## 第9章Python语言离线钱包开发

★本章导读★

对很多人来说，区块链还非常神秘，大多数人接触区块链的第一个产品多是钱包。可以说，钱包是通向区块链世界的一个窗户。本章我们将用Python语言编写一个命令行版本的区块链钱包，捅破钱包这层窗户纸，可以更深刻的理解区块链技术。

★知识要点★

通过本章内容的学习，您将掌握以下知识：

* 区块链钱包的相关术语
* 区块链钱包的原理
* 助记词的原理与实现
* 私钥存储的原理与实现
* Coin交易的原理与实现
* ERC-20同质化Token标准
* Token交易的原理与实现

### 9.1 区块链钱包原理

随着区块链和数字货币的普及，钱包必将成为第一批走进大众生活的区块链应用。本节，我们将采用理论与代码实践相结合的方式去介绍钱包的核心原理、助记词的生成方式以及私钥存储。

#### 9.1.1 区块链钱包的核心原理

对于钱包，大家并不陌生，在移动支付并未占据主导的那些年月，钱包是我们出行的必备物品。由于移动支付所带来的便捷和种种好处，现在大多数人已经不再携带钱包了，有些人即使携带钱包，更多的也是保存银行卡或身份证，这样的功能更像是卡包。从某种意义上说，区块链钱包更像是“卡包”的功能，它内部不会直接存放“现金”，实际保存的是用户的私钥，因为有了私钥就代表拥有一切。这就跟此前介绍的内容联系上了，其实无论任何区块链项目，私钥都是最重要的元素。

实际上，在前面我们已经使用过钱包产品了，以太坊的客户端Geth它本身就具有钱包的功能，否则它是无法帮助账户完成交易以及合约调用的。区块链钱包按照联网情况、节点数据同步情况来划分可以分为全节点钱包、冷钱包、热钱包、中心化钱包以及轻钱包。冷钱包和热钱包主要是指从联网情况上来说，私钥一直处于区块链网络中的属于热钱包，冷钱包则是只有在交易的时候联网，交易后立即断开。全节点钱包很好理解，它会同步全部区块的数据，而轻钱包则只保存跟自己相关的数据。中心化钱包则是完全依赖某公司或机构的中心化服务器，比如交易所就是中心化钱包，它负责保管用户的私钥。

钱包按照展现形式又可以分为手机钱包、网页钱包、硬件钱包、纸钱包、脑钱包等，纸钱包和脑钱包主要是指记录私钥的方式靠纸和大脑，现在使用最广泛的是手机钱包，毕竟大家已经离不开智能手机了，网页钱包就是类似MetaMask这样的浏览器插件钱包，硬件钱包多是离线钱包，一般借助一个小屏幕显示二维码，在交易的那一刻联网或借助其他设备完成交易。我们要做的钱包是一个只有交易时才会联网、命令行展现形式的钱包，所以我管它叫命令行版离线钱包。

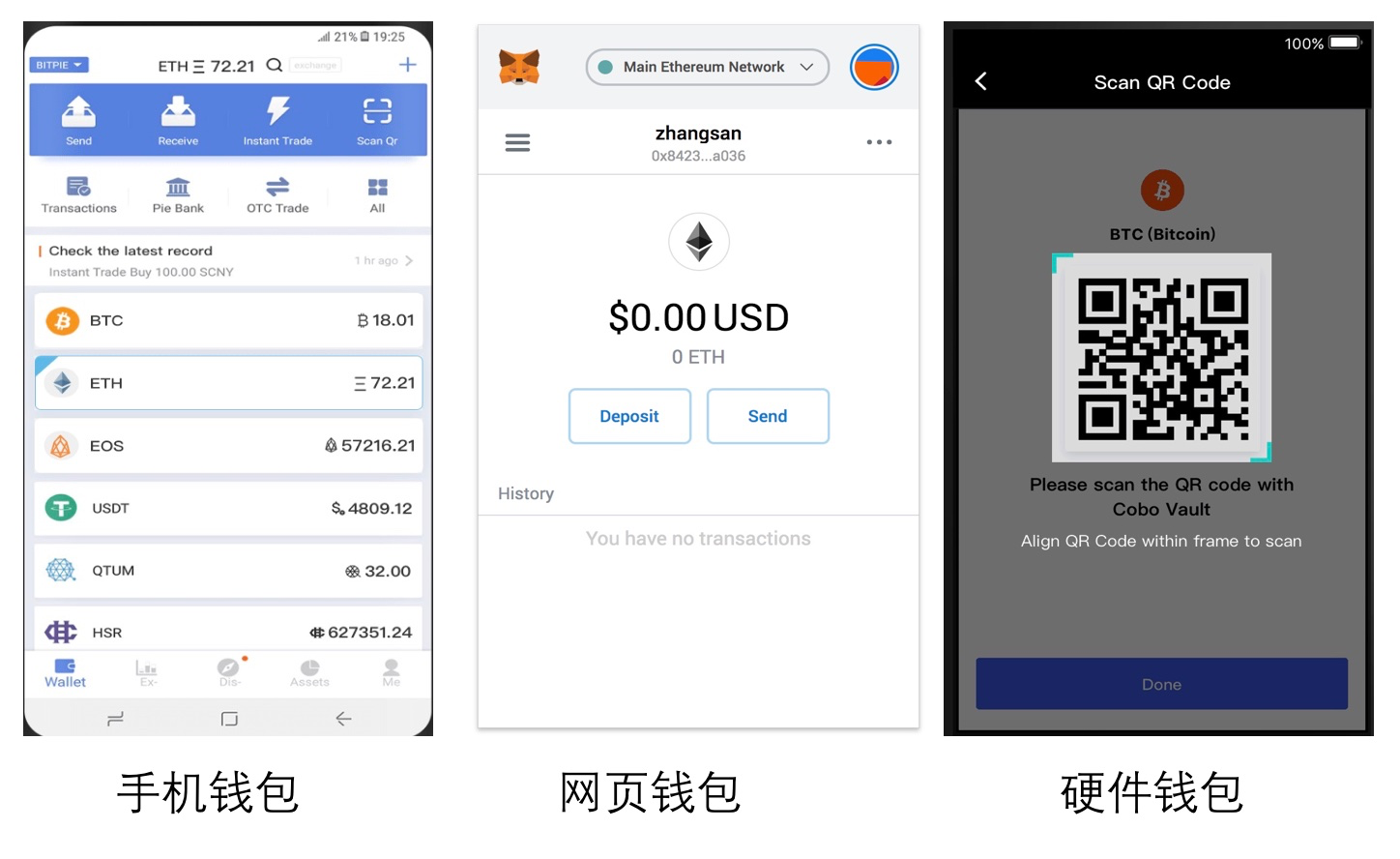


图9-1 钱包的分类

区块链钱包到底是怎么回事已经很清楚了，钱包最关键的功能是保存私钥。私钥光靠钱包存储也并不安全，任何一款钱包产品都会提醒用户自己备份好私钥。但是，私钥这样敏感的内容不适合手机拍照，最稳妥的方式是用纸抄录下来，并秘密保存起来。用纸抄又面临着抄错的风险，毕竟私钥是一大长串无规律字符，记忆起来太反人类了。

私钥存储的问题，包括关于比特币的其他问题，在比特币社区都会有人发布改进思路，进行热烈讨论后，最终形成比特币改进提案（Bitcoin Improvement Proposals，简称BIP），由此可见，BIP对于比特币是非常重要的。在BIP32中，提出了比特币钱包的改进提案，改进方式是通过一个seed可以产生一树状结构存储多组秘钥对（公钥和私钥），这样的好处是只备份seed就可以备份整个体系内的私钥，适合公司或集团化的私钥管理，这种钱包被命名为分层确定性钱包（Hierarchical Deterministic wallet ，简称HD Wallet）。

此后，又有提案继续改进钱包，在BIP39提案中，将seed用一组方便记忆和书写的单词表示，这就有了助记词的概念。在BIP44提案中，基于HD Wallet的特点，又针对钱包路径进行了定义，通过多层次的目录表示，让钱包可以支持多币种。目录定义方式如下：

m / purpose' / coin\_type' / account' / change / address\_index

我们来分别介绍一下目录内的各个元素：

* purpose 提案编号，比如39，44
* coin\_type 币种，可以是比特币（0）或以太坊（60），或其他数字货币
* account 再细分独立的逻辑性亚账户，比如0，1，2
* change HD钱包两个子树，一个用来接收地址，一个用来找零
* address\_index 地址编号，在这一层，编号不同可以对应多个不同子账户

下图9-2是一款有图形用户界面的以太坊钱包（Ganache），它就是一款测试用的分层确定性钱包，我们可以直接在图中看到助记词以及HD PATH信息。图中第一个以太坊地址**0x48C3FfAB87c6E3C1eeF47BF7d7e5ef9F36F26e00**所对应的HD PATH是m/44'/60'/0'/0/0。

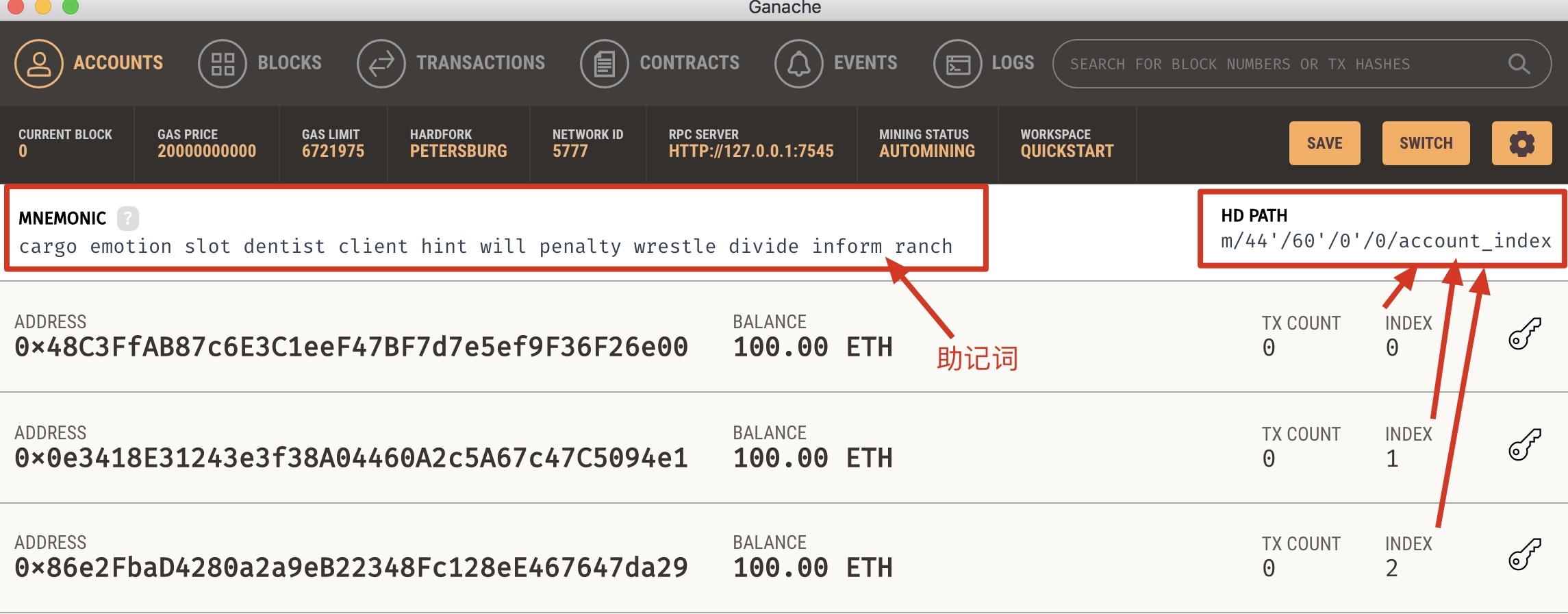


图9-2 Ganache客户端展示

说到这儿，读者应该猜到了，我们要实现的应该是一个分层确定性钱包。总结下来这个过程就是利用助记词生成一个seed，使用seed基于不同路径（HD PATH）可以推导出不同的秘钥对，也就是不同的子账户了。如下图所示：

