Solana 教程

1. 资料

- 教程
  - https://www.soldev.app/course
  - https://www.solanazh.com/
  - https://decert.me/tutorial/sol-dev/
- anchor：https://www.anchor-lang.com
- 论坛：https://soldev.cn
- Spl token docs：https://spl.solana.com/
- Solana web3.js docs：https://github.com/solana-labs/solana-web3.js.git

2. 基础知识

2.1 核心概念

2.1.1 账户和交易

- 所有信息都保存在 **Account** 里

```rust
pub struct Account {
    /// lamports in the account
    pub lamports: u64, // 余额
    /// data held in this account
    #[serde(with = "serde_bytes")]
    pub data: Vec<u8>, // 存储内容，合约账户是编译后的代码
    /// the program that owns this account. If executable, the program that loads this account.
    pub owner: Pubkey,
    /// this account's data contains a loaded program (and is now read-only)
    pub executable: bool, // 合约账户此项为 true
    /// the epoch at which this account will next owe rent
    pub rent_epoch: Epoch,
}
```

- **Transaction** 包含一个 **Message** 和多个签名 **Signature**，每个 Message 包含多个交易指令 **Instruction**。目前 Transaction 分成两种，Legacy 和 V0 版本，以下是 V0 版本的介绍（V0 比 Legacy 主要多了 ALTs，见 6.1）。

```rust
pub struct Message {
    /// The message header, identifying signed and read-only `account_keys`.
    /// Header values only describe static `account_keys`, they do not describe
    /// any additional account keys loaded via address table lookups.
    pub header: MessageHeader,

    /// List of accounts loaded by this transaction.
    #[serde(with = "short_vec")]
    pub account_keys: Vec<Pubkey>,

    /// The blockhash of a recent block.
    pub recent_blockhash: Hash,

    /// Instructions that invoke a designated program, are executed in sequence,
    /// and committed in one atomic transaction if all succeed.
    ///
    /// # Notes
    ///
    /// Program indexes must index into the list of message `account_keys` because
    /// program id's cannot be dynamically loaded from a lookup table.
    ///
    /// Account indexes must index into the list of addresses
    /// constructed from the concatenation of three key lists:
    ///   1) message `account_keys`
    ///   2) ordered list of keys loaded from `writable` lookup table indexes
    ///   3) ordered list of keys loaded from `readable` lookup table indexes
    #[serde(with = "short_vec")]
    pub instructions: Vec<CompiledInstruction>,

    /// List of address table lookups used to load additional accounts
    /// for this transaction.
    #[serde(with = "short_vec")]
    pub address_table_lookups: Vec<MessageAddressTableLookup>,
}

pub enum VersionedMessage {
    Legacy(LegacyMessage),
    V0(v0::Message),
}

```

```rust
pub struct VersionedTransaction {
    /// List of signatures
    #[serde(with = "short_vec")]
    pub signatures: Vec<Signature>,
    /// Message to sign.
    pub message: VersionedMessage,
}
```

- 交易指令 Instruction

```rust
pub struct CompiledInstruction {
    /// Index into the transaction keys array indicating the program account that executes this instruction.
    pub program_id_index: u8, // 合约地址
    /// Ordered indices into the transaction keys array indicating which accounts to pass to the program.
    #[serde(with = "short_vec")]
    pub accounts: Vec<u8>, // 使用到的 Account
    /// The program input data.
    #[serde(with = "short_vec")]
    pub data: Vec<u8>, // 二进制数据
}
```

- **ATA 账户（非常重要）：PDA 账号的一种，参考 5.6 以及一些其他资料**
- slot：
  - https://solana.stackexchange.com/questions/8846/what-is-the-difference-between-a-solana-slot-and-block
  - https://www.helius.dev/blog/solana-slots-blocks-and-epochs

2.1.2 合约

- 系统合约 Native Program
  - System Program: 创建账号，转账等作用
  - BPF Loader Program: 部署和更新合约
  - Vote program: 创建并管理用户POS代理投票的状态和奖励
- 普通合约 On Chain Program
  - 官方部署的 Token、ATA 合约
  - 用户通过 BPF Loader Program 创建一个普通合约，所以它的 owner 都是 BPF Loader
  - 代币余额合约的 owner 是 Token

2.2 SPL 代币

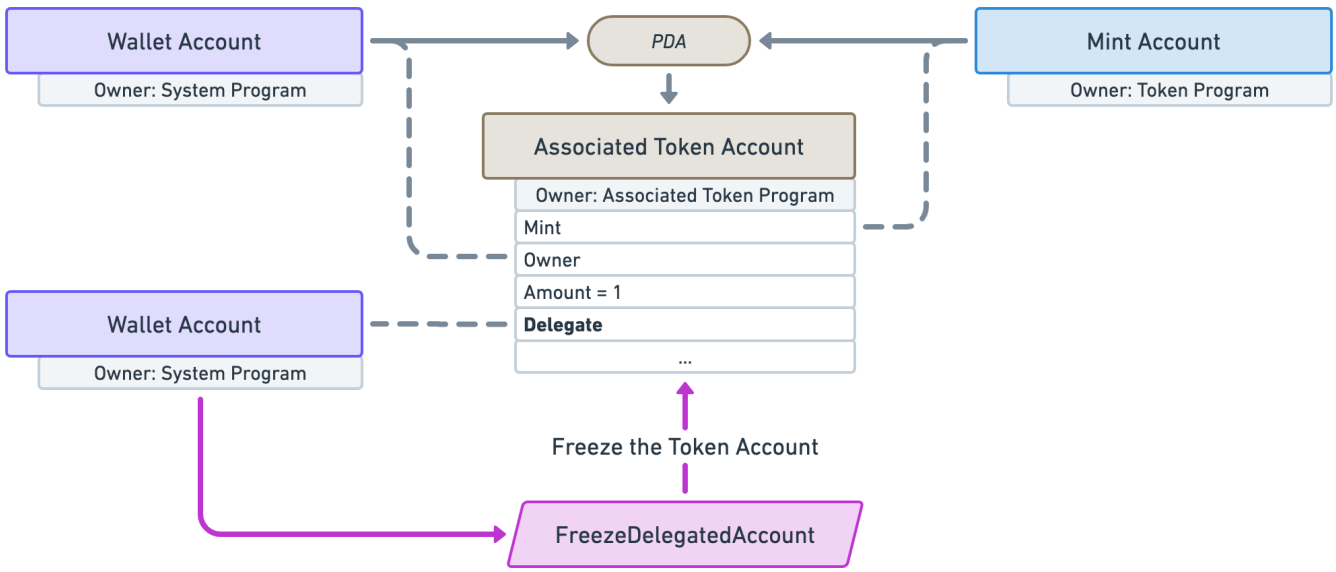
- 一个 SPL 代币仅仅是一个归Token合约管理的普通的Account对象，**代币信息 Mint Account** 结构为：

```rust
pub struct Mint {
    /// Optional authority used to mint new tokens. The mint authority may only be provided during
    /// mint creation. If no mint authority is present then the mint has a fixed supply and no
    /// further tokens may be minted.
    pub mint_authority: COption<Pubkey>,
    /// Total supply of tokens.
    pub supply: u64,
    /// Number of base 10 digits to the right of the decimal place.
    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>,
}
```

