser()
parser.add_argument('--net', type=str, choices=['develop', 'mainnet', 'testnet'], default='develop')
parser.add_argument('--prikey', type=str, default='1111111111111111111111111111112')
parser.add_argument('args', nargs='+')
args = parser.parse_args()

user = pxsol.wallet.Wallet(pxsol.core.PriKey.base58_decode(args.prikey))
prog_pubkey = pxsol.core.PubKey.base58_decode('GS5XPyzsXRec4sQzxJSpeDYHaTnZyYt5BtpeNXYuH1SM')
data_pubkey = prog_pubkey.derive_pda(b'data' + user.pubkey.p)


def init():
    rq = pxsol.core.Requisition(prog_pubkey, [], bytearray())
    rq.account.append(pxsol.core.AccountMeta(user.pubkey, 3))
```

```
    rq.account.append(pxsol.core.AccountMeta(data_pubkey, 1))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    rq.data = bytearray().join([
        bytearray([220, 59, 207, 236, 108, 250, 47, 100]),
    ])
    tx = pxsol.core.Transaction.requisition_decode(user.pubkey, [rq])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([user.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
    r = pxsol.rpc.get_transaction(txid, {})
    for e in r['meta']['logMessages']:
        print(e)


def update():
    rq = pxsol.core.Requisition(prog_pubkey, [], bytearray())
    rq.account.append(pxsol.core.AccountMeta(user.pubkey, 3))
    rq.account.append(pxsol.core.AccountMeta(data_pubkey, 1))
    rq.account.append(pxsol.core.AccountMeta(pxsol.program.System.pubkey, 0))
    rq.data = bytearray().join([
        bytearray([219, 200, 88, 176, 158, 63, 253, 127]),
        len(args.args[1].encode()).to_bytes(4, 'little'),
        args.args[1].encode(),
    ])
    tx = pxsol.core.Transaction.requisition_decode(user.pubkey, [rq])
    tx.message.recent_blockhash = pxsol.base58.decode(pxsol.rpc.get_latest_blockhash({})['blockhash'])
    tx.sign([user.prikey])
    txid = pxsol.rpc.send_transaction(base64.b64encode(tx.serialize()).decode(), {})
    pxsol.rpc.wait([txid])
    r = pxsol.rpc.get_transaction(txid, {})
    for e in r['meta']['logMessages']:
        print(e)


def load():
    info = pxsol.rpc.get_account_info(data_pubkey.base58(), {})
    print(base64.b64decode(info['data'][0])[8 + 32 + 1 + 4:].decode())


if __name__ == '__main__':
    eval(f'{args.args[0]}()')
```

运行:

```
$ solana-test-validator -l /tmp/solana-ledger
$ anchor deploy
# Program Id: GS5XPyzsXRec4sQzxJSpeDYHaTnZyYt5BtpeNXYuH1SM

$ python tests/pxsol-ss-anchor.py update "The quick brown fox jumps over the lazy dog"
$ python tests/pxsol-ss-anchor.py load
# The quick brown fox jumps over the lazy dog

