

solanaNotes

账户

账户有几种：

- 1. program： 存储可执行代码， info无状态
- 2. data： 由program创建， 用英语存储和管理程序状态
- 3. native program： solana内置的程序
- 4. Sysvar： 存储network cluster state的特殊账户

每个账户有一个唯一的地址， 以Ed25519格式表示为32字节

1.accountInfo

账户最大为10mb， 存储的数据结构称为accountInfo

account	data	executable	lamports	owner
type	Bytes	Boolean	Number	Program Address
usage	存储账户状态	标志着该账户是否是程序	账户余额的数字形式	拥有该账户的程序的公钥

As a key part of the Solana Account Model, every account on Solana has a designated "owner", specifically a program. Only the program designated as the owner of an account can modify the data stored on the account or deduct the lamport balance. It's important to note that while only the owner may deduct the balance, anyone can increase the balance.

作为 Solana 账户模型的关键部分，Solana 上的每个账户都有一个指定的“所有者”，特别是一个程序。只有指定为帐户所有者的程序才能修改帐户中存储的数据或扣除 lamport 余额。需要注意的是，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 信息

To store data on-chain, a certain amount of SOL must be transferred to an account. The amount transferred is proportional to the size of the data stored on the account. This concept is commonly referred to as "rent". However, you can think of "rent" more like a "deposit" because the SOL allocated to an account can be fully recovered when the account is closed.

为了将数据存储链上，必须将一定数量的SOL转入账户。传输的金额与帐户中存储的数据大小成正比。这个概念通常被称为“租金”。但是，您可以将“租金”视为“押金”，因为分配给帐户的 SOL 在帐户关闭时可以完全收回。

