Solona 开发概念

概念

eBPF：（extended Berkeley Packet Filter）
账户模型：Account Model
交易和指令：Transactions and Instructions
PDA：Program Derived Addresses
CPI：Cross Program Invocation

Spl token

eBPF

为了编写和执行智能合约，区块链往往需要一套编程语言和图灵完备的计算环境。

以太坊上的智能合约通常使用高级语言Solidity来编写，而Solidity编译产生的字节码则运行在一个叫做以太坊虚拟机的环境中。

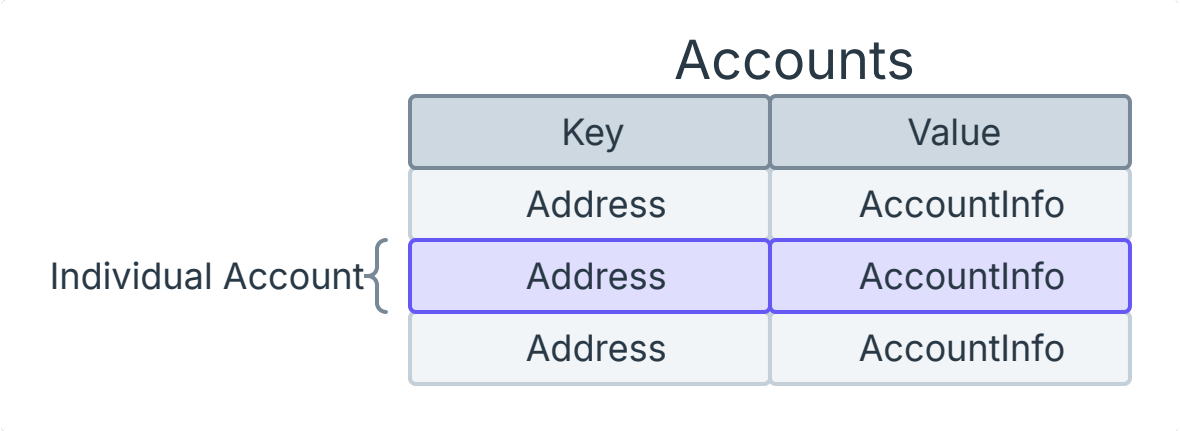
Solana并没有选择开发全新的虚拟环境和语言，而是充分利用了现有的优秀技术。原本用于拓展Linux内核功能的eBPF（extended Berkeley Packet Filter）虚拟机被Solana选中并作为底层的执行环境。

相较于只支持解释执行的EVM，eBPF能够以即时编译（JIT）模式直接将字节码转换成处理器可以直接执行的机器指令，从而更高效地运行程序。

eBPF拥有一套高效的指令集和成熟的基础设施。开发者只需要使用Rust语言即可编写智能合约。LLVM编译框架提供了一个eBPF的后端，**利用它可以直接将这些Rust语言编写的程序编译成可运行在eBPF虚拟机上的字节码。**

Solana 账户模型

在 Solana 上，所有数据都存储在所谓的“帐户”中。 Solana 上的数据组织方式类似 于键值存储，其中数据 库中的每个条目称为“帐户(Account)”。



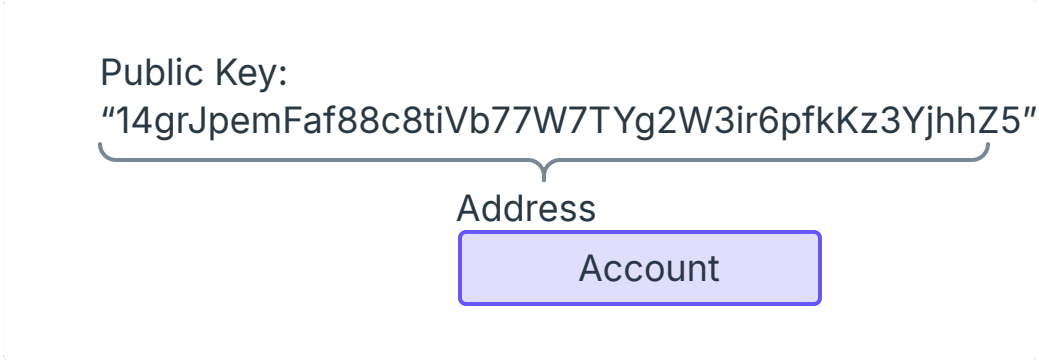
账户

要点

- 账户最多可存储10MB 数据，这些数据可以由可执行的程序代码或程序状态组成。
- 账户需要以 SOL 形式存入租金，与存储数据的数量成比例，当账户关闭时，该金额可以 全额退款。
- 每个帐户都有一个程序“所有者”。 只有拥有帐户的程序可以修改其数据或扣除其 lamport 余额。 然而，任何人都可以增加的余额。
- 程序 (智能合约) 是无状态账户，用于存储可执行代码。
- 数据账户是由程序创建的，用于存储和管理程序状态。
- 原生程序是内置程序，包括在 Solana 运行时内。
- Sysvar 帐户是存储网络集群状态的特殊帐户。

账户

每个帐户都可以通过其唯一的地址来识别，用 Ed25519 `PublicKey` 格式。 你可以将地址的视为账户的唯一标识符。

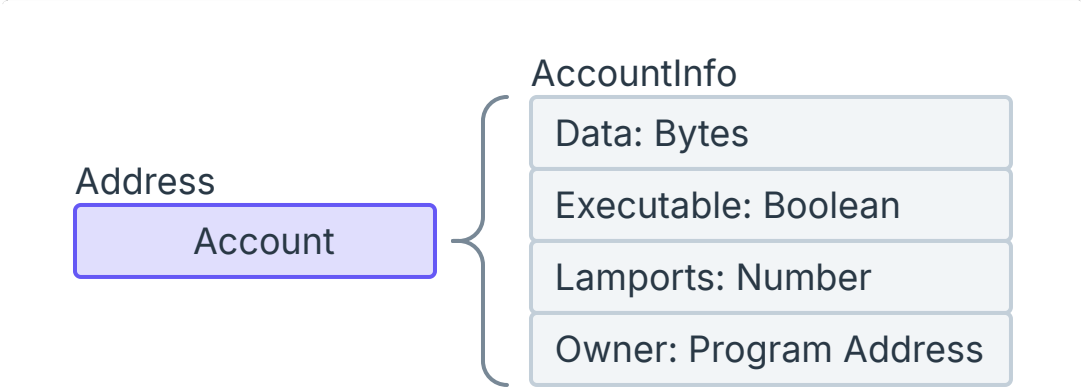


帐户地址

此账户及其地址之间的关系可以被视为键值对， 而地址是定位相应的链上账户数据的键。

帐户信息

帐户有一个 最大大小为 10MB(10 Mega Bytes) ，并且 Solana 上每个帐户上存储的数据有以下结构称为 AccountInfo。



AccountInfo

每个账户的 `AccountInfo` 包含以下字段：

- `data`: 存储帐户状态的字节数组。 如果帐户是程序(智能合约)，则存储可执行程序代码。 此字段是，通常称为“帐户数据”。
- `executable`: 表示帐户是否为程序的布尔标识符。

- `lamports`：账户余额的数字表示方式为 lamports，这是SOL中最小的单位(1 SOL = 10亿 lamports)。
- `owner`：指定哪一个程序的公钥 (程序ID) 拥有该帐户。

作为Solana 账户模式的一个关键部分，Solana 上的每个账户都有一个指定为“所有者”，特 别是一个程序。 只有指定为帐户所有者的程序可以修改帐户中存储的数据或扣除 lamport 余额。 重要的是要注意到，虽然只有所有者可以扣除余额，但任何人都可以增加余额。

INFO

若要在链上存储数据，必须将一定数量的SOL转账到一个帐户。 转账金额与账户上存储的 数据的大小成比例。 这一概念通常称为“租用”。 然而，仍然存在着这种情况。你可以认 为"出租"更像"保证金"，因为分配给帐户的SOL可以在关闭帐户时完全恢复。

原生程序

Solana 包含少数原生程序，这些程序是验证器实现的一部分，并为网络提供了各种核心功 能。 你可以这里找到原生程序的完 整列表。

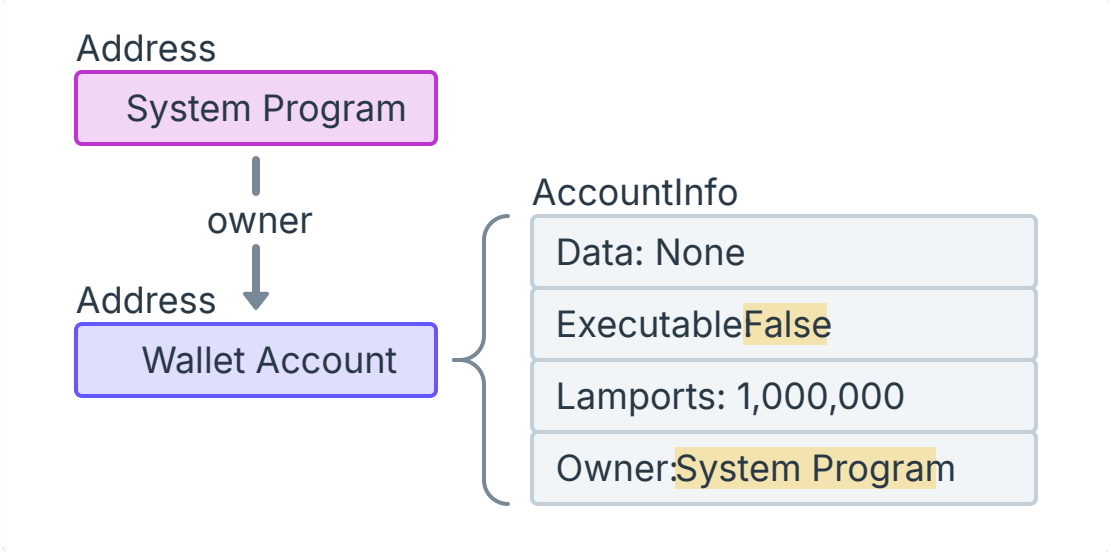
当在 Solana 上开发自定义程序时，你通常会与两个原生程序交互，即系统程序和 BPF Loader。

系统程序

默认情况下，所有新账户都属 于系统程序。 系统程序执行几项关键任务，例如：

- 新账户创建: 只有系统程序可以创建新帐户。
- 空间分配: 设置每个帐户数据字节的容量。
- 分配程序所有者: 一旦系统程序创建了一个帐户，它可以将指定的程序所有者重新分配到不同的程序帐户。 这是自定义程序如何获取系统程序创建的新账户的所有权。

在Solana上，“钱包”只是系统程序拥有的帐户。 钱包的 lamport 余额是账户拥有的 SOL 金额。



系统账户

INFO

只有系统程序拥有的账户才能用作交易费付款人。

▼ 系统程序有哪些

Native Programs in the Solana Runtime

Solana contains a small handful of native programs, which are required to run validator nodes. Unlike third-party programs, the native programs are part of the validator implementation and can be upgraded as part of cluster upgrades. Upgrades may occur to add features, fix bugs, or improve performance. Interface changes to individual instructions should rarely, if ever, occur. Instead, when change is needed, new instructions are added and previous ones are marked deprecated. Apps can upgrade on their own timeline without concern of breakages across upgrades.

For each native program the program id and description each supported instruction is provided. A transaction can mix and match instructions from different programs, as well include instructions from on-chain programs.

System Program

Create new accounts, allocate account data, assign accounts to owning programs, transfer lamports from System Program owned accounts and pay transaction fees.

- Program id: `11111111111111111111111111111111`
- Instructions: SystemInstruction

Config Program

Add configuration data to the chain and the list of public keys that are permitted to modify it

- Program id: `Config1111111111111111111111111111111111111`
- Instructions: config_instruction

Unlike the other programs, the Config program does not define any individual instructions. It has just one implicit instruction, a "store" instruction. Its instruction data is a set of keys that gate access to the account, and the data to store in it.

Stake Program

Create and manage accounts representing stake and rewards for delegations to validators.

- Program id: `Stake11111111111111111111111111111111111111`
- Instructions: StakeInstruction

Vote Program

Create and manage accounts that track validator voting state and rewards.

- Program id: `Vote111111111111111111111111111111111111111`
- Instructions: VoteInstruction

Address Lookup Table Program

- Program id: `AddressLookupTab1e1111111111111111111111111`

- Instructions: AddressLookupTableInstruction

BPF Loader

Deploys, upgrades, and executes programs on the chain.

- Program id: `BPFLoaderUpgradeab1e11111111111111111111111`
- Instructions: LoaderInstruction

The BPF Upgradeable Loader marks itself as "owner" of the executable and program-data accounts it creates to store your program. When a user invokes an instruction via a program id, the Solana runtime will load both your the program and its owner, the BPF Upgradeable Loader. The runtime then passes your program to the BPF Upgradeable Loader to process the instruction.

More information about deployment

Ed25519 Program

Verify ed25519 signature program. This program takes an ed25519 signature, public key, and message. Multiple signatures can be verified. If any of the signatures fail to verify, an error is returned.

- Program id: `Ed25519SigVerify111111111111111111111111111`
- Instructions: new_ed25519_instruction

The ed25519 program processes an instruction. The first `u8` is a count of the number of signatures to check, which is followed by a single byte padding. After that, the following struct is serialized, one for each signature to check.