$ python tests/pxsol-ss-anchor.py update "片云天共远, 永夜月同孤."
$ python tests/pxsol-ss-anchor.py load
# 片云天共远, 永夜月同孤.
```

10.5 Solana/更多开发者工具/Pinocchio? Pinocchio!

Solana 开发团队正在积极打造更多工具, 帮助开发者更高效地编写和测试链上程序. 其中一个有趣的项目是 pinocchio, 它在本书创作期间首次发布版本, 因此我也对它进行了探索, 并且迫不及待的想将它纳入本书的内容里.

10.5.1 Pinocchio?

Pinocchio 是一个零依赖(zero-dependency), no_std 的 rust 库, 用来编写 solana 链上程序. 它不依赖 solana-program, 而是直接契合 svm loader 与程序之间的原始 abi(一个序列化后的字节数组), 通过零拷贝解析把输入映射为程序可用的类型, 从而在以下方面受益:

- 更小二进制: 避免引入庞大的 sdk 依赖
- 更低资源消耗: 入口解析与 cpi 过程更省计算
- 更强掌控: 按需解析输入, 可禁用内存分配器(allocator), 强化可预测性
- 适合追求极致体积与性能, 或被依赖地狱困扰的项目

零依赖是 pinocchio 的核心卖点, 使用 pinocchio 时不像使用 solana-program, 不需要引入上百个依赖, 但它的功能确是和 solana-program 相当的. 它提供了类似的类型与功能, 在实际开发过程中可以平替 solana-program, 也可以很方便的从现有的 solana-program 代码迁移过来.

但是需要搞清楚一个区别, pinocchio 和 solana-program 一样, 是一个链上程序开发库, 而不是一个完整的类似 anchor 的开发框架. 它不提供类似 anchor 的宏与代码生成, 也不提供类似 anchor 的本地测试与部署工具. 它只是一个更轻量的链上程序开发库, 让你可以更高效地编写链上程序.

10.5.2 快速开始

要在项目中使用 pinocchio, 只需添加其为依赖即可. 在本文档创作时, pinocchio 的最新版本是 0.9.2, 你可以根据需要选择合适的版本.

```
[dependencies]
pinocchio = "0.9"
```

之后在 `src/lib.rs` 中使用 pinocchio 替代 solana-program, 例如:

```
use pinocchio::{
  account_info::AccountInfo,
  entrypoint,
  msg,
  ProgramResult,
  pubkey::Pubkey
};

entrypoint!(process_instruction);

pub fn process_instruction(
  _program_id: &Pubkey,
  _accounts: &[AccountInfo],
  _instruction_data: &[u8],
) -> ProgramResult {
  msg!("Hello from my program!");
  Ok(())
}
```

熟悉 solana-program 的同学会觉得这段非常眼熟, 只是类型与宏来自 pinocchio.

Pinocchio 还有一些进阶的用法, 允许你更灵活地控制入口与解析过程, 或者禁用一些不需要的功能, 以进一步减小二进制体积与降低计算消耗. 我们在这里不去赘述, 你可以参考 pinocchio 的文档 了解更多. 作为对初学者的建议, 如果你只是想快速上手, 可以先按上面的方式使用 pinocchio, 之后再根据需要逐步探索更高级的用法.

10.5.3 迁移指南: 从 solana-program 到 pinocchio

如果你已经有一个使用 solana-program 的链上程序, 想要迁移到 pinocchio, 这里有一些建议的步骤:

入口替换

原先的 `solana_program::entrypoint::process_instruction` -> 使用 pinocchio 的 `entrypoint!` 或 `lazy_program_entrypoint!` 宏.

类型替换

大多数 solana_program 的类型在 pinocchio 中都有对应的替代品, 只需将导入路径替换即可. 例如:

- `solana_program::pubkey::Pubkey` -> `pinocchio::pubkey::Pubkey`
- `solana_program::account_info::AccountInfo` -> `pinocchio::account_info::AccountInfo`

日志替换

- `solana_program::msg!` -> `pinocchio::msg!`
- 如果需要格式化, 那么需要引入 `pinocchio-log` 的 `log!` 宏

cpi 与 sysvars

- 将 `solana_program::program::invoke*` 与 `sysvar` 访问, 替换为 pinocchio 提供的接口(命名与用法保持直觉, 一般改动不大)

从作者的直觉角度来看, pinocchio 本质上并不是另一个 solana program sdk, 而是更靠近 abi 的最小集合. 它只提供了写链上程序所需的核心拼装件, 但把控制权还给了你, 它不会对这些链上概念进行过多的包装, 也不会对它们进行过多的抽象. 对追求极致与可控的 solana 开发者来说, 它能在可观地降低二进制体积与计算资源消耗的同时, 让工程更简洁, 更稳定.

在下一小节, 我们将使用 pinocchio 来重写前面章节的链上数据存储程序, 以展示它的实际用法与效果.

10.6 Solana/更多开发者工具/Pinocchio 重写简单数据存储合约

本节内容基于 pxsol-ss, 旨在展示如何使用 pinocchio 重写它. 最终成品课程代码位于 pxsol-ss-pinocchio 仓库.

10.6.1 迁移工作

我们遇到的第一个问题是要修改 `rent_exemption` 和 `bump_seed` 的获取方式. 下面是修改前后的对比.

旧