- **每个用户拥有的代币信息即 Token Account**

```rust
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, // 数量
    /// If `delegate` is `Some` then `delegated_amount` represents
    /// 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 value logs the rent-exempt reserve. An
    /// Account is required to be rent-exempt, so the value is used by the Processor to ensure that
    /// wrapped SOL 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>,
}
```



- 上图中左上角是代币持有人账户，右上角是代币信息账户，中间是代币持有人持有信息账户，左下角和下方可以不管
- 特别要区分 ATA 的两个 owner：Associated Token Program 是这个 ATA 的创建者，其中的代币的所有者 owner 是某个人的钱包 Wallet Account，前者是 Account 的字段，后者是 Account 的 data 段中的字段，参考 https://soldev.cn/topics/41

2.3 命令行

- https://www.solanazh.com/course/1-4

3. 通过 RPC 与 Solana 交互

- Docs：https://www.quicknode.com/docs/solana
- HTTP 接口

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "getBalance",
    "params": [
        "83astBRguLMdt2h5U1Tpdq5tjFoJ6noeGwaY3mDLVcri"
    ]
}
```

- Websocket 接口

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "accountSubscribe", // 对应都有 unsubscribe 方法
    "params": [
        "CM78CPUeXjn8o3yroDHxUtKsZZgoy4GPkPPXfouKNH12",
        {
        "encoding": "jsonParsed",
        "commitment": "finalized"
        }
    ]
}
```

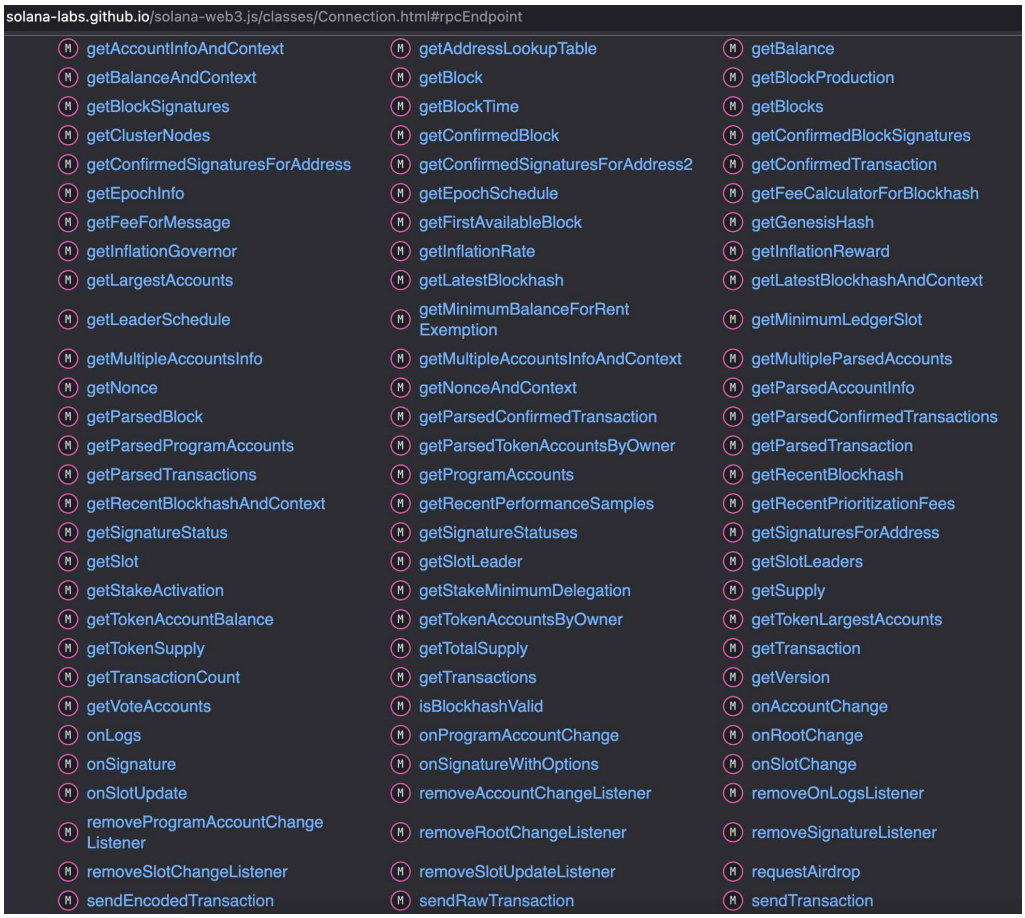
- 'finalized' - 节点将查询由超过集群中超多数确认为达到最大封锁期的最新区块，表示集群已将此区块确认为已完成。
- 'confirmed' - 节点将查询由集群的超多数投票的最新区块。
- 'processed' - 节点将查询最新的区块。注意，该区块可能被集群跳过。
- 一些 API：https://www.solanazh.com/course/2-2

4. 与 Solana 合约交互

4.1 通用交互

- 使用 `@solana/web3.js` 库
- 创建 RPC Connection，创建后就可以使用所有的 RPC 方法

```
let url =  'https://api.devnet.solana.com';
rpcConnection = new Connection(url);
```



- 账号对象是 `Keypair`

```
let secretKey = Uint8Array.from(JSON.parse('[24,xxx,119]'));
const keypair = Keypair.fromSecretKey(secretKey);
console.log("address:", keypair.publicKey.toString())
```

- 获取测试代币

```
// 如果账户小于 0.5 SOL，就空投 1 SOL
await airdropIfRequired(
  connection,
  keypair.publicKey,
  1 * LAMPORTS_PER_SOL,
  0.5 * LAMPORTS_PER_SOL,
);

// 也可以用命令行工具：solana airdrop 1
```