BPFLoader 程序

BPF Loader 是指定为网络上所有其他程序的“所有者”的程序，不包括原生程序。 它负责部署、升级和 执行自定义程序。

Sysvar 帐户

Sysvar账户是位于预定义地址的特殊账户，可以访问集群状态数据。 这些帐户使用网络集 群的数据动态更新。 你可以在这里找 到 Sysvar 账户的完整列表。

自定义程序

在Solana，“智能合约”称为 programs。 程序是一个包含可执 行代码的帐户，由一个设置为真的 “executable” 标志表示。

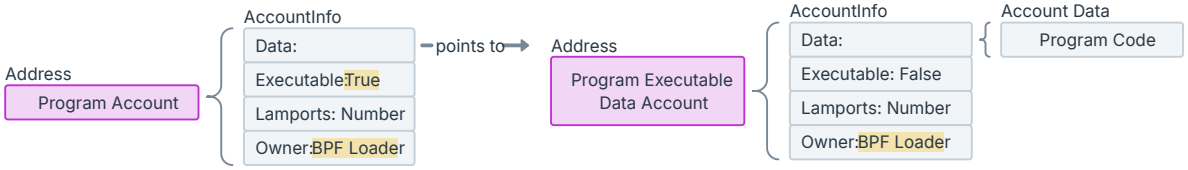
关于程序部署过程的更详细解释，请参阅此文档 的部署程序。

程序账户

当新程序 为已部署时， 从技术上讲，Solana 上创建了三个独立的账户：

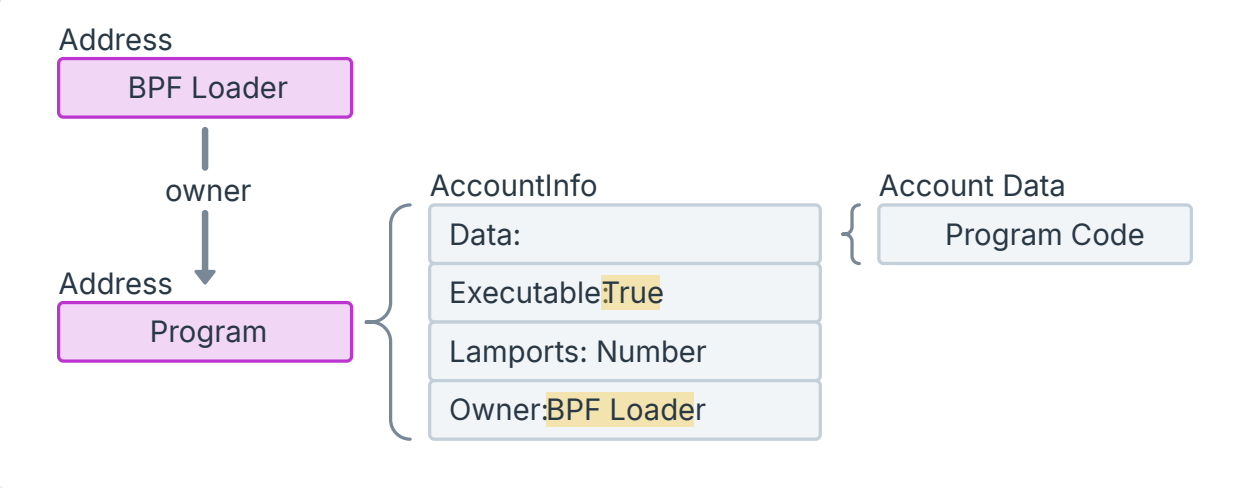
- **程序账户**: 代表链上程序的主账户。 此帐户存储可执行数据帐户的地址(存储编译的程序代码)和程序的更新权限(授权地址更改程序)。
- **程序可执行数据帐户**:包含可执行文件的帐户程序的字节码。
- **缓冲帐户**：一个临时帐户，在程序正在部署或升级时存储字节代码。 处理完成后，数据将转入程序可执行数据账户，并关闭缓冲账户。

例如，这里是 Solana 浏览器上的 Token 扩展 的程序账户链 接及其相应 的程序可执行数据帐户。



程序和可执行数据账户

为了简单起见，你可以将“程序帐户”视为程序本身。



程序账户

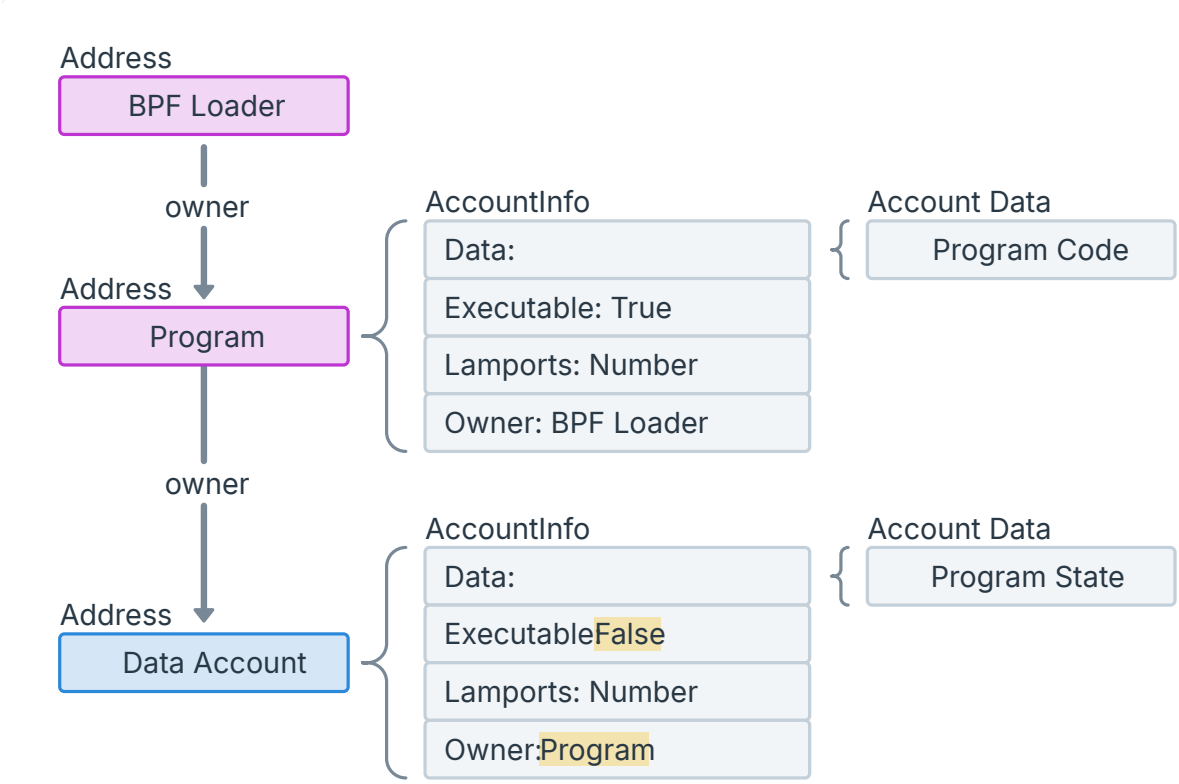
INFO

“程序帐户”的地址通常称为“程序ID”，用于调用该程序。

数据帐户

Solana 程序是“无状态”的，这意味着程序帐户只包含程序可执行字节代码。 要存储和修改 额外数据，必须创建新的帐户。 这些账户通常称为“数据账户”。

数据账户可以存储在所有者程序的代码中定义的任意数据。



数据账户

注意，只有 System Program 可以创建新帐 户。 一旦系统程序创建了一个帐户，它就可以将新帐户的所有权转移到另一个程序。

换言之，为自定义程序创建数据账户需要两个步骤：

1. 调用系统程序来创建一个帐户，然后将所有权转到一个自定义程序
2. 调用现在拥有账户的自定义程序，然后初始化程序代码中定义的账户数据

这个数据账户创建过程常常是抽象为一步，但是了解基础过程是有用的。

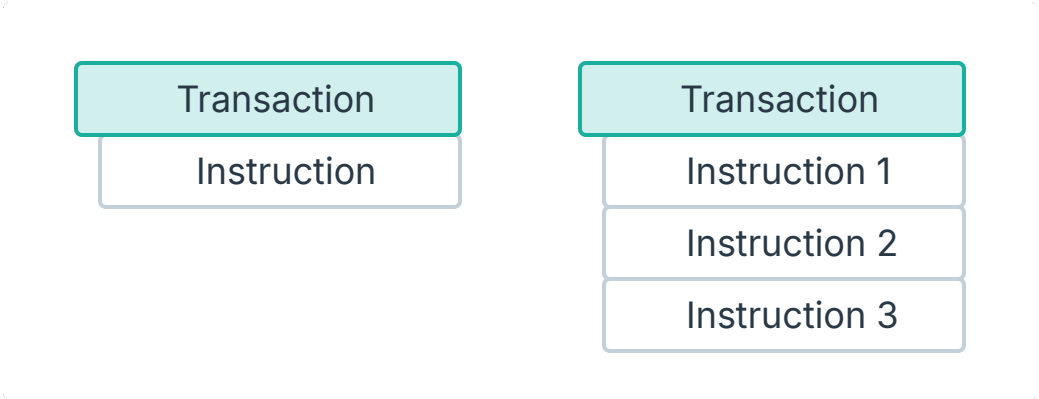
交易和指令

在 Solana上，我们发送 交易与网络交互。 交 易包括一个或多个指令, 每个交易代表一个待 处理的特定操作。 指令的执行逻辑是存储在部署到 Solana 网络的 programs 上。每个程序存储自己的一组指令。

以下是关于交易执行方式的关键细节：

- 执行顺序：如果一个交易包括多个指令，这些指令将按照它们添加到交易中的顺序进行处 理。
- 原子性：交易是原子的，意味着它要么完全完成并且所有指令都成功处理，要么完全失 败。 如果交易中的任何指令失败，则不会执行任何指令。

为简单起见，可以将交易视为请求处理一个或多个指令。



简化交易

你可以将交易想象成一个信封，其中每个指令是您填写并放入信封中的文件。 然后我们发 出信封来处理文档，就像在网络上发送一个交易来处理我们的指令一样。

关键要点

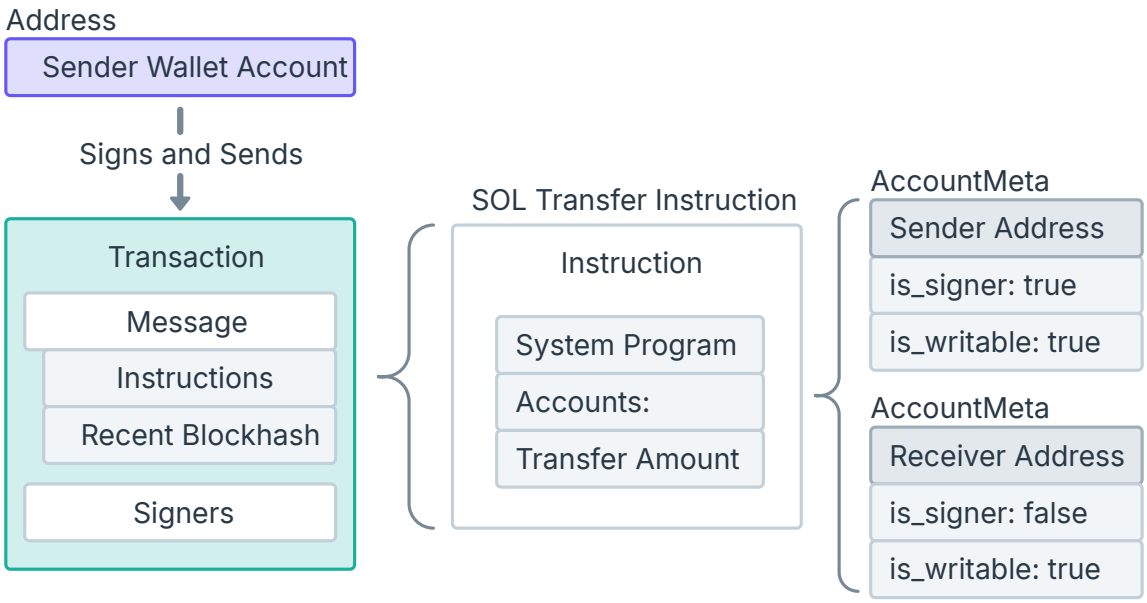
- Solana 交易由与网络上各种程序进行交互的指令组成，其中每个指令代表一个特定操 作。
- 每个指令指定执行指令的程序、指令所需的账户以及指令执行所需的数据。
- 交易中的指令按照它们列出的顺序进行处理。
- 交易是原子的，意味着要么所有指令都成功处理，要么整个交易失败。
- 交易的最大大小为1232字节。

基本示例

以下是代表从发送方向接收方转移 SOL 的单个指令的交易的图示。

Solana 上的个人“钱包”是由系统程序 拥有的账 户。 作为 Solana 账户模型 的一部分，只有拥有帐户的程序才允 许修改帐户上的数据。

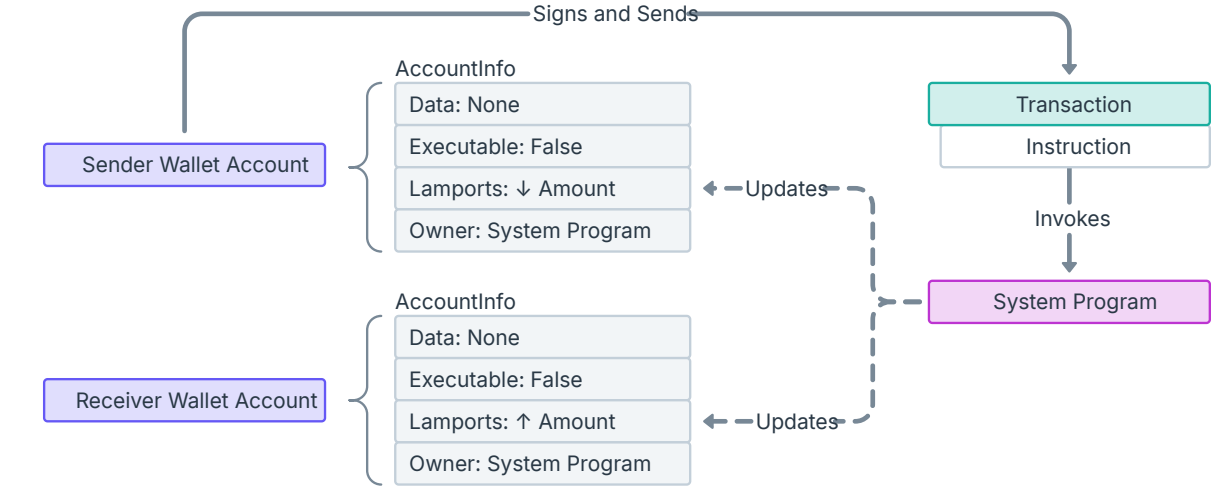
因此，从“钱包”账户转移 SOL 需要发送一个交易来调用 System Program 上的转移指令。