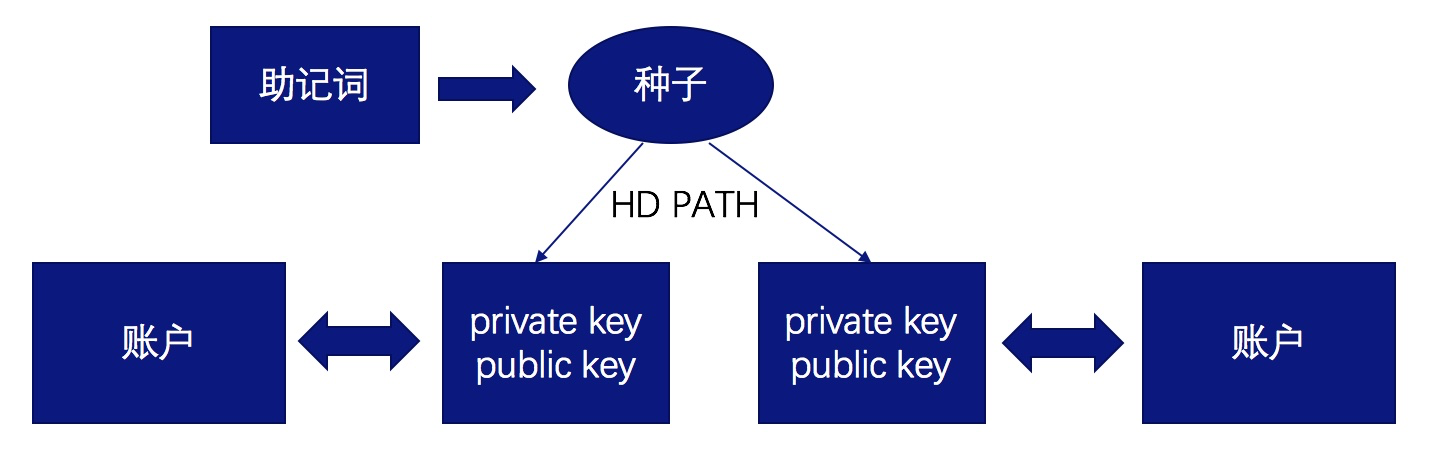


图9-3 助记词、私钥、账户之间的关系

获得私钥之后呢？私钥都在手里了，还有什么是做不了的？

#### 9.1.2 助记词如何生成与验证

此前，我们已经介绍了钱包的核心原理，核心思想就是通过助记词推导出私钥，然后就可以为所欲为了。接下来我们来介绍如何生成助记词，首先，可以先了解一下助记词的生成原理。如下图9-4所示。

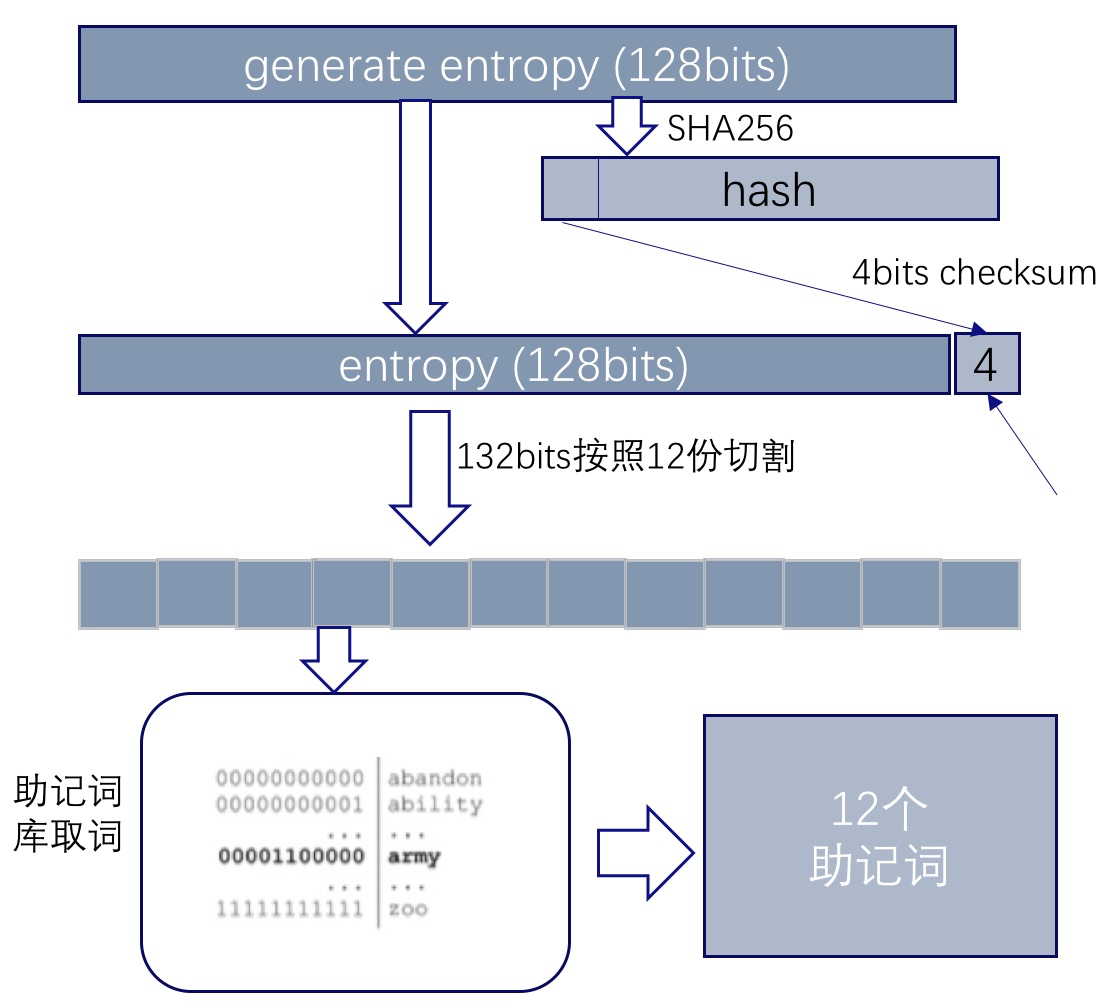


图9-4 助记词产生原理

总结一下该图（以12个单词助记词为例），步骤如下：

1. 生成无序状态熵（entropy），它是一个128字节的随机数
2. 对entropy计算哈希，取前4字节获得校验和
3. 将校验和连接在entropy尾部，形成132字节的数
4. 对132字节的数据进行12等份切割
5. 每一份数据换算为十进制后到助记词库（BIP39，词库内包含2048个单词）内查找对应编号的单词
6. 将12个单词按顺序连接形成最终助记词

上述过程看上去很复杂，但通过Python来编写代码比较容易，因为已经有人把它们封装好了。借助Mnemonic这个第三方库可以非常便捷地体验助记词。调用两个函数就可以创建助记词了。示例代码如下，第2行使用了Mnemonic库，并且指定使用英文助记词，第3行代码则是随机创建了助记词。

1. from mnemonic import ( Mnemonic )
2. mnemonic = Mnemonic("english")
3. mnemonic\_sentence = mnemonic.generate()

使用Mnemonic对象，也可以指定简体中文、法语等语言作为助记词。另外，generate函数在执行时也可以指定强度，一般选择strength = 128（熵的位数），即 12 个单词。

执行该代码，就可以看到类似下面的效果，如果重复执行，每次的助记词会不一样。

1. >>> mnemonic.generate(strength=128)
2. 'perfect parent search food traffic ceiling sense mixture brother hello try safe'

介绍到这里，助记词已经不再是问题。有了助记词，就可以推导出种子，通过种子就可以获得私钥以及账户地址了。接下来我们介绍如何利用助记词推导出私钥以及账户地址，步骤如下：

1. 通过助记词和随机噪声生成种子seed
2. 获取seed中的masterPrivateKey
3. 借助eth\_account库的Account 对象利用masterPrivateKey生成一个账户对象 account
4. 获取account对象中的 address（账户地址）和privateKey（私钥）。

下面我们定义一个 class Wallet 来完成上面的功能：

首先安装所需库并引入：

1. $ pip install eth\_account
2. $ pip install eth\_keys
3. $ pip install Mnemonic

Wallet 实现代码如下：

首先在初始化方法 \_\_init\_\_ 中初始化配置信息configuration，账户信息account和助记词mnemonic\_sentence。

1. class Wallet(object):
2. def \_\_init\_\_(self, configuration):
3. self.conf = configuration
4. self.account = None
5. self.mnemonic\_sentence = None

而后定义 create 函数来创建新钱包，它接收 2 个参数：password 代表口令， restore\_sentence则是助记词，默认无需填写，这样会新建一个新的钱包；若填写了，则表示用之前生成过的助记词和密码来解锁和显示已有钱包的账户信息。

1. class Wallet(object):
2. def \_\_init\_\_(self, configuration):
3. self.conf = configuration
4. self.account = None
5. self.mnemonic\_sentence = None
6. def create(self, password='', restore\_sentence=None):
7. """
8. create 函数可以创建新钱包，
9. :param password: 作为存储在 configuration 中的的钱包密码和随机的噪声
10. （钱包软件在创建私钥 Private Key 时，有必要引入随机性来保证密码学上的私钥安全）
11. :param restore\_sentence: 用于从助记词恢复钱包
12. :return: 带有私钥的对象
13. """
14. extra\_entropy = password
16. *# 助记词对象*
17. mnemonic = Mnemonic("english")
18. if restore\_sentence is None:
19. *# 生成助记词*
20. self.mnemonic\_sentence = mnemonic.generate(strength=128)
21. else:
22. self.mnemonic\_sentence = restore\_sentence
23. *# 1. 通过助记词和随机噪声生成种子seed*
24. *## 如果用户输入了助记词和 password，那么 seed 就是之前生成过的确定值，从而私钥和 address 也是确定的*
25. seed = mnemonic.to\_seed(self.mnemonic\_sentence, extra\_entropy)
26. *# 2. 获取**seed中的masterPrivateKey*
27. master\_private\_key = seed[32:]
28. *# 3. 借助 eth\_account 库的 Account 对象利用 masterPrivateKey 生成一个账户对象 account*
29. self.account = Account.privateKeyToAccount(master\_private\_key)
30. *# update config address*
31. self.conf.update\_eth\_address(self.account.address)
32. *# convert 类似 b'\xfe1h\xc5B\x14tV\xbe\xfe.. to 0xfe3168c54..*
33. priv\_key = keys.PrivateKey(self.account.privateKey)
34. *# update config public key*
35. pub\_key = priv\_key.public\_key
36. self.conf.update\_public\_key(pub\_key.to\_hex())
37. return self

上述代码中的configuration定义了一些关于 keystore（后面会介绍）的存储位置的相关信息，这里先不需要关注，重点观察助记词生成 seed，以及 seed 生成 account和对应私钥、地址的过程即可。

若想测试该类及其成员函数，可以在终端（命令行）中进行测试，这里使用 Python 中实现命令行界面的库click来完成代码测试，首先安装 click库，命令如下。

