Aichain

黄皮书

AICHAIN 基金会

2018年1月

目 录

摘要

AICHAIN的目标是为更复杂的AI应用提供一个公有区块链平台，能够让数据资源方、应用开发方、运行平台资源方和用户在这个区块链上自由发布和使用各自的资源和应用，以更低的技术门槛和成本将AI应用生态建设到区块链平台之上。

AICHAIN将构建一个良性的生态圈，激励更多人参与到智能化应用的开发与落地；推动人工智能在可信、可靠的环境中发展；让私人产生的数据，转化成给每个人的更精准化的服务。

AICHAIN需要实现的几个关键功能：

1. 基于比特币的区块链+非芯片化的挖矿算法：比特币HASH，LTC-scrypt，DASH-X11 都已经芯片化，ETH则是占用了大量显卡资源，高端显卡现在也在被大公司垄断。设计一个占用较少的显卡资源，但同时又不会被轻易芯片化的算法，目的是让区块链更安全，让更多的用户能够有计算能力、有权利去部署自己的应用。

2. 在比特币区块链的基础上增加智能应用部署和使用的功能，在交易本身的数据基础上，增加可定制化的应用或资源单元信息部署，支持可执行程序类、数据资源类、运行平台资源类等多种应用资源类型，具备极好的可扩展性

3. 将区块链和应用运行环境分离，采用外挂docker运行环境的方案，提供标准的、可升级的、可定制化的、可支持多种编程语言的APP运行环境。AICHAIN的节点程序会附带上一个公开标准和环境参数的docker IMG应用环境镜像，这个运行环境可以不断升级，甚至被用户改造定制，部署到自己的AICHAIN节点上。

4. 通过在区块链上部署智能应用信息单元，而不是完整的智能应用数据，使得应用数据的尺寸可以很大，不会影响区块链的运行效率。由应用的发布者提供应用数据资源的下载地址、或直接服务入口，只是把这些资源的描述信息发布在智能应用信息单元里。

5. 为应用提供方和使用方提供身份信息验证、区块链交易验证的功能，给应用开发者预留出足够的定制化空间，适合更复杂应用的开发及运行需求。

私钥、公钥和地址定义

2.1 公钥密码学

非对称加密需要两个（一对）密钥：公开密钥（publickey）和私有密钥（privatekey），用公钥对数据进行加密后，只有对应的私钥才能解密；反之如果私钥用于加密，则只有对应的公钥才能解密。通信双方无须交换密钥就可以建立保密通信。

在AICHAIN系统中沿用了比特币里的规范，私钥由32字节的随机数组成，通过私钥可以算出公钥，公钥经过一系列哈希及编码算法就得到了AICHAIN的地址。所以地址其实是公钥的另一种表现形式，可以理解为公钥的摘要。

2.2 相关算法

在私钥、公钥及地址的相关运算中，用到了基于secp256k1椭圆曲线乘法的签名算法、SHA-256、RIPEMD-160，和Base58编码。

2.2.1椭圆曲线签名算法

椭圆曲线在密码学中的使用是在1985年由Neal Koblitz和Victor Miller分别独立提出的。它的主要优势是在某些情况下它比其他的算法（比如RSA）使用更小的密钥但提供相当的或更高等级的安全性。

比特币使用了基于secp256k1椭圆曲线数学的公钥密码学算法。它包含私钥与公钥，私钥用于对交易进行签名，将签名与原始数据发送给整个虚拟币网络，公钥则用于整个网络中的节点对交易有效性进行验证。签名算法保证了交易是由拥有对应私钥的人所发出的。

2.2.2 哈希函数

SHA-256是一种哈希函数。

RIPEMD-160也是在生成地址时用到的一种哈希函数，其输出长度为20字节（160位）。比特币用它减少标识接收方的字节数。

2.2.3 Base58编码

可读性编码算法，类似古典密码学里的置换算法，理论上并不是密码学理论的核心内容。可读性编码算法不是为了保护数据的安全性，而是为了可读性。以二进制进行传输的信息是不具备可读性的，数字与字母组成的字符串才更容易被识别。可读性编码不改变信息内容，只改变信息内容的表现形式（部分编码算法还加入了容错校验功能，以保证传输过程中数据的准确性和完整性）。

Base64是常见的可读性编码算法，所谓Base64，即是说在编码过程中使用了64种字符：大写A到Z、小写a到z、数字0到9、“+” 和 “/”。

Base58是Bitcoin中使用的一种编码方式，主要用于产生Bitcoin的钱包地址。相比Base64，Base58不使用数字” 0”，字母大写” O”，字

母大写” I”，和字母小写” i”，以及” +” 和” /” 符号。

设计Base58主要的目的是：

1. 避免混淆。在某些字体下，数字0和字母大写O，以及字母大写I和字母小写l会非常相似。

2. 不使用” +” 和” /” 的原因是，非字母或数字的字符串难以作为账号的一部分被接受。

3. 没有标点符号，通常不会被从中间分行。

4. 使大部分的软件支持双击选择整个字符串。

AICHAIN也使用Base58算法来对公钥的Hash160及私钥进行编码，以生成地址及WIF (Wallet import Format)格式的私钥。

2.3 私钥与公钥

私钥其实是使用SHA-256生成的32字节 (256位) 的随机数，有效私钥的范围则取决于比特币使用的secp256k1 椭圆曲线数字签名标准。大小介于0x1 到0xFFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140之间的数几乎都是合法的私钥。

在私钥的前面加上版本号，后面添加压缩标志和附加校验码，（所谓附加校验码，就是对私钥经过2次SHA-256运算，取两次哈希结果的前四字节），然后再对其进行Base58编码，就可以得到我们常见的WIF (Wallet import Format)格式的私钥。

私钥经过椭圆曲线乘法运算，可以得到公钥。公钥是椭圆曲线上的点，并具有x和y坐标。公钥有两种形式：压缩的与非压缩的。早期比特币均使用非压缩公钥，现在大部分客户端默认使用压缩公钥。

由于数学原理，从私钥推算公钥是可行的，从公钥逆推私钥是不可能的。

初识比特币的人常有一种误解，认为比特币公钥就是地址，这是不正确的。从公钥到地址还要经过一些运算。

2.4 地址的生成

椭圆曲线算法生成的公钥信息比较长，压缩格式的有33字节，非压缩的则有65字节。地址是为了减少接收方所需标识的字节数。地址的生成步骤如下：

1. 生成私钥与公钥

2. 将公钥通过SHA256哈希算法处理得到32字节的哈希值

3. 后对得到的哈希值通过RIPEMD-160算法来得到20字节的哈希值 —— Hash160

4. 把版本号+Hash160组成的21字节数组进行

双次SHA256哈希运算，得到的哈希值的头4个字节作为校验和，放置21字节数组的末尾。

5. 对组成25位数组进行Base58编码，就得到地址。

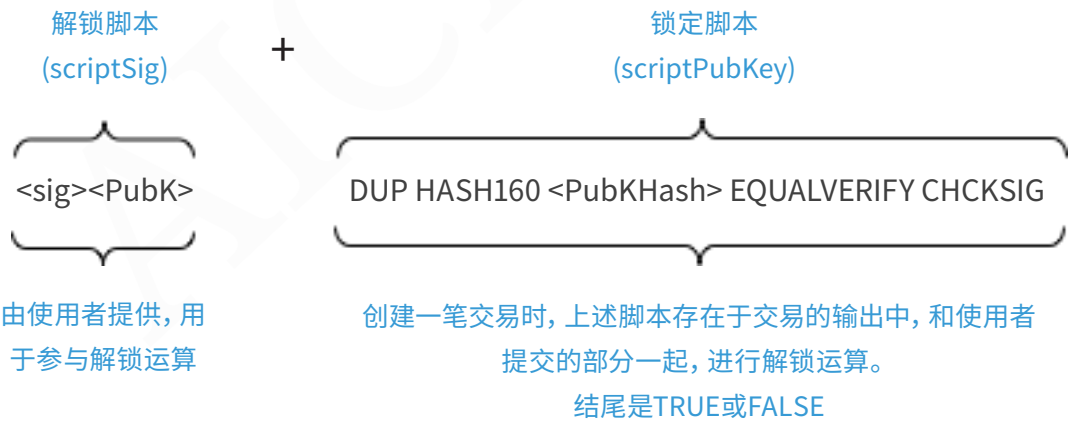
由于椭圆曲线乘法以及哈希函数的特性，我们可以从私钥推导出公钥，也可以从公钥推导出地址，而这个过程是不可逆的。也正因如此，在整个系统中，私钥是最关键的部分。私钥泄露也就意味着丢失了一切。