SOL转移

发送者账户必须包含在交易上作为签名者(`is_signner`)，以批准扣除他们的 lamport 余额。 发送者和接收方的账户必须是可变的(`is_wrable`)，因为指令修改了两个账户的 lamport 余额。

交易一旦发送，系统程序将被调用来处理传输的指令。 然后，系统程序相应更新的发送者 和接受者账户的 lamport 余额。



SOL转移过程

简单 SOL 转移

这是一个使用 `SystemProgram.transfer` 方法构建SOL转移指令的 Solana Playground示例：

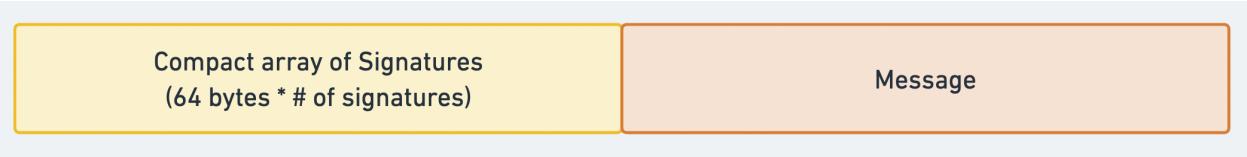
```
// Define the amount to transfer
const transferAmount = 0.01; // 0.01 SOL

// Create a transfer instruction for transferring SOL from walle
const transferInstruction = SystemProgram.transfer({
  fromPubkey: sender.publicKey,
  toPubkey: receiver.publicKey,
  lamports: transferAmount * LAMPORTS_PER_SOL, // Convert transf
});

// Add the transfer instruction to a new transaction
const transaction = new Transaction().add(transferInstruction);
```

运行脚本并检查记录到控制台的交易详细信息。在下面的部分中，我们将详细介绍发生的情 况。 在下面的部分，我们过一遍运行时发生什么的细节。

交易

Solana 的交易包 括：

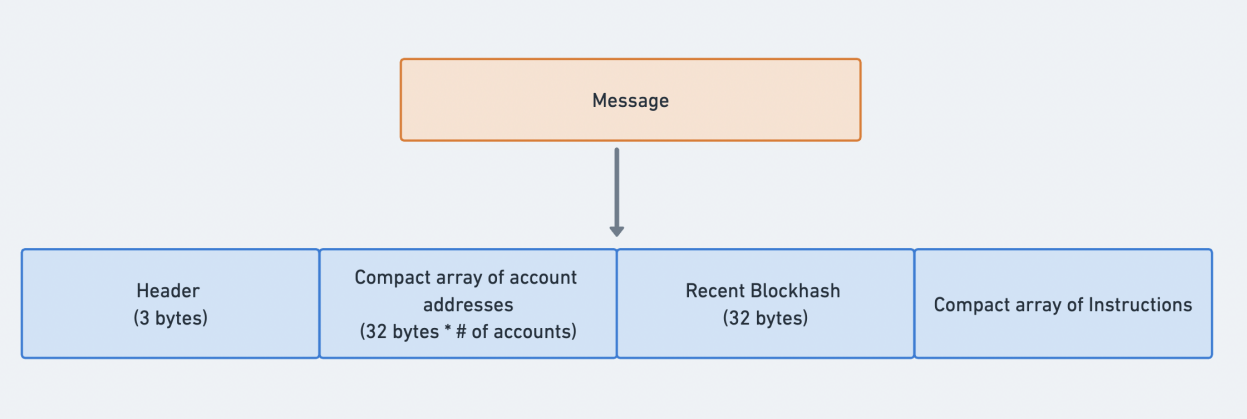
1. 签名： 包含在交易中的签名数组。
2. 消息： 要原子处理的指令列表。



交易格式

交易消息的结构包括：

- 消息头：指定签名者和只读账户的数量。
- 账户地址：指令在交易中所 需的账户地址数组。
- 最新的 Blockhash：作为交易的时间 戳。
- 指令：要执行的指令数组。



交易消息

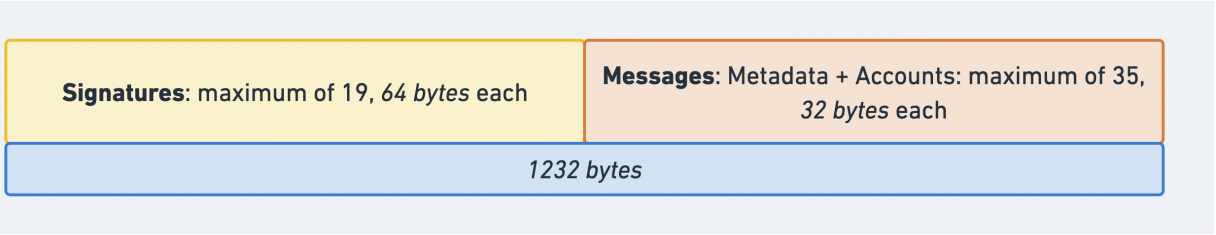
交易大小

Solana网络遵循最大传输单元（MTU）大小为1280字节，与 IPv6 MTU大小约束一致，以确保快速可 靠地通过 UDP 传输集群信息。 在计算必要的标头后（IPv6的40字节和8

字节的片段 头），1232 字节仍可用于数据包， 例如序列化交易。

这意味着 Solana 交易的总大小限制为 1232 字节。签名和消息的组合不能超过此限制。

- 签名：每个签名需要64字节。签名的数量可以根据交易的要求而变化。 签名数量可以不 同，取决于交易的要求。
- 消息：消息包括指令、账户和附加元数据，每个账户需要32字节。 账户加上元数据的组 合大小可以根据交易中包含的指令而变化。

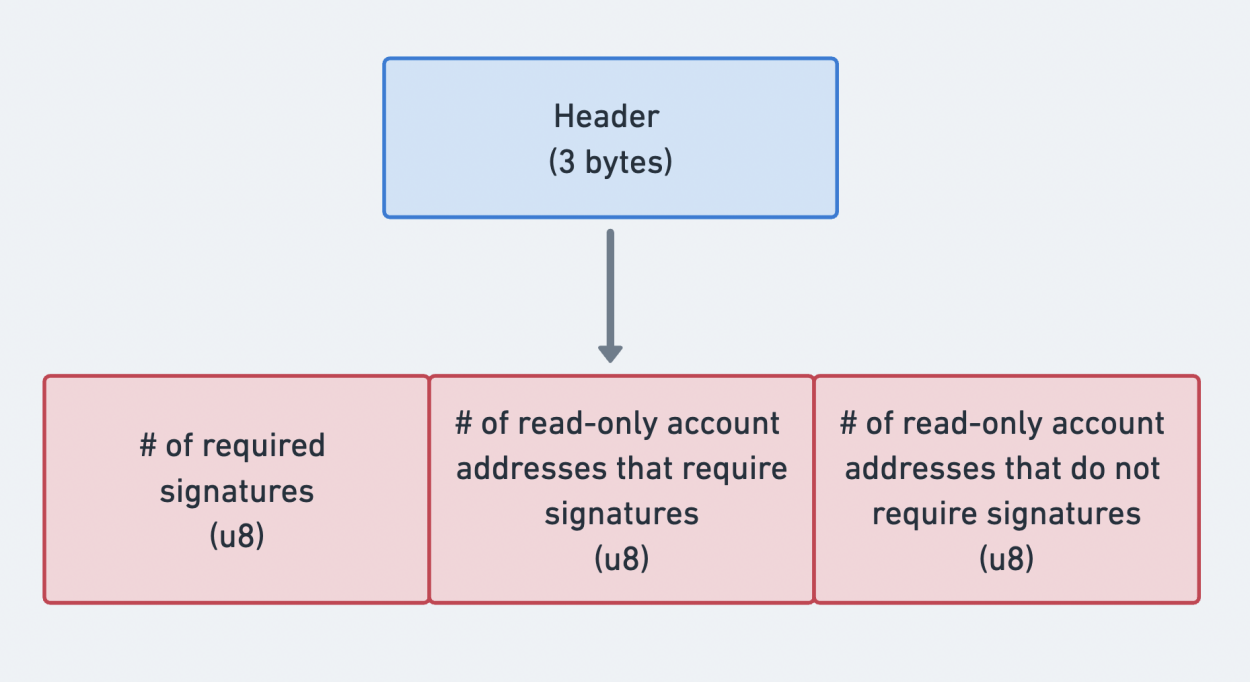


交易格式

消息头

消息头具 体规定了交易账户地址数组中包含的账户的权限。 它由三个字节组成，每个字节含有一个 u8 整数，它们共同规定：

1. 交易所需的签名数量。
2. 需要签名的只读账户地址的数量。
3. 不需要签名的只读账户地址的数量。



消息头

紧凑数组格式

在交易消息的上下文中，紧凑数组指的是以以下格式序列化的数组：

1. 数组的长度，编码为 compact-u16。
2. 编码长度后按顺序列出数组的各个项。

紧凑数组格式

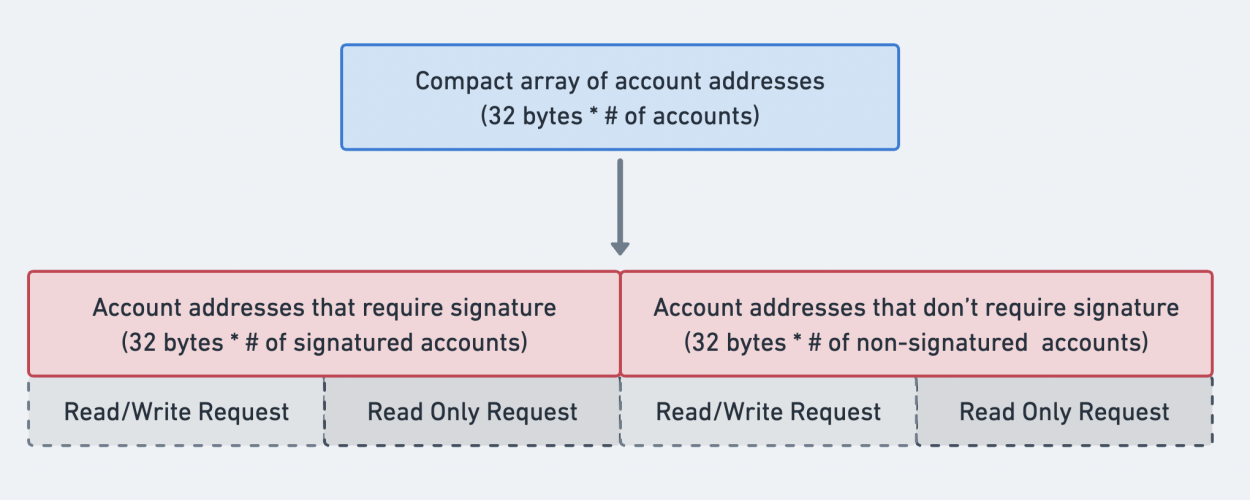
这种编码方法用于指定交易消息中 的账户地址和指令数 组的长度。

账户地址数组

交易消息包括一个数组，其中包含所 有账户地址 ，这些地址是交易内指令所需的。

该数组以一个 compact-u16 编码开 始，后跟按账户权限排序的地址。 消息头中的元数据用于确定每个部分中的账户数量。

- 可写且签名者账户
- 只读且签名者的账户
- 可写且非签名者的账户
- 只读且非签名者的账户



账户地址的紧凑数组

最近的块哈希

所有交易都包括一 个最近的区块哈希， 用作交易的时间戳。 区块哈希用于防止重复和消除过时的交易。

交易的区块哈希的最大年龄为150个区块（假设每个区块时间为400毫秒，约1分钟）。 如果 交易的区块哈希比最新的区块哈希旧150个区块，那么它被视为已过期。 这意味着在特定时 间范围内未处理的交易将永远不会被执行。

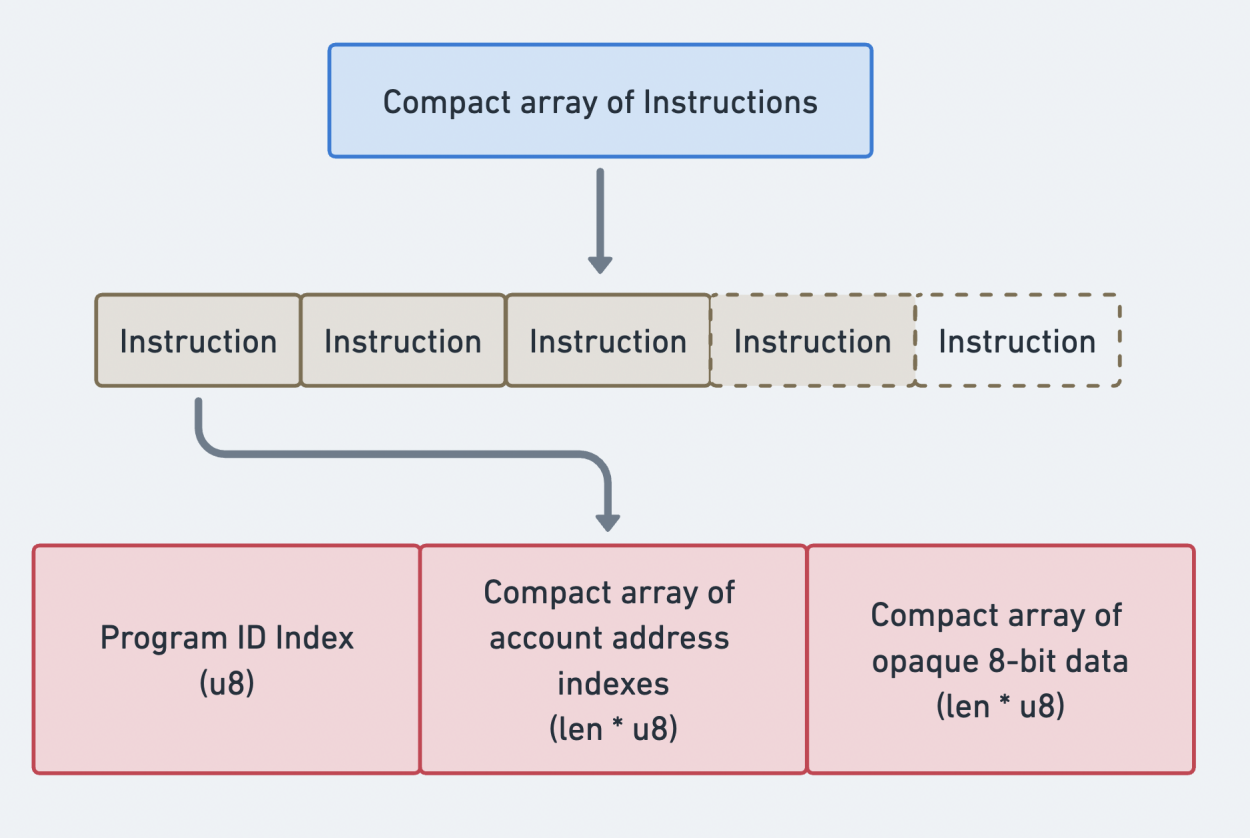
你可以使用`getLatestBlockhash` RPC方法来获 取当前的区块哈希以及区块哈希将有效的最后一个区块高度。 以下是一个 在Solana Playground 上的示例。