1. $ pip install click

在根目录中新建 cli 目录，并在其下新建一个new\_wallet.py 文件用来测试钱包的创建功能：

1. import click
2. import getpass
3. import sys
4. sys.path.append("..")
5. from Wallet import ( Wallet )
6. from configuration import (Configuration,)
7. @click.command()
8. def new\_wallet():
9. """Creates new wallet and store encrypted keystore file."""
10. *# 提示用户输入密钥库文件的密码*
11. password = getpass.getpass('Passphrase from keystore: ')
12. configuration = Configuration().load\_configuration()
13. wallet = Wallet(configuration).create(password)
14. click.echo('Account address: %s' % str(wallet.get\_address()))
15. click.echo('Account pub key: %s' % str(wallet.get\_public\_key()))
16. click.echo('Remember these words to restore eth-wallet: %s' % wallet.get\_mnemonic())
17. if \_\_name\_\_ == '\_\_main\_\_':
18. new\_wallet()

在命令行执行“python3 new\_wallet.py”并按提示输入口令，就可以看到wallet的创建信息，屏幕上显示了地址、公钥和助记词的信息（记得保存、抄写好自己的助记词和口令，这是恢复钱包和执行交易的最关键信息）。执行效果如下图9-6所示：



图9-6 助记词、地址生成演示

#### 9.1.3 如何存储私钥

虽然我们把助记词搞定了，但是让用户每次都输入助记词再来操作钱包也是非常反人性的，为了给用户更好的体验，不能每次输入助记词之后才能交易，为此，钱包还是要想办法把私钥存储起来，同时也要确保私钥的安全。

我们知道私钥非常重要，但如果只是将私钥保存到电脑的某个位置，如果某天被 黑客攻击，黑客拿到该私钥文件后，便可以依据私钥对钱包资产进行肆意操作。所以，我们需要某种加密方式，对这串难记的私钥地址进行加密，然后再进行存储，这样即使黑客拿到了加密后的文件，他没有密码，也很难破解出真正的私钥。

那么如何做到更好的操作体验和更安全的存储方式呢？以太坊使用了keystore。keystore 是一种存储钱包对应私钥的方式。keystore 对私钥进行加密后，保存为 json 文件存储在本地。

使用keystore的好处有哪些呢？第一很安全，攻击者不仅需要破解keystore，还需要知道你的钱包口令；第二很方便，使用者无需记忆一串毫无规律的私钥或者频繁使用助记词来解锁钱包以及发送交易。下图9-7解释了私钥和keystore文件的对应关系。

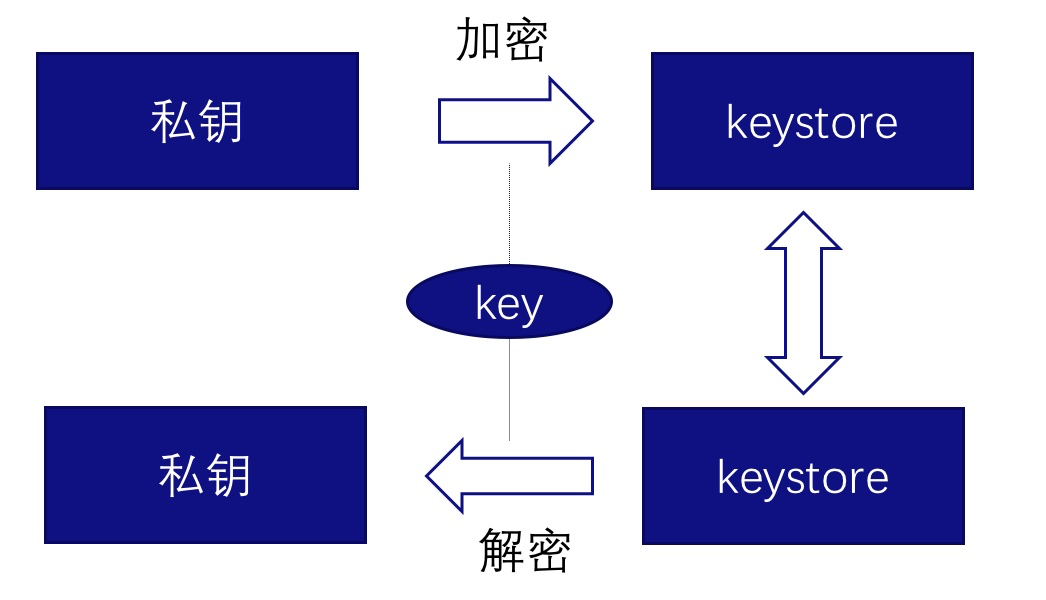


图 9-7 keystore 加、解密图示

我们使用的Metamask钱包实际上也是在内部保存了keystore文件，因此我们解锁钱包时需要一个口令即可。解锁之后得到私钥，就可以使用私钥对一笔笔交易进行签名了。

若读者对keystore文件感兴趣，可以看看下图 9-8，它是一个keystore文件的示例，大概的说明如下，address代表账户地址，crypto显示了该文件的加密算法信息，这里特别强调一下mac，口令正确后的解析结果要求与mac必须一致，否则认为口令错误。



图 9-8 keystore 文件概览

keystore文件的生成使用的是对称加密算法，这个道理很简单，因为我们使用同一个口令加密和解密。keystore文件中的mac是用来核验口令正确的，使用任何一个口令都可以处理该文件，但只有最终与mac匹配才算解密成功。

在了解了私钥存储的原理后，下面我们用尝试用代码来实现keystore 的加、解密过程。

首先处理加密，我们为Wallet增加一个名为save\_keystore的成员函数。由于钱包本身包含私钥信息，所以传入一个口令就具备加密的条件了。具体编码时，通过 eth\_account库提供的Account.encrypt函数构建keystore结构，再调用keystore的存储接口，将私钥存储为文件。示例代码如下。

1. def save\_keystore(self, password):
2. """
3. 将[私钥+passphrase] 通过 Account.encrypt 加密成 keystore JSON 文件
4. 保存在本地目录中。
5. :param password: 用户输入的 passphrase
6. :return: 存储路径
7. """
8. create\_directory(self.conf.keystore\_location)
9. keystore\_path = self.conf.keystore\_location + self.conf.keystore\_filename
10. encrypted\_private\_key = Account.encrypt(self.account.privateKey, password)
11. with open(keystore\_path, 'w+') as outfile:
12. json.dump(encrypted\_private\_key, outfile, ensure\_ascii=False)
13. return keystore\_path

接下来再搞定解密。解密需要使用eth\_account 库提供的Account.decrypt函数，它需要读取本地存储的 keystore 文件，并且需要解锁的口令。接下来，我们再实现一个load\_keysotre函数，代码如下。

1. def load\_keystore(self, password):
2. """
3. 从解密的 keystore 中加载钱包帐户（私钥）
4. :param password: 用户输入的 passphrase
5. :return:  account 钱包账户实例
6. """
7. keystore\_path = self.conf.keystore\_location + self.conf.keystore\_filename
8. with open(keystore\_path) as keystore:
9. keyfile\_json = json.load(keystore)
10. try:
11. private\_key = Account.decrypt(keyfile\_json, password)
12. except ValueError:
13. raise InvalidPasswordException()
14. self.set\_account(private\_key)
15. return self

接下来，我们对再对创建钱包的new\_wallet 函数稍加改造，加入save\_keysotre这部分代码，这样就可以在创建钱包的同时，将钱包的私钥等敏感信息存储到 keystore 文件中，示例代码如下。

1. @click.command()
2. def new\_wallet():
3. """Creates new wallet and store encrypted keystore file."""
4. *# 提示用户输入密钥库文件的密码*
5. password = getpass.getpass('Passphrase from keystore: ')
6. configuration = Configuration().load\_configuration()
7. wallet = Wallet(configuration).create(password)
8. wallet.save\_keystore(password)
9. click.echo('Account address: %s' % str(wallet.get\_address()))
10. click.echo('Account pub key: %s' % str(wallet.get\_public\_key()))
11. click.echo('Keystore path: %s' % configuration.keystore\_location + configuration.keystore\_filename)
12. click.echo('Remember these words to restore eth-wallet: %s' % wallet.get\_mnemonic())
13. if \_\_name\_\_ == '\_\_main\_\_':
14. new\_wallet()

在命令行再次执行“python3 new\_wallet.py”并输入 passphrase（密码）后，就可以看到创建了钱包账户，并打印出了地址、公钥、keystore 文件和助记词的信息。执行效果如下图9-9所示。

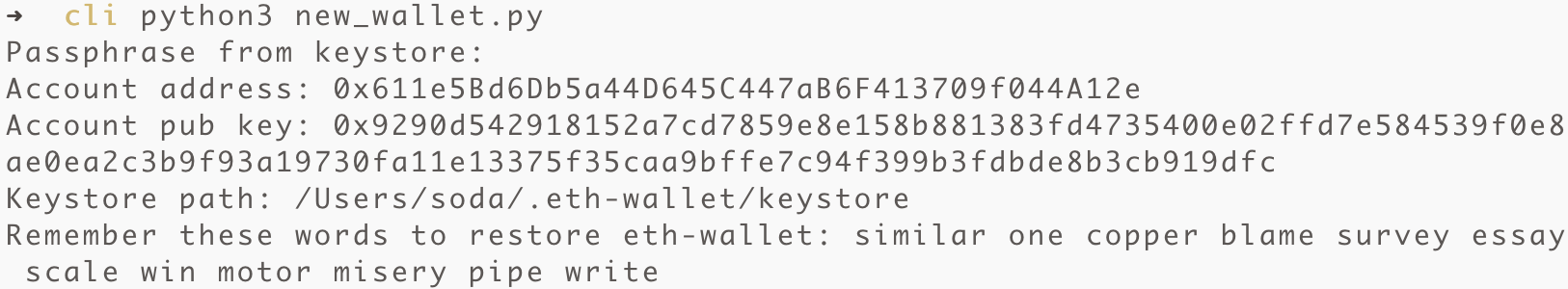


图 9-9 新建钱包时的keystore 存储信息

### 9.2 区块链钱包核心功能实现

在前一节，我们理解了钱包的原理，助记词和私钥之间的关系，以及私钥如何存储的问题。仔细一想，其实我们已经完成了钱包的创建工作，接下来我们介绍如何实现钱包的其他核心功能。