```
let rent_exemption = solana_program::rent::Rent::get()?.minimum_balance(data.len());
let bump_seed = solana_program::pubkey::Pubkey::find_program_address(&[&account_user.key.to_bytes()], program_id).1;
```

新

```
let rent_exemption = pinocchio::sysvars::rent::Rent::get()?.minimum_balance(data.len());
let bump_seed = &[pinocchio::pubkey::find_program_address(&[&account_user.key()[..]], program_id).1];
```

判断程序扩展账户是否已经存在.

旧

```
if **account_data.try_borrow_lamports().unwrap() == 0 { ...
```

新

```
if account_data.lamports() == 0 { ...
```

调用系统程序, 创建程序扩展账户. 在 pinocchio 中, 相关的系统调用被封装在 `pinocchio_system` crate 中, 并且提供了更简洁的用法. 下面是修改前后的对比.

旧

```
solana_program::program::invoke_signed(
    &solana_program::system_instruction::create_account(
        account_user.key,
        account_data.key,
        rent_exemption,
        data.len() as u64,
        program_id,
    ),
    accounts,
    &[&[&account_user.key.to_bytes(), &[bump_seed]]],
)?;
```

新

```
pinocchio_system::instructions::CreateAccount {
    from: &account_user,
    to: &account_data,
    lamports: rent_exemption,
    space: data.len() as u64,
    owner: program_id,
}
.invoke_signed(&[signer])?;
```

修改程序扩展账户的数据

旧

```
account_data.data.borrow_mut().copy_from_slice(data);
```

新

```
account_data.try_borrow_mut_data().unwrap().copy_from_slice(data);
```

修改账户余额

旧

```
**account_user.lamports.borrow_mut() = account_user.lamports() + account_data.lamports() - rent_exemption;
**account_data.lamports.borrow_mut() = rent_exemption;
```

新

```
*account_user.try_borrow_mut_lamports().unwrap() += account_data.lamports() - rent_exemption;
*account_data.try_borrow_mut_lamports().unwrap() = rent_exemption;
```

我们的迁移工作已经完成, 下面是完整代码.

```
use pinocchio::sysvars::Sysvar;

pinocchio::entrypoint!(process_instruction);

pub fn process_instruction(
    program_id: &pinocchio::pubkey::Pubkey,
    accounts: &[pinocchio::account_info::AccountInfo],
    data: &[u8],
) -> pinocchio::ProgramResult {
    let account_user = accounts[0];
    let account_data = accounts[1];

    let rent_exemption = pinocchio::sysvars::rent::Rent::get()?.minimum_balance(data.len());
    let bump_seed = &[pinocchio::pubkey::find_program_address(&[&account_user.key()[..]], program_id).1];
    let signer_seed = pinocchio::seeds!(account_user.key(), bump_seed);
    let signer = pinocchio::instruction::Signer::from(&signer_seed);

    // Data account is not initialized. Create an account and write data into it.
    if account_data.lamports() == 0 {
        pinocchio_system::instructions::CreateAccount {
            from: &account_user,
            to: &account_data,
            lamports: rent_exemption,
            space: data.len() as u64,
            owner: program_id,
        }
        .invoke_signed(&[signer])?;
        account_data.try_borrow_mut_data().unwrap().copy_from_slice(data);
        return Ok(());
    }

    // Fund the data account to let it rent exemption.
```

```
    if rent_exemption > account_data.lamports() {
        pinocchio_system::instructions::Transfer {
            from: &account_user,
            to: &account_data,
            lamports: rent_exemption - account_data.lamports(),
        }
        .invoke()?;
    }

    // Withdraw excess funds and return them to users. Since the funds in the pda account belong to the program, we do
    // not need to use instructions to transfer them here.
    if rent_exemption < account_data.lamports() {
        *account_user.try_borrow_mut_lamports().unwrap() += account_data.lamports() - rent_exemption;
        *account_data.try_borrow_mut_lamports().unwrap() = rent_exemption;
    }
    // Realloc space.
    account_data.resize(data.len())?;
    // Overwrite old data with new data.
    account_data.try_borrow_mut_data().unwrap().copy_from_slice(data);

    Ok(())
}
```

最后我们来测试一下这个程序是否能正常工作. 运行以下命令, 进行编译, 部署和测试.