指令数组

交易消息包括一个包含所 有请求处理的指令的 数组。 交易消息中的指令采用以下格式：CompiledInstruction。

与账户地址数组类似，这个紧凑数组以一 个compact-u16编码开始，后跟一个 指令数组。数组中的每个指令指定以下信息： 对于每个指令所需的每个账户，必须指定以 下信息：

1. **程序ID**：标识将处理指令的链上程序。这表示为指向账户地址数组中的一个账户地址的u8索引。 这是一个 u8 索引，指向帐户地址数组中的帐户地址。
2. **账户地址索引的紧凑数组**：指向每个指令所需的账户地址数组的u8索引数组。
3. **不透明u8数据的紧凑数组**：特定于被调用程序的u8字节数组。 此数据指定要在程序上调用的指令，以及指令需要的任何附加数据（例如函数参数）。



指令的紧凑数组

示例交易结构

以下是包括单个 SOL转账 指令的交易结构示 例。 它显示了消息细节，包括头部、账户密钥、区块哈希和指令，以及交易的签名。

- `header`：包括用于指定`accountKeys`数组中的读/写和签名者权限的数据。
- `accountKeys`：包括交易中所有指令的账户地址。
- `recentBlockhash`：交易创建时包含的区块哈希。
- `instructions`：包括交易中所有指令。 每个指令中的`account`和`programIdIndex`通 过索引引用`accountKeys`数组。
- `signatures`：包括交易中指令所需的所有签名。 通过使用相应账户的私钥对交易消息进行签名来创建签名。

```
"transaction": {
    "message": {
      "header": {
        "numReadonlySignedAccounts": 0,
        "numReadonlyUnsignedAccounts": 1,
        "numRequiredSignatures": 1
      },
      "accountKeys": [
        "3z9vL1zjN6qyAFHhHQdWYRTFAcy69pJydkZmSFBKHg1R",
        "5snoUseZG8s8CDFHrXY2ZHaCrJYsW457piktDmhyb5Jd",
        "11111111111111111111111111111111"
      ],
      "recentBlockhash": "DzfXchZJoLMG3cNftcf2sw7qatkkuwQf4xH15N
      "instructions": [
        {
          "accounts": [
            0,
            1
          ],
          "data": "3Bxs4NN8M2Yn4TLb",
          "programIdIndex": 2,
          "stackHeight": null
        }
```

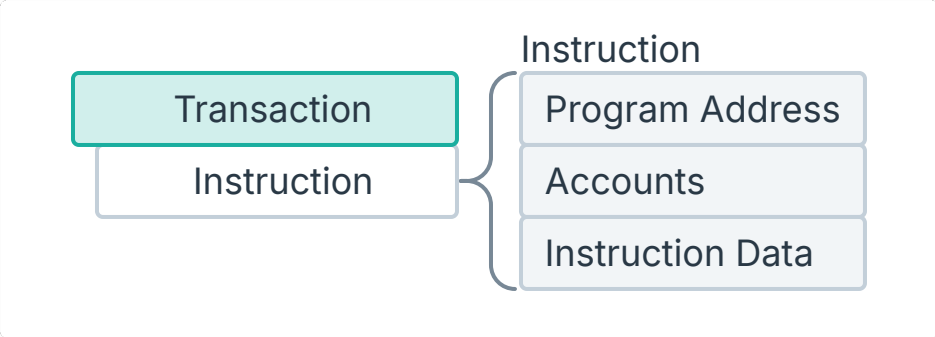
```
      ],
      "indexToProgramIds": {}
    },
    "signatures": [
      "5LrcE2f6uvydKRquEJ8xp19heGxSvqsVbcqUeFoiWbXe8JNip7ftPQNTA
    ]
  }
```


指令

一 个指令是 对链上执行特定操作的请求，也是程序中最小 的连续执行逻辑单元。

构建要添加到交易中的指令时，每个指令必须包括以下信息：

- **程序地址**：指定被调用的程序。
- **账户**：列出每个指令读取或写入的每个账户，包括其他程序，使用 `AccountMeta` 结构。
- **指令数据**：一个字节数组，指定要在程序上调用 的指令处理程序，以及指令处理程序所需 的任何附加数据（函数参数）。



交易指令

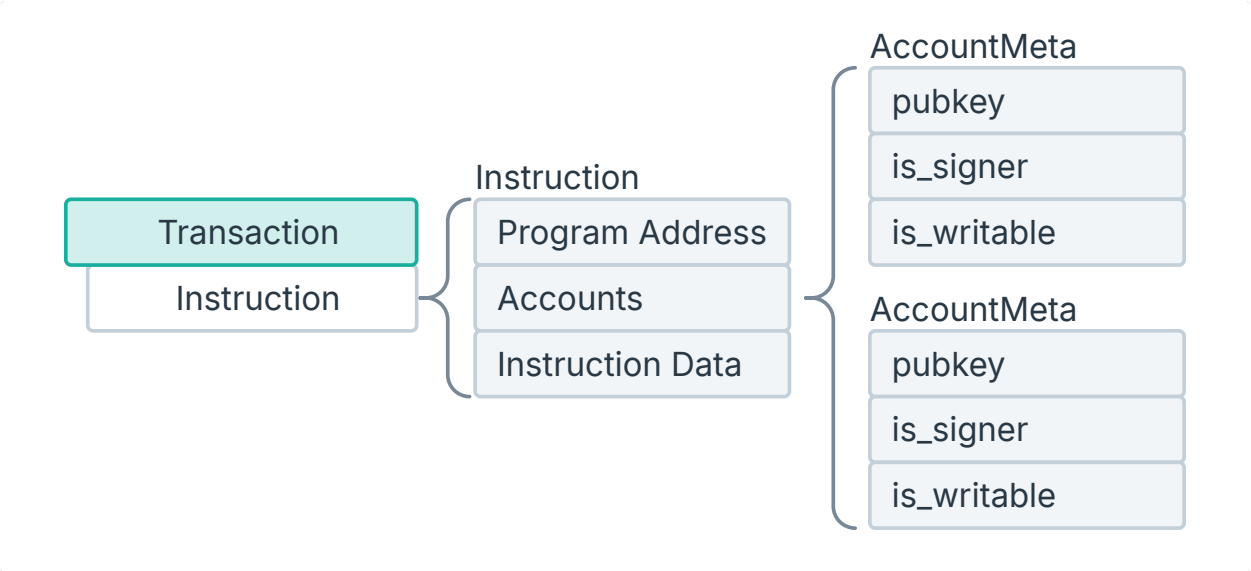
账户元数据

对于每个指令所需的每个账户，必须指定以下信息：

- `pubkey`：账户的链上地址

- `is_signer`：指定账户是否在交易中作为签名者
- `is_writable`：指定账户数据是否将被修改

这些信息被称 为账户元数据 。



账户元数据

通过指定指令所需的所有账户，以及每个账户是否可写，可以并行处理交易。

例如，两个不包含写入相同状态的账户的交易可以同时执行。

示例指令结构

以下是一个 SOL 转账指令结构的示例，详 细说明了指令所需的账户密钥、程序 ID 和数据。

- `keys`：包括每个指令所需的`AccountMeta` （账户元数据）。
- `programId`：包含执行指令的程序地址。
- `data`：指令数据，作为字节缓冲区

```
{
  "keys": [
    {
      "pubkey": "3z9vL1zjN6qyAFHhHQdWYRTFAcy69pJydkZmSFBKHg1R",
```

```
      "isSigner": true,
      "isWritable": true
    },
    {
      "pubkey": "BpvxsLYKQZTH42jjtWHZpsVSa7s6JVwLKwBptPSHXuZc",
      "isSigner": false,
      "isWritable": true
    }
  ],
  "programId": "11111111111111111111111111111111",
  "data": [2,0,0,0,128,150,152,0,0,0,0,0]
}
```

```
// Use Playground cluster connection
const connection = pg.connection;

// Use Playground wallet as sender, generate random keypair as receiver
const sender = pg.wallet.keypair;
const receiver = new Keypair();

// Check and log balance before transfer
const preBalance1 = await connection.getBalance(sender.publicKey);
const preBalance2 = await connection.getBalance(receiver.publicKey);

console.log("sender prebalance:", preBalance1 / LAMPORTS_PER_SOL);
console.log("receiver prebalance:", preBalance2 / LAMPORTS_PER_SOL);
console.log("\n");

// Define the amount to transfer
const transferAmount = 0.01; // 0.01 SOL

// Instruction index for the SystemProgram transfer instruction
const transferInstructionIndex = 2;

// Create a buffer for the data to be passed to the transfer instruction
const instructionData = Buffer.alloc(4 + 8); // uint32 + uint64
// Write the instruction index to the buffer
instructionData.writeUInt32LE(transferInstructionIndex, 0);
// Write the transfer amount to the buffer
instructionData.writeBigUInt64LE(BigInt(transferAmount * LAMPORTS_PER_SOL), 4);

// Manually create a transfer instruction for transferring SOL from sender to receiver
const transferInstruction = new TransactionInstruction({
  keys: [
    { pubkey: sender.publicKey, isSigner: true, isWritable: true },
    { pubkey: receiver.publicKey, isSigner: false, isWritable: true },
  ],
  programId: SystemProgram.programId,
  data: instructionData,
});
```

扩展示例

构建程序指令的详细信息通常由客户端库抽象掉。 但是，如果没有可用的库，你总是可以 手动构建指令。

手动SOL转账

这是一个 Solana Playground 示 例，展示了如何手动构建 SOL 转账指令：

```
// Define the amount to transfer
const transferAmount = 0.01; // 0.01 SOL

// Instruction index for the SystemProgram transfer instruction
const transferInstructionIndex = 2;

// Create a buffer for the data to be passed to the transfer ins
const instructionData = Buffer.alloc(4 + 8); // uint32 + uint64
// Write the instruction index to the buffer
instructionData.writeUInt32LE(transferInstructionIndex, 0);
// Write the transfer amount to the buffer
instructionData.writeBigUInt64LE(BigInt(transferAmount * LAMPORT

// Manually create a transfer instruction for transferring SOL f
const transferInstruction = new TransactionInstruction({
  keys: [
    { pubkey: sender.publicKey, isSigner: true, isWritable: true
    { pubkey: receiver.publicKey, isSigner: false, isWritable: t
  ],
  programId: SystemProgram.programId,
  data: instructionData,
});

// Add the transfer instruction to a new transaction
const transaction = new Transaction().add(transferInstruction);
```

在背后，使用 `SystemProgram.transfer` 方法 的简单例子在功能上等同于上面更详 细的示例。 `SystemProgram.transfer` 方法简单地隐藏了为每个指令所需的账户创建指令 数据缓冲区和 `AccountMeta` 的细节。

程序派生地址 (PDA)

程序派生地址 (PDA) 为 Solana 上的开发人员提供了两种主要用例：

- **确定性帐户地址**

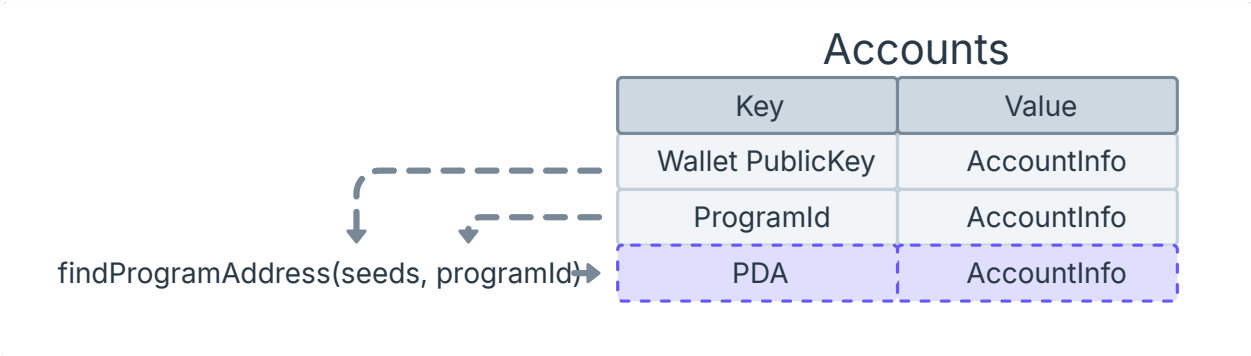
  ：PDA 提供一种机制，通过可选“种子”（预定义输入）和特定程序 ID 的组合来确定性地派生地址。

- **启用程序签名**

  ：Solana 运行时允许程序为源自其程序 ID 的 PDA 进行“签名”。

您可以将 PDA 视为一种根据预定义的一组输入（例如字符串、数字和其他帐户地址）在链上创建类似哈希图的结构的方法。

这种方法的好处是，它消除了记住确切地址的需要。相反，你只需要记住用于推导该地址的特定输入。



程序导出地址

重要的是要明白，仅仅导出程序导出地址 (PDA) 并不会自动在该地址创建链上账户。以 PDA 作为链上地址的账户必须通过用于导出地址的程序明确创建。您可以将导出 PDA 视为在地图上查找地址。仅仅拥有一个地址并不意味着在该位置构建了任何东西。