- 发送交易，先写 instruction，然后包装成message、transaction，最后发送

```
const txInstructions = [
```

```
  SystemProgram.transfer({
    fromPubkey: keyPair.current.publicKey, //this.publicKey,
    toPubkey: new PublicKey(toPublicKey), //destination,
    lamports: toCount, //amount,
  }), // 这个是 web3js 自带的 instruction
];

let latestBlockhash = await connection.getLatestBlockhash("finalized");
const messageV0 = new TransactionMessage({
  payerKey: keyPair.current.publicKey,
  recentBlockhash: latestBlockhash.blockhash,
  instructions: txInstructions,
}).compileToV0Message();
const trx = new VersionedTransaction(messageV0);
trx.sign([keyPair.current]);
return await connection.sendTransaction(trx);
```

- 钱包相关：https://www.solanazh.com/course/3-2（这块感觉没太大必要学）
- 调用合约
  - 首先要确认合约函数需要的参数，比如 SPL Token 的合约（即所谓 **Token Program**）需要三个账户，分别是你的 SPL ATA 账户、对方的 SPL ATA 账户和你的 SOL 账户，另外需要 amount 这个参数

> 🎨 这里调用 Token Program 的 Transfer 函数，实际上 web3.js 里已经帮我们包装了这个函数，可以调用 `createMint` 函数 from `@solana/spl-token`

> 🥛 下面用的是 V0 交易，Legacy 交易调用方法可以看 5.2 中的合约调用

```
    /// Transfers tokens from one account to another either directly or via a
    /// delegate.  If this account is associated with the native mint then equal
    /// amounts of SOL and Tokens will be transferred to the destination
    /// account.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner/delegate
    ///   0. `[writable]` The source account.
    ///   1. `[writable]` The destination account.
    ///   2. `[signer]` The source account's owner/delegate.
    ///
    ///   * Multisignature owner/delegate
    ///   0. `[writable]` The source account.
    ///   1. `[writable]` The destination account.
    ///   2. `[]` The source account's multisignature owner/delegate.
    ///   3. ..3+M `[signer]` M signer accounts.
    Transfer {
        /// The amount of tokens to transfer.
        amount: u64,
    },
```

- 我们要先构造 instruction，再是 message，再是 transaction。
  - instruction 的定义如下：

```
/**
* Transaction Instruction class
*/
export class TransactionInstruction {
    /**
    * Public keys to include in this transaction
    * Boolean represents whether this pubkey needs to sign the transaction
    */
    keys: Array<AccountMeta>; // 对应函数需要的账户参数，比如上述就提到了要三个账户信息
    /**
    * Program Id to execute
    */
    programId: PublicKey; // 合约地址
    /**
    * Program input
    */
    data: Buffer; // 核心要构造的
    constructor(opts: TransactionInstructionCtorFields);
}

/**
```

```
22 * Account metadata used to define instructions
23 */
24 type AccountMeta = {
25     /** An account's public key */
26     pubkey: PublicKey; // 公钥
27     /** True if an instruction requires a transaction signature matching
   `pubkey` */
28     isSigner: boolean; // 是否是签名人 == 是否是付钱的人 == 是否是你
29     /** True if the `pubkey` can be loaded as a read-write account. */
30     isWritable: boolean; // 是否能写入，像转账的话，发送方和接收方都要
   writable，具体可以看合约函数的注释里的要求
31 };
```

- Message 构造，新版我们要构造 `MessageV0`，它通过 `TransactionMessage` 类的 `compileToV0Message` 函数来得到，所以我们主要是要得到 `TransactionMessage` 类，然后调用 `compileToV0Message` 函数就可以了。

```
1  export class TransactionMessage {
2      payerKey: PublicKey;
3      instructions: Array<TransactionInstruction>;
4      recentBlockhash: Blockhash;
5      constructor(args: TransactionMessageArgs);
6      static decompile(message: VersionedMessage, args?: DecompileArgs):
   TransactionMessage;
7      compileToLegacyMessage(): Message;
8      compileToV0Message(addressLookupTableAccounts?:
   AddressLookupTableAccount[]): MessageV0;
9  }
10
11 type TransactionMessageArgs = {
12     payerKey: PublicKey;
13     instructions: Array<TransactionInstruction>;
14     recentBlockhash: Blockhash;
15 }; // 这里是核心要传入的三个参数
16
17 /**
18 * Blockhash as Base58 string.
19 */
20 type Blockhash = string;
```

- 最后构造 Transaction 比较容易，直接把 Message 填进 `VersionedTransaction` 就行

```
1  const txInstructions =
```

```
2
3 const message = new TransactionMessage({
4   payerKey: this.keypair.publicKey,
5   recentBlockhash: latestBlockhash.blockhash,
6   instructions: txInstructions
7 }).compileToV0Message();
8
9 const trx = new VersionedTransaction(messageV0);
```

- 所以我们核心要构造 instrcution 的 data（或称 buffer） 段，我们可以先看一下 buffer-layout，这个 layout 定义我们的 data 数据格式，我们需要一个 u8 的 instruction（和 Solidity 的函数选择器一样）然后是函数的参数 u64
  - https://github.com/solana-labs/solana-program-library/blob/master/token/program/src/instruction.rs
  - 函数选择器使用3的原因是：上面代码中可以看到，`Transfer` 在 `enum` 中序号为 3

```
1  // 下面定义 transferInstructionData 定义，使用时不需要写下面这些
2  export interface TransferInstructionData {
3      instruction: TokenInstruction.Transfer;
4      amount: bigint;
5  }
6
7  /** TODO: docs */
8  export const transferInstructionData = struct<TransferInstructionData>
   ([u8('instruction'), u64('amount')]);
9
10 // 实际使用的代码如下：
11 function createTransferInstruction(
12     source,
13     destination,
14     owner,
15     amount,
16     programId
17 ) {
18     const keys = [
19             { pubkey: source, isSigner: false, isWritable: true },
20             { pubkey: destination, isSigner: false, isWritable: true },
21             { pubkey: owner, isSigner:true, isWritable: false}
22     ];
23
24     const data = Buffer.alloc(9);
25     data.writeUInt8(3); // 这个3表示transfer指令
26     const bigAmount = BigInt(amount);
```

```
    data.writeBigInt64LE(bigAmount, 1) // 写入 amount 这个 u64

    return new TransactionInstruction({ keys, programId, data });
}
```