我们要花掉一个地址上的资产，需要构造一笔交易，同时使用这个地址对应的私钥签名。而如果我们要将资产转移到某个地址上，只需要转账给他公开的地址就行了。

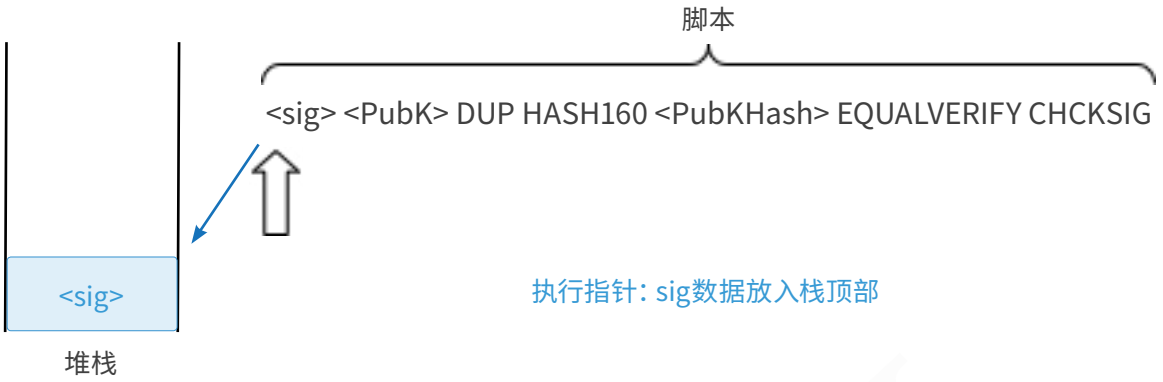
2.5 多种地址形态

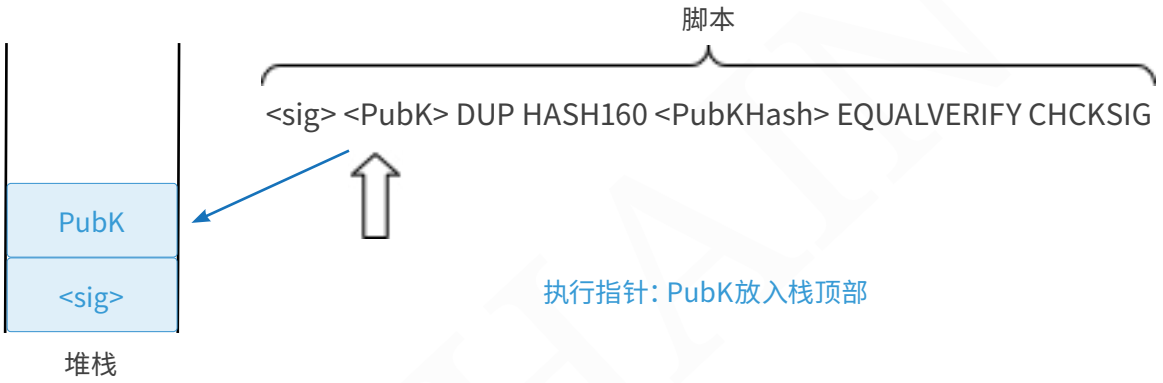
2.5.1 P2PKH（Pay to Public Key Hash）

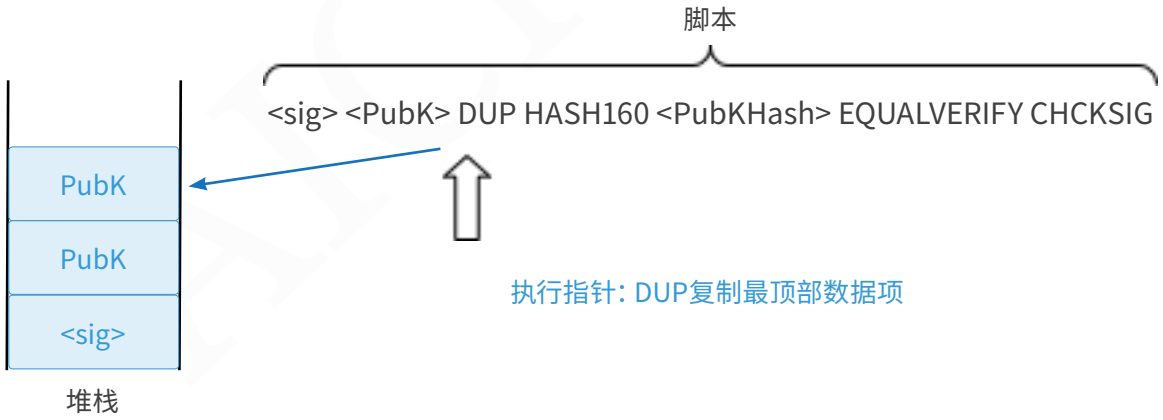
现在的比特币网络上，大部分交易都是以P2PKH的方式进行的，以下是P2PKH的锁定脚本与解锁脚本：

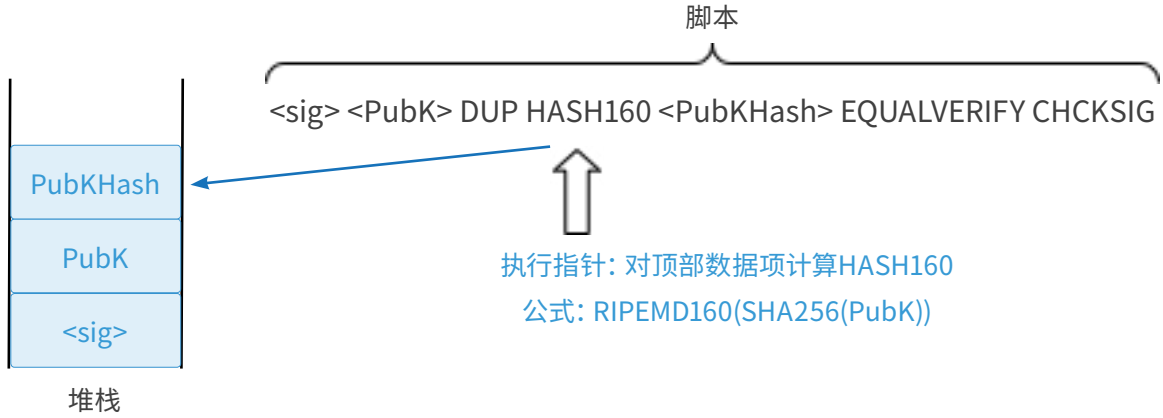


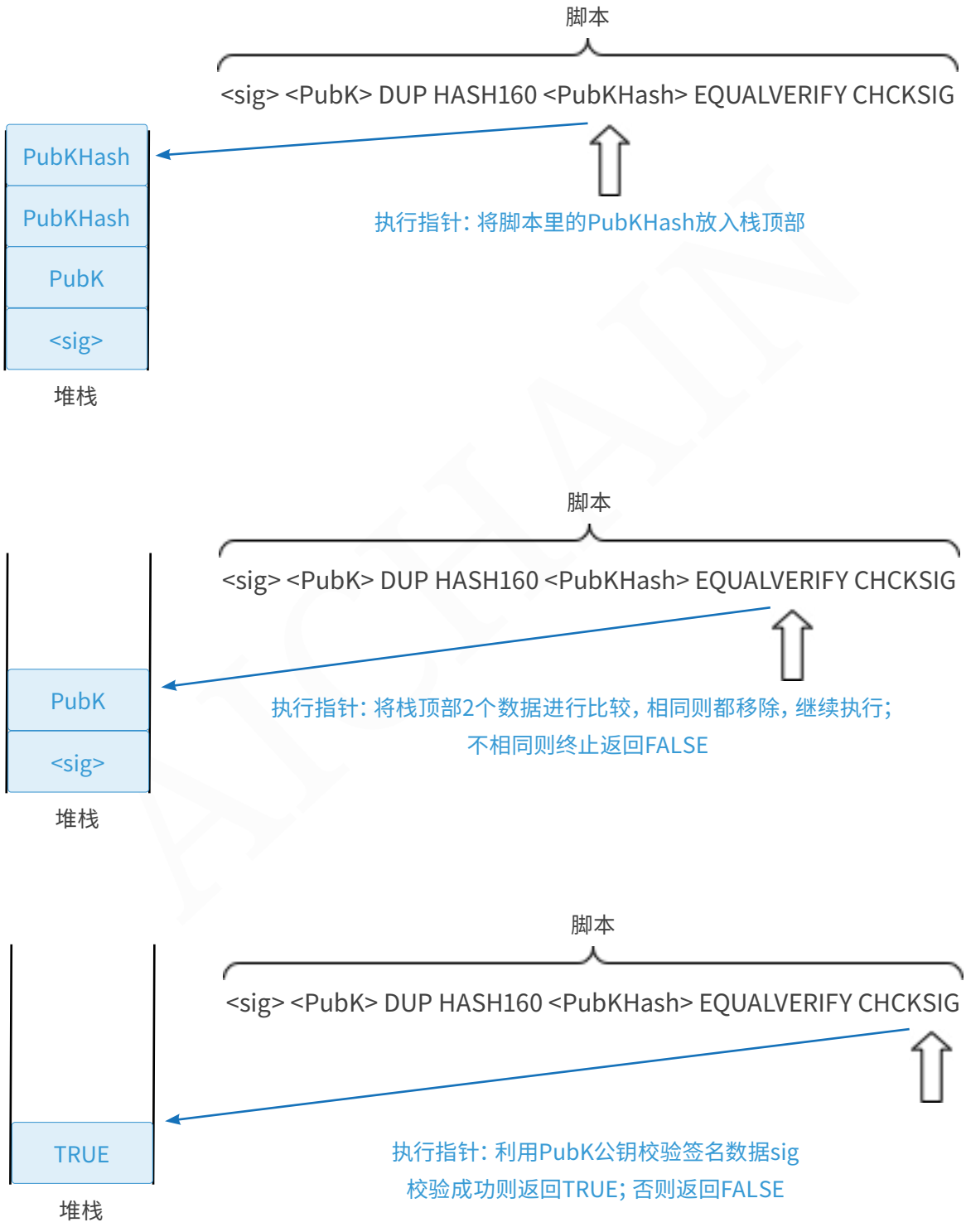
<sig>表示签名，<PubK>表示Public Key，具体操作步骤如下：


脚本
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHCKSIG
<sig>
堆栈
执行指针：sig数据放入栈顶部


脚本
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHCKSIG
PubK
<sig>
堆栈
执行指针：PubK放入栈顶部


脚本
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHCKSIG
PubK
PubK
<sig>
堆栈
执行指针：DUP复制最顶部数据项


脚本
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHCKSIG
PubKHash
PubK
<sig>
堆栈
执行指针：对顶部数据项计算HASH160
公式：RIPEMD160(SHA256(PubK))


脚本
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHCKSIG
PubKHash
PubKHash
PubK
<sig>
堆栈
执行指针：将脚本里的PubKHash放入栈顶部
脚本
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHCKSIG
PubK
<sig>
堆栈
执行指针：将栈顶部2个数据进行比较，相同则都移除，继续执行；
不相同则终止返回FALSE
脚本
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHCKSIG
TRUE
堆栈
执行指针：利用PubK公钥校验签名数据sig
校验成功则返回TRUE；否则返回FALSE

可以看出，主要验证两个验证，第一是Public Key是否能够转换成正确的地址，第二是Signature是否正确，也就是证明你是否是这个Public Key的主人，即你掌握着对应的私钥Private Key。