信息

本节将介绍派生 PDA 的细节。程序如何使用 PDA 进行签名的细节将在跨程序调用 (CPI) 部分中讨论， 因为它需要这两个概念的上下文。

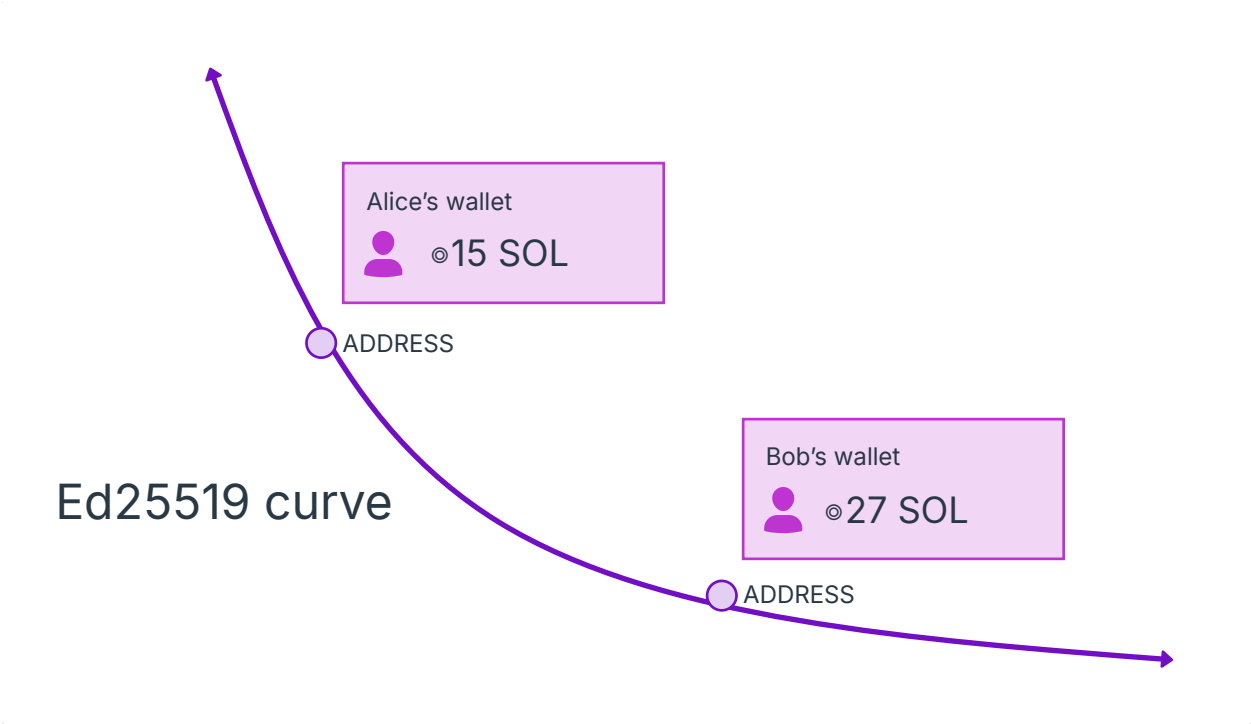
关键点#

- PDA 是使用用户定义的种子、碰撞种子和程序的 ID 的组合确定性地得出的地址。
- PDA 是不符合 Ed25519 曲线的地址，没有相应的私钥。
- Solana 程序可以以编程方式“签名”使用其程序 ID 派生的 PDA。
- 派生 PDA 不会自动创建链上账户。
- 使用 PDA 作为地址的帐户必须通过 Solana 程序内的专用指令明确创建。

什么是 PDA

PDA 是确定性派生的地址，看起来像标准公钥，但没有关联的私钥。这意味着没有外部用户可以为该地址生成有效签名。但是，Solana 运行时允许程序以编程方式为 PDA“签名”，而无需私钥。

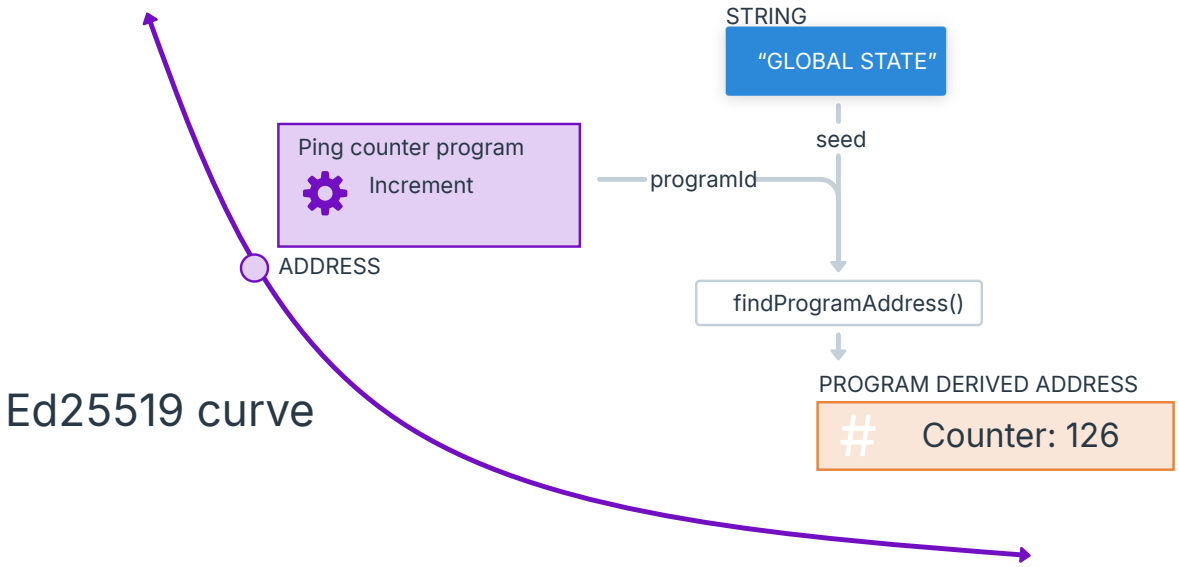
就上下文而言，Solana 密钥 对是 Ed25519 曲线（椭圆曲线密码学）上的点，具有公钥和对应的私钥。我们经常使用公钥作为新链上账户的唯一 ID，并使用私钥进行签名。



曲线地址

PDA 是使用一组预定义的输入故意得出的偏离 Ed25519 曲线的点。不在 Ed25519 曲线上的点没有有效的对应私钥，不能用于加密操作（签名）。

然后，**PDA 可用作链上帐户的地址（唯一标识符），从而提供一种轻松存储、映射和获取程序状态的方法。**



曲线外地址

如何派生 PDA

PDA 的推导需要 3 个输入。

- **可选种子**

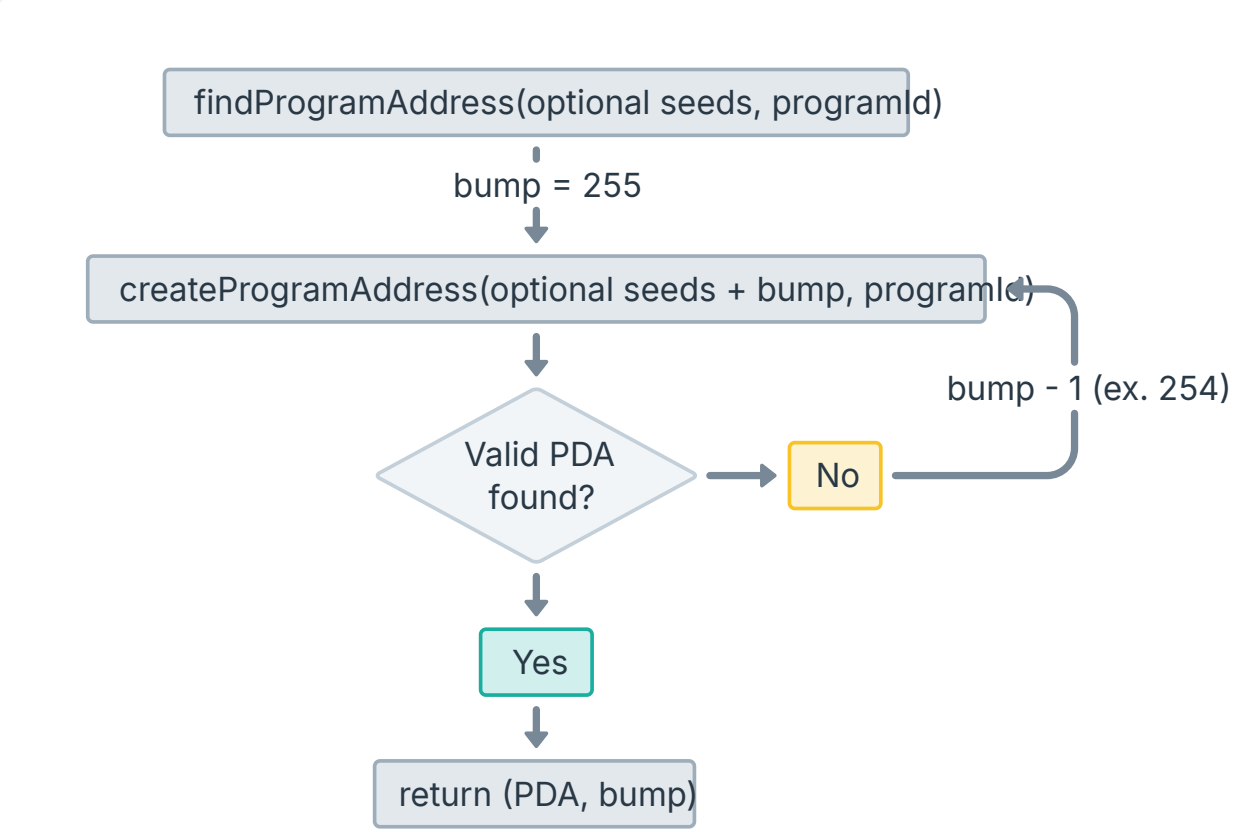
  ：用于派生 PDA 的预定义输入（例如字符串、数字、其他帐户地址）。这些输入将转换为字节缓冲区。

- **碰撞种子**

  ：一个额外的输入（值在 255-0 之间），用于保证生成有效的 PDA（偏离曲线）。在生成 PDA 以“碰撞” Ed25519 曲线上的点时，此碰撞种子（以 255 开头）会附加到可选种子中。碰撞种子有时被称为“随机数”。

- **程序 ID**

  ：PDA 所源自的程序的地址。这也是可以代表 PDA“签名”的程序



PDA 推导

下面的示例包含 Solana Playground 的链接，您可以在浏览器内编辑器中运行示例。

查找程序地址#

要派生 PDA，我们可以使用 `findProgramAddressSync` 中的方法。其他编程语言（例如 Rust `@solana/web3.js` ）中也有此函数的等效函数，但在本节中，我们将使用 Javascript 来介绍示例。

使用该 `findProgramAddressSync` 方法时我们传入：

- 将预定义的可选种子转换为字节缓冲区，以及
- 用于派生 PDA 的程序 ID（地址）

一旦找到有效的 PDA，`findProgramAddressSync` 则返回用于派生 PDA 的地址（PDA）和碰撞种子。

下面的示例在不提供任何可选种子的情况下派生出 PDA。

```
import { PublicKey } from "@solana/web3.js";

const programId = new PublicKey("11111111111111111111111111111111

const [PDA, bump] = PublicKey.findProgramAddressSync([], program

console.log(`PDA: ${PDA}`);
console.log(`Bump: ${bump}`);
```

您可以在Solana Playground上运行此示例 。PDA 和 bump seed 输出将始终相同：

```
PDA: Cu7NwqCXSmsR5vgGA3Vw9uYVViPi3kQvkbKByVQ8nPY9Bump: 255
```

下面的一个例子添加了一个可选种子“helloWorld”。

```
import { PublicKey } from "@solana/web3.js";

const programId = new PublicKey("11111111111111111111111111111111
const string = "helloWorld";

const [PDA, bump] = PublicKey.findProgramAddressSync(
  [Buffer.from(string)],
  programId,
);

console.log(`PDA: ${PDA}`);
console.log(`Bump: ${bump}`);
```

您也可以在 Solana Playground上运行此示例。PDA 和 bump seed 输出将始终相同：

```
PDA: 46GZzzetjCURsdFPb7rcnspbEMnCBXe9kpjrsZAkKb6XBump: 254
```

请注意，碰撞种子为 254。这意味着 255 在 Ed25519 曲线上得出了一个点，并且不是有效的 PDA。

返回的 bump seed `findProgramAddressSync` 是可选种子和程序 ID 的给定组合得出有效 PDA 的第一个值（255-0 之间）。

信息

第一个有效的碰撞种子被称为“规范碰撞”。为了程序安全，建议仅在使用 PDA 时使用规范碰撞。

创建程序地址#

在底层，`findProgramAddressSync` 将迭代地将额外的随机种子 (nonce) 附加到种子缓冲区并调用该 `createProgramAddressSync` 方法。随机种子的起始值为 255，然后逐个减少 1，直到找到有效的 PDA（偏离曲线）。

```
import { PublicKey } from "@solana/web3.js";

const programId = new PublicKey("11111111111111111111111111111111
const string = "helloWorld";
const bump = 254;

const PDA = PublicKey.createProgramAddressSync(
  [Buffer.from(string), Buffer.from([bump])],
  programId,
);

console.log(`PDA: ${PDA}`);
```

在Solana Playground上运行上述示例 。给定相同的种子和程序 ID，PDA 输出将与前一个输出匹配：

```
PDA: 46GZzzetjCURsdFPb7rcnspbEMnCBXe9kpjrsZAkKb6X
```