4.2 SPL-Token 库使用

- 参考：https://www.soldev.app/course/token-program
- 创建 Mint

```
const tokenMint = await createMint(
  connection,
  payer,
  mintAuthority,
  freezeAuthority,
  decimal
);

// 其底层逻辑是使用 System Program 创建新账户，然后初始化成一个 Mint，如下：
import * as web3 from '@solana/web3'
import * as token from '@solana/spl-token'

async function buildCreateMintTransaction(
  connection: web3.Connection,
  payer: web3.PublicKey,
  decimals: number
): Promise<web3.Transaction> {
  const lamports = await token.getMinimumBalanceForRentExemptMint(connection);
// 需要支付最低的租金
  const accountKeypair = web3.Keypair.generate();
  const programId = token.TOKEN_PROGRAM_ID

  const transaction = new web3.Transaction().add( // Legacy 格式的交易
    web3.SystemProgram.createAccount({
      fromPubkey: payer,
      newAccountPubkey: accountKeypair.publicKey,
      space: token.MINT_SIZE,
      lamports,
      programId,
    }),
    token.createInitializeMintInstruction(
      accountKeypair.publicKey,
      decimals,
      payer,
      payer,
```

```
      programId
    )
  );

  return transaction
}
```

- 创建 Token Account

```
const tokenAccount = await createAccount(
  connection,
  payer,
  mint,
  owner,
  keypair // 选填，表示 Token Account 的地址，默认就是 ATA
);

// 底层逻辑如下
import * as web3 from '@solana/web3'
import * as token from '@solana/spl-token'

async function buildCreateTokenAccountTransaction(
  connection: web3.Connection,
  payer: web3.PublicKey,
  mint: web3.PublicKey
): Promise<web3.Transaction> {
  const mintState = await token.getMint(connection, mint)
  const accountKeypair = await web3.Keypair.generate()
  const space = token.getAccountLenForMint(mintState);
  const lamports = await connection.getMinimumBalanceForRentExemption(space);
  const programId = token.TOKEN_PROGRAM_ID

  const transaction = new web3.Transaction().add(
    web3.SystemProgram.createAccount({
      fromPubkey: payer,
      newAccountPubkey: accountKeypair.publicKey,
      space,
      lamports,
      programId,
    }),
    token.createInitializeAccountInstruction(
      accountKeypair.publicKey,
      mint,
      payer,
      programId
```

```
    )
  );

  return transaction
}
```

- 创建 ATA

```
const associatedTokenAccount = await createAssociatedTokenAccount(
  connection,
        payer,
        mint,
        owner,
);
// 一般会用另一个函数 getOrCreateAssociatedTokenAccount

// 底层逻辑如下
import * as web3 from '@solana/web3'
import * as token from '@solana/spl-token'

async function buildCreateAssociatedTokenAccountTransaction(
  payer: web3.PublicKey,
  mint: web3.PublicKey
): Promise<web3.Transaction> {
  const associatedTokenAddress = await token.getAssociatedTokenAddress(mint, payer, false);

  const transaction = new web3.Transaction().add(
    token.createAssociatedTokenAccountInstruction(
      payer,
      associatedTokenAddress,
      payer,
      mint
    )
  )

  return transaction
}
```

- Mint Token 给某人的 Token Account。一般 Mint 之后就会将 mint authority 设置成 null，或者也可以将其设置成某个自动程序

```
const transactionSignature = await mintTo(
```

```
  connection,
  payer,
  mint,
  destination,
  authority,
  amount
);

// 底层逻辑
import * as web3 from '@solana/web3'
import * as token from '@solana/spl-token'

async function buildMintToTransaction(
  authority: web3.PublicKey,
  mint: web3.PublicKey,
  amount: number,
  destination: web3.PublicKey
): Promise<web3.Transaction> {
  const transaction = new web3.Transaction().add(
    token.createMintToInstruction(
      mint,
      destination,
      authority,
      amount
    )
  )

  return transaction
}
```

- 转移代币到另一个人的 Token Account

```
const transactionSignature = await transfer(
  connection,
  payer,
  source,
  destination,
  owner,
  amount
)

// 底层逻辑
import * as web3 from '@solana/web3'
import * as token from '@solana/spl-token'

```

```
async function buildTransferTransaction(
  source: web3.PublicKey,
  destination: web3.PublicKey,
  owner: web3.PublicKey,
  amount: number
): Promise<web3.Transaction> {
  const transaction = new web3.Transaction().add(
    token.createTransferInstruction(
      source,
      destination,
      owner,
      amount,
    )
  )

  return transaction
}
```

4.3 Program Log

- 资料：https://read.cryptodatabytes.com/p/solana-analytics-starter-guide-part
- Solana cli 获取交易信息包括 Program Log，可以使用 `solana confirm <TRANSACTION_SIGNATURE>`

4.4 WebSocket

- https://www.quicknode.com/guides/solana-development/getting-started/how-to-create-websocket-subscriptions-to-solana-blockchain-using-typescript#create-an-account-subscription
- `onProgramAccountChange`：监控**属于**某个 Program 的 Account 的账户变化

4.5 HTTP

- `getProgramAccounts`：获取**属于**某个 Program 的 Account 的账户，可以配合 `onProgramAccountChange`

5. Solana Native 合约开发

5.1 Hello world

- 在线开发地址：https://beta.solpg.io/
- Hello world 代码：

```rust
// src/lib.rs
use solana_program::{
    account_info::AccountInfo,
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
    msg,
};

// Declare and export the program's entrypoint
entrypoint!(process_instruction); // 我们的 instruction 默认调用的就是这个

// Program entrypoint's implementation
pub fn process_instruction(
    _program_id: &Pubkey, // Public key of the account the hello world program was loaded into
    _accounts: &[AccountInfo], // The account to say hello to
    _instruction_data: &[u8], // Ignored, all helloworld instructions are hellos
) -> ProgramResult {
    msg!("Hello World Rust program entrypoint"); // msg! 合约执行后会在 log 中打印出来

    Ok(())
}
```

- 用户调用代码：

```ts
// client/client.ts
console.log("My address:", pg.wallet.publicKey.toString());
const balance = await pg.connection.getBalance(pg.wallet.publicKey);
console.log(`My balance: ${balance / web3.LAMPORTS_PER_SOL} SOL`);

// create an empty transaction
const transaction = new web3.Transaction();

// add a hello world program instruction to the transaction
transaction.add(
new web3.TransactionInstruction({
    keys: [],
    programId: new web3.PublicKey(pg.PROGRAM_ID),
}),
); // instruction 默认调用的就是函数的 entry

console.log("Sending transaction...");
```

```
const txHash = await web3.sendAndConfirmTransaction(
    pg.connection,
    transaction,
    [pg.wallet.keypair],
);
console.log("Transaction sent with hash:", txHash);
```