#### 9.2.1 钱包如何支持Coin转移

钱包最核心要支持的功能便是Coin转移，也就是指以太坊的ETH相关的转账交易（实际以太坊所有的操作都是交易）。

在上一节中我们新建了钱包，但是新创建的钱包是没有任何ETH余额的，我们不妨到测试链的水龙头（<https://goerlifaucet.com/>）去领取一些。

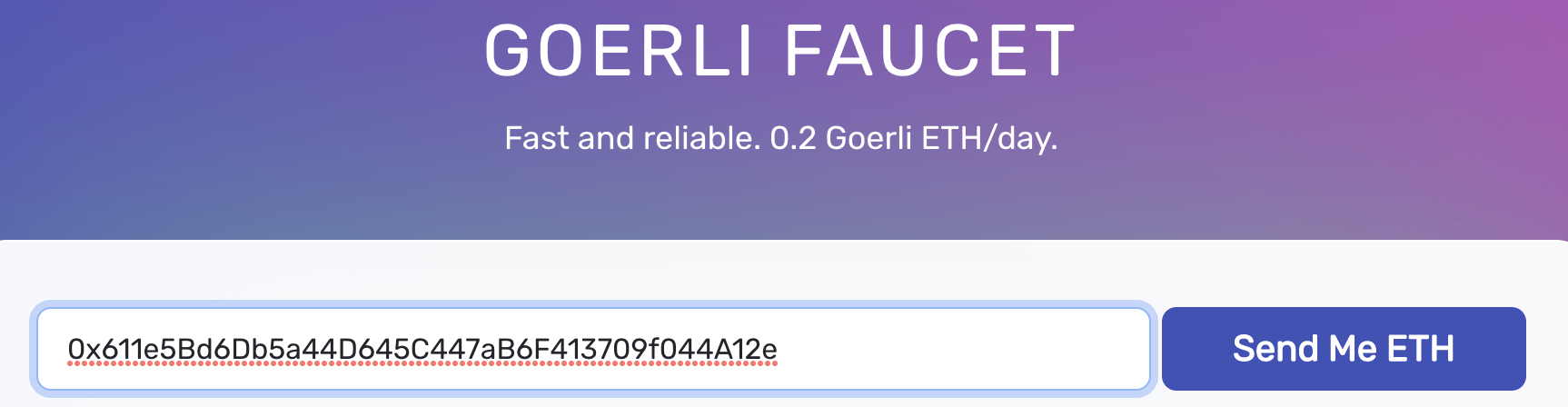


图 9.2.1-3 faucet领取0.2 个Goerli ETH

有了ETH，便可以尝试着从钱包发送交易给其他账户，在和以太坊交互的过程中，有两个易用的中间件需要了解和掌握，分别是Infura 和Web3.py，下面我们分别来介绍：

Web3.py 是一个用于与以太坊交互的 Python 库。它常见于去中心化应用程序 (dapps) 的应用中，来帮助客户端发送交易、与智能合约交互、读取区块数据以及各种其他用途。其原始 API 派生自 Web3.js的Javascript API。

安装Web3.py：

1. $ pip install web3==4.9.1
2. $ pip install python-dotenv

Infura是一种 IaaS（Infrastructure as a Service）产品，目的是为了降低开发者和用户访问以太坊数据的门槛，通俗来说，Infura作为供应商，可以利用其背后庞大的负载均衡网络，让开发者的 Dapp 快速接入以太坊网络，而不需要在本地运行昂贵的以太坊节点。包括 Metamask、Uniswap 等著名的 Web3 应用都利用了Infura的 API 服务。

Infura使用方法：访问并注册 Infura 官网，获取API KEY，而后将Infura 提供的 API KEY 存放到根目录的 .env 文件中，详细步骤可以参考这个链接：<https://blog.infura.io/post/getting-started-with-infura-28e41844cc89> 。

在根目录新建一个 infura.py文件，该文件主要用于使用 we3.py 和 Infura 提供的接口连接以太坊网络。代码主要内容是导入 web3 库，获取环境变量中的 Infura API KEY，将其拼接为可访问的 infura\_url，而后使用 Web3.HTTPProvider将其包装成一个 web3 对象：

1. from web3 import ( Web3, )
2. from dotenv import load\_dotenv
3. import os
4. load\_dotenv()
5. API\_KEY = os.getenv("API\_KEY")
6. infura\_url = "https://goerli.infura.io/v3/"+API\_KEY
7. class Infura:
8. def \_\_init\_\_(self):
9. self.web3 = Web3(Web3.HTTPProvider(infura\_url))
10. def get\_web3(self):
11. return self.web3

定义好 class Infura后，对 Wallet.py 中的 class Wallet 稍加修改，将 Infura API 包含到 class Wallet 里面：

1. from infura import ( Infura )
2. class Wallet(object):
3. def \_\_init\_\_(self, configuration):
4. *# the rest of the code...*
5. self.w3 = None
6. def create(self, password='', restore\_sentence=None):
7. *# the rest of the code...*
8. self.w3 = Infura().get\_web3()
9. return self

为了处理交易过程中可能发生的错误 Error，在根目录下面新建一个exceptions.py 文件：

1. class InsufficientFundsException(Exception):
2. """当用户想要发送ETH但地址余额不足时抛出"""
3. pass
4. class InsufficientERC20FundsException(Exception):
5. """当用户想要发送ERC-20但地址余额不足时抛出"""
6. pass
7. class ERC20NotExistsException(Exception):
8. """当用户想要操作钱包中不存在的 Token时抛出"""
9. pass
10. class InvalidTransactionNonceException(Exception):
11. """当出现重复的 nonce 或其他问题时抛出"""
12. pass
13. class InvalidValueException(Exception):
14. pass
15. class InvalidPasswordException(Exception):
16. pass
17. class InfuraErrorException(Exception):
18. """当钱包无法正确连接到 infura 节点时触发 """

至此，我们拥有了 web3.py和Infura API，下面就可以编写代码实现 ETH 的交易发送了。

发送 ETH 交易的详细过程如下：

1. 从命令行输入 recipient 地址和要发送的ETH 数量
2. 从用户输入读取 passphrase，读取本地keystore，解密出 PrivateKey，稍后用作交易签名
3. 从 class Infura中获取 web3 对象，作为访问以太坊网络的接口对象。
4. 构建待发送的交易对象字典 raw\_txn{}
5. 调用web3.eth.account.signTransaction，将交易对象使用私钥进行签名
6. 调用web3.eth.sendRawTransaction，发送已签名和被序列化的交易对象给以太坊网络处理；该函数接受HexBytes 类型的 tx\_hash返回值
7. 获取到tx\_hash返回值后，在对应的etherscan网站查看tx\_hash.hex()这笔交易的交易详情

详解代码如下：

1. import click
2. import sys
3. sys.path.append("..")
4. import getpass
5. from Wallet import ( Wallet )
6. from infura import ( Infura, )
7. from configuration import ( Configuration, )
8. from exceptions import \*
9. from web3.exceptions import ( InvalidAddress, )
10. @click.command()
11. @click.option('-t', '--to', default='', prompt='To address:',
12. help='Ethereum address where to send amount.')
13. @click.option('-v', '--value', default='', prompt='Value to send:',
14. help='Ether value to send.')
15. @click.option('--token', default=None,
16. help='Token symbol.')
17. def send\_transaction(to, value, token):
18. """
19. 签署和发送交易 Sign and send transaction.
20. :param to: 接受者的 address
21. :param token\_symbol: None表示发送ETH； 否则发送 ERC20 Token
22. :param gas\_price\_speed: gas price 将乘以这个数字来加速交易
23. """
24. *# 从用户输入读取 passphrase*
25. passphrase = getpass.getpass('passphrase from keystore: ')
26. *# load 本地的 keystore file*
27. configuration = Configuration().load\_configuration()
28. try:
29. *# send ETH 交易*
30. if token is None:
31. to\_address = to
32. wallet = Wallet(configuration).load\_keystore(passphrase)
33. web3 = Infura().get\_web3()
34. try: float(value)  *# 输入检查*
35. except ValueError: raise InvalidValueException()
37. *# 交易对象构建 transaction object*
38. raw\_txn = {
39. 'nonce': web3.eth.getTransactionCount(wallet.get\_address()),
40. 'from': wallet.get\_address(),
41. 'to': to\_address,
42. 'value': web3.toWei(value, 'ether'),
43. 'gasPrice': 25000000000,
44. 'chainId': 5  *# goerli Testnet*
45. }
47. *# 执行消息调用或交易并返回预估消耗的 gas fee*
48. gas = web3.eth.estimateGas(raw\_txn)
49. raw\_txn['gas'] = gas *# 修正 gas*
50. *# 使用私钥签署交易*
51. signed\_tx = web3.eth.account.signTransaction(raw\_txn, wallet.get\_account().privateKey)
52. *# 发送已签名和被序列化的交易对象；返回 HexBytes 类型的 txhash*
53. tx\_hash = web3.eth.sendRawTransaction(signed\_tx.rawTransaction)
55. print("Transaction hash: " , tx\_hash.hex())
56. except InsufficientFundsException:
57. click.echo('Insufficient ETH funds! Check balance on your address.')
58. except InsufficientERC20FundsException:
59. click.echo('Insufficient ERC20 token funds! Check balance on your address.')
60. except InvalidAddress:
61. click.echo('Invalid recipient(to) address!')
62. except InvalidValueException:
63. click.echo('Invalid value to send!')
64. except InvalidPasswordException:
65. click.echo('Incorrect password!')
66. except InfuraErrorException:
67. click.echo('Wallet is not connected to Ethereum network!')
68. except ERC20NotExistsException:
69. click.echo('This token is not added to the wallet!')
70. if \_\_name\_\_ == '\_\_main\_\_':
71. send\_transaction()

在命令行中执行交易，分别输入接受者地址 to\_address 和交易金额 value，然后输入 passphrase 解锁 keystore中的私钥签署交易、发送交易，而后可以发现命令行中打印出了交易哈希：



图 9.2.1-4 执行 send\_transaction 并打印交易哈希

获取交易哈希后，便可以到对应以太坊网络的etherscan 查看交易的详细信息：<https://goerli.etherscan.io/tx/0x46d8c54c9ae44a9d3ece6312439b76607885ef717bc428626cebef08e56d3001> 该交易详情如下图：