规范碰撞#

“规范碰撞”是指第一个生成有效 PDA 的碰撞种子（从 255 开始，以 1 为单位递减）。为了程序安全，建议仅使用从规范碰撞生成的 PDA。

使用前面的示例作为参考，下面的示例尝试使用从 255-0 的每个碰撞种子来派生 PDA。

```
import { PublicKey } from "@solana/web3.js";

const programId = new PublicKey("11111111111111111111111111111111
const string = "helloWorld";

// Loop through all bump seeds for demonstration
for (let bump = 255; bump >= 0; bump--) {
  try {
    const PDA = PublicKey.createProgramAddressSync(
      [Buffer.from(string), Buffer.from([bump])],
      programId,
    );
    console.log("bump " + bump + ": " + PDA);
  } catch (error) {
    console.log("bump " + bump + ": " + error);
  }
}
```

在Solana Playground上运行示例，您应该看到以下输出：

```
bump 255: Error: Invalid seeds, address must fall off the curve
bump 254: 46GZzzetjCURsdFPb7rcnspbEMnCBXe9kpjrsZAkKb6X
bump 253: GBNWBGxKmdcd7JrMnBdZke9Fumj9sir4rpbruwEGmR4y
bump 252: THfBMgduMonjaNsCisKa7Qz2cBoG1VCUYHyso7UXYHH
bump 251: EuRrNqJAofo7y3Jy6MGvF7eZAYegqYTwH2dnLCwDDGdP
bump 250: Error: Invalid seeds, address must fall off the curve
...
// remaining bump outputs
```

正如预期的那样，碰撞种子 255 会引发错误，而第一个得出有效 PDA 的碰撞种子是 254。

但是，请注意，碰撞种子 253-251 都派生出具有不同地址的有效 PDA。这意味着，给定相同的可选种子和 `programId`，具有不同值的碰撞种子仍然可以派生出有效的 PDA。

警告

在构建 Solana 程序时，建议包含安全检查，以验证传递给程序的 PDA 是否使用规范的 bump 派生。如果不这样做，可能会引入漏洞，从而允许向程序提供意外帐户。

创建 PDA 账户#

Solana Playground上的这个示例程序 演示了如何使用 PDA 作为新账户的地址来创建账户。示例程序是使用 Anchor 框架编写的。

在该`lib.rs`文件中，您将找到以下程序，其中包含一条指令，用于使用 PDA 作为帐户地址来创建新帐户。新帐户存储了地址`user`和`bump`用于派生 PDA 的种子。

库文件

```
use anchor_lang::prelude::*;

declare_id!("75GJVCJNhaukaa2vCCqhreY31gaphv7XTScBChmr1ueR");

#[program]
pub mod pda_account {
    use super::*;

    pub fn initialize(ctx: Context<Initialize>) -> Result<()>
{
        let account_data = &mut ctx.accounts.pda_account;
        // store the address of the `user`
        account_data.user = *ctx.accounts.user.key;
        // store the canonical bump
        account_data.bump = ctx.bumps.pda_account;
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(mut)]
```

```
    pub user: Signer<'info>,

    #[account(
        init,
        // set the seeds to derive the PDA
        seeds = [b"data", user.key().as_ref()],
        // use the canonical bump
        bump,
        payer = user,
        space = 8 + DataAccount::INIT_SPACE
    )]
    pub pda_account: Account<'info, DataAccount>,
    pub system_program: Program<'info, System>,
}

#[account]

#[derive(InitSpace)]
pub struct DataAccount {
    pub user: Pubkey,
    pub bump: u8,
}
```

用于派生 PDA 的种子包括硬编码字符串和指令中提供的账户 `data` 地址。Anchor 框架会自动派生出规范种子。`userbump`

```
#[account(
    init,
    seeds = [b"data", user.key().as_ref()],
    bump,
    payer = user,
    space = 8 + DataAccount::INIT_SPACE
)]
pub pda_account: Account<'info, DataAccount>,
```

该`init`约束指示 Anchor 调用系统程序以使用 PDA 作为地址创建新帐户。在底层，这是通过 CPI 完成 的。

```
#[account(
    init,
    seeds = [b"data", user.key().as_ref()],
    bump,
    payer = user,
    space = 8 + DataAccount::INIT_SPACE
)]
pub pda_account: Account<'info, DataAccount>,
```

在位于上面提供的 Solana Playground 链接内的测试文件 (`pda-account.test.ts`) 中，您将找到派生 PDA 的 Javascript 等效代码。

```
const [PDA] = PublicKey.findProgramAddressSync(
  [Buffer.from("data"), user.publicKey.toBuffer()],
  program.programId,
);
```

然后发送交易以调用`initialize`指令，使用 PDA 作为地址创建一个新的链上账户。发送交易后，PDA 将用于获取在该地址创建的链上账户。

```
it("Is initialized!", async () => {
  const transactionSignature = await program.methods
    .initialize()
    .accounts({
      user: user.publicKey,
      pdaAccount: PDA,
    })
    .rpc();

  console.log("Transaction Signature:", transactionSignature);
});

it("Fetch Account", async () => {
```

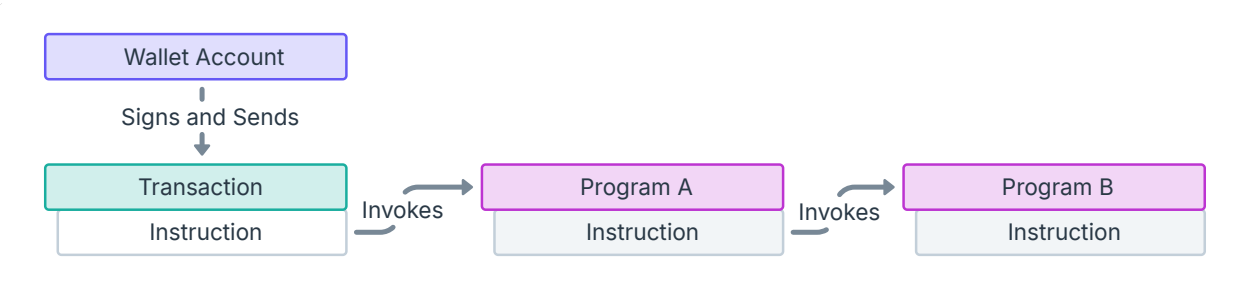
```
  const pdaAccount = await program.account.dataAccount.fetch(PDA
  console.log(JSON.stringify(pdaAccount, null, 2));
});
```

请注意，如果您`initialize`使用同一地址作为种子多次调用该指令`user`，则交易将失败。这是因为派生地址上已经存在一个帐户。

跨程序调用 (CPI)

跨程序调用 (CPI) 是指一个程序调用另一个程序的指令。此机制允许 Solana 程序具有可组合性。

您可以将指令视为程序向网络公开的 API 端点，将 CPI 视为内部调用另一个 API 的一个 API。



跨程序调用

当一个程序向另一个程序发起跨程序调用 (CPI) 时：

- 调用调用程序 (A) 的初始交易的签名者特权扩展到被调用程序 (B)
- 被调用程序（B）可以对其他程序进行进一步的 CPI，最大深度为 4（例如 B→C、C→D）
- 这些程序可以根据其程序 ID为PDA进行“签名”

信息

Solana 程序运行时定义了一个名为 的常量`max_invoke_stack_height`，其值设置 为 5。这表示程序指令调用堆栈的最大高度。事务指令的堆栈高度从 1 开始，每次程序调用另一条指令时，堆栈高度都会增加 1。此设置有效地将 CPI 的调用深度限制为 4。

关键点

- CPI 使 Solana 程序指令能够直接调用另一个程序上的指令。
- 调用者程序的签名者权限扩展到被调用者程序。
- 在制作 CPI 时，程序可以代表根据其自身程序 ID 派生的 PDA“签名”。
- 被调用程序可以对其他程序进行额外的 CPI，最大深度为 4。

如何编写 CPI

编写 CPI 指令遵循与构建 添加到交易的指令相同的模式。在底层，每个 CPI 指令必须指定以下信息：

- **程序地址**：指定被调用的程序
- **帐户**：列出指令读取或写入的每个帐户，包括其他程序
- **指令数据**：指定要调用程序中的哪条指令，以及该指令所需的任何其他数据（函数参数）

根据您调用的程序，可能会有带有辅助函数的包可用于构建指令。然后，程序使用包中的以下任一函数执行 CPI `solana_program`：

- `invoke`
  - 当没有 PDA 签名者时使用
- `invoke_signed`
  - 当调用程序需要使用从其程序 ID 派生的 PDA 进行签名时使用

基本 CPI

该 `invoke` 函数用于制作不需要 PDA 签名者的 CPI。制作 CPI 时，提供给调用者程序的签名者会自动扩展到被调用者程序。

```
pub fn invoke(
    instruction: &Instruction,
    account_infos: &[AccountInfo<'_>]
) -> Result<(), ProgramError>
```

这是 Solana Playground `invoke` 上的一个示例程序，它使用函数调用系统程序上的传输指令来 制作 CPI 。您还可以参考基本 CPI 指南了解更多详细信息。

```
use anchor_lang::prelude::*;
use anchor_lang::solana_program::{program::invoke, system_instruction};

declare_id!("55xRZZnhSk1aN6seNTj75mThJEjZkBRYPQJ8qvKVh1eC");

#[program]
pub mod cpi_invoke {
    use super::*;

    pub fn sol_transfer(ctx: Context<SolTransfer>, amount: u64) -> Result<()> {
        let from_pubkey: AccountInfo = ctx.accounts.sender.to_account_info();
        let to_pubkey: AccountInfo = ctx.accounts.recipient.to_account_info();
        let program_id: AccountInfo = ctx.accounts.system_program.to_account_info();

        let instruction: &Instruction =
            &system_instruction::transfer(from_pubkey: &from_pubkey.key(), to_pubkey: &to_pubkey.key(), lamports: amount);

        invoke(instruction, account_infos: &[from_pubkey, to_pubkey, program_id])?;
        Ok(())
    }
}

0 implementations
#[derive(Accounts)]
pub struct SolTransfer<'info> {
    #[account(mut)]
    sender: Signer<'info>,
    #[account(mut)]
    recipient: SystemAccount<'info>,
    system_program: Program<'info, System>,
}
```

具有 PDA 签名者的 CPI

此 `invoke_signed` 函数用于制作需要 PDA 签名者的 CPI。用于派生签名者 PDA 的种子将作为 传递到函数 `invoke_signed` 中 `signer_seeds`。

您可以参考程序派生地址页面来了解有关如何派生 PDA 的详细信息。

```
pub fn invoke_signed(
    instruction: &Instruction,
    account_infos: &[AccountInfo<'_>],
```

```
    signers_seeds: &[&[&[u8]]]
) -> Result<(), ProgramError>
```

运行时使用授予调用程序的权限来确定可以向被调用程序扩展哪些权限。此处的权限指的是签名者和可写帐户。例如，如果调用程序正在处理的指令包含签名者或可写帐户，则调用程序可以调用也包含该签名者和/或可写帐户的指令。

尽管 PDA 没有私钥，但它们仍可通过 CPI 充当指令中的签名者。要验证 PDA 是否来自调用程序，必须将用于生成 PDA 的种子包含在内`signers_seeds`。

处理 CPI 时，Solana 运行时 在内部`create_program_address` 使用调用程序的`signers_seeds` 和进行调用`program_id`。如果找到有效的 PDA，则该地址将 添加为有效签名者。

这是Solana Playground上的一个示例程序 ，它使用`invoke_signed`函数通过 PDA 签名者调用系统程序上的转账指令来 制作 CPI 。您可以参考PDA 签名者 CPI 指南 了解更多详细信息。

```rust
use anchor_lang::prelude::*;
use anchor_lang::solana_program::{program::invoke_signed, system_instruction};

declare_id!("EyxvVL2akUZZHx4DXzYzCroKLmigPrS2WgpSetKzM9wh");

#[program]
pub mod cpi_invoke_signed {
    use super::*;

    pub fn sol_transfer(ctx: Context<SolTransfer>, amount: u64) -> Result<()> {
        let from_pubkey: AccountInfo = ctx.accounts.pda_account.to_account_info();
        let to_pubkey: AccountInfo = ctx.accounts.recipient.to_account_info();
        let program_id: AccountInfo = ctx.accounts.system_program.to_account_info();

        let seed: Pubkey = to_pubkey.key();
        let bump_seed: u8 = ctx.bumps.pda_account;
        let signer_seeds: &[&[&[u8]]] = &[&[b"pda", seed.as_ref(), &[bump_seed]]];

        let instruction: &Instruction =
            &system_instruction::transfer(from_pubkey: &from_pubkey.key(), to_pubkey: &to_pubkey.key(), lamports: amount);

        invoke_signed(
            instruction,
            account_infos: &[from_pubkey, to_pubkey, program_id],
            signers_seeds: signer_seeds,
        )?;
        Ok(())
    }
}

0 implementations
#[derive(Accounts)]
pub struct SolTransfer<'info> {
    #[account(
        mut,
        seeds = [b"pda", recipient.key().as_ref()],
        bump,
    )]
    pda_account: SystemAccount<'info>,
```

Solana 上的代币

代币是代表不同类别资产所有权的数字资产。代币化使产权数字化，成为管理可替代资产和非可替代资产的基本组成部分。

- 可替代代币代表相同类型和价值的可互换和可分割的资产（例如 USDC）。
- 非同质化代币（NFT）代表不可分割资产（例如艺术品）的所有权。

本节将介绍 Solana 上代币的基本表示方式。这些代币被称为 SPL（Solana 程序库）代币。

- 代币程序包含与网络上的代币（可替代和不可替代）交互的所有指令逻辑。
- 铸币账户代表一种特定类型的代币，并存储有关代币的全局元数据，例如总供应量和铸币权限（授权创建代币新单位的地址）。
- 代币账户跟踪个人所有权，即特定地址拥有多少特定类型的代币（铸币账户）单位。

信息

目前有两个版本的代币程序。原始 代币程序 和 代币扩展程序 (Token2022)。代币扩展程序的功能与原始代币程序相同，但具有附加功能和改进。代币扩展程序是用于创建新代币（铸造账户）的推荐版本。