2.native program

常见的有the System Program和BPF loader

3.System program

所有新建账户默认归system program所有

系统程序执行以下关键任务：

1. New Account Creation
2. Space Allocation
3. Assign Program Ownership

钱包只是系统程序拥有的账户。只有系统程序拥有的账户才可以作为交易费用的支付者

4.BPFLoader

BPF 加载程序是指定为网络上所有其他程序（不包括本机程序）的“所有者”的程序。它负责部署、升级和执行自定义程序。

5.Sysvar Accounts

Sysvar 帐户是位于预定义地址的特殊帐户，可提供对集群状态数据的访问。这些帐户会使用有关网络集群的数据动态更新。

6.Custom Programs

智能合约被称为程序Program，指包含可执行代码的账户，且拥有executable=true

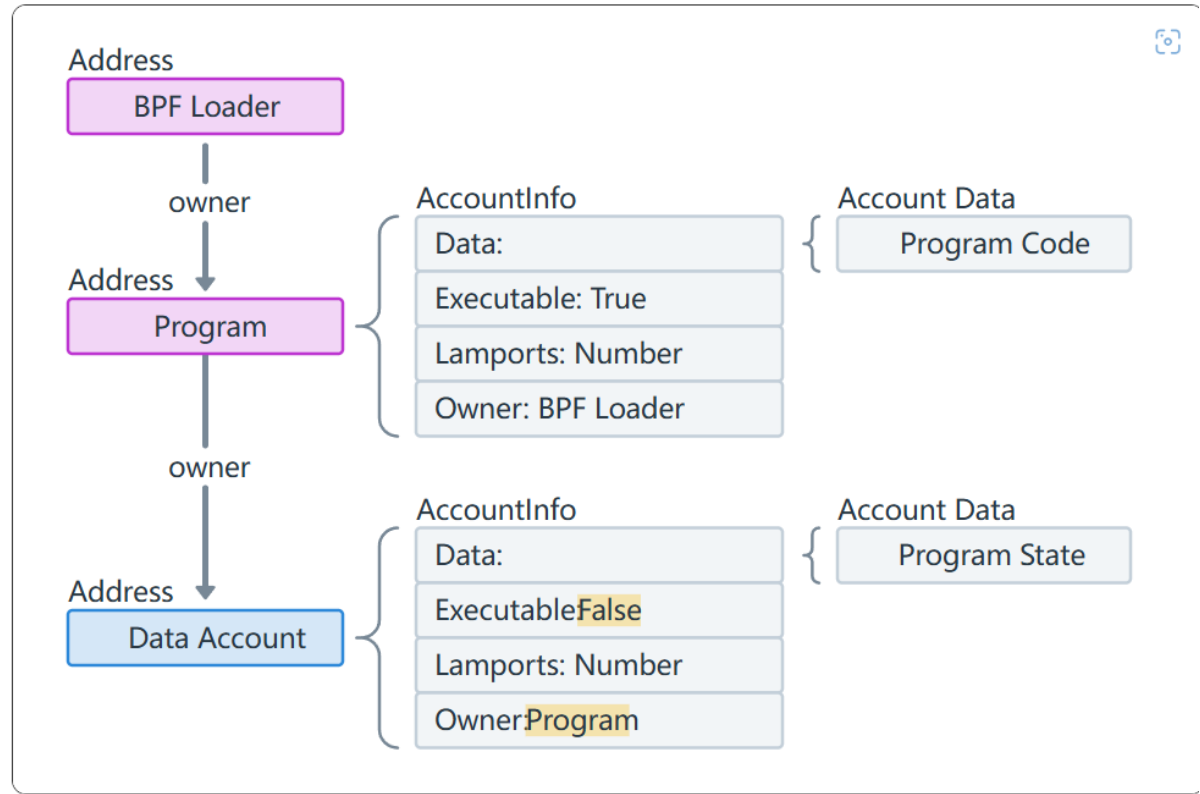
创建program account时会创建三个独立账户：

- 1. program account：代表链上程序的主账户。该帐户存储可执行数据帐户（2）的地址（存储已编译的程序代码）和程序的更新权限（被授权对程序进行更改的地址）
- 2. program executable data account：包含程序的可执行二进制代码的账户
- 3. buffer account：在主动部署或升级程序时存储字节代码的临时帐户。该过程完成后，数据将传输到程序可执行数据帐户（2），并且缓冲区帐户将关闭。

可近似将program account当作程序自身

7.data account

因为程序账户无法靠自身存储和修改数据，所以它会创建新账户存储修改数据，这些账户称为data account



创建data account有两个步骤：

- 1. 调用系统程序创建账户，并转移所有权给程序账户
- 2. 调用已拥有新账户的程序账户，按照程序代码中的定义初始化账户数据

以下是一个程序部署的示例：

Overview 概述	<div>InspectRefresh</div>
Signature 签名	3h3t3gDA2Hu4XqZk2pM4T4meg3Z8zxmenpkeSBiNgvuoXHT6h5xkEhP8m4d47gfxVtZU5u6B8s8tb8x6bB6d9qUIs
Result 结果	Success 成功
Timestamp 时间戳	Aug 2, 2024 at 20:20:43 GMT+08:00 2024 年 8 月 2 日 20:20:43 GMT+08:00
Confirmation Status 确认状态	FINALIZED 最终确定的
Confirmations 确认信息	MAX 最大限度
Slot 代币口	315,976,988
Recent Blockhash 最近的区块哈希	8GbAcX2XJPEjaxEyqKNWRHSsKxYzzW7PfevcSmpqSmnu
Fee (SOL) 费用 (SOL)	0.00001 0.00001
Compute units consumed 消耗的计算单元	2,670
Transaction Version 交易版本	LEGACY 遗产