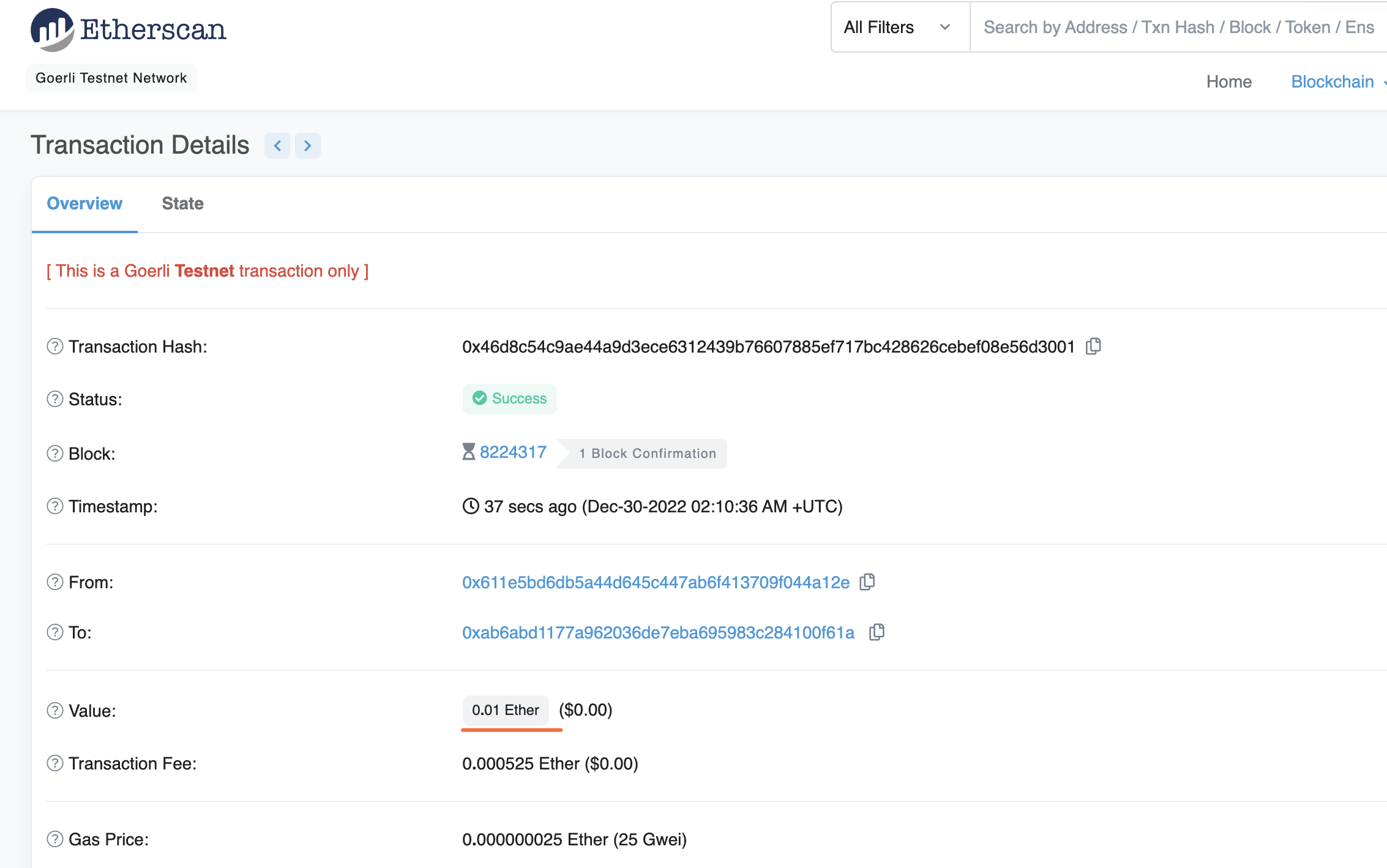


图 9.2.1-5 goerli.etherscan.io交易详情显示

#### 9.2.2 钱包如何支持Coin查询

虽然刚刚实现了ETH Coin转移，但是Coin余额的查询还只能在网络中查询，本部分我们要在钱包的命令行中实现支持余额的查询功能。

余额查询相对转移交易简单许多，它只是查询以太坊网络中的数据，而不会涉及签名等操作。查询余额的核心方法是 Web3.py 提供的Eth.getBalance：

在 class Wallet中增加get\_balance方法：

1. def get\_balance(self, address):
2. """
3. 从以太坊网络中读取账户余额
4. :return: 用户账户的以太币数量
5. """
6. self.w3 = Infura().get\_web3()
7. *# eth.get\_balanc 获取余额*
8. *# web3.fromWei 将 wei 值转换为以太币的数量*
9. eth\_balance = self.w3.fromWei(self.w3.eth.get\_balance(address), 'ether')
10. return eth\_balance

在cli文件夹下新建get\_balance.py文件：

1. import click
2. import sys
3. sys.path.append("..")
4. from Wallet import ( Wallet )
5. from configuration import ( Configuration, )
6. from web3.exceptions import \*
7. from exceptions import \*
8. @click.command()
9. @click.option('-t', '--token', default=None,
10. help='Token symbol.')
11. def get\_balance(token):
12. """获取账户余额."""
13. configuration = Configuration().load\_configuration()
14. try:
15. if token is None:
16. wallet\_address = Wallet(configuration).get\_address()
17. eth\_balance = Wallet(configuration).get\_balance(wallet\_address)
18. click.echo('Balance on address %s is: %sETH' % (wallet\_address, eth\_balance))
19. except InvalidAddress:
20. click.echo('Invalid address or wallet does not exist!')
21. if \_\_name\_\_ == '\_\_main\_\_':
22. get\_balance()

在命令行中执行 “python3 get\_balance.py”，可以发现函数在查询以太坊网络后打印出 ETH 余额如下图：



图 9.2.2-1钱包余额显示

#### 9.2.3 ERC20标准实现与部署

正常情况下，支持ETH转移和余额查询已经具备钱包的基本功能了，但是由于以太坊是支持智能合约的，因此基于以太坊平台可以有更多的玩法，利用以太坊发行项目代币（token）就是其一。利用代币兑换ETH的方式进行项目初期融资，这种行为被称为ICO（Initial Coin Offering，首次币发行）。由于很多项目方只编写了白皮书，随便提交一些代码，就开始进行ICO公开募资，随着一些交易所的参与以及曝光，更多的民众会参与到这种代币的购买中，但很多项目最终的结局是不了了之，民众获得的token自然也就没有了价值，变成了毫无价值的空气币。

鉴于大量项目方利用ICO割韭菜、发空气币的行为存在，我国政府在2017年9月4日颁布法案，将该行为定性为非法。token能够风靡一时，ICO起到了巨大的作用，虽然ICO被禁止了，但token本身的一些技术特性，仍然值得研究。以太坊社区的开发者甚至推出了一系列通用的token标准。众多token标准中使用最广泛的就是ERC-20和ERC-721。

ERC-20和ERC-721代表了不同的token标准，ERC-721代表了非同质化token，也就是token彼此是唯一的，不能互相交换，以太坊平台中最成功的游戏CryptoKitties（加密猫）曾经把一只猫卖到1万多人民币，加密猫中的一只猫对应的实际上就是一个ERC-721代币。

ERC-20代表的是同质化token，相同token之间是完全等价，可以互换的，ICO发行的项目代币正是ERC-20标准的token。目前市面上的钱包都会支持ERC-20标准的token，对于我们来说，虽然ICO是违法的，但是在项目内部使用token是没有问题的，这个token更像是我们熟知的积分的升级版。

下面，我们来具体介绍一下ERC-20标准及其实现，具体可以参考官方网址：https://eips.ethereum.org/EIPS/eip-20。ERC-20的标准如下：

在上一章中，我们将ERC-20标准合约已经实现了。接下来可以去remix环境运行一下试试。可以先在 Remix部署一下，在 Remix 上部署合约需要一下几个步骤：

1. 切换到Deploy部署页面
2. 选择环境 – 这里可以使用 MetaMask 注入所连接的公链的环境，比如 MetaMask连接的是 Polygon 公链，那么 Remix 交互的公链就是Polygon
3. 选择要部署的合约
4. 点击 Deploy 进行部署，Remix 就会自动帮我们将合约部署到对应的链上

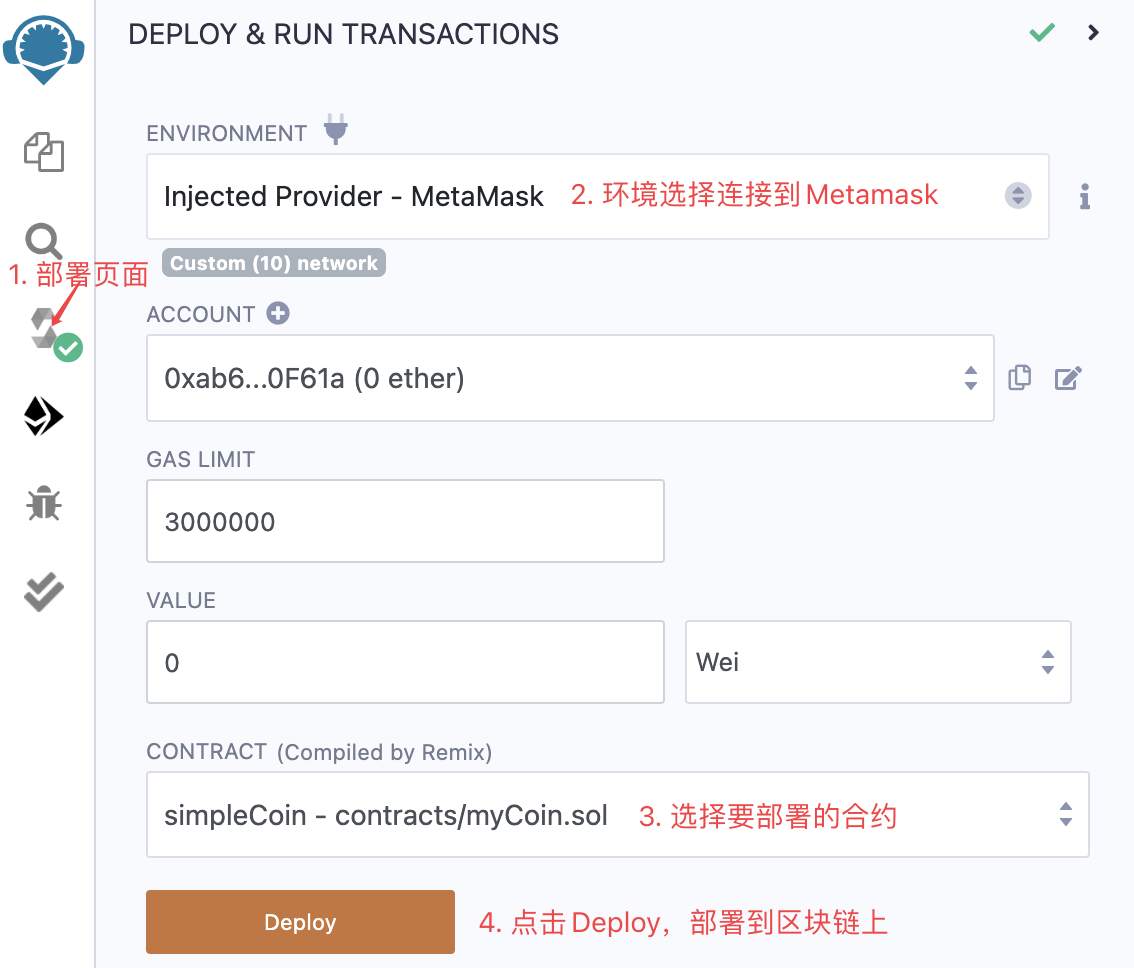


图 9.2.3-1 Remix部署合约到链上

部署后，我们可以尝试给自己的创建的账户发行一些token，然后顺便查看一下余额。

下面我们按步骤调用合约的函数，笔者会给自己发了5个“YKT” Coin



图 9.2.3-2 调用合约的 mint 和 balanceOf 函数

考虑后面需要根据合约实现更复杂的功能，我们需要把合约的ABI（Application Binary Interface，应用程序二进制接口）拷贝出来，并将其复制并保存到离线钱包主目录的erc20/abi.json 文件下：



图 9.2.3-3 获取合约的 ABI

#### 9.2.4 钱包如何支持Token转移

在前面的章节中，离线钱包实现了 ETH Coin的转移，Token的转移也是类似的，不同的是这里的转移实质上是智能合约里的函数调用，整体思路和之前ETH Coin的转移差不多，只不过Token转移主要是针对合约调用，下面，我们分步骤实现。

首先在根目录下新建一个 contract.py文件，在其中我们实现一个Contract 类来封装调用以太坊合约所需要的一些属性，同时针对合约增加一些常用的方法，如在离线钱包中增加持有的Token类别，列出钱包下的所持有 Token等。

在构造函数代码中，w3 是 Web3.py 对象，它代表一个连接到以太坊网络的客户端。eth.contract 方法用于获取指定合约地址上的合约对象。其中，address 参数指定了合约地址，abi 参数则指定了合约的接口描述（ABI），其中get\_abi\_json() 函数是用于读取本地erc20/abi.json文件的函数。调用 eth.contract 方法后返回的合约对象可以用来调用合约的函数、获取合约的信息等。

class Contract详细代码如下：

1. from infura import ( Infura, )
2. from utils import ( get\_abi\_json,)
3. class Contract:
4. """ERC20 代币的 Abstraction"""
5. def \_\_init\_\_(self, configuration, address):
6. self.conf = configuration
7. self.address = address
8. self.w3 = Infura().get\_web3()
9. *# 获取指定合约地址上的合约对象*
10. self.contract = self.w3.eth.contract(address=address, abi=get\_abi\_json())
11. self.contract\_decimals = self.contract.functions.decimals().call()
12. def add\_new\_contract(self, contract\_symbol, contract\_address):
13. """Add ERC20 token to the wallet"""
14. self.conf.add\_contract\_token(contract\_symbol, contract\_address)
15. def get\_balance(self, wallet\_address):
16. """Get wallet's ballance of self.contract"""
17. return self.contract.functions.balanceOf(wallet\_address).call() / (10 \*\* self.contract\_decimals)
18. def get\_decimals(self):
19. """返回“小数”位数"""
20. return self.contract\_decimals