```
$ python make.py deploy
# 2025/05/20 16:06:38 main: deploy program pubkey="T6vZUAQyiFfX6968XoJVmXxpbZwtnKfQbNNBYrcxkcp"

# Save some data.
$ python make.py save "The quick brown fox jumps over the lazy dog"

# Load data.
$ python make.py load
# The quick brown fox jumps over the lazy dog.

# Save some data and overwrite the old data.
$ python make.py save "片云天共远, 永夜月同孤."
# Load data.
$ python make.py load
# 片云天共远, 永夜月同孤.
```

10.6.2 关键数据对比

/	**pxsol-ss**	**pxsol-ss-pinocchio**
编译时间	1m10.242s	0m1.930s
编译产物大小	75K	29K
依赖项目数量	189	7

通过以上这些关键数据对比, 我们可以看到使用 pinocchio 重写后的程序在编译时间, 编译产物大小和依赖项目数量上都有显著的提升. 这也说明了 pinocchio 作为 solana-program 的替代品, 在简化开发流程和提升效率方面具有很大的优势. 我们强烈推荐在新的 solana 程序开发中使用 pinocchio.

10.7 Solana/更多开发者工具/web3.js 快速上手

我们开发了一个简单的 dapp, 演示如何使用 `solana/web3.js` 为我们的 on-chain program 提供前端交互功能. 你可以参考并扩展到自己的项目中. 本文使用的例子是使用 react + vite 的前端, 连接 phantom 钱包, 并通过上节课介绍的基于 pinocchio 的数据存储程序来读写 pda 数据.

项目地址仍然位于 https://github.com/mohanson/pxsol-ss-pinocchio.

为了方便读者们学习, 我已经将该项目部署到 solana 主网上, 程序地址是 `9RctzLPHP58wrnoGCbb5FpFKbmQb6f53i5PsebQZSaQL`, 你可以直接使用该程序进行测试.

> 在部署程序时作者因为失误发生了数次错误, 总共花费了大约 1.5 sol 的手续费才完成了部署, 也就是大约 300 美元, 请大家未来在操作主网时务必小心谨慎, 避免不必要的损失.

10.7.1 作者的碎碎念

我本身并非前端工程师, 对前端的了解仅仅限于能做出一个简单的 demo, 但我觉得 web3.js 的学习曲线并不陡峭, 只要你有一定的前端基础, 并且了解 solana 的基本概念, 就能很快上手.

另外现在的 ai 非常强大, 我在写本文的前端代码时大量使用了 github copilot 和 chatgpt, 这些工具能极大地提高开发效率, 但也会带来一些问题, 例如生成的代码可能并不完全正确, 需要你有一定的判断能力来辨别和修正错误. 所以我建议大家在学习和使用这些工具时, 不要完全依赖它们, 而是要结合自己的理解和经验.

在此处我只介绍最核心的代码片段, 以及一些需要注意的点. 如果你想了解完整的代码, 请参考项目源码.

10.7.2 建立连接

```
import { Connection, PublicKey } from '@solana/web3.js'

export const PROGRAM_ID = new PublicKey('9RctzLPHP58wrnoGCbb5FpFKbmQb6f53i5PsebQZSaQL')
export const RPC_ENDPOINT = import.meta.env.VITE_SOLANA_RPC
  || 'https://api.mainnet-beta.solana.com'

export const connection = new Connection(RPC_ENDPOINT, 'confirmed')
```

10.7.3 连接浏览器里的 phantom 钱包