Signature签名内容主要是此交易摘要（也就是交易信息的Hash）与私钥进行运算获得的签名结果（一般是一个坐标数据r,s）。验证时将签名结果、交易摘要与公钥进行运算，最终获得一个验证签名的结果：TRUE 或者 FALSE。

2.5.2 P2PK (Pay to Public Key)

P2PK锁定版脚本形式如下：

<Public Key A> OP_CHECKSIG

用于解锁的脚本是一个简单签名：

<Signature from Private Key A>

经由交易验证软件确认的组合脚本为：

<Signature from Private Key A> <Public Key A> OP_CHECKSIG

根据上方的规则去运行就可以发现，此规则比P2PKH要简单的多，只有一步验证，少了上方的地址验证。其实，P2PKH被创建主要目的一方面为使地址更简短，使之更方便使用，核心内容还是P2PK的。

2.5.3 P2SH (Pay to Script Hash)

通用的M-N多重签名锁定脚本形式为：

M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG

其中，N是存档公钥总数，M是要求激活交易的最少公钥数。

例如，2-3多重签名条件：

2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG

上述锁定脚本可由含有签名和公钥的脚本予以解锁：

OP_0 <Signature B> <Signature C>

OP_0为占位符，没啥实际意义。

两个脚本组合将形成一个验证脚本：

OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG

P2SH是MS多重签名的简化版本，如果使用P2SH进行和上方相同的2-3多重签名条件，步骤如下：

锁定脚本：

2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG

对锁定脚本，首先采用SHA256哈希算法，随后对其运用RIPEMD160算法。20字节的脚本为：

8ac1d7a2fa204a16dc984fa81cfd-f86a2a4e1731

于是锁定脚本变为：

OP_HASH160 8ac1d7a2fa204a16dc-

984fa81cfdf86a2a4e1731 OP_EQUAL

此锁定脚本要比原先使用MS的锁定脚本要简短的多，当接收方要使用此交易中的UTXO时，需要提交解锁脚本（这里又可称为赎回脚本）：

<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>

与锁定脚本相结合：

<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG> OP_HASH160 8ac1d7a2fa204a16dc984fa81cfdf86a2a4e1731 OP_EQUAL

使用2.5.1章节中的运算规则，就能很明显的得知，验证过程分两步，首先验证的是接收方附上的赎回脚本是否符合发送方的锁定脚本，如果是，便执行该脚本，进行多重签名的验证。

P2SH的特点是，将制作脚本的责任给了接收方，好处是可以暂缓节点存储的压力。

多重签名地址是利用2个或2个以上的公钥，生成一个地址，在使用N个公钥生成多重签名地址的时候，可以约定签名验证时需要至少M个私钥对交易进行签名。满足以下条件：

N >= 2

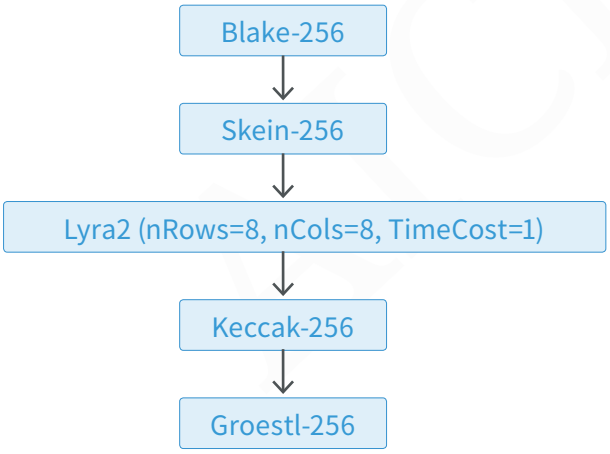
N >= M >= 1

挖矿算法

3.1 Lyra2DC (DC – Dynamic Complexity)

为了应对未来的ASIC（专用集成电路）的威胁，我们提出了一种基于NIST5的可以自定义参数的链式算法“Lyra2DC”（DC指的是动态复杂性）。此算法是基于vertcoin 以及monocoin使用的POW 算法Lyra2RE改进而来。“Lyra2DC” 是基可降低功耗和降低GPU温度的目的所专门设计的。Lyra2（此链式算法的主要部分）允许我们独立地调整内存使用和时间成本，提高芯片化设计的难度。

Lyra2DC该链式算法由5种不同的散列函数组成：Keccak, Skein, Groestl, Blake and Lyra2。



利用业界公认的哈希算法，我们能够常创造迄今为止最安全，最鲁棒，最持久的链式算法，该算法可以很容易在GPU上实现并且抗ASIC。因为未来的不可预见性，目前我们决定不实施“N元素”计划。但是当需要的时候，我们可以利用Lyra2DC进行灵活调整。

由于该算法的连锁性，基于GPU平台的算法运算将难以优化，这意味着我们可以从减少功耗和降低热量方面下手。采用Lyra2RE的Vertcoin相比其他硬币一直有着最高的$/Day/Normalized MH/s，但由于高能耗的缘故，Scrypt-N 硬币的散列率将下降。

根据Lyra2白皮书中所详细阐述的一样Lyra2在本质上是严格序列化的，以“密码海绵”作为核心。这就意味着该算法的并行化是基本不可能的，因为其每一步的计算都依赖于上一步的计算结果。

跟Scrypt-N不同的是，（该算法）的时间成本和内存成本是分开的，这使得我们可以独立地控制这两个参数。相比于对于Lyra2的开发，ASIC对于Scrypt-N的开发要容易得多，因为要增加Scrypt中的N元素，只需要将该算法多做几次迭代即可。在Lyra2下，虽然增加时间成本值涉及更多的迭代，而增加内存需求则意味着任何潜在的ASIC设备将在硬件上为每个线程设计更多的内存。在未来，如果ASIC曾针对Lyra2做过开发，当我们需要更高的内存要求时，这些ASIC将不能正常工作。

许多加密货币声称具有抗ASIC的算法，但其中许多仅仅是因为还没有针对他们的ASIC被制造出来。据传言称，针对X11的FPGA已经存在，并且Neoscrypt只使用更多轮的加密功能。相反地，Lyra2DC旨在从核心做到抗ASIC，在将来通过我们算法只改变参数而不是改变整个算法的能力为矿工减少困扰。这也将释放更多的时间去开发新功能，而不用担心每次出现ASIC威胁时就不断地实施新算法。

3.2 Password Hashing Schemes (PHS)

像前面讨论的一样，对一个PHS的基本要求就是要做到不可逆，所以从它的输出中恢复密码在计算上是不可行的。而且，一个好的PHS的输出应该与随机的比特串不可区分，从而防止攻击者根据感知模式丢弃部分密码空间。原则上，这些要求可以简单地通过使用安全的散列函数来实现，散列函数本身确保针对派生密码的最佳攻击方式是暴力破解（可能通过字典或者一些“常见”的密码模式进行辅助）。

所以，现在PHS所做的就是通过技术手段使得暴力破解代价达到最高。为了达到这个目的所采用的第一个策略就是输入不仅包括用户可记忆的密码本身，还包括一串称为salt的随机比特序列。这种随机变量的存在可以阻碍一些基于预先建立的通用密码表的攻击，例如，攻击者被迫从头开始为每个不同的salt创建一个新的表。salt可以看作是从密码得到的一大组可能的密钥的索引，并且不需要被记住或保密。

第二个策略是以有意地提高猜测密码的计算资源的消耗，比如处理的时间和内存的使用。这也会增加验证输入正确密码的合法用户的成本，这意味着算法需要进行配置，以便放置在目标平台上的负担几乎难以被人察觉。因此，PHS对于合法用户及其平台的计算成本上限最终取决于合法用户及其平台他们本身，对攻击者也是如此。例如，运行一个单个PHS实例的人类用户不会去考虑散列过程需要1秒的时间才能运行，并使用一小部分的机器空闲内存，比如，20MB。另一方面，考虑到密码的散列过程不能分解为一些可并行化的小进程，要想实现每秒1000个密码的吞吐量需要20GB的内存和1000个处理单元。

第三个策略是使用一些低并行性的设计，该策略在PHS同时涉及到处理时间和内存使用的时候显得尤其有用。理由如下：对一个具备p个处理核心的攻击者来说，为每个处理核心分配一个密码进行猜测运算或者并行处理一个猜测运算没有区别。所以处理速度都是提高了p倍：在两种场景下，总的密码猜测的吞吐量是相同的。然而，一个涉及可配置内存使用情况的序列化设计会给那些没有足够内存并行运行p个猜测的攻击者带来一些损失。例如，假设测试一个猜测涉及m个字节的内存和n个指令的执行。也假设攻击者的设备有100m的内存和1000个处理核心，并且每个核心每秒可以执行n条指令。在这个场景下，对于严格的顺序算法（每个核心上计算一个），每秒最多能测试100个猜测，另外的900个核心由于没有内存运行而保持空闲状态。

为了深入了解PHS解决方案所面临的挑战，在下文中，我们将讨论攻击者使用的平台的主要特征和现有的解决方案如何避免这些威胁。

攻击平台：

任何PHS所面临的危险性最高的威胁来自于那些受益于“经济规模”的平台，尤其是在有廉价的大规模并行处理硬件可以使用的时候。这些平台中最显著的例子就是图形处理单元（GPUs）和由FPGAs组成的定制硬件。

1 图形处理单元。随着对高清实时渲染的需求不断增加，传统的图像处理单元（GPUs）集成了大量的处理核心以加速它的并性能力。然而最近，GPUs从特定的平台进化为通用的计算设备，并开始支持标准化的语言，用来帮助它利用其计算能力（如CUDA，OpenCL）。因此，它们变得更加集中地使用在更多的目的，包括密码破解。

由于现在的GPUs在单个设备中包含上千个处理

核心，其在并行执行多线程的任务变得简单和便宜。例如，Nvidia Tesla K20X（顶级GPUs之一）共有2688个处理单元，工作频率为732MHz，以及6GB的共享DRAM，带宽为每秒250GB。其计算能力可以通过使用主机资源进一步扩展，即使这也有可能限制内存的吞吐量。假设这个GPU用来攻击一个运行时间在1s内且占用内存小于2.23MB的PHS，其很容易实现每秒2688个密码的测试。然而，随着内存的使用量增加，由于GPU的内存限制为6GB，这个数字会有所下降。例如，如果一个连续的PHS需要20MB的DRAM，那么可以同时使用的最大的核心数目变成300个，仅占总数的11%。