定义好 Contract 类后，就可以向离线钱包内添加 Token 了，在 cli 文件夹下新建一个 add\_token.py文件，将 Token 信息添加到 configuration 中：

1. import click
2. import sys
3. sys.path.append("..")
4. from contract import ( Contract, )
5. from configuration import ( Configuration, )
6. @click.command()
7. @click.option('-c', '--contract\_address', default='', prompt='Contract address',
8. help='Contract address.')
9. @click.option('-s', '--symbol', default='', prompt='Token symbol',
10. help='Token symbol.')
11. def add\_token(contract\_address, symbol):
12. """Add new ERC20 contract."""
13. configuration = Configuration().load\_configuration()
14. try:
15. contract = Contract(configuration, contract\_address)
16. contract.add\_new\_contract(symbol, contract\_address)
17. click.echo('New coin was added! %s %s' % (symbol, contract\_address))
18. if \_\_name\_\_ == '\_\_main\_\_':
19. ­add\_token()

在命令行中调用­“python3 add\_token.py”，来将合约对应的 ERC-20 Token添加到钱包中：

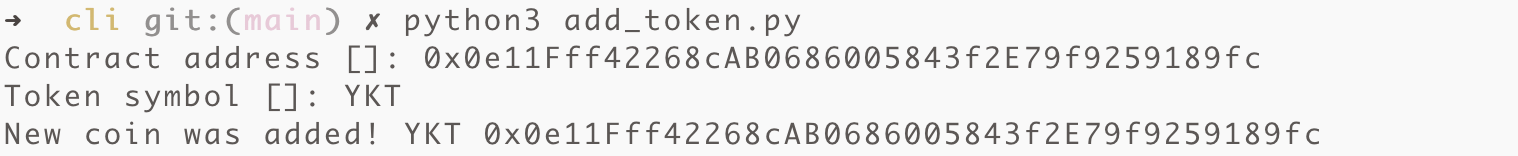


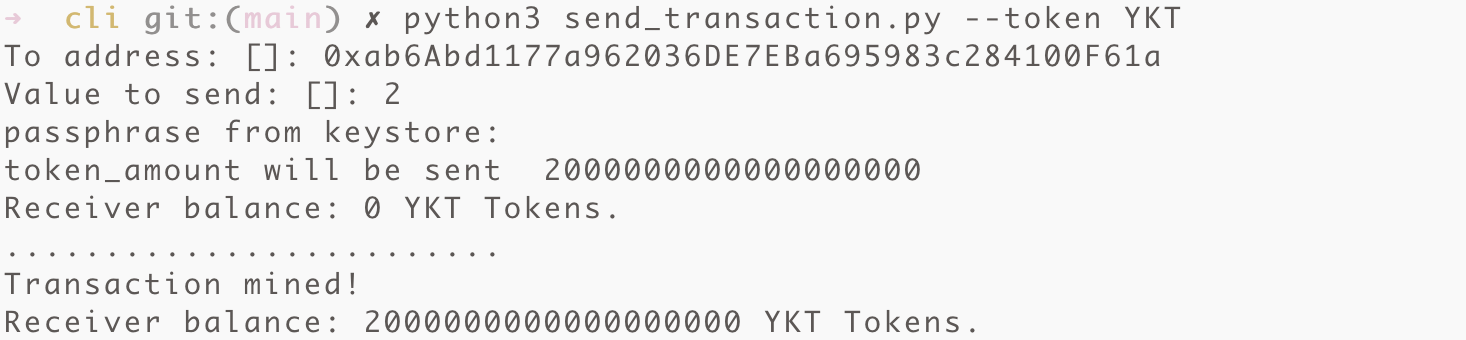
图 9- 离线钱包添加 ERC-20 Token

现在我们的离线钱包里有了 YKT 这种类型的 ERC-20 Token，再加上之前在Remix 里对离线钱包账户 mint出的 5 枚YKT代币，现在我们有了资本去进行真实的交易，下面就可以从钱包转出我们在合约里自定义的 ERC20 标准的代币，Amazing！

下面为了实现 ERC-20 Token的转账功能，我们再来对 send\_transaction.py稍加完善：增加一个 token 参数，来让 send\_transaction 函数支持 ERC-20 代币的转账，

1. @click.option('--token', default=None,
2. help='Token symbol.')
3. def send\_transaction(to, value, token):
4. try:
5. if token is None:
6. *# .... 发送 ETH 的交易内容*
7. else:  *# token is not None*
8. try:  *# check if token symbol is added to the wallet*
9. contract\_address = configuration.contracts[token]
10. except KeyError:
11. raise ERC20NotExistsException()
12. *# 构建一个 contract 对象，方便后续调用方法*
13. contract = Contract(configuration, contract\_address)
14. *# 获取代币精度，一般是 18 位*
15. erc20\_decimals = contract.get\_decimals()
16. *# 获取 web3.py 的客户端对象，用来调用链上合约内的方法*
17. contract\_w3 = contract.contract
18. *# 把 Token 数量改造成合约熟悉的格式（如 18 位）*
19. token\_amount = int(float(value) \* (10 \*\* erc20\_decimals))
20. *# 获取钱包余额，检查是否足够转账*
21. erc20\_balance = contract.get\_balance(wallet.get\_address())
22. if float(value) > erc20\_balance:
23. raise InsufficientERC20FundsException()
24. print('token\_amount will be sent ', token\_amount )
25. print(f"Receiver balance: {contract\_w3.functions.balanceOf(to\_address).call()} {token} Tokens.")
26. *# 构建 transaction 交易对象*
27. *# data：调用合约的 transfer 函数，参数分别是 address 和 amt*
28. raw\_txn = {
29. "from": wallet.get\_address(),
30. "gasPrice": web3.eth.gasPrice,
31. "gas": 500000,
32. "to": contract\_address,
33. "value": "0x0",
34. "data": contract\_w3.encodeABI('transfer', args=(to\_address, token\_amount)),
35. "nonce": web3.eth.getTransactionCount(wallet.get\_address())
36. }
37. *# 签署交易*
38. signed\_txn = web3.eth.account.signTransaction(raw\_txn, wallet.get\_account().privateKey)
39. *# 发送交易*
40. tx\_hash = web3.eth.sendRawTransaction(signed\_txn.rawTransaction)
41. *# 等待交易执行成功 ...*
42. while True:
43. tx\_receipt = web3.eth.getTransactionReceipt(tx\_hash)
44. if tx\_receipt is None:
45. print('.', end='', flush=True)
46. time.sleep(0.2)
47. else:
48. print("\nTransaction mined!")
49. print(f"Receiver balance: {contract\_w3.functions.balanceOf(to\_address).call()} {token} Tokens.")
50. break
51. except InsufficientFundsException:
52. click.echo('Insufficient ETH funds! Check balance on your address.')
53. except InsufficientERC20FundsException:
54. click.echo('Insufficient ERC20 token funds! Check balance on your address.')
55. except InvalidAddress:
56. click.echo('Invalid recipient(to) address!')
57. except InvalidValueException:
58. click.echo('Invalid value to send!')
59. except InvalidPasswordException:
60. click.echo('Incorrect password!')
61. except InfuraErrorException:
62. click.echo('Wallet is not connected to Ethereum network!')
63. except ERC20NotExistsException:
64. click.echo('This token is not added to the wallet!')
65. if \_\_name\_\_ == '\_\_main\_\_':
66. send\_transaction()

下面可以在命令行中执行发送 ERC-20 代币的函数：“python3 send\_transaction.py --token YKT”，输入 recipient address，value 和 passphrase，便会调用合约的 transfer 函数进行转账。



转账完成后，在MetaMask添加该代币，可以看到确实被转账了 2 个 YKT！



#### 9.2.5 钱包如何支持Token查询

写到这里，我们的钱包还差一个功能，就是查询Token的余额。Token余额查询与Token转移功能没有太大区别，只是不需要私钥就可以完成。

回顾前面实现的 class Contract，其中有一段获取合约内账户余额的函数，其功能便是调用合约中的balanceOf 函数，去查询对应地址的token持有量：

1. class Contract:
2. def get\_balance(self, wallet\_address):
3. """Get wallet's ballance of self.contract"""
4. return self.contract.functions.balanceOf(wallet\_address).call() / (10 \*\* self.contract\_decimals)

根据该函数，我们可以在命令行中进行余额查询的实现：在 cli 文件夹下对 get\_banlance.py文件进行稍许改动，增加 “—token”的命令行选项，让其除了支持查询 ETH，也能支持 ERC-20的查询：

1. @click.option('-t', '--token', default=None,
2. help='Token symbol.')
3. def get\_balance(token):
4. """获取账户余额."""
5. configuration = Configuration().load\_configuration()
6. wallet\_address = Wallet(configuration).get\_address()
7. try:
8. if token is None:  *# 查询 ETH 余额*
9. *# code for  ETH 余额查询 ...*
10. if token is not None:
11. try:  *# check if token is added to the wallet*
12. contract\_address = configuration.contracts[token]
13. except KeyError:
14. raise ERC20NotExistsException()
15. *# 构建 contract 对象来访问以太坊网络*
16. contract = Contract(configuration, contract\_address)
17. token\_balance = contract.get\_balance(wallet\_address)
18. *# token\_balance, address = get\_balance(configuration, token)*
19. click.echo('Balance on address %s is: %s%s' % (wallet\_address, token\_balance, token))
20. except InvalidAddress:
21. click.echo('Invalid address or wallet does not exist!')
22. except InfuraErrorException:
23. click.echo('Wallet is not connected to Ethereum network!')
24. except ERC20NotExistsException:
25. click.echo('This token is not added to the wallet!')
26. if \_\_name\_\_ == '\_\_main\_\_':
27. get\_balance()

下面在命令行中测试运行，初始时，合约给钱包账户空投了 5 个 token，而后转让出去了 2 个，现在还剩 3.0 个 YKT token，没问题！



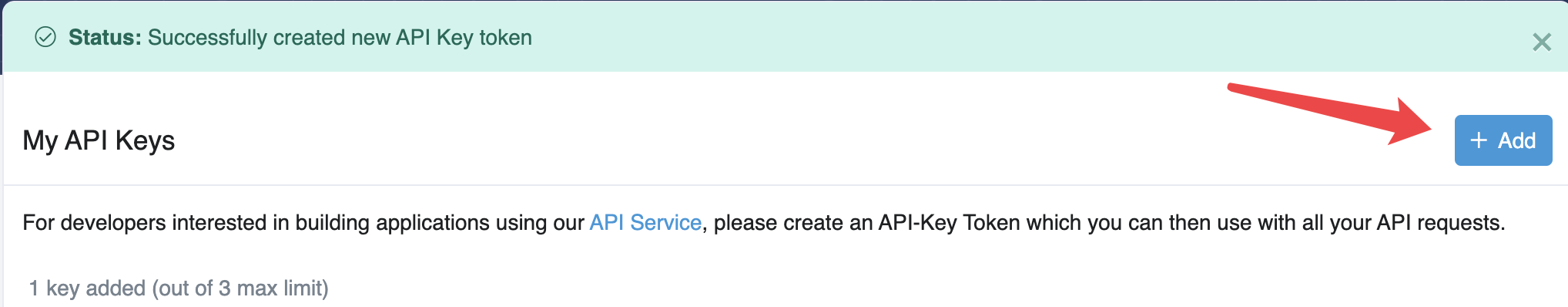
#### 9.2.6 交易明细查询

作为一个钱包，用户交易的明细也是需要给提供的，接下来我们来给钱包增加一个Token交易明细查询的功能。智能合约里并未记录Token交易的明细，想要使用 Python 语言查询交易明细，一般有 2 种方法：