和正常交易类似的info

Account Input(s) 账户输入				
#	ADDRESS 地址	CHANGE (SOL) 改变 (SOL)	POST BALANCE (SOL) 后期余额 (SOL)	DETAILS 细节
1	89t8iWq7eNwU1X4g7bgSxKUFRQtLZhiRofafZze21pRr	<div>-1.398428161.39842816</div>	<div>7.192420927.19242092</div>	<div>Fee PayerSignerWritable 费用支付者签名者可写</div>
2	3YUZzUEG11ea2qJdPTq2ho6G9dc9A7jh1aEHnYrMy4P4	<div>+0.001398960.00139896</div>	<div>0.001398960.00139896</div>	<div>SignerWritable 签名者可写</div>
3	28uGHiXjm4oZKC6yMAUEkZvQJaqmGUcknEUrPrmp6qNb	<div>+2.795115122.79511512</div>	<div>2.795115122.79511512</div>	<div>Writable 可写</div>
4	BRshBofk8wxPawJ4xsFPZ9VndVjqP24wJUHsqR1VksoR	<div>-1.398103921.39810392</div>	<div>000</div>	<div>Writable 可写</div>
5	System Program 系统程序	<div>0</div>	<div>0.0000000010.000000001</div>	<div>Program 程序</div>
6	BPF Upgradeable Loader BPF 可升级加载器	<div>0</div>	<div>0.0000000010.000000001</div>	<div>Program 程序</div>
7	Sysvar: Clock 系统变量: 时钟	<div>0</div>	<div>0.001169280.00116928</div>	
8	Sysvar: Rent 西斯瓦尔: 租金	<div>0</div>	<div>0.00100920.0010092</div>	

play ground默认的账户输入

#1 System Program: Create Account 系统程序：创建帐户<> Raw

Program 程序System Program 系统程序

From Address 发件人地址89t8iWq7eNwU1X4g7bgsxKUFRQtLZhiRofafZze21pRr

New Address 新地址3YUZzUEG11ea2qJdPTq2ho6G9dc9A7jhiaEHnYrMy4P4

Transfer Amount (SOL) 转账金额 (SOL)0.001398960.00139896

Allocated Data Size 分配的数据大小36 byte(s) 36 字节

Assigned Program Id 分配的程序 IDBPF Upgradeable Loader BPF 可升级加载器

#2 BPF Upgradeable Loader: Deploy With Max Data Len BPF 可升级加载器：使用 Max Data Len 进行部署<> Raw

Program 程序BPF Upgradeable Loader BPF 可升级加载器

Authority 权威89t8iWq7eNwU1X4g7bgsxKUFRQtLZhiRofafZze21pRr

Buffer Account 缓冲帐户BRshBofk8wxPawJ4xsFPZ9VNdVjqP24wJUH5qR1Vksor

Clock Sysvar 时钟系统变量Sysvar: Clock 系统变量：时钟

Max Data Len 最大数据长度401424

Payer Account 付款人账户89t8iWq7eNwU1X4g7bgsxKUFRQtLZhiRofafZze21pRr

Program Account 计划帐户3YUZzUEG11ea2qJdPTq2ho6G9dc9A7jhiaEHnYrMy4P4

Program Data Account 程序数据帐户2BuGMiXjm4oZKC6yMAUEkZvQJaqmGUcknEUrPrmp6qNb

Rent Sysvar 出租西斯瓦尔Sysvar: Rent 西斯瓦尔：租金

System Program 系统程序System Program 系统程序

INNER INSTRUCTIONS 内部说明

#2.1 System Program: Create Account 系统程序：创建帐户<> Raw

一点部署时的流程detail

#2.1 System Program: Create Account 系统程序: 创建帐户 <> Raw

Program 程序	System Program 系统程序
From Address 发件人地址	89t8iWq7eNwU1X4g7bgsxKUFRQtLZhiRofafZze21pRr
New Address 新地址	2BuGMiXjm4oZKC6yMAUEkZvQJaqmGUcknEUrPrmp6qNb
Transfer Amount (SOL) 转账金额 (SOL)	2.79511512
Allocated Data Size 分配的数据大小	401469 byte(s) 401469 字节
Assigned Program Id 分配的程序 ID	BPF Upgradeable Loader BPF 可升级加载器