5.2 合约结构

- 一个合约如下所示：
  - 合约的数据存在哪里呢？存在传入的 account 里面
  - 存在哪个 account 里面呢？account.owner == program_id 的那个 account，只有这样，该 program 才有权限写入数据，这意味着合约逻辑和数据存储是分开的，我们必须专门开一个数据存储的 account 提供给program来写入数据

```
use borsh::{BorshDeserialize, BorshSerialize};
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    msg,
    program_error::ProgramError,
    pubkey::Pubkey,
};

/// Define the type of state stored in accounts
#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct GreetingAccount {
    /// number of greetings
    pub counter: u32,
}

// Declare and export the program's entrypoint
entrypoint!(process_instruction);

// Program entrypoint's implementation
pub fn process_instruction(
    program_id: &Pubkey, // Public key of the account the hello world program was loaded into
    accounts: &[AccountInfo], // The account to say hello to, 即 instruction 的 keys 字段
    _instruction_data: &[u8], // Ignored, all helloworld instructions are hellos
) -> ProgramResult {
```

```rust
    msg!("Hello World Rust program entrypoint");

    // Iterating accounts is safer than indexing
    let accounts_iter = &mut accounts.iter();

    // Get the account to say hello to
    let account = next_account_info(accounts_iter)?; // 获取账户用迭代器好一些

    // The account must be owned by the program in order to modify its data
    if account.owner != program_id {
        msg!("Greeted account does not have the correct program id");
        return Err(ProgramError::IncorrectProgramId);
    }

    // Increment and store the number of times the account has been greeted
    // 我们的目的是：将用户传入的字节数据反序列化、修改，最后再保存回去
    // 这里的 try_from_slice 是 Borsh 这个包的方法，用来序列化数据
    let mut greeting_account =
GreetingAccount::try_from_slice(&account.data.borrow())?;
    // 将读取到的数据加 1
    greeting_account.counter += 1;
    // 再把数据写入到存储账户里
    greeting_account.serialize(&mut *account.data.borrow_mut())?;

    msg!("Greeted {} time(s)!", greeting_account.counter);

    Ok(())
}
```

- 对应的调用函数：就是新建一个**数据**账户 GreetingAccount，然后调用合约的入口函数

 这里用的是 Legacy 交易，没有用第 4 部分的 V0 交易

```rust
// No imports needed: web3, borsh, pg and more are globally available

/**
 * The state of a greeting account managed by the hello world program
 */
class GreetingAccount {
  counter = 0;
  constructor(fields: { counter: number } | undefined = undefined) {
    if (fields) {
      this.counter = fields.counter;
    }
```

```
  }
}

/**
 * Borsh schema definition for greeting accounts
 */
const GreetingSchema = new Map([
  [GreetingAccount, { kind: "struct", fields: [["counter", "u32"]] }],
]);

/**
 * The expected size of each greeting account.
 */
const GREETING_SIZE = borsh.serialize(
  GreetingSchema,
  new GreetingAccount()
).length;

// Create greetings account instruction
const greetingAccountKp = new web3.Keypair();
const lamports = await pg.connection.getMinimumBalanceForRentExemption(
  GREETING_SIZE
);
const createGreetingAccountIx = web3.SystemProgram.createAccount({
  fromPubkey: pg.wallet.publicKey,
  lamports,
  newAccountPubkey: greetingAccountKp.publicKey,
  programId: pg.PROGRAM_ID,
  space: GREETING_SIZE,
});

// Create greet instruction
const greetIx = new web3.TransactionInstruction({
  keys: [
    {
      pubkey: greetingAccountKp.publicKey,
      isSigner: false,
      isWritable: true,
    },
  ],
  programId: pg.PROGRAM_ID,
});

// Create transaction and add the instructions
const tx = new web3.Transaction(); // 这里用的是 Legacy 交易，第 4 部分是 V0 交易
tx.add(createGreetingAccountIx, greetIx);

```

```
// Send and confirm the transaction
const txHash = await web3.sendAndConfirmTransaction(pg.connection, tx, [
  pg.wallet.keypair,
  greetingAccountKp,
]);
console.log(`Use 'solana confirm -v ${txHash}' to see the logs`);

// Fetch the greetings account
const greetingAccount = await pg.connection.getAccountInfo(
  greetingAccountKp.publicKey
);

// Deserialize the account data
const deserializedAccountData = borsh.deserialize(
  GreetingSchema,
  GreetingAccount,
  greetingAccount.data
);

console.log(
  `deserializedAccountData.counter :${deserializedAccountData.counter}`
);
```

5.3 项目架构

- 由于只有一个 entry_point，那我们没办法像 Solidity 一样调用不同的函数怎么办呢？我们就从 data 中获取用户想执行的函数即可
- 我们组织我们的代码如下

```
├── src
│   ├── entrypoint.rs # 定义合约入口函数，最终会调用"processor"里面定义的具体逻辑
│   ├── error.rs # 定义各种 error
│   ├── instruction.rs # 定义各个指令的数据结构
│   ├── lib.rs # rust工程的基本结构而存在，里面也可以定义一些脚手架工具函数
│   ├── processor.rs # 具体执行函数
│   └── state.rs # 在链上要存储的结构数据，类似 Model
```

✏️ Solana 的官方合约，例如 Token Program 就是按照如上的格式写的，可以去看看 Token Program 的代码就比较清楚了。具体来说，用户调用 Token Program 的一个函数，它会进入 entrypoint.rs 的主函数，主函数会根据 data 段的第一个字节来将其 data 反序列化，然后调用对应 processor.rs 中的处理函数。

反序列化用 `try_from_slice`，序列化 `serialize`

5.4 错误处理

- 合约错误返回 `ProgramError`，我们可以自定义 `PromgramError::Custom(u32)`，这比较容易让人想到使用 enum，因为 enum 里的值可以直接转成 u32
- 例子

```rust
#[derive(Clone, Debug, Eq, Error, FromPrimitive, PartialEq)]
pub enum HelloWorldError {
    #[error("Not owned by HelloWolrd Program")]
    NotOwnedByHelloWrold,
}

// 实现了 From trait 之后，就可以使用 into 函数来转变类型，从 HelloWorldError 到 ProgramError::Custom(u32)
impl From<HelloWorldError> for ProgramError {
    fn from(e: HelloWorldError) -> Self {
        ProgramError::Custom(e as u32)
    }
}

// 为了在 error 的时候显示报错信息，还需要实现 PrintProgramError 这个 trait
impl PrintProgramError for HelloWorldError {
    fn print<E>(&self)
    where
        E: 'static + std::error::Error + DecodeError<E> + PrintProgramError + FromPrimitive,
    {
        match self {
            HelloWorldError::NotOwnedByHelloWrold => msg!("Error: Greeted account does not have the correct program id!"),
        }
    }
}
```

5.5 VSCode 中开发

- 创建工程：`cargo new --lib xxx`，加入依赖 `cargo add solana-program`，在 `src/lib.rs` 中写合约
- 主要需要在 `Cargo.toml` 中加入

```
[features]
no-entrypoint = []

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

- 构建项目