```
import type { Transaction } from '@solana/web3.js'

type PhantomProvider = {
  isPhantom?: boolean
  publicKey?: PublicKey
  connect(opts?: { onlyIfTrusted?: boolean }): Promise<{ publicKey: PublicKey }>
  disconnect(): Promise<void>
  signTransaction(tx: Transaction): Promise<Transaction>
}

declare global { interface Window { solana?: PhantomProvider } }

export async function connectPhantom(): Promise<PublicKey> {
  if (!window.solana?.isPhantom) throw new Error('Phantom not found')
  return (await window.solana.connect()).publicKey
}
```

10.7.4 派生程序扩展账户

前端需保持和链上程序相同的派生种子:

```
import { PublicKey } from '@solana/web3.js'

export async function deriveDataPda(user: PublicKey): Promise<[PublicKey, number]> {
  return PublicKey.findProgramAddress([user.toBuffer()], PROGRAM_ID)
}
```

10.7.5 读取账户数据

```
import { Connection, PublicKey } from '@solana/web3.js'

export async function fetchUserData(conn: Connection, user: PublicKey): Promise<Uint8Array | null> {
  const [pda] = await deriveDataPda(user)
  const info = await conn.getAccountInfo(pda, { commitment: 'confirmed' })
  return info ? info.data : null
}

export function decodeUtf8(data: Uint8Array | null): string {
  return data ? new TextDecoder().decode(data) : ''
}
```

10.7.6 构造写入指令

```
import { TransactionInstruction, PublicKey, SystemProgram } from '@solana/web3.js'

export async function buildWriteIx(user: PublicKey, payload: Uint8Array): Promise<TransactionInstruction> {
  const [pda] = await deriveDataPda(user)
  return new TransactionInstruction({
    programId: PROGRAM_ID,
    keys: [
      { pubkey: user, isSigner: true, isWritable: true },
      { pubkey: pda, isSigner: false, isWritable: true },
      { pubkey: SystemProgram.programId, isSigner: false, isWritable: false },
    ],
    data: payload,
  })
}
```

备注: 若你项目里需要 `Buffer`, 可使用 `data: Buffer.from(payload)`, 并 `import { Buffer } from 'buffer'`.

10.7.7 发送交易并确认

```
import { Connection, Transaction, TransactionInstruction, PublicKey } from '@solana/web3.js'

type PhantomProvider = { signTransaction(tx: Transaction): Promise<Transaction> }

export async function sendAndConfirm(
  conn: Connection,
  user: PublicKey,
  ix: TransactionInstruction,
  wallet: PhantomProvider,
) {
  const tx = new Transaction().add(ix)
  tx.feePayer = user
  const { blockhash, lastValidBlockHeight } = await conn.getLatestBlockhash('finalized')
  tx.recentBlockhash = blockhash
```

```
  const signed = await wallet.signTransaction(tx)
  const sig = await conn.sendRawTransaction(signed.serialize(), { preflightCommitment: 'finalized' })
  await conn.confirmTransaction({ signature: sig, blockhash, lastValidBlockHeight }, 'finalized')
  return sig
}
```

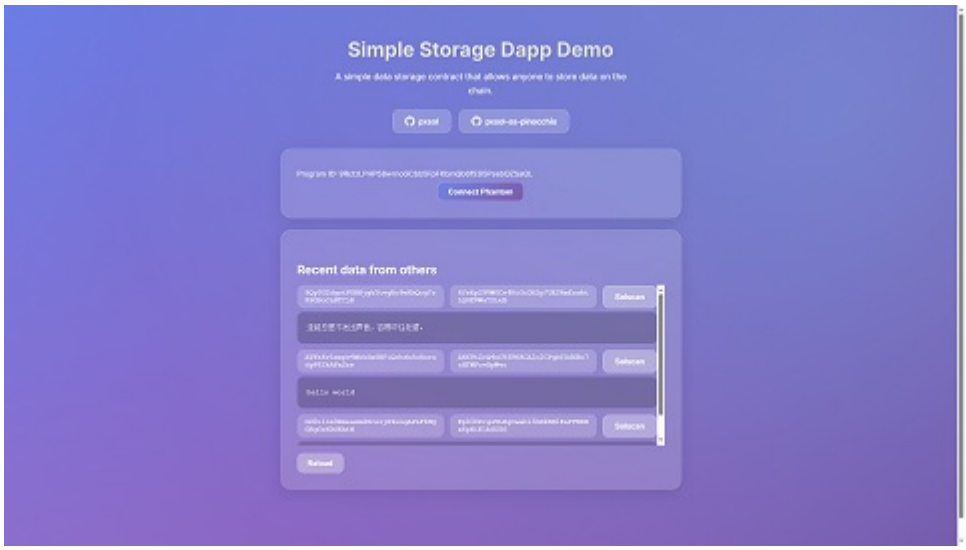
10.7.8 运行

您可以在本地运行该前端项目:

```
$ npm run dev
# Open http://localhost:5173
# Connect Phantom wallet and save/load data.
```

或者直接访问我们部署好的线上版本: https://pxsol-ss-pinocchio.vercel.app/.

首先点击 connect 连接钱包, 然后在输入框中输入任意字符串, 点击 save 保存到链上, 再刷新页面, 即可读取刚才保存的数据.



10.8 Solana/更多开发者工具/web3.js 的常见坑与规避

这里汇总一下我在使用 solana/web3.js + 浏览器钱包 phantom 开发时容易踩到的问题与解决方案. 下面这些问题都是我实际遇到过的, 并且花了不少时间调试才搞明白.

10.8.1 Phantom 钱包的运行环境

Phantom 钱包仅在两种来源下工作:

- `localhost` (http/https 均可)
- `https` 站点(必须是安全源)

在非 https 的远程域名下, phantom 不会注入到浏览器环境里, 您无法连接到 phantom 钱包.

10.8.2 官方公共 rpc 的 cors / 403

直接从浏览器请求 `https://api.mainnet-beta.solana.com` 常见 403 或 cors 拒绝. 此问题的原因是官方公共 rpc 主动限制了浏览器的访问. 要解决此问题, 有两种做法:

1. 使用允许浏览器访问, 带 api key 的提供商 rpc, 作者采用了此方式并使用了 helius.
2. 或者在开发服务器/后端配置反向代理, 让前端走同源路径.

Vite 代理示例:

```
// vite.config.ts
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      '/rpc': {
        target: 'https://your.provider.rpc/solana-mainnet?api-key=XXXX',
        changeOrigin: true,
        secure: true,
      },
    },
  },
})
```

前端代码中使用相对地址:

```
const RPC_ENDPOINT = import.meta.env.VITE_SOLANA_RPC || '/rpc'
```

10.8.3 交易体积限制

Solana 交易大小上限为 1232 字节, 排除掉签名等开销, 实际可用空间更少. 如果您在交易中携带了大量数据, 可能会遇到 `Transaction too large` 错误.

10.9 Solana/更多开发者工具/solana-py 和 solders 库的结合使用

工具 solana-py 是一个 solana 的 python 库, 用于与 solana 区块链进行交互. 它提供了与 solana 节点通信的 api, 适合开发者快速构建与 solana 网络交互的应用程序. 该库底层依赖 solders 库, solders 是一个更底层的 rust 语言编写的 solana 库, 提供了对 solana 协议和数据结构的直接访问.

您可以这么理解:

- solders: 提供了对 solana 协议和数据结构的直接访问
- solana-py: 构建在 solders 之上, 提供了更高层次的 api, 适合快速开发和与 solana 网络交互的应用

下面的示例代码展示了如何使用 solana-py 和 solders 库来获取我们的简单链上数据合约的所有交易历史记录, 并打印出该程序所拥有的所有账户的数据.

```
"""
Fetch all transaction history for a Solana program and print stored data for each account owned by the program.