Program Instruction Logs 程序指令日志 <> Raw

System Program Instruction

#1 系统程序说明

> Program returned success

> 程序返回成功

BPF Upgradeable Loader Instruction

#2 BPF 可升级加载器指令

> Program invoked: System Program

> 调用的程序: 系统程序

> Program returned success

> 程序返回成功

> Deployed program 3YUZzUEG11ea2qJdPTq2ho6G9dc9A7jhiaEHnYrMy4P4

> 已部署程序 3YUZzUEG11ea2qJdPTq2ho6G9dc9A7jhiaEHnYrMy4P4

> Program returned success

> 程序返回成功

挺小一示例程序花了我挺多token，但多数都是后面能退回给我的rent。花的最多的是Program Executable Data Account，存储这个地址的program account花的反而很少。这两个都有一个相同的Upgrade Authority，指向fee payer和signer也就是我的账户。

在尝试close了program account后，两个account剩下的token都退回到我原有账户。此时的token差值才是实际上的损耗

来点anchor

以下是一个anchor框架的案例程序的实践，目的在于使用程序派生地址PDA作为账户地址存储用户信息。这里也有playground的同款链接<https://beta.solpg.io/66734b7bcffcf4b13384d1ad>

0.PDA?

PDA 是使用用户定义的种子、bump种子和程序 ID 的组合确定性导出的地址。导出地址对应的账户存在与否，由之前是否显式地创建决定。在没有创建时访问PDA所在的账户是无效的。PDA可以被solana程序通过ID签名，从而不需要私钥。当调用了api产生的第一个有效的bump种子称为“canonical bump”（规范凹凸），而官方文档建议在使用PDA时使用规范bump，否则可能访问到空账户或者不属于运行程序的无效账户

PDA的bump值范围为0~255，每次调用api时的bump值创建账户时，给定相同的可选种子和programId，具有不同值的bump种子仍然可以派生出有效的PDA（可能无效而且不推荐使用）。在anchor框架中，bump值可以由anchor自己推导出有效值，不用自己指定，从而避免了会遇到的问题。

1.Starter Code

项目结构如下

src/lib.rs

tests/anchor.test.ts

```
// src/lib.rs
use anchor_lang::prelude::*;

declare_id!("8KPzbM2Cwn4Yjak7QYAEH9wyoQh86NcBicaLuzPaejdW");

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

    pub fn create(_ctx: Context<Create>) -> Result<> {
        ok(())
    }

    pub fn update(_ctx: Context<Update>) -> Result<> {
        ok(())
    }

    pub fn delete(_ctx: Context<Delete>) -> Result<> {
        ok(())
    }
}

#[derive(Accounts)]
#[instruction(message: String)]
pub struct Create<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(
        init,
        seeds = [b"message".user.key().as_ref()],
        bump,
        payer = user,
        space = 8 + 32 + 4 + message.len() + 1
    )]
    pub message_account: Account<'info, MessageAccount>,
    pub system_program: Program<'info, System>,
};

#[derive(Accounts)]
pub struct Update {}

#[derive(Accounts)]
pub struct Delete {}

#[account]
pub struct MessageAccount {
    pub user: Pubkey,
```

```
pub message: String,
pub bump: u8,
}
```

这段代码定义了一个message account和创建此message account需要的数据结构

grammar explanation:

1. `#[derive(Accounts)]` 用于注释表示指令所需账户列表的结构，这个结构的每一个字段都是一个账户。
2. 结构中的每个帐户（字段）都用帐户类型进行注释（例如 `signer<'info>`），并且可以使用约束进一步注释（例如 `#[account(mut)]`）。账户类型和账户限制用于对传递给指令的账户执行安全检查。
3. 每个字段命名对账户验证没有影响，但是推荐描述性账户名称

对Create struct功能的描述:

1. 定义了create指令所需的账户，比如 `user: signer<'info>` 代表创建message account,并通过 `#[account(mut)]` 标记为可变的,因为它为新账户付费,而且只能是签名者批准交易,因为lamport将从账户扣除
2. `message_account: Account<'info, MessageAccount>` 这个创建的新账户用于存储用户的信息,init约束表示将在指令中创建账户,seeds和bump表明账户的地址是PDA(program derived Address),payer=user指定"给创建新用户"付费的账户,space指定分配给新账户数据字段的字节数
3. `system_program: Program<'info, System>` 这个是创建新账户时需要调用的系统账户,init调用系统程序来创建分配有指定space的新账户,并将所有权重新分配给当前程序
4. `#[instruction(message: String)]` 使Create能够从create指令访问message参数.
5. seeds和bump一起使用来指定账户的地址是PDA.seeds的两个参数: `b"message"` 是第一个种子的硬编码字符串, `user.key().as_ref()` 引用user的公钥作为第二个种子, `bump` 告诉anchor自动查找并使用正确的bump种子.最终anchor将使用seeds和bump导出PDA
6. space 的各参数:
 1. Anchor Account discriminator (identifier): 8 bytes
anchor账户鉴别符（标识符）：8字节
 2. User Address (Pubkey): 32 bytes
用户地址（公钥）：32字节
 3. User Message (String): 4 bytes for length + variable message length
用户消息（字符串）：4字节长度+可变消息长度
 4. PDA Bump seed (u8): 1 byte
PDA bump种子 (u8): 1 字节

再对对应的create函数修改:


```
pub fn create(ctx: Context<Create>, message: String) -> Result<()> {
    msg!("Create Message: {}", message);
    let account_data = &mut ctx.accounts.message_account;
    account_data.user = ctx.accounts.user.key();
    account_data.message = message;
    account_data.bump = ctx.bumps.message_account;
    Ok(())
}
```

功能描述:create通过 `ctx: Context<Create>` 获得对 `Create` 结构中指定账户的访问,通过 `message` 获得需要存储的用户信息

函数逻辑:

1. 用 `msg!()` 宏打印消息到logs
2. 初始化账户数据 `message`, `user` 为用户账户的地址, `message` 为消息, `bump` 用于后续派生PDA

类似的可仿照实现 `update` 结构:

```
#[derive(Accounts)]
#[instruction(message: String)]
pub struct Update<'info> {
    #[account(mut)]
    pub user: Signer<'info>,

    #[account(
        mut,
        seeds = [b"message", user.key().as_ref()],
        bump = message_account.bump,
        realloc = 8 + 32 + 4 + message.len() + 1,
        realloc::payer = user,
        realloc::zero = true,
    )]
    pub message_account: Account<'info, MessageAccount>,
    pub system_program: Program<'info, System>,
}
```

和 `create` 有所区别的在于,使用了 `realloc` 调整账户的数据大小,比如 `realloc::payer` 可能与存储在 `message_account` 上的不同,而 `bump` 与存储在 `message_account` 上的相同.相同的 `seeds` 和 `bump` 保证了 PDA 的不变

接下来实现 `update` 函数:

```
pub fn update(ctx: Context<Update>, message: String) -> Result<()> {
    msg!("Update Message: {}", message);
    let account_data = &mut ctx.accounts.message_account;
    account_data.message = message;
    Ok(())
}
```

再来实现 `Delete` 结构

```
#[derive(Accounts)]
pub struct Delete<'info> {
    #[account(mut)]
    pub user: Signer<'info>,

    #[account(
        mut,
        seeds = [b"message", user.key().as_ref()],
        bump = message_account.bump,
        close= user,
    )]
    pub message_account: Account<'info, MessageAccount>,
}
```

其中 `close= user` 指定账户将被关闭,其lamports将转移给user,对应的user标记为mut.奇怪的是此时没有系统程序的参与

最后实现delete函数:

```
pub fn delete(_ctx: Context<Delete>) -> Result<()> {
    msg!("Delete Message");
    ok(())
}
```

功能被结构 `Delete` 之中的close实现,这个函数无事可干

此时如果你也是用Playground的在线ide,就可以直接 `build && deploy` 了

2.test

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

describe("pda", () => {
    > to be added firstly
    it("Create Message Account", async () => {});

    it("Update Message Account", async () => {});

    it("Delete Message Account", async () => {});
});
```

这是test最开始的样子,向第一个it前添加以下code