2 现场可编程门阵列（FPGAs）。FPGA是一个可编程逻辑模块的集合，并与存储元件连接在一起，形成一个可编程的高性能集成电路。由于其根据特殊的任务来配置，它们可以针对其目的进行高度优化（如使用流水线）。因此，只要在底层硬件中有足够的资源（即逻辑门和存储器），FPGA可能会产生比使用相同成本的通用CPU所能实现的成本效益更高的解决方案。与GPU相比，由于后者的能量消耗相当低，因此FPGA也或许会具有优势。如果以定制逻辑硬件（ASIC）的形式集成电路，则可以进一步降低FPGA的能耗。

下面介绍了最近一个使用FPGA进行密码破解的例子。作者使用RIVYERA S3-5000集群与128个FPGA对PBKDF2-SHA-512进行了比较，结果表明，在并行处理5,376个密码的架构中，每秒测试的密码数量为356,352个。有趣的是，作者注意到得到这些结果可能的一个原因是由于大部分底层SHA-2处理是使用设备的内存缓存来执行的（比DRAM快得多），PBKDF2算法的内存使用量很小。例如，对于需要20MB内存运行的PHS，所产生的吞吐量可能会低得多，尤其是考虑到所采用的FPGA可以具有高达64MB的DRAM，并且因此最多可以并行处理三个密码而不是5,376个。

有趣的是，即使是最先进的集群，比如较新的RIVYERA V7-2000T，在处理一个需要相似的内存使用量PHS时也会有麻烦。这个功能强大的集群除了每个FPGA提供的20 GB共享DRAM之外，还可以运行四个Xilinx Virtex-7 FPGA和多达128 GB的共享DRAM。尽管其功能更加强大，对于一个严格要求20 MB运行的PHS，原则上仍然不能对超过2600个密码进行并行测试。

3.3 Scrypt

下面两个章节我们简单介绍一下Scrypt算法和Lyra2算法。Scrypt被使用在Litecoin等加密货币中。相对于bitcoin所使用的SHA256的第一代POW而言，Scypt可以认为是第二代POW算法， 而我们所使用的Lyra2则是比Scrypt更加能够抵御ASIC攻击的第三代POW算法。通常，在文献中提供的密码的哈希化的解决方案有：PBKDF2, bcrypt和scrypt。但是在这三种解决方案中，只有scrypt采用了PHS方式，即平衡了内存利用和快速处理之间的矛盾，因此性能上可以与Lyra2相提并论。它的特性主要有以下几点。

scrypt的设计侧重于耦合内存和时间成本。为此，scrypt采用了“顺序存储器硬件”功能的概念：这是一种渐近地使用了几乎与需要的操作数一样多的内存的算法。使用并行的方式实现不能渐近地获得成本的显著降低。因此，如果在算法的常规操作中使用的操作的数量和存储器的数量都是（R），则无记忆攻击的复杂性（即，

存储器使用减少到（1））变成Ω（R2），其中R是系统参数。下面的步骤组成scrypt的操作（参见算法1）。首先，初始化pb-long存储块Bi。这是通过使用HMK-SHA-256的PBKDF2算法作为潜在的散列函数和一次迭代来完成的。然后，通过“顺序存储器硬件”ROMix功能来处理每个Bi（增量或并行）。基本上，ROMix通过对Bi进行迭代散列来初始化Rb-long元素的阵列M。然后随机访问M的R位置，在这个（严格顺序的）过程中更新内部状态变量X，以确定这些位置在内存中确实可用。ROMix采用的哈希函数称为BlockMix，它模拟具有任意（b-long）输入和输出长度的函数。

算法 1 Scrypt.

```
参数：  h      d BlockMix 's internal hash function output length
输入：pwd     d The password
输入：salt    d A random salt
输入：  k      d The key length
输入：  b      d The  block  size,  satisfying  b = 2r • h
输入：  R      d Cost parameter (memory usage and processing time)
输入：  p      d Parallelism parameter
输出：  K      d The password-derived key
(B0...Bp–1)  ←PBKDF2HM AC–SHA–256(pwd, salt, 1, p • b)
for i ← 0 to p – 1 do
    Bi  ←ROMix(Bi, R)
end for
    K  ←PBKDF2HMAC–SHA–256(pwd, B0 " B1 " ... " Bp–1, 1, k)
return K  d Outputs the k-long key
function ROMix(B, R)    d Sequential memory-hard  function
    X  ← B
    for i ← 0 to R – 1 do     d Initializes memory array M
          Vi  ← X   ; X ←BlockMix(X)
    end  for
    for i ← 0 to R – 1 do     d Reads random positions of M
          j ← Integerif y(X) mod R
          X ←BlockMix(X (+) Mj )
    end  for
    return  X
end function
function BlockMix(B)   d b-long in/output hash function
    Z ← B2r–1     d r = b/2h, where h = 512 for Salsa20/8
    for i ← 0 to 2r – 1 do
          Z ← Hash(Z (+) Bi)   ;   Yi ← Z
    end  for
    return (Y0, Y2, ..., Y2r–2, Y1, Y3, Y2r–1)
end function
```

3.4 Lyra2

就像任何PHS一样，Lyra2输入一个salt和一个密码，创建一个伪随机输出，这个输出可以被用作密码算法的密钥材料或作为认证字符串。在内部，该方案的内存被组织为一个矩阵，在整个密码散列过程中，该矩阵被期望保留在内存中：由于其单元以迭代的方式不停被读取和写入，因此若丢弃用于保存内存的单元会导致在访问时需要重新计算一次，直到这个点最终被修改结束。矩阵的构造和访问是使用下面的海绵吸收、挤压和双工操作（即它的内部状态从不重置为零）的有状态组合完成的，这样就确保了整个过程的有序性。此外，定义了初始化后矩阵单元被重复访问的次数。

算法 2 The Lyra2 算法.

```
参数： H      d 块大小为 b (单位 bits) 的海绵和潜在的置换 f
参数： Hρ     d Reduced-round sponge for use in the Setup and Wandering
phases
参数： ω      d Number of bits to be used in rotations (recommended: a multiple of
W )
输入： pwd    d The password
输入： salt   d A salt
输入： T      d Time cost, in number of iterations (T ″ 1)
输入： R      d Number of rows in the memory matrix
输入： C      d Number of columns in the memory matrix (recommended: C • ρ ″ ρmax)
输入： k      d The desired hashing output length, in bits
输出： K      d The password-derived k-long hash
1: d Bootstrapping phase: Initializes the sponge's state and local variables
2: d Byte representation of input parameters (others can be added)
3: params ← len(k) ″ len(pwd) ″ len(salt) ″ T ″ R ″ C
4: H.absorb(pad(pwd ″ salt ″ params))    d Padding rule: 10*1.
5: gap ← 1   ;   stp ← 1   ;   wnd ← 2   ;   sqrt ← 2      d Initializes visitation
step and window
6: prev0 ← 2   ;   row1 ← 1   ;   prev1 ← 0
7: d Setup phase: Initializes a (R × C) memory matrix, it's cells having b bits each
8: for (col ← 0 to C −1) do {M [0][C −1−col] ← Hρ.squeeze(b)} end for
9: for (col ← 0 to C −1) do {M [1][C −1−col] ← M [0][col] (+) Hρ.duplex(M [0][col], b)}
end for
10: for (col ← 0 to C −1) do {M [2][C −1−col] ← M [1][col] (+) Hρ.duplex(M [1][col], b)}
end for
11: for (row0 ← 3 to R − 1) do    d Filling Loop: initializes remainder rows
12:    d Columns Loop: M [row0] is initialized; M [row1] is updated
13:    for (col ← 0 to C − 1) do
14:    rand ← Hρ.duplex(M [row1][col] [+] M [prev0][col] [+] M [prev1][col], b)
15:    M [row0][C − 1 − col] ← M [prev0][col] (+) rand
16:    M [row1][col] ← M [row1][col] (+) rot(rand)  d rot(): right rotation by ω bits
```

```
17:     end  for
18:     prev0 ← row0   ;  prev1 ← row1   ;  row1 ← (row1 + stp) mod wnd
19:     if   (row1 = 0) then     d Window fully revisited
20:     d Doubles window and adjusts  step
21:     wnd ← 2 • wnd   ;    stp ← sqrt + gap   ;    gap ← -gap
22:     if (gap = -1) then {sqrt ← 2 • sqrt} end if    d Doubles sqrt every other
iteration
23:     end if
24: end for
25: d Wandering phase: Iteratively overwrites pseudorandom cells of the memory ma-
trix
26: d Visitation  Loop: 2R • T  rows revisited in pseudorandom fashion
27: for (wCount ← 0 to R • T - 1) do
28:     row0 ← lsw(rand) mod R    ;    row1 ← lsw(rot(rand)) mod R     d Picks pseudo-
random rows
29:     for (col ← 0 to C - 1) do  d Columns  Loop: updates M [row0,1]
30:     d Picks  pseudorandom columns
31:     col0 ← lsw(rot2(rand)) mod C    ;    col1 ← lsw(rot3(rand)) mod C
32:     rand ← Hρ.duplex(M [row0][col] [+] M [row1][col] [+] M [prev0][col0] [+] M [prev1]
[col1], b)
33:     M [row0][col] ← M [row0][col] (+) rand    d Updates ftrst pseudorandom row
34:     M [row1][col] ← M [row1][col] (+) rot(rand)    d Updates second pseudorandom
row
35:     end for      d End of Columns  Loop
36:     prev0 ← row0    ;     prev1 ← row1 d Next iteration revisits most recently up-
dated rows
37: end for   d End of Visitation Loop
38: d  Wrap-up  phase:  output computation
39: H.absorb(M [row0][0])    d Absorbs a ftnal column with full-round sponge
40: K ← H.squeeze(k)         d Squeezes k bits with full-round sponge
41: return K                 d  Provides k-long bitstring as output
```