```
cargo build-sbf
```

- 部署合约

```
solana program deploy target/deploy/helloworld.so
```

- 在新建一个 bin 文件，`cargo new --bin xxx`，在 `src/main.rs` 中添加 client 代码，注意这里用的是 rust 版的 solana_sdk，上面用的是 web3.js

```
use std::str::FromStr;

use solana_sdk::signature::Signer;
use solana_rpc_client::rpc_client;
use solana_sdk::signer::keypair;
use solana_sdk::transaction;
use solana_program::instruction;
use solana_program::pubkey;

const RPC_ADDR: &str = "https://api.devnet.solana.com";


fn main() {
    let helloworld = pubkey::Pubkey::from_str("FbLTBNZmc77xJpf4whkr4t7vdctjsk8DBkfuksqtQ7g8").unwrap();

    let me = keypair::Keypair::from_base58_string("VtqQ...xs8");
    println!("me is {}", me.pubkey());

    let client = rpc_client::RpcClient::new(RPC_ADDR);

    let account_metas = vec![
```

```
        instruction::AccountMeta::new(me.pubkey(), true),
    ];

    let instruction = instruction::Instruction::new_with_bytes(
        helloworld,
        "hello".as_bytes(),
        account_metas,
    );
    let ixs = vec![instruction];

    let latest_blockhash = client.get_latest_blockhash().unwrap();
    let sig = client.send_and_confirm_transaction(&transaction::Transaction::new_signed_with_payer(
        &ixs,
        Some(&me.pubkey()),
        &[&me],
        latest_blockhash,
    )).unwrap();

    println!("tx:{}", sig);
}
```

5.6 PDA 账号

- PDA 账户（Program derived addresses），由程序派生出的地址，只有程序 program_id 有权签名的帐户密钥
- 生成方法：
  - 链下

```
/**
 * Async version of findProgramAddressSync
 * For backwards compatibility
 *
 * @deprecated Use {@link findProgramAddressSync} instead
 */
static async findProgramAddress(
    seeds: Array<Buffer | Uint8Array>,
    programId: PublicKey,
): Promise<[PublicKey, number]> {
    return this.findProgramAddressSync(seeds, programId);
}
```

- 链上

```
pub fn find_program_address(seeds: &[&[u8]], program_id: &Pubkey) ->
(Pubkey, u8) {
    Self::try_find_program_address(seeds, program_id)
        .unwrap_or_else(|| panic!("Unable to find a viable program address
bump seed"))
}
```

- ATA 账户是特定的 PDA 账户，它是专门用来存放用户的某个 SPL Token 的

5.7 合约间调用 CPI

- `invoke`，不需要签名

```
use solana_program::{
    account_info::{AccountInfo, next_account_info},
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
    instruction,
    msg, program::invoke,
};


// Declare and export the program's entrypoint
entrypoint!(process_instruction);

// Program entrypoint's implementation
pub fn process_instruction(
    _program_id: &Pubkey, // Public key of the account the hello world
program was loaded into
    accounts: &[AccountInfo], // The account to say hello to
    _instruction_data: &[u8], // Ignored, all helloworld instructions are
hellos
) -> ProgramResult {

    // Iterating accounts is safer than indexing
    let accounts_iter = &mut accounts.iter();

    // Get the account to say hello to
    let account = next_account_info(accounts_iter)?;
    let helloworld = next_account_info(accounts_iter)?;

```

```
    msg!("invoke program entrypoint from {}", account.key);

    let account_metas = vec![
        instruction::AccountMeta::new(*account.key, true),
    ];

    let instruction = instruction::Instruction::new_with_bytes(
        *helloworld.key,
        "hello".as_bytes(),
        account_metas,
    );

    let account_infos = [
        account.clone(),
    ];

    invoke(&instruction, &account_infos[..]) // invoke 的 account_infos 这个参数和 instruction 的 account_metas 参数数据结构不一样
}
```

- `invoke_signed`，需要签名。下面是一个合约，其中 payer 是调用者，vault 是合约的 PDA 账户（事先通过链下计算得到地址、seed 即 b"vault"、bump，然后调用合约传入PDA地址和bump来创建），合约创建了一个 PDA 账户 vault，vault 是属于这个合约的

```
use borsh::{BorshSerialize, BorshDeserialize};
use solana_program::{
     pubkey::Pubkey,
    entrypoint::ProgramResult,
    program::invoke_signed,
    system_instruction,
    account_info::{
        AccountInfo,
        next_account_info,
    },
};
// The custom instruction processed by our program. It includes the
// PDA's bump seed, which is derived by the client program. This
// definition is also imported into the off-chain client program.
// The computed address of the PDA will be passed to this program via
// the `accounts` vector of the `Instruction` type.
#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct InstructionData {
   pub vault_bump_seed: u8,
   pub lamports: u64,
}
```

```rust

// The size in bytes of a vault account. The client program needs
// this information to calculate the quantity of lamports necessary
// to pay for the account's rent.
pub static VAULT_ACCOUNT_SIZE: u64 = 1024;
/
// The entrypoint of the on-chain program, as provided to the
// `entrypoint!` macro.
fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let payer = next_account_info(account_info_iter)?;
    // The vault PDA, derived from the payer's address
    let vault = next_account_info(account_info_iter)?;

    let mut instruction_data = instruction_data;
    let instr = InstructionData::deserialize(&mut instruction_data)?;
    let vault_bump_seed = instr.vault_bump_seed;
    let lamports = instr.lamports;
    let vault_size = VAULT_ACCOUNT_SIZE;

    // Invoke the system program to create an account while virtually
    // signing with the vault PDA, which is owned by this caller program.
    invoke_signed(
        &system_instruction::create_account(
            &payer.key,
            &vault.key,
            lamports,
            vault_size,
            &program_id,
        ),
        &[
            payer.clone(),
            vault.clone(),
        ],
        // A slice of seed slices, each seed slice being the set
        // of seeds used to generate one of the PDAs required by the
        // callee program, the final seed being a single-element slice
        // containing the `u8` bump seed.
        &[
            &[
                b"vault",
                payer.key.as_ref(),
                &[vault_bump_seed],
```

```
            ],
        ]
    )?;