```
const program = pg.program;
const wallet = pg.wallet;

const [messagePda, messageBump] = PublicKey.findProgramAddressSync(
    [Buffer.from("message"), wallet.publicKey.toBuffer()],
    program.programId,
);
```

program行允许访问客户端库,wallet行是我的plaground 钱包.其余的演示如何用程序中指定的种子导出pda

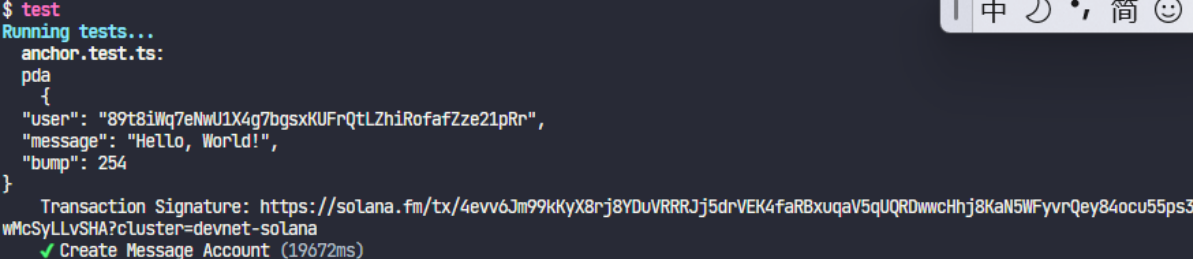
接下来写create的test部分:

```
it("Create Message Account", async () => {
  const message = "Hello, World!";
  const transactionSignature = await program.methods
    .create(message)
    .accounts({
      messageAccount: messagePda,
    })
    .rpc({ commitment: "confirmed" });

  const messageAccount = await program.account.messageAccount.fetch(
    messagePda,
    "confirmed",
  );

  console.log(JSON.stringify(messageAccount, null, 2));
  console.log(
    "Transaction Signature:",
    `https://solana.fm/tx/${transactionSignature}?cluster=devnet-solana`,
  );
});
```

签名,获取messageAccount.输出日志



A terminal window showing the execution of a test. The output includes the test name 'Create Message Account', the message 'Hello, World!', the bump value 254, and the transaction signature. The transaction signature is a long URL starting with 'https://solana.fm/tx/'. The test passed successfully, indicated by a green checkmark and the message 'Create Message Account (19672ms)'.

```
$ test
Running tests...
anchor.test.ts:
pda
{
  "user": "89t8iWq7eNwU1X4g7bgsxKUfrQtLZhiRofafZze21pRr",
  "message": "Hello, World!",
  "bump": 254
}
Transaction Signature: https://solana.fm/tx/4evv6Jm99kKyX8rj8YDuVRRRj5drVEK4faRBxuqaV5qUQRDwmcHhj8KaN5WFyvrQey84ocu55ps3wMcSyllvSHA?cluster=devnet-solana
✓ Create Message Account (19672ms)
```

继续update部分:

```
it("Update Message Account", async () => {
  const message = "Hello, Solana!";
  const transactionSignature = await program.methods
    .update(message)
    .accounts({
      messageAccount: messagePda,
    })
    .rpc({ commitment: "confirmed" });

  const messageAccount = await program.account.messageAccount.fetch(
    messagePda,
    "confirmed",
  );

  console.log(JSON.stringify(messageAccount, null, 2));
  console.log(
    "Transaction Signature:",
    `https://solana.fm/tx/${transactionSignature}?cluster=devnet-solana`,
  );
});
```

还有delete:

```
it("Delete Message Account", async () => {
  const transactionSignature = await program.methods
    .delete()
    .accounts({
      messageAccount: messagePda,
    })
    .rpc({ commitment: "confirmed" });

  const messageAccount = await program.account.messageAccount.fetchNullable(
    messagePda,
    "confirmed",
  );