用户可以根据目标平台的资源对Lyra2的执行时间进行微调。

在 UTXO 交易模型中扩展应用信息

4.1 应用部署原则

AICHAIN在UTXO交易模型基础上，在交易输出的VOUT数据之后扩展出一个空间，用于在区块链上存放智能应用的描述信息。依靠版本号来区分，允许这种交易部署智能应用到区块链上。

智能应用单元的信息需要包含：

1 应用类型：直接提供服务、提供可执行文件、运行平台资源或纯数据资源；

2 访问地址：直接提供服务的地址或可执行文件资源的获取地址；

3 应用描述：名称、版本、资费等信息。

4 所有者的公钥：用于生成智能应用的唯一地址。

为了节省区块链上的存储空间，只需要将一部分关键信息存放到区块链上。

使用部署应用时的交易ID，作为智能应用的位置信息。

通过转账AICHAIN Token（AIT）到这些智能应用约定的地址上，来获取使用这些智能应用的资格或权限。

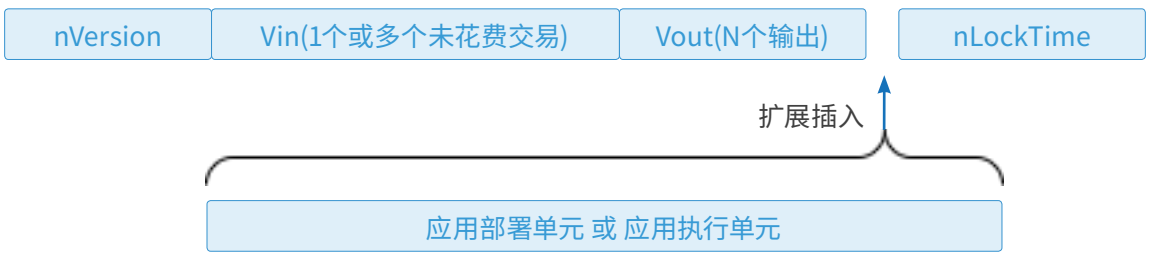
4.2 对交易数据的扩展

基于UTXO模型的交易数据主要由2个部分组成：输入和输出。

输入部分：由1个或多个未花费交易信息（unspent_tx_id, vout_n）和签名数据组成。

输出部分：由1个或多个转账目标地址（script_pubkey），以及输出到每个地址的AIT的数值。

在AICHAIN中对交易数据进行了扩展，在VOUT输出数据之后，增加了APP单元区域，可以用来部署1个APP应用单元；或者可以执行1个APP应用。



约定nVersion的最高位字节的第5个bit作为部署应用单元的标志，当这个bit位为1时，表示本交易包含至少1个APP应用部署单元。

条件为：(nVersion & 0x08000000) > 0 成立，表示交易内包含APP应用部署单元。

约定nVersion的最高位字节的第6个bit作为应用执行单元的标志，当这个bit位为1时，表示本交易包含至少1个APP应用执行单元。

条件为：(nVersion & 0x04000000) > 0 成立，表示交易内包含APP应用执行单元。

4.3 应用部署单元 ADU

APP应用部署单元约定名称为：ADU。

4.3.1 可自定义数据内容

允许第三方应用依靠发布ADU申请获得一个区块链上的数据存储区，用于存放[key,value]这种形式的数据，第三方应用可以自定义这些数据字段名称和内容含义，来满足应用系统的需求。

字段名	类型定义	含义
ADU _ PUBKEY	std::vector<unsigned char> pubkey	由应用发布者提供一个私钥对应的公钥
ADU _ TYPE	uint32	应用类型: 0:reserved 1:自定义数据 Others: reserved
ADU _ KEY _ NAME[0]	std::vector<char> keyname keyname[0] 表示名称长度	字段名称
ADU _ KEY _ VALUE[0]	std::vector<unsigned char> keyvalue keyvalue[0] 表示数据长度	字段值
…	…	
ADU _ KEY _ NAME[n]	std::vector<char> keyname keyname[0] 表示名称长度	字段名称
ADU _ KEY _ VALUE[n]	std::vector<unsigned char> keyvalue keyvalue[0] 表示数据长度	字段值

· ADU_PUBKEY: 由应用发布者使用自己的一个私钥来生成的公钥，第一个字节描述公钥数据长度，之后是公钥数据。

例如: 公钥16进制数据:

```
03cc149f66520680d85e18a01a8261a2746ee45fb5fb07ad13e9c316e1c955553d
```

第一个字节的数值表示了整个公钥数据的长度:

```
chHeader=pubkey _ bin[0];
if (chHeader == 2 || chHeader == 3)
    return 33; // total length = 33 bytes
if (chHeader == 4 || chHeader == 6 || chHeader == 7)
return 65; // total length = 65 bytes
```

· ADU_TYPE: 为ADU类型，目前暂定为1: 自定义数据; 其它数值保留以后再使用。

· 在区块链节点本地存储时，上述数据是依靠以下信息字段存储到数据库里的:

ADU_ADDR	ADU_PUBKEY	keyname	keyvalue
ADU _ ADDR _ user	ADU _ PUBKEY _ user	ADU _ KEY _ NAME[0]	ADU _ KEY _ VALUE[0]
		…	…
ADU _ ADDR _ user	ADU _ PUBKEY _ user	ADU _ KEY _ NAME[n]	ADU _ KEY _ VALUE[n]

[key:value]的数据都会对应到一个PUBKEY; 每个不同的PUBKEY可以对应有不同的[key:value]的数据。

每一个自定义数据单元ADU对应的所有数据内容，依靠ADU_ADDR标识出来。

4.3.2 应用案例: 标准应用单元信息

针对智能应用资源，可以按照4.3.1章节的规则，标准应用单元信息的ADU里定义以下几个[key:value]字段:

Keyname	KeyValue	含义
AIUNIT _ TYPE	uint32	参见第5.4章节
AIUNIT _ DATA _ HASH	uint32	应用和数据资源数据的HASH值
AIUNIT _ FEE	int64	使用应用的费用，单位0.00000001 AIT
AIUNIT _ NAME	std::vector<char> name name[0] 表示长度	AI应用单元的名称
AIUNIT _ INFO _ URL	std::vector<unsigned char> info info[0] 表示数据长度	当前定义为：info_url 用于获取应用名称等详细描述信息的地址。

· AIUNIT_TYPE：标准应用类型，用来表示这个标准应用单元是可执行文件、数据资源、可运行平台和直接服务。目前暂定义为这4种。

· AIUNIT_HASH：应用可执行数据的HASH值，用于校验应用的数据，防止被篡改。当应用数据发生改变时，需要发出AEU去更新这个字段数据到ADU单元对应的数据区内。

· AIUNIT_FEE：资费是使用这个AI应用资源时，需要支付多少AIT，单位是：0.00000001 AIT（最小单位）。

· AIUNIT_INFO_URL：应用的名称、功能介绍、应用资源获取的位置url、和使用说明等。这个url指定为HTTP GET请求返回约定格式的JSON信息体。定义如下：

```
{
   "version":"1.0.1",
   "name":"Dog or Cat?",
   "resource":"https://myapplication.com/DogCat.zip",
   "description":"This is an application to know it is a dog or cat from image",
   "runtime":[
      {
         "vm _ type":"docker",
         "version":"0.01"
      }
   ]
}
```

其中resource是应用资源的获取地址。这个部分在AI行业内部需要推行一个统一的数据接口规则，以确保兼容性。

4.4 应用执行单元 AEU

APP应用执行单元约定名称为：AEU。

4.4.1 自定义数据修改执行单元信息

允许用户通过发送交易到指定的ADU地址上，然后能够修改对应的自定义数据区的内容。

其中读取操作是不需要发送交易，可以直接读取。

而写入和删除操作是需要发送交易，才可以执行的。

· 读操作，不会产生真实交易，用户从本地节点接口直接获取数据：

字段名	类型定义	含义
AEU _ SIGNATURE	std::vector<unsigned char> signature signature[0] 表示长度	AEU数据单元签名
ADU _ txid	uint256	ADU在区块链上部署时所在的交易id
usrPubKey	std::vector<unsigned char> pubkey	由应用执行者（用户）提供一个私钥对应的公钥
OP _ CODE=1	uint8	0: reserved 1: read
ADU _ PUBKEY _ user	std::vector<unsigned char> pubkey pubkey[0] 表示长度	指定读取哪个公钥下的keyname 可以为空，表示读取所有的公钥下的keyname数值
ADU _ KEY _ NAME	std::vector<char> keyname keyname[0] 表示长度	字段名称 可以为空，表示读取对应公钥下的所有keyname的数值。

这个请求对应的是读取对应于：ADU_ADDR:usrPubKey:ADU_KEY_NAME 对应的keyvalue数据内容。

AEU_SIGNATURE：是上述AEU除了AEU_SIGNATURE字段的数据区的签名结果。使用usrPubKey进行签名验证。

· 删除操作，会产生真实交易，并修改区块链节点的数据库内容。只允许对执行者自己的公钥对应的数据进行操作：

字段名	类型定义	含义
AEU _ SIGNATURE	std::vector<unsigned char> signature signature[0] 表示长度	AEU数据单元签名
ADU _ txid	uint256	ADU在区块链上部署时所在的交易id
usrPubKey	std::vector<unsigned char> pubkey	由应用执行者（用户）提供一个私钥对应的公钥
OP _ CODE=2	uint8	2: delete
ADU _ KEY _ NAME[0]	std::vector<char> keyname keyname[0] 表示长度	字段名称
…	…	
ADU _ KEY _ NAME[n]	std::vector<char> keyname keyname[0] 表示长度	字段名称

· 写入操作，会产生真实交易，并修改区块链节点的数据库内容，只允许对执行者自己的公钥对应的数据进行操作：

字段名	类型定义	含义
AEU _ SIGNATURE	std::vector<unsigned char> signature signature[0] 表示长度	AEU数据单元签名
ADU _ txid	uint256	ADU在区块链上部署时所在的交易id
usrPubKey	std::vector<unsigned char> pubkey	由应用执行者（用户）提供一个私钥对应的公钥
OP _ CODE=3	uint8	3: write