1. 第一种是比较快速的方法，即使用etherscan提供的索引服务（indexing services），可以很快速地查询地址对应的交易，文档在：<https://docs.etherscan.io/>
2. 二是比较慢的方法，使用 Infura+Web3.py 去遍历每一个区块：使用 eth\_getBlockByNumber的JSON-RPC方法获取详细的区块信息（等同于使用web3.eth.getBlock(blocknumber)），使用web3.eth.getTransaction(txhash) 遍历交易哈希，并根据每一个交易中的发件人/收件人地址去过滤、判断此交易是不是我们需要的。

这里我们出于对快速完成需求的考虑，使用第一种方法来搞定需求。

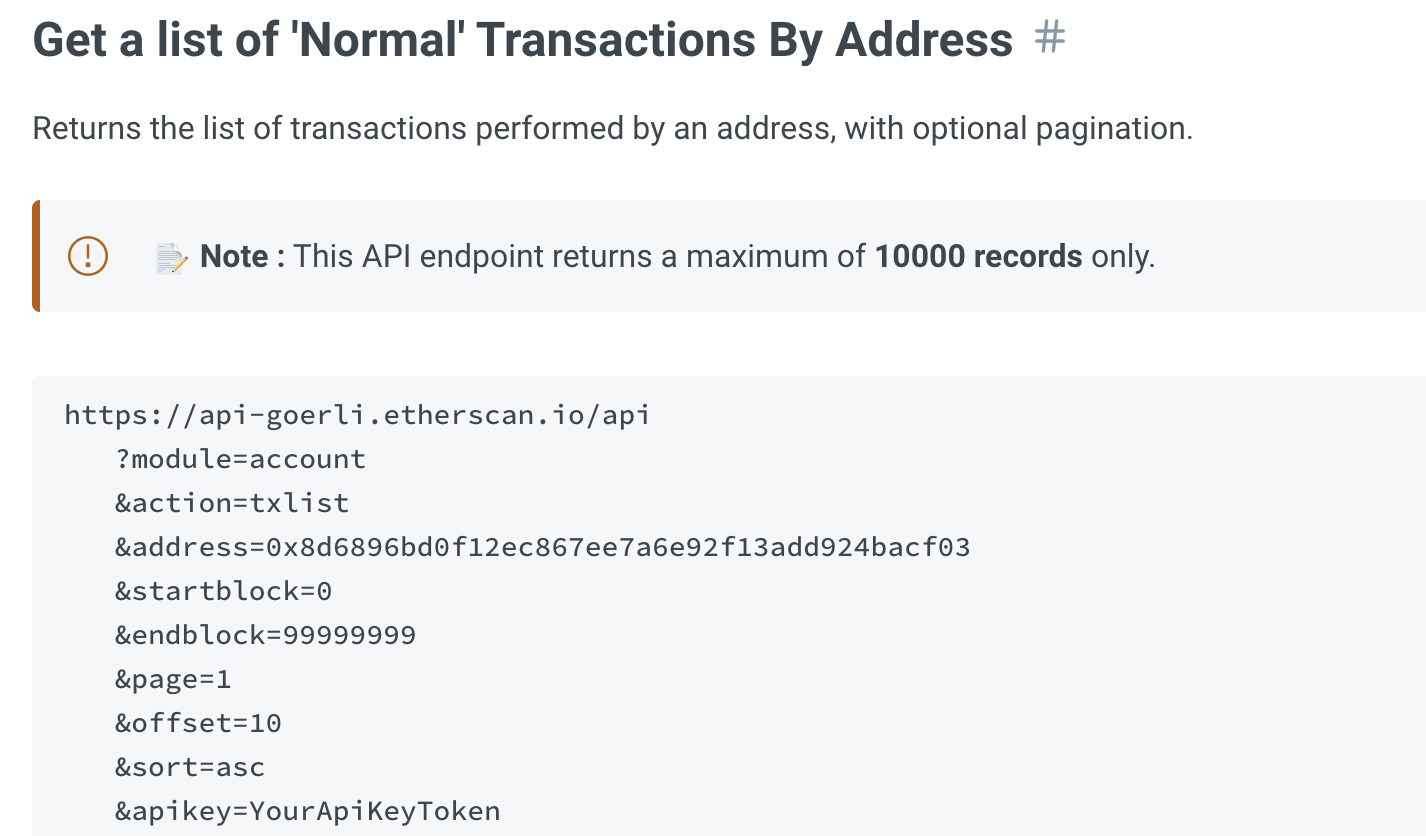
首先需要在 <https://etherscan.io/register> 使用邮箱进行注册，而后在 myapikey 页面 <https://etherscan.io/myapikey> 点击“Add”， 新建一个 apikey：



下面需要观察 etherscan 提供的接口，打开 <https://docs.etherscan.io/v/goerli-etherscan/api-endpoints/accounts#get-a-list-of-normal-transactions-by-address> ，在文档中可以找到这2个 API：

* Get a list of 'Normal' Transactions By Address
* Get a list of 'ERC20 - Token Transfer Events' by Address

这 2 个API接口的作用分别是查看 2 个地址之间交易和查看合约、地址之间的 ERC-20 Token 交易。



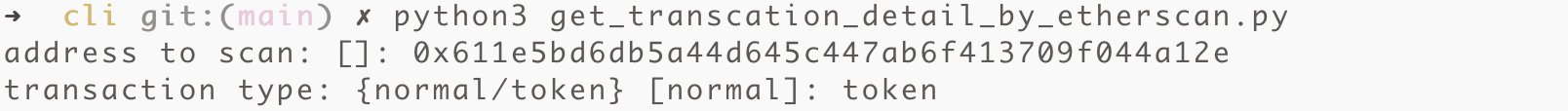
该 API 限制了返回最大 10000 条，不过足够我们使用了。下面尝试使用 apikey 构建HTTP交易请求。在cli 文件夹下新建一个get\_transcation\_detail.py 文件，作为命令行查询交易详情的调用函数，其内容主要包括：

1. 分别构建 normal（地址互相交易）和 token（ERC-20 交易）这 2 种交易类型
2. 使用 requests 库的 get 方法调用查询该接口，并输出 response返回的交易明细的json内容

具体代码如下：

1. import requests as rq
2. import pprint
3. @click.command()
4. @click.option('-d', '--address', default='', prompt='address to scan:',
5. help='input Ethereum address to list transactions.')
6. @click.option('-t', '--type\_tx', default='normal', prompt='transaction type: {normal/token}',
7. help='input Ethereum address to list transactions.')
8. def get\_transcation\_detail\_by\_etherscan(address, type\_tx):
9. """
10. [https://etherscan.io/register] [https://etherscan.io/myapikey]
11. :param type\_tx:
12. - normal 是 2 个地址之间的互相转账交易
13. - token 类型是合约和地址之间的交互交易
14. 分别对应着参数中的 txlist 或 tokentx
15. """
16. tx\_type\_map = {"normal": "txlist", "token": "tokentx"}
17. ETHERSCAN\_API\_KEY = '7KRJKR5DI9RD6YX73JF7278KM9U6AP2HUB'
18. base\_url = "https://api-goerli.etherscan.io/api\
19. ?module=account\
20. &action={}\
21. &address={}\
22. &startblock=0\
23. &endblock=99999999\
24. &page=1&offset=100&sort=asc\
25. &apikey={}\
26. ".format( tx\_type\_map[type\_tx], address, ETHERSCAN\_API\_KEY).replace(' ','')
27. print('base\_url', base\_url)
28. res = rq.get(base\_url)
29. data = res.json()
30. pprint.pprint(data["result"])
31. if \_\_name\_\_ == '\_\_main\_\_':
32. *#**0x611e5Bd6Db5a44D645C447aB6F413709f044A12e*
33. get\_transcation\_detail\_by\_etherscan()

尝试在命令行中调用“python3 get\_transcation\_detail\_by\_etherscan.py”，分别输入待查询的地址和交易类型（默认是“normal”，若要查询 ERC-20交易，则输入“token”），请求截图如下：



其中返回内容截选如下：

1. {'blockHash': '0xec4c2ebd5f3c027cc686a67fc758f566d6fe947afd100587acac01e7678f9fb4',
2. 'blockNumber': '8233484',
3. 'confirmations': '9164',
4. 'contractAddress': '0x0e11fff42268cab0686005843f2e79f9259189fc',
5. 'cumulativeGasUsed': '3966267',
6. 'from': '0x0000000000000000000000000000000000000000',
7. 'gas': '71936',
8. 'gasPrice': '2500000014',
9. 'gasUsed': '71936',
10. 'hash': '0xa78eba092734deaaa8bf37e251b40419d9cca768bbcaf2e5c7860e6167d4e60e',
11. 'input': 'deprecated',
12. 'nonce': '14',
13. 'timeStamp': '1672502388',
14. 'to': '0x611e5bd6db5a44d645c447ab6f413709f044a12e',
15. 'tokenDecimal': '18',
16. 'tokenName': 'YKToken',
17. 'tokenSymbol': 'YKT',
18. 'transactionIndex': '6',
19. 'value': '5000000000000000000'},

从交易明细种可以看到：

1. contractAddress为链上部署的合约地址
2. to 字段中存放的正是我们的离线钱包地址
3. value 是5000000000000000000即 5e18，正好是 5 个单位的token数

至此，我们便实现了 token 的交易明细查询。

#### 9.2.7事件订阅

在上一小节中，我们通过主动请求Etherscan 的 API ，响应获得了地址相关的交易明细，但是如果我们在链上部署了一个重要的合约，可能某些情况下希望能实时地接收到合约的调用和执行情况，而不是只能不断地主动发送请求，为此，本节我们需要实现对合约的事件订阅（Event subscription）功能。

首先观察合约中的事件 emit：

1. function transfer(address to, uint256 value) override public returns (bool) {
2. *// 转出用户余额充足*
3. require(value <= \_balances[msg.sender]);
4. require(to != address(0));
5. *// 调整账本*
6. \_balances[msg.sender] = \_balances[msg.sender] - value;
7. \_balances[to] = \_balances[to] + value;
8. emit Transfer(msg.sender, to, value);
9. return true;
10. }

在合约中，每次调用 transfer 函数，都会 emit 出一个 Transfer 事件，其中记录了调用者msg.sender、接收人to和金额 value。

假设我们对合约的 Transfer 事件进行监听，具体步骤实现如下：

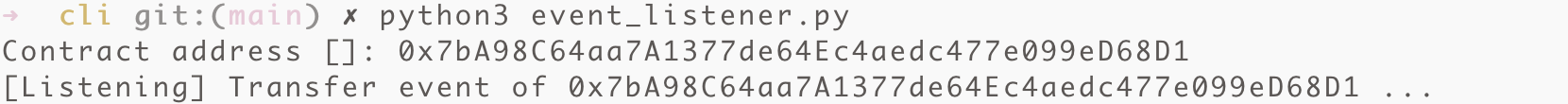
1. 通过 web3.py 的contract.events.Transfer.createFilter 函数，为合约中的 Transfer event创建一个过滤器
2. 引入asyncio作为 python 的异步框架，通过asyncio.get\_event\_loop获取当前事件循环，并每隔 2s 运行一次 event\_filter 函数，去监听最新区块里的合约事件
3. 如果合约事件命中，则打印该事件相关信息