Network: mainnet-beta
Program: 9RctzLPHP58wrnoGCbb5FpFKbmQb6f53i5PsebQZSaQL

This script uses the JSON-RPC getSignaturesForAddress pagination to walk the full history of the program (within node's
retention), then fetches program-owned accounts and prints their data.
"""
import datetime
import itertools
import os
import solana.rpc.api
import solders
import solders.rpc.responses
import typing

DEFAULT_PROGRAM_ID = '9RctzLPHP58wrnoGCbb5FpFKbmQb6f53i5PsebQZSaQL'
DEFAULT_RPC = os.environ.get('SOLANA_RPC', 'https://api.mainnet-beta.solana.com')


def get_all_program_sigs(
    client: solana.rpc.api.Client,
    program: solders.pubkey.Pubkey,
) -> typing.Generator[solders.rpc.responses.RpcConfirmedTransactionStatusWithSignature, None, None]:
    cursor = None
    limits = 256
    for _ in itertools.repeat(0):
        resp = client.get_signatures_for_address(program, before=cursor, limit=limits)
        sigs = resp.value
        if not sigs:
            break
        for s in sigs:
            yield s
        cursor = sigs[-1].signature


def get_all_program_pdas(
    client: solana.rpc.api.Client,
    program: solders.pubkey.Pubkey,
) -> typing.List[solders.rpc.responses.RpcKeyedAccount]:
    resp = client.get_program_accounts(program, encoding='base64')
    pdas = resp.value
```

```
    return pdas


def main():
    program_key = solders.pubkey.Pubkey.from_string(DEFAULT_PROGRAM_ID)
    client = solana.rpc.api.Client(DEFAULT_RPC)
    print('main: get_all_program_sigs')
    for e in get_all_program_sigs(client, program_key):
        print(f'main: datetime={datetime.datetime.fromtimestamp(e.block_time)}, sig={e.signature}')
    print('main: get_all_program_pdas')
    for e in get_all_program_pdas(client, program_key):
        print('main:    owner:', e.account.owner)
        print('main:      pda:', e.pubkey)
        print('main: lamports:', e.account.lamports)
        print('main:     data:',  e.account.data.decode())
        print()


if __name__ == '__main__':
    main()
```

运行后的结果如下, 它成功获取到了作者存储在链上的数据:

```
main: get_all_program_sigs
main: datetime=2025-10-13 14:42:45, sig=4k2rTsRW2s1GKxsaDmDx2sM9rRVTzGS7LgVnTphAPNQ4ZDpbhLeLLSrVLeRQLxBEpfWsTGFoAn3uuJDzr6
main: datetime=2025-10-13 14:40:22, sig=5WduQv7NGXpnHUYWwrjhyzWtHHP6BRj2os4iuj8JpYCHGMeiT2wRzGzgm2yF5PdKcAuux7LhXkDe1B79oF
main: datetime=2025-10-13 14:33:51, sig=EPJQwvr3WpVHZ6Jdr4DWrx5sKVScb7yrz2YRfGHyPqTpjNL4WnmvoquA9HATsPCQrzpSsyPx5WUUQt7Ger
main: datetime=2025-10-13 11:31:49, sig=3wXnUFazxCz5zbvMcptVXKNyjuP45zbuc5UzmpxwKzK8VaQzTsuwoMdXhJTBaowhJ5LWuVwuFghUDvG7pu
main: get_all_program_pdas
main:    owner: 9RctzLPHP58wrnoGCbb5FpFKbmQb6f53i5PsebQZSaQL
main:      pda: Ep838PrgsVLKgCwab15hNkBB1EaFFBKKx6pNiE1bGD5G
main: lamports: 1120560
main:     data: 片云天共远, 永夜月同孤.
```

这个示例的原理只是使用 `get_signatures_for_address` 和 `get_all_program_sigs` 这两个 rpc 接口, 结合 solders 库来解析数据, 以便打印出我们存储在链上的数据. 通过这种方式, 我们可以很方便地查看任何 solana 程序的交易历史和账户数据.

11. Solana/书后

2024 年的区块链世界, solana 无疑是最耀眼的明星之一.

这个曾在熊市中历经网络中断, 质疑声四起的公链, 用一年时间完成了令人惊叹的蜕变. 它的 defi 协议的 tvl 持续攀升, nft 市场交易量屡创新高, meme 币文化蓬勃发展, 开发者社区日益壮大. Solana 正在用实际行动向世界证明: 真正的技术创新终将突破质疑的迷雾.

11.1 动笔

2024 年 3 月, 我做了一个决定: 系统性地学习 solana, 并将这个过程记录成教程. 我在当年 10 月正式动笔撰写本书.

彼时, 虽然 solana 的文档已经较为完善, 但学习资源仍显零散. 许多有潜力的开发者因为学习曲线陡峭而却步. 在当时的区块链开发者社区, 关于 solana 的学习资料或课程寥寥无几.

从环境配置到工具链使用, 从链上程序开发到链上交互, 我试图构建一个完整的知识体系. 书中的每一个代码示例, 我都会反复调试; 每一个概念解释, 我都会斟酌措辞. 技术写作不同于代码编写, 它需要在准确性和可读性之间找到微妙的平衡点.

写作非常消耗精力, 也非常耗时. 每天工作之余, 我都会挤出时间来写作. 有时候, 一天只能写几百字, 但我坚持每天都要写一点, 积少成多.

11.2 转机

2024 年 12 月的某个下午, 我收到了来自 solana 基金会的消息: 他们愿意为这本书提供赞助支持.

说实话, 那一刻的心情很复杂. 惊喜之余, 更多的是责任感. 这不仅是对我前期工作的认可, 更意味着这本书将承载更多开发者的期待. 从此刻起, 这些文字由一个人的学习笔记转变为一份需要对社区负责的技术文档.

基金会的支持让我能够投入更多时间和精力, 也开始让我肆无忌惮的在主网上测试各种功能: 过去我总是担心测试费用问题, 因为在主网上部署一个程序通常需要耗费几十到几百美元. 但有了赞助, 我可以更大胆地尝试各种功能, 这极大地提升了写作效率和内容的丰富度.

11.3 煎熬

2025 年 11 月, 我终于在键盘上敲下了最后一个句号.

从动笔到完稿, 整整十三个月. 这段时间里, solana 生态经历了数次重大更新, 我不得不多次回头修订已经写好的章节. 有时候, 一个 api 的变更就意味着数十处代码需要重写; 更何况此时负责 solana 节点代码维护的 anza-xyz 团队正在大刀阔斧的重构 solana 节点软件, 许多项目结构和代码位置都发生了变化, 这让我不得不反复确认书中的内容是否仍然准确无误.

最艰难的时刻, 是在深夜面对一个似是而非的技术点, 翻遍文档和源码仍无法确认的时刻. 那种焦虑感, 就像是在黑暗中摸索, 不知道前方是死路还是豁然开朗. 好在 solana 社区的开发者们总是乐于助人, discord 上的讨论, github 上的 issue, 甚至是一些素未谋面的开发者的私信, 都给了我莫大的帮助.

技术写作是孤独的. 没有同事的讨论, 没有即时的反馈, 只有眼前的微光和耳畔的敲击. 但正是这份孤独, 让我能够静下心来, 深入思考每一个技术细节.

11.4 感谢

书桌的右侧, 是几盆陪伴我度过漫长写作时光的绿色植物.

铜钱草在水培瓶中舒展着圆润的叶片, 石斛的气根顽强地攀附在树皮上, 风车草的叶片像伞一样张开, 芦荟则静静地积蓄着生命的力量. 它们不会说话.

特别要感谢 solana 基金会的赞助与信任.

同时感谢所有在创作过程中给予帮助的朋友们, 尤其是那些提供反馈意见的早期读者.

Solana 生态仍在快速演进, 新的特性和工具不断涌现. 这本书无法涵盖所有内容, 但我希望它能成为你探索 solana 世界的一个可靠起点. 技术的学习没有捷径, 唯有动手实践, 不断思考, 才能真正掌握其中的精髓.

路漫漫其修远兮, 吾将上下而求索.