ADU _ KEY _ NAME[0]	std::vector<char> keyname keyname[0] 表示长度	字段名称
ADU _ KEY _ VALUE[0]	std::vector<unsigned char> key _ value key _ value[0] 表示长度	字段值
…	…	
ADU _ KEY _ NAME[n]	std::vector<char> keyname keyname[0] 表示长度	字段名称
ADU _ KEY _ VALUE[n]	std::vector<unsigned char> key _ value key _ value[0] 表示长度	

4.4.2 应用案例：标准应用执行单元信息

在用户发起使用标准智能应用时，发出的交易包含AEU，内部对自定义数据区是不需要做操作。对应到写入操作AEU单元中的[keyname:keyvalue]为空。（以后也许会有第三方应用有需求让用户发起使用时，提交一些数据，可以通过AEU单元中的[keyname:keyvalue]来提交）。

另外在AEU中的usrPubKey将用来区分用户身份。在调用应用提供者提供的服务时，会要求提供这个公钥来验证身份；同时可以利用这个公钥和usrPrivKey私钥来和应用提供方进行加密数据通讯。

应用部署和执行

5.1 ADU 位置标识

当需要使用某个APP应用时，需要首先从区块链中定位到APP应用单元的位置。基于UTXO模型，当APP应用单元部署在交易数据中时，其唯一的标识就是部署这个应用时的交易ID：

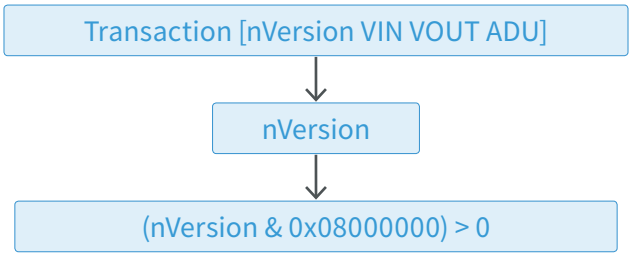
字段名	类型	含义
ADU _ txid	**uint256**	用于部署APP应用单元的交易ID

当某一个用户发起想获取对应的APP应用时，可以利用ADU_txid在区块链中找到对应的交易数据，然后提取对应的交易数据里包含的ADU单元信息。

5.2 ADU 单元的 P2SH 地址

当需要使用某个APP应用时，需要向这个APP应用的地址转账AIT。为了充分利用区块链交易来记录APP应用单元被使用的历史记录，需要通过一种方式保证APP应用对应地址的唯一性，所有和这个APP应用地址的交易都是可以被单独审计的。

APP应用在部署时，需要由创建者发起一个交易，才能将ADU信息单元放入到交易数据中。



创建这种交易时，ADU单元信息会包含一个公钥字段： ADU_PUBKEY，由发起部署者提供这个公钥。发起部署者拥有对应的私钥：ADU_PRIVKEY。

创建完原始交易后，就可以知道交易ID了：

字段名	类型	含义
tx _ hash	uint256	部署应用时创建的交易ID

这个交易ID是交易本身的HASH，这个数值作为私钥参与到ADU地址生成运算中：

Tx交易ID，256bits的数据作为私钥：PrivKey_TX

通过这个私钥生成公钥：PubKey_TX

利用这个公钥和和ADU_PUBKEY来生成一个M/N（M和N都是2）的多重签名的P2SH类型的地址：ADU_ADDR。这个地址就作为对应于ADU应用单元的地址。

掌握ADU_PRIVKEY私钥的人，可以控制ADU_ADDR上的AIT资产，在进行转账时，只需要使用ADU_PRIVKEY和PrivKey_TX，对转账交易进行多重签名即可完成。

这种定义方式能够保证每个APP应用地址的唯一性，且和区块链位置信息绑定。

5.3 应用的运行环境

AICHAIN节点配备一个docker运行环境+标准的docker IMG作为基础运行平台，在IMG中预先内置好多种编程语言运行环境，将其中可支持的编程语言，版本等都作为统一的规范，公开发布；包括docker IMG启动后自动下载APP和执行的入口及规则。这样开发者可以在这个docker环境下开发调试自己的应用；而这个docker运行环境也能够随着AICHAIN节点程序升级，一起更新，不断完善运行环境。

同时为了方便以后的扩展以及保留应用发布者更自由的定制空间，ADU里的应用类型还可以包含直接提供服务的入口，而不需要运行环境来支撑。

5.4 智能应用的类型

5.4.1 提供可执行文件

约定这个应用类型定义：AIUNIT_TYPE = 1

约定为适用于docker IMG环境下的可执行文件包，从应用开发者角度来说这种可执行文件包一般是包含：应用部署+应用执行 2个部分。

约定规则是：

1） 可执行文件包封装格式为zip

2） 约定应用类型为：webapp，终端最终使用浏览器来展现应用。

3） 在zip封装包内根目录下，提供ADU_start.sh，作为统一的运行入口

这样应用开发者可以依据AICHAIN发布的docker IMG环境，定制开发这个可执行文件包内的ADU_start.sh脚本，来完成应用的部署和启动运行过程。有着足够的灵活性和定制空间。

5.4.2 提供直接服务入口

约定这个应用类型定义：AIUNIT_TYPE = 2

约定直接服务入口为：WEBAPP模式，使用浏览器展现

这种应用不需要下载可执行文件数据，ADU详细信息中的resource对应的url地址是直接提供服务的入口。

5.4.3 提供运行平台资源

约定这个应用类型定义：AIUNIT_TYPE = 3

允许具有个性化定制的应用运行环境资源的服务商，将自己的应用运行平台资源发布到区块链中。

约定用户发出交易使用这种资源时，应用提供方需要分配给用户一个应用运行平台，允许用户在运行平台上输入要运行的应用资源resource地址。

5.4.4 数据资源

对于数据类的资源约定AIUNIT_TYPE = 4;

允许具有带标记的数据资源的提供商，将数据资源描述发布到区块链中，提供给需要用数据做深度学习的人使用。

约定用户发出交易使用这种资源时，应用提供方需要提供给用户一个数据资源下载地址（resource地址），允许用户下载数据资源。这种下载类的数据资源，提供方也是可以验证使用者身份的。

5.4.5 扩展其它类型资源

约定AIUNIT_TYPE当前最大值不超过0x0000ffff，前2字节预留以后使用。

约定之后其它资源类型的扩展，从AIUNIT_TYPE = 5 开始，到0x0000ffff。

5.5 使用 ADU 应用

5.5.1 包含有AEU单元的交易数据

当需要使用某个ADU应用时，需要向这个ADU_ADDR地址转账约定数量AIT; 并且在这个交易中包含AEU单元信息，通过以下ADU_tx_id信息来指定要使用的ADU应用。

这是一种标准的转账交易数据，为了标识交易内部包含AEU单元，约定nVersion的最高位字节的第6个bit作为标志，当这个bit位为1时，表示本交易包含AEU单元。

对于上述的使用APP应用的特殊交易，AICHAIN的节点会在接收到后，先用ADU_tx_id信息提取部署APP应用的交易数据，校验转账交易中，输出给APP应用的地址是否是正确的（按照5.2章节的规则）。然后提取APP应用单元信息，校验转账交易中，输出给APP应用地址的AIT数量是否满足ADU应用单元里约定的费用。