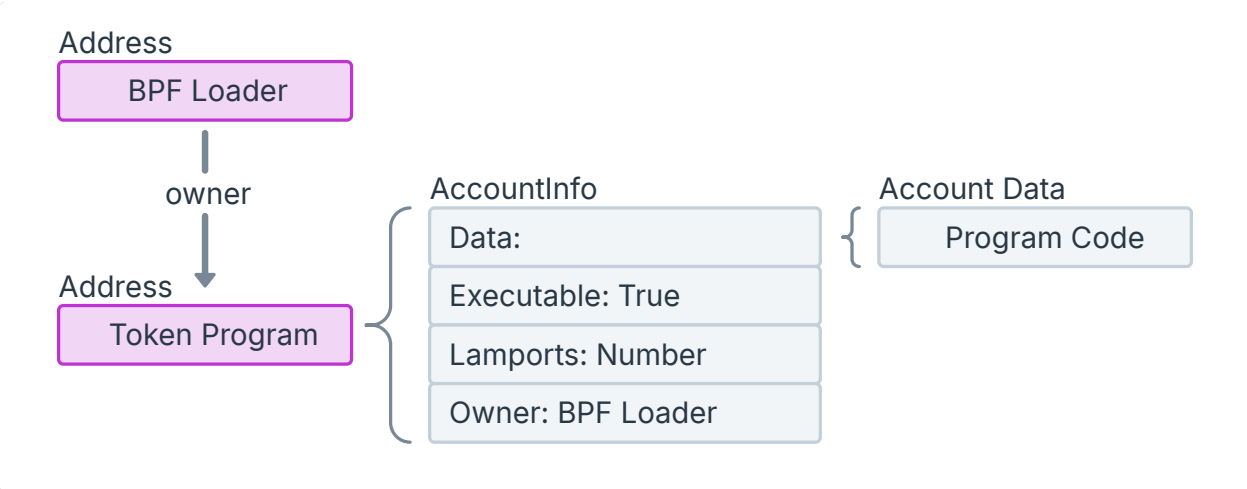
关键点#

- 代币代表对可替代（可互换）或不可替代（独特）资产的所有权。
- 代币程序包含与网络上的可替代代币和不可替代代币进行交互的所有指令。
- 代币扩展程序是代币程序的新版本，它在保持相同核心功能的同时包含附加功能。
- Mint 账户代表网络上的唯一代币，并存储总供应量等全局元数据。
- 代币账户追踪特定铸币账户的代币个人所有权。
- 关联代币账户是使用源自所有者和铸币账户地址的地址创建的代币账户。

代币程序#

代 币程序 包含与网络上的代币（可替代和不可替代）交互的所有指令逻辑。 Solana 上的所有代币实际上都是 代币程序拥有的数据账户。

您可以在此处找到代币程序说明的完整列表 。



代币程序

一些常用的指令包括：

- `InitializeMint`

  ：创建一个新的铸币账户来代表一种新类型的代币。

- `InitializeAccount`

  ：创建一个新的代币账户来持有特定类型代币（mint）的单位。

- `MintTo`

  ：创建特定类型代币的新单位并将其添加到代币账户。这会增加代币的供应量，并且只能由铸币账户的铸币机构完成。

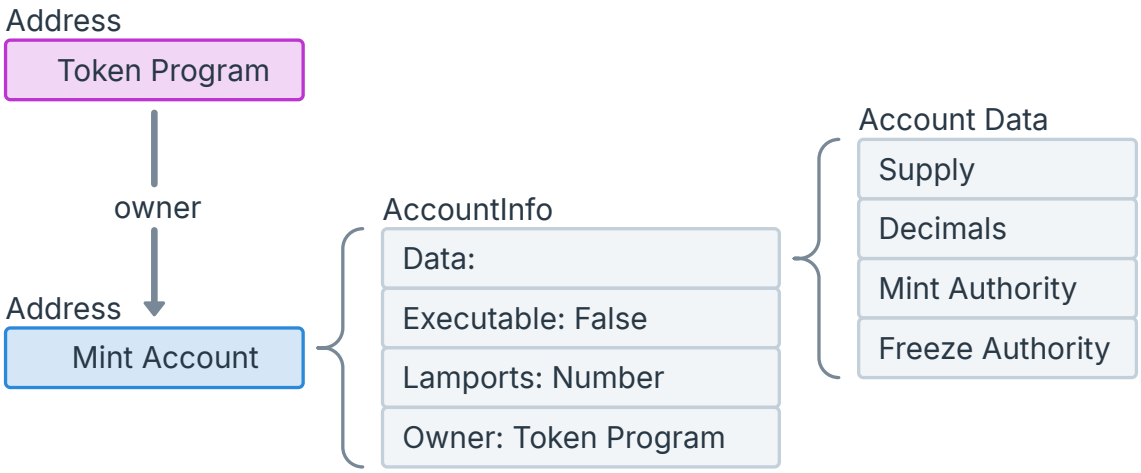
- `Transfer`

  ：将特定类型的代币单位从一个代币账户转移到另一个代币账户。

Mint 帐户#

Solana 上的代币由代币计划拥有的Mint 账户地址唯一标识 。此帐户实际上是特定代币的全局计数器，并存储以下数据：

- 供应量：代币总供应量
- 小数：token 的小数精度
- 铸币权：该账户有权创建新的代币单位，从而增加供应量
- 冻结权限：有权冻结从“代币账户”转出的代币的账户



Mint 帐户

每个 Mint 帐户中存储的完整详细信息包括以下内容：

```
pub struct Mint {
    /// Optional authority used to mint new tokens. The mint aut
    /// be provided during mint creation. If no mint authority i
    /// then the mint has a fixed supply and no further tokens m
    /// minted.
    pub mint_authority: COption<Pubkey>,
    /// Total supply of tokens.
    pub supply: u64,
    /// Number of base 10 digits to the right of the decimal pla
    pub decimals: u8,
    /// Is `true` if this structure has been initialized
```

```
    pub is_initialized: bool,
    /// Optional authority to freeze token accounts.
    pub freeze_authority: COption<Pubkey>,
}
```

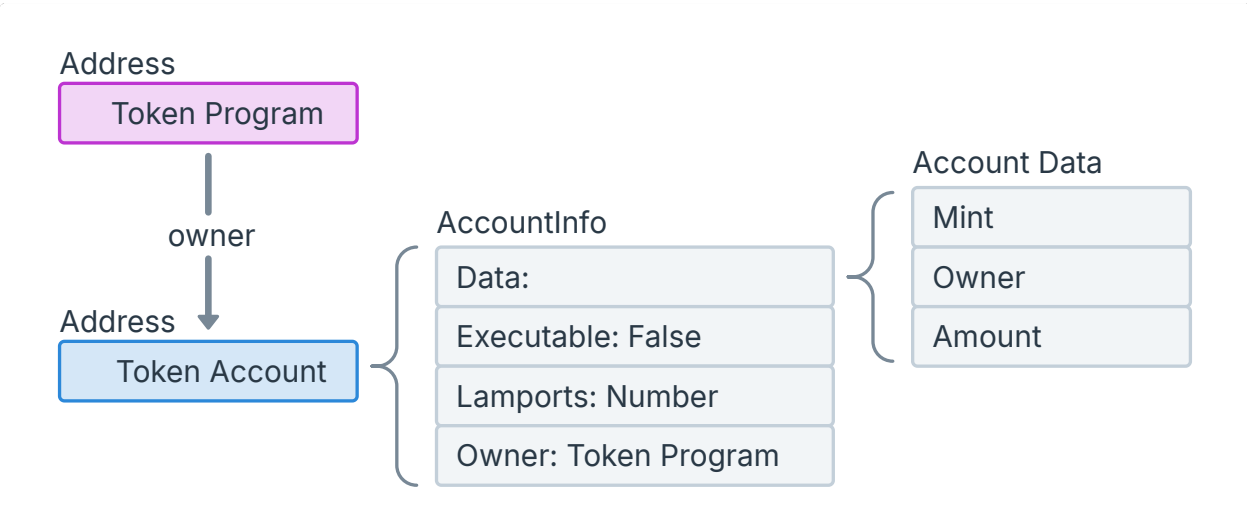
作为参考，这里是 Solana Explorer 到 USDC Mint 账户的链接。

代币账户#

为了追踪特定代币的每个单位的个人所有权，必须创建代币计划拥有的另一种数据账户。此账户称为 代币账户。

代币账户中存储的最常见引用数据包括以下内容：

- Mint：代币账户持有的代币类型
- 所有者：有权将代币转出代币账户的账户
- 金额：代币账户当前持有的代币单位



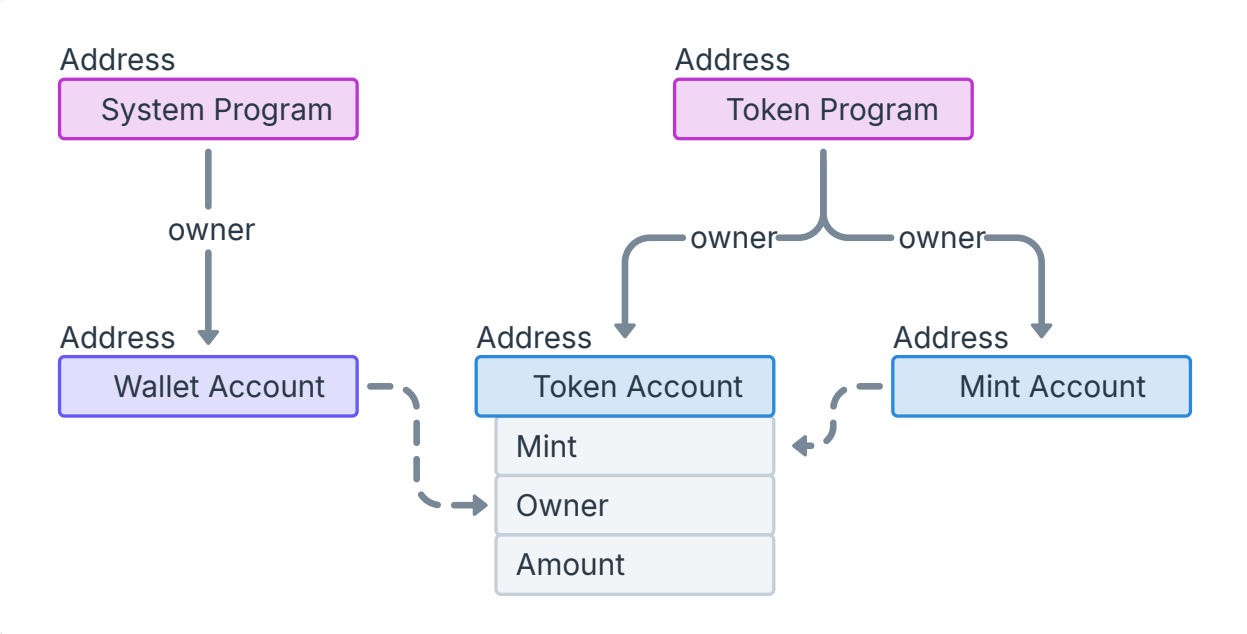
代币账户

每个代币账户上存储的完整详细信息包括以下内容：

```
pub struct Account {
    /// The mint associated with this account
    pub mint: Pubkey,
```

```
    /// The owner of this account.
    pub owner: Pubkey,
    /// The amount of tokens this account holds.
    pub amount: u64,
    /// If `delegate` is `Some` then `delegated_amount` represer
    /// the amount authorized by the delegate
    pub delegate: COption<Pubkey>,
    /// The account's state
    pub state: AccountState,
    /// If is_native.is_some, this is a native token, and the va
    /// rent-exempt reserve. An Account is required to be rent-e
    /// the value is used by the Processor to ensure that wrappe
    /// accounts do not drop below this threshold.
    pub is_native: COption<u64>,
    /// The amount delegated
    pub delegated_amount: u64,
    /// Optional authority to close the account.
    pub close_authority: COption<Pubkey>,
}
```

如果钱包想要拥有某种代币的单位，则需要为特定类型的代币 (mint) 创建一个代币账户，并将钱包指定为代币账户的所有者。钱包可以为同一种类型的代币创建多个代币账户，但每个代币账户只能由一个钱包拥有并持有一种类型的代币的单位。



帐户关系

信息

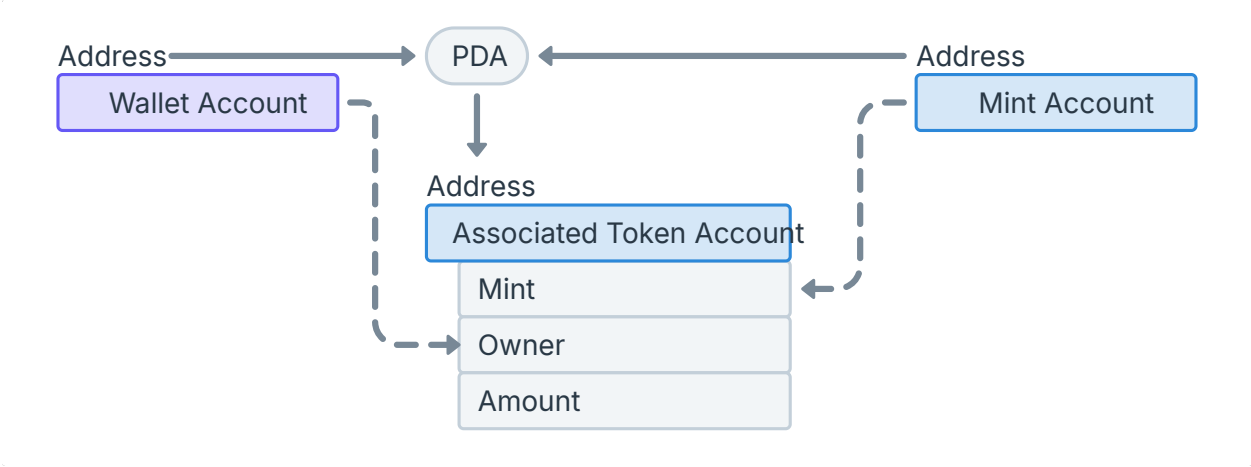
请注意，每个代币账户的数据都包含一个字段，用于标识谁对该特定代币账户拥有权限。这与AccountInfo `owner` 中指定的程序所有者不同 ，后者是所有代币账户的代币程序。