  console.log("Expect Null:", JSON.stringify(messageAccount, null, 2));
  console.log(
    "Transaction Signature:",
    `https://solana.fm/tx/${transactionSignature}?cluster=devnet-solana`,
  );
});
```

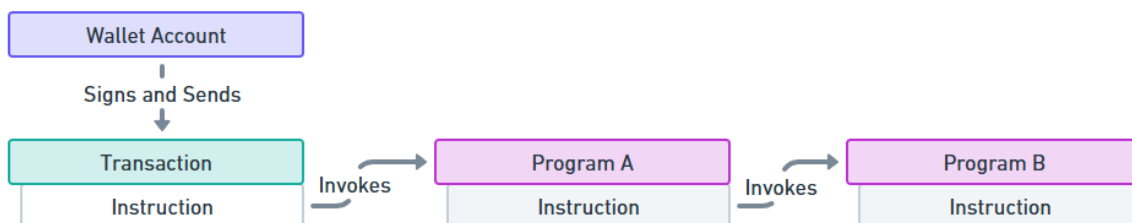
写完了就test看一眼吧

```
✓ Update Message Account (1483ms)
Expect Null: null
Transaction Signature: https://solana.fm/tx/3vHQWMAPCBHiGY75u1PQsQjCiTvUvisTRwyQeUvFoMvK1gE2Ha9Z?cluster=devnet-solana
✓ Delete Message Account (1375ms)
3 passing (23s)
$ tree
Process error: Command `tree` not found
```

3.Cross Program Invocation(CPI)

前两节的程序将在这里开始得到修改,用CPI实现从这个程序与其他程序的交互。

CPI: 一个程序调用另一个程序的指令, 另一个程序的指令类似于向互联网公开的API。



发起CPI时, 在A上的签名者权限会扩展到被调用的B上。这样的迭代调用深度最高为4, 程序可以代表从其 ID 派生的PDA进行“签名”。

在solana_program crate中的 `invoke` 和 `invoke_signed` 各自对应没有PDA签名和有PDA签名的调用。

用CPI实现transfer的案例:

```
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 = ctx.accounts.sender.to_account_info();
        let to_pubkey = ctx.accounts.recipient.to_account_info();
        let program_id = ctx.accounts.system_program.to_account_info();

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

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

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

可以看到，invoke时需要传入的参数有instruction和一个切片

式。在底层，每个 CPI 指令必须指定以下信息：

- **Program address:** Specifies the program being invoked
程序地址：指定被调用的程序
- **Accounts:** Lists every account the instruction reads from or writes to, including other programs
帐户：列出指令读取或写入的每个帐户，包括其他程序
- **Instruction Data:** Specifies which instruction on the program to invoke, plus any additional data required by the instruction (function arguments)
指令数据：指定要调用程序上的哪条指令，以及该指令所需的任何其他数据（函数参数）

实际上调用的是system的transfer，transfer需要from和to两个公钥，所以传的切片里有这三个地址

再看一个 `invoke_signed` 的例子：

```
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 = ctx.accounts.pda_account.to_account_info();
        let to_pubkey = ctx.accounts.recipient.to_account_info();
        let program_id = ctx.accounts.system_program.to_account_info();

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

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

        invoke_signed(
            instruction,
            [&from_pubkey, to_pubkey, program_id],
            signer_seeds,
        )?;
        ok(())
    }
}

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

```

和前一个对比:

1. sender不同, 这个是从pda_account开始send sol
2. recipient对 `invoke_signed` 提供seed, pda提供bump seed并一起签名

4.CPI in anchor

继续更改之前的程序, 从Update开始: (这里的修改和Delete的struct相同)

```

use anchor_lang::system_program::{transfer, Transfer};
//将这行添加到开头

```

```
//下面对struct Update修改
#[derive(Accounts)]
#[instruction(message: String)]
pub struct Update<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(
        mut,
        seeds = [b"vault", user.key().as_ref()],
        bump,
    )]
    pub vault_account: SystemAccount<'info>, //新增
    #[account(
        mut,
        seeds = [b"message", user.key().as_ref()],
        bump = message_account.bump,
        realloc = 8 + 32 + 4 + message.len() + 1,
        realloc::payer = user,
        realloc::zero = true,
    )]
    pub message_account: Account<'info, MessageAccount>,
    pub system_program: Program<'info, System>,
}
```