通过上述校验检查后，AICHAIN的节点，会执行2个动作: 将转账交易广播到网络中 和 运行APP应用（用户事后再运行这个APP应用也是可以的）。

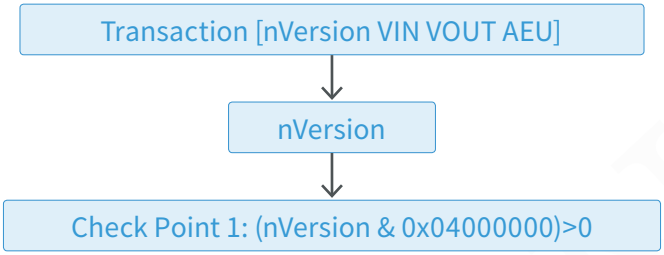
5.5.2 交易数据校验

当需要使用某个ADU应用时，需要发起交易向这个ADU_ADDR地址转账约定数量的AIT，包含AEC执行单元。

对于这种交易数据的校验点有：

1 交易的版本号检查

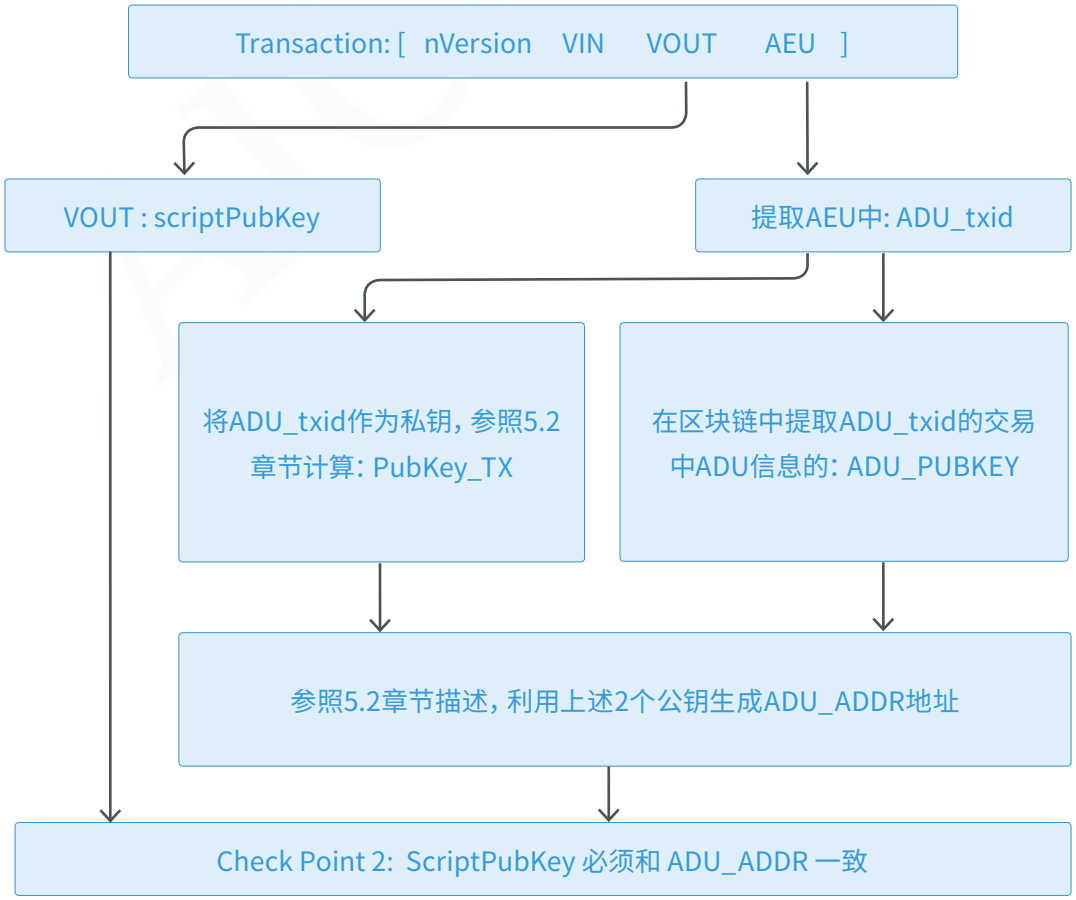
条件为：(nVersion & 0x04000000) > 0 必须成立。



2 交易的输出地址和AEC执行单元中指定的ADU的地址是否一致：

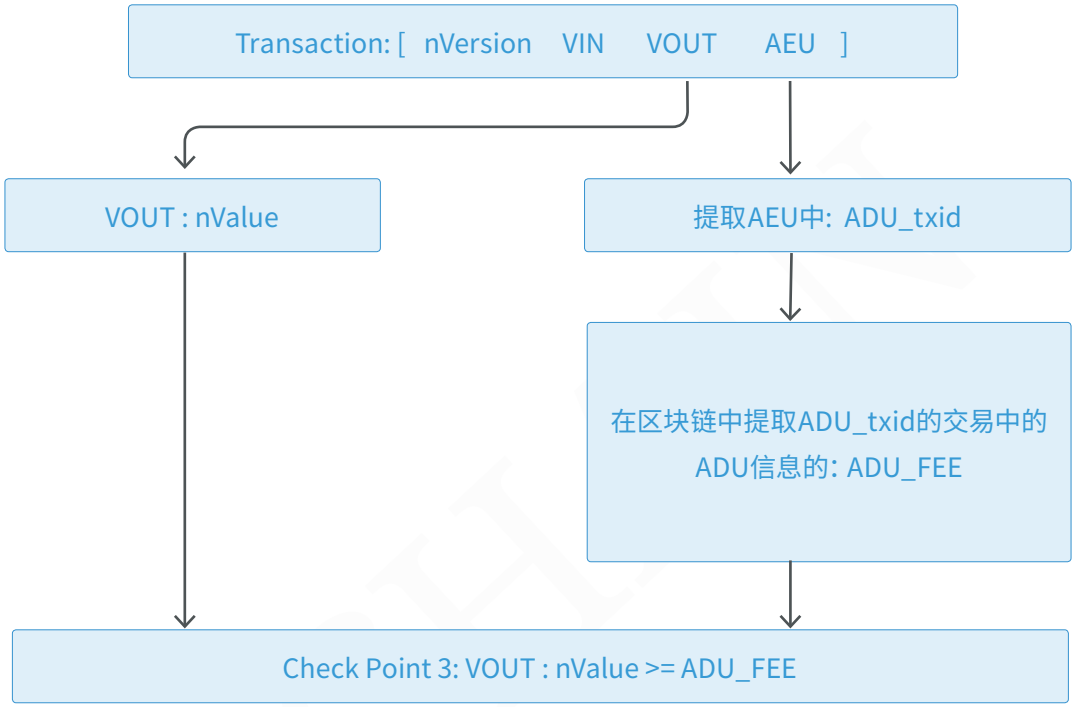
利用AEC执行单元里的ADU_txid信息可以在区块链上提取出ADU信息数据，参照5.2章节计算出ADU_ADDR地址，和交易的vout中的输出地址进行对比，必须一致。

当交易有多个VOUT单元时，需要匹配每一个VOUT的输出地址，有一个输出地址和ADU_ADDR地址匹配上就表示一致，同时需要记录下来这个VOUT对应的转账AIT数量。



3. ADU地址在本次交易中接收到的AIT数量是否足够:

在第2步的基础上，检查vout中的ADU_ADDR地址接收到的AIT数量，必须大于或等于ADU单元信息中的ADU_FEE数值。



在交易签名校验通过的基础上，上述3个校验点必须同时满足，才能判定为合法交易，否则AICHAIN节点将拒绝这笔交易，保证AEC执行信息与转账交易信息的一致性。

5.5.3 应用提供方校验使用者身份

当需要使用应用时，可以给应用服务方提供一种手段，校验使用者身份信息。约定利用ADU单元里的ADU_PUBKEY和使用者自己的私钥: USER_PRIVKEY，采用ECC非对称加密算法，对应用服务方下发的随机密码进行解密，输入正确的随机密码后，开始进入使用应用服务。

约定: 在使用者发起使用应用时，通过浏览器访问webapp入口时，必须传递自己的公钥。

约定: 使用者提供的公钥，必须是发起使用ADU应用交易时，所使用的签名用的私钥对应的公钥。这样应用提供方有能力校验AIT支付信息。

例如: https://myapplication.com/index.html?usrPubKey=

03cc149f66520680d85e18a01a8261a2746ee45fb5fb07ad13e9c316e1c955553d

字段名	类型	含义
usrPubKey	Hexadecimal string	the public key data corresponding to the user's private key

应用服务方利用上述信息，就可以生成随机访问密码，加密后传递给使用者，通过接口调用解密，或者使用离线的解密工具解密密码后，提交进入正常的应用使用。

应用方加密：使用USER_PUBKEY 和 ADU_PRIVKEY

使用方解密：使用ADU_PUBKEY 和 USER_PRIVKEY

应用提供方有足够的发挥空间可以自行定制身份验证手段。

5.6 运行流程

AICHAIN约定了4种角色：数据提供方、应用提供方、运行平台资源提供方、资源消费方。前面三方都是不同类型的资源；资源消费者是使用这些资源的用户。消费者可以是普通个人，也可以是正在开发AI应用的公司（需要大量数据用于机器学习）。