    Ok(())
}
```

5.8 系统变量

- 一些系统变量：https://www.solanazh.com/course/6-4

5.9 ALTs

较复杂，暂略

6. Solana Anchor 合约开发

6.1 Anchor 基础

- 合约每个函数主要有两个输入：
  - 所有用到的账户及其属性 `ctx`
  - Data 字段 `instruction_data`

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

declare_id!("AHeB5XUYxhxH2nHq8fqUg5xic1qvGDDcqYDaxCZdK1bm"); // Anchor.toml中也写这个地址

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

    pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
        msg!("Greetings from: {:?}", ctx.program_id);
        Ok(())
    }

    pub fn instruction_one(ctx: Context<InstructionAccounts>, instruction_data: u64) -> Result<()> {
        // 调用这个函数的时候我们需要传入使用到的 Accounts，我们会从这些 Account 中读取或者写入信息
        // 这些需要使用到的 Accounts 我们就用 #[derive(Accounts)] 来定义成一个结构体
        ctx.accounts.account_name.data = instruction_data;
```

```
        Ok(())
    }
}

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

#[derive(Accounts)]
pub struct InstructionAccounts {
    // #[account] 表示一些 constraint，如果传入的 Account 不满足条件就不会执行 program
    // 这里的 init 表示的是，用户只需要提供一个地址，anchor 将对这个地址做初始化，具体步骤包括：
    // 1. 调用 SystemProgram.create_account 2. 分配空间 3. 设置默认值等
    // 要注意，这里 space 前面的 8 是 anchor 自带的，所以要取 AccountStruct 内的数据需要 offset 8 开始
    #[account(init, payer = user, space = 8 + 8)]
    pub account_name: Account<'info, AccountStruct>,
    #[account(mut)] // 对于 signer 这是必须的
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
    // 可以看到我们传入的 Account 可以有多种类型，每一种有不同的 validation 条件
    // 完整的类型列表：https://docs.rs/anchor-lang/latest/anchor_lang/accounts/index.html
    // constraint 列表：https://docs.rs/anchor-lang/latest/anchor_lang/derive.Accounts.html
}

// Define custom program account type，这里表示的是传入 Account 的 Data 段的结构
#[account]
pub struct AccountStruct {
    data: u64
}
#[account]
pub struct Counter {
    pub count: u64,
}
```

- 在测试网发布修改 `Anchor.toml` 中的 cluster 为 `"devnet"`
- 发布合约的私钥在 `target/deploy/xxx.json` 中，上面 declare_id 对应这个私钥的公钥，即合约地址
- 测试代码

```
import * as anchor from "@coral-xyz/anchor"
```

```
import { Program } from "@coral-xyz/anchor"
import { expect } from "chai"
import { AnchorCounter } from "../target/types/anchor_counter"

describe("anchor-counter", () => {
  // Configure the client to use the local cluster.
  const provider = anchor.AnchorProvider.env()
  anchor.setProvider(provider)

  const program = anchor.workspace.AnchorCounter as Program<AnchorCounter>

  const counter = anchor.web3.Keypair.generate()

  it("Is initialized!", async () => {
      // Add your test here.
      const tx = await program.methods
        .initialize()
        .accounts({ counter: counter.publicKey })
        .signers([counter])
        .rpc()

      const account = await program.account.counter.fetch(counter.publicKey)
      expect(account.count.toNumber()).to.equal(0)
    })

  it("Incremented the count", async () => {})
})
```

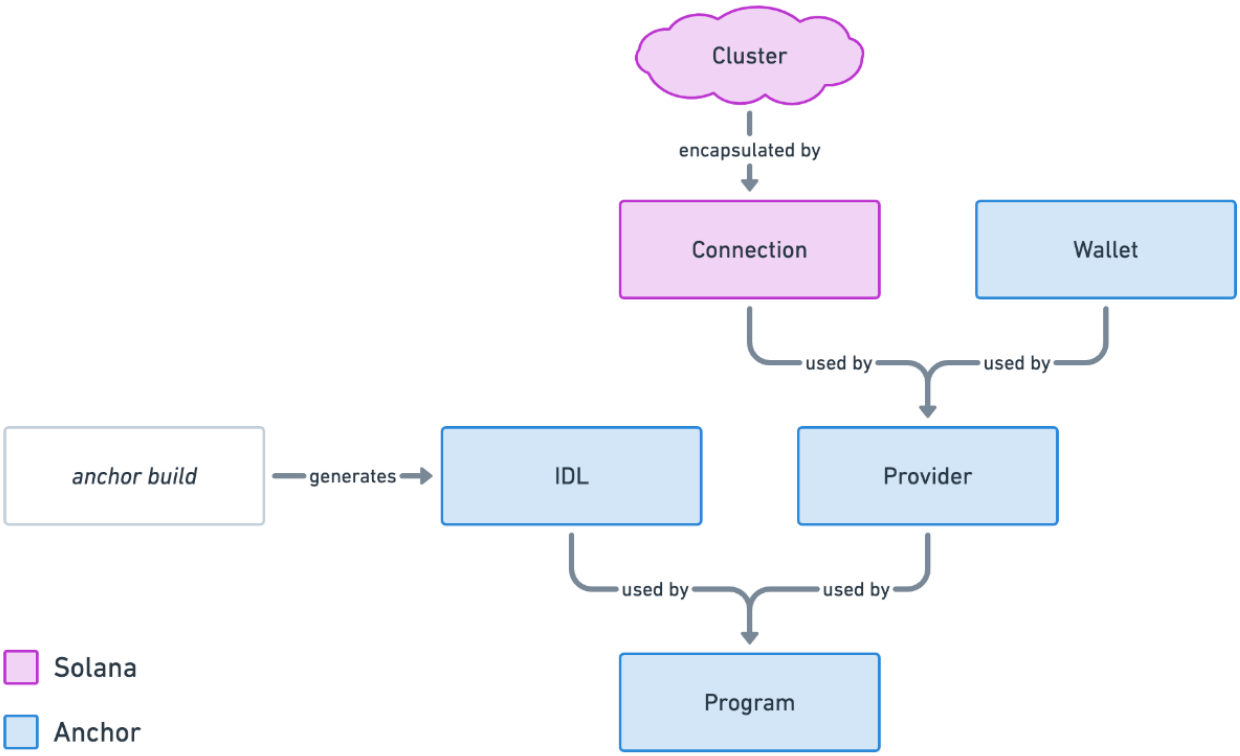
6.2 Anchor 开发框架

- 一个Anchor工程主要包含了
  - `declare_id` 宏声明的合约地址，用于创建对象的owner
  - `#[derive(Accounts)]` 修饰的Account对象，用于表示存储和指令账户
  - `program` 模块，这里面写主要的合约处理逻辑
- `ctx` 包含
  - `ctx.program_id`
  - `ctx.accounts`
  - `ctx.remaining_accounts`，所有传入但没有在 Accounts 结构体中声明的账户
  - `ctx.bumps`，在 Accounts 结构体中 PDA 的 bump