新增的vault将在用户修改message时接受sol并存储这些token

在update的函数中补全这一功能的实现：

```
pub fn update(ctx: Context<Update>, message: String) -> Result<()> {
    msg!("Update Message: {}", message);
    let account_data = &mut ctx.accounts.message_account;
    account_data.message = message;

+   let transfer_accounts = Transfer {
+       from: ctx.accounts.user.to_account_info(),
+       to: ctx.accounts.vault_account.to_account_info(),
+   };
+   let cpi_context = CpiContext::new(
+       ctx.accounts.system_program.to_account_info(),
+       transfer_accounts,
+   );
+   transfer(cpi_context, 1_000_000)?;
    ok(())
}
```

可以看到anchor封装了cpi，简洁地实现了transfer

在delete函数里也加上sol的处理

```
+ pub fn delete(ctx: Context<Delete>) -> Result<()> {
+     msg!("Delete Message");

+     let user_key = ctx.accounts.user.key();
+     let signer_seeds: [&[u8]] =
+         [&[b"vault", user_key.as_ref(), &[ctx.bumps.vault_account]]];
```

```

+
+     let transfer_accounts = Transfer {
+         from: ctx.accounts.vault_account.to_account_info(),
+         to: ctx.accounts.user.to_account_info(),
+     };
+     let cpi_context = CpiContext::new(
+         ctx.accounts.system_program.to_account_info(),
+         transfer_accounts,
+     ).with_signer(signer_seeds);
+     transfer(cpi_context, ctx.accounts.vault_account.lamports())?;
+     ok()
+ }

```

这里的cpi加上了 `with_signer`，也是实现了功能

再rebuild程序并deploy。和ethereum不一样的是，新程序可以直接在旧程序的地址上部署，共享相同的id

新的test如下：（不熟ts，直接给）

```

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

describe("pda", () => {

    const program = pg.program;
    const wallet = pg.wallet;

    const [messagePda, messageBump] = PublicKey.findProgramAddressSync(
        [Buffer.from("message"), wallet.publicKey.toBuffer()],
        program.programId,
    );
    const [vaultPda, vaultBump] = PublicKey.findProgramAddressSync(
        [Buffer.from("vault"), wallet.publicKey.toBuffer()],
        program.programId,
    );
    it("Create Message Account", async () => {
        const message = "Hello, world!";
        const transactionSignature = await program.methods
            .create(message)
            .accounts({
                messageAccount: messagePda,
            })
            .rpc({ commitment: "confirmed" });

        const messageAccount = await program.account.messageAccount.fetch(
            messagePda,
            "confirmed",
        );

        console.log(JSON.stringify(messageAccount, null, 2));
        console.log(
            "Transaction Signature:",
            `https://solana.fm/tx/${transactionSignature}?cluster=devnet-solana`,
        );
    });
});

```



```

it("Update Message Account", async () => {
  const message = "Hello, Solana!";
  const transactionSignature = await program.methods
    .update(message)
    .accounts({
      messageAccount: messagePda,
      vaultAccount: vaultPda
    })
    .rpc({ commitment: "confirmed" });

  const messageAccount = await program.account.messageAccount.fetch(
    messagePda,
    "confirmed",
  );

  console.log(JSON.stringify(messageAccount, null, 2));
  console.log(
    "Transaction Signature:",
    `https://solana.fm/tx/${transactionSignature}?cluster=devnet-solana`,
  );
});

it("Delete Message Account", async () => {
  const transactionSignature = await program.methods
    .delete()
    .accounts({
      messageAccount: messagePda,
      vaultAccount: vaultPda
    })
    .rpc({ commitment: "confirmed" });

  const messageAccount = await program.account.messageAccount.fetchNullable(
    messagePda,
    "confirmed",
  );

  console.log("Expect Null:", JSON.stringify(messageAccount, null, 2));
  console.log(
    "Transaction Signature:",
    `https://solana.fm/tx/${transactionSignature}?cluster=devnet-solana`,
  );
});
});

```

就是不知道为啥我只有10sol，在前一个程序部署后剩下6.any，test并再次部署+test后我的sol增加到了11.6…有空一定得溯源看看怎么多出来的