定义以下4个公司（或个人）：

A：是专门做图片标记的公司，期望提供AI所需要的数据来赚钱。

B：是专门开发AI应用的公司，期望依靠开发出来的AI应用赚钱。

C：是具有很多显卡服务器，具有tensorflow和caffe平台运行环境的公司，期望靠出租这些运行平台资源赚钱。

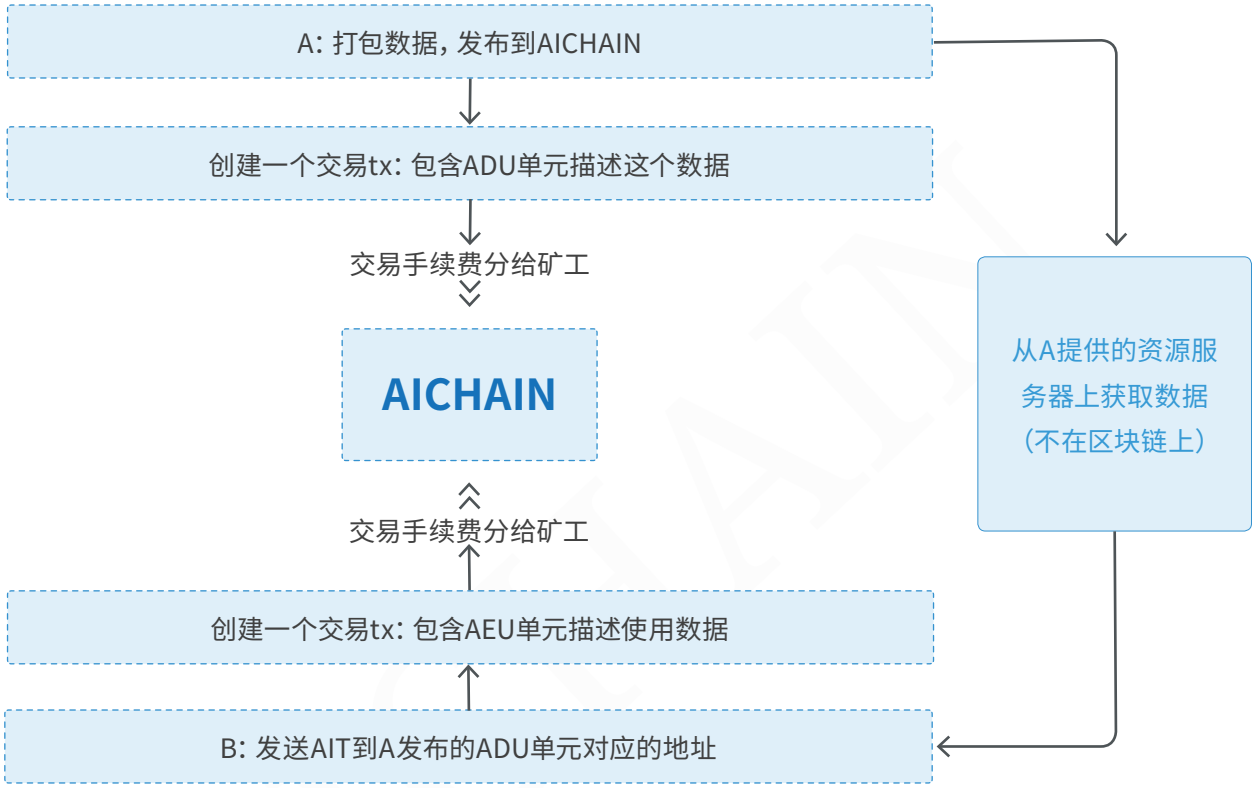
D：是一个普通的用户，他手里有一堆猫和狗的图片，期望能够有工具帮他分类存储。

5.6.1 开发者使用数据资源

开发者需要数据完成机器学习，开发AI应用

A: 是数据提供方

B: 是资源消费方



A: 有猫和狗的图片，以及每个图片的标记结果（是猫还是狗）

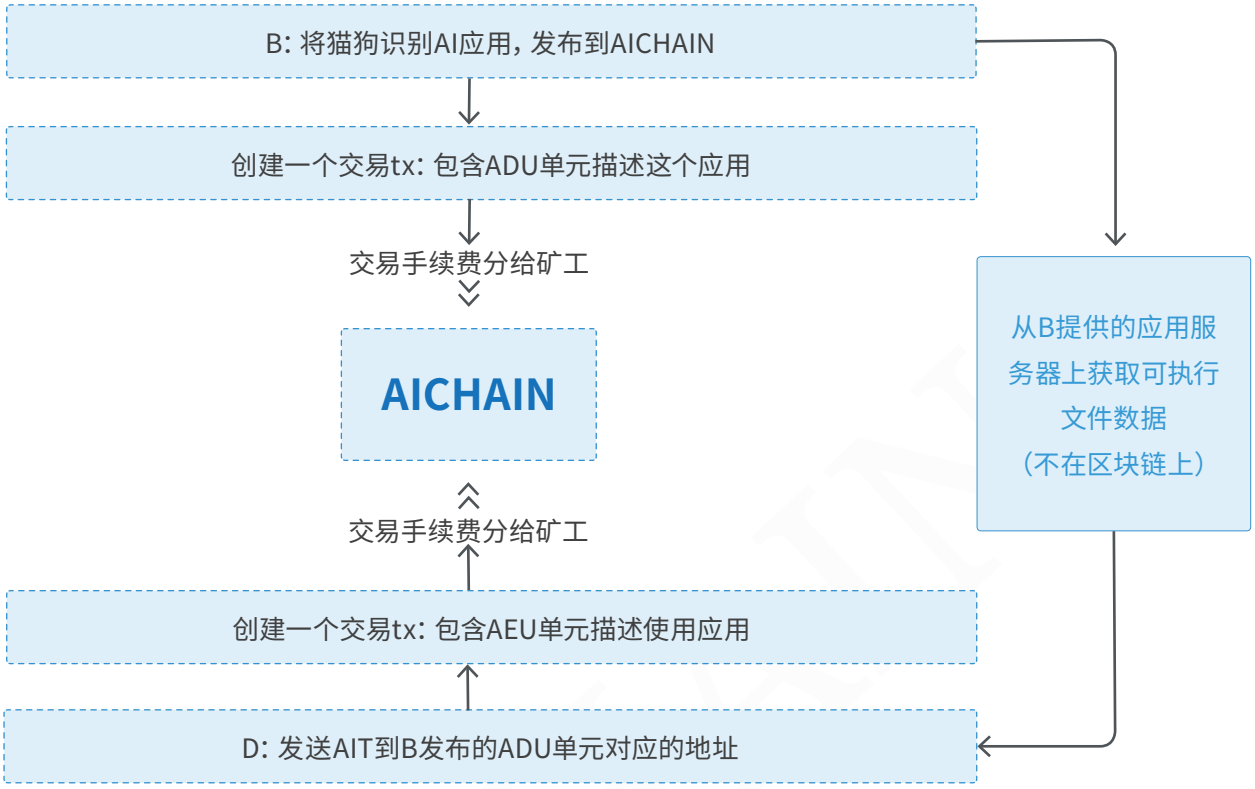
B: 图片猫狗识别AI应用的开发者，一家公司。

5.6.2 用户使用应用

用户在自己的电脑上使用猫狗图片识别的AI应用

B: 是应用提供方

D: 用户



B: 将开发好的猫狗识别AI应用信息部署到区块链上，提供AI应用可执行文件数据的下载。

D: 用户自己有显卡电脑，可以支持运行猫狗识别AI应用。

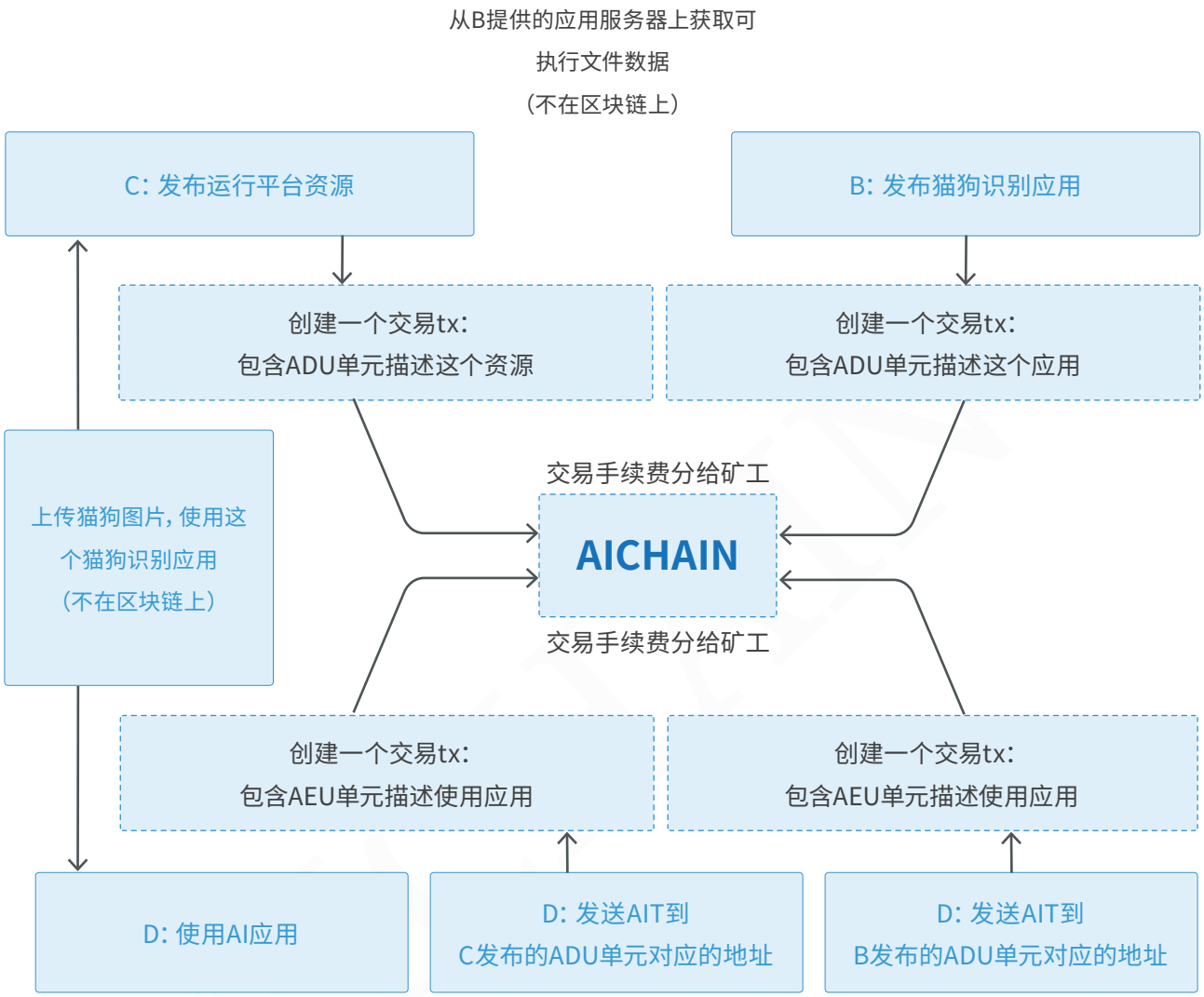
5.6.3 用户使用运行平台资源和应用

用户在第三方的运行平台上使用猫狗图片识别的AI应用

B: 是应用提供方

C: 是运行平台资源提供方

D: 用户



B: 将开发好的猫狗识别AI应用信息部署到区块链上，提供AI应用可执行文件数据的下载。

D: 用户没有合适的电脑来运行猫狗识别AI应用。需要借用C提供的平台。需要转账给C提供的运行资源ADU的地址。

C: 分配一个显卡服务器的AI应用运行平台资源给D。

D: 转账AIT给B提供的猫狗识别应用的ADU单元的地址。

D: D在C分配的AI应用运行平台上发起载入和运行B提供的猫狗识别AI应用。然后可以上传图片，完成分类。

参考文献

[1] http://www.coinwarz.com/cryptocurrency/

[2]https://github.com/leocalm/Lyra/blob/master/Lyra2/Lyra2ReferenceGuide.pdf

[3] https://bitcointalk.org/index.php?topic=586407.0

[4] http://phoenixcoin.org/archive/neoscrypt_v1.pdf

[5] https://en.bitcoin.it/wiki/Category:History

[6] https://github.com/bitcoinbook/bitcoinbook

[7] https://en.bitcoin.it/wiki/Wallet_import_format

[8] https://en.bitcoin.it/wiki/List_of_address_prefixes

[9] https://en.bitcoin.it/wiki/Transaction

[10] https://github.com/ethereum/wiki/wiki/White-Paper

[11] https://en.bitcoin.it/wiki/Pay_to_script_hash

[12] https://en.bitcoin.it/wiki/Transaction#Pay-to-PubkeyHash

[13] https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki

[14] https://en.bitcoin.it/wiki/Multisignature

[15] https://docs.docker.com/engine/userguide/eng-image/baseimages/

[16]https://docs.docker.com/engine/userguide/eng-image/multistage-build/

[17] https://gist.github.com/ericjang/959c03168c0bdfac1ca3

[18]https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/docker/README.md

[19]https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/docker

[20] https://hub.docker.com/r/bvlc/caffe/

[21] https://github.com/BVLC/caffe/tree/master/docker

[22]http://tleyden.github.io/blog/2014/10/25/running-caffe-on-aws-gpu-instance-via-docker/

[23] https://en.wikipedia.org/wiki/Elliptic-curve_cryptography