关联代币账户#

为了简化查找特定铸币厂和所有者的代币账户地址的过程，我们经常使用关联代币账户。

关联代币账户是一种代币账户，其地址由所有者的地址和铸币账户的地址确定性地得出。您可以将关联代币账户视为特定铸币者和所有者的“默认”代币账户。

重要的是要理解关联代币账户并不是不同类型的代币账户。它只是一个具有特定地址的代币账户。



关联代币账户

这引入了 Solana 开发中的一个关键概念： 程序派生地址 (PDA)。从概念上讲，PDA 提供了一种使用一些预定义输入生成地址的确定性方法。这使我们能够在以后轻松找到帐户的地址。

以下是Solana Playground 示例，它导出 USDC 关联代币账户地址和所有者。它将始终 为同一铸币厂和所有者生成相同的地址。

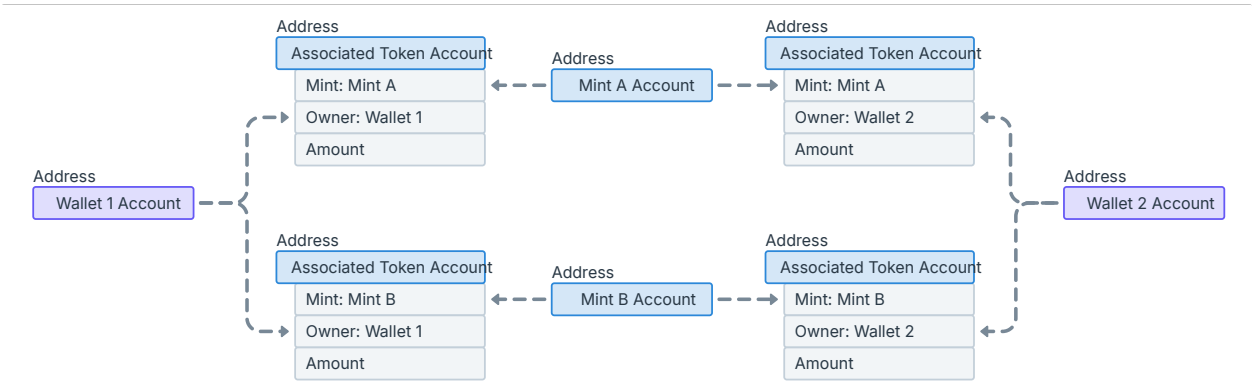
```
import { getAssociatedTokenAddressSync } from "@solana/spl-toker

const associatedTokenAccountAddress = getAssociatedTokenAddressS
  USDC_MINT_ADDRESS,
  OWNER_ADDRESS,
);
```

具体来说，关联代币账户的地址是使用以下输入得出的。这是一个 Solana Playground 示例，它生成与上一个示例相同的地址。

```
import { PublicKey } from "@solana/web3.js";

const [PDA, bump] = PublicKey.findProgramAddressSync(
  [
    OWNER_ADDRESS.toBuffer(),
    TOKEN_PROGRAM_ID.toBuffer(),
    USDC_MINT_ADDRESS.toBuffer(),
```

```
    ],
    ASSOCIATED_TOKEN_PROGRAM_ID,
  );
```

如果两个钱包要存放相同类型的代币，则每个钱包都需要有自己的代币账户来存放特定的铸币账户。下图展示了这种账户关系。



账户关系扩展

代币示例

CLI可用于试验 SPL 代币。在下面的示例中，我们将使用 Solana Playground终端直接在浏览器中运行 CLI 命令`spl-token`，而无需在本地安装 CLI。

创建代币和账户需要 SOL 作为账户租金押金和交易费。如果这是您第一次使用 Solana Playground，请创建一个 Playground 钱包并`solana airdrop`在 Playground 终端中运行命令。您还可以使用公共 网络水龙头获取 devnet SOL 。

```
solana airdrop 2
```

运行`spl-token --help`以获取可用命令的完整描述。

```
spl-token --help
```

或者，您可以使用以下命令在本地安装 spl-token CLI。这需要先安装 Rust。

信息

在以下部分中，运行 CLI 命令时显示的帐户地址将与下面显示的示例输出不同。请在继续操作时使用 Playground 终端中显示的地址。例如，输出的地址是您的`create-token`Playground 钱包设置为铸币机构的铸币帐户。

创建新代币

要创建新的代币（mint 账户），请在 Solana Playground 终端中运行以下命令。

```
spl-token create-token
```

您应该会看到类似于下面的输出。您可以 使用 和在Solana Explorer上检查代币和交易详细信息。`AddressSignature`

在下面的示例输出中，新代币的唯一标识符（地址）
是 `99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg`。

终端

```
Creating token 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg

Address:  99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg
Decimals:  9

Signature: 44fvKfT1ezBUwdzrCys3fvCdFxbLMnNvBstds76QZyE6cXag5N
upBprSXwxPTzzjrC3cA6nvUZaLFTvmcKyzxrm1
```

新代币最初没有供应。您可以使用以下命令检查代币的当前供应量：

`spl-token supply <TOKEN_ADDRESS>`

对新创建的令牌运行`supply`命令将返回以下值`0`：

`spl-token supply 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg`

在底层，创建一个新的 Mint 账户需要发送一个包含两个指令的交易。以下是 Solana Playground上的 Javascript 示例。

1. 调用系统程序来创建一个具有足够空间用于存储 Mint 账户数据的新账户，然后将所有权转移到代币程序。
2. 调用Token Program将新账户的数据初始化为Mint账户

创建代币账户

要持有特定代币的单位，您必须首先创建一个 代币账户。要创建新的代币账户，请使用以下命令：

```
spl-token create-account [OPTIONS] <TOKEN_ADDRESS>
```

例如，在 Solana Playground 终端中运行以下命令：

```
spl-token create-account 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75F
```

返回以下输出：

- `AfB7uwBEsGtrrBqPTVqEgzWed5XdYfM1psPNLmf7EeX9`

  是为保存命令中指定的代币单位而创建的代币账户的地址`create-account`。

```
Creating account AfB7uwBEsGtrrBqPTVqEgzWed5XdYfM1psPNLmf7EeX9

Signature: 2BtrynuCLX9CNofFiaw6Yzbx6hit66pup9Sk7aFjwU2NEbFz7N
CHD9w9sWhrCfEd73XveAGK1DxFpJoQZPXU9tS1
```

默认情况下，该`create-account`命令会创建一个 关联的代币账户，并以您的钱包地址作为代币账户所有者。

您可以使用以下命令创建具有不同所有者的令牌帐户：

```
spl-token create-account --owner <OWNER_ADDRESS> <TOKEN_ADDRESS>
```

例如运行以下命令：

```
spl-token create-account --owner 2i3KvjDCZWxBsqcxBHpdEaZYQwQSYE6
```

返回以下输出：

- `Hmyk3FSw4cfsuAes7sanp2oxSkE9ivaH6pMzDzbacqmtcreate-account99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg-owner2i3KvjDCZWxBsqcxBHpdEaZYQwQSYE6LXUMx5VjY5XrR`

  是为保存命令 ( )中指定的代币单位而创建的代币账户的地址，并由标志 ( )

后指定的地址拥有。当你需要为另一个用户创建代币账户时，这很有用。

终端

```
Creating account Hmyk3FSw4cfsuAes7sanp2oxSkE9ivaH6pMzDzbacqmt

Signature: 44vqKdfzspT592REDPY4goaRJH3uJ3Ce13G4BCuUHg35dVUbHu
GTHvqn4ZjYF9BGe9QrjMfe9GmuLkQhSZCBQuEt
```

在底层，创建关联代币账户需要一条调用 关联代币程序的指令。以下是Solana Playground上的 Javascript 示例 。

关联令牌程序使用跨程序调用 来处理：

- 调用系统程序

  创建一个新账户，使用提供的PDA作为新账户的地址

- 调用Token程序

  为新账户初始化Token账户数据。

或者，使用随机生成的密钥对（不是关联代币账户）创建新代币账户需要发送包含两个指令的交易。以下是Solana Playground上的 Javascript 示例 。

1. 调用系统程序来创建一个具有足够空间用于存储令牌账户数据的新账户，然后将所有权转移给令牌程序。
2. 调用Token Program将新账户的数据初始化为Token账户

铸造代币#

要创建代币的新单位，请使用以下命令：

```
spl-token mint [OPTIONS] <TOKEN_ADDRESS> <TOKEN_AMOUNT> [--] [RE
```

例如运行以下命令：

```
spl-token mint 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg 100
```

返回以下输出：

- `99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg`是铸造代币的铸币账户的地址（增加总供应量）。
- `AfB7uwBEsGtrrBqPTVqEgzWed5XdYfM1psPNLmf7EeX9`是您的钱包代币账户的地址，代币单位正在被铸造到该地址（数量增加）。

终端

```
Minting 100 tokens
  Token: 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg
  Recipient: AfB7uwBEsGtrrBqPTVqEgzWed5XdYfM1psPNLmf7EeX9

Signature: 2NJ1m7qCraPSBAVxbr2ssmWZmBU9Jc8pDtJAnyZsZJRcaYCYMq
q1oRY1gqA4ddQno3g3xcnny5fzr1dvsnFKMEqG
```

要将代币铸造到不同的代币账户，请指定预期接收者代币账户的地址。例如，运行以下命令：

`spl-token mint 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg 100 --Hmyk3FSw4cfsuAes7sanp2oxSkE9ivaH6pMzDzbacqmt`

返回以下输出：

- `99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg`是铸造代币的铸币账户的地址（增加总供应量）。
- `Hmyk3FSw4cfsuAes7sanp2oxSkE9ivaH6pMzDzbacqmt`是代币账户的地址，代币单位正在被铸造到该账户（数量增加）。

终端

```
Minting 100 tokens
  Token: 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg
  Recipient: Hmyk3FSw4cfsuAes7sanp2oxSkE9ivaH6pMzDzbacqmt

Signature: 3SQvNM3o9DsTiLwcEkSPT1Edr14RgE2wC54TEjonEP2swyVCp2
jPWYWdD6RwXUGpvDNUkKWzVBZVFShn5yntxVd7
```

在底层，创建新的代币单位需要调用代币程序上的指令。该指令必须由铸币机构签署。该指令将新的代币单位铸币到代币账户，并增加铸币账户的总供应量。以下是Solana

Playground `MintTo` 上的 Javascript 示例 。

转移代币

要在两个代币账户之间转移代币单位，请使用以下命令：

```
spl-token transfer [OPTIONS] <TOKEN_ADDRESS> <TOKEN_AMOUNT> <REC
```

例如运行以下命令：

```
spl-token transfer 99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg
```

返回以下输出：

- `AfB7uwBEsGtrrBqPTVqEgzWed5XdYfM1psPNLmf7EeX9`是代币转出的代币账户的地址。这将是您用于转移指定代币的代币账户的地址。
- `Hmyk3FSw4cfsuAes7sanp2oxSkE9ivaH6pMzDzbacqmt`是代币将被转移到的代币账户的地址。

终端

```
Transfer 100 tokens
  Sender: AfB7uwBEsGtrrBqPTVqEgzWed5XdYfM1psPNLmf7EeX9
  Recipient: Hmyk3FSw4cfsuAes7sanp2oxSkE9ivaH6pMzDzbacqmt

Signature: 5y6HVwV8V2hHGLTVmTmdySRiEUCZnWmkasAvJ7J6m7JR46obbG
KCBqUFgLpZu5zQGwM4Xy6GZ4M5LKd1h6Padx3o
```

在底层，转移代币需要调用代币程序上的指令。此指令必须由发送者的代币账户所有者签名。该指令将代币单位从一个代币账户转移到另一个代币账户。这是Solana Playground `Transfer`上的 Javascript 示例 。

重要的是要了解，发送者和接收者都必须拥有针对要转移的特定类型代币的现有代币账户。发送者可以在交易中包含其他指令，以创建接收者的代币账户，该账户通常是关联代币账户。

创建代币元数据

代币扩展程序允许将额外的可定制元数据（如名称、符号、图像链接）直接存储在 Mint 帐户中。

信息

要使用 Token Extensions CLI 标志，请确保您已本地安装 CLI，版本 3.4.0 或更高版本：

```
cargo install --version 3.4.0 spl-token-cli
```

要创建启用元数据扩展的新令牌，请使用以下命令：

```
spl-token create-token --program-id TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb--enable-metadata
```

该命令返回以下输出：

- `BdhzpzhTD1MFqBiwNdrRy4jFo2FHFufw3n9e8sVjJczP`

  是启用元数据扩展后创建的新代币的地址。

终端

```
Creating token BdhzpzhTD1MFqBiwNdrRy4jFo2FHFufw3n9e8sVjJczP u
nder program TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb
To initialize metadata inside the mint, please run `spl-token
initialize-metadata BdhzpzhTD1MFqBiwNdrRy4jFo2FHFufw3n9e8sVjJ
czP <YOUR_TOKEN_NAME> <YOUR_TOKEN_SYMBOL> <YOUR_TOKEN_URI>`,
and sign with the mint authority.

Address:  BdhzpzhTD1MFqBiwNdrRy4jFo2FHFufw3n9e8sVjJczP
Decimals:  9

Signature: 5iQofFeXdYhMi9uTzZghcq8stAaa6CY6saUwcdnELST13eNSif
iuLbvR5DnRt311frkCTUh5oecj8YEvZSB3wfai
```

一旦创建了启用元数据扩展的新令牌，请使用以下命令初始化元数据。

```
spl-token initialize-metadata <TOKEN_MINT_ADDRESS> <YOUR_TOKEN_N
```

代币 URI 通常是指向您想要与代币关联的链下元数据的链接。您可以 在此处找到 JSON 格式的示例。

例如，运行以下命令将把附加元数据直接存储在指定的 mint 帐户上：

```
spl-token initialize-metadata BdhzpzhTD1MFqBiwNdrRy4jFo2FHFufw3r
```

然后，您可以在浏览器上查找铸币账户的地址以检查元数据。例如，这是在 SolanaFm 浏览器上启用元数据扩展后创建的代币。

您可以在 元数据扩展指南中了解更多信息。 有关各种 Token 扩展的更多详细信息，请参阅 Token 扩展 入门指南 和SPL 文档。