6.3 Anchor 合约调用

- 调用格式

```
// 1. 直接调用并发送
await program.methods
  .instructionName(instructionDataInputs) // 这里是函数参数
  .accounts({})
  .signers([]) // 如果 program 的 provider 的 wallet 是唯一的 signer，那这一行可以省略
  .rpc()

// 2. 还可以不用 rpc() 还可以用 transaction()
const transaction = await program.methods
  .instructionName(instructionDataInputs)
  .accounts({})
  .transaction()

await sendTransaction(transaction, connection)

// 3. 甚至还可以只包装成 instruction
// creates first instruction
const instructionOne = await program.methods
  .instructionOneName(instructionOneDataInputs)
  .accounts({})
  .instruction()

// creates second instruction
const instructionTwo = await program.methods
  .instructionTwoName(instructionTwoDataInputs)
  .accounts({})
  .instruction()

// add both instruction to one transaction
const transaction = new Transaction().add(instructionOne, instructionTwo)

// send transaction
await sendTransaction(transaction, connection)
```



- IDL 文件
  - 导入 `import idl from "./idl.json"`，采用 `@coral-xyz/anchor` 包，用的时候强转类型：

`new Program(idl as Idl, programId)`

  - https://github.com/metaplex-foundation/shank：生成 Native 合约的 IDL

```
{
  "version": "0.1.0",
  "name": "counter",
  "instructions": [
    {
      "name": "initialize",
      "accounts": [
        { "name": "counter", "isMut": true, "isSigner": true },
        { "name": "user", "isMut": true, "isSigner": true },
        { "name": "systemProgram", "isMut": false, "isSigner": false }
      ],
      "args": []
    },
    {
      "name": "increment",
      "accounts": [
        { "name": "counter", "isMut": true, "isSigner": false },
        { "name": "user", "isMut": false, "isSigner": true }
```

```
      ],
      "args": []
    }
  ],
  "accounts": [
    {
      "name": "Counter",
      "type": {
        "kind": "struct",
        "fields": [{ "name": "count", "type": "u64" }]
      }
    }
  ]
}
```

- Provider

```
import { AnchorProvider, setProvider } from "@coral-xyz/anchor"

const provider = new AnchorProvider(connection, wallet, {})
setProvider(provider)
```

- Prgram
  - https://coral-xyz.github.io/anchor/ts/classes/Program.html
- 获取 Program 对应的数据账户

```
const accounts = await program.account.counter.all()

const accounts = await program.account.counter.all([
    {
        memcmp: {
            offset: 8, // 注意 Anchor 会自动给 Account data 段最前面加 8 个字节，所以都要从 offset 8 开始
            bytes: bs58.encode((new BN(0, 'le')).toArray()),
        },
    },
])

const account = await program.account.counter.fetch(ACCOUNT_ADDRESS)

const accounts = await
program.account.counter.fetchMultiple([ACCOUNT_ADDRESS_ONE,
```

```
    ACCOUNT_ADDRESS_TWO])
```

6.4 Anchor PDA

- Anchor 中定义如下。当传入一个 PDA 的时候，Anchor 就会检查 `#[account]` 看是否符合 seeds 和 bump，若不符合就直接不进入 program logic

```
// 基础用法
#[derive(Accounts)]
struct ExampleAccounts {
  #[account(
    seeds = [b"example_seed"],
    bump
  )]
  pub pda_account: Account<'info, AccountType>,
}

// 使用其他字段内容
#[derive(Accounts)]
#[instruction(instruction_data: String)] // 可以使用 data 段的数据
pub struct Example<'info> {
    #[account(
        seeds = [b"example_seed", user.key().as_ref(),
instruction_data.as_ref()],
        bump
    )]
    pub pda_account: Account<'info, AccountType>,
    #[account(mut)]
    pub user: Signer<'info>
}

// init 的使用
#[derive(Accounts)]
pub struct InitializePda<'info> {
    #[account(
        init,
        seeds = [b"example_seed", user.key().as_ref()],
        bump,
        payer = user,
        space = 8 + 8
    )]
    pub pda_account: Account<'info, AccountType>,
    #[account(mut)]
    pub user: Signer<'info>,
    // init 使用必须 include system_program
```

```
    pub system_program: Program<'info, System>,
}

#[account]
pub struct AccountType {
    pub data: u64,
}

// #[instruction] 的调用
pub fn example_instruction(
    ctx: Context<Example>,
    input_one: String,
    input_two: String,
    input_three: String,
) -> Result<()> {
    ...
    Ok(())
}

#[derive(Accounts)]
#[instruction(input_one:String, input_two:String)]
pub struct Example<'info> {
    ...
}
```

- 空间大小分配：https://www.anchor-lang.com/docs/space
- `init_if_needed`

```
#[program]
mod example {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(
        init_if_needed,
        payer = payer,
        associated_token::mint = mint,
        associated_token::authority = payer
    )]
    pub token_account: Account<'info, TokenAccount>,
```

```
    pub mint: Account<'info, Mint>,
     #[account(mut)]
    pub payer: Signer<'info>,
    pub system_program: Program<'info, System>,
    pub token_program: Program<'info, Token>,
    pub associated_token_program: Program<'info, AssociatedToken>,
    pub rent: Sysvar<'info, Rent>,
}
```

- Realloc：重新给一个账户分配空间，如果空间变大 payer 付更多钱否则拿回部分钱

```
#[derive(Accounts)]
#[instruction(instruction_data: String)]
pub struct ReallocExample<'info> {
    #[account(
        mut,
        seeds = [b"example_seed", user.key().as_ref()],
        bump,
        realloc = 8 + 4 + instruction_data.len(), // 8 是 anchor 的 discriminator, 4 是 borsh String 存储长度
        realloc::payer = user,
        realloc::zero = false, // 是否需要将新分配空间设置成 0
    )]
    pub pda_account: Account<'info, AccountType>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>, // 一定需要引入 System Program
}

#[account]
pub struct AccountType {
    pub data: String,
}
```

- Close 关闭一个账户

```
pub fn close(ctx: Context<Close>) -> Result<()> {
    Ok(())
}

#[derive(Accounts)]
pub struct Close<'info> {
    #[account(mut, close = receiver)]
```

```
    pub data_account: Account<'info, AccountType>,
    #[account(mut)]
    pub receiver: Signer<'info>
}
```

- 应用可以参考 https://www.soldev.app/course/anchor-pdas 中的 Lab

6.5 Anchor CPI

参考：https://www.soldev.app/course/anchor-cpi

7. Solana DApp 开发实践

8. Solana 合约安全