我们在 cli 文件夹下创建event\_listener.py文件，文件详细代码如下：

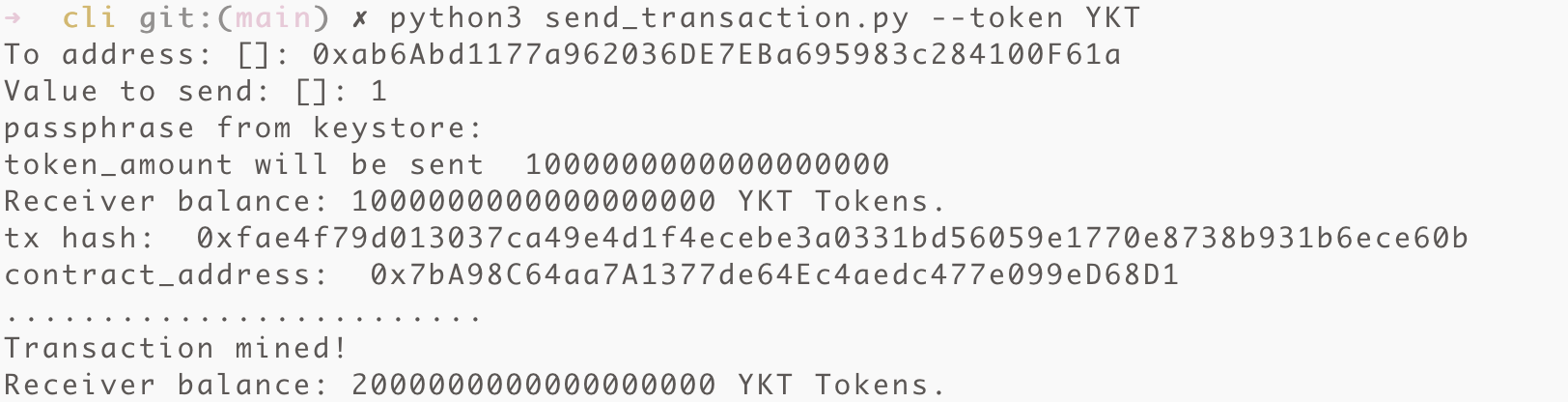
1. import click
2. import sys
3. sys.path.append("..")
4. from contract import ( Contract, )
5. from infura import ( Infura, )
6. from configuration import ( Configuration, )
7. import asyncio
8. from pprint import  pprint
9. def handle\_event(event):
10. """定义函数来处理事件并打印到控制台"""
11. pprint(event)
12. print("Contract address: ", event.address)
13. print('event.event', event.event)
14. print("transactionHash: ", event.transactionHash.hex())
15. async def log\_loop(event\_filter, poll\_interval):
16. """
17. 异步定义函数循环
18. 1. 这个循环设置一个事件过滤器, 寻找新的 event
19. 2. 这个循环在轮询间隔运行
20. """
21. while True:
22. for event\_ in event\_filter.get\_new\_entries():
23. handle\_event(event\_)
24. await asyncio.sleep(poll\_interval)
25. @click.command()
26. @click.option('-c', '--contract\_address', default='', prompt='Contract address',
27. help='Contract address.')
28. def event\_listener(contract\_address):
29. """
30. 为最新的区块（latest block）创建一个过滤器，并监听合约的 Transfer 事件是否被触发
31. 每隔 2s 运行一次 event\_filter 函数去监听最新区块里的合约事件
32. """
33. configuration = Configuration().load\_configuration()
34. w3 = Infura().get\_web3()
35. contract\_address = w3.toChecksumAddress(contract\_address)
36. contract = Contract(configuration, contract\_address)
37. *# print('contract.contract', contract.contract)*
38. *# 监听合约里的 mint 函数里 emit 的 Transfer event：*
39. event\_filter = contract.contract.events.Transfer.createFilter(fromBlock='latest')
40. loop = asyncio.get\_event\_loop()
41. try:
42. print("[Listening] Transfer event of {} ...".format(contract\_address))
43. loop.run\_until\_complete(
44. asyncio.gather(
45. log\_loop(event\_filter, 2)))
46. finally:
47. *# close loop to free up system resources*
48. loop.close()
49. if \_\_name\_\_ == "\_\_main\_\_":
50. event\_listener()

下面我们结合前面讲过的 send\_transaction.py 中的发送 ERC-20 函数，来测试合约中的 Transfer Event事件订阅。

首先调用“python3 event\_listener.py”，开始进行事件监听，因为此时没有任何合约的 transfer调用，所以控制台暂时没有任何信息输出：



新建一个命令行窗口，调用 “python3 send\_transaction.py --token YKT”来发送 ERC-20 代币，该命令会调用合约的 transfer 函数并 emit Transfer 事件：



等待 transaction 完成后，回到事件监听命令行窗口，发现该函数成功输出了我们的转账事件：



其中可以看到：event 的event事件是“Transfer”，也就是合约中定义被 emit 的事件，value 也正是我们向 0xa6 地址转账的数量，这说明上述代码成功实现了该事件订阅功能。

### 疑难解答

#### 1：Coin与Token交易的区别

答：Coin和Token这两个概念容易混淆，有点傻傻分不清楚。Coin是指基于区块链平台发行的本源币，比如比特币、以太币，一般来说Coin是需要矿工通过算力比拼获得的系统奖励，在区块链平台交易时需要消耗这样的本源币；Token是利用智能合约，基于区块链平台发行的项目代币，区块链早期很多项目都是通过发行自己的Token来募资（ICO），也有人会把Token翻译为通证。

#### 2：助记词与私钥的关系

答：了解区块链的人都清楚，私钥非常重要，掌握了私钥也就相当于掌握了资产所有权。私钥是一长串无序、无规律的字符，无法被人类识别，为此钱包设计了助记词。通过助记词可以推导出私钥，因此可以认为钱包助记词等同于私钥。

#### 3：钱包为什么可以支持不同token？

答：这与智能合约的原理有关，用以太坊举例来说，合约部署后会运行在节点的EVM中。当想要调用合约时，必须知道合约的ABI（函数接口）、合约地址，如果Token都遵循一定的标准（比如ERC20标准），那么对应的ABI接口也是相同的，因此钱包想要支持不同的Token，只要知道它对外公布的合约地址就可以了。

### 实训

#### 1：交易明细查询

由于智能合约里并未记录Token交易的明细，我们想要查询明细需要借助第4章介绍过的事件过滤功能，处理思路就是利用FilterLogs（ethclient）方法，查询区块内与合约地址有关的日志数据，并根据crypto.Keccak256Hash([]byte("Transfer(address,address,uint256)"))过滤交易事件，得到最终的Token交易明细。FilterLogs的原型如下：

func (ec \*Client) FilterLogs(ctx context.Context, q ethereum.FilterQuery) ([]types.Log, error)

具体处理步骤如下：

步骤01：定义包

package main  
  
import (  
 "context"  
 "fmt"  
 "math/big"  
 \_ "math/big"  
  
 "github.com/ethereum/go-ethereum"  
 "github.com/ethereum/go-ethereum/common"  
  
 "github.com/ethereum/go-ethereum/crypto"  
 "github.com/ethereum/go-ethereum/ethclient"  
)

步骤02：连接到geth

func main() {  
 //1. 连接到geth  
 conn, err := ethclient.Dial("http://localhost:8545")  
 if err != nil {  
 fmt.Println("failed to connet to geth", err)  
 return  
 }  
 defer conn.Close()  
 ...  
}

步骤03：查询所有logs

//2. 查询日志  
 // 先设置过滤条件，设为空  
 query := ethereum.FilterQuery{  
 Addresses: []common.Address{},  
 Topics: [][]common.Hash{{}},  
 }  
  
 //查询全部日志  
 logs, err := conn.FilterLogs(context.Background(), query)  
 if err != nil {  
 fmt.Println("failed to FilterLogs", err)  
 return  
 }

步骤04：过滤并解析logs

这一步关键是对logs的解析，读者可以先把每一条logs的数据（v.MarshalJSON()）打印分析一下，根据数据来决定如何处理。

//3.过滤并解析logs  
 cAddress := common.HexToAddress("0x4b6388442c218751604CC3aec7512efE850C7D15")  
 //过滤的事件hash值  
 topicHash := crypto.Keccak256Hash([]byte("Transfer(address,address,uint256)"))  
  
 for \_, v := range logs {  
 if cAddress == v.Address {  
 if len(v.Topics) == 3 {  
 if v.Topics[0] == topicHash {  
 //数据解析部分  
 from := v.Topics[1].Bytes()[len(v.Topics[1].Bytes())-20:]  
 to := v.Topics[2].Bytes()[len(v.Topics[2].Bytes())-20:]  
 val := big.NewInt(0)  
 val.SetBytes(v.Data)  
 fmt.Printf("BlockNumber : %d\n", v.BlockNumber)  
 fmt.Printf("from : 0x%x\n", from)  
 fmt.Printf("to : 0x%x\n", to)  
 fmt.Printf("from : %d\n\n", val.Int64())  
 }  
 }  
 }  
 }

步骤05：执行代码

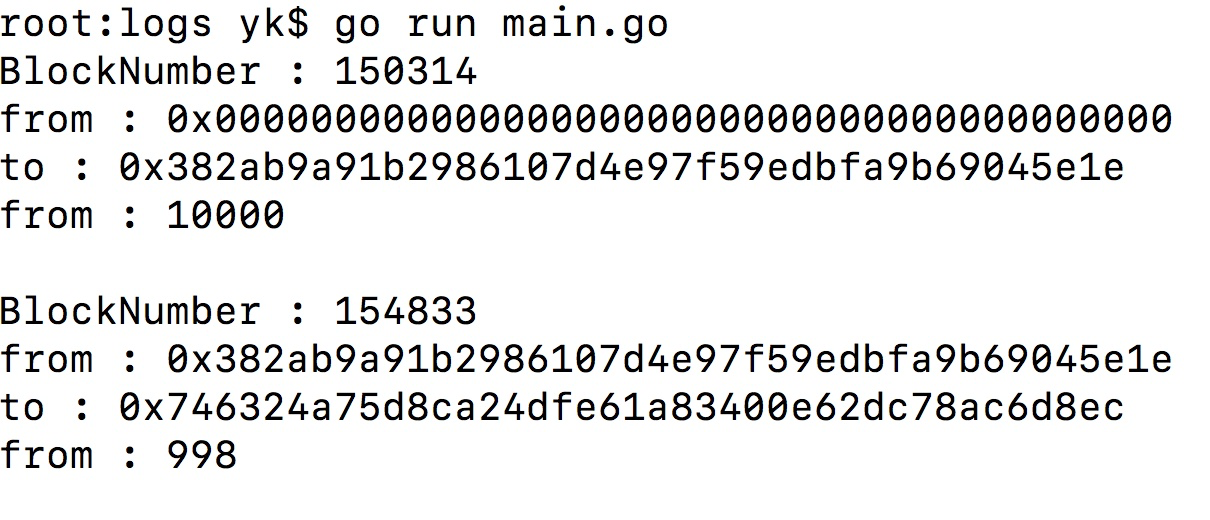


image-20200313154452052

从结果中，可以看到我们通过合约发起的两笔交易，第一次是通过mint发行给0x382ab9a91b2986107d4e97f59edbfa9b69045e1e了10000个token，第二次是0x382ab9a91b2986107d4e97f59edbfa9b69045e1e转账给0x746324a75d8ca24dfe61a83400e62dc78ac6d8ec了998个token，与我们之前的操作完全对得上。

### 本章总结

本章完整地介绍了区块链钱包项目从原理到实现，内容包括钱包的关键术语、核心原理、助记词和私钥保存、Coin交易以及Token交易等知识点。通过本章的学习，读者可以掌握区块链钱包的开发过程，对区块链技术和项目开发有了更深刻的认知。